# 复习计划

1.数据库索引，基本调优，查询要掌握

2.基本的数据结构要会写，常见的几种排序，几种查询

3.深入理解jvm书籍早上拿来看,最基本要理解gc算法，垃圾什么时候被回收，类是如何被加载，java的内存模型，多线程和各种锁

ParallelScavenge :并行清除

4面试常问的计算机网络，要能熟练的说出来，tcp ip http 等在网络上的应用，三次握手，四次挥手

5.晚上刷leetcode，牛客题库，看面经复习

6.学习前端,最低要求要回写页面，要能写js

.

7.复习学习linux,git,还有三大框架

8．常见的设计模式

9.写好简历，辞职开始秋招

------------2019.09.07写于一个渣渣秋招时危机感的复习计算，

if you want something,just do it！ ——mrmuqan

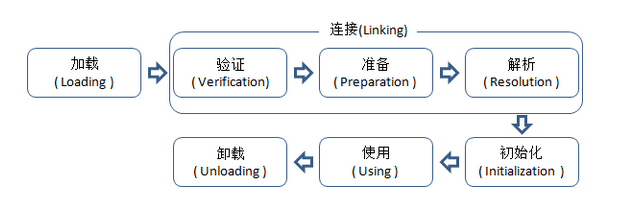
1. jvm

# 加载的时机

* 遇到new、getstatic、putstatic或invokestatic这4个字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这4条指令最常见的Java场景是：使用new关键字实例化对象的时候、读取或设置一个类的静态字段（被final修饰、已在编译期把结果放入到常量池的静态变量除外）的时候，以及调用一个类的静态方法的时候。
* 使用java.lang.reflect包的方法对类进行反射调用时候，如果类没有进行初始化，则需先触发其初始化。
* 当初始化一个类的时候，发现其父类还没有进行过初始化，则需要先触发父类的初始化。
* 当虚拟机启动时，用户需要制定一个要执行的主类（包含main（）方法的那个类），虚拟机会先初始化这个类。

# 类加载过程

　　类从被加载到虚拟机内存开始 ，到卸载出内存为止，它的整个生命周期分为：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）七个阶段。其中验证、准备和解析三个部分统称为连接（Linking）。

加载

* 在加载阶段，虚拟机需要完成以下三件事情：
  + 通过一个类的全限定名来获取定义此类的二进制字节流。
  + 将这个二进制字节流所代表的静态存储结构转化为方法区的运行时数据结构。
  + 在Java堆中生成一个能代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口。

## 验证

　验证是连接阶段的第一步，这一阶段的目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

　不同虚拟机对类验证的实现不同，但大致上都会完成下面四个阶段的检验过程：

### 文件格式验证

　这个阶段验证字节流是否符合Class文件格式的规范，并且能够被当前版本的虚拟机处理。例如：是否以魔数0xCAFEBABE（作用是表示是一个可以被虚拟机使用的Class文件）开头；主、次版本号是否在当前虚拟机的处理范围之类；指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量；Class文件中各个部分及文件本身是否有被删除或附加的其他信息。

　该验证阶段的主要目的是保证输入的字节流能够正确的被解析并存储于方法区之中，格式上符合描述一个Java类型信息的要求。

　这阶段的验证是基于字节流进行的，经过了这个阶段的验证之后，字节流才会进入内存的方法区中进行存储，所以后面的三个验证阶段全是基于方法区的存储结构进行的。

### 元数据验证

　第二阶段是对字节码描述的信息进行语义分析的，以保证描述的信息符合Java语言规范的要求。

　如：这个类是否有父类（除了java.lang.Object外，其余的类都有父类）；这个类的父类是否继承了不允许被继承的类（被final修饰的类）；如果这个类不是抽象类，是否实现了其父类或接口中要求实现的所有方法。

### 字节码验证

　第三个阶段是整个验证过程中最复杂的一个阶段，主要工作是进行数据流和控制流进行分析。这个阶段将对类的方法体进行校验分析。

　例如：保证方法体中的类型转换是有效的，比如可以将一个子类对象赋值给父类数据类型，这是安全的，但是把父类对象赋值给子类数据类型，则是危险和不合法的。

### 符号引用验证

　最后一个阶段的校验发生在虚拟机将符号引用转化为直接引用的时候。

　例如：符号引用中通过字符串描述的全限定名是否能找到对应的类；符号引用的类、字段和方法的访问性（private、protected、default、public）是否可被当前类访问。

## 准备

　准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配。这个时候进行内存分配的仅包括类变量（被static修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在Java堆中。其次这里说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为：

public static int value = 123;

 　那么变量value在准备阶段之后的初始值为0而不是123，因为这个时候尚未执行任何Java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器<clinit>()方法之中，所以把value赋值为123的动作将在初始化阶段才会被执行。

　至于“特殊情况”，是指:

public static final int value = 123;

 　如果类字段的字段属性表中存在ConstantValue属性，则会在准备初始阶段初始化为指定的值，所以在标注为final之后，value的值在准备阶段初始化为123而不是0。

## 解析

　解析阶段是虚拟机将常量池中的符号引用替换为直接引用的过程。

* + 符号引用（Symbolic References）：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能够无歧义地定位到目标即可。符号引用于虚拟机实现的内存布局无关，引用的目标并不一定加载到了内存中。
  + 直接引用（Direct References）：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机实现的内存布局相关，同一符号引用在不同虚拟机实例上翻译出来的直接引用一般不同。如果有了直接引用，那么引用的目标一定存在了内存中。

