# 基础

## 基本数据类型

八种基本数据类型：int、short、float、double、long、boolean、byte、char。

封装类分别是：Integer、Short、Float、Double、Long、Boolean、Byte、Character。



## Switch能否用string做参数

jdk7之前 switch 只能支持 byte、short、char、int 这几个基本数据类型和其对应的封装类型。switch后面的括号里面只能放int类型的值，但由于byte，short，char类型，它们会?自动?转换为int类型（精精度小的向大的转化），所以它们也支持。jdk1.7后 整形，枚举类型，字符串都可以。

**为什么jdk1.7后又可以用string类型作为switch参数呢？**

其实，jdk1.7并没有新的指令来处理switch string，而是通过调用switch中string.hashCode,将string转换为int从而进行判断。

## Object有哪些公用方法

Object是所有类的父类，任何类都默认继承Object

clone 保护方法，实现对象的浅复制，只有实现了Cloneable接口才可以调用该方法，否则抛出CloneNotSupportedException异常。

equals 在Object中与==是一样的，子类一般需要重写该方法。

hashCode 该方法用于哈希查找，重写了equals方法一般都要重写hashCode方法。这个方法在一些具有哈希功能的Collection中用到。

getClass final方法，获得运行时类型

wait 使当前线程等待该对象的锁，当前线程必须是该对象的拥有者，也就是具有该对象的锁。 wait() 方法一直等待，直到获得锁或者被中断。 wait(long timeout) 设定一个超时间隔，如果在规定时间内没有获得锁就返回。

调用该方法后当前线程进入睡眠状态，直到以下事件发生

1、其他线程调用了该对象的notify方法。 2、其他线程调用了该对象的notifyAll方法。 3、其他线程调用了interrupt中断该线程。 4、时间间隔到了。 5、此时该线程就可以被调度了，如果是被中断的话就抛出一个InterruptedException异常。

notify 唤醒在该对象上等待的某个线程。

notifyAll 唤醒在该对象上等待的所有线程。

toString 转换成字符串，一般子类都有重写，否则打印句柄。

**对象的引用分为四种级别**，从而使程序能更加灵活的控制对象的生命周期。这四种级别由高到低依次为：强引用、软引用、弱引用和虚引用。

1、强引用:最普遍的一种引用方式，如String s = "abc"，变量s就是字符串“abc”的强引用，只要强引用存在，则垃圾回收器就不会回收这个对象。

2、软引用（SoftReference）:用于描述还有用但非必须的对象，如果内存足够，不回收，如果内存不足，则回收。一般用于实现内存敏感的高速缓存，软引用可以和引用队列ReferenceQueue联合使用，如果软引用的对象被垃圾回收，JVM就会把这个软引用加入到与之关联的引用队列中。

3、弱引用（WeakReference）:弱引用和软引用大致相同，弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

4、虚引用（PhantomReference）:就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。 虚引用主要用来跟踪对象被垃圾回收器回收的活动。

虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

## 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。

## String为什么要设计成 immutable的

不可变对象可以提高String Pool的效率和安全性。

1）如果你知道一个对象是不可变的，那么需要拷贝这个对象的内容时，就不用复制它的本身而只是复制它的地址，复制地址（通常一个指针的大小）需要很小的内存效率也很高。对于同时引用这个“ABC”的其他变量也不会造成影响。

2）不可变对象对于多线程是安全的，因为在多线程同时进行的情况下，一个可变对象的值很可能被其他进程改变，这样会造成不可预期的结果，而使用不可变对象就可以避免这种情况。

## String,StringBuffer,StringBuild比较

String和StringBuffer都是final类，他们生成的对象在堆中都是不可变的，在他们内部都是靠属性char数组实现的，那为什么StringBuffer可以在对象中追加字符串呢？

因为String中的char数组是finall的，也就是常量，是不可改变 的，而StringBuffer继承自抽象类AbstractStringBuilder，他的内部实现靠他的父类完 成，AbstractStringBuilder内的char数组是变量，可以用append追加

## 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

**try?catch?finally，try里有return，finally还执行么**

肯定会执行。finally{}块的代码。 只有在try{}块中包含遇到System.exit(0)。 之类的导致Java虚拟机直接退出的语句才会不执行。

## Excption与Error包结构

**OOM你遇到过哪些情况，SOF你遇到过哪些情况**

Throwable是 Java 语言中所有错误或异常的超类。 Throwable包含两个子类: Error 和 Exception 。它们通常用于指示发生了异常情况。 Throwable包含了其线程创建时线程执行堆栈的快照，它提供了printStackTrace()等接口用于获取堆栈跟踪数据等信息。

Java将可抛出(Throwable)的结构分为三种类型：

被检查的异常(Checked Exception)。 运行时异常(RuntimeException)。 错误(Error)。

运行时异常RuntimeException

定义 : RuntimeException及其子类都被称为运行时异常。 特点 : Java编译器不会检查它 也就是说，当程序中可能出现这类异常时，倘若既"没有通过throws声明抛出它"，也"没有用try-catch语句捕获它"，还是会编译通过。

例如，除数为零时产生的ArithmeticException异常，数组越界时产生的IndexOutOfBoundsException异常，fail-fail机制产生的ConcurrentModificationException异常等，都属于运行时异常。

堆内存溢出 OutOfMemoryError（OOM）

除了程序计数器外，虚拟机内存的其他几个运行时区域都有发生OutOfMemoryError(OOM)异常的可能。

Java Heap 溢出。 一般的异常信息：java.lang.OutOfMemoryError:Java heap spacess。 java堆用于存储对象实例，我们只要不断的创建对象，并且保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象，就会在对象数量达到最大堆容量限制后产生内存溢出异常。

堆栈溢出 StackOverflow （SOF）

StackOverflowError 的定义： 当应用程序递归太深而发生堆栈溢出时，抛出该错误。 因为栈一般默认为1-2m，一旦出现死循环或者是大量的递归调用，在不断的压栈过程中，造成栈容量超过1m而导致溢出。

