

Java 中应该使用什么数据类型来代表价格?

如果不是特别关心内存和性能的话，使用 BigDecimal，否则使用预定义精度的 double 类型。

a = a + b 与 a += b 的区别

+ = 隐式的将加操作的结果类型强制转换为持有结果的类型。如果两个这个整型相加，如 byte、short 或者 int，首先会将它们提升到 int 类型，然后在执行加法操作。如果加法操作的结果比 a 的最大值要大，则 a+b 会出现编译错误，但是 a += b 没问题，如下：

```
byte a = 127;  
byte b = 127;  
b = a + b; // error : cannot convert from int to byte  
b += a; // ok
```

(译者注：这个地方应该表述的有误，其实无论 a+b 的值为多少，编译器都会报错，因为 a+b 操作会将 a、b 提升为 int 类型，所以将 int 类型赋值给 byte 就会编译出错)

String, StringBuffer 与 StringBuilder 的区别?

String 是字符串常量,final 修饰;StringBuffer 字符串变量(线程安全);StringBuilder 字符串变量(线程不安全)。

String 和 StringBuffer

String 和 StringBuffer 主要区别是性能: String 是不可变对象,每次对 String 类型进行操作都等同于产生了一个新的 String 对象,然后指向新的 String 对象.所以尽量不

在对 String 进行大量的拼接操作,否则会产生很多临时对象,导致 GC 开始工作,影响系统性能.

StringBuffer 是对对象本身操作,而不是产生新的对象,因此在通常在有大量拼接的情况下我们建议使用 StringBuffer.

但是需要注意现在 JVM 会对 String 拼接做一定的优化: String s= "This is only " +" simple" +" test" 会被虚拟机直接优化成 String s= "This is only simple test" ,此时就不存在拼接过程.

StringBuffer 和 StringBuilder

StringBuffer 是线程安全的可变字符串,其内部实现是可变数组.StringBuilder 是 java 5.0 新增的,其功能和 StringBuffer 类似,但是非线程安全.因此,在没有多线程问题的前提下,使用 StringBuilder 会取得更好的性能.

为什么重写 equals 一定要重写 hashCode

1. 使用 hashCode 方法提前校验, 可以避免每一次比对都调用 equals 方法, 提高效率
2. 保证是同一个对象, 如果重写了 equals 方法, 而没有重写 hashCode 方法, 会出现 equals 相等的对象, hashCode 不相等的情况, 重写 hashCode 方法就是为了避免这种情况的出现。

hashCode 的作用:

1. Java 对象的存储是存在内存当中, 可当我们查找一个对象时, 按照常规思路, 我们需要把要查找的对象与内存中的对象一一匹配.但是这样的匹配会占用较多的时间。
2. 所以, 加入我们把内存分为很多区域, 一个区域又可以存放很多对象, 在查找想要对

象时，我们只需要知道对象在那块区域，然后在那块区域和其他对象一一匹配就行，

这样极大节约了时间。

3. `hashCode()`方法就是计算在哪块区域的；`equals()`方法就是进行匹配
4. 重写了`equals()`方法就必须重写`hashCode()`方法。原因：如果我们只重写`equals()`方法。不重写`hashCode()`方法就无法计算得到我们要查找的对象和目标对象是否在同一块区域，没有在同一块区域计算机就没法调用`equals()`方法进行匹配。

构造器 Constructor 是否可被 override

在讲继承的时候我们就知道父类的私有属性和构造方法并不能被继承，所以`Constructor`也就不能被`override`,但是可以`overload`,所以你可以看到一个类中有多少个构造函数的情况。

抽象类必须要有抽象方法吗？

不需要，抽象类不一定非要有抽象方法。

示例代码：

```
abstract class Cat {  
    public static void sayHi() {  
        System.out.println("hi~");  
    }  
}
```

普通类和抽象类有哪些区别？

- 普通类不能包含抽象方法，抽象类可以包含抽象方法。
- 抽象类不能直接实例化，普通类可以直接实例化。

接口和抽象类有什么区别？

- 实现：抽象类的子类使用 `extends` 来继承；接口必须使用 `implements` 来实现接口。
- 构造函数：抽象类可以有构造函数；接口不能有。
- `main` 方法：抽象类可以有 `main` 方法，并且我们能运行它；接口不能有 `main` 方法。
- 实现数量：类可以实现很多个接口；但是只能继承一个抽象类。
- 访问修饰符：接口中的方法默认使用 `public` 修饰；抽象类中的方法可以是任意访问修饰符。
- 成员变量：接口只能声明 `public` 静态常量；抽象类没限制。
- 抽象类可以实现接口，接口只能继承接口

成员变量与局部变量的区别有那些？

1. 从语法形式上，看成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被 `public, private, static` 等修饰符所修饰，而局部变量不能被访问控制修饰符及 `static` 所修饰；成员变量和局部变量都能被 `final` 所修饰；
2. 从变量在内存中的存储方式来看，成员变量是对象的一部分，而对象存在于堆内存，局部变量存在于栈内存
3. 从变量在内存中的生存时间上看，成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动消失。
4. 成员变量如果没有被赋初值，则会自动以类型的默认值而赋值（**一种情况例外被 `final` 修饰但没有被 `static` 修饰的成员变量必须显示地赋值**）；而局部变量则不会自动赋值。

为什么要使用克隆？

想对一个对象进行处理，又想保留原有的数据进行接下来的操作，就需要克隆了，

Java 语言中克隆针对的是类的实例。

深拷贝和浅拷贝区别是什么？

- 浅拷贝只是复制了对象的引用地址，两个对象指向同一个内存地址，所以修改其中一个任意的值，另一个值都会随之变化，这就是浅拷贝（例：assign()）
- 深拷贝是将对象及值复制过来，两个对象修改其中任意的值另一个值不会改变，这就是深拷贝（例：JSON.parse()和 JSON.stringify()，但是此方法无法复制函数类型）

什么是反射？

反射主要是指程序可以访问、检测和修改它本身状态或行为的一种能力

Java 反射：

在 Java 运行时环境中，对于任意一个类，能否知道这个类有哪些属性和方法？对于任意一个对象，能否调用它的任意一个方法

Java 反射机制主要提供了以下功能：

- 在运行时判断任意一个对象所属的类。
- 在运行时构造任意一个类的对象。
- 在运行时判断任意一个类所具有的成员变量和方法。
- 在运行时调用任意一个对象的方法。

什么是反射？

主要是指程序可以访问、检测和修改它本身状态或行为的一种能力

JAVA 机制反射是在运行状态中，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一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制

什么是 java 序列化？什么情况下需要序列化？

简单说就是为了保存在内存中的各种对象的状态（也就是实例变量，不是方法），并且可以把保存的对象状态再读出来。虽然你可以用你自己的各种各样的方法来保存 object states，但是 Java 给你提供一种应该比你自己好的保存对象状态的机制，那就是序列化。

什么情况下需要序列化：

- a) 当你想把的内存中的对象状态保存到一个文件中或者数据库中时候；
- b) 当你想用套接字在网络上传送对象的时候；
- c) 当你想通过 RMI 传输对象的时候；

方法重载重写

关于重载和重写，你应该知道以下几点：

- 1、重载是一个编译期概念、重写是一个运行期间概念。
- 2、重载遵循所谓“编译期绑定”，即在编译时根据参数变量的类型判断应该调用哪个方法。
- 3、重写遵循所谓“运行期绑定”，即在运行的时候，根据引用变量所指向的实际对象的类型来调用方法。
- 4、因为在编译期已经确定调用哪个方法，所以重载并不是多态。而重写是多态。重载只是一种语言特性，是一种语法规则，与多态无关，与面向对象也无关。

(注：严格来说，重载是编译时多态，即静态多态。但是，Java 中提到的多态，在不特别说明的情况下都指动态多态)

重写的条件

参数列表必须完全与被重写方法的相同；返回类型必须完全与被重写方法的返回类型相同；

访问级别的限制性一定不能比被重写方法的强；访问级别的限制性可以比被重写方法的弱；

重写方法一定不能抛出新的检查异常或比被重写的方法声明的检查异常更广泛的检查异常重写的方法能够抛出更少或更有限的异常（也就是说，被重写的方法声明了异常，但重写的方法可以什么也不声明）不能重写被标示为 final 的方法；如果不能继承一个方法，则不能重写这个方法

重载的条士大夫是

被重载的方法必须改变参数列表；被重载的方法可以改变返回类型；被重载的方法可以改变访问修饰符；被重载的方法可以声明新的或更广的检查异常；方法能够在同一个类中或者在一个子类中被重载。

方法的重载：在同一个类中,出现多个同名的方法,参数列表不同,与返回值类型,修饰符无关

方法的重写：子类中出现和父类中一模一样的方法(包括返回值类型,方法名,参数列表)

方法重写的注意事项:

- 1.重写的方法必须要和父类一模一样(包括返回值类型,方法名,参数列表)
- 2.重写的方法可以使用@Override 注解来标识

3.子类中重写的方法的访问权限不能低于父类中方法的访问权限

权限修饰符 : private < 默认(什么都不写) < protected < public

为什么要重写方法:

1.当父类中的方法无法满足子类需求的时候,需要方法重写

2.当子类具有特有的功能的时候,就需要方法重写

重写的应用

子类可以根据需要, 定义特定于自己的行为。既沿袭了父类的功能名称, 又根据

子类的需要重新实现父类方法, 从而进行扩展增强。

静态类型与实际类型

```
public class test {  
  
    static class father {  
  
        void run() {  
  
            System.out.println("father run");  
  
        }  
    }  
  
    static class son extends father{  
  
        void run() {  
  
            System.out.println("son run");  
  
        }  
    }  
}
```

```
public void sayHello(son son) {  
    System.out.println("son");  
}  
  
public void sayHello(father father) {  
    System.out.println("father");  
}  
  
public static void main(String[] args) {  
    father a = new son();  
    a.run();  
    test t = new test();  
    t.sayHello(a);  
    System.out.println(a.getClass());  
}
```

输出结果：

son run

father

class old.test\$son

解释：

father a = new son()

这里面 father 是静态类型，son 是实际类型。静态类型是在编译期可知的，而实际类型是在运行期才可以知道，所以当运行 run() 时，取的是子类方法，而将 a 作为参数传入时，是以 father 这个类型传入的。这也说明了重载是静态的，而重写是动态的。

ArrayList

扩容机制

- ArrayList() 会使用长度为零的数组
- ArrayList(int initialCapacity) 会使用指定容量的数组
- public ArrayList(Collection<? extends E> c) 会使用 c 的大小作为数组容量
- add(Object o) 首次扩容为 10，再次扩容为上次容量的 1.5 倍
- addAll(Collection c) 没有元素时，扩容为 Math.max(10, 实际元素个数)，有元素时为 Math.max(原容量 1.5 倍, 实际元素个数)

fail-fast 与 fail-safe

- ArrayList 是 fail-fast 的典型代表，遍历的同时不能修改，尽快失败
- CopyOnWriteArrayList 是 fail-safe 的典型代表，遍历的同时可以修改，原理是读写分离

```
public class Test8 {  
  
    public static void main(String[] args) {  
  
        List<String> list = new ArrayList<>();  
  
        list.add("A");  
  
        list.add("B");  
  
        list.add("C");  
    }  
}
```

```
list.add("A");

//Exception in thread "main" java.util.ConcurrentModificationException

for (String s :list) {

    list.remove(s);

}

for (int i = 0; i < list.size(); i++) {

    list.remove(list.get(i));

    list.remove(i);

    i--;

}

Iterator<String> iterator= list.iterator();

while (iterator.hasNext()){

    String next = iterator.next();

    // Exception in thread "main" java.util.ConcurrentModificationException

    list.remove(next);

    //可以删除

    iterator.remove();

}
```

```
        System.out.println(list);
    }
}
```

ArrayList 与 LinkedList 的比较

ArrayList

- a) 基于数组，需要连续内存
- b) 随机访问快（指根据下标访问）
- c) 尾部插入、删除性能可以，其它部分插入、删除都会移动数据，因此性能会低
- d) 可以利用 cpu 缓存，局部性原理

LinkedList

- e) 基于双向链表，无需连续内存
- f) 随机访问慢（要沿着链表遍历）
- g) 头尾插入删除性能高
- h) 占用内存多

HashSet 有哪些特点

请问 HashSet 有哪些特点？

HashSet 实现自 set 接口，set 集合中元素无序且不能重复；

那么 HashSet 如何保证元素不重复？

因为 HashSet 底层是基于 HashMap 实现的，当你 new 一个 HashSet 时候，实际上是 new 了一个 map，执行 add 方法时，实际上调用 map 的 put 方法，value 始终是 PRESENT，所以根据 HashMap 的一个特性：将一个 key-value 对放入 HashMap 中时，

首先根据 key 的 hashCode() 返回值决定该 Entry 的存储位置，如果两个 key 的 hash 值相同，那么它们的存储位置相同。

如果这两个 key 的 equals 比较返回 true。那么新添加的 Entry 的 value 会覆盖原来的 Entry 的 value，key 不会覆盖。因此，如果向 HashSet 中添加一个已经存在的元素，新添加的集合元素不会覆盖原来已有的集合元素；

源码分析

先来看一下无参的构造函数：

```
public HashSet() {  
    map = new HashMap<>();  
}
```

很显然，当你 new 一个 HashSet 的时候，实际上是 new 了一个 HashMap

再来看一下 add 方法：

```
private static final Object PRESENT = new Object();  
public boolean add(E e) {  
    return map.put(e, PRESENT) == null;  
}
```

定义一个虚拟的 Object PRESENT 是向 map 中插入 key-value 对应的 value，因为 HashSet 中只需要用到 key，而 HashMap 是 key-value 键值对；所以，向 map 中添加键值对时，键值对的值固定是 PRESENT。

源码中 HashSet 的绝大部分方法都是通过调用 HashMap 的方法来实现的，其他的方法，就请大家自己查阅一下源码吧。

对于 HashSet 中保存的对象，请注意正确重写其 equals 和 hashCode 方法，以保证放入的对象的唯一性。

List 和 Set 比较，各自的子类比较

对比一：ArrayList 与 LinkedList 的比较

- 1、ArrayList 是实现了基于动态数组的数据结构,因为地址连续, 一旦数据存储好了, 查询操作效率会比较高 (在内存里是连着放的) 。
- 2、因为地址连续, ArrayList 要移动数据, 所以插入和删除操作效率比较低。
- 3、LinkedList 基于链表的数据结构, 地址是任意的, 所以在开辟内存空间的时候不需要等一个连续的地址, 对于新增和删除操作 add 和 remove, LinkedList 比较占优势。
- 4、因为 LinkedList 要移动指针, 所以查询操作性能比较低。

适用场景分析:

当需要对数据进行对此访问的情况下选用 ArrayList, 当需要对数据进行多次增加删除修改时采用 LinkedList。

对比二：ArrayList 与 Vector 的比较

- 1、Vector 的方法都是同步的, 是线程安全的, 而 ArrayList 的方法不是, 由于线程的同步必然要影响性能。因此, ArrayList 的性能比 Vector 好。
- 2、当 Vector 或 ArrayList 中的元素超过它的初始大小时, Vector 会将它的容量翻倍, 而 ArrayList 只增加 50% 的大小, 这样。ArrayList 就有利于节约内存空间。
- 3、大多数情况不使用 Vector, 因为性能不好, 但是它支持线程的同步, 即某一时刻只有一个线程能够写 Vector, 避免多线程同时写而引起的不一致性。
- 4、Vector 可以设置增长因子, 而 ArrayList 不可以。

适用场景分析:

- 1、Vector 是线程同步的，所以它也是线程安全的，而 ArrayList 是线程异步的，是不安全的。如果不考虑到线程的安全因素，一般用 ArrayList 效率比较高。
- 2、如果集合中的元素的数目大于目前集合数组的长度时，在集合中使用数据量比较大的数据，用 Vector 有一定的优势。

对比三： HashSet 与 TreeSet 的比较

1.TreeSet 是二叉树实现的， TreeSet 中的数据是自动排好序的，不允许放入 null 值。

2.HashSet 是哈希表实现的， HashSet 中的数据是无序的，可以放入 null，但只能放入一个 null，两者中的值都不能重复，就如数据库中唯一约束。

3.HashSet 要求放入的对象必须实现 GetHashCode()方法，放入的对象，是以 hashCode 码作为标识的，而具有相同内容的 String 对象， hashCode 是一样的，所以放入的内容不能重复。但是同一个类的对象可以放入不同的实例。

适用场景分析：

HashSet 是基于 Hash 算法实现的，其性能通常都优于 TreeSet。我们通常都应该使用 HashSet，在我们需要排序的功能时，我们才使用 TreeSet。

HashMap

底层数据结构，1.7 与 1.8 有何不同？

- 1.7 数组 + 链表，1.8 数组 + (链表 | 红黑树)

为何要用红黑树，为何一上来不树化，树化阈值为何是 8，何时会树化，何时会退化为链表？

- 红黑树用来避免 DoS 攻击，防止链表超长时性能下降，树化应当是偶然情况

- hash 表的查找，更新的时间复杂度是 $O(1)$ ，而红黑树的查找，更新的时间复杂度是 $O(\log_2 n)$ ，TreeNode 占用空间也比普通 Node 的大，如非必要，尽量还是使用链表。
 - hash 值如果足够随机，则在 hash 表内按泊松分布，在负载因子 0.75 的情况下，长度超过 8 的链表出现概率是 0.00000006，选择 8 就是为了让树化几率足够小
- 树化两个条件：链表长度超过树化阈值；数组容量 $>= 64$
 - 退化情况 1：在扩容时如果拆分树时，树元素个数 $<= 6$ 则会退化链表
 - 退化情况 2：remove 树节点时，若 root、root.left、root.right、root.left.left 有一个为 null，也会退化为链表

索引如何计算？ hashCode 都有了，为何还要提供 hash() 方法？数组容量为何是 2 的 n 次幂？

- 计算对象的 hashCode()，再进行调用 HashMap 的 hash() 方法进行二次哈希，最后 $\& (capacity - 1)$ 得到索引
- **二次 hash() 是为了综合高位数据，让哈希分布更为均匀**
- 计算索引时，如果是 2 的 n 次幂可以使用位与运算代替取模，效率更高；扩容时 $hash \& oldCap == 0$ 的元素留在原来位置，否则新位置 = 旧位置 + oldCap
- 但 ①、②、③ 都是为了配合容量为 2 的 n 次幂时的优化手段，例如 Hashtable 的容量就不是 2 的 n 次幂，并不能说哪种设计更优，应该是设计者综合了各种因素，最终选择了使用 2 的 n 次幂作为容量

