前言

我的一个学弟小牛（公众号：**后端技术小牛说**）整理了一份 Java 八股文背诵版（113 道高频面试题附答案），我在此基础上做了部分优化和调整，现在分享给大家。

> 学弟小牛的 GitHub 开源仓库：https://github.com/autoencoder-github/interviewtop
>
> 欢迎 star 关注。
>
> 学弟小牛的在线面试题库：http://interviewtop.top
>
> 欢迎阅览。

如果大家能背会的话，差不多就能“吊打”面试官一波了。

Java 基础篇（45 道）

Java 语言具有哪些特点?

- Java 为纯面向对象的语言。它能够直接反应现实生活中的对象。
- 具有平台无关性。Java 利用 Java 虚拟机运行字节码，无论是在 Windows、Linux 还是 MacOS 等其它平台对 Java 程序进行编译，编译后的程序可在其它平台运行。
- Java 为解释型语言，编译器把 Java 代码编译成平台无关的中间代码，然后在 JVM 上解释运行，具有很好的可移植性。
- Java 提供了很多内置类库。如对多线程支持，对网络通信支持，最重要的一点是提供了垃圾回收器。
- Java 具有较好的安全性和健壮性。Java 提供了异常处理和垃圾回收机制，去除了 C++中难以理解的指针特性。

JDK 与 JRE 有什么区别?

- JDK：Java 开发工具包（Java Development Kit），提供了 Java 的开发环境和运行环境。
- JRE：Java 运行环境(Java Runtime Environment)，提供了 Java 运行所需的环境。
- JDK 包含了 JRE。如果只运行 Java 程序，安装 JRE 即可。要编写 Java 程序需安装 JDK.

简述 Java 基本数据类型

- byte: 占用 1 个字节，取值范围-128 ~ 127
- short: 占用 2 个字节，取值范围-2^15^ ~ 2^15^-1
- int：占用 4 个字节，取值范围-2^31^ ~ 2^31^-1
- long：占用 8 个字节
- float：占用 4 个字节
- double：占用 8 个字节
- char: 占用 2 个字节
- boolean：占用大小根据实现虚拟机不同有所差异

简述自动装箱拆箱

对于 Java 基本数据类型，均对应一个包装类。

装箱就是自动将基本数据类型转换为包装器类型，如 int->Integer

拆箱就是自动将包装器类型转换为基本数据类型，如 Integer->int

简述 Java 访问修饰符

- default: 默认访问修饰符，在同一包内可见
- private: 在同一类内可见，不能修饰类
- protected : 对同一包内的类和所有子类可见，不能修饰类
- public: 对所有类可见

构造方法、成员变量初始化以及静态成员变量三者的初始化顺序?

先后顺序：静态成员变量、成员变量、构造方法。

详细的先后顺序：父类静态变量、父类静态代码块、子类静态变量、子类静态代码块、父类非静态变量、父类非静态代码块、父类构造函数、子类非静态变量、子类非静态代码块、子类构造函数。

Java 代码块执行顺序

- 父类静态代码块（只执行一次）
- 子类静态代码块（只执行一次）
- 父类构造代码块
- 父类构造函数
- 子类构造代码块
- 子类构造函数
- 普通代码块

面向对象的三大特性?

继承：对象的一个新类可以从现有的类中派生，派生类可以从它的基类那继承方法和实例变量，且派生类可以修改或新增新的方法使之更适合特殊的需求。

封装：将客观事物抽象成类，每个类可以把自身数据和方法只让可信的类或对象操作，对不可信的进行信息隐藏。

多态：允许不同类的对象对同一消息作出响应。不同对象调用相同方法即使参数也相同，最终表现行为是不一样的。

为什么 Java 语言不支持多重继承?

为了程序的结构能够更加清晰从而便于维护。假设 Java 语言支持多重继承，类 C 继承自类 A 和类 B，如果类 A 和 B 都有自定义的成员方法 `f()`，那么当代码中调用类 C 的 `f()` 会产生二义性。

Java 语言通过实现多个接口间接支持多重继承，接口由于只包含方法定义，不能有方法的实现，类 C 继承接口 A 与接口 B 时即使它们都有方法 `f()`，也不能直接调用方法，需实现具体的 `f()` 方法才能调用，不会产生二义性。

多重继承会使类型转换、构造方法的调用顺序变得复杂，会影响到性能。

简述 Java 的多态

Java 多态可以分为编译时多态和运行时多态。

编译时多态主要指方法的重载，即通过参数列表的不同来区分不同的方法。

运行时多态主要指继承父类和实现接口时，可使用父类引用指向子类对象。

运行时多态的实现：主要依靠方法表，方法表中最先存放的是 Object 类的方法，接下来是该类的父类的方法，最后是该类本身的方法。如果子类改写了父类的方法，那么子类和父类的那些同名方法共享一个方法表项，都被认作是父类的方法。因此可以实现运行时多态。

Java 提供的多态机制?

Java 提供了两种用于多态的机制，分别是重载与覆盖。

重载：重载是指同一个类中有多个同名的方法，但这些方法有不同的参数，在编译期间就可以确定调用哪个方法。

覆盖：覆盖是指派生类重写基类的方法，使用基类指向其子类的实例对象，或接口的引用变量指向其实现类的实例对象，在程序调用的运行期根据引用变量所指的具体实例对象调用正在运行的那个对象的方法，即需要到运行期才能确定调用哪个方法。

重载与覆盖的区别?

- 覆盖是父类与子类之间的关系，是垂直关系；重载是同一类中方法之间的关系，是水平关系。
- 覆盖只能由一个方法或一对方法产生关系；重载是多个方法之间的关系。
- 覆盖要求参数列表相同；重载要求参数列表不同。
- 覆盖中，调用方法体是根据对象的类型来决定的，而重载是根据调用时实参表与形参表来对应选择方法体。
- 重载方法可以改变返回值的类型，覆盖方法不能改变返回值的类型。

接口和抽象类的相同点和不同点?