栈溢出的原因：

递归调用。 大量循环或死循环。 全局变量是否过多。 数组、List、map数据过大。

**error和exception有什么区别?**

error 表示恢复不是不可能但很困难的情况下的一种严重问题。比如说内存溢出。不可能指望程序能处理这样的情况。 exception 表示一种设计或实现问题。也就是说，它表示如果程序运行正常，从不会发生的情况。

调用sleep不会释放对象锁。 wait是Object类的方法，对此对象调用wait方法导致本线程放弃对象锁

方法中的局部变量使用final修饰后，放在堆中，而不是栈中。

## Override和Overload的含义去区别

重载 Overload方法名相同，参数列表不同(个数、顺序、类型不同任意其一)与返回类型无关。 重写 Override 覆盖。 将父类的方法覆盖。 重写方法重写：方法名相同，访问修饰符只能大于被重写的方法访问修饰符，方法签名个数，顺序个数类型相同。

Override（重写）

• 方法名、参数、返回值相同。

• 子类方法不能缩小父类方法的访问权限。

• 子类方法不能抛出比父类方法更多的异常(但子类方法可以不抛出异常)。

• 存在于父类和子类之间。

• 方法被定义为final不能被重写。

Overload（重载）

• 参数类型、个数、顺序至少有一个不相同。

• 不能重载只有返回值不同的方法名。

• 存在于父类和子类、同类中。

而重载的规则

1、必须具有不同的参数列表。 2、可以有不同的返回类型，只要参数列表不同就可以了。 3、可以有不同的访问修饰符。 4、可以抛出不同的异常。

重写方法的规则

1、参数列表必须完全与被重写的方法相同，否则不能称其为重写而是重载。 2、返回的类型必须一直与被重写的方法的返回类型相同，否则不能称其为重写而是重载。 3、访问修饰符的限制一定要大于被重写方法的访问修饰符（public>protected>default>private）。 4、重写方法一定不能抛出新的检查异常或者比被重写方法申明更加宽泛的检查型异常。

例如： 父类的一个方法申明了一个检查异常IOException，在重写这个方法是就不能抛出Exception,只能抛出IOException的子类异常，可以抛出非检查异常。

Overloaded的方法是可以改变返回值的类型

## Interface与abstract类的区别

Interface 只能有成员常量，只能是方法的声明。 Abstract class可以有成员变量，可以声明普通方法和抽象方法。

interface是接口，所有的方法都是抽象方法，成员变量是默认的public static final 类型。接口不能实例化自己。

abstract class是抽象类，至少包含一个抽象方法的累叫抽象类，抽象类不能被自身实例化，并用abstract关键字来修饰。

## Static class 与non static class的区别

static class（内部静态类）

1、用static修饰的是内部类，此时这个内部类变为静态内部类；对测试有用。 2、内部静态类不需要有指向外部类的引用。 3、静态类只能访问外部类的静态成员，不能访问外部类的非静态成员。

non static class（非静态内部类）

1、非静态内部类需要持有对外部类的引用。 2、非静态内部类能够访问外部类的静态和非静态成员。 3、一个非静态内部类不能脱离外部类实体被创建。 4、一个非静态内部类可以访问外部类的数据和方法。

## Foreach（增强for循环）与正常for循环效率对比

ArrayList：ArrayList是采用数组的形式保存对象的，这种方式将对象放在连续的内存块中，所以插入和删除时比较麻烦，查询比较方便。

LinkList：LinkList是将对象放在独立的空间中，而且每个空间中还保存下一个空间的索引，也就是数据结构中的链表结构，插入和删除比较方便，但是查找很麻烦，要从第一个开始遍历。

结论：

需要循环数组结构的数据时，建议使用普通for循环，因为for循环采用下标访问，对于数组结构的数据来说，采用下标访问比较好。

需要循环链表结构的数据时，一定不要使用普通for循环，这种做法很糟糕，数据量大的时候有可能会导致系统崩溃。

## java反射的作用于原理

什么是Java的反射呢？

Java 反射是可以让我们在运行时，通过一个类的Class对象来获取它获取类的方法、属性、父类、接口等类的内部信息的机制。

这种动态获取信息以及动态调用对象的方法的功能称为JAVA的反射。

**反射的作用？**

反射就是：在任意一个方法里：

1.如果我知道一个类的名称/或者它的一个实例对象， 我就能把这个类的所有方法和变量的信息找出来(方法名，变量名，方法，修饰符，类型，方法参数等等所有信息)

2.如果我还明确知道这个类里某个变量的名称，我还能得到这个变量当前的值。

3.当然，如果我明确知道这个类里的某个方法名+参数个数类型，我还能通过传递参数来运行那个类里的那个方法。

**反射机制主要提供了以下功能：**

在运行时判断任意一个对象所属的类。

在运行时构造任意一个类的对象。

在运行时判断任意一个类所具有的成员变量和方法。

在运行时调用任意一个对象的方法。

生成动态代理。

**反射的原理？**

JAVA语言编译之后会生成一个.class文件，反射就是通过字节码文件找到某一个类、类中的方法以及属性等。

**反射的实现API有哪些？**

反射的实现主要借助以下四个类：

Class：类的对象

Constructor：类的构造方法

Field：类中的属性对象

Method：类中的方法对象

java的反射(reflection)机制

开放性和原因连接(causally-connected)是反射系统的两大基本要素

开放性：能够自描述和自控制

原因连接：指对反射系统自描述的改变能够立即反映到系统底层的实际状态和行为上的情况，反之亦然。

Java的反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制

Java的反射机制主要提供了以下功能

在运行时判断任意一个对象所属的类

在运行时判断任意一个类所具有的成员变量和方法

在运行时构造任意一个类的对象

在运行是调用任意一个对象的方法和变量

生成动态代理

子类覆盖父类的方法时，只能比父类抛出更少的异常

只有记住抽象类与普通类的唯一区别就是不能创建实例对象和允许有abstract方法。

## Java1.7与1.8,1.9,10 新特性

1.7

switch中可以使用字串了

运用List tempList = new ArrayList<>(); 即泛型实例化类型自动推断

语法上支持集合，而不一定是数组

新增一些取环境信息的工具方法

Boolean类型反转，空指针安全,参与位运算

两个char间的equals

安全的加减乘除

map集合支持并发请求，且可以写成 Map map = {name:"xxx",age:18};