介绍一下 put 方法流程，1.7 与 1.8 有何不同？

1. HashMap 是懒惰创建数组的，首次使用才创建数组
2. 计算索引（桶下标）

3. 如果桶下标还没人占用，创建 Node 占位返回
4. 如果桶下标已经有人占用
 - a) 已经是 TreeNode 走红黑树的添加或更新逻辑
 - b) 是普通 Node，走链表的添加或更新逻辑，如果链表长度超过树化阈值，走树化逻辑
5. 返回前检查容量是否超过阈值，一旦超过进行扩容
6. 不同
 - a) 链表插入节点时，1.7 是头插法，1.8 是尾插法
 - b) 1.7 是大于等于阈值且没有空位时才扩容，而 1.8 是大于阈值就扩容
 - c) 1.8 在扩容计算 Node 索引时，会优化

加载因子为何默认是 0.75f

- 在空间占用与查询时间之间取得较好的权衡
- 大于这个值，空间节省了，但链表就会比较长影响性能
- 小于这个值，冲突减少了，但扩容就会更频繁，空间占用多

多线程下会有啥问题？

1. 扩容死链 (1.7)
2. 数据错乱 (1.7, 1.8)

key 能否为 null，作为 key 的对象有什么要求？

1. HashMap 的 key 可以为 null，但 Map 的其他实现则不然
2. 作为 key 的对象，必须实现 hashCode 和 equals，并且 key 的内容不能修改（不可变）

String 对象的 hashCode() 如何设计的，为啥每次乘的是 31

- 目标是达到较为均匀的散列效果，每个字符串的 hashCode 足够独特
1. 字符串中的每个字符都可以表现为一个数字，称为 S_i ，其中 i 的范围是 $0 \sim n - 1$
 2. 散列公式为： $S_0 * 31^{n-1} + S_1 * 31^{n-2} + \dots + S_{i-1} * 31^{n-1-i} + \dots$
 $\llbracket S \rrbracket_{(n-1)} \llbracket *31 \rrbracket^{n-1}$
 3. 31 代入公式有较好的散列特性，并且 $31 * h$ 可以被优化为
 - a) 即 $32 * h - h$
 - b) 即 $2^5 * h - h$
 - c) 即 $h \ll 5 - h$

如何避免哈希碰撞

关于哈希碰撞有许多种方法，这里主要针对 HashMap 源码，介绍 HashMap 的避免哈希碰撞的思路

hash(key)函数

我们观察到，在 put() 函数到 putVal() 函数中，会传入 hash(key) 这个参数，那么我们来看看 hash() 这个函数在做些什么

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

注释： $>>>$ 表示右移取高位的操作， $>>> 16$ 表示取高十六位的操作

我们观察到，调用了 key.hashCode() 函数生成哈希值，并与哈希值自身的高十六位进行异或运算；读到这里读者可能会和笔者一样产生疑问，既然这个 hash(key) 是为了得到 key 值的哈希值，那么 key.hashCode() 已经返回了哈希值，又为何要与哈希值自身的高十六位

进行异或运算呢？

先说答案：

- 为了让得到的哈希值更加散列化。

这里我们这样进行解释，在确定加入项在哈希表中对应的位置时，我们通常采用`&`的操作来完成取余的操作，而通常情况下，我们采用十六位掩码与加入项进行`&`操作确定加入项在哈希值表中对应的位置。

若不进行上文中提到的哈希值与哈希值自身的高十六位进行异或运算，则始终是取哈希值的低十六位与十六位掩码进行`&`操作，哈希值高出十六位的部分始终无法参与运算，更容易发生哈希碰撞。

而这里，哈希值与哈希值自身的高出十六位的部分进行异或运算，我们假设存在某种情况下，两个哈希值的低十六位相同，但是高出十六位的部分不同，也能运算产生不同的结果，从而被摆放到哈希表不同的位置上，使得产出的结果更加趋向散列化。

- 2^n 次幂的与操作

众所周知，我们通过 $x \& (n-1)$ 操作完成 x 取余操作从而得到映射位置，如，我们要得到 $18 \bmod 16$ 的结果，则是将 18 与 $n-1$ 即 15 进行`&`操作，如下：

在这里的`&`操作一定要取 2^n 次幂减 1，也就是得到每一位上全部为 1 的数，因为如果 x 要进行`&`操作的树的位上存在不为 1 的数，则无论 x 的某一位上为 1 或者为 0，得到的最终结果都只能为 0，这样更容易让产生的树趋向于一个数，从而容易导致哈希碰撞如以下具体的计算过程：

```
1011 & 101 = 0001
```

```
1001 & 101 = 0001
```

为此，我们在初始化一个 `HashMap` 的 `table` 表项长度，输入我们的理想 `HashMap`

的 table 表项的长度时， hashMap 总是会将长度扩容至最接近该数的一个 2 次幂的数值，方便避免后续操作产生的哈希碰撞等问题

Java 中几种 Map 的顺序

- HashMap 的排序是无序的
- TreeMap 按照 key 的顺序进行排序的
- LinkedHashMap 按照 key 的插入的顺序排序的

HashMap 为什么冲突超过 8 才将链表转为红黑树而不直接用红黑树？

- 默认使用链表， 链表占用的内存更小
- 正常情况下，想要达到冲突为 8 的几率非常小，如果真的发生了转为红黑树可以保证极端情况下的效率

HashMap 真的是大于 8 就转换成红黑树， 小于 6 就变成链表吗

当链表长度大于 8 并且数组长度大于 64 时，才会转换为红黑树。

如果链表长度大于 8，但是数组长度小于 64 时，还是会进行扩容操作，不会转换为红黑树。因为数组的长度较小，应该尽量避开红黑树。因为红黑树需要进行左旋，右旋，变色操作来保持平衡，所以当数组长度小于 64，使用数组加链表比使用红黑树查询速度要更快、效率要更高。

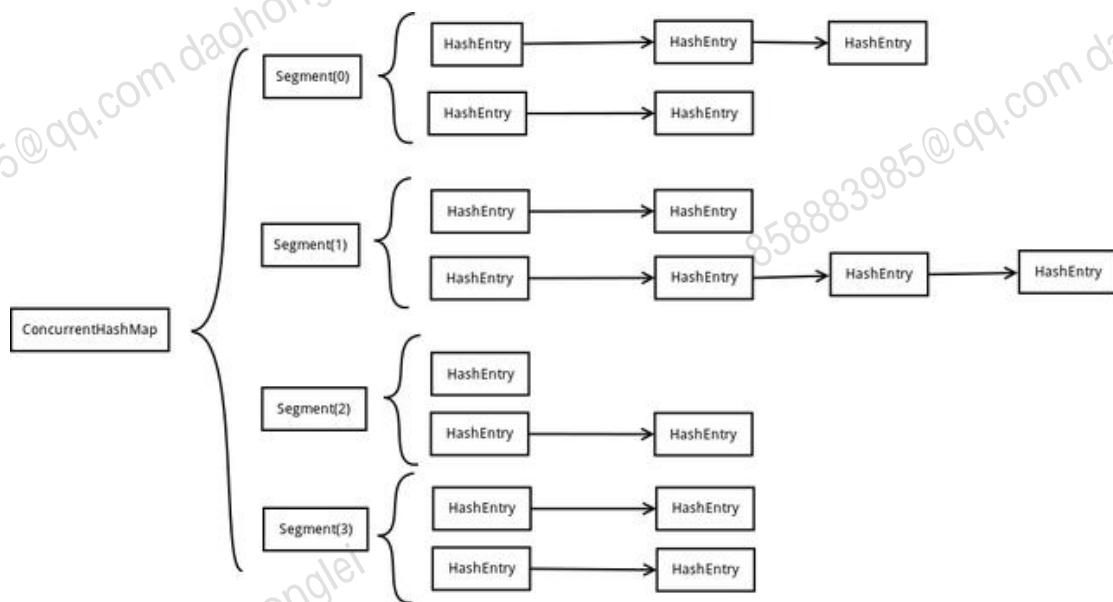
不是元素小于 6 的时候一定会变成链表，只有 resize 的时候才会根据 UNTREEIFY_THRESHOLD 进行转换

ConcurrentHashMap 在 jdk1.7 和 jdk1.8 中的不同

jdk 1.7 底层结构是：数组（Segment） + 链表（HashEntry 节点）组成

使用分段锁技术，将整个数据结构分段（默认为 16 段）进行存储，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。如下图是

ConcurrentHashMap 的内部结构图：



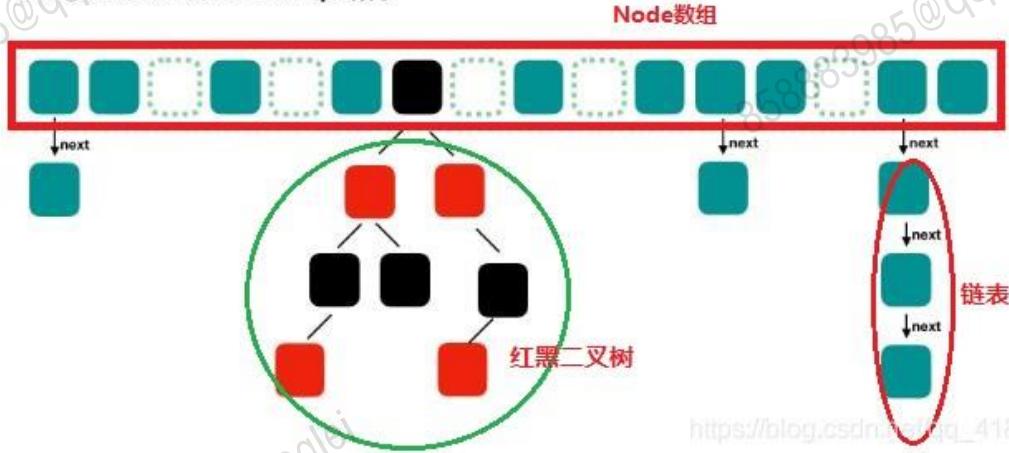
从上面的结构我们可以了解到，ConcurrentHashMap 定位一个元素的过程需要进行两次 Hash 操作。

