# 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。



## 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。

## String,StringBuffer,StringBuild

**三者比较**

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

**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。

**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

**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。

## Exception,Error,Throwable

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 。

**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。

**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

## java反射

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。

**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

## 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

**Override（子类重写父类方法重写）和Overload（同一类中方法重载）**

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

## 其它

**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

**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。**

**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。

**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。**

**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。

**Switch能用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。

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**

Clone，equals，hashCode，getClass，wait，notify，notifyAll，toString

**对象的引用分为四种级别**

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

Boolean b = map != null ? map.get("test") : false; 运行以上的代码会报空指针

原因是通过反编译以后map.get(“test”)变成了：((Boolean)map.get("test")).booleanValue())

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。

如果代码这么写，就不会报错

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>

## TCP，HTTP1.0，HTTP1.1

**TCP 有哪些状态，相应状态的含义**

LISTEN - 侦听来自远方TCP端口的连接请求；

SYN-SENT -在发送连接请求后等待匹配的连接请求；

SYN-RECEIVED - 在收到和发送一个连接请求后等待对连接请求的确认；

ESTABLISHED- 代表一个打开的连接，数据可以传送给用户；

FIN-WAIT-1 - 等待远程TCP的连接中断请求，或先前的连接中断请求的确认；

FIN-WAIT-2 - 从远程TCP等待连接中断请求；

CLOSE-WAIT - 等待从本地用户发来的连接中断请求；

CLOSING -等待远程TCP对连接中断的确认；

LAST-ACK - 等待原来发向远程TCP的连接中断请求的确认；

TIME-WAIT -等待足够的时间以确保远程TCP接收到连接中断请求的确认；

CLOSED - 没有任何连接状态；

**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

**TCP“3次握手，4次挥手”**

TCP是一种面向连接的单播协议，在发送数据前，通信双方必须在彼此间建立一条连接。所谓的“连接”，其实是客户端和服务器的内存里保存的一份关于对方的信息，如ip地址、端口号等。

**三次握手**

第一次握手：客户端发送网络包，服务端收到了。这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。

第二次握手：服务端发送网络包，客户端收到了。这样客户端就能得出结论：服务端的发送能力、客户端的接收能力是正常的。

第三次握手：客户端发包，服务端收到了。这样服务端就能得出结论：客户端的接收、发送能力，服务端的发送、接收能力是正常的。

**四次挥手**

TCP连接是双向传输的对等的模式，就是说双方都可以同时向对方发送或接收数据。当有一方要关闭连接时，会发送指令告知对方，我要关闭连接了。这时对方会回一个ACK，此时一个方向的连接关闭。但是另一个方向仍然可以继续传输数据，等到发送完了所有的数据后，会发送一个FIN段来关闭此方向上的连接。接收方发送ACK确认关闭连接。注意，接收到FIN报文的一方只能回复一个ACK, 它是无法马上返回对方一个FIN报文段的，因为结束数据传输的“指令”是上层应用层给出的，我只是一个“搬运工”，我无法了解“上层的意志”

其实3次握手的目的并不只是让通信双方都了解到一个连接正在建立，还在于利用数据包的选项来传输特殊的信息，交换初始序列号ISN。

3次握手是指发送了3个报文段，4次挥手是指发送了4个报文段。注意，SYN和FIN段都是会利用重传进行可靠传输的。

**DDOS防护**

Nginx优化，最好隐藏其版本号

页面优化，隐藏一些头部信息，避免黑客扫描到

IP黑名单

IP日访问次数

限制并发数：限制单个IP的并发数，限制总并发数，这里建议大家使用漏桶算法限流，来整形流量请求。

配置CDN

配置缓存

## 时间复杂度

基本操作执行次数 T（n）

渐进时间复杂度（asymptotic time complectiy） :若存在函数 f（n），使得当n趋近于无穷大时，T（n）/ f（n）的极限值为不等于零的常数，则称 f（n）是T（n）的同数量级函数。记作 T（n）= O（f（n）），称O（f（n）） 为算法的渐进时间复杂度，简称时间复杂度。

渐进时间复杂度用大写O来表示，所以也被称为大O表示法。

## 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

## 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

# 集合

## Java中常用的集合有哪些？

**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:

PriorityQueue, （优先级队列）

ConcurrentLinkedQueue, （基于链表的并发队列）

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

ArrayBlockingQueue, （基于数组的并发阻塞队列）

LinkedBlockingQueue, （基于链表的FIFO阻塞队列）

PriorityBlockingQueue, （带优先级的无界阻塞队列）

SynchronousQueue （并发同步阻塞队列）

Deque:

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

ArrayDeque, （数组双端队列）

**Java的数据结构相关的类实现原理？**

Java常用的数据包括数组，链表，树。有些类是基于单一的数据结构实现的，比如ArrayList，Vector是基于数组实现的，LinkedList是基于链表实现的。有些类是基于复合数据结构实现的，比如HashMap是基于数组+链表或数组+树的数据结构实现的，LinkedHashMap更是在HashMap的基础上再加上一层链表数据结构来维护集合的顺序。

## ArrayList，Vector

**ArrayList，Vector的区别**

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。

## HashMap

**讲讲自己对HashMap的理解？**

HashMap是以key-value键值对存储数据的集合，JDK1.7以前，HashMap底层使用的是数组+单向链表的数据结构（在数据结构中，一般称之为“链表散列“），其主要包含put，get方法，put方法用于存储数据，get用于获取数据。当使用put方法时，首先会根据key的hashcode与数组大小长度-1进行逻辑与运算，也就是通过求模计算出该数据在数组中存储的位置。然后遍历该位置下的链表，使用equals判断这个链表是否包含这个key，如果存在则替换该entry对象的值，并将旧的value返回。如果不存在则将新数据放在链表的首位。存储完成以后会判断数组的大小是否超过阀值，超过则对数组进行扩容，并遍历整个hashmap，将数据放入新的数组中。get方法则相对简单，只是通过hashcode找到数组对应位置，遍历数组，使用equals方法查找是否包含对应的值，有则返回，无则返回null。

JDK1.8之后HashMap数据结构做了改动，在链表数据达到8个时会转换为使用红黑树的数据结构存储，put，get方法也做了相应的改动。put方法存储数据时会先判断当前数据结构是否为红黑树，是则通过红黑树的方式存储数据，而且扩容的时候也有些改变，1.7是遍历整个数组，并将链表的数据一个一个重新计算hashcode到新的数组中，但1.8则是先生成新的两个链表，再放到新的数组中。

**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

**HashMap为什么用红黑树，红黑树的特点？**

**HashMap使用红黑树，无非也是为了加快检索速度。**

**红黑树的特点：**

（1）每个节点或者是黑色，或者是红色。

（2）根节点是黑色。

（3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]

（4）如果一个节点是红色的，则它的子节点必须是黑色的。

（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。[这里指到叶子节点的路径]

**HashMap是否线程安全？**

HashMap非线程安全的，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

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。

**HashMap和HashTable的区别**

Hashtable是线程安全的，基本上每个方法都使用了synchronized来保证线程安全性。

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

**HashMap和WeakHashMap的区别**

HashMap的key是对实际对象的强引用,而WeakHashMap的key是对实际对象的弱引用。弱引用（WeakReference）的特性是：当gc线程发现某个对象只有弱引用指向它，那么就会将其销毁并回收内存。WeakReference也会被加入到引用队列queue中。

理解了相关概念之后，对WeakHashMap的实际应用感到很好奇。然后发现tomcat的源码里，实现缓存时会用到WeakHashMap。

**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。

**HashMap的hashcode的作用**

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。

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；

**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。

**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。

## ConcurrentHashMap

**ConcurrentHashMap 是如何实现的？**

**1.7、1.8 实现有何不同？为什么这么做？**

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

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。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。

**ConcurrentHashMap的size方法是如何实现的？**

看源码是使用baseCount与CounterCell类进行累加，baseCount是一个volatile变量，ConcurrentHashMap的put方法后有个addCount方法会对这个变量做 CAS 加法。而CounterCel，在没有并发的情况下，使用一个 baseCount volatile 变量就足够了，当并发的时候，CAS 修改 baseCount 失败后，就会使用 CounterCell 类了，会创建一个这个对象，通常对象的 volatile value 属性是 1。在计算 size 的时候，会将 baseCount 和 CounterCell 数组中的元素的 value 累加，得到总的大小，但这个数字仍旧可能是不准确的。因为都是并发插入的。

**如果你觉得size方法返回值不准确，那如果让你自己实现，你觉得应该怎么实现呢？**

Size是因为有并发插入操作才会使得其不准确，如果要让其准确返回，除非启用单线程做累加操作，或者可以用AtomicInteger变量进行记录。

**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。

## TreeMap和LinkedHashMap

**TreeMap和LinkedHashMap是如何保证它的顺序的？**

TreeMap只能根据key来排序，是不能根据value来排序的，默认是根据key的自然排序来组织（比如integer的大小，String的字典排序），自定义排序的话需要实现Comparator的compare方法。

LinkedHashMap底层维护了一个链表来保证数据之间的顺序。

## 其它

**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。

**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。

**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？

# JVM

## JVM虚拟机内存划分或者问内存模型

JDK1.7时JVM内存划分为程序计数器，虚拟机栈，本地方法栈，堆，方法区，JDK1.8将方法区移入到了直接内存中，称为元空间。

**程序计数器**主要有两个作用：在多线程的情况下，程序计数器用于记录当前线程执行的位置。

字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。

**虚拟机栈：**用于描述Java方法执行的状态，每个方法在执行的同时都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。

局部变量表主要存放了编译器可知的各种数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）。

Java方法有两种返回方式：return 语句，抛出异常

**本地方法栈：**虚拟机栈的作用非常相似，区别是：虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的Native方法。

**堆：**在虚拟机启动时创建，唯一目的就是存放对象实例，所有的对象实例以及数组都要在堆上分配。Java堆还可以细分为：新生代和老年代：再细致一点有：eden区、s0区、s1区都属于新生代，tentired 区属于老年代。

**方法区：**是各个线程共享内存区域，用于存储已被虚拟机加载的类信息，常量，静态变量，即时编译器编译后的代码等数据。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载。HotSpot 虚拟机中方法区也常被称为 “永久代。

**直接内存**

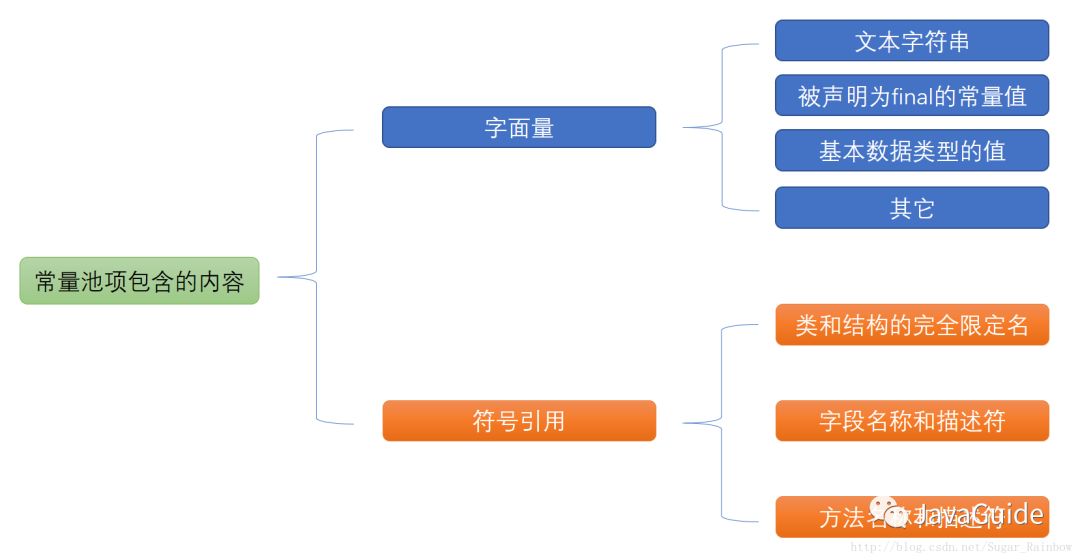
直接内存并不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用。而且也可能导致 OutOfMemoryError 异常出现。

**运行时常量池**

运行时常量池是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有常量池信息（用于存放编译期生成的各种字面量和符号引用）

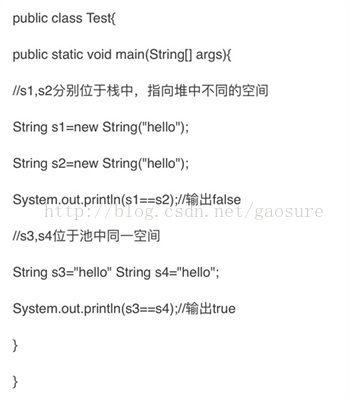
既然运行时常量池时方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

JDK1.7及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆（Heap）中开辟了一块区域存放运行时常量池。



**常量池的作用**非常重要，常量池中除了包含代码中所定义的各种基本类型（如int、long等等）和对象型（如String及数组）的常量值外，还包含一些以文本形式出现的符号引用，比如：类和接口的全限定名，字段的名称和描述符，方法和名称和描述符。

Double包装类没有实现常量池技术，因此Double d1 = 1.0; 相当于Double d1 = new Double(1.0);，是从堆new一个对象，



结果分析：  用new String() 创建的字符串不是常量，不能在编译器就能确定，所以new String()创建的字符串不放入常量池中，他们有自己的地址空间。  String对象(内存)的不变性机制会使修改String字符串时，产生大量的对象，因为每次改变字符串，都会生成一个新的String。Java为了更有效的使用内存，常量池在编译期遇见String字符串时，它会检查该池内是否已经存在相同的String字符串，如果找到，就把新变量的引用指向现有的字符串对象，不创建任何新的String常量对象，没找到再创建新的。所以对一个字符串对象的任何修改，都会产生一个新的字符串对象，原来的依然存在，等待垃圾回收。  String a = "test";  String b = "test";  String b = b + "java";  a、b同时指向常量池中的常量值”test“，b = b + "java"之后，b原先指向一个常量，内容为"test"，通过对b进行+"java"操作后，b之前所指向的那个值没有改变，但此时b不指向原来那个变量值了，而指向了另一个String变量，内容为”test java“。原来那个变量还存在于内存之中，只是b这个变量不再指向它了。