相同点:

- 都不能被实例化。
- 接口的实现类或抽象类的子类需实现接口或抽象类中相应的方法才能被实例化。

不同点：

- 接口只能有方法定义，不能有方法的实现，而抽象类可以有方法的定义与实现。
- 实现接口的关键字为 implements，继承抽象类的关键字为 extends。一个类可以实现多个接口，只能继承一个抽象类。
- 当子类和父类之间存在逻辑上的层次结构，推荐使用抽象类，有利于功能的累积。当功能不需要，希望支持差别较大的两个或更多对象间的特定交互行为，推荐使用接口。使用接口能降低软件系统的耦合度，便于日后维护或添加删除方法。

简述抽象类与接口的区别

抽象类：体现的是 is-a 的关系，如对于 man is a person，就可以将 person 定义为抽象类。

接口：体现的是 can 的关系。是作为模板实现的。如设置接口 fly，plane 类和 bird 类均可实现该接口。

一个类只能继承一个抽象类，但可以实现多个接口。

简述内部类及其作用

- 成员内部类：作为成员对象的内部类。可以访问 private 及以上外部类的属性和方法。外部类想要访问内部类属性或方法时，必须要创建一个内部类对象，然后通过该对象访问内部类的属性或方法。外部类也可访问 private 修饰的内部类属性。
- 局部内部类：存在于方法中的内部类。访问权限类似局部变量，只能访问外部类的 final 变量。
- 匿名内部类：只能使用一次，没有类名，只能访问外部类的 final 变量。
- 静态内部类：类似类的静态成员变量。

Java 语言中关键字 static 的作用是什么?

static 的主要作用有两个：

- 为某种特定数据类型或对象分配与创建对象个数无关的单一的存储空间。
- 使得某个方法或属性与类而不是对象关联在一起，即在不创建对象的情况下可通过类直接调用方法或使用类的属性。

具体而言 static 又可分为 4 种使用方式：

- 修饰成员变量。用 static 关键字修饰的静态变量在内存中只有一个副本。只要静态变量所在的类被加载，这个静态变量就会被分配空间，可以使用“类.静态变量”和“对象.静态变量”的方法使用。
- 修饰成员方法。static 修饰的方法无需创建对象就可以被调用。static 方法中不能使用 this 和 super 关键字，不能调用非 static 方法，只能访问所属类的静态成员变量和静态成员方法。
- 修饰代码块。JVM 在加载类的时候会执行 static 代码块。static 代码块常用于初始化静态变量。static 代码块只会被执行一次。
- 修饰内部类。static 内部类可以不依赖外部类实例对象而被实例化。静态内部类不能与外部类有相同的名字，不能访问普通成员变量，只能访问外部类中的静态成员和静态成员方法。

为什么要把 String 设计为不可变?

- 节省空间：字符串常量存储在 JVM 的字符串池中可以被用户共享。
- 提高效率：String 可以被不同线程共享，是线程安全的。在涉及多线程操作中不需要同步操作。
- 安全：String 常被用于用户名、密码、文件名等使用，由于其不可变，可避免黑客行为对其恶意修改。

简述 String/StringBuffer 与 StringBuilder

String 类采用利用 final 修饰的字符数组进行字符串保存，因此不可变。如果对 String 类型对象修改，需要新建对象，将老字符和新增加的字符一并存进去。

StringBuilder，采用无 final 修饰的字符数组进行保存，因此可变。但线程不安全。

StringBuffer，采用无 final 修饰的字符数组进行保存，可理解为实现线程安全的 StringBuilder。

判等运算符==与 equals 的区别?

== 比较的是引用，equals 比较的是内容。

如果变量是基础数据类型，== 用于比较其对应值是否相等。如果变量指向的是对象，== 用于比较两个对象是否指向同一块存储空间。

equals 是 Object 类提供的方法之一，每个 Java 类都继承自 Object 类，所以每个对象都具有 equals 这个方法。Object 类中定义的 equals 方法内部是直接调用 == 比较对象的。但通过覆盖的方法可以让它不是比较引用而是比较数据内容。

简述 Object 类常用方法

- hashCode：通过对象计算出的散列码。用于 map 型或 equals 方法。需要保证同一个对象多次调用该方法，总返回相同的整型值。
- equals：判断两个对象是否一致。需保证 equals 方法相同对应的对象 hashCode 也相同。
- toString: 用字符串表示该对象
- clone:深拷贝一个对象

Java 中一维数组和二维数组的声明方式?

一维数组的声明方式：

```
type arrayName[]
type[] arrayName
```

二维数组的声明方式：

```
type arrayName[][]
type[][] arrayName
type[] arrayName[]
```

其中 type 为基本数据类型或类，arrayName 为数组名字

简述 Java 异常的分类

Java 异常分为 Error（程序无法处理的错误），和 Exception（程序本身可以处理的异常）。这两个类均继承 Throwable。

Error 常见的有 StackOverFlowError、OutOfMemoryError 等等。

Exception 可分为运行时异常和非运行时异常。对于运行时异常，可以利用 try catch 的方式进行处理，也可以不处理。对于非运行时异常，必须处理，不处理的话程序无法通过编译。

简述 throw 与 throws 的区别

throw 一般是用在方法体的内部，由开发者定义当程序语句出现问题后主动抛出一个异常。

throws 一般用于方法声明上，代表该方法可能会抛出的异常列表。

出现在 Java 程序中的 finally 代码块是否一定会执行?

当遇到下面情况不会执行。

- 当程序在进入 try 语句块之前就出现异常时会直接结束。
- 当程序在 try 块中强制退出时，如使用 System.exit(0)，也不会执行 finally 块中的代码。

其它情况下，在 try/catch/finally 语句执行的时候，try 块先执行，当有异常发生，catch 和 finally 进行处理后程序就结束了，当没有异常发生，在执行完 finally 中的代码后，后面代码会继续执行。值得注意的是，当 try/catch 语句块中有 return 时，finally 语句块中的代码会在 return 之前执行。如果 try/catch/finally 块中都有 return 语句，finally 块中的 return 语句会覆盖 try/catch 模块中的 return 语句。

final、finally 和 finalize 的区别是什么?

- final 用于声明属性、方法和类，分别表示属性不可变、方法不可覆盖、类不可继承。
- finally 作为异常处理的一部分，只能在 try/catch 语句中使用，finally 附带一个语句块用来表示这个语句最终一定被执行，经常被用在需要释放资源的情况下。
- finalize 是 Object 类的一个方法，在垃圾收集器执行的时候会调用被回收对象的 finalize()方法。当垃圾回收器准备好释放对象占用空间时，首先会调用 finalize()方法，并在下一次垃圾回收动作发生时真正回收对象占用的内存。

简述泛型

泛型，即“参数化类型”，解决不确定对象具体类型的问题。在编译阶段有效。在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型在类中称为泛型类、接口中称为泛型接口和方法中称为泛型方法。

简述泛型擦除

Java 编译器生成的字节码是不包涵泛型信息的，泛型类型信息将在编译处理是被擦除，这个过程被称为泛型擦除。

简述注解

Java 注解用于为 Java 代码提供元数据。作为元数据，注解不直接影响你的代码执行，但也有一些类型的注解实际上可以用于这一目的。

其可以用于提供信息给编译器，在编译阶段时给软件提供信息进行相关的处理，在运行时处理写相应代码，做对应操作。

简述元注解

元注解可以理解为注解的注解，即在注解中使用，实现想要的功能。其具体分为：

- @Retention: 表示注解存在阶段是保留在源码，还是在字节码（类加载）或者运行期（JVM 中运行）。
- @Target：表示注解作用的范围。
- @Documented：将注解中的元素包含到 Javadoc 中去。
- @Inherited：一个被@Inherited 注解了的注解修饰了一个父类，如果他的子类没有被其他注解修饰，则它的子类也继承了父类的注解。
- @Repeatable：被这个元注解修饰的注解可以同时作用一个对象多次，但是每次作用注解又可以代表不同的含义。

简述 Java 中 Class 对象

java 中对象可以分为实例对象和 Class 对象，每一个类都有一个 Class 对象，其包含了与该类有关的信息。

获取 Class 对象的方法：

```
Class.forName("类的全限定名")
实例对象.getClass()
类名.class
```

Java 反射机制是什么?