第一次 Hash 定位到 Segment，第二次 Hash 定位到元素所在的链表的头部。

jdk1.8 底层结构是：Node 数组+链表 / 红黑树 其实就是 JDK 1.8 中的 hashMap 结构

Node 数组用来存放树或者链表的头结点，当一个链表中的数量到达一个数目时，会使查询速率降低，所以到达一定阈值时，会将一个链表转换为一个红黑二叉树，通告查询的速率。相互转化跟 JDK1.8 版本的 hashMap 保持一致

Java8 ConcurrentHashMap 结构



https://blog.csdn.net/qq_41884976

这两个的不同点太多了...，既包括了HashMap中的不同点，也有其他不同点，比如：

1. JDK8中没有分段锁了，而是使用synchronized来进行控制
2. JDK8中的扩容性能更高，支持多线程同时扩容，实际上JDK7中也支持多线程扩容，因为JDK7中的扩容是针对每个Segment的，所以也可能多线程扩容，但是性能没有JDK8高，因为JDK8中对于任意一个线程都可以去帮助扩容
3. JDK8中的元素个数统计的实现也不一样了，JDK8中增加了CounterCell来帮助计数，而JDK7中没有，JDK7中是put的时候每个Segment内部计数，统计的时候是遍历每个Segment对象加锁统计

ConcurrentHashMap 和 Hashtable 的区别？

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。

底层数据结构：

JDK1.7 的 ConcurrentHashMap 底层采用 分段的数组+链表 实现，JDK1.8 采用的数据结构跟 HashMap1.8 的结构一样，数组+链表/红黑二叉树。Hashtable 是采用 数组+链表 的形式。

实现线程安全的方式（重要）：

① 在 JDK1.7 的时候，ConcurrentHashMap（分段锁）对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。到了 JDK1.8 的时候已经摒弃了 Segment 的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。

② Hashtable(同一把锁) : 使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

JDK1.8 中的 ConcurrentHashMap 是如何保证线程安全的？

主要利用Unsafe操作+synchronized关键字。

Unsafe操作的使用仍然和JDK7中的类似，主要负责并发安全的修改对象的属性或数组某个位置的值。

synchronized主要负责在需要操作某个位置时进行加锁（该位置不为空），比如向某个位置的链表进行插入结点，向某个位置的红黑树插入结点。

JDK8中其实仍然有分段锁的思想，只不过JDK7中段数是可以控制的，而JDK8中是数组的每一个位置都有一把锁。

当向ConcurrentHashMap中put一个key,value时，

1. 首先根据key计算对应的数组下标i，如果该位置没有元素，则通过自旋的方法去向该位置赋值。
2. 如果该位置有元素，则synchronized会加锁
3. 加锁成功之后，在判断该元素的类型
 - a. 如果是链表节点则进行添加节点到链表中
 - b. 如果是红黑树则添加节点到红黑树
4. 添加成功后，判断是否需要进行树化
5. addCount，这个方法的意思是ConcurrentHashMap的元素个数加1，但是这个操作也是需要并发安全的，并且元素个数加1成功后，会继续判断是否要进行扩容，如果需要，则会进行扩容，所以这个方法很重要。
6. 同时一个线程在put时如果发现当前ConcurrentHashMap正在进行扩容则会去帮助扩容。

模板 2：

储存 Map 数据的数据时被 volatile 关键字修饰，一旦被修改，其他线程就可见修改。

因为是数组存储，所以只有改变数组内存值是才会触发 volatile 的可见性

如果 put 操作时 hash 计算出的槽点内没有值，采用自旋+CAS 保证 put 一定成功，且不会覆盖其他线程 put 的值

如果 put 操作时节点正在扩容，即发现槽点为转移节点，会等待扩容完成后再进行 put 操作，保证扩容时老数组不会变化

对槽点进行操作时会锁住槽点，保证只有当前线程能对槽点上的链表或红黑树进行操

作

红黑树旋转时会锁住根节点，保证旋转时线程安全

为什么 ConcurrentHashMap 的 key value 不能为 null, map 可以?

源码

```
if (key == null || value == null) throw new NullPointerException();
```

二义性

假定 ConcurrentHashMap 也可以存放 value 为 null 的值。那不管是 HashMap 还是 ConcurrentHashMap 调用 map.get(key) 的时候，如果返回了 null，那么这个 null，
都有两重含义：

- 1.这个 key 从来没有在 map 中映射过。
- 2.这个 key 的 value 在设置的时候，就是 null。

为什么 map 允许 value=null

对于 HashMap 的正确使用场景是在单线程下使用。

在单线程中，当我们得到的 value 是 null 的时候，我可以使用 hashMap.containsKey(key)
方法来区分上面说的两重含义。

所以当 map.get(key) 返回的值是 null，在 HashMap 中虽然存在二义性，但是结合
containsKey 方法可以避免二义性。

为什么 ConcurrentHashMap 不允许

ConcurrentHashMap 的使用场景为多线程。

用反证法来推理，假设 concurrentHashMap 允许存放值为 null 的 value。

这时有 A、B 两个线程。线程 A 调用 concurrentHashMap.get(key)方法,返回为 null, 我们还是不知道这个 null 是没有映射的 null 还是存的值就是 null。我们假设此时返回为 null 的真实情况就是因为这个 key 没有在 map 里面映射过。那么我们可以用 concurrentHashMap.containsKey(key)来验证我们的假设是否成立, 我们期望的结果是返回 false。

但是在我们调用 concurrentHashMap.get(key)方法之后, containsKey 方法之前, 有一个线程 B 执行了 concurrentHashMap.put(key,null)的操作。那么我们调用 containsKey 方法返回的就是 true 了。这就与我们的假设的真实情况不符合了。也就是上面说的**二义性**。

扩容期间在未迁移到的 hash 桶插入数据会发生什么?

只要插入的位置扩容线程还未迁移到, 就可以插入, 当迁移到该插入的位置时, 就会阻塞等待插入操作完成再继续迁移。

扩容过程中, 读访问能否访问到数据? 怎么实现的?

可以的。当数组在扩容的时候, 会对当前操作节点进行判断, 如果当前节点还没有被设置成 fwd 节点, 那就可以进行读写操作, 如果该节点已经被处理了, 那么当前线程也会加入到扩容的操作中去。

并发情况下, 各线程中的数据可能不是最新的, 那为什么 get 方法不需要加锁?

get 操作全程不需要加锁是因为 Node 的成员 val 是用 volatile 修饰的, 在多线程环境下线程 A 修改结点的 val 或者新增节点的时候是对线程 B 可见的。

HashMap 和 ConcurrentHashMap 的区别

- 1、HashMap 不是线程安全的, 而 ConcurrentHashMap 是线程安全的。

2、**ConcurrentHashMap 采用锁分段技术**，将整个 Hash 桶进行了分段 segment，也就是将这个大的数组分成了几个小的片段 segment，而且每个小的片段 segment 上面都有锁存在，那么在插入元素的时候就需要先找到应该插入到哪一个片段 segment，然后再在这个片段上面进行插入，而且这里还需要获取 segment 锁。

3、**ConcurrentHashMap 让锁的粒度更精细一些，并发性能更好。**

Map 集合遍历日常开发最常使用，简单总结五种方法差异。

①、**Iterator+entrySet 写法【推荐 JDK8 以下】**， Map.Entry 是 Map 接口的内部接口，获取迭代器，然后依次取出每个迭代器里面的 Map.Entry

```
Iterator<Map.Entry<Integer,String>> iterator=map.entrySet().iterator();

while(iterator.hasNext()){

    Map.Entry<Integer,String> entry=iterator1.next();

    System.out.println(entry.getKey());

    System.out.println(entry.getValue());

}
```

②、**Iterator+keyset 写法【不推荐，只能获取 key，然后通过 key 获取对应的 value，重复计算】**

```
Iterator<Integer> iterator=map.keySet().iterator();

while (iterator.hasNext()){

    Integer key=iterator.next();

    System.out.println(key);
```

```
        System.out.println(map.get(key));  
    }  
}
```

③、foreach 遍历方式【JDK8 以下推荐写法】

```
for(Map.Entry<Integer,String> entry:map.entrySet()){  
  
    System.out.println(entry.getKey());  
  
    System.out.println(entry.getValue());  
  
};
```

④：lambda 表达式遍历【JDK8 推荐写法，简捷】

```
map.forEach((key,value)->{  
  
    System.out.println(key);  
  
    System.out.println(value);  
  
});
```