小结：

1、**常量池维护的常量仅仅是【-128至127】这个范围内的常量，如果常量值超过这个范围，就会从堆中创建对象，不再从常量池中取。**如：Integer i1 = 400；Integer i2 = 400；很明显超过了127，无法从常量池中获取常量，就用从堆中new新的Integer对象，这是i1和i2就不相等了。

**2、String类型也实现了常量池技术，但是稍微有点不同，String型是先检测常量池中有没有对应字符串，如果有，则取出来，如果没有，则把当前的添加进去。**

**Java堆是如何实现的？**

新生代中由Eden和Survivor0，Survivor1组成，默认三者的比例是8：1：1，一般对象都在新生代的Eden区生成，新生代的回收机制采用复制算法，在MinorGC的时候，我们都留一个存活区用来存放存活的对象，真正进行的区域是Eden+其中一个存活区，当我们的对象时长超过一定年龄时（默认15，可以通过参数设置），将会把对象放入老年代，当然大的对象会直接进入老年代。老年代采用的回收算法是标记整理算法。

## Java内存模型

Java内存模型（Java Memory Model ,JMM）就是一种符合内存模型的规范。

Java内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存的变量的主内存中拷贝过来的，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。

而JMM就作用于工作内存和主存之间数据同步过程。他规定了如何做数据同步以及什么时候做数据同步。

**Java内存模型的实现**

了解Java多线程的朋友都知道，在Java中提供了一系列和并发处理相关的关键字，比如volatile、synchronized、final、concurren包等。其实这些就是Java内存模型封装了底层的实现后提供给程序员使用的一些关键字。

**堆内存分配**

为了提升内存分配效率，在年轻代的Eden区HotSpot虚拟机使用了两种技术来加快内存分配 ，分别是bump-the-pointer和TLAB（Thread-Local Allocation Buffers）。由于Eden区是连续的，因此bump-the-pointer技术的核心就是跟踪最后创建的一个对象，在对象创建时，只需要检查最后一个对象后面是否有足够的内存即可，从而大大加快内存分配速度；而对于TLAB技术是对于多线程而言的， 它会为每个新创建的线程在新生代的Eden Space上分配一块独立的空间，这块空间称为TLAB（Thread Local Allocation Buffer），在TLAB上分配内存不需要加锁，一般JVM会优先在TLAB上分配内存，如果对象过大或者TLAB空间已经用完，则仍然在堆上进行分配。

## JVM调优

调优工具：Jconsole,jProfile,visualVM

如何调优：

堆信息查看：

可查看堆空间大小分配（年轻代，年老代，持久代）

提供即时的垃圾回收功能

垃圾监控

有了堆信息查看方面的功能，我们一般可以顺利解决以下问题：

年老代年轻代大小划分是否合理

内存泄漏

垃圾回收算法设置是否合理

查看FullGC的频率

Linux使用jstat命令查看jvm的GC情况，通常运行命令如下：

jstat -gc 12538 5000

即会每5秒一次显示进程号为12538的java进成的GC情况

性能调优建议： jvm调优没有一个固定模板配置说必须如何操作，它需要根据系统的情况不同对待。 但是可以有如下建议：

1、初始化内存和最大内存尽量保持一致，避免内存不够用继续扩充内存。最大内存不要超过物理内存，例如内存8g，你可以设置最大内存4g/6g但是不能超过8g否则加载类的时候没有空间会报错。

2、gc/full gc频率不要太高、每次gc时间不要太长、根据系统应用来定。

## 类加载机制

**什么是双亲委派机制？**

双亲委派机制的前提是双亲委派模型，它要求除顶层启动类加载器外其余类加载器都应该有自己的父类加载器；类加载器之间通过复用关系来复用父加载器的代码。当需要加载一个类时，首先会检查请求加载的类是否已经被加载过；若没有被加载过：递归调用父类加载器的loadClass()；方法进行加载，如果父类加载器和启动类加载器均无法加载请求，则调用自身的加载功能。

实现双亲委派模型的代码都集中在java.lang.ClassLoader的loadClass()方法中；

**双亲委派模型的优点：**

Java类伴随其类加载器具备了带有优先级的层次关系，确保了在各种加载环境的加载顺序。

保证了运行的安全性，防止不可信类扮演可信任的类。

**双亲委派过程：**

当一个类加载器收到类加载任务时，立即将任务委派给它的父类加载器去执行，直至委派给最顶层的启动类加载器为止。如果父类加载器无法加载委派给它的类时，将类加载任务退回给它的下一级加载器去执行。

变量覆盖：ParentClass ubclass = new SubClass();

子类跟父类有相同普通变量的情况下，subClass.protectedField输出的是父类定义的变量

**什么情况下我们需要破坏双亲委派模型；**

双亲委派模型很好地解决了各个类加载器的基础类统一问题(越基础的类由越上层的加载器进行加载)，基础类之所以被称为“基础”，是因为它们总是作为被调用代码调用的API。但是，如果基础类又要调用用户的代码，那该怎么办呢。

或者是由于用户对程序的动态性的追求导致的，例如OSGi的出现。在OSGi环境下，类加载器不再是双亲委派模型中的树状结构，而是进一步发展为网状结构。

**对象在内存中的初始化过程**

而 JVM 虚拟机执行 class 字节码的过程可以分为七个阶段：加载、验证、准备、解析、初始化、使用、卸载。

类的加载指的是将类的.class文件以二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个。Class对象，用来封装类在方法区内的数据结构。然后会对文件进行JVM规范校验，代码逻辑校验。校验通过以后，会在堆中为类的静态变量及成员变量分配内存，并将其初始化为默认值，接着执行构造代码块，对对象进行初始化，在堆中为类的对象分配内存。

**类加载器**

– Java 虚拟机自带的加载器

根类加载器（Bootstrap，使用C++编写，程序员无法在java代码中获得该类）

扩展类加载器（Extension，使用java代码实现）

系统类加载器（System，使用java代码实现）

– 用户自定义的类加载器

必须继承java.lang.ClassLoader类

用户可以定制类的加载方式

Class的getClassLoader()方法如果这个类是根类加载器加载会返回一个空

Class clazz = Class.forName("java.lang.String");

clazz.getClassLoader();//返回null

**类的初始化顺序**

1. 父类--静态变量

2. 父类--静态初始化块

3. 子类--静态变量

4. 子类--静态初始化块

5. 父类--变量

6. 父类--初始化块

7. 父类--构造器

8. 子类--变量

9. 子类--初始化块

10.子类--构造器

**类的加载指的是**将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。一个类只有一个Class对象，只有JAVA虚拟机才能创建Class对象。

**将class文件加载到内存到过程：**

加载：查找并加载类的二进制数据，加载到内存里面就会生成class对象，即调用ClassLoader的loaderClass方法。

连接

– 验证：确保被加载的类的正确性

– 准备：为类的 静态变量分配内存，并将其初始化为默认值

– 解析： 把类中的符号引用转换为直接引用

初始化：为类的静态变量赋予正确的初始值，这个正确的初始值，指我们赋予静态变量的值。如：

private static int a = 3;在准备阶段会被a设为默认值0，执行初始化时把3赋予了a。静态代码块从上到下执行

**Java程序对类的使用方式可分为两种：**主动使用，被动使用

所有的Java虚拟机实现必须在每个类或接口被Java程序“ 首次主动使用”时才初始化他们

**主动使用（ 六种）**

– 创建类的实例

– 访问某个类或接口的静态变量，或者对该静态变量赋值。（int c = Test.a;或Test.a=b）

– 调用类的静态方法

– 反射（如Class.forName(“com.shengsiyuan.Test”)）

– 初始化一个类的子类

– Java虚拟机启动时被标明为启动类的类（JavaTest）。比如说有Test.java这个文件，编译后有Test.class,Parent.class,Child.class，然后启动时使用java Test（Test有main方法），这就是被标明为启动类的类

除了以上六种情况，其他使用Java类的方式都被看作是对类的被动使用，都不会导致类的初始化

类加载七个阶段以及案例分析

而 JVM 虚拟机执行 class 字节码的过程可以分为七个阶段：加载、验证、准备、解析、初始化、使用、卸载。

加载：在这个阶段，JVM 的主要目的是将字节码从各个位置（网络、磁盘等）转化为二进制字节流加载到内存中，接着会为这个类在 JVM 的方法区创建一个对应的 Class 对象，这个 Class 对象就是这个类各种数据的访问入口。

验证：JVM规范校验，代码逻辑校验。

准备（重要）：

1.内存分配的对象：为类成员变量分配即静态变量内存，在准备阶段，JVM 只会为「类变量」分配内存，而不会为「类成员变量」分配内存。

2.初始化的类型。在准备阶段，JVM 会为类变量分配内存，并为其初始化。但是这里的初始化指的是为变量赋予 Java 语言中该数据类型的零值，而不是用户代码里初始化的值。

例如下面的代码在准备阶段之后，sector 的值将是 0，而不是 3。public static int sector = 3;

但如果一个变量是常量（被 static final 修饰）的话，那么在准备阶段，属性便会被赋予用户希望的值。

例如下面的代码在准备阶段之后，number 的值将是 3，而不是 0。

public static final int number = 3;

解析：当通过准备阶段之后，JVM 针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符 7 类引用进行解析。这个阶段的主要任务是将其在常量池中的符号引用替换成直接其在内存中的直接引用。

初始化（重要）：到了初始化阶段，用户定义的 Java 程序代码才真正开始执行。在这个阶段，JVM 会根据语句执行顺序对类对象进行初始化，一般来说当 JVM 遇到下面 5 种情况的时候会触发初始化：

遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，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条指令的最常见的Java代码场景是：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、已在编译器把结果放入常量池的静态字段除外）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的时候，如果类没有进行过初始化，则需要先触发其初始化。

当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

当虚拟机启动时，用户需要指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主类。

当使用 JDK1.7 动态语言支持时，如果一个 java.lang.invoke.MethodHandle实例最后的解析结果 REF\_getstatic,REF\_putstatic,REF\_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出触发其初始化。

使用：当 JVM 完成初始化阶段之后，JVM 便开始从入口方法开始执行用户的程序代码。这个阶段也只是了解一下就可以。

卸载：当用户程序代码执行完毕后，JVM 便开始销毁创建的 Class 对象，最后负责运行的 JVM 也退出内存。这个阶段也只是了解一下就可以。

案例分析1:

class Grandpa

{

static

{

System.out.println("爷爷在静态代码块");

}

public Grandpa() {

System.out.println("我是爷爷~");

}

}

class Father extends Grandpa

{

static

{

System.out.println("爸爸在静态代码块");

}

public Father()

{

System.out.println("我是爸爸~");

}

}

class Son extends Father

{

static

{

System.out.println("儿子在静态代码块");

}

public Son()

{

System.out.println("我是儿子~");

}

}

public class InitializationDemo

{

public static void main(String[] args)

{

new Son(); //入口

}

}

输出结果：

爷爷在静态代码块

爸爸在静态代码块

儿子在静态代码块

我是爷爷~

我是爸爸~

我是儿子~

静态字段，只有直接定义这个字段的类才会被初始化（执行静态代码块），因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。

案例分析2

public class Book {

public static void main(String[] args)

{

staticFunction();

}

static Book book = new Book();

static

{

System.out.println("书的静态代码块");

}

{

System.out.println("书的普通代码块");

}

Book()

{

System.out.println("书的构造方法");

System.out.println("price=" + price +",amount=" + amount);

}

public static void staticFunction(){

System.out.println("书的静态方法");

}

int price = 110;

static int amount = 112;

}

上面这个例子的输出结果是：

书的普通代码块

书的构造方法

price=110,amount=0

书的静态代码块

书的静态方法

下面我们一步步来分析一下代码的整个执行流程。

在上面两个例子中，因为 main 方法所在类并没有多余的代码，我们都直接忽略了 main 方法所在类的初始化。但在这个例子中，main 方法所在类有许多代码，我们就并不能直接忽略了。

当 JVM 在准备阶段的时候，便会为类变量分配内存和进行初始化。此时，我们的 book 实例变量被初始化为 null，amount 变量被初始化为 0。

当进入初始化阶段后，因为 Book 方法是程序的入口，根据我们上面说到的类初始化的五种情况的第四种：当虚拟机启动时，用户需要指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主类。JVM 会对 Book 类进行初始化。

JVM 对 Book 类进行初始化首先是执行类构造器（按顺序收集类中所有静态代码块和类变量赋值语句就组成了类构造器），后执行对象的构造器（先收集成员变量赋值，后收集普通代码块，最后收集对象构造器，最终组成对象构造器）。

对于 Book 类，其类构造方法可以简单表示如下：

static Book book = new Book();

static

{

System.out.println("书的静态代码块");

}

static int amount = 112;

于是首先执行static Book book = new Book();这一条语句，这条语句又触发了类的实例化。与类构造器不同，于是 JVM 执行 Book 类的成员变量，再搜集普通代码块，最后执行类的构造方法，于是其执行语句可以表示如下：

int price = 110;

{

System.out.println("书的普通代码块");

}

Book()

{

System.out.println("书的构造方法");

System.out.println("price=" + price +", amount=" + amount);

}

于是此时 price 赋予 110 的值，输出：「书的普通代码块」、「书的构造方法」。而此时 price 为 110 的值，而 amount 的赋值语句并未执行，所以只有在准备阶段赋予的零值，所以之后输出「price=110,amount=0」。

当类实例化完成之后，JVM 继续进行类构造器的初始化：

static Book book = new Book(); //完成类实例化

static

{

System.out.println("书的静态代码块");

}

static int amount = 112;

即输出：「书的静态代码块」，之后对 amount 赋予 112 的值。

到这里，类的初始化已经完成，JVM 执行 main 方法的内容。

public static void main(String[] args)

{

staticFunction();

