# java 面试题

**什么时候会栈/堆溢出**

方法执行时创建的栈帧超过栈的[深度](https://so.csdn.net/so/search?q=%E6%B7%B1%E5%BA%A6&spm=1001.2101.3001.7020)（递归太深）

对象存在堆中，不断new对象会堆溢出

**java创建对象的过程**

定义符号引用

检查类是否被加载过

分配[内存](https://so.csdn.net/so/search?q=%E5%86%85%E5%AD%98&spm=1001.2101.3001.7020)空间

字段设置默认值

设置对象头:标记为新生代/老年代，哈希码，元数据信息等

执行[构造方法](https://so.csdn.net/so/search?q=%E6%9E%84%E9%80%A0%E6%96%B9%E6%B3%95&spm=1001.2101.3001.7020)

**java堆分代**

根据存活周期，将堆分为新生代，老年代，永久代

分代原因：不同代用不同的Gc算法，提高GC性能

新生代：存放新对象，存活率低，每次gc回收80%的空间，gc频率高，用复制算法（eden区：存放对象的区域）

老年代：新生代中多次gc后仍存活的对象，存活率高，gc频率低，用标记-整理/清除

永久代（方法区）：存储类信息，常量，静态变量等，一般不GC

**对象什么时候会进入老年代**

新生代GC15次后，-XX:MaxTenuringThreshold=10 设置阈值

大对象直接进入老年代，-XX:PretenureSizeThreshold 设置阈值字节大小

**垃圾回收机制**

对象创建时，GC监控对象的地址，大小，使用情况

增加堆内存，能降低GC频率，但垃圾也越多，每次GC的停顿时间也越长

触发Young GC（新生代GC，Minor GC）：新生代中Eden 区内存不足

触发Full GC(老年代GC，Major JC)：整个堆(新生和老年)的回收

System.gc()手动触发，不会立即触发GC,在内存不够时才真正执行

老年代/永久代空间不足，大对象直接分片到老年代失败

**垃圾回收器**

Serial:GC时用户线程全部暂停(STW)，GC单线程工作(标记并回收)，业务越多，服务器停顿时间越长

Parallel：Serial基础上，GC多线程工作，减少停顿时间

CMS(和用户线程并发，多线程)：STW后只标记根对象的第一层(短停顿)，用户线程继续运行，CMS也并发标记垃圾(容易产生错/漏标)；再次STW，检测垃圾标注是否错误(避免错标)

错标：对象设为null,认为是垃圾，1秒后对象又建立引用

漏标：留着下一次GC

G1(java默认):内存分区；每个区域独立回收

**垃圾回收GC算法，哪个效率最高**

**标记-清除**：标记要回收的对象，标完后统一回收(效率低，有空间碎片)

**标记-整理**：清除后将存活的对象都向一端移动，无空间碎片，效率低

**复制**：内存分两块，只用一块(Eden区)。满后将存活对象复制到另一块，清除用完的块，无空间碎片，效率高，内存利用率减半

**分代收集**：新生代GC大量死去，少量存活，用复制；老年代GC大量存活，少量死去，用标记清理\整理

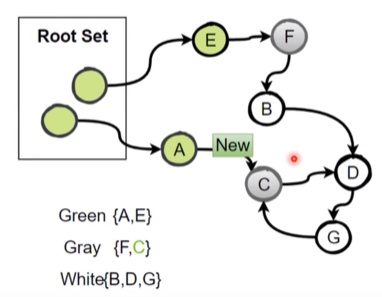
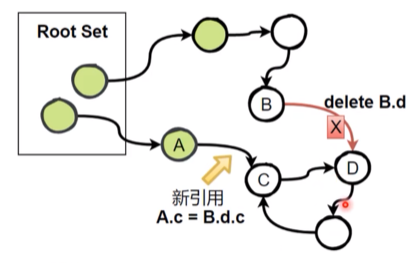
**垃圾对象检测算法**