⑤：stream 流遍历 Map【JDK8 不推荐写法，重复计算】

```
map.entrySet().stream().forEach((Map.Entry<Integer, String> entry) -> {  
  
    System.out.println(entry.getKey());  
  
    System.out.println(entry.getValue());  
  
});
```

如果 Map 集合存在一些中间处理，可以过滤操作，使用流式遍历也很方便。

Java 泛型的类型擦除

Java 泛型这个特性是从 JDK 1.5 才开始加入的，因此为了兼容之前的版本，Java 泛型的实现采取了“伪泛型”的策略，即 Java 在语法上支持泛型，但是在编译阶段会进行所

谓的“类型擦除”（Type Erasure），将所有的泛型表示（尖括号中的内容）都替换为具体的类型（其对应的原生态类型），就像完全没有泛型一样。理解类型擦除对于用好泛型是很有帮助的，尤其是一些看起来“疑难杂症”的问题，弄明白了类型擦除也就迎刃而解了。

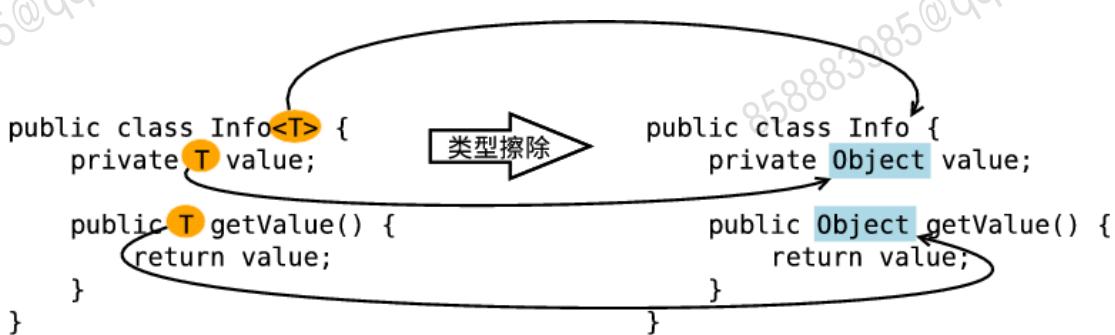
泛型的类型擦除原则是

- 消除类型参数声明，即删除<>及其包围的部分。
- 根据类型参数的上下界推断并替换所有的类型参数为原生态类型：如果类型参数是无限制通配符或没有上下界限定则替换为 Object，如果存在上下界限定则根据子类替换原则取类型参数的最左边限定类型（即父类）。
- 为了保证类型安全，必要时插入强制类型转换代码。
- 自动产生“桥接方法”以保证擦除类型后的代码仍然具有泛型的“多态性”。

1 擦除类定义中的类型参数

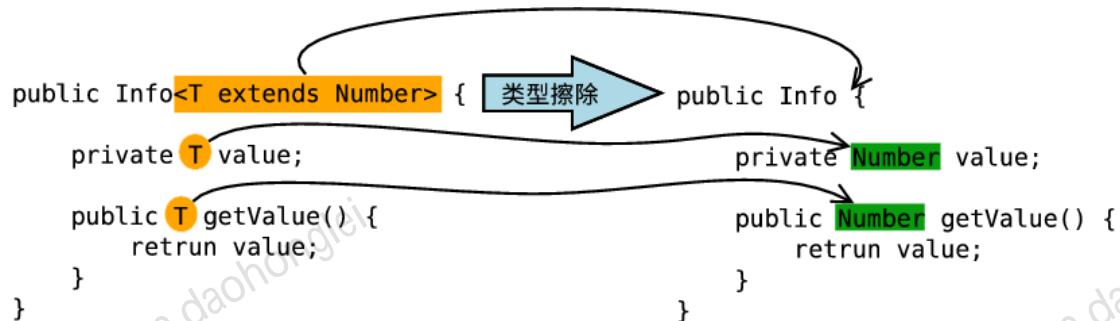
1.1 无限制类型擦除

当类定义中的类型参数没有任何限制时，在类型擦除中直接被替换为 Object，即形如<T>和<?>的类型参数都被替换为 Object。



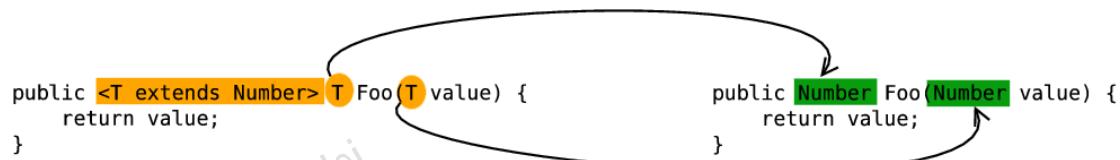
1.2 有限制类型擦除

当类定义中的类型参数存在限制（上下界）时，在类型擦除中替换为类型参数的上界或者下界，比如形如<T extends Number>和<? extends Number>的类型参数被替换为Number，<? super Number>被替换为Object。



2 擦除方法定义中的类型参数

擦除方法定义中的类型参数原则和擦除类定义中的类型参数是一样的，这里仅以擦除方法定义中的有限制类型参数为例。



BIO、NIO 和 AIO 的区别

Java BIO：同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

Java NIO：同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有 I/O 请求时才启动一个线程进行处理。

Java AIO：异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的

I/O 请求都是由 OS 先完成了再通知服务器应用去启动线程进行处理。

NIO 比 BIO 的改善之处是把一些无效的连接挡在了启动线程之前，减少了这部分资源的浪费（因为我们都知道每创建一个线程，就要为这个线程分配一定的内存空间）

AIO 比 NIO 的进一步改善之处是将一些暂时可能无效的请求挡在了启动线程之前，比如在 NIO 的处理方式中，当一个请求来的话，开启线程进行处理，但这个请求所需要的资源还没有就绪，此时必须等待后端的应用资源，这时线程就被阻塞了。

适用场景分析：

BIO 方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4 以前的唯一选择，但程序直观简单易理解，如之前在 Apache 中使用。

NIO 方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4 开始支持，如在 Nginx，Netty 中使用。

AIO 方式适用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用 OS 参与并发操作，编程比较复杂，JDK7 开始支持，在成长中，Netty 曾经使用过，后来放弃。

线程和进程的区别？

简而言之，进程是程序运行和资源分配的基本单位，一个程序至少有一个进程，一个进程至少有一个线程。进程在执行过程中拥有独立的内存单元，而多个线程共享内存资源，减少切换次数，从而效率更高。线程是进程的一个实体，是 cpu 调度和分

派的基本单位，是比程序更小的能独立运行的基本单位。同一进程中的多个线程之间可以并发执行。

多线程作用

1、利用多核 cpu

现在的服务器大多都是多核 cpu (双核、4 核、8 核等)，如果程序只跑单线程，就会浪费 cpu 资源。只有跑多线程时，多核 cpu 才能有效利用，它能让多段逻辑同时执行。

单核 cpu 也可以跑多线程，不过是“假的”，同一时间处理器只会处理一段逻辑，只不过线程切换比较快，看着像是多线程“同时”在运行。从程序运行效率的角度来看，单核 CPU 不但不会发挥出多线程的优势，反而会因为在单核 CPU 上运行多线程导致线程上下文的切换，而降低程序整体的效率。

2、防止阻塞

单核 CPU 我们还是要应用多线程，就是为了防止阻塞。试想，如果单核 CPU 使用单线程，那么只要这个线程阻塞了，比方说远程读取某个数据（或其他 IO 操作（磁盘，数据库，网络等等）），数据迟迟未返回又没有设置超时时间，那么你的整个程序在数据返回回来之前就停止运行（阻塞）了。多线程可以防止这个问题，多条线程同时运行，哪怕一条线程的代码执行读取数据阻塞，也不会影响其它任务的执行。

在单核 cpu 的情况下，多线程的这个消除阻塞的作用还可以叫做“并发”，这和并行是有着本质的不同的。并发是“伪并行”，看似并行，而实际上还是一个 CPU 在执行一切事物，只是切换的太快，我们没法察觉罢了。

3、建模

这是另外一个没有这么明显的优点了。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（任务相对独立，互不影响），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。

4、什么时候用多线程

4.1 线程越多越好吗？

线程必然不是越多越好，线程切换也是要开销的，当你增加一个线程的时候，增加的额外开销要小于该线程能够消除的阻塞时间，这才叫物有所值。

Linux 自从 2.6 内核开始，就会把不同的线程交给不同的核心去处理。

Windows 也从 NT.4.0 开始支持这一特性。

4.2 什么时候才有必要用多线程？

多核 CPU——计算密集型任务。此时要尽量使用多线程，可以提高任务执行效率，例如加密解密，数据压缩解压缩（视频、音频、普通数据），否则只能使一个核心满载，而其他核心闲置。

单核 CPU——计算密集型任务。此时的任务已经把 CPU 资源 100% 消耗了，就没必要也不可能使用多线程来提高计算效率了；相反，如果要做人机交互，最好还是要用多线程，避免用户没法对计算机进行操作。