}

即输出：「书的静态方法」。

分析方法论

从上面几个例子可以看出，分析一个类的执行顺序大概可以按照如下步骤：

确定类变量的初始值。在类加载的准备阶段，JVM 会为类变量初始化零值，这时候类变量会有一个初始的零值。如果是被 final 修饰的类变量，则直接会被初始成用户想要的值。

初始化入口方法。当进入类加载的初始化阶段后，JVM 会寻找整个 main 方法入口，从而初始化 main 方法所在的整个类。当需要对一个类进行初始化时，会首先初始化类构造器，之后初始化对象构造器。

初始化类构造器。初始化类构造器是初始化类的第一步，其会按顺序收集类变量的赋值语句、静态代码块，最终组成类构造器由 JVM 执行。

初始化对象构造器。初始化对象构造器是在类构造器执行完成之后的第二部操作，其会按照执行类成员变成赋值、普通代码块、对象构造方法的顺序收集代码，最终组成对象构造器，最终由 JVM 执行。

如果在初始化 main 方法所在类的时候遇到了其他类的初始化，那么继续按照初始化类构造器、初始化对象构造器的顺序继续初始化。如此反复循环，最终返回 main 方法所在类。

面试题一

class Singleton{

//private static Singleton singleton = new Singleton();

public static int counter1;

public static int counter2 = 0;

private static Singleton singleton = new Singleton();

private Singleton(){

counter1++;

counter2++；

｝

public static Singleton getInstance(){

return singleton;

}

}

public class MyTest

{

public static void main(String[] args){

Singleton singleton = Singleton.getInstance();

输出counter1跟counter2;

}

}

如果把private static Singleton singleton = new Singleton();放在最前面的话是输出1跟0。

原因：类加载器在准备阶段会先将counter1跟counter2都赋予默认值0，这时再依次执行静态代码块，首先实例化singleton把值都设为了1，然后发现counter2有显示赋值，所以将用户自己设定的值0赋予counter2，导致counter2最后的结果是0

面试题二

注意：此处是针对final的静态变量

class FinalTest1{

public static final int x = 6/3;

static {

System.out.println("FinalTest2.enclosing\_method()");

}

public FinalTest1() {

System.out.println("FinalTest2.FinalTest2()");

}

}

class FinalTest2{

public static final int x = new Random().nextInt(100);

static {

System.out.println("FinalTest2.enclosing\_method()");

}

public FinalTest2() {

System.out.println("FinalTest2.FinalTest2()");

}

}

public class Test1 {

public static void main(String[] args) {

//只输出2，因为x在编译期就已确定，所以使用的时候不会去初始化类

System.out.println(FinalTest1.x);

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

//输出x的随机数跟FinalTest2.enclosing\_method()，因为x需要运行时才能确定，所以要对类进行初始化

//进不进行初始化就看这个静态代码块是否会执行。

System.out.println(FinalTest2.x);

}

}

面试题三：与上题做对比

class Parent{

static int a = 4;

static {

System.out.println("Parent.enclosing\_method()");

}

}

class Child extends Parent{

static int a = 5;

static {

System.out.println("Child.enclosing\_method()");

}

}

public class Test3 {

static {

System.out.println("Test3.enclosing\_method()");

}

public static void main(String[] args) {

/\*

\* 输出：

\* Test3.enclosing\_method()

Parent.enclosing\_method()

Child.enclosing\_method()

5

\*/

System.out.println(Child.a);

}

}

同一个类加载器不允许对一个类初始化两次

程序中对子类对主动使用会导致父类被初始化；但父类但主动使用并不会导致子类初始化（不可能说生成一个object类会导致所以子类初始化。）

只有当程序访问当静态变量或静态方法确实在当前类或当前接口中定义时，才可以认为是对类或接口对主动使用。如：

面试题四：只是对父类对主动使用，没有对子类主动使用

class Parent1{

static int a = 4;

static {

System.out.println("Parent.enclosing\_method()");

}

static void doSomething() {

System.out.println("Parent.doSomething()");

}

}

class Child1 extends Parent1{

static {

System.out.println("Child.enclosing\_method()");

}

}

public class Test4 {

/\* 输出次序：子类不会被初始化

Parent.enclosing\_method()

4

Parent.doSomething()

\*/

public static void main(String[] args) {

System.out.println(Child1.a);

Child1.doSomething();

}

}

Binary names：类的全称，简单的说就是包名加类名，如果是内部类加内部类名称，如：java.net.URLClassLoader$3$1。其中$3$1指第三个匿名内部类中的第一个匿名内部类

调用ClassLoader类的loadClass方法加载一个类，并不是对类的主动使用，不会导致类的初始化。即指执行到加载，之后的连接与初始化都没有发生。如下例子：

class CL{

static {

System.out.println("CL.enclosing\_method()");

}

}

public class Test5 {

/\* 输出：

---------

CL.enclosing\_method()

\*/

public static void main(String[] args) throws ClassNotFoundException {

ClassLoader loader = ClassLoader.getSystemClassLoader();

Class<?> clazz = loader.loadClass("com.classloader.CL");

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

clazz = Class.forName("com.classloader.CL");

}

}

String 类和常量池

String 对象的两种创建方式：

String str1 = "abcd";

String str2 = new String("abcd");

这两种不同的创建方法是有差别的，第一种方式是在常量池中拿对象，第二种方式是直接在堆内存空间创建一个新的对象。只要使用new方法，便需要创建新的对象。

String 类型的常量池比较特殊。它的主要使用方法有两种：

•直接使用双引号声明出来的 String 对象会直接存储在常量池中。

•如果不是用双引号声明的 String 对象，可以使用 String 提供的 intern 方法。String.intern() 是一个 Native 方法，它的作用是：如果运行时常量池中已经包含一个等于此 String 对象内容的字符串，则返回常量池中该字符串的引用；如果没有，则在常量池中创建与此 String 内容相同的字符串，并返回常量池中创建的字符串的引用。

String s1 = new String("计算机");

String s2 = s1.intern();

String s3 = "计算机";

System.out.println(s2);//计算机

System.out.println(s1 == s2);//false，因为一个是堆内存中的String对象一个是常量池中的String对象，

System.out.println(s3 == s2);//true，因为两个都是常量池中的String对象

String 字符串拼接

String str1 = "str";

String str2 = "ing";

String str3 = "str" + "ing";//常量池中的对象

String str4 = str1 + str2; //在堆上创建的新的对象

String str5 = "string";//常量池中的对象

System.out.println(str3 == str4);//false

System.out.println(str3 == str5);//true

System.out.println(str4 == str5);//false

String s1 = new String("abc");这句话创建了几个对象？

创建了两个对象:先有字符串"abc"放入常量池，然后 new 了一份字符串"abc"放入Java堆(字符串常量"abc"在编译期就已经确定放入常量池，而 Java 堆上的"abc"是在运行期初始化阶段才确定)，然后 Java 栈的 str1 指向Java堆上的"abc"。

8种基本类型的包装类和常量池

•Java 基本类型的包装类的大部分都实现了常量池技术，即Byte,Short,Integer,Long,Character,Boolean；这5种包装类默认创建了数值[-128，127]的相应类型的缓存数据，但是超出此范围仍然会去创建新的对象。

•两种浮点数类型的包装类 Float,Double 并没有实现常量池技术。

Integer i1 = 33;

Integer i2 = 33;

System.out.println(i1 == i2);// 输出true

Integer i11 = 333;

Integer i22 = 333;

System.out.println(i11 == i22);// 输出false

Double i3 = 1.2;

Double i4 = 1.2;

System.out.println(i3 == i4);// 输出false

应用场景：

•Integer i1=40；Java 在编译的时候会直接将代码封装成Integer i1=Integer.valueOf(40);，从而使用常量池中的对象。

•Integer i1 = new Integer(40);这种情况下会创建新的对象。

Integer i1 = 40;

Integer i2 = new Integer(40);

System.out.println(i1==i2);//输出false

Integer比较更丰富的一个例子:

Integer i1 = 40;

Integer i2 = 40;

Integer i3 = 0;

Integer i4 = new Integer(40);

Integer i5 = new Integer(40);

Integer i6 = new Integer(0);

System.out.println("i1=i2 " + (i1 == i2));

System.out.println("i1=i2+i3 " + (i1 == i2 + i3));

System.out.println("i1=i4 " + (i1 == i4));

System.out.println("i4=i5 " + (i4 == i5));

System.out.println("i4=i5+i6 " + (i4 == i5 + i6));

System.out.println("40=i5+i6 " + (40 == i5 + i6));

结果：

i1=i2 true

i1=i2+i3 true

i1=i4 false

i4=i5 false

i4=i5+i6 true

40=i5+i6 true

语句i4 == i5 + i6，因为+这个操作符不适用于Integer对象，首先i5和i6进行自动拆箱操作，进行数值相加，即i4 == 40。然后Integer对象无法与数值进行直接比较，所以i4自动拆箱转为int值40，最终这条语句转为40 == 40进行数值比较。

## GC

**Object类的finalize方法的实现原理**

新建一个对象时，在JVM中会判断该对象对应的类是否重写了finalize方法，且finalize方法体不为空，则把该对象封装成Finalizer对象，并添加到Finalizer链表。

Finalizer类中会初始化一个FinalizerThread类型的线程，负责从一个引用队列中获取Finalizer对象，并执行该Finalizer对象的runFinalizer方法，最终会执行原始对象的finalize方法

你验证过finalize方法是否会执行么？

自己写一个类实现重写finalize方法，再new一个很大的对象，以此触发GC。

**什么时候一个对象会被GC？**

被GC的对象首先会被标记为是否可回收，一般采用有向图的方式记录和管理堆(heap)中的所有对象，从根节点向下检索，不能检索到的则视为这个对象不可用，并标记为可回收。标记后的对象并不会立即回收，对象没有覆盖finalize()方法,或者finalize()方法已经被虚拟机调用过此对象会被直接回收。否则这个对象将会放置在一个队列之中,并在稍后由一个由虚拟机线程去执行其finalize()方法，做一些清除前的工作，例如资源释放等；直至下一次垃圾回收动作发生时才会真正回收对象占用的内存空间。

任何一个对象的finalize()方法都只会被系统自动调用一次,如果对象面临下一次回收,它的finalize()方法不会被再次执行

**为什么要在这种时候对象才会被GC？**

不太清楚设计思路，大概是觉得可能有些对象会使用频繁，但使用间隙可能比较大，比如一些工具类。

**堆回收**

Eden跟Survivor0,Survivor1采用的垃圾回收方式是著名的**“停止-复制（Stop-and-copy）”**清理法（将Eden区和一个Survivor中仍然存活的对象拷贝到另一个Survivor中）

老年代用的算法是标记-整理算法（串行收集器）

**关于方法区即永久代的回收**

永久代的回收有两种：常量池中的常量，无用的类信息，常量的回收很简单，没有引用了就可以被回收。对于无用的类进行回收，必须保证3点：

1. 类的索引实例都已经被回收。2.加载类的ClassLoader已经被回收。3.类对象的Class对象没有被引用（即没有通过反射引用该类的地方）

**GC策略或问垃圾收集算法**

1 引用计数

引用计数器采用分散式管理方式，通过计数器记录对象是否被引用。当计数器为0时，说明此对象已经不再被使用，可进行回收

2**复制（Copying）、**

**3标记-清除（Mark-Sweep）**

**标记失去引用的对象并回收，这个算法会产生碎片**

**4标记-整理（Mark-Compact）**

在标记清除的基础上整理内存空间

**计算对象是否引用**

**1.采用标记计数的方法：**给内存中的对象给打上标记，对象被引用一次，计数就加1，引用被释放了，计数就减一，当这个计数为0的时候，这个对象就可以被回收了。当然，这也就引发了一个问题：循环引用的对象是无法被识别出来并且被回收的。所以就有了第二种方法：

**2.采用根搜索算法：**从一个根出发，搜索所有的可达对象，这样剩下的那些对象就是需要被回收的判断完了哪些对象是没用的，这样就可以进行回收了

**垃圾收集器**

垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收

答： 对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。可以。程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

新生代的垃圾回收器包括Serial、ParNew、Parallel Scavenge，老年代的垃圾回收器包括CMS、Serial Old、Parallel Old。其中新生代的三种垃圾回收器都采用了复制算法，并且会触发STOP THE WORLD。CMS使用标记清除算法，Serial Old、Parallel Old使用标记整理算法。

**Serial收集器**是一个**单线程收集器**（-XX：+UseSerialGC），这个“单线程”不只是说它只会使用一个CPU或者一条线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它垃圾收集结束。

**ParNew收集器**是Serial**收集器的多线程**版本（-XX:+UseParNewGC -XX:ParallelGCThreads），它能够与CMS收集器配合工作，因此，在运行在Server模式下的虚拟机中，ParNew收集器是首选的新生代收集器。

**Parallel Scavenge**这也是一个并行的新生代垃圾收集器，不同于其他收集器（以尽可能缩短垃圾收集时用户线程的停顿时间为目的），它是唯一一个以达到一个可控制的吞吐量为目标的垃圾收集器。

**CMS收集器**的GC周期由6个阶段组成。其中4个阶段(名字以Concurrent开始的)与实际的应用程序是并发执行的，而其他2个阶段需要暂停应用程序线程。

初始标记：为了收集应用程序的对象引用需要暂停应用程序线程，该阶段完成后，应用程序线程再次启动。

并发标记：从第一阶段收集到的对象引用开始，遍历所有其他的对象引用。

并发预清理：改变当运行第二阶段时，由应用程序线程产生的对象引用，以更新第二阶段的结果。

重标记：由于第三阶段是并发的，对象引用可能会发生进一步改变。因此，应用程序线程会再一次被暂停以更新这些变化，并且在进行实际的清理之前确保一个正确的对象引用视图。这一阶段十分重要，因为必须避免收集到仍被引用的对象。

并发清理：所有不再被应用的对象将从堆里清除掉。

并发重置：收集器做一些收尾的工作，以便下一次GC周期能有一个干净的状态。

一个常见的误解是,CMS收集器运行是完全与应用程序并发的。我们已经看到，事实并非如此，即使“stop-the-world”阶段相对于并发阶段的时间很短。

应该指出，尽管CMS收集器为老年代垃圾回收提供了几乎完全并发的解决方案，然而年轻代仍然通过“stop-the-world”方法来进行收集。对于交互式应用，停顿也是可接受的，背后的原理是年轻代的垃圾回收时间通常是相当短的。