1.8

允许在接口中有默认方法实现

Lambda表达式

函数式接口

方法和构造函数引用

Lambda的范围

内置函数式接口

Streams

Parallel Streams

Map

时间日期API

Annotations

1.9

Jigsaw 项目;模块化源码

简化进程API

轻量级 JSON API

钱和货币的API

改善锁争用机制

代码分段缓存

智能Java编译, 第二阶段

HTTP 2.0客户端

Kulla计划: Java的REPL实现

10

本地变量类型推断

统一JDK仓库

垃圾回收器接口

G1的并行Full GC

应用程序类数据共享

ThreadLocal握手机制

## 类的抽象，继承

abstract的method是否可同时是static,是否可同时是native，是否可同时是synchronized

答： 都不能

接口是否可继承接口? 抽象类是否可实现(implements)接口? 抽象类是否可继承实体类(concrete class)

答： 接口可以继承接口。抽象类可以实现(implements)接口，抽象类是否可继承实体类，但前提是实体类必须有明确的构造函数

抽象类可以有构造方法，而且实例化子类时也可以初始化其构造方法。但是抽象类本身不能实例化。

构造器Constructor是否可被override

答： 构造器Constructor不能被继承，因此不能重写Overriding，但可以被重载Overloading

float型float f=3.4是否正确?

答: 不正确。精度不准确,应该用强制类型转换，如下所示：float f=(float)3.4

## ？Arrays.sort函数基础与非基础类型的排序方式

Java系统提供的Arrays.sort函数。对于基础类型，底层使用快速排序。对于非基础类型，底层使用归并排序。请问是为什么？

## 位运算符

**按位运算符**

1. 按位与运算（&）：二元运算符。当被运算的两个值都为1时，运算结果为1；否则为0。

2. 按位或运算（|）：二元运算符。当被运算的两个值都为0时，运算结果为0；否则为1。

3. 按位异或运算（^）：二元运算符。当被运算的两个值中任意一个为1，另一个为0 时，运算结果为1；否则为 0。

4. 按位非运算（~）：一元运算符。当被运算的值为 1时，运算结果为0；当被运算的值为0 时，运算结果为1。

**移位运算符**

1. 左移位（<<）：将操作符左侧的操作数向左移动操作符右侧指定的位数。移动的规则是在二进制的低位补0。

2. 有符号右移位（>>）：将操作符左侧的操作数向右移动操作符右侧指定的位数。移动的规则是，如果被操作数的符号为正，则在二进制的高位补 0；如果被操作数的符号为负，则在二进制的高位补1。

3. 无符号右移位（>>>）：将操作符左侧的操作数向右移动操作符右侧指定的位数。移动的规则是，无论被操作数的符号是正是负，都在二进制位的高位补 0。

## 多态实现原理

JAVA使用了后期绑定的概念。当向对象发送消息时，在编译阶段，编译器只保证被调用方法的存在，并对调用参数和返回类型进行检查，但是并不知道将被执行的确切代码，被调用的代码直到运行时才能确定。JAVA中除了static方法、final方法（private方法属于）之外，其他的方法都是后期绑定。后期绑定会涉及到JVM管理下的一个重要的数据结构——方法表，方法表以数组的形式记录当前类及其所有父类的可见方法字节码在内存中的直接地址。

动态绑定具体的调用过程为：

1.首先会找到被调用方法所属类的全限定名

2.在此类的方法表中寻找被调用方法，如果找到，会将方法表中此方法的索引项记录到常量池中（这个过程叫常量池解析），如果没有，编译失败。

3.根据具体实例化的对象找到方法区中此对象的方法表，再找到方法表中的被调用方法，最后通过直接地址找到字节码所在的内存空间。

最后说明，域和静态方法都是不具有多态性的，任何的域访问操作都将由编译器解析，因此不是多态的。静态方法是跟类，而并非单个对象相关联的。对动态绑定还有不明白的请看资料链接，个人感觉分析的很到位

## Java中常见的异常有哪些？

java.lang.nullpointerexception

java.lang.classnotfoundexception

java.lang.arrayindexoutofboundsexception 数组下标越界

java.lang.illegalargumentexception 方法的参数错误

算术异常类：ArithmeticExecption

类型强制转换异常：ClassCastException

文件未找到异常：FileNotFoundException

方法未找到异常：NoSuchMethodException

字符串转换为数字异常：NumberFormatException

高并发是请求，指的是多个客户端同一时刻向服务端发送请求， 它是一种现象。

多线程是处理，指的是同一时刻多个执行者处理同一类的任务， 它有具体的实现。

并行 多核cpu的情况， 多个任务执行者并行处理任务

并发 单个cpu的情况下，cpu间断性的执行多个任务。

## http1.0和http1.1的区别

http1.0协议使用非持久连接，即在非持久连接下，一个tcp连接只传输一个web对象,http1.1默认使用持久连接，一个连接可以传输多个对象

http1.1对请求和响应增加了请求头和响应头

http1.1支持断点续传，http1.0不支持

http1.1通过请求头部使得在一台web服务器可以同一个IP，端口，使用不同的主机名创建多个web虚拟站点

http1.1有身份认证机制，http提供特殊的状态码和头部来帮助web站点执行身份认证

## 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

## StringBuilder和“+”

String s = "abc" + "ddd";通过编译后的源码发现，+在源码中会编译成使用StringBuilder的append完成拼接，因此在Java中无论使用何种方式进行字符串连接，实际上都使用的是StringBuilder。但是最好不要将+用在for循环中，因为会创建多个StringBuilder。

# 集合

## 常用的集合

**常用的集合主要在java.util包下面：**

主要接口有List,Set,Map,Queue,Deque,SortedMap,SortedSet

List:

线程不安全：ArrayList, LinkedList, AttributeList, RoleList

线程安全：CopyOnWriteArraylist ,Stack, Vector

Set:

线程不安全：EnumSet, HashSet, LinkedHashSet, TreeSet

线程安全：ConcurrentSkipListSet, CopyOnWriteArraySet

Map:

线程不安全：EnumMap, HashMap, IdentityHashMap, LinkedHashMap, Properties, TreeMap

线程安全：ConcurrentHashMap, ConcurrentSkipListMap, Hashtable

SortedMap继承于map:

ConcurrentSkipListMap, TreeMap

SortedSet继承于set:

ConcurrentSkipListSet, TreeSet

queue:

1.ArrayDeque, （数组双端队列）

2.PriorityQueue, （优先级队列）

3.ConcurrentLinkedQueue, （基于链表的并发队列）

4.DelayQueue, （延期阻塞队列）（阻塞队列实现了BlockingQueue接口）

5.ArrayBlockingQueue, （基于数组的并发阻塞队列）

6.LinkedBlockingQueue, （基于链表的FIFO阻塞队列）

7.LinkedBlockingDeque, （基于链表的FIFO双端阻塞队列）

8.PriorityBlockingQueue, （带优先级的无界阻塞队列）

9.SynchronousQueue （并发同步阻塞队列）

Deque:

ArrayDeque, LinkedBlockingDeque, LinkedList

**fail-fast与fail-safe有什么区别？**

快速失败（fail—fast）: 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（增加、删除、修改），则会抛出Concurrent Modification Exception。场景：java.util包下的集合类都是快速失败的，不能在多线程下发生并发修改（迭代过程中被修改）。

安全失败（fail—safe）: 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。

## 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

**HashMap的hashcode的作用**

hashCode的存在主要是用于查找的快捷性，如Hashtable，HashMap等，hashCode是用来在散列存储结构中确定对象的存储地址的。一般的地方不需要重载hashCode，只有当类需要放在HashTable、HashMap、HashSet等等hash结构的集合时才会重载hashCode，那么为什么要重载hashCode呢？

**为什么要重载equal方法？**

因为Object的equal方法默认是两个对象的引用的比较，意思就是指向同一内存,地址则相等，否则不相等；如果你现在需要利用对象里面的值来判断是否相等，则重载equal方法。

为什么equals()相等，hashCode就一定要相等，而hashCode相等，却不要求equals相等?

1、因为是按照hashCode来访问小内存块，所以hashCode必须相等。

2、HashMap获取一个对象是比较key的hashCode相等和equal为true。之所以hashCode相等，却可以equal不等，就比如ObjectA和ObjectB他们都有属性name，那么hashCode都以name计算，所以hashCode一样，但是两个对象属于不同类型，所以equal为false。

**为什么需要hashCode?**

1、通过hashCode可以很快的查到小内存块。 2、通过hashCode比较比equal方法快，当get时先比较hashCode，如果hashCode不同，直接返回false。

**集合**

Collection 是对象集合， Collection 有两个子接口 List 和 Set

List 可以通过下标 (1,2..) 来取得值，值可以重复。 Set 只能通过游标来取值，并且值是不能重复的。

ArrayList ， Vector ， LinkedList 是 List 的实现类

ArrayList 是线程不安全的， Vector 是线程安全的，这两个类底层都是由数组实现的。

LinkedList 是线程不安全的，底层是由链表实现的。

Map 是键值对集合

HashTable 和 HashMap 是 Map 的实现类。

HashTable 是线程安全的，不能存储 null 值。

HashMap 不是线程安全的，可以存储 null 值。

Stack类：继承自Vector，实现一个后进先出的栈。提供了几个基本方法，push、pop、peak、empty、search等。

Queue接口：提供了几个基本方法，offer、poll、peek等。已知实现类有LinkedList、PriorityQueue等。

**HashMap和HashTable的区别**

1、hashMap去掉了HashTable?的contains方法，但是加上了containsValue()和containsKey()方法

2、Hashtable不允许 null 值(key 和 value 都不可以)，HashMap允许 null 值(key和value都可以)。

3、两者的遍历方式大同小异，Hashtable仅仅比HashMap多一个elements方法。

4、HashTable使用Enumeration，HashMap使用Iterator

**JDK8中的HashMap**

JDK8中采用的是位桶+链表/红黑树（有关红黑树请查看红黑树）的方式，也是非线程安全的。当某个位桶的链表的长度达到某个阀值的时候，这个链表就将转换成红黑树。

JDK6,7中的ConcurrentHashmap主要使用Segment来实现减小锁粒度，把HashMap分割成若干个Segment，在put的时候需要锁住Segment，get时候不加锁，使用volatile来保证可见性，当要统计全局时（比如size），首先会尝试多次计算modcount来确定，这几次尝试中，是否有其他线程进行了修改操作，如果没有，则直接返回size。如果有，则需要依次锁住所有的Segment来计算。

jdk7中ConcurrentHashmap中，当长度过长碰撞会很频繁，链表的增改删查操作都会消耗很长的时间，影响性能。

jdk8 中完全重写了concurrentHashmap,代码量从原来的1000多行变成了 6000多 行，实现上也和原来的分段式存储有很大的区别。

JDK8中采用的是位桶+链表/红黑树（有关红黑树请查看红黑树）的方式，也是非线程安全的。当某个位桶的链表的长度达到某个阀值的时候，这个链表就将转换成红黑树。

JDK8中，当同一个hash值的节点数不小于8时，将不再以单链表的形式存储了，会被调整成一颗红黑树（上图中null节点没画）。这就是JDK7与JDK8中HashMap实现的最大区别。

**ConcurrentHashMap能完全替代HashTable吗？**

HashTable虽然性能上不如ConcurrentHashMap，但并不能完全被取代，两者的迭代器的一致性不同的，HashTable的迭代器是强一致性的，而ConcurrentHashMap是弱一致的。 ConcurrentHashMap的get，clear，iterator 都是弱一致性的。 Doug Lea 也将这个判断留给用户自己决定是否使用ConcurrentHashMap。