单核 CPU——IO 密集型任务，使用多线程还是为了人机交互方便。

多核 CPU——IO 密集型任务，这就更不用说了，跟单核时候原因一样。

线程的 run() 和 start() 有什么区别

每个线程都是通过某个特定 Thread 对象所对应的方法 run() 来完成其操作的，方法 run() 称为线程体。通过调用 Thread 类的 start() 方法来启动一个线程。

start() 方法来启动一个线程，真正实现了多线程运行。这时无需等待 run 方法体代码执行完毕，可以直接继续执行下面的代码；这时此线程是处于就绪状态，并没有运行。然后通过此 Thread 类调用方法 run() 来完成其运行状态，这里方法 run() 称为线程体，它包含了要执行的这个线程的内容，Run 方法运行结束，此线程终止。然后 CPU 再调度其它线程。

run() 方法是在本线程里的，只是线程里的一个函数，而不是多线程的。如果直接调用 run()，其实就相当于是调用了一个普通函数而已，直接待用 run() 方法必须等待 run() 方法执行完毕才能执行下面的代码，所以执行路径还是只有一条，根本就没有线程的特征，所以在多线程执行时要使用 start() 方法而不是 run() 方法。

创建线程有哪几种方式？

①. 继承 Thread 类创建线程类

- 定义 Thread 类的子类，并重写该类的 run 方法，该 run 方法的方法体就代表了线程要完成的任务。因此把 run() 方法称为执行体。
- 创建 Thread 子类的实例，即创建了线程对象。
- 调用线程对象的 start() 方法来启动该线程。

②. 通过 Runnable 接口创建线程类

- 定义 runnable 接口的实现类，并重写该接口的 run() 方法，该 run() 方法的方法体同样是该线程的线程执行体。

- 创建 Runnable 实现类的实例，并依此实例作为 Thread 的 target 来创建 Thread 对象，该 Thread 对象才是真正的线程对象。

- 调用线程对象的 start()方法来启动该线程。

③. 通过 Callable 和 Future 创建线程

- 创建 Callable 接口的实现类，并实现 call()方法，该 call()方法将作为线程执行体，并且有返回值。
- 创建 Callable 实现类的实例，使用 FutureTask 类来包装 Callable 对象，该 FutureTask 对象封装了该 Callable 对象的 call()方法的返回值。
- 使用 FutureTask 对象作为 Thread 对象的 target 创建并启动新线程。
- 调用 FutureTask 对象的 get()方法来获得子线程执行结束后的返回值。

Runnable 和 Callable 有什么区别？

有点深的问题了，也看出一个 Java 程序员学习知识的广度。

- 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。

线程的几种状态总结

线程在一定条件下，状态会发生变化。线程一共有以下几种状态：

- 1、新建状态 (New)：新创建了一个线程对象。
- 2、就绪状态 (Runnable)：线程对象创建后，其他线程调用了该对象的 start() 方法。该状态的线程位于“可运行线程池”中，变得可运行，只等待获取 CPU 的使用权，即在就绪状态的进程除 CPU 之外，其它的运行所需资源都已全部获得。
- 3、运行状态 (Running)：就绪状态的线程获取了 CPU，执行程序代码。

4、阻塞状态(Blocked)：阻塞状态是线程因为某种原因放弃 CPU 使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

阻塞的情况分三种：

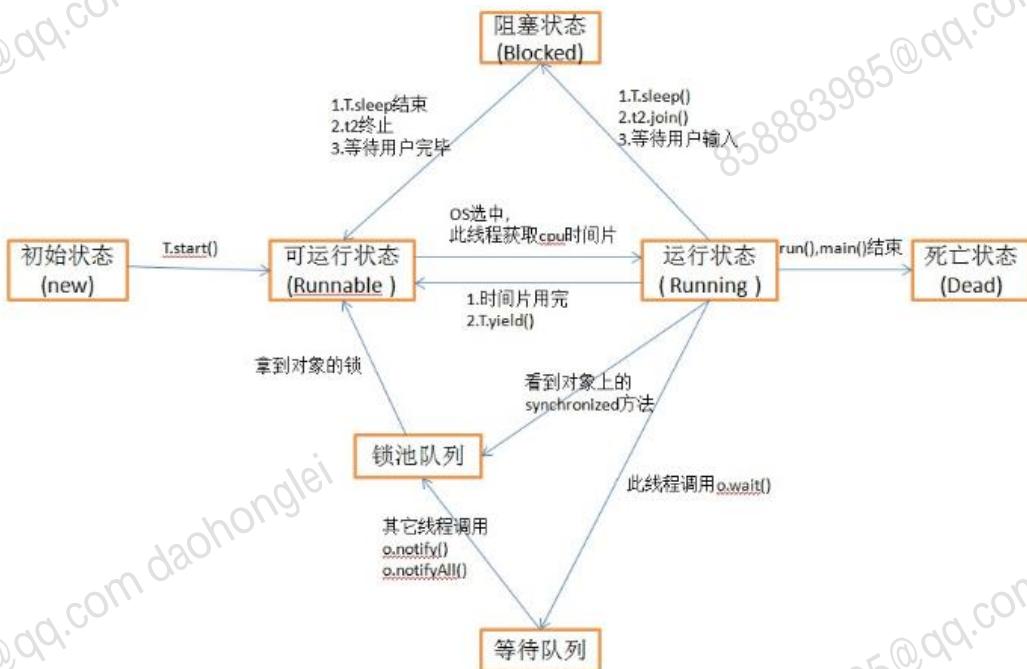
①. 等待阻塞：运行的线程执行 `wait()` 方法，该线程会释放占用的所有资源，JVM 会把该线程放入“等待池”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用 `notify()` 或 `notifyAll()` 方法才能被唤醒，

②. 同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则 JVM 会把该线程放入“锁池”中。

③. 其他阻塞：运行的线程执行 `sleep()` 或 `join()` 方法，或者发出了 I/O 请求时，JVM 会把该线程置为阻塞状态。当 `sleep()` 状态超时、`join()` 等待线程终止或者超时，或者 I/O 处理完毕时，线程重新转入就绪状态。

5、死亡状态(Dead)：线程执行完了或者因异常退出了 `run()` 方法，该线程结束生命周期。

线程变化的状态转换图如下：



在 Java 虚拟机 (HotSpot) 中，monitor 是由 ObjectMonitor 实现的，其主要数据结构

如下（位于 HotSpot 虚拟机源码 `ObjectMonitor.hpp` 文件，C++ 实现的）

```
ObjectMonitor() {  
    _header      = NULL;  
    _count       = 0; // 记录该线程获取锁的次数  
    _waiters     = 0; // 当前有多少处于 wait 状态的 thread  
    _recursions  = 0; // 锁的重入次数
```

```
_object      = NULL;  
  
_owner       = NULL;  
  
_WaitSet     = NULL; //处于 wait 状态的线程，会被加入到_WaitSet  
  
_WaitSetLock = 0 ;  
  
_Responsible = NULL ;  
  
_succ        = NULL ;  
  
_cxq         =NULL ;  
  
FreeNext     = NULL ;  
  
_EntryList   = NULL ; //处于等待锁 block 状态的线程，会被加入到该列表  
  
_SpinFreq    = 0 ;  
  
_SpinClock   = 0 ;  
  
OwnerIsThread = 0 ;  
  
}
```

ObjectMonitor 中有两个队列，_WaitSet 和 _EntryList，用来保存 ObjectWaiter 对象列表(每个等待锁的线程都会被封装成 ObjectWaiter 对象)，_owner 指向持有 ObjectMonitor 对象的线程，

当多个线程同时访问一段同步代码时，首先会进入 _EntryList 集合，当线程获取到对象的 monitor 后进入 _Owner 区域并把 monitor 中的 owner 变量设置为当前线程同时 monitor 中的计数器 count 加 1，

若线程调用 wait() 方法，将释放当前持有的 monitor，owner 变量恢复为 null，count 自减 1，同时该线程进入 WaitSe t 集合中等待被唤醒。

若当前线程执行完毕也将释放 monitor(锁)并复位变量的值，以便其他线程进入获取

monitor(锁)。如下图所示:



```
void ATTR ObjectMonitor::enter(TRAPS) {  
    // The following code is ordered to check the most common cases first  
    // and to reduce RTS->RTO cache line upgrades on SPARC and IA32 processors.  
  
    Thread * const Self = THREAD ;  
  
    void * cur ;  
  
    // 通过 CAS 尝试把 monitor 的_owner 字段设置为当前线程  
  
    cur = Atomic::cmpxchg_ptr (Self, &_owner, NULL) ;  
  
    if (cur == NULL) {  
  
        // Either ASSERT recursions == 0 or explicitly set recursions = 0.  
  
        assert (_recursions == 0 , "invariant") ;  
  
        assert (_owner == Self, "invariant") ;  
  
        // CONSIDER: set or assert OwnerIsThread == 1  
  
        return ;  
  
    }  
}
```