挑战

当我们在真实的应用中使用CMS收集器时，我们会面临两个主要的挑战，可能需要进行调优：

堆碎片;对象分配率高

CMS收集器并没有任何碎片整理的机制。

第二个挑战就是应用的对象分配率高。如果获取对象实例的频率高于收集器清除堆里死对象的频率，并发算法将再次失败。

G1算法：

G1把整个堆划分为一个一个等大小的区域（region）。内存的回收和划分都以region为单位；

增量收集把对堆空间划分成一系列内存块，使用时，先使用其中一部分（不会全部用完），垃圾收集时 把之前用掉的部分中的存活对象再放到后面没有用的空间中

回收步骤：

初始标记（Initial Marking） stop the world

并发标记（Concurrent Marking）

最终标记暂停（Final Marking Pause） stop the world

存活对象计算及清除（Live Data Counting and Cleanup）

参考：<https://my.oschina.net/u/2302503/blog/1632775>

G1垃圾回收器是为多处理器和大内存的服务器而设计的，它根据运行JVM过程中构建的停顿预测模型（Pause Prediction Model）计算出来的历史数据来预测本次收集需要选择的Region数量，然后尽可能（不是绝对）满足GC的停顿时间，G1期望能让JVM的GC成为一件简单的事情。G1旨在延迟性和吞吐量之间取得最佳的平衡，它尝试解决有如下问题的Java应用：

堆大小能达到几十G甚至更大，超过50%的堆空间都是存活的对象；

对象分配和晋升的速度随着时间的推移有很大的影响；

堆上严重的碎片化问题；

可预测的停顿时间，避免长时间的停顿。

开启G1

在JDK9之前，JDK7和JDK8默认都是ParallelGC垃圾回收。到了JDK9，G1才是默认的垃圾回收器。所以如果JDK7或者JDK8需要使用G1的话，需要通过参数（-XX:+UseG1GC）显示执行垃圾回收器。

**GC机制和原理**

**GC分哪两种， Minor GC 和Full GC有什么区别？什么时候会触发Full GC？**

Minor GC:

Eden区域满了，或者新创建的对象大小 > Eden所剩空间

当JVM无法为一个新的对象分配空间时会触发Minor GC，比如当Eden区满了。

执行Minor GC操作时，不会影响到永久代。

质疑常规的认知，所有的Minor GC都会触发“全世界的暂停(stop-the-world)”，停止应用程序的线程。对于大部分应用程序，停顿导致的延迟都是可以忽略不计的。

Major GC:清理永久代，但是由于很多MojorGC 是由MinorGC 触发的，所以有时候很难将MajorGC 和MinorGC区分开。

FullGC：是清理整个堆空间—包括年轻代和永久代。Full GC的时候会先触发Minor GC。

新生代直接晋升到老年代的大对象超过了老年代的剩余空间，引发Full GC。

Perm永久代空间不足会触发Full GC，可以让CMS清理永久代的空间。

System.gc()引起的Full GC，可以设置DisableExplicitGC来禁止调用System.gc引发Full GC

**分别采用什么算法？**

Eden跟Survivor0,Survivor1采用的垃圾回收方式是著名的**“停止-复制（Stop-and-copy）”**清理法（将Eden区和一个Survivor中仍然存活的对象拷贝到另一个Survivor中）

老年代用的算法是标记-整理算法（串行收集器）

关于方法区即永久代的回收，永久代的回收有两种：常量池中的常量，无用的类信息，常量的回收很简单，没有引用了就可以被回收。对于无用的类进行回收，必须保证3点：

1. 类的索引实例都已经被回收。2.加载类的ClassLoader已经被回收。3.类对象的Class对象没有被引用（即没有通过反射引用该类的地方）

**YGC的时候，有些对象可能会发生晋升，如果晋升失败怎么处理？**

标记阶段时，会把对象和对应的对象头数据保存在两个栈中，如果晋升失败的话，就把该对象的对象头复原，然后提早进行一次老年代的回收，防止下次进行YGC的时候发生晋升失败

**垃圾收集器配置**

深入JVM-垃圾收集器常用的GC参数

1.与串行回收器相关的参数

-XX:+UseSerialGC:在新生代和老年代使用串行收集器

-XX:SurvivorRatio:设置eden区大小和survivor区大小的比例

-XX:PretenureSizeThreshold:设置大对象直接进入老年代的阈值。当对象的大小超过这个值时，将直接在老年代分配。

-XX:MaxTenuringThreshold:设置对象进入老年代的年龄的最大值。每一次Minor GC后，对象年龄就加1。任何大于这个年龄的对象，一定会进入老年代。

2.与并行GC相关的参数

-XX:+UseParNewGC:在新生代使用并行收集器

-XX:+UseParallelOldGC:老年代使用并行回收收集器

-XX:ParallelGCThreads:设置用于垃圾回收的线程数。通常情况下可以和CPU数量相等，但在CPU数量较多的情况下，设置相对较小的数值也是合理的。

-XX:MaxGCPauseMillis:设置最大垃圾收集停顿时间。他的值是一个大于0的整数。收集器在工作时，会调整Java堆大小或者其他参数，尽可能把停顿时间控制在MaxGCPauseMillis以内。

-XX:GCTimeRatio:设置吞吐量大小。它是0-100的整数。假设GCTimeRatio的值为n，那么系统将花费不超过1/(1+n)的时间用于垃圾收集。

-XX:+UseAdaptiveSizePolicy:打开自适应GC策略。在这种模式下，新生代的大小、eden和survivor的比例、晋升老年代的对象年龄等参数会被自动调整，已达到在堆大小、吞吐量和停顿时间之间的平衡点。

3.与CMS回收期相关的参数

-XX:+UseConcMarkSweepGC:新生代使用并行收集器，老年代使用CMS+串行收集器

-XX:ParallelCMSThreads:设定CMS的线程数量

-XX:CMSInitiatingOccupancyFraction:设置CMS收集器在老年代空间被使用多少后触发，默认为68%

-XX:+UseCMSCompactAtFullCollection:设置CMS收集器完成垃圾收集后是否要进行一次内存碎片的整理

-XX:CMSFullGCsBeforeCompaction:设定进行多少次CMS垃圾回收后，进行一次内存压缩

-XX:+CMSClassUnloadingEnabled:允许对类元数据区进行回收

-XX:CMSInitiatingPermOccupancyFraction:当永久区占用率达到这一百分比时，启动CMS回收(前提是-XX:+CMSClassUnloadingEnabled激活了)

-XX:UseCMSInitiatingOccupancyOnlyn:表示只在到达阈值的时候才进行CMS回收

-XX:+CMSIncrementalMode:使用增量模式，比较适合单CPU。增量模式在JDK 8中标记为废弃，并将在JDK 9中彻底移除。

4.与G1回收期相关的参数

-XX:+UseG1GC:使用G1回收器

-XX:MaxGCPauseMillis:设置最大垃圾收集停顿时间

-XX:GCPauseIntervalMillis:设置停顿间隔时间

5.TLAB相关

-XX:+UseTLAB:开启TLAB分配

-XX:+PrintTLAB:打印TLAB相关分配信息

-XX:TLABSize:设置TLAB大小

-XX:+ResizeTLAB:自动调整TLAB大小

6.其他参数

-XX:+DisableExplicitGC:禁用显式GC

-XX:+ExplicitGCInvokesConcurrent:使用并发方式处理显式GC

Class a = new Class();此时a叫实例，而不能说是对象。实例在栈中，对象在堆中，操作实例实际上是通过实例的指针间接操作对象。多个实例可以指向同一个对象。

## 内存溢出，内存泄漏

**内存泄露的两个条件：无用，无法回收**

长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露因为长生命周期对象持有它的引用而导致不能被回收。其中一个例子：HashSet中存储的对象改变了其用来计算hashcode的字段

新生代的垃圾收集器命名为“minor gc”，老生代的GC命名为"Full Gc 或者Major GC".其中用System.gc()强制执行的是Full Gc.

判断对象是否需要回收的方法有两种： 1.引用计数 2.对象引用遍历

程序计数器是唯一一个不会出现 OutOfMemoryError 的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。

堆内存溢出 OutOfMemoryError（OOM）

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，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。

年老代被占满：java.lang.OutOfMemoryError:java heap space

持久代被占满：java.lang.OutOfMemoryError:permGen space

Java 虚拟机栈会出现两种异常：StackOverFlowError 和 OutOfMemoryError。

•StackOverFlowError： 若Java虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前Java虚拟机栈的最大深度的时候，就抛出StackOverFlowError异常。

•OutOfMemoryError： 若 Java 虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了，无法再动态扩展了，此时抛出OutOfMemoryError异常。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。

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。

## ClassLoader

**ClassLoader 做什么的？**

顾名思义，它是用来加载 Class 的。它负责将 Class 的字节码形式转换成内存形式的 Class 对象。字节码的本质就是一个字节数组 []byte，它有特定的复杂的内部格式。

每个 Class 对象的内部都有一个 classLoader 字段来标识自己是由哪个 ClassLoader 加载的。

JVM 运行实例中会存在多个 ClassLoader，不同的 ClassLoader 会从不同的地方加载字节码文件。

JVM 中内置了三个重要的 ClassLoader，分别是 BootstrapClassLoader、ExtensionClassLoader 和 AppClassLoader。

BootstrapClassLoader 负责加载 JVM 运行时核心类，这些类位于 JAVA\_HOME/lib/rt.jar 文件中，比如 java.util.\*、java.io.\*、java.nio.\*、java.lang.\* 等等。

ExtensionClassLoader 负责加载 JVM 扩展类，比如 swing 系列、内置的 js 引擎、xml 解析器 等等，这些库名通常以 javax 开头，它们的 jar 包位于 JAVA\_HOME/lib/ext/\*.jar 中。

AppClassLoader 才是直接面向我们用户的加载器，它会加载 Classpath 环境变量里定义的路径中的 jar 包和目录。

那些位于网络上静态文件服务器提供的 jar 包和 class文件，jdk 内置了一个 URLClassLoader，ExtensionClassLoader 和 AppClassLoader 都是 URLClassLoader 的子类。

**ClassLoader 传递性**

程序在运行过程中，遇到了一个未知的类，它会选择哪个 ClassLoader 来加载它呢？虚拟机的策略是使用调用者 Class 对象的 ClassLoader 来加载当前未知的类。何为调用者 Class 对象？就是在遇到这个未知的类时，虚拟机肯定正在运行一个方法调用（静态方法或者实例方法），这个方法挂在哪个类上面，那这个类就是调用者 Class 对象。前面我们提到每个 Class 对象里面都有一个 classLoader 属性记录了当前的类是由谁来加载的。

因为 ClassLoader 的传递性，所有延迟加载的类都会由初始调用 main 方法的这个 ClassLoader 全全负责，它就是 AppClassLoader。

**Class.forName**

forName 方法同样也是使用调用者 Class 对象的 ClassLoader 来加载目标类。不过 forName 还提供了多参数版本，可以指定使用哪个 ClassLoader 来加载

通过这种形式的 forName 方法可以突破内置加载器的限制，通过使用自定类加载器允许我们自由加载其它任意来源的类库。根据 ClassLoader 的传递性，目标类库传递引用到的其它类库也将会使用自定义加载器加载。

自定义加载器

自定义类加载器分为以下几个步骤：

1、定义类。继承ClassLoader

2、重写ClassLoader的 findClass方法

ClassLoader 里面有三个重要的方法 loadClass()、findClass() 和 defineClass()。

loadClass() 方法是加载目标类的入口，它首先会查找当前 ClassLoader 以及它的双亲里面是否已经加载了目标类，如果没有找到就会让双亲尝试加载，如果双亲都加载不了，就会调用 findClass() 让自定义加载器自己来加载目标类。ClassLoader 的 findClass() 方法是需要子类来覆盖的，不同的加载器将使用不同的逻辑来获取目标类的字节码。拿到这个字节码之后再调用 defineClass() 方法将字节码转换成 Class 对象。

自定义类加载器不易破坏双亲委派规则，不要轻易覆盖 loadClass 方法。否则可能会导致自定义加载器无法加载内置的核心类库。在使用自定义加载器时，要明确好它的父加载器是谁，将父加载器通过子类的构造器传入。如果父类加载器是 null，那就表示父加载器是「根加载器」。

**Class.forName vs ClassLoader.loadClass**

这两个方法都可以用来加载目标类，它们之间有一个小小的区别，那就是 Class.forName() 方法可以获取原生类型的 Class，而 ClassLoader.loadClass() 则会报错。

URLClassLoader v1 = new URLClassLoader(new URL[]{new URL(v2dir)});

Class<?> depv1Class = v1.loadClass("Dep");

Object depv1 = depv1Class.getConstructor().newInstance();

depv1Class.getMethod("print").invoke(depv1);

即使是同样的字节码用不同的 ClassLoader 加载出来的类都不能算同一个类

## Linux下面查看Jvm性能信息的命令

1. **jstat**： 用于查看Jvm的堆栈信息，能够查看eden,survivor,old,perm等堆区的的容量，利用率信息，对于查看系统是不是有内存泄漏以及参数设置是否合理有不错的意义。例如’’’ jstat -gc 12538 5000 —- 即会每5秒一次显示进程号为12538的java进成的GC情况 ‘’’
2. **jstack**：用来查看Jvm当前的线程dump的，可以看到当前Jvm里面的线程状况，对于查找blocked线程比较有意义
3. **jmap**：用来查看Jvm当前的heap dump的，可以看出当前Jvm中各种对象的数量，所占空间等等；尤其值得一提的是这个命令可以导出一份binary heap dump的bin文件，这个文件能够直接用Eclipse Memory Anayliser来分析，并找出潜在的内存泄漏的地方。
4. **非jvm命令—netstat**：通过这个命令可以看到Linux系统当前在各个端口的链接状态，比如查看数据库连接数等