Java 反射机制是指在程序的运行过程中可以构造任意一个类的对象、获取任意一个类的成员变量和成员方法、获取任意一个对象所属的类信息、调用任意一个对象的属性和方法。反射机制使得 Java 具有动态获取程序信息和动态调用对象方法的能力。可以通过以下类调用反射 API。

- Class 类：可获得类属性方法
- Field 类：获得类的成员变量
- Method 类：获取类的方法信息
- Construct 类：获取类的构造方法等信息

序列化是什么?

序列化是一种将对象转换成字节序列的过程，用于解决在对对象流进行读写操作时所引发的问题。序列化可以将对象的状态写在流里进行网络传输，或者保存到文件、数据库等系统里，并在需要的时候把该流读取出来重新构造成一个相同的对象。

简述 Java 序列化与反序列化的实现

序列化：将 java 对象转化为字节序列，由此可以通过网络对象进行传输。

反序列化：将字节序列转化为 java 对象。

具体实现：实现 Serializable 接口，或实现 Externalizable 接口中的 writeExternal()与 readExternal()方法。

简述 Java 的 List

List 是一个有序队列，在 Java 中有两种实现方式:

ArrayList 使用数组实现，是容量可变的非线程安全列表，随机访问快，集合扩容时会创建更大的数组，把原有数组复制到新数组。

LinkedList 本质是双向链表，与 ArrayList 相比插入和删除速度更快，但随机访问元素很慢。

Java 中线程安全的基本数据结构有哪些

- HashTable: 哈希表的线程安全版，效率低

- ConcurrentHashMap：哈希表的线程安全版，效率高，用于替代 HashTable
- Vector：线程安全版 Arraylist
- Stack：线程安全版栈
- BlockingQueue 及其子类：线程安全版队列

简述 Java 的 Set

Set 即集合，该数据结构不允许元素重复且无序。Java 对 Set 有三种实现方式：

HashSet 通过 HashMap 实现，HashMap 的 Key 即 HashSet 存储的元素，Value 系统自定义一个名为 PRESENT 的 Object 类型常量。判断元素是否相同时，先比较 hashCode，相同后再利用 equals 比较，查询 O(1)

LinkedHashSet 继承自 HashSet，通过 LinkedHashMap 实现，使用双向链表维护元素插入顺序。

TreeSet 通过 TreeMap 实现的，底层数据结构是红黑树，添加元素到集合时按照比较规则将其插入合适的位置，保证插入后的集合仍然有序。查询 O(logn)

简述 Java 的 HashMap

JDK8 之前底层实现是数组 + 链表，JDK8 改为数组 + 链表/红黑树。主要成员变量包括存储数据的 table 数组、元素数量 size、加载因子 loadFactor。HashMap 中数据以键值对的形式存在，键对应的 hash 值用来计算数组下标，如果两个元素 key 的 hash 值一样，就会发生哈希冲突，被放到同一个链表上。

table 数组记录 HashMap 的数据，每个下标对应一条链表，所有哈希冲突的数据都会被存放到同一条链表，Node/Entry 节点包含四个成员变量：key、value、next 指针和 hash 值。在 JDK8 后链表超过 8 会转化为红黑树。

若当前数据/总数据容量>负载因子，Hashmap 将执行扩容操作。默认初始化容量为 16，扩容容量必须是 2 的幂次方、最大容量为 1<< 30 、默认加载因子为 0.75。

为何 HashMap 线程不安全

在 JDK1.7 中，HashMap 采用头插法插入元素，因此并发情况下会导致环形链表，产生死循环。

虽然 JDK1.8 采用了尾插法解决了这个问题，但是并发下的 put 操作也会使前一个 key 被后一个 key 覆盖。

由于 HashMap 有扩容机制存在，也存在 A 线程进行扩容后，B 线程执行 get 方法出现失误的情况。

简述 Java 的 TreeMap

TreeMap 是底层利用红黑树实现的 Map 结构，底层实现是一棵平衡的排序二叉树，由于红黑树的插入、删除、遍历时间复杂度都为 O(logN)，所以性能上低于哈希表。但是哈希表无法提供键值对的有序输出，红黑树可以按照键的值的大小有序输出。

ArrayList、Vector 和 LinkedList 有什么共同点与区别?

- ArrayList、Vector 和 LinkedList 都是可伸缩的数组，即可以动态改变长度的数组。
- ArrayList 和 Vector 都是基于存储元素的 Object[] array 来实现的，它们会在内存中开辟一块连续的空间来存储，支持下标、索引访问。但在涉及插入元素时可能需要移动容器中的元素，插入效率较低。当存储元素超过容器的初始化容量大小，ArrayList 与 Vector 均会进行扩容。
- Vector 是线程安全的，其大部分方法是直接或间接同步的。ArrayList 不是线程安全的，其方法不具有同步性质。LinkedList 也不是线程安全的。
- LinkedList 采用双向列表实现，对数据索引需要从头开始遍历，因此随机访问效率较低，但在插入元素的时候不需要对数据进行移动，插入效率较高。

HashMap 和 Hashtable 有什么区别?

- HashMap 是 Hashtable 的轻量级实现，HashMap 允许 key 和 value 为 null，但最多允许一条记录的 key 为 null.而 HashTable 不允许。
- HashTable 中的方法是线程安全的，而 HashMap 不是。在多线程访问 HashMap 需要提供额外的同步机制。
- Hashtable 使用 Enumeration 进行遍历，HashMap 使用 Iterator 进行遍历。

如何决定使用 HashMap 还是 TreeMap?