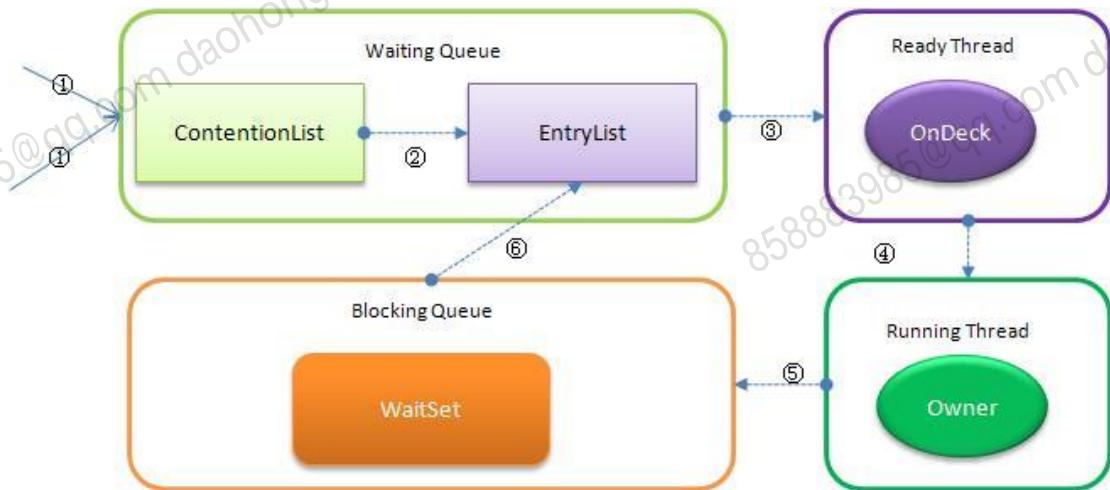
```
// 设置之前的 owner 指向当前线程，说明当前线程已经持有锁，此次为重入，  
_recursions 自增  
  
if (cur == Self) {  
  
    // TODO-FIXME: check for integer overflow! BUGID 6557169.  
  
    _recursions ++ ;  
  
    return ;  
  
}  
  
// 如果之前_owner 指向的 BasicLock 在当前线程栈上，说明当前线程是第一次进入该  
monitor  
  
// 设置 recursions 为 1， owner 为当前线程，该线程成功获得锁并返回  
  
if (Self->is_lock_owned ((address)cur)) {  
  
    assert (_recursions == 0, "internal state error");  
  
    _recursions = 1 ;  
  
    // Commute owner from a thread-specific on-stack BasicLockObject address to  
    // a full-fledged "Thread *".  
  
    _owner = Self ;  
  
    OwnerIsThread = 1 ;  
  
    return ;  
  
}
```

从上面代码中，可以看出，if (cur == Self)，说明设置之前的 owner 指向当前线程，说明当前线程已经持有锁，此次为重入，_recursions 自增，即 synchronized 是一把可重入锁

synchronized 还是一把非公平锁，新的线程进来是可以有抢占优先级的。

线程状态及状态转换

当多个线程同时请求某个对象监视器时，对象监视器会设置几种状态用来区分请求的线程：
Contention List: 所有请求锁的线程将被首先放置到该竞争队列 Entry List: Contention List 中那些有资格成为候选人的线程被移到 Entry List
Wait Set: 那些调用 wait 方法被阻塞的线程被放置到 Wait Set
OnDeck: 任何时刻最多只能有一个线程正在竞争锁，该线程称为 OnDeck Owner: 获得锁的线程
!Owner: 释放锁的线程



为什么要用线程池

那先要明白什么是线程池

线程池是指在初始化一个多线程应用程序过程中创建一个线程集合，然后在需要执行新的任务时重用这些线程而不是新建一个线程。

使用线程池的好处

- 1、线程池改进了一个应用程序的响应时间。由于线程池中的线程已经准备好了且等待被分配任务，应用程序可以直接拿来使用而不用新建一个线程。
- 2、线程池节省了 CLR 为每个短生存周期任务创建一个完整的线程的开销并可以在任务完成后回收资源。

- 3、线程池根据当前在系统中运行的进程来优化线程时间片。
- 4、线程池允许我们开启多个任务而不用为每个线程设置属性。
- 5、线程池允许我们为正在执行的任务的程序参数传递一个包含状态信息的对象引用。
- 6、线程池可以用来解决处理一个特定请求最大线程数量限制问题。

线程池都有哪几种工作队列

```
ExecutorService executorService = Executors.newFixedThreadPool(10);
```

在类 Executors 中，我们可以看到不同线程池维护的工作队列是不同的，

newCachedThreadPool 使用 SynchronousQueue 同步队列，

newFixedThreadPool 使用 LinkedBlockingQueue 链表型阻塞队列，

newSingleThreadExecutor 使用 LinkedBlockingQueue 链表型阻塞队列，

newScheduledThreadPool 使用 DelayQueue 延时队列，

newSingleThreadScheduledExecutor 使用 DelayQueue 延时队列，

它们都是实现了并发包 java.util.concurrent 中的 BlockingQueue，下面说说这个接口。

BlockingQueue 阻塞队列

继承于队列 Queue，遵循先进先出原则（FIFO），队列提供几种基本的操作，添加元素（队尾）、移除元素（队头）、取出队头元素（不移除），每种操作都有两个方法，一种有可能抛异常，一种返回操作成功或失败。

在这个基础上，阻塞队列增加了操作锁，保证了数据安全，当然这个具体实现是在子类中完成，接口仅仅描述方法的特点，还增加两种不同的操作实现。下面描述这

四种不同类型的操作：

操作，可能抛异常

操作，不抛异常（特殊如类转换异常、空指针、参数异常不属于，仅当队列已满
不会跑状态异常）

无限期阻塞线程直至操作成功

有时间限制的操作

添加移除检查

`add(e)remove(o)element()`

`offer(e)poll()peek()`

`put(e)take()\`

`offer(e,time, unit)poll(time, unit)\`

以及增加了拷贝 `drainTo`，如线程池的 `shutdownNow` 就是用它完成工作队列的
清除以及队列中数据的拷贝，还有其他如对比元素 `contains`，剩余容量查询
`remainingCapacity` 等。

实现的子类

`ArrayBlockingQueue` 数组型阻塞队列

`LinkedBlockingQueue` 链表型阻塞队列

`SynchronousQueue` 同步队列 必须先 `tabke`,然后在 `add`,本身不存储元素

`PriorityBlockingQueue` 优先阻塞队列 元素实现 `Comparable` 接口,每次
`tabke` 出最大或者最小

`DelayQueue` 延时队列 元素实现 `Delay` 接口,每次取出过期元素

`ArrayBlockingQueue`

特点:

初始化一定容量的数组

使用一个重入锁， 默认使用非公平锁， 入队和出队共用一个锁， 互斥

是有界设计， 如果容量满无法继续添加元素直至有元素被移除

使用时开辟一段连续的内存， 如果初始化容量过大容易造成资源浪费， 过小

易添加失败

LinkedBlockingQueue

特点:

内部使用节点关联， 会产生多一点内存占用

使用两个重入锁分别控制元素的入队和出队， 用 Condition 进行线程间的唤醒和等待

有边界的，在默认构造方法中容量是 Integer.MAX_VALUE

非连续性内存空间

DelayQueue

特点:

无边界设计

添加 (put) 不阻塞， 移除阻塞

元素都有一个过期时间

取元素只有过期的才会被取出

SynchronousQueue

特点:

内部容量是 0

每次删除操作都要等待插入操作

每次插入操作都要等待删除操作

一个元素，一旦有了插入线程和移除线程，那么很快由插入线程移交给移除

线程，这个容器相当于通道，本身不存储元素

在多任务队列，是最快的处理任务方式。

PriorityBlockingQueue

特点：

无边界设计，但容量实际是依靠系统资源影响

添加元素，如果超过 1，则进入优先级排序

DelayQueue

```
public class DelayTest {  
  
    private static DelayQueue<TaskInfo> queue = new DelayQueue<>(); // 延迟队列  
  
    private static ExecutorService es = Executors.newFixedThreadPool(10); // 3 个线  
程的线程池  
  
    public static void main(String[] args){  
  
        new Thread(()->{  
  
            while (true){  
  
                if (queue.size() <=0){  
  
                    getTask();  
  
                }  
  
            }  
        }  
    }  
}
```

```
}).start();

new Thread()->{

    while(true){

        try {

            TaskInfo task = queue.take();

            es.submit()->{

                System.out.println("执行任务: " + task.getId() + ":" + task.getData());

            });

        } catch (InterruptedException e) {

            throw new RuntimeException(e);

        }

    }

}).start();

}

//模拟从数据库获取 将来 10 秒中内即将执行的任务

public static void getTask(){

    Random r = new Random();

    int t = r.nextInt(2);

    if(t==0){

        return;

    }

}
```

```
queue.offer(new TaskInfo(1,1,"任务 1",1000));  
  
queue.offer(new TaskInfo(2,2,"任务 2",2000));  
  
queue.offer(new TaskInfo(3,3,"任务 3",3000));  
  
queue.offer(new TaskInfo(4,4,"任务 4",4000));  
  
queue.offer(new TaskInfo(5,5,"任务 5",5000));  
  
queue.offer(new TaskInfo(6,6,"任务 6",6000));  
  
queue.offer(new TaskInfo(7,7,"任务 7",7000));  
  
queue.offer(new TaskInfo(8,8,"任务 8",8000));  
  
}  
  
}  
  
public class TaskInfo implements Delayed {  
  
    private int id;  
  
    private int type;  
  
    private String data;  
  
    private long executeTime;  
  
    public TaskInfo(int id, int type, String data, long executeTime) {  
  
        this.id = id;  
  
        this.type = type;  
  
        this.data = data;  
  
        this.executeTime = TimeUnit.NANOSECONDS.convert(executeTime,  
TimeUnit.MILLISECONDS) + System.nanoTime();  
  
    }  
}
```

```
public int getId() {  
    return id;  
}  
  
public void setId(int id) {  
    this.id = id;  
}  
  
public int getType() {  
    return type;  
}  
  
public void setType(int type) {  
    this.type = type;  
}  
  
public String getData() {  
    return data;  
}  
  
public void setData(String data) {  
    this.data = data;  
}  
  
public long getExecuteTime() {  
    return executeTime;  
}  
  
public void setExecuteTime(long executeTime) {
```

```
this.executeTime = executeTime;

}

@Override

public long getDelay(TimeUnit unit) {

    return unit.convert(this.executeTime - System.nanoTime(),

TimeUnit.NANOSECONDS);

}

@Override

public int compareTo(Delayed o) {

    TaskInfo msg = (TaskInfo) o;

    return this.executeTime > msg.executeTime ? 1 : (this.executeTime <

msg.executeTime ? -1 : 0);

}

}
```