内存相关问题

**1.内存泄露**是指分配出去的内存没有被回收回来，由于失去了对该内存区域的控制(例如你把它的地址给弄丢了)，因而造成了资源的浪费。Java 中一般不会产生内存泄露，因为有垃圾回收器自动回收垃圾，但这也不绝对，Java堆内也可能发生内存泄露（Memory Leak; 当我们 new 了对象，并保存了其引用，但是后面一直没用它，而垃圾回收器又不会去回收它，这边会造成内存泄露

**2.内存溢出**是指程序所需要的内存超出了系统所能分配的内存（包括动态扩展）的上限

**3.符号引用**：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，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到了内存中。

**4.直接引用**：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的，同一个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会相同。如果有了直接引用，那说明引用的目标必定已经存在于内存之中了。

**5.双亲委派模型**：表示类加载器之间的加载顺序从顶至下的层次关系，加载器之间的父子关系一般都是通过组合来实现，而不是继承。可以防止内存中出现多份同样的字节码，并确保加载顺序

**6.双亲委派模型的工作过程**是：在loadClass函数中，首先会判断该类是否被加载过，加载过则进行下一步—-解析，否则进行加载；如果一个类加载器收到了类加载器的请求，先不会自己尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求（它的搜说范围中没有找到所需的类时，子加载类才会尝试自己去加载）

**7.静态分派和动态分派**：静态分派发生在编译阶段，是指依据静态类型(变量声明时定义的变量类型)来决定方法的执行版本，例如方法重载中依据参数的定义类型来定位具体应该执行的方法；动态分派发生在运行期，根据变量实例化时的实际类型来决定方法的执行版本，例如方法重写；目前的 Java 语言（JDK1.6）是一门**静态多分派、动态单分派**的语言。

**8.动态分派具体实现**Java虚拟机是通过在方法区中建立一个虚方法表，通过使用方法表的索引来代替元数据查找以提高性能。虚方法表中存放着各个方法的实际入口地址，如果子类没有覆盖父类的方法，那么子类的虚方法表里面的地址入口与父类是一致的；如果重写父类的方法，那么子类的方法表的地址将会替换为子类实现版本的地址。方法表是在类加载的连接阶段（验证、准备、解析）进行初始化，准备了子类的初始化值后，虚拟机会把该类的虚方法表也进行初始化。

## 其它

**JVM编译优化**

当虚拟机发现某个方法或代码块的运行特别频繁的时候，就会把这些代码认定为“热点代码”。为了提高热点代码的执行效率，在运行时，即时编译器（Just In Time Compiler ）会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化。

**Java对象访问方式**

1 通过句柄访问

通过句柄访问的实现方式中，JVM堆中会划分单独一块内存区域作为句柄池，句柄池中存储了对象实例数据（在堆中）和对象类型数据（在方法区中）的指针。这种实现方法由于用句柄表示地址，因此十分稳定。

2通过直接指针访问

通过直接指针访问的方式中，reference中存储的就是对象在堆中的实际地址，在堆中存储的对象信息中包含了在方法区中的相应类型数据。这种方法最大的优势是速度快，在HotSpot虚拟机中用的就是这种方式。

**符号引用和直接引用的概念：**

符号引用与虚拟机实现的布局无关，引用的目标并不一定要已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。

直接引用可以是指向目标的指针，相对偏移量或是一个能间接定位到目标的句柄。如果有了直接引用，那引用的目标必定已经在内存中存在。

# 算法和数据结构

## 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：**顺序查找算法，二分查找又称折半查找，插值查找，斐波那契查找

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。

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。

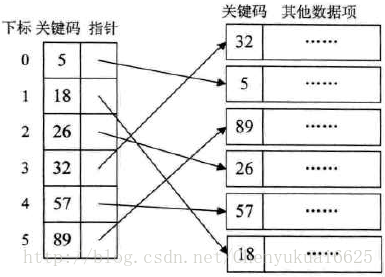
**无序队列查找：**索引查找（稠密索引，分块索引，倒排索引 ），树表查找（二叉树查找算法，平衡查找树之2-3查找树，平衡查找树之红黑树，B树和B+树）

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。

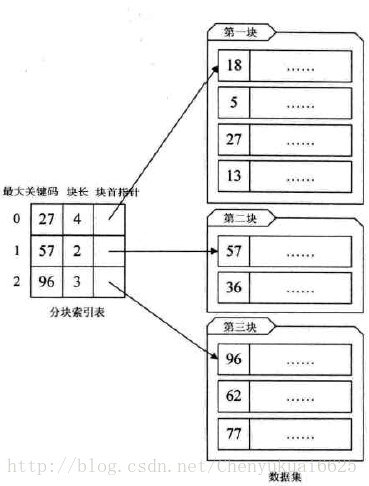
1.2稠密索引   
  稠密索引是指在线性索引中，将数据集中的每个记录对应一个索引项。左边的图像为索引序列，它是是按照关键码有序排列的。索引项有序。



1.3 分块索引

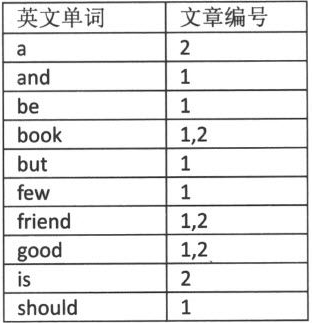
稠密索引因为索引项与数据集的记录个数相同，所以空间代价很大。为了减少索引的个数，我们可以对数据集进行分块，使其分块有序，然后再对每一块建立一个索引项，从而减少索引项的个数。

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。



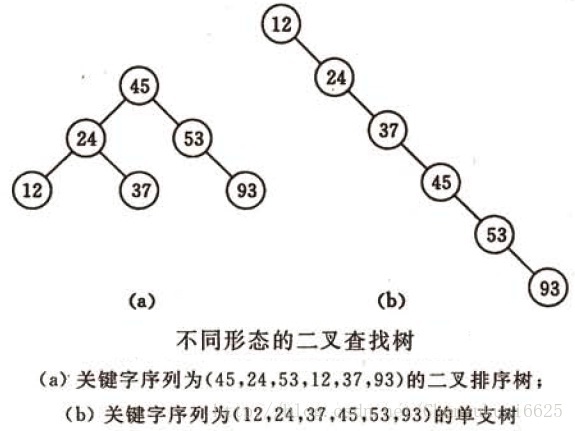
在这里这张单词表就是索引表，索引项的通用结构是次关键码和记录号表。 其中记录号表存储具有相同次字关键字的所有记录的记录号（可以指向记录的指针或者是该记录的主关键字）。因为这种查找方法是通过属性值来确定记录的位置，而不是通过记录来确定属性值，所以我们称其为倒排索引。

## 树

**镜像二叉树有两种算法，一种是递归，一种是迭代。**

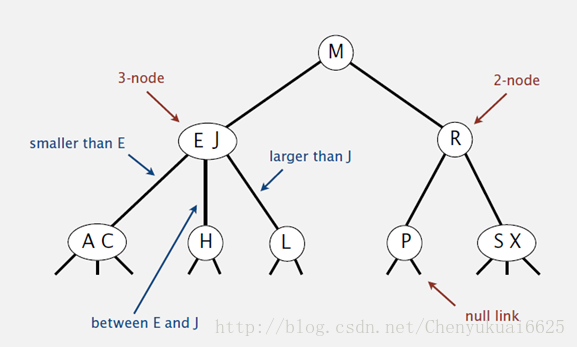
**二叉树查找**

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。



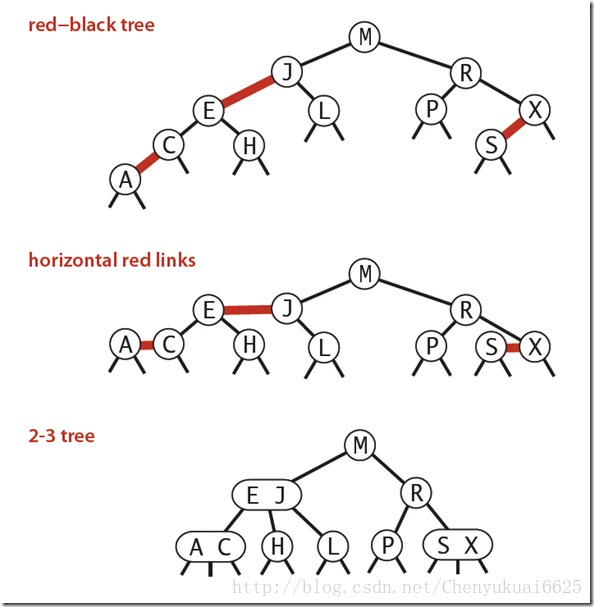
**平衡查找树之2-3查找树**

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）



**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。



**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:



## ？红黑树是什么？红黑树的实现原理和应用场景,时间复杂度?

它的查找、插入和删除操作的时间复杂度是O(lgn)

## 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

## 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>

# 锁

## 锁的分类

锁有哪些分类：悲观锁、乐观锁、独占锁、共享锁、公平锁、非公平锁、分布式锁、自旋锁

**读写锁的机制是什么？**

在Java中读写锁（ReadWritelock）的机制是基于AQS的一种实现，保证读读共享、读写互斥、写写互斥，如果要说机制的话，得从AQS说起：

AQS是Java实现的一种锁机制，互斥锁、读锁写锁、条件产量、信号量、栅栏都是AQS的衍生物，主要工作基于CHL队列，volatile关键字修饰状态state，线程通过CAS去修改状态符，成功了则获取成功，失败了就进队列等待，等待唤醒。

AQS中还有一个很重要的概念就是自旋：在等待唤醒的时候，经常会使用自旋（while(!cas())）的方式，不停地尝试获取锁，直到被其他线程获取成功

**共享与独占的区别在于：**

CHL队列中的节点的模式是独占还是共享，当一个线程成功修改了stat状态，表示获取了锁，如果线程所在的节点为SHARED，将开始一个读锁传递的过程，从头结点，向队列后续节点传递唤醒，直到队列结束或者遇到了EXCLUSIVE的节点，等待所有激活的读操作完成，然后进入到独享模式

公平与非公平的区别就在于线程第一次获取锁时，也就是执行修改stat操作时，是进队列还是直接修改状态，这是基本的工作机制

Java除了AQS还有其他的锁支持么？

在java中，synchronized关键字，是语言自带的，也叫内置锁，synchronized关键字，被synchronized修饰的方法或者代码块，在同一时间内，只允许一个线程执行，是独享锁

**JVM内存模型中主内存其实就是jvm中堆，工作内存就是线程的栈**

**jvm中有以下三种锁(由上到下越来越“重量级”)：偏向锁,轻量级锁,重量级锁**

**偏向锁**：适用于只有一个线程访问同步块场景。加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。

**轻量级锁**： 可以有多个线程交替进入临界区，在竞争不激烈的时候，稍微自旋等待一下就能获得锁。这也是CAS的实现方式。竞争的线程不会阻塞，提高了程序的响应速度。但是会消耗CPU

**重量级锁**：比如Synchronized它会利用java对象的监视器锁（monitor）实现，同一临界区内只允许一个线程操作。线程竞争不使用自旋，不会消耗CPU。

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 适用场景 |
| 偏向锁 | 加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。 | 如果线程间存在锁竞争，会带来**额外的锁撤销的消耗**。 | 适用于**只有一个线程访问**同步块场景。 |
| 轻量级锁 | 竞争的线程不会阻塞，提高了程序的响应速度。 | 如果始终得不到锁竞争的线程使用**自旋会消耗CPU**。 | 追求**响应时间**。  同步块执行速度非常快。 |
| 重量级锁 | 线程竞争不使用自旋，不会消耗CPU。 | **线程阻塞**，响应时间缓慢。 | 追求**吞吐量**。  同步块执行速度较长。 |

**java中锁的实现类有哪些？**

ReentrantLock,ReentrantReadWriteLock,StampedLock,AbstractQueuedSynchronizer

## volatile

**volatile的两大特性**：禁止重排序、内存可见性

volatile是一种类型修饰符，被volatile声明的变量表示随时可能发生变化，每次使用时，都必须从变量i对应的内存地址读取，编译器对操作该变量的代码不再进行优化。它是基于底层c++实现的。

**volatile的使用场景**

volatile一般用于状态标志，或者开销较低的“读－写锁”策略

**普通变量和volatile变量的异同：**

对于普通变量：读操作会优先读取工作内存的数据，如果工作内存中不存在，则从主内存中拷贝一份数据到工作内存中；写操作只会修改工作内存的副本数据，这种情况下，其它线程就无法读取变量的最新值

对于volatile变量（表层）：读操作时JMM会把工作内存中对应的值设为无效，要求线程从主内存中读取数据；写操作时JMM会把工作内存中对应的数据刷新到主内存中，这种情况下，其它线程就可以读取变量的最新值

**volatile变量的内存可见是基于内存屏障（Memory Barrier）实现的**

内存屏障：又称内存栅栏，是一个CPU命令，在程序运行时，为了提高执行性能，编译器和处理器会对指令进行重排序，JMM为了保证在不同的编译器和CPU上有相同的结果（可见性），通过插入特定类型的内存屏障来禁止特定类型的编译器重排序和处理重排序，插入一条内存屏障会告诉编译器和CPU：不管什么指令都不能和这条内存屏障指令重排序（禁止重排序）

**Volatile的原理:**

工作内存Work Memory其实就是对CPU寄存器和高速缓存的抽象，或者说每个线程的工作内存也可以简单理解为CPU寄存器和高速缓存。

那么当写两条线程Thread-A与Threab-B同时操作主存中的一个volatile变量i时，Thread-A写了变量i，那么：

Thread-A发出LOCK#指令

发出的LOCK#指令锁总线（或锁缓存行），同时让Thread-B高速缓存中的缓存行内容失效

Thread-A向主存回写最新修改的i

Thread-B读取变量i，那么：

Thread-B发现对应地址的缓存行被锁了，等待锁的释放，缓存一致性协议会保证它读取到最新的值

由此可以看出，volatile关键字的读和普通变量的读取相比基本没差别，差别主要还是在变量的写操作上。

**竞态条件**：当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在竞态条件。

## Synchronized

**Synchronized实现原理**

synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性

Synchronized是JVM实现的一种锁，其中锁的获取和释放分别是monitorenter和monitorexit指令，该锁在实现上分为了偏向锁、轻量级锁和重量级锁，其中偏向锁在1.6是默认开启的，轻量级锁在多线程竞争的情况下会膨胀成重量级锁，有关锁的数据都保存在对象头中