如果对 Map 进行插入、删除或定位一个元素的操作更频繁，HashMap 是更好的选择。如果需要对 key 集合进行有序的遍历，TreeMap 是更好的选择。

HashSet 中，equals 与 hashCode 之间的关系?

equals 和 hashCode 这两个方法都是从 object 类中继承过来的，equals 主要用于判断对象的内存地址引用是否是同一个地址；hashCode 根据定义的哈希规则将对象的内存地址转换为一个哈希码。HashSet 中存储的元素是不能重复的，主要通过 hashCode 与 equals 两个方法来判断存储的对象是否相同：

- 如果两个对象的 hashCode 值不同，说明两个对象不相同。
- 如果两个对象的 hashCode 值相同，接着会调用对象的 equals 方法，如果 equlas 方法的返回结果为 true，那么说明两个对象相同，否则不相同。

fail-fast 和 fail-safe 迭代器的区别是什么?

- fail-fast 直接在容器上进行，在遍历过程中，一旦发现容器中的数据被修改，就会立刻抛出 ConcurrentModificationException 异常从而导致遍历失败。常见的使用 fail-fast 方式的容器有 HashMap 和 ArrayList 等。
- fail-safe 这种遍历基于容器的一个克隆。因此对容器中的内容修改不影响遍历。常见的使用 fail-safe 方式遍历的容器有 ConcurrentHashMap 和 CopyOnWriteArrayList。

Collection 和 Collections 有什么区别?

- Collection 是一个集合接口，它提供了对集合对象进行基本操作的通用接口方法，所有集合都是它的子类，比如 List、Set 等。
- Collections 是一个包装类，包含了很多静态方法、不能被实例化，而是作为工具类使用，比如提供的排序方法：Collections.sort(list);提供的反转方法：Collections.reverse(list)。

《**Java 核心技术八股文**》预计一个月左右会有一次内容更新和完善，大家在我的公众号 **沉默王二** 后台回复“**07**”即可获取最新版！如果觉得内容不错的话，欢迎转发分享！

也欢迎关注小牛的公众号：**后端技术小牛说**，不定期分享技术文和一线大厂内推机会。

Java 并发编程（39 道）

简述Java内存模型（JMM）

Java内存模型定义了程序中各种变量的访问规则：

- 所有变量都存储在主存，每个线程都有自己的工作内存。
- 工作内存中保存了被该线程使用的变量的主存副本，线程对变量的所有操作都必须在工作空间进行，不能直接读写主内存数据。
- 操作完成后，线程的工作内存通过缓存一致性协议将操作完的数据刷回主存。

简述as-if-serial

编译器会对原始的程序进行指令重排序和优化。但不管怎么重排序，其结果都必须和用户原始程序输出的预定结果保持一致。

简述happens-before/八大规则

- 程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作；
- 锁定规则：一个unLock操作先行发生于后面对同一个锁的lock操作；
- volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作；
- 传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C；
- 线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作；
- 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生；
- 线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行；
- 对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始；

as-if-serial 和 happens-before 的区别

as-if-serial 保证单线程程序的执行结果不变，happens-before 保证正确同步的多线程程序的执行结果不变。

简述原子性操作

一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行，这就是原子性操作。

简述线程的可见性

可见性指当一个线程修改了共享变量时，其他线程能够立即得知修改。volatile、synchronized、final 关键字都能保证可见性。

简述有序性

虽然多线程存在并发和指令优化等操作，但在本线程内观察该线程的所有执行操作是有序的。

简述Java中volatile关键字作用

- 保证变量对所有线程的可见性。当一个线程修改了变量值，新值对于其他线程来说是立即可以得知的。
- 禁止指令重排。使用 volatile 变量进行写操作，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器进行重排序。

Java线程的实现方式

- 实现Runnable接口
- 继承Thread类
- 实现Callable接口

简述Java线程的状态

线程状态有 NEW、RUNNABLE、BLOCK、WAITING、TIMED_WAITING、THERMINATED

- NEW：新建状态，线程被创建且未启动，此时还未调用 start 方法。
- RUNNABLE：运行状态。表示线程正在JVM中执行，但是这个执行，不一定真的在跑，也可能在排队等CPU。
- BLOCKED：阻塞状态。线程等待获取锁，锁还没获得。
- WAITING：等待状态。线程内run方法执行完Object.wait()/Thread.join()进入该状态。
- TIMED_WAITING：限期等待。在一定时间之后跳出状态。调用Thread.sleep(long) Object.wait(long) Thread.join(long)进入状态。其中这些参数代表等待的时间。
- TERMINATED：结束状态。线程调用完run方法进入该状态。

简述线程通信的方式

- volatile 关键词修饰变量，保证所有线程对变量访问的可见性。
- synchronized关键词。确保多个线程在同一时刻只能有一个处于方法或同步块中。
- wait/notify方法
- IO通信

简述线程池

没有线程池的情况下，多次创建，销毁线程开销比较大。如果在开辟的线程执行完当前任务后复用已创建的线程，可以降低开销、控制最大并发数。

线程池创建线程时，会将线程封装成工作线程 Worker，Worker 在执行完任务后还会循环获取工作队列中的任务来执行。

将任务派发给线程池时，会出现以下几种情况

- 核心线程池未满，创建一个新的线程执行任务。
- 如果核心线程池已满，工作队列未满，将线程存储在工作队列。
- 如果工作队列已满，线程数小于最大线程数就创建一个新线程处理任务。
- 如果超过大小线程数，按照拒绝策略来处理任务。

线程池参数：

- corePoolSize：常驻核心线程数。超过该值后如果线程空闲会被销毁。
- maximumPoolSize：线程池能够容纳同时执行的线程最大数。
- keepAliveTime：线程空闲时间，线程空闲时间达到该值后会被销毁，直到只剩下 corePoolSize 个线程为止，避免浪费内存资源。
- workQueue：工作队列。
- threadFactory：线程工厂，用来生产一组相同任务的线程。
- handler：拒绝策略。

拒绝策略有以下几种：

- AbortPolicy：丢弃任务并抛出异常
- CallerRunsPolicy：重新尝试提交该任务
- DiscardOldestPolicy 抛弃队列里等待最久的任务并把当前任务加入队列
- DiscardPolicy 表示直接抛弃当前任务但不抛出异常。

简述Executor框架

Executor框架目的是将任务提交和任务如何运行分离开来的机制。用户不再需要从代码层考虑设计任务的提交运行，只需要调用Executor框架实现类的Execute方法就可以提交任务。