**那么什么是强一致性和弱一致性呢？**

get方法是弱一致的，是什么含义？可能你期望往ConcurrentHashMap底层数据结构中加入一个元素后，立马能对get可见，但ConcurrentHashMap并不能如你所愿。换句话说，put操作将一个元素加入到底层数据结构后，get可能在某段时间内还看不到这个元素，若不考虑内存模型，单从代码逻辑上来看，却是应该可以看得到的。

下面将结合代码和java内存模型相关内容来分析下put/get方法。put方法我们只需关注Segment#put，get方法只需关注Segment#get，在继续之前，先要说明一下Segment里有两个volatile变量：count和table；HashEntry里有一个volatile变量：value。

总结:ConcurrentHashMap的弱一致性主要是为了提升效率，是一致性与效率之间的一种权衡。要成为强一致性，就得到处使用锁，甚至是全局锁，这就与Hashtable和同步的HashMap一样了。

如何线程安全的使用 HashMap。这个无非就是以下三种方式：Hashtable ConcurrentHashMap Synchronized Map

**SynchronizedMap**：synchronizedMap() 方法后会返回一个 SynchronizedMap 类的对象，而在 SynchronizedMap 类中使用了 synchronized 同步关键字来保证对 Map 的操作是线程安全的。

**Collection包结构，与Collections的区别**

Collection 是集合类的上级接口，子接口主要有Set、List 、Map。

Collecions 是针对集合类的一个帮助类， 提供了操作集合的工具方法，一系列静态方法实现对各种集合的搜索、排序线性、线程安全化等操作。

## HashMap

hashmap扩容过程，1.7 1.8的区别

1.7

Index的计算方式：return h & (length-1);

扩容方式：循环遍历所有的node节点，再一个个获取位置存入新的数组，扩容时也是扩为原来的两倍。

1.8

Index的计算方式 : h & (n-1)

扩容时方式：遍历数组，将每个桶先分成两个桶再插入到新的数组中

1.8新增：h = (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);将高位参与运算是为了在table长度比较小时，减少碰撞。



如果还是产生了频繁的碰撞，会发生什么问题呢？作者注释说，他们使用树来处理频繁的碰撞

Arrays.sort实现原理和Collection实现原理

# Java安全机制

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

**对称加密**

采用单钥密码系统的加密方法，同一个密钥可以同时用作信息的加密和解密，这种加密方法称为对称加密，也称为单密钥加密。而因为加密和解密都使用同一个密钥，如何把密钥安全地传递到解密者手上就成了必须要解决的问题。

**非对称加密**

非对称加密算法需要两个密钥来进行加密和解密，分别是公钥和私钥。需要注意的一点，这个公钥和私钥必须是一对的，如果用公钥对数据进行加密，那么只有使用对应的私钥才能解密，反之亦然。由于加密和解密使用的是两个不同的密钥，因此，这种算法叫做非对称加密算法。

**HTTPS一般使用的加密与HASH算法**

非对称加密算法：RSA,DSA/DSS,DH

对称加密算法：AES,RC4,3DES

HSAH算法：MD5,SHA1,SHA256

AES，全称为“Advanced Encryption Standard”，中文名“高级加密标准”，在密码学中又称 Rijndael 加密法，是美国联邦政府采用的一种区块加密标准。AES 加密算法作为新一代的数据加密标准汇聚了强安全性、高性能、高效率、易用和灵活等优点。AES 设计有三个密钥长度：128，192，256 位。相对而言，AES 的 128 密钥比 DES 的 56 密钥强了 1021 倍。

**数字签名证书**非对称加密已经灰常安全了，但是还有一个破绽：  
服务器A公布了自己的公钥，我的电脑是用服务器A的公钥加密数据后再发给服务器A的；这时候服务器B侵入了我的电脑，把我用来加密的公钥换成了它的公钥，于是我发出去的数据就会被服务器B的私钥破解了。肿么防止公钥被篡改呢？  
对，我们想到了前面的消息摘要，服务器A把公钥丢给我的时候，同时去CA申请一份数字证书，其实主要就是公钥的消息摘要，有了这份证书，当我再用公钥加密的时候，我就可以先验证一下当前的公钥是否确定是服务器A发送给我的

# 设计者模式

## 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。

# 数据结构与算法

## 常用数据结构？

一、线性表

1.数组实现：使用一块连续的内存空间保存数据，长度固定，但插入和删除数据需要移动大量元素。

2.链表：链表中数据不是在内存中连续存储的，而是通过指向下一个元素的指针来联系的。长度不固定，但是插入和删除元素十分方便，只需更改指针指向即可。

二、栈与队列

栈：是一种后进先出的结构，可以使用数组或者是链表来实现它，对于堆栈中的数据的所有操作都是在栈的顶部完成的。

队列：实现了先入先出的语义 。队列也可以使用数组和链表来实现，队列只允许在队尾添加数据，在队头删除数据。

三、树与二叉树？

四、图

## 查找算法

有序数据查找

一、顺序查找算法

二、二分查找又称折半查找

三、插值查找

如果在1~50000的数中查找数字5，按照正常的思维应该是从下标为0的地方开始查找，这就是经验，我们此时不会再从中点（25000）进行查找

在二分查找中查找点计算如下：   
    mid=(low+high)/2, 即mid=low+1/2\*(high-low);   
  通过类比，我们可以将查找的点改进为如下：   
    mid=low+(key-a[low])/(a[high]-a[low])\*(high-low)，

插值查找的核心思想就是基于二分查找算法，将查找点的选择改进为自适应选择，可以提高查找效率。

四、斐波那契查找

斐波那契数列：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…….（从第三个数开始，后边每一个数都是前两个数的和）。然后我们会发现，随着斐波那契数列的递增，前后两个数的比值会越来越接近0.618，利用这个特性，我们就可以将黄金比例运用到查找技术中。