同步代码块是使用monitorenter和monitorexit指令实现的,同步方法（在这看不出来需要看JVM底层实现）依靠的是方法修饰符上的ACCSYNCHRONIZED实现。

可利用javap工具查看生成的class文件信息来分析Synchronize的实现。

synchronized用的锁是存在Java对象头里的，那么什么是Java对象头呢？Hotspot虚拟机的对象头主要包括两部分数据：Mark Word（标记字段）、Klass Pointer（类型指针）

原文：https://mp.weixin.qq.com/s/nU0M3QVE209S79bCrrwrSg

Synchronized的语义底层是通过一个monitor的对象来完成，其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。

## Atomic

**atomic的实现原理**

以AtomicInteger为例，他的值是存在一个使用volatile修饰的int里面，由此保证该值的可见性，再通过自旋 + CAS（乐观锁），比较更新value值，由此保证操作的原子性。如果更新失败，重新获取旧值，然后更新。

Java 8做出的改进和努力

在Java 8中引入了4个新的计数器类型，LongAdder、LongAccumulator、DoubleAdder、DoubleAccumulator。他们都是继承于Striped64。

**在LongAdder 与AtomicLong有什么区别？**

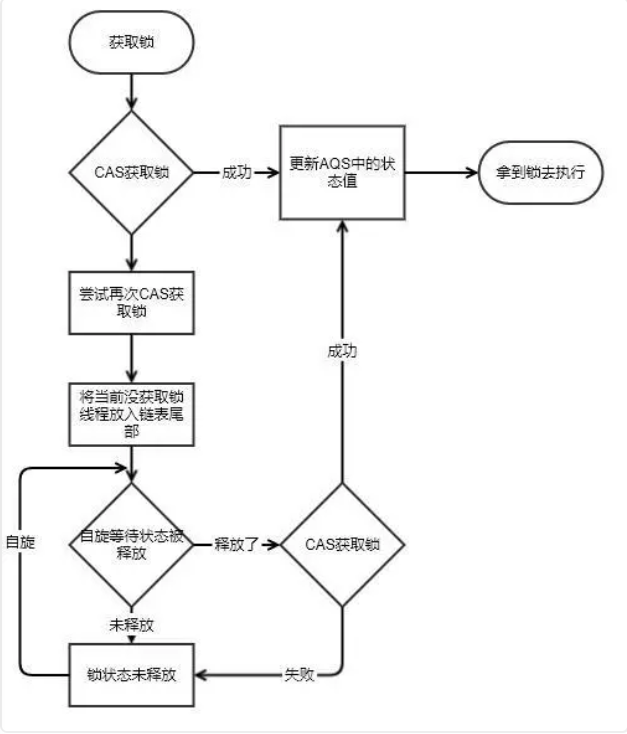
Atomic\*遇到的问题是，只能运用于低并发场景。因此LongAddr在这基础上引入了分段锁的概念。大概就是当竞争不激烈的时候，所有线程都是通过CAS对同一个变量（Base）进行修改，当竞争激烈的时候，会将根据当前线程哈希到对于Cell上进行修改（多段锁）。

## Lock，ReentrantLock

**ReentrantLock实现原理**

ReentrantLock是基于AQS实现的，AQS内部维系一个状态值，双向链表，当线程需要获取锁的时候，本质上是通过CAS来获取状态值修改，如果当场没获取到，会将该线程放在线程等待链表中。

**锁的存储结构就两个东西:"双向链表" + "int类型状态"**



**ReentrantLock的调用过程**

经过观察ReentrantLock把所有Lock接口的操作都委派到一个Sync类上，该类继承了AbstractQueuedSynchronizer

锁实现（加锁）

简单说来，AbstractQueuedSynchronizer会把所有的请求线程构成一个CLH队列，当一个线程执行完毕（lock.unlock()）时会激活自己的后继节点，但正在执行的线程并不在队列中，而那些等待执行的线程全部处于阻塞状态，经过调查线程的显式阻塞是通过调用LockSupport.park()完成，而LockSupport.park()则调用sun.misc.Unsafe.park()本地方法，再进一步，HotSpot在Linux中中通过调用pthread\_mutex\_lock函数把线程交给系统内核进行阻塞。与synchronized相同的是，这也是一个虚拟队列，不存在队列实例，仅存在节点之间的前后关系。令人疑惑的是为什么采用CLH队列呢？原生的CLH队列是用于自旋锁，但Doug Lea把其改造为阻塞锁。

当有线程竞争锁时，该线程会首先尝试获得锁，这对于那些已经在队列中排队的线程来说显得不公平，这也是非公平锁的由来，与synchronized实现类似，这样会极大提高吞吐量。

如果已经存在Running线程，则新的竞争线程会被追加到队尾，具体是采用基于CAS的Lock-Free算法，因为线程并发对Tail调用CAS可能会导致其他线程CAS失败，解决办法是循环CAS直至成功。

## CAS原理及缺点

CAS是通过unsafe类的compareAndSwap方法实现的， 它将内存位置的内容与给定值进行比较，只有在相同的情况下，将该内存位置的内容修改为新的给定值。

compareAndSwap方法的第一个参数是要修改的对象，第二个参数是对象中要修改变量的偏移量，第三个参数是修改之前的值，第四个参数是预想修改后的值。通过将偏移量与修改之前的值做比较，则将将偏移量修改为预想修改的值，并返回true.

当然CAS一定要volatile变量配合

unsafe.compareAndSwapInt(this, valueOffset, expect, update);

compareAndSet这个方法主要调用unsafe.compareAndSwapInt这个方法，这个方法有四个参数，其中第一个参数为需要改变的对象，第二个为偏移量(即之前求出来的valueOffset的值)，第三个参数为期待的值，第四个为更新后的值。整个方法的作用即为若调用该方法时，value的值与expect这个值相等，那么则将value修改为update这个值，并返回一个true，如果调用该方法时，value的值与expect这个值不相等，那么不做任何操作，并范围一个false。

**CAS的缺点：**

1.CPU开销较大：在并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，循环往复，会给CPU带来很大的压力。

2.不能保证代码块的原子性：CAS机制所保证的只是一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证3个变量共同进行原子性的更新，就不得不使用Synchronized了。

3.ABA问题：这是CAS机制最大的问题所在。

如线程1从内存X中取出A，这时候另一个线程2也从内存X中取出A，并且线程2进行了一些操作将内存X中的值变成了B，然后线程2又将内存X中的数据变成A，这时候线程1进行CAS操作发现内存X中仍然是A，然后线程1操作成功。虽然线程1的CAS操作成功，但是整个过程就是有问题的。比如链表的头在变化了两次后恢复了原值，但是不代表链表就没有变化。

所以JAVA中提供了AtomicStampedReference/AtomicMarkableReference来处理会发生ABA问题的场景，主要是在对象中额外再增加一个标记来标识对象是否有过变更。

## 死锁

**死锁产生的四个条件：**

1）互斥。共享资源同时只能被一个线程访问。

2）占有且等待。线程T1在取得共享资源A的时候，请求等待资源B的时候并不释放资源A。

3）不可抢占。其他线程不能强行抢占线程的资源。

4）循环等待条件。线程T1在持有资源A1，同时在请求等待获取资源B，线程T2在持有资源B,然后在请求等待线程T1的持有资源，形成了交叉闭环申请。

处理死锁的方法可有以下4种：

1）死锁预防。由于互斥这个条件无法避免。操作系统本身所具有的特点。可对死锁产生的其他三个条件进行破坏。首先条件2，占有且等待，可以一次性申请所有的资源，可以破坏掉占有且等待。条件三不可抢占。当线程去请求其他资源时，如果获取不到锁，可以主动释放自己的锁，这样不可抢占的条件也被破坏掉了。条件四循环等待条件。可以对申请的资源进行编号，按序访问，这样线性的去申请资源，则不会造成交叉循环。

2）死锁避免。就不用去破坏形成死锁的其他条件。在资源动态分配的过程中。用某种方法判断防止进入不安全状态。从而避免发生死锁。可以使用银行家算法。死锁避免的算法会导致系统开销的增加。

3）死锁检测。死锁预防和死锁检测都是死锁发生之前的预防策略。死锁检测是通过系统设置的检测机构及时的判断当前系统是否处于死锁状态，并精确的确定当前死锁相关的进程和资源，执行死锁解除策略

4）死锁解除。这是与死锁检测结合使用的。它使用的方式是剥夺。就是讲进程所占有的资源强行收回，分配给其他进程。

避免死锁最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁。

**死锁排查**

1 、 使用top命令查看cpu占用资源较高的PID

2、 通过jps 找到当前用户下的java程序PID：执行 jps -l 能够打印出所有的应用的PID，找到有一个PID和这个cpu使用100%一样的ID！！就知道是哪一个服务了。

3、 使用 pidstat -p 1 3 -u -t

4 、 找到cpu占用较高的线程TID

5、 将TID转换为十六进制的表示方式

6、 通过jstack -l 输出当前进程的线程信息

7、 查找 TID对应的线程(输出的线程id为十六进制)，找到对应的代码

## 其它

**happen-befor 原则，Volatile 关键字（线程的可见性），内存屏障**

（1）happen-befor原则定义了内存模型执行过程中的定律，就像1+1 = 2，不可能被打破的jvm的运行机制都依赖于这个原则，是jvm的“宪法”

（2）volatile关键字：提供内存可见性和禁止内存重排序

**CAS与Synchronized的关系**

Synchronized属于悲观锁，Synchronized关键字会让没有得到锁资源的线程进入BLOCKED状态，而后在争夺到锁资源后恢复为RUNNABLE状态，这个过程中涉及到操作系统用户模式和内核模式的转换，代价比较高。

CAS属于乐观锁，它会让当前线程处于忙循环，一直去争取获取锁。

**什么时候使用CAS锁**

atomic类，以及lock类的底层实现都会设计到CAS锁。在Java1.6以上的版本,synchronized转变为重量级锁之前，也会采用CAS机制。

**Lock与synchronized的区别**

          1. Lock的加锁和解锁都是由java代码配合native方法（调用操作系统的相关方法）实现的，而synchronize的加锁和解锁的过程是由JVM管理的

          2. 当一个线程使用synchronize获取锁时，若锁被其他线程占用着，那么当前只能被阻塞，直到成功获取锁。而Lock则提供超时锁和可中断等更加灵活的方式，在未能获取锁的条件下提供一种退出的机制。

          3. 一个锁内部可以有多个Condition实例，即有多路条件队列，而synchronize只有一路条件队列；同样Condition也提供灵活的阻塞方式，在未获得通知之前可以通过中断线程以    及设置等待时限等方式退出条件队列。

         4. synchronize对线程的同步仅提供独占模式，而Lock即可以提供独占模式，也可以提供共享模式

AbstractQueuedSynchronizer通过构造一个基于阻塞的CLH队列容纳所有的阻塞线程，而对该队列的操作均通过Lock-Free（CAS）操作，但对已经获得锁的线程而言，ReentrantLock实现了偏向锁的功能。

synchronized的底层也是一个基于CAS操作的等待队列，但JVM实现的更精细，把等待队列分为ContentionList和EntryList，目的是为了降低线程的出列速度；当然也实现了偏向锁，从数据结构来说二者设计没有本质区别。但synchronized还实现了自旋锁，并针对不同的系统和硬件体系进行了优化，而Lock则完全依靠系统阻塞挂起等待线程。

当然Lock比synchronized更适合在应用层扩展，可以继承AbstractQueuedSynchronizer定义各种实现，比如实现读写锁（ReadWriteLock），公平或不公平锁；同时，Lock对应的Condition也比wait/notify要方便的多、灵活的多。

**synchronized和ReentrantLock的区别**

synchronized是和if、else、for、while一样的关键字，ReentrantLock是类，这是二者的本质区别。

ReentrantLock比synchronized的扩展性体现在几点上：

（1）ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁

（2）ReentrantLock可以获取各种锁的信息

（3）ReentrantLock可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的。ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的应该是对象头中mark word，这点我不能确定。

# 多线程

## 线程，进程

线程是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。进程之间是隔离的，进程是拥有资源的基本单位， 线程是CPU调度的基本单位。

**Java中线程具有五种状态：**

初始化；可运行；运行中；阻塞；销毁



但是，线程不同状态之间的转化是谁来实现的呢？是JVM吗？

并不是。JVM需要通过操作系统内核中的TCB（Thread Control Block）模块来改变线程的状态，这一过程需要耗费一定的CPU资源。

**一个线程会因为以下原因而放弃CPU：**

时间片用完了，java虚拟机让当前线程暂时放弃CPU,转到就绪状态,使其它线程获得运行机会。

当前线程因为某些原因而进入阻塞状态

线程结束运行

**线程在以下情况会停止：**

run方法正常执行完毕

run方法执行过程中抛出一个未捕获的异常

调用stop方法(不推荐使用)

**进程的停止，当一个进程中所有的前台线程停止后，该进程结束。**

**什么时候会出现僵死进程**

线程无法结束的时候会僵死进程，一般是出现死锁。

**什么是协程**

协程，英文Coroutines，是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程

最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。

**一个线程OOM，进程里其他线程还能运行么？**

答案是还能运行

在多线程环境下，每个线程拥有一个栈和一个程序计数器。栈和程序计数器用来保存线程的执行历史和线程的执行状态，是线程私有的资源。其他的资源（比如堆、地址空间、全局变量）是由同一个进程内的多个线程共享。

当一个线程抛出OOM异常后，即使是堆内存溢出，它所占据的内存资源会全部被释放掉，从而不会影响其他线程的运行！如果是栈溢出，结论也是一样。

JvisualVM看堆的变化

**多线程有什么用？**

（1）发挥多核CPU的优势

（2）防止阻塞

（3）便于建模：假设有一个大的任务A，单线程编程，那么就要考虑很多，建立整个程序模型比较麻烦。但是如果把这个大的任务A分解成几个小任务，任务B、任务C、任务D，分别建立程序模型，并通过多线程分别运行这几个任务，那就简单很多了。