**引用计数法**：对象没有被引用就是垃圾,但两个垃圾相互引用将无法回收

**根搜索法(推荐)**：可达性分析，从根对象出发，根据引用能到达的对象就不是垃圾

**GC根对象**：栈，，类静态属性，常量，本地方法栈中引用的对象

**三色标记法**：弥补G1错标问题，并发GC下(用户边使用，GC边标记产生错标)，已遍历的标绿，下一次将要遍历的标灰，其他标白



**循环引用的对象如何进行gc**  
将引用指针指向null,gc通过可达性分析到达不了对象后就会被回收

**java标识符的命名规则**

数字,字母,下划线，$组成，不能数字开头

**java如何跳出多重嵌套循环**

在循环外添加标记A：,然后break A;

break和continue不美观

**switch中支持哪些数据类型**

char，byte、short、int 及包装类,Enum,String

**Integer a = new Integer(1); Integer b = new Integer(1) a==b吗？**

Integer的-128~127缓存在内存中，从内存中取，是同一个对象

**Collection和Collections的区别**

Collection是接口，list,set的父接口

Collections是类，包含集合操作的静态方法(搜索，排序，线程安全化)

**隐式/显式转换**

隐式(自动)转换：小范围自动转为大范围 (1==1.0f,结果为true)

显示(强制)类型转换：大范围强制转为小范围类型 byte b = (byte)k;

**如果byte a=1,b=2,c；那么c = a + b 会报错吗？ c+=a会报错吗？为什么**

加减乘除运算规则(双目都用final修饰的类型不会转换)  
 有double都转成 double  
 有float都转成 float  
 有long都转成 long  
 其他都转成 int  
+=类型的赋值运算会将类型自动强转，所以不会报错

**三目运算符的返回类型怎么判断？**

一定要有返回值， 返回类型为范围大的

int num1 = 1; float num2 = 1.0f;

intNumber = (1==1) ? num1 : num2; 会报错，返回类型是float

**if(a+1.0=4.0) 可以吗**

不可用，赋值的返回值是赋值结果，if条件必须是布尔类型

**为什么浮点运算会有误差，该用什么类型**

4.0-3.6=0.40000001

10进制转2进制计算，2进制的小数无法精确表达，有误差

用Bigdecimal

System.out.println(1.5F == 1.5); //true 小数可以用二进制表示  
System.out.println(1.4F == 1.4); //false 小数不能用二进制完全表示（丢失精度）

**值传递和引用传递**

参数是基本类型，传递字面量值

参数是引用类型，传递对象的地址值

**浅拷贝和深拷贝**

浅拷贝：只拷贝对象指针，新对象改变，原对象也会改变，使用等号

深拷贝：拷贝原对象的所有数据，新对象改变，原对象不会改变

**String,StringBuffer,StringBuilder的区别**

String: 每次修改都返回新对象，效率低

StrngBuffer: 线程安全，可对自身进行操作,内容可变(不返回新对象)

StringBuilder: 线程不安全，可对自身进行操作,内容可变(不返回新对象)

速度StringBuilder>StringBuffer>String

**String str = "aa" 和 String str = new String("aa")的区别**

前者先去常量池找 "aa",没有则创建一个，有则直接引用。创建了0/1个对象

后者先去常量池找 “aa”,没有则创建一个，有则直接引用,然后再去堆中new一个str对象。创建了1/2个对象

**String a = “a” + new String(“b”) 创建了几个对象？**

四个，常量池中的”a”, “b”, 堆中的 对象a(拼接后的新对象)和 new String(“b”)

**java中创建对象的方式有哪些**

new ：效率高

反射： 类加载器创建，比较繁琐，性能比直接的java代码要慢很多

序列化：

Clone：

**线程安全的集合有哪些**

Collections.synchronizedList()

hashtable, concurrentHashMap，

CopyOnWriterArraySet

**将集合转换成线程安全的方式**

map = Collections.synchronizedMap(map);  
set = Collections.synchronizedSet(set);  
list = Collections.synchronizedList(list);