斐波那契查找与折半查找很相似，他是根据斐波那契序列的特点对有序表进行分割的。他要求开始表中记录的个数为某个斐波那契数小1，即n=F(k)-1；开始将key值（要查找的数据）与第F(k-1)位置的记录进行比较(即mid=low+F(k-1)-1)，比较结果也分为三种：   
  （1）相等，mid位置的元素即为所求；   
  （2）大于，low=mid+1，k-=2。说明：low=mid+1说明待查找的元素在[mid+1,high]范围内，k-=2 说明范围[mid+1,high]内的元素个数为n-(F(k-1))= Fk-1-F(k-1)=Fk-F(k-1)-1=F(k-2)-1个，所以可以递归的应用斐波那契查找。   
  （3）小于，high=mid-1，k-=1。说明：low=mid+1说明待查找的元素在[low,mid-1]范围内，k-=1 说明范围[low,mid-1]内的元素个数为F(k-1)-1个，所以可以递归 的应用斐波那契查找。

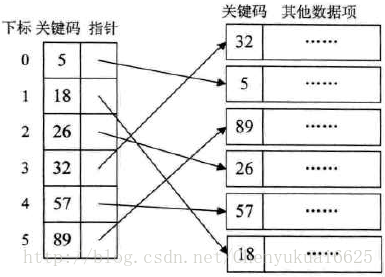
无序队列查找

一 索引查找

一个索引由若干个索引项构成，每个索引项至少应包含关键字和其对应的记录在存储器中的位置等信息。

索引按照结构可以分为线性索引、树形索引和多级索引。我们这里就只介绍线性索引技术。所谓线性索引就是将索引项集合组织为线性结构，也称为索引表。我们重点介绍三个线性索引：稠密索引、分块索引、和倒排索引。

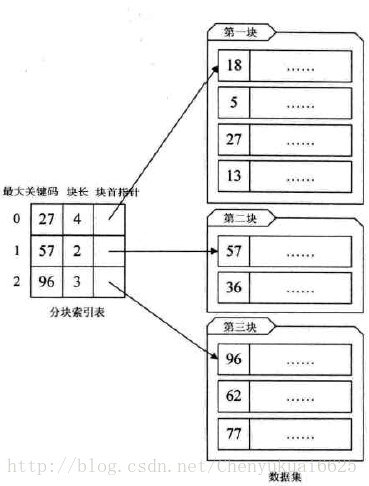
1.2稠密索引   
  稠密索引是指在线性索引中，将数据集中的每个记录对应一个索引项。左边的图像为索引序列，它是是按照关键码有序排列的。索引项有序。



1.3 分块索引

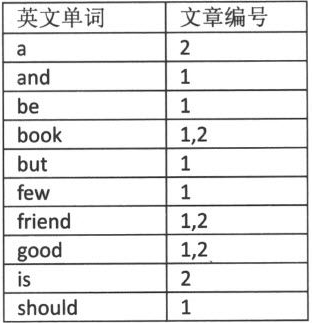
稠密索引因为索引项与数据集的记录个数相同，所以空间代价很大。为了减少索引的个数，我们可以对数据集进行分块，使其分块有序，然后再对每一块建立一个索引项，从而减少索引项的个数。

每个数据块保存了某个范围的数据，块内无序，但块间有序



1.4 倒排索引

看一个例子，假设有以下两篇文章：   
  (1) Books and friends should be few but good .   
  (2) A good book is a good friend.   
  假设我们忽略掉如“books”，“friends”中的复数”s”以及如“A”这样的大小写差异。我们可以整理出这样一张单词表，如下图所示，并将单词做了排序，也就是表格显示了每个不同的单词分别出现在哪篇文章中，比如“good”它在两篇文章中都有出现，而is只有在文章2中才有。

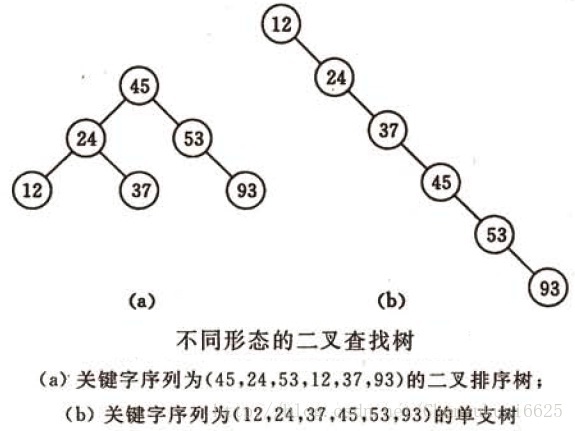


在这里这张单词表就是索引表，索引项的通用结构是次关键码和记录号表。 其中记录号表存储具有相同次字关键字的所有记录的记录号（可以指向记录的指针或者是该记录的主关键字）。因为这种查找方法是通过属性值来确定记录的位置，而不是通过记录来确定属性值，所以我们称其为倒排索引。