**start()方法和run()方法的区别？**

只有调用了start()方法，才会表现出多线程的特性，不同线程的run()方法里面的代码交替执行。如果只是调用run()方法，那么代码还是同步执行的，必须等待一个线程的run()方法里面的代码全部执行完毕之后，另外一个线程才可以执行其run()方法里面的代码。

**Runnable接口和Callable接口的区别？**

Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

## 线程池

项目中的应用案例：

案例：kafka消费时使用线程池接收数据。

对账的时候使用线程池操作。

**线程池使用：**

**Java中线程池的实现方式**

单个线程: Executors.newSingleThreadExecutor();

缓存线程: Executors.newCachedThreadPool();

固定线程Executors.newFixedThreadPool(2);

定时线程: Executors.newScheduledThreadPool(3);

**ThreadPoolExecutor**

ThreadPoolExecutor 继承 AbstractExecutorService；AbstractExecutorService 实现 ExecutorService， ExecutorService 继承 Executor

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) {...}

corePoolSize : 核心线程数的大小

maximumPoolSize : 线程池中允许的最大线程数

keepAliveTime : 空闲线程允许的最大的存活时间

unit : 存活时间的单位

workQueue : 阻塞任务队列

threadFactory : 线程工厂用来创建线程

handler : 拒绝策略，针对当队列满了时新来任务的处理方式

**线程池的执行过程：**

在刚刚创建线程池的时候，内部线程的数量是 0，当首个任务进行添加的时候，会根据参数的配置进行线程的创建，并随着任务数的增加，会逐渐创建新的线程直到线程数达到核心线程的大小。这时，如果再添加任务会将任务放置到阻塞队列当中,如果队列装不下了这时会判断当前线程的数量是否达到了最大线程数，如果未达到，新添加的任务会由最大重新创建线程并立马执行。如果队列满了，而且已经达到最大线程数了，这时再添加任何就会由拒绝策略来处理，默认的拒绝策略是抛出异常。核心线程数不会被关闭，会一直等待任务到来。

**使用线程池的优点是什么**

虽然线程是个轻量级的东西， 但是对于互联网应用来说，如果每个用户的请求都创建一个线程，那会非常得多，服务器也是难于承受， 再说了，众多的线程去竞争CPU，不断切换，也会让CPU调度不堪重负，很多线程将不得不等待。

线程可以预先创建，当线程池的线程刚创建时，让他们进入阻塞状态：等待某个任务的到来。 如果任务来了，那就好办，唤醒其中一个线程，让它拿到任务去执行即可。

**线程池原理**

原文：<https://mp.weixin.qq.com/s/LExpStGw7XXp0M1E_T0yKA>

https://mp.weixin.qq.com/s/DJPuYVhRGRXkkUGMuDdrSw

根据源码可以发现整个线程池大致分为 3 个部分，1. 是创建 worker 线程，2. 添加任务到 workQueue; 3.worker 线程执行具体任务

线程池每个线程会包装成worker，用来循环执行take任务。

第一步是生产者，也就是任务提供者他执行了一个 execute() 方法，本质上就是往这个内部队列里放了一个任务。

之前已经创建好了的 Worker 线程会执行一个 while 循环 ---> 不停的从这个 内部队列里获取任务。(这一步是竞争的关系，都会抢着从队列里获取任务，由这个队列内部实现了线程安全。)

获取得到一个任务后，其实也就是拿到了一个 Runnable 对象(也就是 execute(Runnabletask) 这里所提交的任务)，接着执行这个 Runnable 的 run() 方法，而不是 start()，为什么线程池在调度的时候执行的是 Runnable 的 run() 方法，而不是 start() 方法呢？线程池这个场景中却恰好就是要利用它只是一个普通方法调用。如果我们往一个核心、最大线程数为 2 的线程池里丢了 1000 个任务，那么它会额外的创建 1000 个线程，同时每个任务都是异步执行的，一下子就执行完毕了。

所谓线程池本质是一个hashSet。多余的任务会放在阻塞队列中。

只有当阻塞队列满了后，才会触发非核心线程的创建。所以非核心线程只是临时过来打杂的。直到空闲了，然后自己关闭了。

线程池提供了两个钩子（beforeExecute，afterExecute）给我们，我们继承线程池，在执行任务前后做一些事情。

线程池原理关键技术：锁（lock,cas）、阻塞队列、hashSet（资源池）

**当线程池在执行一个任务时抛出未捕获异常，这个任务还会执行吗？**



右图放大后的那一块，也就是内部队列并没有其他线程往里边丢任务执行 execute() 方法。

而一旦发生未捕获的异常后， Worker1 被回收，顺带的它所调度的线程 task1（这个task1 也就是在执行一个 while 循环消费左图中的那个队列） 也会被回收掉。

新创建的 Worker2 会取代 Worker1 继续执行 while 循环从内部队列里获取任务，但此时这个队列就一直会是空的，所以也就是处于 Waiting 状态。

原文：<https://mp.weixin.qq.com/s/SpxYhfExN-GwTYflkhvKRg>

因此：这个任务不会再继续执行了

线程池在调度的时候执行的是 Runnable 的 run() 方法，而不是 start() 方法呢？

线程池执行某个任务后抛出了未捕获异常，这个任务就不会再执行了。

**线程池遇到未捕获异常会怎样？**

Java中线程执行的任务接口java.lang.Runnable如果 run() 方法中抛出了RuntimeExceptio将会怎么处理了？通常java.lang.Thread对象运行设置一个默认的异常处理方法：java.lang.Thread.setDefaultUncaughtExceptionHandler(UncaughtExceptionHandler)而这个默认的静态全局的异常捕获方法是直接输出异常堆栈。当然，我们可以覆盖此默认实现，只需要一个自定义的java.lang.Thread.UncaughtExceptionHandler接口实现即可。

而在线程池中却比较特殊。默认情况下，线程池 java.util.concurrent.ThreadPoolExecutor 会Catch住所有异常， 当任务执行完成(java.util.concurrent.ExecutorService.submit(Callable))获取其结果 时(java.util.concurrent.Future.get())会抛出此RuntimeException。也就是说当我们想线程池 ThreadPoolExecutor(java.util.concurrent.ExecutorService)提交任务时， 如果不理会任务结果（Feture.get()），那么此异常将被线程池吃掉。可以通过继承ThreadPoolExecutor并覆盖ThreadPoolExecutor.afterExecute 方法，我们才能捕获到任务的异常(RuntimeException)。

线程池有未抓取异常时不会导致后续任务中断，但是没有异常信息的输出，难以定位问题，对此，有三种方案：

）自己封装一个RunnableWrapper、CallableWrapper，将传入的任务包裹其中，在run方法中try-catch，并统一对未抓取异常进行异常处理/输出；

）继承ThreadPoolExecutor的方式来创建线程池，重写其afterExecute(Thread, Throwable)方法，判断第二个参数非空时获得其异常信息，统一进行处理；（注意：仅对 executor.execute(..) 启动的任务有效，executor.submit(..)的无效）

）创建线程池时，传入自定义的ThreadFactory，在其创建线程时，使用setUncaughtExceptionHandler(..)设置每个线程的未抓取异常处理句柄，统一进行处理；（注意：仅对 executor.execute(..) 启动的任务有效，executor.submit(..)的无效）

**executor与submit的区别**

1、接收的参数不一样

2、submit有返回值，而execute没有

3、submit方便Exception处理：意思就是如果你在你的task里会抛出checked或者unchecked exception，而你又希望外面的调用者能够感知这些exception并做出及时的处理，那么就需要用到submit，通过捕获Future.get抛出的异常。

**executor与submit的区别**

1、接收的参数不一样

2、submit有返回值，而execute没有

3、submit方便Exception处理：意思就是如果你在你的task里会抛出checked或者unchecked exception，而你又希望外面的调用者能够感知这些exception并做出及时的处理，那么就需要用到submit，通过捕获Future.get抛出的异常。

**什么是线程安全**

如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是线程安全的。

这个问题有值得一提的地方，就是线程安全也是有几个级别的：

（1）不可变

像String、Integer、Long这些，都是final类型的类，任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用

（2）绝对线程安全

不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java中也有，比方说CopyOnWriteArrayList、CopyOnWriteArraySet

（3）相对线程安全

相对线程安全也就是我们通常意义上所说的线程安全，像Vector这种，add、remove方法都是原子操作，不会被打断，但也仅限于此，如果有个线程在遍历某个Vector、有个线程同时在add这个Vector，99%的情况下都会出现ConcurrentModificationException，也就是fail-fast机制。

（4）线程非安全

这个就没什么好说的了，ArrayList、LinkedList、HashMap等都是线程非安全的类

**线程池配置**

高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

（1）高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换

（2）并发不高、任务执行时间长的业务要区分开看：

a）假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务

b）假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换

（3）并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考其他有关线程池的文章。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

## 线程间通信

**线程间有哪些通信方式？**

线程间的通信模式有多种,可以是等待通知模式，共享内存模式，管道通信模式

等待通知模式可以使用wait() 和通知 notify() 方法来进行线程间通讯，也可以使用join()方法，在 join 线程完成后会调用 notifyAll() 方法，是在 JVM 实现中调用。。

共享内存模式可以使用volatile共享内存，也可以使用类似于CountDownLatch，CyclicBarrier，Semaphore这样的并发工具进行通信。

管道通信模式可以使用PipedWriter，PipedReader，writer.connect(reader)

也可以使用阻塞队列BlockingQueue来共享数据

**wait,notify为什么必须同步代码块执行的，否则会抛出IllegalMonitorStateException。**

因为使用wait,notify时，当前线程必须拥有对象监视器。使用wait方法时，线程释放对象监视器的所有权，等待直到另一个线程通过调用notify或notifyAll来通知等待对象监视器的线程们并唤醒。在某一时刻，只有一个线程占有对象监视器。该方法必须同步执行的，否则会抛出IllegalMonitorStateException。

其次，由于CPU执行是随机的，所以我们必须对流程进行管理，也就是同步，通过在同步块中并结合 wait 和 notify 方法，我们可以手动对线程的执行顺序进行调整。

比如：假设有2个线程，分别是生产者和消费者，他们有各自的任务。

1.1生产者检查条件（如缓存满了）-> 1.2生产者必须等待

2.1消费者消费了一个单位的缓存 -> 2.2重新设置了条件（如缓存没满） -> 2.3调用notifyAll()唤醒生产者

我们希望的顺序是： 1.1->1.2->2.1->2.2->2.3

但是由于CPU执行是随机的，可能会导致 2.3 先执行，1.2 后执行，这样就会导致生产者永远也醒不过来了！

**wait,notify的原理**

不知道，因为看源码都是使用了native方法，没去看了。

**Semaphore有什么作用？**

Semaphore就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore有一个构造函数，可以传入一个int型整数n，表示某段代码最多只有n个线程可以访问，如果超出了n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果Semaphore构造函数中传入的int型整数n=1，相当于变成了一个synchronized了。

**CyclicBarrier和CountDownLatch的区别？**

两个看上去有点像的类，都在java.util.concurrent下，都可以用来表示代码运行到某个点上，二者的区别在于：

（1）CyclicBarrier的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch则不是，某线程运行到某个点上之后，只是给某个数值-1而已，该线程继续运行

（2）CyclicBarrier只能唤起一个任务，CountDownLatch可以唤起多个任务

（3）CyclicBarrier可重用，CountDownLatch不可重用，计数值为0该CountDownLatch就不可再用了

**Volatile的作用**

使用Volatile修饰的变量具有可见性，在虚拟机内存中有主内存和工作内存的概念，每一条线程对应一个工作内存，一般线程需要使用某个变量时会先从主内存中复制该变量到工作内存中，操作完成后再写回主内存中，使用volatile修饰后，当一个线程修改了变量的值，新的值会立刻同步到主内存当中。而其他线程读取这个变量的时候，也会从主内存中拉取最新的变量值。使用volatile还能防止指令重排。

**CountDownLatch原理**

countDownLatch.await()方法时会当前线程就会进入了一个死循环当中，在这个死循环里面，会不断的进行判断，通过调用tryAcquireShared方法，不断判断我们上面说的那个计数器，看看它的值是否为0了，如果是为0的话，tryAcquireShared就会返回1，然后跳出循环。需要注意的是，说是在不停的循环，其实也并非在不停的执行for循环里面的内容，因为在后面调用parkAndCheckInterrupt（）方法时，在这个方法里面是会调用 LockSupport.park(this);，来禁用当前线程的。

**？CyclicBarrier原理**

其核心代码是 ReentrantLock 以及 Condition 的共享唤醒线程，多个线程竞争锁，保证计数器parties为原子操作，然后当parties执行为0时候，执行方法，此时所有阻塞的线程继续执行

**Semaphore有什么作用**

Semaphore就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore有一个构造函数，可以传入一个int型整数n，表示某段代码最多只有n个线程可以访问，如果超出了n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果Semaphore构造函数中传入的int型整数n=1，相当于变成了一个synchronized了。

**？Semaphore原理**

Semaphore有两种模式，公平模式和非公平模式。公平模式就是调用acquire的顺序就是获取许可证的顺序，遵循FIFO；而非公平模式是抢占式的，也就是有可能一个新的获取线程恰好在一个许可证释放时得到了这个许可证，而前面还有等待的线程。其内部是基于AQS的共享模式，AQS的状态表示许可证的数量，在许可证数量不够时，线程将会被挂起；而一旦有一个线程释放一个资源，那么就有可能重新唤醒等待队列中的线程继续执行。

## ？ThreadLocal

ThreadLocal什么时候会出现OOM的情况？为什么？

线程池的一个线程使用完ThreadLocal对象之后，再也不用，由于线程池中的线程不会退出，线程池中的线程的存在，同时ThreadLocal变量也会存在，占用内存！造成OOM溢出！