简述Executor的继承关系

- Executor：一个接口，其定义了一个接收Runnable对象的方法executor，该方法接收一个Runable实例执行这个任务。
- ExecutorService：Executor的子类接口，其定义了一个接收Callable对象的方法，返回 Future 对象，同时提供execute方法。
- ScheduledExecutorService：ExecutorService的子类接口，支持定期执行任务。
- AbstractExecutorService：抽象类，提供 ExecutorService 执行方法的默认实现。
- Executors：实现ExecutorService接口的静态工厂类，提供了一系列工厂方法用于创建线程池。
- ThreadPoolExecutor：继承AbstractExecutorService，用于创建线程池。
- ForkJoinPool: 继承AbstractExecutorService，Fork 将大任务分叉为多个小任务，然后让小任务执行，Join 是获得小任务的结果，类似于map reduce。
- ThreadPoolExecutor：继承ThreadPoolExecutor，实现ScheduledExecutorService，用于创建带定时任务的线程池。

简述线程池的状态

- Running：能接受新提交的任务，也可以处理阻塞队列的任务。
- Shutdown：不再接受新提交的任务，但可以处理存量任务，线程池处于running时调用shutdown方法，会进入该状态。
- Stop：不接受新任务，不处理存量任务，调用shutdownnow进入该状态。
- Tidying：所有任务已经终止了，worker_count（有效线程数）为0。
- Terminated：线程池彻底终止。在tidying模式下调用terminated方法会进入该状态。

简述线程池类型

- newCachedThreadPool 可缓存线程池，可设置最小线程数和最大线程数，线程空闲1分钟后自动销毁。
- newFixedThreadPool 指定工作线程数量线程池。
- newSingleThreadExecutor 单线程Executor。
- newScheduleThreadPool 支持定时任务的指定工作线程数量线程池。
- newSingleThreadScheduledExecutor 支持定时任务的单线程Executor。

简述阻塞队列

阻塞队列是生产者消费者的实现具体组件之一。当阻塞队列为空时，从队列中获取元素的操作将会被阻塞，当阻塞队列满了，往队列添加元素的操作将会被阻塞。具体实现有：

- ArrayBlockingQueue：底层是由数组组成的有界阻塞队列。
- LinkedBlockingQueue：底层是由链表组成的有界阻塞队列。
- PriorityBlockingQueue：阻塞优先队列。
- DelayQueue：创建元素时可以指定多久才能从队列中获取当前元素
- SynchronousQueue：不存储元素的阻塞队列，每一个存储必须等待一个取出操作

- LinkedTransferQueue：与LinkedBlockingQueue相比多一个transfer方法，即如果当前有消费者正等待接收元素，可以把生产者传入的元素立刻传输给消费者。
- LinkedBlockingDeque：双向阻塞队列。

谈一谈ThreadLocal

ThreadLocal 是线程共享变量。ThreadLoacl 有一个静态内部类 ThreadLocalMap，其 Key 是 ThreadLocal 对象，值是 Entry 对象，ThreadLocalMap是每个线程私有的。

- set 给ThreadLocalMap设置值。
- get 获取ThreadLocalMap。
- remove 删除ThreadLocalMap类型的对象。

存在的问题：对于线程池，由于线程池会重用 Thread 对象，因此与 Thread 绑定的 ThreadLocal 也会被重用，造成一系列问题。

比如说内存泄漏。由于 ThreadLocal 是弱引用，但 Entry 的 value 是强引用，因此当 ThreadLocal 被垃圾回收后，value 依旧不会被释放，产生内存泄漏。

聊聊你对Java并发包下unsafe类的理解

对于 Java 语言，没有直接的指针组件，一般也不能使用偏移量对某块内存进行操作。这些操作相对来讲是安全（safe）的。

Java 有个类叫 Unsafe 类，这个类使 Java 拥有了像 C 语言的指针一样操作内存空间的能力，同时也带来了指针的问题。这个类可以说是 Java 并发开发的基础。

Java中的乐观锁与CAS算法

乐观锁认为数据发送时发生并发冲突的概率不大，所以读操作前不上锁。

到了写操作时才会进行判断，数据在此期间是否被其他线程修改。如果发生修改，那就返回写入失败；如果没有被修改，那就执行修改操作，返回修改成功。

乐观锁一般都采用 Compare And Swap（CAS）算法进行实现。顾名思义，该算法涉及到了两个操作，比较（Compare）和交换（Swap）。

CAS 算法的思路如下：

- 该算法认为不同线程对变量的操作时产生竞争的情况比较少。
- 该算法的核心是对当前读取变量值 E 和内存中的变量旧值 V 进行比较。
- 如果相等，就代表其他线程没有对该变量进行修改，就将变量值更新为新值 N。
- 如果不等，就认为在读取值 E 到比较阶段，有其他线程对变量进行过修改，不进行任何操作。

ABA问题及解决方法简述

CAS 算法是基于值来做比较的，如果当前有两个线程，一个线程将变量值从 A 改为 B ，再由 B 改回为 A ，当前线程开始执行 CAS 算法时，就很容易认为值没有变化，误认为读取数据到执行 CAS 算法的期间，没有线程修改过数据。

juc 包提供了一个 AtomicStampedReference，即在原始的版本下加入版本号戳，解决 ABA 问题。

简述常见的Atomic类

在很多时候，我们需要的仅仅是一个简单的、高效的、线程安全的++或者--方案，使用synchronized关键字和lock固然可以实现，但代价比较大，此时用原子类更加方便。基本数据类型的原子类有：

- AtomicInteger 原子更新整形
- AtomicLong 原子更新长整型
- AtomicBoolean 原子更新布尔类型

Atomic数组类型有：

- AtomicIntegerArray 原子更新整形数组里的元素
- AtomicLongArray 原子更新长整型数组里的元素
- AtomicReferenceArray 原子更新引用类型数组里的元素。

Atomic引用类型有：

- AtomicReference 原子更新引用类型
- AtomicMarkableReference 原子更新带有标记位的引用类型，可以绑定一个 boolean 标记
- AtomicStampedReference 原子更新带有版本号的引用类型

FieldUpdater类型：

- AtomicIntegerFieldUpdater 原子更新整形字段的更新器
- AtomicLongFieldUpdater 原子更新长整形字段的更新器
- AtomicReferenceFieldUpdater 原子更新引用类型字段的更新器

简述Atomic类基本实现原理

以AtomicIntger 为例。

方法getAndIncrement，以原子方式将当前的值加1，具体实现为：

- 在 for 死循环中取得 AtomicInteger 里存储的数值
- 对 AtomicInteger 当前的值加 1

- 调用 compareAndSet 方法进行原子更新
- 先检查当前数值是否等于 expect
- 如果等于则说明当前值没有被其他线程修改，则将值更新为 next，
- 如果不是会更新失败返回 false，程序会进入 for 循环重新进行 compareAndSet 操作。

简述CountDownLatch

CountDownLatch这个类使一个线程等待其他线程各自执行完毕后再执行。是通过一个计数器来实现的，计数器的初始值是线程的数量。每当一个线程执行完毕后，调用countDown方法，计数器的值就减1，当计数器的值为0时，表示所有线程都执行完毕，然后在等待的线程就可以恢复工作了。只能一次性使用，不能reset。