二、树表查找

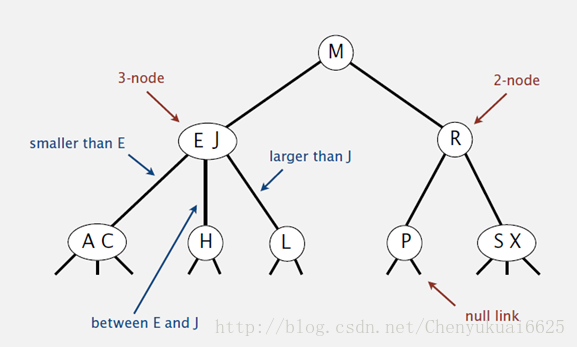
2.1 二叉树查找算法（最简单的树表查找算法）

有没有一种既可以使得插入和删除效率不错，又可以比较高效的实现查找的算法？这是有的，二叉树查找算法就可以实现这样的功能。



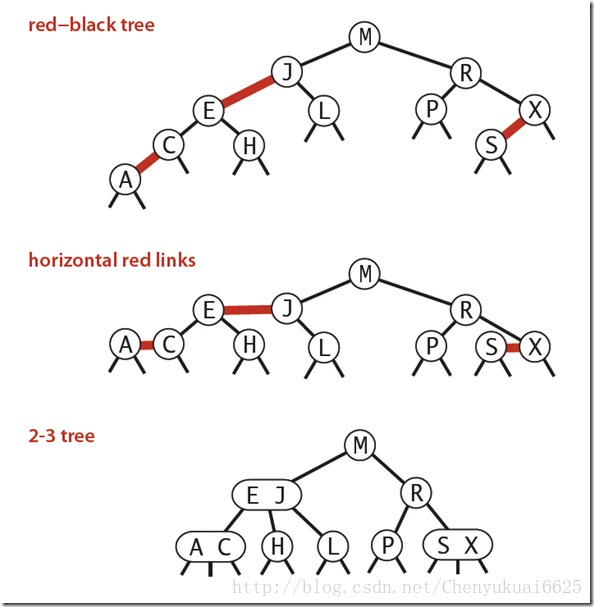
2.2 平衡查找树之2-3查找树

和二叉树不一样，2-3树中每个节点保存1个或者两个key值。对于普通的2节点(2-node)，它保存1个key和左右两个孩子（或没有孩子）。对应3节点(3-node)，保存两个Key和三个孩子（或没有孩子）



2.3 平衡查找树之红黑树

红黑树的思想就是对2-3查找树进行编码，尤其是对2-3查找树中的3-nodes节点添加额外的信息。红黑树中将节点之间的链接分为两种不同类型，红色链接，他用来链接两个2-nodes节点来表示一个3-nodes节点。黑色链接用来链接普通的2-3节点。特别的，使用红色链接的两个2-nodes来表示一个3-nodes节点，并且向左倾斜，即一个2-node是另一个2-node的左子节点。这种做法的好处是查找的时候不用做任何修改，和普通的二叉查找树相同。



2.4 B树和B+树

**B 树**可以看作是对2-3查找树的一种扩展，即他允许每个节点有M-1个子节点。

* 根节点至少有两个子节点
* 每个节点有M-1个key，并且以升序排列
* 位于M-1和M key的子节点的值位于M-1 和M key对应的Value之间
* 其它节点至少有M/2个子节点

下图是一个M=4 阶的B树:



**B+**树是对B树的一种变形树，它与B树的差异在于：

* 有k个子结点的结点必然有k个关键码；
* 非叶结点仅具有索引作用，跟记录有关的信息均存放在叶结点中。
* 树的所有叶结点构成一个有序链表，可以按照关键码排序的次序遍历全部记录。

如下图，是一个B+树:



## 排序算法

**排序算法大体可分为两种：**

一种是比较排序，时间复杂度O(nlogn) ~ O(n^2)，主要有：冒泡排序，选择排序，插入排序，归并排序，堆排序，快速排序等

另一种是非比较排序，时间复杂度可以达到O(n)，主要有：计数排序，基数排序，桶排序等。

排序算法稳定性的简单形式化定义为：**如果Ai = Aj，排序前Ai在Aj之前，排序后Ai还在Aj之前，则称这种排序算法是稳定的。**通俗地讲就是保证排序前后两个相等的数的相对顺序不变。



### 冒泡排序

冒泡排序算法的运作如下：

比较相邻的元素，如果前一个比后一个大，就把它们两个调换位置。

对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。

针对所有的元素重复以上的步骤，除了最后一个。

持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较

void Sort(int A[], int n)

{

for (int j = 0; j < n - 1; j++) // 每次最大元素就像气泡一样"浮"到数组的最后

{

for (int i = 0; i < n - 1 - j; i++) // 依次比较相邻的两个元素,使较大的那个向后移

{

if (A[i] > A[i + 1]) // 如果条件改成A[i] >= A[i + 1],则变为不稳定的排序算法

{

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

}

}

}

### 选择排序

选择排序也是一种简单直观的排序算法。它的工作原理很容易理解：初始时在序列中找到最小（大）元素，放到序列的起始位置作为已排序序列；然后，再从剩余未排序元素中继续寻找最小（大）元素，放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

void SelectionSort(int A[], int n)

{

for (int i = 0; i < n - 1; i++) // i为已排序序列的末尾

{

int min = i;

for (int j = i + 1; j < n; j++) // 未排序序列

{

if (A[j] < A[min]) // 找出未排序序列中的最小值

{

min = j;

}

}

if (min != i) // 放到已排序序列的末尾，该操作很有可能把稳定性打乱，所以选择排序是不稳定的排序算法

{

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

}

}

### 插入排序

步骤：

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

简单点说，就是将后面的元素去除跟前面的一个个比，看哪个位置合适自己

for (int i = 1; i < n; i++) // 类似抓扑克牌排序

{

int get = A[i]; // 右手抓到一张扑克牌

int j = i - 1; // 拿在左手上的牌总是排序好的

while (j >= 0 && A[j] > get) // 将抓到的牌与手牌从右向左进行比较

{

A[j + 1] = A[j]; // 如果该手牌比抓到的牌大，就将其右移

j--;

}

A[j + 1] = get; // 直到该手牌比抓到的牌小(或二者相等)，将抓到的牌插入到该手牌右边(相等元素的相对次序未变，所以插入排序是稳定的)

}

插入排序不适合对于数据量比较大的排序应用

### 二分插入排序

对于插入排序，如果比较操作的代价比交换操作大的话，可以采用**二分查找法**来减少比较操作的次数，我们称为**二分插入排序**

for (int i = 1; i < n; i++)

{

int get = A[i]; // 右手抓到一张扑克牌

int left = 0; // 拿在左手上的牌总是排序好的，所以可以用二分法

int right = i - 1; // 手牌左右边界进行初始化

while (left <= right) // 采用二分法定位新牌的位置

{

int mid = (left + right) / 2;

if (A[mid] > get)

right = mid - 1;

else

left = mid + 1;

}

for (int j = i - 1; j >= left; j--) // 将欲插入新牌位置右边的牌整体向右移动一个单位

{

A[j + 1] = A[j];