原文：<https://blog.csdn.net/GoGleTech/article/details/78318712>

ThreadLocal终极篇 https://mp.weixin.qq.com/s/\_gQ0uJTTk4Ws8fDojK61og

## 其它

**Java中如何获取到线程dump文件**

死循环、死锁、阻塞、页面打开慢等问题，打线程dump是最好的解决问题的途径。所谓线程dump也就是线程堆栈，获取到线程堆栈有两步：

（1）获取到线程的pid，可以通过使用jps命令，在Linux环境下还可以使用ps -ef | grep java

（2）打印线程堆栈，可以通过使用jstack pid命令，在Linux环境下还可以使用kill -3 pid

另外提一点，Thread类提供了一个getStackTrace()方法也可以用于获取线程堆栈。这是一个实例方法，因此此方法是和具体线程实例绑定的，每次获取获取到的是具体某个线程当前运行的堆栈，

**一个线程如果出现了运行时异常会怎么样**

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：如果这个线程持有某个某个对象的监视器，那么这个对象监视器会被立即释放

**sleep方法和wait方法有什么区别**

这个问题常问，sleep方法和wait方法都可以用来放弃CPU一定的时间，不同点在于如果线程持有某个对象的监视器，sleep方法不会放弃这个对象的监视器，wait方法会放弃这个对象的监视器

**生产者消费者模型的作用是什么**

（1）通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率，这是生产者消费者模型最重要的作用

（2）解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要收到相互的制约

**ThreadLocal有什么用**

简单说ThreadLocal就是一种以空间换时间的做法，在每个Thread里面维护了一个以开地址法实现的ThreadLocal.ThreadLocalMap，把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了

**为什么wait()方法和notify()/notifyAll()方法要在同步块中被调用**

这是JDK强制的，wait()方法和notify()/notifyAll()方法在调用前都必须先获得对象的锁

wait()方法和notify()/notifyAll()方法在放弃对象监视器时有什么区别？

wait()方法和notify()/notifyAll()方法在放弃对象监视器的时候的区别在于：wait()方法立即释放对象监视器，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器。

**为什么要使用线程池**

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目。

**怎么检测一个线程是否持有对象监视器**

Thread类提供了一个holdsLock(Object obj)方法，当且仅当对象obj的监视器被某条线程持有的时候才会返回true，注意这是一个static方法，这意味着"某条线程"指的是当前线程。

**FutureTask是什么**

FutureTask表示一个异步运算的任务。FutureTask里面可以传入一个Callable的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于FutureTask也是Runnable接口的实现类，所以FutureTask也可以放入线程池中。

**Linux环境下如何查找哪个线程使用CPU最长**

（1）获取项目的pid，jps或者ps -ef | grep java，这个前面有讲过

（2）top -H -p pid，顺序不能改变

这样就可以打印出当前的项目，每条线程占用CPU时间的百分比。注意这里打出的是LWP，也就是操作系统原生线程的线程号

使用"top -H -p pid"+"jps pid"可以很容易地找到某条占用CPU高的线程的线程堆栈，从而定位占用CPU高的原因，一般是因为不当的代码操作导致了死循环。

最后提一点，"top -H -p pid"打出来的LWP是十进制的，"jps pid"打出来的本地线程号是十六进制的，转换一下，就能定位到占用CPU高的线程的当前线程堆栈了。

**Java中用到的线程调度算法是什么**

抢占式。一个线程用完CPU之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

**Thread.sleep(0)的作用是什么**

为了让某些优先级比较低的线程也能获取到CPU控制权，可以使用Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡CPU控制权的一种操作。

**Java中interrupted 和 isInterruptedd方法的区别？**

interrupted() 和 isInterrupted()的主要区别是前者会将中断状态清除而后者不会。Java多线程的中断机制是用内部标识来实现的，调用Thread.interrupt()来中断一个线程就会设置中断标识为true。**当中断线程调用静态方式Thread.interrupted()来检查中断状态时，中断状态会被清零。而非静态方法isInterrupted()用来查询其它线程的中断状态且不会改变中断状态标识。**简单的说就是任何抛出InterruptedException异常的方法都会将中断状态清零。无论如何，一个线程的中断状态有有可能被其它线程调用中断来改变

**为什么wait和notify方法要在同步块中调用？**

一个原因是为了避免wait和notify之间产生竞态条件。还有一个原因就是wait,notify都要获取对象的锁

**如果你提交任务时，线程池队列已满。会时发会生什么？**

这个问题问得很狡猾，许多程序员会认为该任务会阻塞直到线程池队列有空位。事实上如果一个任务不能被调度执行那么ThreadPoolExecutor’s submit()方法将会抛出一个RejectedExecutionException异常。

**Java线程池中submit() 和 execute()方法有什么区别？**

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中, 而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

当我们考察异常处理的时候，又会发现另外一个不同。当你使用execute提交的任务抛出异常时，此异常将会交由未捕捉异常处理过程来处理(uncaught exception handler)，当你没有显式指定一个异常处理器的话，默认情况下仅仅会通过System.err打印出错误堆栈。当你用submit来提交一个任务的时候，这个任务一旦抛出异常(无论是否是运行时异常)，那这个异常是任务返回对象的一部分。对这样一种情形，当你调用Future.get()方法的时候，这个方法会重新抛出这个异常，并且会使用ExecutionException进行包装。

## java程序，主进程需要等待多个子进程结束之后再执行后续的代码，有哪些方案可以实现

**1.join方法**

比如在线程B中调用了线程A的join()方法，直到线程A执行完毕后，才会继续执行线程B。

public static void main(String[] args) throws InterruptedException {

Vector<Thread> vector = new Vector<>();

for(int i=0;i<5;i++) {

Thread childThread= new Thread(new Runnable() {

@Override

public void run() {

// TODO Auto-generated method stub

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("子线程被执行");

}

});

vector.add(childThread);

childThread.start();

}

for(Thread thread : vector) {

thread.join();

}

System.out.println("主线程被执行");

}

**2.等待多线程完成的CountDownLatch**

public static void main(String[] args) throws InterruptedException {

final CountDownLatch latch = new CountDownLatch(5);

for(int i=0;i<5;i++) {

Thread childThread= new Thread(new Runnable() {

@Override

public void run() {

// TODO Auto-generated method stub

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("子线程被执行");

latch.countDown();

}

});

childThread.start();

}

latch.await();//阻塞当前线程直到latch中的值

System.out.println("主线程被执行");

}

**3.同步屏障CyclicBarrier**

public static void main(String[] args) throws InterruptedException, BrokenBarrierException {

final CyclicBarrier barrier = new CyclicBarrier(5);

for(int i=0;i<4;i++) {

Thread childThread= new Thread(new Runnable() {

@Override

public void run() {

// TODO Auto-generated method stub

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("子线程被执行");

try {

barrier.await();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (BrokenBarrierException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

});

childThread.start();

}

barrier.await();//阻塞当前线程直到latch中的值

System.out.println("主线程被执行");

}

CylicBarrier是控制一组线程的同步，初始化的参数：5的含义是包括主线程在内有5个线程，所以只能有四个子线程，这与CountDownLatch是不一样的。

countDownLatch和cyclicBarrier有什么区别呢，他们的区别：countDownLatch只能使用一次，而CyclicBarrier方法可以使用reset()方法重置，所以CyclicBarrier方法可以能处理更为复杂的业务场景。cyclicBarrier只会在最后一个线程结束时才会给主线程发送信号，而countDownLatch会在每个线程结束时都发送一个信号

5.FutureTast可用于闭锁，类似于CountDownLatch的作用

public class Test5 {

public static void main(String[] args) {

MyThread td = new MyThread();

//1.执行 Callable 方式，需要 FutureTask 实现类的支持，用于接收运算结果。

FutureTask<Integer> result1 = new FutureTask<>(td);

new Thread(result1).start();

FutureTask<Integer> result2 = new FutureTask<>(td);

new Thread(result2).start();

FutureTask<Integer> result3 = new FutureTask<>(td);

new Thread(result3).start();

Integer sum;

try {

sum = result1.get();

sum = result2.get();

sum = result3.get();

//这里获取三个sum值只是为了同步，并没有实际意义

System.out.println(sum);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (ExecutionException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} //FutureTask 可用于 闭锁 类似于CountDownLatch的作用，在所有的线程没有执行完成之后这里是不会执行的

System.out.println("主线程被执行");

}

}

class MyThread implements Callable<Integer> {

@Override

public Integer call() throws Exception {

int sum = 0;

Thread.sleep(1000);

for (int i = 0; i <= 10; i++) {

sum += i;

}

System.out.println("子线程被执行");

return sum;

}

}

**6.使用callable+future**

Callable+Future最终也是以Callable+FutureTask的形式实现的。 在这种方式中调用了： Future future = executor.submit(task);

public class Test6 {

public static void main(String[] args) throws InterruptedException, ExecutionException {

ExecutorService executor = Executors.newCachedThreadPool();

Task task = new Task();

Future<Integer> future1 = executor.submit(task);

Future<Integer> future2 = executor.submit(task);

//获取线程执行结果，用来同步

Integer result1 = future1.get();

Integer result2 = future2.get();

System.out.println("主线程执行");

executor.shutdown();

}

}

class Task implements Callable<Integer>{

@Override public Integer call() throws Exception {

int sum = 0;

//do something;

System.out.println("子线程被执行");

return sum;

}

}

1）CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同：CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；而CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；另外，CountDownLatch是不能够重用的，而CyclicBarrier是可以重用的。

2）Semaphore其实和锁有点类似，它一般用于控制对某组资源的访问权限。CountDownLatch类实际上是使用计数器的方式去控制的，不难想象当我们初始化CountDownLatch的时候传入了一个int变量这个时候在类的内部初始化一个int的变量，每当我们调用countDownt()方法的时候就使得这个变量的值减1，而对于await()方法则去判断这个int的变量的值是否为0，是则表示所有的操作都已经完成，否则继续等待。

实际上如果了解AQS的话应该很容易想到可以使用AQS的共享式获取同步状态的方式来完成这个功能。而CountDownLatch实际上也就是这么做的。

# 设计者模式

GOF23种设计模式精解

创建型

1. Factory Method（工厂方法）

2. Abstract Factory（抽象工厂）

3. Builder（建造者）

4. Prototype（原型）

5. Singleton（单例）

结构型

6. Adapter Class/Object（适配器）

7. Bridge（桥接）

8. Composite（组合）

9. Decorator（装饰）

10. Facade（外观）

11. Flyweight（享元）

12. Proxy（代理）

行为型

13. Interpreter（解释器）

14. Template Method（模板方法）

15. Chain of Responsibility（责任链）

16. Command（命令）

17. Iterator（迭代器）

18. Mediator（中介者）

19. Memento（备忘录）

20. Observer（观察者）

21. State（状态）

22. Strategy（策略）

23. Visitor（访问者）

## 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。

**结构型模式**



## 适配器模

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

**类的适配器模式**，核心思想就是：有一个Source类，拥有一个方法，待适配，目标接口时Targetable，通过Adapter类，将Source的功能扩展到Targetable里

**对象的适配器模式**：基本思路和类的适配器模式相同，只是将Adapter类作修改，这次不继承Source类，而是持有Source类的实例，以达到解决兼容性的问题。

**接口的适配器**是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。

## 外观模式（Facade）

外观模式是为了解决类与类之家的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度，该模式中没有涉及到接口。（Computer类，那么，CPU、Memory、Disk他们之间将会相互持有实例，产生关系，这样会造成严重的依赖）

## 桥接模式（Bridge）

桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。桥接的用意是：将抽象化与实现化解耦，使得二者可以独立变化，像我们常用的JDBC桥DriverManager一样

## 组合模式（Composite）

组合模式有时又叫部分-整体模式在处理类似树形结构的问题时比较方便，

使用场景：将多个对象组合在一起进行操作，常用于表示树形结构中，例如二叉树，数等。

## 享元模式（Flyweight）

享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。

行为型模式，共11种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。



## 策略模式（strategy）

策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。策略模式多用在算法决策系统中。

比如：A是一个接口，提供calculate(String exp)方法，B是一个抽象类，提供split(String exp,String opt)方法。C类继承B实现A，在C的calculate(String exp)方法中调用B的split方法，并做其它操作。

## 模板方法模式（Template Method）

解释一下模板方法模式，就是指：一个抽象类中，有一个主方法，再定义1...n个方法，可以是抽象的，也可以是实际的方法，定义一个类，继承该抽象类，重写抽象方法，通过调用抽象类，实现对子类的调用，

## 观察者模式（Observer）

当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。

## 迭代子模式（Iterator）

顾名思义，迭代器模式就是顺序访问聚集中的对象，一是需要遍历的对象，即聚集对象，二是迭代器对象，用于对聚集对象进行遍历访问。

## 责任链模式（Chain of Responsibility）

接下来我们将要谈谈责任链模式，有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传递，直到某一对象决定处理该请求。

## 命令模式（Command）

命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开，熟悉Struts的同学应该知道，Struts其实就是一种将请求和呈现分离的技术，其中必然涉及命令模式的思想！

## 备忘录模式（Memento）

主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象，个人觉得叫备份模式更形象些，通俗的讲下：假设有原始类A，A中有各种属性，A可以决定需要备份的属性，备忘录类B是用来存储A的一些内部状态，类C呢，就是一个用来存储备忘录的，且只能存储，不能修改等操作。

## 状态模式（State）

核心思想就是：当对象的状态改变时，同时改变其行为，很好理解！状态模式就两点：1、可以通过改变状态来获得不同的行为。2、你的好友能同时看到你的变化。

## 访问者模式（Visitor）

访问者模式就是一种分离对象数据结构与行为的方法，通过这种分离，可达到为一个被访问者动态添加新的操作而无需做其它的修改的效果。

## 中介者模式（Mediator）

中介者模式也是用来降低类类之间的耦合的，因为如果类类之间有依赖关系的话，不利于功能的拓展和维护，因为只要修改一个对象，其它关联的对象都得进行修改。

## 解释器模式（Interpreter）

基本就这样，解释器模式用来做各种各样的解释器，如正则表达式等的解释器等等！

# 开放题

## 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。