**list,set,map的区别**

list:数组，有序存储

set:无重复值，无序存储

map:键值对存储，不允许键重复，无序

**fail-fast机制**

遍历集合时，结构改变立刻抛出异常

场景：遍历迭代hashmap，arrayList,中用remove移除元素，就会抛出异常

多线程中，一个线程迭代，另一个线程remove元素，也会抛出异常

**哈希算法**

任意长度的输入，通过散列算法，变成固定长度的输出

**什么是哈希表(散列表)，使用场景有哪些**

key直接访问数据，一个列表，计算下标快速访问

根据下标快速定位元素是否存在，返回下标

**Array、ArrayList和LinkedList的区别**

Array:原生数组, 存基本数据类型/引用类型，大小固定

ArrayList：底层是数组，只能存引用类型，动态扩容，访问效率高

LinkedList：底层是双向链表，只能存引用类型，动态扩容，插入/删除效率高

**Java中队列和栈用数组还是链表实现，为什么**

链表实现，栈和队列中增删为主

**ArrayList扩容机制**

初始化容量：

new ArrayList<>(); //初始容量为10

new ArrayList<>(n); //初始容量为指定的容量n

new ArrayLIst<>(set); //初始容量为指定集合的大小

扩容：数组满后，创建1.5倍容量的新数组，原数组内容拷贝过去

**LinkedList新增和删除数据默认是在头部还是尾部**

add()默认从尾部添加

remove()删除头部并返回

removeFirst()

removeLast()

**LinkedList为什么不用单向链表，而用双向链表**

双向链标又前后指针，遍历时可进可退

**1.7后为什么由双向循环链表改为双向链表**

去掉头节点，增加头/尾节，表头/尾插入时能快速获得表头/尾节点

**Set如何去重**

先用hashCode判断，不同则添加，再用equals判断，不同则添加，相同则表示重复去除

**实现map接口的类有哪些，区别是什么**  
**HashMap**,链地址法解决冲突，数组+链表+红黑树实现，链表长度大于8的时候转为红黑树,允许key/value为null，默认初始大小是16，每次2n扩容  
**Hashtable:**不允许key/value为Null,默认大小是11，每次扩容为原来的2n+1，方法用synchronized修饰，线程安全，数据操作时锁住整张表，速度慢  
**ConcurrentHashMap(**多线程下使用,分段锁,不使用synchronized),线程安全，map拆成多个桶，只锁一个桶，不影响其他桶使用，速度快。  
**LinkedHashMap:**HashMap+双向链表，按照插入顺序遍历  
**TreeMap**：数据根据key排序,按排序遍历

**HashMap如何解决哈希冲突**

链地址法

**&和&&的区别**

&：与运算符

&&：短路与运算符，前面的是false则后面的不用看

例：username != null && username.equals("") 必须使用&&来短路,否则null值用equals会报错

**java中获取当前时间的方法**

LocalDate、LocalTime、LocalDateTime

**servlet处于网络模型的哪一层**

servlet接受请求,并返回处理结果，处于应用层

servlet作用域：request,session,servletcontext

**逻辑左移，逻辑右移，算数右移**

<< 逻辑左移=算数左移，低位补0

>> 算数右移,带符号右移，高位用符号位补

>>> 逻辑右移，高位补0

取模运算(%)结果的符号怎么判断

余数的符号跟被除数(前面的数)符号相同

**其他零散点**

+= 会进行自动强转(自动装箱功能)

%和\*优先级相同

-n = ~n+1

'0' 强转为int是48

**类加载器**

不同的类加载器加载同一个类，在方法区生成两个不同的类，堆中生成两个不同的类实例，不指定类加载器，默认使用系统类加载器  
  
**双亲委派机制**

类加载器有层次结构，请求加载时，先向上查找是否加载过(加载过的不再加载)，再向下委派(当层无法加载则让下层加载)，避免重复加载类(同个类被不同类加载器加载产生两个类)  
自定义类加载器->系统类加载器->扩展类加载器->启动类加载器

**全盘委托机制**

类加载器加载类时，该类所依赖和引用的其他类都由该类加载器加载

**Jvm的内存结构**

程序计数器：记录正在执行的指令地址,线程私有

Java虚拟机栈：存放局部变量、操作数栈、动态链接、方法出口等信息,线程私有

本地方法栈： 非java的接口(C语言)，java调用和操作系统相关API交互。

Java堆：存对象,线程共享

方法区：存储类信息（构造方法、接口定义）、常量、静态变量、即时编译后的字节码，常量池等数据,线程共享

**常量池**

1.8之后，运行时常量池在方法区中，属于元空间一部分

字符串常量在堆中的字面量常量池中

**对象结构**

1.对象头包含Mark Word, 类型指针，（数组对象有数组长度）

Mark Word：用于存储对象自身的运行时数据，如哈希码、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等