}

A[left] = get; // 将抓到的牌插入手牌

}

当n较大时，二分插入排序的比较次数比直接插入排序的最差情况好得多，但比直接插入排序的最好情况要差，所当以元素初始序列已经接近升序时，直接插入排序比二分插入排序比较次数少。二分插入排序元素移动次数与直接插入排序相同，依赖于元素初始序列。

### 希尔排序

希尔排序，也叫**递减增量排序**，是插入排序的一种更高效的改进版本。希尔排序是**不稳定**的排序算法。

public void ShellSort() {

int [] A = { 5, 2, 9, 4, 7, 6, 1, 3, 8 };// 从小到大希尔排序

int h = 0;

int n = A.length;

while (h <= n) // 生成初始增量

{

h = 3 \* h + 1;

}

while (h >= 1)

{

System.out.println("h="+h);

for (int i = h; i < n; i++)

{

int j = i - h;

int get = A[i];

while (j >= 0 && A[j] > get)

{

A[j + h] = A[j];

j = j - h;

System.out.print("--");

printInt(A);

}

A[j + h] = get;

System.out.print("-=");

printInt(A);

}

System.out.println("=======");

h = (h - 1) / 3; // 递减增量

}

}

希尔排序通过将比较的全部元素分为几个区域来提升插入排序的性能。这样可以让一个元素可以一次性地朝最终位置前进一大步。然后算法再取越来越小的步长进行排序，算法的最后一步就是普通的插入排序，但是到了这步，需排序的数据几乎是已排好的了（此时插入排序较快）。

### 快速排序

快速排序使用分治策略(Divide and Conquer)来把一个序列分为两个子序列。步骤为：

1.从序列中挑出一个元素，作为"基准"(pivot).

2.把所有比基准值小的元素放在基准前面，所有比基准值大的元素放在基准的后面（相同的数可以到任一边），这个称为分区(partition)操作。

3.对每个分区递归地进行步骤1~2，递归的结束条件是序列的大小是0或1，这时整体已经被排好序了。

public void quickSortMain() {

int A[] = { 5, 2, 9, 4, 7, 6, 1, 3, 8 }; // 从小到大快速排序

int left = 0;

int n = A.length;

QuickSort(A, 0, n - 1);

}

public void QuickSort(int A[], int left, int right)

{

if (left >= right)

return;

int pivot\_index = Partition(A, left, right); // 基准的索引

System.out.println("pivot\_index="+pivot\_index);

QuickSort(A, left, pivot\_index - 1);

QuickSort(A, pivot\_index + 1, right);

}

public int Partition(int A[], int left, int right) // 划分函数

{

System.out.println("left="+left+" right="+right);

int pivot = A[right]; // 这里每次都选择最后一个元素作为基准

int tail = left - 1; // tail为小于基准的子数组最后一个元素的索引

for (int i = left; i < right; i++) // 遍历基准以外的其他元素

{

if (A[i] <= pivot) // 把小于等于基准的元素放到前一个子数组末尾

{

Swap(A, ++tail, i);

System.out.print("--");

printInt(A);

}

}

Swap(A, tail + 1, right); // 最后把基准放到前一个子数组的后边，剩下的子数组既是大于基准的子数组

System.out.print("-=");

printInt(A); // 该操作很有可能把后面元素的稳定性打乱，所以快速排序是不稳定的排序算法

return tail + 1; // 返回基准的索引

}

public void Swap(int A[], int i, int j)

{

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

**快速排序会有数据安全性问题吗？**

### 归并排序

归并排序的实现分为**递归实现**与**非递归(迭代)实现**。递归实现的归并排序是算法设计中**分治策略**的典型应用，我们**将一个大问题分割成小问题分别解决**，然后用所有小问题的答案来解决整个大问题。非递归(迭代)实现的归并排序首先进行是两两归并，然后四四归并，然后是八八归并，一直下去直到归并了整个数组。

归并排序算法主要依赖归并(Merge)操作。归并操作指的是将两个已经排序的序列合并成一个序列的操作，归并操作步骤如下：

1.申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列

2.设定两个指针，最初位置分别为两个已经排序序列的起始位置

3.比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置

4.重复步骤3直到某一指针到达序列尾

5.将另一序列剩下的所有元素直接复制到合并序列尾

### 堆排序

堆排序是指利用堆这种数据结构所设计的一种选择排序算法。堆是一种近似完全二叉树的结构（通常堆是通过一维数组来实现的），并满足性质：以最大堆（也叫大根堆、大顶堆）为例，其中父结点的值总是大于它的孩子节点。

我们可以很容易的定义堆排序的过程：

1.由输入的无序数组构造一个最大堆，作为初始的无序区

2.把堆顶元素（最大值）和堆尾元素互换

3.把堆（无序区）的尺寸缩小1，并调用heapify(A, 0)从新的堆顶元素开始进行堆调整

4.重复步骤2，直到堆的尺寸为1

**Java系统提供的Arrays.sort函数。对于基础类型，底层使用快速排序。对于非基础类型，底层使用归并排序。请问是为什么？**

这是考虑到排序算法的稳定性。对于基础类型，相同值是无差别的，排序前后相同值的相对位置并不重要，所以选择更为高效的快速排序，尽管它是不稳定的排序算法；而对于非基础类型，排序前后相等实例的相对位置不宜改变，所以选择稳定的归并排序

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

拜占庭将军问题及Raft算法

拜占庭将军问题：两台服务器之间因为网络攻击，导致通信内容不一致的情况

解决这个问题，可以使用Raft算法。

什么是 Raft 算法？

Raft 算法是一种简单易懂的共识算法。它依靠 状态机 和 主从同步 的方式，在各个节点之间实现数据的一致性。类似于zookeeper

<https://mp.weixin.qq.com/s/4oXDWHbs5_gtVWrnxJs_UQ>

# 问题定位

## 一个java内存泄漏的排查案例

把堆dump下来再用MAT等工具进行分析，也可以用更轻量级的在线分析，用jmap查看存活的对象情况（jmap -histo:live [pid]），定位到一个HashTable有问题后用神器btrace跟踪Hashtable.put调用的堆栈。

# 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。