　解析动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行。

## 初始化

　类初始化阶段是类加载过程的最后一步，在这个阶段才真正执行类中定义的Java程序代码。

　在准备阶段，常量已经赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序制定的主观去初始化类的变量。或者从另外一个角度来说：初始化阶段是执行类构造器<clinit>()方法的过程。

* + <clinit>()方法是由编译器自动收集类中所有类变量的赋值动作和静态语句块（static{}块）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块中可以赋值，但是不能访问。
  + <clinit>()方法与类的构造函数不同，它不需要显示地调用父类构造器，虚拟机会保证在子类的<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。
  + 接口中如果有变量的赋值操作，那么接口与类一样也会生成<clinit>()方法。但接口与类不同的是，执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法，只有当父接口中的变量被使用时父接口才会被初始化。另外，接口的实现类在初始化时也一样不会执行接口的<clinit>()方法。
  + 虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确地加锁和同步，如果多个线程同时去初始化一个类，那么只有一个线程去执行这个类的<clinit>()方法，其他线程都需要阻塞等待。

1. **排序**

**算法：Bubblesort(s[],n)**

**输入:n个元素组成的一个序列s[],每个元素由[0…n-1]之间的下标确定,元素之间可以比较大小**

**输出:重新调整S[]中元素的次序，使得它们按照非降次序排列**

**{**

**从s[0]和s[1]开始，依次检查每对相邻的元素；**

**只要它们位置颠倒，则交换位置；**

**反复执行上述操作，直到每一对相邻元素的次序都符合要求；**

**}**



# 面经

1. 为什么重写equals还要重写hashcode?

equals方法在object中比较的是两个对象是否具有相同的引用，如果有，则返回true

“==”比较的是两个内存地址是否相等，如果相等则表示引用的是同一个对象。

String类重写了equals方法，使其比较的是字符的序列，而不再是内存地址

public boolean equals(Object obj) { return (this == obj); }

答: hashCode 方法用于散列集合的查找，equals 方法用于判断两个对象是否相等，如果只重写了 equals 方法，两个对象 equals 返回了true，但是如果没有重写 hashCode 方法，集合还是会插入元素。这样集合中就出现了重复元素了。分析hashmap的put方法

**public V put(K key, V value) {**

**if (key == null)**

**return putForNullKey(value);**

**//通过key的hashcode计算hash值**

**int hash = hash(key.hashCode());**

**//通过hash值和table数组的长度计算出元素存放在table数组的位置**

**int i = indexFor(hash, table.length);**

**//table[i]的位置已经存在元素，遍历链表**

**for (Entry<K,V> e = table[i]; e != null; e = e.next) {**

**Object k;**

**//调用 equals 方法判断key是否相等，若相等，该key对应的键值对已经存在，用新的value取代旧的value**

**if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {**

**V oldValue = e.value;**

**e.value = value;**

**e.recordAccess(this);**

**return oldValue;**

**}**

**}**

**modCount++;**

**// 若该key对应的键值对不存在，将key-value封装成Entry对象添加到table[i]处 ，头插法。**

**addEntry(hash, key, value, i);**

**return null;**

**}**

**1.HashMap 的 put 方法实际上是先调用 hashCode 定位到数组的位置**

**2.如果该数组的位置上已经存在元素了，即 table[i] ！= null，那么遍历链表，调用 equals 方法判断key是否相等。如果相等，表明这个key对应的键值对已经存在，那么新的 value 会覆盖掉旧的 value。如果遍历链表都没有找到key，那么表明这个 key 对应的键值对不存在，直接插入，作为链表的头节点。**

**总结**:hashmap中key不可以重复，value可以重复，往hashmap中插入元素的时候，先要定位到数组位置：

int hash = hash(key.hashCode());

//通过hash值和table数组的长度计算出元素存放在table数组的位置

int i = indexFor(hash, table.length);

如果没有重写hashcode方法，那么就无法定位到同一个位置，hashmap里面会插入相同的key值，所以需要重写hashcode方法

如果只重写了一个equals方法，那么就无法定位到同一个位置，集合还是会插入元素。

如果重写了hashcode方法，确保两个对象都能够定位到相同的位置，那么就可以遍历这条单向链表，使用equals方法判断两个对象是否相同，如果相同，那么就不插入了（HashMap的实现仍然插入，但是覆盖掉旧的value）。如果不相同，就插入到链表的头节点处。

final关键字

**1.被final修饰的类不可继承**

**2.被final修饰分方法不可重写**

**3.被final修饰的变量不可改变，变量不可改变说的是变量的引用不变，变量的值可以改变**

**4.被final修饰的方法jvm会尝试为之寻求内联，这对提升Java的效率是非常重要的，加入不能确定方法会不会被继承，那么尽量将方法定义成为final的**

**5.被final修饰的常量，在编译阶段会存入调用类的常量池中，可以参考类的加载机制最后部分和java运行时内存区域**

**注意：内联需要去了解学习！！！**

**static关键字**

**1.被static修饰的方法属于类方法，可以直接类名.方法名来调用，而不需要重新new一个类出来**

**2.被static修饰的变量，属于类变量，可以直接类名.变量名，来调用，而不需要重新new一个类出来**

**被static修饰的方法和变量统一属于类的静态资源，是类的实例之间共享的，还言之，一处变，处处变。**

**静态块**

public class A

2 {

3 private static int a = B();

4

5 static

6 {

7 System.out.println("Enter A.static block");

8 }

9

10 public static void main(String[] args)

11 {

12 new A();

13 }

14

15 public static int B()

16 {

17 System.out.println("Enter A.B()");

18 return 1;

19 }

20 }

**静态块里面的代码只执行一次，且只在初始化类的时候执行**