简述CyclicBarrier

CyclicBarrier 主要功能和CountDownLatch类似，也是通过一个计数器，使一个线程等待其他线程各自执行完毕后再执行。但是其可以重复使用（reset）。

简述Semaphore

Semaphore即信号量。Semaphore 的构造方法参数接收一个 int 值，设置一个计数器，表示可用的许可数量即最大并发数。使用 acquire 方法获得一个许可证，计数器减一，使用 release 方法归还许可，计数器加一。如果此时计数器值为0，线程进入休眠。

简述Exchanger

Exchanger类可用于两个线程之间交换信息。可简单地将Exchanger对象理解为一个包含两个格子的容器，通过exchanger方法可以向两个格子中填充信息。线程通过exchange 方法交换数据，第一个线程执行exchange 方法后会阻塞等待第二个线程执行该方法。当两个线程都到达同步点时这两个线程就可以交换数据当两个格子中的均被填充时，该对象会自动将两个格子的信息交换，然后返回给线程，从而实现两个线程的信息交换。

简述ConcurrentHashMap

JDK7采用锁分段技术。首先将数据分成 Segment 数据段，然后给每一个数据段配一把锁，当一个线程占用锁访问其中一个段的数据时，其他段的数据也能被其他线程访问。

get 除读到空值不需要加锁。该方法先经过一次再散列，再用这个散列值通过散列运算定位到 Segment，最后通过散列算法定位到元素。put 须加锁，首先定位到 Segment，然后进行插入操作，第一步判断是否需要对 Segment 里的 HashEntry 数组进行扩容，第二步定位添加元素的位置，然后将其放入数组。

JDK8的改进

- 取消分段锁机制，采用CAS算法进行值的设置，如果CAS失败再使用 synchronized 加锁添加元素

- 引入红黑树结构，当某个槽内的元素个数超过8且 Node数组 容量大于 64 时，链表转为红黑树。
- 使用了更加优化的方式统计集合内的元素数量。

synchronized底层实现原理

Java 对象底层都会关联一个 monitor，使用 synchronized 时 JVM 会根据使用环境找到对象的 monitor，根据 monitor 的状态进行加解锁的判断。如果成功加锁就成为该 monitor 的唯一持有者，monitor 在被释放前不能再被其他线程获取。

synchronized在JVM编译后会产生monitorenter 和 monitorexit 这两个字节码指令，获取和释放 monitor。这两个字节码指令都需要一个引用类型的参数指明要锁定和解锁的对象，对于同步普通方法，锁是当前实例对象；对于静态同步方法，锁是当前类的 Class 对象；对于同步方法块，锁是 synchronized 括号里的对象。

执行 monitorenter 指令时，首先尝试获取对象锁。如果这个对象没有被锁定，或当前线程已经持有锁，就把锁的计数器加 1，执行 monitorexit 指令时会将锁计数器减 1。一旦计数器为 0 锁随即就被释放。

synchronized关键词使用方法

- 直接修饰某个实例方法
- 直接修饰某个静态方法
- 修饰代码块

简述Java偏向锁

JDK 1.6 中提出了偏向锁的概念。该锁提出的原因是，开发者发现多数情况下锁并不存在竞争，一把锁往往是由同一个线程获得的。偏向锁并不会主动释放，这样每次偏向锁进入的时候都会判断该资源是否是偏向自己的，如果是偏向自己的则不需要进行额外的操作，直接可以进入同步操作。

其申请流程为：

- 首先需要判断对象的 Mark Word 是否属于偏向模式，如果不属于，那就进入轻量级锁判断逻辑。否则继续下一步判断；
- 判断目前请求锁的线程 ID 是否和偏向锁本身记录的线程 ID 一致。如果一致，继续下一步的判断，如果不一致，跳转到步骤4；
- 判断是否需要重偏向。如果不用的话，直接获得偏向锁；
- 利用 CAS 算法将对象的 Mark Word 进行更改，使线程 ID 部分换成本线程 ID。如果更换成功，则重偏向完成，获得偏向锁。如果失败，则说明有多线程竞争，升级为轻量级锁。

简述轻量级锁

轻量级锁是为了在没有竞争的前提下减少重量级锁出现并导致的性能消耗。

其申请流程为：

- 如果同步对象没有被锁定，虚拟机将在当前线程的栈帧中建立一个锁记录空间，存储锁对象目前 Mark Word 的拷贝。
- 虚拟机使用 CAS 尝试把对象的 Mark Word 更新为指向锁记录的指针
- 如果更新成功即代表该线程拥有了锁，锁标志位将转变为 00，表示处于轻量级锁定状态。
- 如果更新失败就意味着至少存在一条线程与当前线程竞争。虚拟机检查对象的 Mark Word 是否指向当前线程的栈帧
- 如果指向当前线程的栈帧，说明当前线程已经拥有了锁，直接进入同步块继续执行
- 如果不是则说明锁对象已经被其他线程抢占。
- 如果出现两条以上线程争用同一个锁，轻量级锁就不再有效，将膨胀为重量级锁，锁标志状态变为 10，此时Mark Word 存储的就是指向重量级锁的指针，后面等待锁的线程也必须阻塞。

简述锁优化策略

即自适应自旋、锁消除、锁粗化、锁升级等策略偏。

简述Java的自旋锁

线程获取锁失败后，可以采用这样的策略，可以不放弃 CPU ，不停的重试内重试，这种操作也称为自旋锁。

简述自适应自旋锁

自适应自旋锁自旋次数不再人为设定，通常由前一次在同一个锁上的自旋时间及锁的拥有者的状态决定。

简述锁粗化

锁粗化的思想就是扩大加锁范围，避免反复的加锁和解锁。

简述锁消除

锁消除是一种更为彻底的优化，在编译时，Java编译器对运行上下文进行扫描，去除不可能存在共享资源竞争的锁。

简述Lock与ReentrantLock

Lock接口是 Java并发包的顶层接口。

可重入锁 ReentrantLock 是 Lock 最常见的实现，与 synchronized 一样可重入。ReentrantLock 在默认情况下是非公平的，可以通过构造方法指定公平锁。一旦使用了公平锁，性能会下降。

简述AQS

AQS（AbstractQuenedSynchronizer）抽象的队列式同步器。AQS是将每一条请求共享资源的线程封装成一个锁队列的一个结点（Node），来实现锁的分配。AQS是用来构建锁或其他同步组件的基础框架，它使用一个 volatile int state 变量作为共享资源，如果线程获取资源失败，则进入同步队列等待；如果获取成功就执行临界区代码，释放资源时会通知同步队列中的等待线程。