类型指针：对象指向它的类元数据的指针，虚拟机通过指针来确定对象是哪个类的实例

2.实例数据：存储的是真正的有效数据，即各个字段的值

3.对齐填充：对象的大小总是8字节的整数倍，充当占位符，补齐空白

**java8新特性**

接口默认方法：接口有具体方法，default修饰

Lambda 表达式：匿名函数 ()->{}

函数式接口：方法作为参数传递

方法引用：类直接引用方法(BaseInfoVO::format)

Stream API：将集合改成流的方式

日期时间API：引进LocalDate、LocalTime、LocalDateTime，改进了日期时间的管理。

Optional 类：解决空指针异常的一种方式

**解释型/编译型语言**

解释型：每次运行都要把源代码解释成机器码(js)

编译型：编译成可执行文件，以后不用再编译(JAVA)，修改代码后需要重新编译

**代码运行过程**

Java源代码编译成jvm可执行的字节码文件

jvm将字节码解释成机器可执行的二进制文件

程序运行

**字节码是什么，有什么作用，为什么能跨平台，为什么C语言不能跨平台**

编译后生成的.class文件，java虚拟机解析字节码后转为机器码在机器上运行，不同平台需要安装对应的java虚拟机，就能跨平台

C语言编译后直接在平台上运行，不依赖于虚拟机，所以不跨平台，但运行速度快

**java的特点，与其他语言的比较**

移植性好，可以跨平台，

自带GC机制，内存分配不同人工管理，

没有指针

面向对象编程

**Error和Exception的区别**

Error:不可预料的异常，JVM不可处理，如OutOfMemoryError等

Exception(父类)：可以捕获的异常  
 RuntimeException 运行时异常（编译时不显式处理，JVM会自行处理，可通过预检查规避(a==null)）空指针异常， 数组下标越界异常，算数类异常， 类型转换异常  
 CheckedException 检查时异常：编译时必须显式处理，IO异常等

**JDK, JRE, JVM的区别**

JDK: Java开发工具包

JRE: Java运行环境

JVM: Java虚拟机

JDK > JRE > JVM

**如何防止内存泄漏**

内存泄漏：对象作为key,修改对象后，hashcode变化，无法再访问到value,value将一直存在占用内存。

避免方式：list，map等集合，大对象使用完后要赋值为null；避免死循环重复创建/往集合添加元素，导致内存不足

**基本数据类型和包装类型有哪些**

基本数据类型：byte, short, int, long(L), float(f), double(d), char(2字节), boolean

包装类型： Byte, Short, Integer, Long, Float, Double, Charater, Boolean

**基本数据类型和包装类型的区别**

基本数据类型：高效，有默认值，对象中的属性在栈中，方法中的局部变量在堆中

包装类型： 默认为null,有额外的方法(equals),需要new,在堆中

**自动拆装箱**

基本数据类型和包装类之间自动转换(赋值和判断)

**Int的取值范围**

-2^31 ~ 2^31-1

**为什么重写equals()还要重写hashcode()**

hashCode重写前：返回对象的内存地址，重写后返回对象哈希值。

equals重写前：返回对象的内存地址，重写后判断对象是否相等

HashMap和HashSet中，先hashcode()比较哈希值，确定存储位置是否相同（相同位置上用红黑树/链表来存储），避免每次都用equals比较，可以提高效率。

hashcode相等，equals不一定相等；

equals相等，hashcode一定相等

**==和equals的区别，equals能比较两个对象是否相同吗**

== :基本数据类型（值是否相同）；引用类型（地址值是否相同）

String重写了hashCode和equals方法,所以equals能比较string是否相同

**i++如何保证线程安全**

I++执行过程(读取i, i+1, 计算后的值赋给i)，非原子操作，存在线程安全问题

解决方式

1.使用synchronized关键字

2.使用AtomicInteger原子类(使用了CAS算法，效率高)

**JUC是什么**

java.util.concurrent包的简称

提供并发编程的工具类，定义线程，线程池、异步 IO 和轻量级任务框架

**JUC中常用工具类**

信号量 - Semaphore

计数器 - CountDownLatch

循环栅栏 - CyclicBarrier

两个线程之间的交换器 - Exchanger

**阻塞队列**

阻塞队列空时，从队列中拿数据会阻塞

阻塞队列满时，往队列里加数据会阻塞

**final, finalize(), finally的区别**

final：最终关键字

finalize():System.gc()时，GC用finalize()回收垃圾，不会立即执行

finally:异常捕获中关键字（记录结束日志，关闭文件流资源，释放锁）

**异常处理机制**

异常捕捉：try...catch...finally

子类异常放在父类异常前面捕获

catch和finally不能同时省略

finally中return语句会使try/catch中的return失效，因为在那之就已经return了

finally中没有return语句：try中会把return的变量地址缓存起来，finally不会影响 try中的return，但是可以修改对象中的内容

异常抛出(用在方法声明后)：throws+异常名:

异常抛出(用在方法体内)：throw+异常名

**高效赋值数组的方式**

System.arraycopy(array1, 0, array4, 0, array1.length);

**执行main方法发生了什么**

类加载，执行静态方法，最后执行main静态方法

**List的add()返回什么类型**

布尔类型

**泛型**

将类型定义为形参，List<String>

**泛型擦除**

泛型信息只存在于编译阶段，java运行期(已经生成字节码文件后)，泛型信息会被擦除

**反射**

运行时获得对象类型、方法等，加载编译期间未知的类，获得类的方法和属性

使用场景，获得类名，实例化

JDBC连接时，使用Class.forName(驱动类名)，加载数据库驱动类

通过getName(),getClass(),getMethod()来获得类的相关信息

缺点：

性能问题：效率要比正常操作慢很多。在性能要求高或经常被执行的代码中少用反射。

安全问题：使用反射通常需要程序的运行没有安全方面的限制。

健壮性问题：反射允许代码执行一些不被允许的操作，会破坏java程序的结构

反射+反序列化创建实例

Class<?> clazz = Class.forName(className);

Object instance = objectMapper.readValue(jsonString, clazz);

**注解的实现原理**

反射实现

**Serializable接口**

类只有实现了Serializable接口，对象才能被序列化。

序列化：对象换为字节序列(二进制，txt文件)

**不实现序列化，也可以存到数据库中，为什么还要序列化？**

下次使用的时候，快速重建副本，便于数据传输，尤其是在远程调用的时候

**反射对单例的破坏**

反射获得私有的构造器，把构造器的[setAccessible](https://so.csdn.net/so/search?q=setAccessible&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/csj731742019/article/details/_blank)属性设为true，无视构造器的私有性，通过getInstance()创建实例，再反射创建一个实例，就不是单例了

解决方案：用全局变量标记是否实例化过，第二次实例化时，抛出异常。

**哈希冲突**

哈希算法的输入是无限的，结果是有限的，存在不同数据计算后得到相同哈希值

**解决哈希冲突的方法**

开放定址法(线性探测法)：从冲突位置开始，hash表中找下个空闲位置放入

链式寻址法：原位置上用链表来存储冲突的值

再hash法:用不同的哈希函数再算一次，直到不冲突为止。性能较差

**再哈希为什么可以得到不同的哈希值**

存在不同的hash函数，不同的hash函数计算的结果也会不一样

**hashmap频繁插入大量数据怎么优化**

初始化时指定较大的容量，避免连续扩容影响效率

**HashMap扩容机制**

默认容量是16，每次扩容为2n(容量都为2^n,如：初始化容量为20则阈值为32)

**为什么HashMap的链表从头插 改为 尾插**

Jdk1.7头插法：T1，T2线程同时对HashMap进行插入，达到扩容条件，都会扩容

链表A-->B，扩容后，AB下标依然相同，仍在同一链表中，不同线程扩容后的新数组是独有

T1线程执行扩容，先拿到A，然后T1线程被挂起。

T2线程进行扩容，将A放入新数组，B头插法插到A前，B.next->A

T1执行，A放入新数组，然后也将B头插法插到A前，B执行完后。发现B还有下一个(T2干的)，又将A插入到B前面

get时，发现A.next=B,B.next=A;形成环状，导致查询出现死循环

Jdk1.8尾插法，直接在末尾插入对应元素，不会变化原节点的next关系,不会出现死循环

**HashMap扩容时的负载因子**

扩容因子=0.75，容量达到75%时进行扩容

负载因子越小则哈希表越疏散，空间浪费严重

负载因子越大则哈希冲突概率越大，代价也越大，会降低效率

**ConcurrentHashMap如何保证扩容时的安全性**

hashmap扩容机制前提下，每个读取操作用CAS判断数据的一致性

**HashMap每次扩容后大小都是2的幂**

通过(2^n-1)&hash值的方式获取数组下标，&运算会比%运算更快。

2的幂会哈希结果更均匀，分布更均匀，减少哈希冲突

**Hashmap为什么长度>8要转为红黑树来提高效率**

链表长度大于8，且总数组长度大于64时转为红黑树，

链表长度小于6时转为链表(避免频繁转换)