子类通过继承同步器并实现它的抽象方法getState、setState 和 compareAndSetState对同步状态进行更改。

AQS获取独占锁/释放独占锁原理：

获取：（acquire）

- 调用 tryAcquire 方法安全地获取线程同步状态，获取失败的线程会被构造同步节点并通过 addWaiter 方法加入到同步队列的尾部，在队列中自旋。
- 调用 acquireQueued 方法使得该节点以死循环的方式获取同步状态，如果获取不到则阻塞。

释放：（release）

- 调用 tryRelease 方法释放同步状态
- 调用 unparkSuccessor 方法唤醒头节点的后继节点，使后继节点重新尝试获取同步状态。

AQS获取共享锁/释放共享锁原理

获取锁（acquireShared）

- 调用 tryAcquireShared 方法尝试获取同步状态，返回值不小于 0 表示能获取同步状态。
- 释放（releaseShared），并唤醒后续处于等待状态的节点。

也欢迎关注小牛的公众号：**后端技术小牛说**，不定期分享技术文和一线大厂内推机会。

Java 虚拟机篇（29 道）

简述JVM内存模型

线程私有的运行时数据区: 程序计数器、Java 虚拟机栈、本地方法栈。

线程共享的运行时数据区：Java 堆、方法区。

简述程序计数器

程序计数器表示当前线程所执行的字节码的行号指示器。

程序计数器不会产生StackOverflowError和OutOfMemoryError。

简述虚拟机栈

Java 虚拟机栈用来描述 Java 方法执行的内存模型。线程创建时就会分配一个栈空间，线程结束后栈空间被回收。

栈中元素用于支持虚拟机进行方法调用，每个方法在执行时都会创建一个栈帧存储方法的局部变量表、操作栈、动态链接和返回地址等信息。

虚拟机栈会产生两类异常：

- StackOverflowError：线程请求的栈深度大于虚拟机允许的深度抛出。

- OutOfMemoryError：如果 JVM 栈容量可以动态扩展，虚拟机栈占用内存超出抛出。

简述本地方法栈

本地方法栈与虚拟机栈作用相似，不同的是虚拟机栈为虚拟机执行 Java 方法服务，本地方法栈为本地方法服务。可以将虚拟机栈看作普通的java函数对应的内存模型，本地方法栈看作由native关键词修饰的函数对应的内存模型。

本地方法栈会产生两类异常：

- StackOverflowError：线程请求的栈深度大于虚拟机允许的深度抛出。
- OutOfMemoryError：如果 JVM 栈容量可以动态扩展，虚拟机栈占用内存超出抛出。

简述JVM中的堆

堆主要作用是存放对象实例，Java 里几乎所有对象实例都在堆上分配内存，堆也是内存管理中最大的一块。Java的垃圾回收主要就是针对堆这一区域进行。 可通过 -Xms 和 -Xmx 设置堆的最小和最大容量。

堆会抛出 OutOfMemoryError异常。

简述方法区

方法区用于存储被虚拟机加载的类信息、常量、静态变量等数据。

JDK6之前使用永久代实现方法区，容易内存溢出。JDK7 把放在永久代的字符串常量池、静态变量等移出，JDK8 中抛弃永久代，改用在本地内存中实现的元空间来实现方法区，把 JDK 7 中永久代内容移到元空间。

方法区会抛出 OutOfMemoryError异常。

简述运行时常量池

运行时常量池存放常量池表，用于存放编译器生成的各种字面量与符号引用。一般除了保存 Class 文件中描述的符号引用外，还会把符号引用翻译的直接引用也存储在运行时常量池。除此之外，也会存放字符串基本类型。

JDK8之前，放在方法区，大小受限于方法区。JDK8将运行时常量池存放堆中。

简述直接内存

直接内存也称为堆外内存，就是把内存对象分配在JVM堆外的内存区域。这部分内存不是虚拟机管理，而是由操作系统来管理。 Java通过DriectByteBuffer对其进行操作，避免了在 Java 堆和 Native堆来回复制数据。

简述Java创建对象的过程

- 检查该指令的参数能否在常量池中定位到一个类的符号引用，并检查引用代表的类是否已被加载、解析和初始化，如果没有就先执行类加载。
- 通过检查通过后虚拟机将为新生对象分配内存。
- 完成内存分配后虚拟机将成员变量设为零值
- 设置对象头，包括哈希码、GC 信息、锁信息、对象所属类的类元信息等。
- 执行 init 方法，初始化成员变量，执行实例化代码块，调用类的构造方法，并把堆内对象的首地址赋值给引用变量。

简述JVM给对象分配内存的策略

- 指针碰撞：这种方式在内存中放一个指针作为分界指示器将使用过的内存放在一边，空闲的放在另一边，通过指针挪动完成分配。
- 空闲列表：对于 Java 堆内存不规整的情况，虚拟机必须维护一个列表记录哪些内存可用，在分配时从列表中找到一块足够大的空间划分给对象并更新列表记录。

Java对象内存分配是如何保证线程安全的

第一种方法，采用CAS机制，配合失败重试的方式保证更新操作的原子性。该方式效率低。

第二种方法，每个线程在Java堆中预先分配一小块内存，然后再给对象分配内存的时候，直接在自己这块"私有"内存中分配。一般采用这种策略。

简述对象的内存布局

对象在堆内存的存储布局可分为对象头、实例数据和对齐填充。

1）对象头主要包含两部分数据： MarkWord、类型指针。

MarkWord 用于存储哈希码（HashCode）、GC分代年龄、锁状态标志位、线程持有的锁、偏向线程ID等信息。

类型指针即对象指向他的类元数据指针，如果对象是一个 Java 数组，会有一块用于记录数组长度的数据。

2）实例数据存储代码中所定义的各种类型的字段信息。

3）对齐填充起占位作用。HotSpot 虚拟机要求对象的起始地址必须是8的整数倍，因此需要对齐填充。

如何判断对象是否是垃圾

1）引用计数法：

设置引用计数器，对象被引用计数器加 1，引用失效时计数器减 1，如果计数器为 0 则被标记为垃圾。会存在对象间循环引用的问题，一般不使用这种方法。

2）可达性分析：

通过 GC Roots 的根对象作为起始节点，从这些节点开始，根据引用关系向下搜索，如果某个对象没有被搜到，则会被标记为垃圾。可作为 GC Roots 的对象包括虚拟机栈和本地方法栈中引用的对象、类静态属性引用的对象、常量引用的对象。