红黑树的平均查找长度为log(n)=log(8)=3

链表的平均查找长度为n/2=8/2=4

红黑树在检索上有更好的性能

**hashmap为什么用红黑树代替链表而不是avl树/b树/跳表**

AVL树必须满足平衡条件，插入/删除都要旋转来保持平衡，适合查询多，修改少的场景，红黑树适合修改多，查询少的场景

B+树的非叶结点不存数据，每个结点能存储的关键字更多。适合大量数据的情况，如果数据不多，数据会“挤在”一个结点里面。效率和链表差不多

跳表需要维护多层链表，用空间换时间，跳表适合排序/范围查找的场景

**Hashmap为什么线程不安全**

JDK1.7中，并发执行扩容时会造成循环链表和数据覆盖  
JDK1.8中，并发执行插入时会发生数据覆盖的情况,（脏数据，幻读，不可重复读的情况）

**HashMap 的get原理**

调用hashcode(key)获得hash值,换为数组下标，定位到指定位置上

该位置没有元素，返回报错

该位置有元素，通过equals遍历链表，直到equals为true，返回节点的值

如果equals一直返回false,返回报错，表示无元素

**HashMap 的put原理**

调用hashcode(key)获得hash值，换为数组下标，定位到指定位置上

该位置没有元素，则插入节点

该位置有元素，通过equals遍历链表，直到equals为true，则替换该节点值

如果equals一直返回false,则在链表最后添加节点

**ConcurrentHashMap 如何计算size**

ConcurrentHashMap是多线程，为了准确要锁住put,remove,clean方法，效率低

Jdk8中在put,remove,clean方法执行前对modCount变量进行+1,判断size计算前后是否有改变，有则重新计算

**HashSet数据结构，跟HashMap有什么区别**

HashSet存放对象，底层也是HashMap,只用到了key，value由默认变量填充

HashMap存放键值对

**ConcurrentHashMap底层是什么**

jdk1.7中ConcurrentHashMap

用分段锁，数据分成n段，每段一个锁，对被访问的段上锁，其他段可以正常访问

分段锁称为Segment(继承自ReentrantLock),内部有一个HsahEntry数组,数组中每个元素是个链表

**jdk1.7中ConcurrentHashMap为什么线程安全，什么时候加锁，什么是分段锁**

使用了分段锁，put时锁住当前分段，get时不加锁

什么是分段锁：只锁住当前分段(几个下标位置共享锁)，分段越多，锁越多

**jdk1.8后ConcurrentHashMap为什么线程安全**

取消分段锁，改为Synchronized + CAS + volatile(锁粒度更细)

数组下标位置为空，进行适用CAS来put数据

数组下标位置已有数据，对该位置的链表加Synchronized锁(偏向-轻量-重量对锁优化)

**强引用，软引用，弱引用，虚引用的区别**

强引用：new出来的对象，内存不足时也不会回收

软引用：内存不足时会回收

弱引用：被GC发现就会回收

虚拟用：被GC发现就会回收

**静态代理和动态代理**

代理模式：一个对象不能/不适合直接引用另一个对象，使用代理对象作为中介

静态代理：程序创建时就手动创建好的代理

动态代理: 程序运行时，反射动态创建的代理

**JDK和GClib代理**

JDK动态代理：目标对象实现一个接口，创建接口代理，不支持类代理

CGLIB动态代理：创建子类代理

**如何查看java程序占用的内存**

使用jconsole工具

使用jstat工具

如 top、ps、vmstat 等命令

**Java命令**

-Xmn设置了新生代大小

-Xss :栈内存大小

-Xms:堆内存大小

-Xmm：堆的最大内存大小

性能上map比array慢很多

**parallelStream 是异步的吗**

是

**雪花算法**

符号位+时间戳+服务器ID+序列号

用于分布式ID生成，性能快，依赖服务器时钟，如果时钟回拨，可能导致生成ID重复