简述java的引用类型

- 强引用： 被强引用关联的对象不会被回收。一般采用 new 方法创建强引用。
- 软引用：被软引用关联的对象只有在内存不够的情况下才会被回收。一般采用 SoftReference 类来创建软引用。
- 弱引用：垃圾收集器碰到即回收，也就是说它只能存活到下一次垃圾回收发生之前。一般采用 WeakReference 类来创建弱引用。
- 虚引用： 无法通过该引用获取对象。唯一目的就是为了能在对象被回收时收到一个系统通知。虚引用必须与引用队列联合使用。

简述标记清除算法、标记整理算法和标记复制算法

- 标记清除算法：先标记需清除的对象，之后统一回收。这种方法效率不高，会产生大量不连续的碎片。
- 标记整理算法：先标记存活对象，然后让所有存活对象向一端移动，之后清理端边界以外的内存
- 标记复制算法：将可用内存按容量划分为大小相等的两块，每次只使用其中一块。当使用的这块空间用完了，就将存活对象复制到另一块，再把已使用过的内存空间一次清理掉。

简述分代收集算法

根据对象存活周期将内存划分为几块，不同块采用适当的收集算法。

一般将堆分为新生代和老年代，对这两块采用不同的算法。

新生代使用：标记复制算法

老年代使用：标记清除或者标记整理算法

简述Serial垃圾收集器

Serial垃圾收集器是单线程串行收集器。垃圾回收的时候，必须暂停其他所有线程。新生代使用标记复制算法，老年代使用标记整理算法。简单高效。

简述ParNew垃圾收集器

ParNew垃圾收集器可以看作Serial垃圾收集器的多线程版本，新生代使用标记复制算法，老年代使用标记整理算法。

简述Parallel Scavenge垃圾收集器

注重吞吐量，即 CPU运行代码时间/CPU耗时总时间（CPU运行代码时间+ 垃圾回收时间）。新生代使用标记复制算法，老年代使用标记整理算法。

简述CMS垃圾收集器

CMS垃圾收集器注重最短时间停顿。CMS垃圾收集器为最早提出的并发收集器，垃圾收集线程与用户线程同时工作。采用标记清除算法。该收集器分为初始标记、并发标记、并发预清理、并发清除、并发重置这么几个步骤。

- 初始标记：暂停其他线程(stop the world)，标记与GC roots直接关联的对象。
- 并发标记：可达性分析过程(程序不会停顿)。
- 并发预清理：查找执行并发标记阶段从年轻代晋升到老年代的对象，重新标记，暂停虚拟机（stop the world）扫描CMS堆中剩余对象。
- 并发清除：清理垃圾对象，(程序不会停顿)。
- 并发重置，重置CMS收集器的数据结构。

简述G1垃圾收集器

和Serial、Parallel Scavenge、CMS不同，G1垃圾收集器把堆划分成多个大小相等的独立区域（Region），新生代和老年代不再物理隔离。通过引入 Region 的概念，从而将原来的一整块内存空间划分成多个的小空间，使得每个小空间可以单独进行垃圾回收。

- 初始标记：标记与GC roots直接关联的对象。
- 并发标记：可达性分析。
- 最终标记：对并发标记过程中，用户线程修改的对象再次标记一下。
- 筛选回收：对各个Region的回收价值和成本进行排序，然后根据用户所期望的GC停顿时间制定回收计划并回收。

简述Minor GC

Minor GC指发生在新生代的垃圾收集，因为 Java 对象大多存活时间短，所以 Minor GC 非常频繁，一般回收速度也比较快。

简述Full GC

Full GC 是清理整个堆空间—包括年轻代和永久代。调用System.gc(),老年代空间不足，空间分配担保失败，永生代空间不足会产生full gc。

常见内存分配策略

大多数情况下对象在新生代 Eden 区分配，当 Eden 没有足够空间时将发起一次 Minor GC。

大对象需要大量连续内存空间，直接进入老年代区分配。

如果经历过第一次 Minor GC 仍然存活且能被 Survivor 容纳，该对象就会被移动到 Survivor 中并将年龄设置为 1，并且每熬过一次 Minor GC 年龄就加 1 ，当增加到一定程度（默认15）就会被晋升到老年代。

如果在 Survivor 中相同年龄所有对象大小的总和大于 Survivor 的一半，年龄不小于该年龄的对象就可以直接进入老年代。

MinorGC 前，虚拟机必须检查老年代最大可用连续空间是否大于新生代对象总空间，如果满足则说明这次 Minor GC 确定安全。如果不，JVM会查看HandlePromotionFailure 参数是否允许担保失败，如果允许会继续检查老年代最大可用连续空间是否大于历次晋升老年代对象的平均大小，如果满足将Minor GC，否则改成一次 FullGC。

简述JVM类加载过程

1）加载：

- 通过全类名获取类的二进制字节流。
- 将类的静态存储结构转化为方法区的运行时数据结构。
- 在内存中生成类的Class对象，作为方法区数据的入口。

2）验证：对文件格式，元数据，字节码，符号引用等验证正确性。

3）准备：在方法区内为类变量分配内存并设置为0值。

4）解析：将符号引用转化为直接引用。

5）初始化：执行类构造器clinit方法，真正初始化。

简述JVM中的类加载器

- BootstrapClassLoader启动类加载器：加载/lib下的jar包和类。 由C++编写。
- ExtensionClassLoader扩展类加载器： /lib/ext目录下的jar包和类。由Java编写。
- AppClassLoader应用类加载器，加载当前classPath下的jar包和类。由Java编写。

简述双亲委派机制

一个类加载器收到类加载请求之后，首先判断当前类是否被加载过。已经被加载的类会直接返回，如果没有被加载，首先将类加载请求转发给父类加载器，一直转发到启动类加载器，只有当父类加载器无法完成时才尝试自己加载。

加载类顺序：BootstrapClassLoader->ExtensionClassLoader->AppClassLoader->CustomClassLoader 检查类是否加载顺序： CustomClassLoader->AppClassLoader->ExtensionClassLoader->BootstrapClassLoader

双亲委派机制的优点

- 避免类的重复加载。相同的类被不同的类加载器加载会产生不同的类，双亲委派保证了Java程序的稳定运行。
- 保证核心API不被修改。
- 如何破坏双亲委派机制
- 重载loadClass()方法，即自定义类加载器。

如何构建自定义类加载器

新建自定义类继承自java.lang.ClassLoader，重写findClass、loadClass、defineClass方法

JVM常见调优参数

- -Xms 初始堆大小
- -Xmx 最大堆大小
- -XX:NewSize 年轻代大小
- -XX:MaxNewSize 年轻代最大值
- -XX:PermSize 永生代初始值
- -XX:MaxPermSize 永生代最大值
- -XX:NewRatio 新生代与老年代的比例

也欢迎关注小牛的公众号：**后端技术小牛说**，不定期分享技术文和一线大厂内推机会。