几种常见的线程池及使用场景

newCachedThreadPool:

- 底层：返回 ThreadPoolExecutor 实例， corePoolSize 为 0；
maximumPoolSize 为 Integer.MAX_VALUE； keepAliveTime 为 60L； unit 为
TimeUnit.SECONDS； workQueue 为 SynchronousQueue(同步队列)
- 通俗：当有新任务到来，则插入到 SynchronousQueue 中，由于
SynchronousQueue 是同步队列，因此会在池中寻找可用线程来执行，若有可以

线程则执行，若没有可用线程则创建一个线程来执行该任务；若池中线程空闲时间超过指定大小，则该线程会被销毁。

- 适用：执行很多短期异步的小程序或者负载较轻的服务器

newFixedThreadPool:

- 底层：返回 ThreadPoolExecutor 实例，接收参数为所设定线程数量 nThread，corePoolSize 为 nThread，maximumPoolSize 为 nThread；keepAliveTime 为 0L(不限时)；unit 为： TimeUnit.MILLISECONDS；WorkQueue 为： new LinkedBlockingQueue<Runnable>() 无解阻塞队列
- 通俗：创建可容纳固定数量线程的池子，每个线程的存活时间是无限的，当池子满了就不在添加线程了；如果池中的所有线程均在繁忙状态，对于新任务会进入阻塞队列中(无界的阻塞队列)

- 适用：执行长期的任务，性能好很多

newSingleThreadExecutor:

- 底层：FinalizableDelegatedExecutorService 包装的 ThreadPoolExecutor 实例，corePoolSize 为 1；maximumPoolSize 为 1；keepAliveTime 为 0L；unit 为： TimeUnit.MILLISECONDS；workQueue 为： new LinkedBlockingQueue<Runnable>() 无解阻塞队列
- 通俗：创建只有一个线程的线程池，且线程的存活时间是无限的；当该线程正繁忙时，对于新任务会进入阻塞队列中(无界的阻塞队列)

- 适用：一个任务一个任务执行的场景

newScheduledThreadPool:

- 底层：创建 ScheduledThreadPoolExecutor 实例， corePoolSize 为传递来的参

数, maximumPoolSize 为 Integer.MAX_VALUE; keepAliveTime 为 0; unit 为: TimeUnit.NANOSECONDS; workQueue 为: new DelayedWorkQueue() 一个按超时时间升序排序的队列

- 通俗: 创建一个固定大小的线程池, 线程池内线程存活时间无限制, 线程池可以支持定时及周期性任务执行, 如果所有线程均处于繁忙状态, 对于新任务会进入 DelayedWorkQueue 队列中, **这是一种按照超时时间排序的队列结构**
- **适用: 周期性执行任务的场景**

线程池任务执行流程:

1. 当线程池小于 corePoolSize 时, 新提交任务将创建一个新线程执行任务, 即使此时线程池中存在空闲线程。
2. 当线程池达到 corePoolSize 时, 新提交任务将被放入 workQueue 中, 等待线程池中任务调度执行
3. 当 workQueue 已满, 且 maximumPoolSize>corePoolSize 时, 新提交任务会创建新线程执行任务
4. 当提交任务数超过 maximumPoolSize 时, 新提交任务由 RejectedExecutionHandler 处理
5. 当线程池中超过 corePoolSize 线程, 空闲时间达到 keepAliveTime 时, 关闭空闲线程
6. 当设置 allowCoreThreadTimeOut(true)时, 线程池中 corePoolSize 线程空闲时间达到 keepAliveTime 也将关闭

备注:

一般如果线程池任务队列采用 LinkedBlockingQueue 队列的话, 那么不会拒

绝任何任务（因为队列大小没有限制），这种情况下，`ThreadPoolExecutor` 最多仅会按照最小线程数来创建线程，也就是说线程池大小被忽略了。

如果线程池任务队列采用 `ArrayBlockingQueue` 队列的话，那么 `ThreadPoolExecutor` 将会采取一个非常负责的算法，比如假定线程池的最小线程数为 4，最大为 8 所用的 `ArrayBlockingQueue` 最大为 10。随着任务到达并被放到队列中，线程池中最多运行 4 个线程（即最小线程数）。即使队列完全填满，也就是说有 10 个处于等待状态的任务，`ThreadPoolExecutor` 也只会利用 4 个线程。如果队列已满，而又有新任务进来，此时才会启动一个新线程，这里不会因为队列已满而拒接该任务，相反会启动一个新线程。新线程会运行队列中的第一个任务，为新来的任务腾出空间。

这个算法背后的理念是：该池大部分时间仅使用核心线程（4 个），即使有适量的任务在队列中等待运行。这时线程池就可以用作节流阀。如果挤压的请求变得非常多，这时该池就会尝试运行更多的线程来清理；这时第二个节流阀—最大线程数就起作用了。

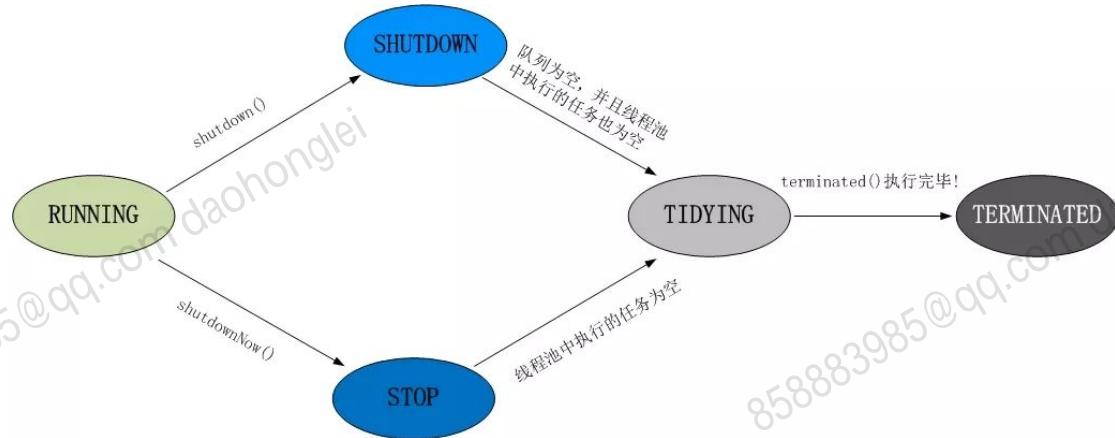
线程池拒绝策略

- 1) `AbortPolicy`: 默认的拒绝策略，始终抛出 `RejectedExecutionException`
- 2) `CallerRunsPolicy`: 提交任务的线程自己去执行该任务
- 3) `DiscardOldestPolicy`: 丢弃最老的任务，其实就是把最早进入工作队列的任务丢弃，然后把新任务加入到工作队列
- 4) `DiscardPolicy`: 相当大胆的策略，直接丢弃任务，没有任何异常抛出

线程池都有哪些状态？

线程池有 5 种状态：Running、ShutDown、Stop、Tidying、Terminated。

线程池各个状态切换框架图：



线程池中 submit() 和 execute() 方法有什么区别？

接收的参数不一样

submit 有返回值，而 execute 没有

submit 方便 Exception 处理

自定义线程池

```
public class Test4 implements Runnable{  
  
    private String name;  
  
    public Test4(String name) {  
  
        this.name = name;  
  
    }  
  
    public static void main(String[] args) throws Exception{  
  
    }
```

```
ThreadPoolExecutor threadPool = new ThreadPoolExecutor(1, 1, 60,
    TimeUnit.SECONDS, new ArrayBlockingQueue<>(10), r -> new
    Thread(r,"test"), (r, executor) -> {
        if (r instanceof Test4) {
            Test4 test4 = (Test4) r;
            System.out.println(test4.getName());
        }
    });
//允许核心线程被回收
threadPool.allowCoreThreadTimeOut(true);
threadPool.execute(new Test4("test1"));
threadPool.execute(new Test4("test2"));
threadPool.execute(new Test4("test3"));

Future<Integer> test4 = threadPool.submit(new MyTask("test4"));
do{
    System.out.println("wait test4");
}while (!test4.isDone());
System.out.println(test4.get());
}

@Override
```

```
public void run() {  
  
    System.out.println(this.name);  
  
}  
  
public String getName() {  
  
    return name;  
  
}  
  
public void setName(String name) {  
  
    this.name = name;  
  
}  
  
static class MyTask implements Callable{  
  
    private String data;  
  
    public MyTask(String data) {  
  
        this.data = data;  
  
    }  
  
    public Object call() throws Exception {  
  
        Thread.sleep(100);  
  
        System.out.println(data);  
  
        return "success";  
  
    }  
  
}  
}  
  
public class Test2<T> implements Callable<T> {
```

```
private T data;

public Test2(T data) {
    this.data = data;
}

public static void main(String[] args) throws ExecutionException,
InterruptedException {
    FutureTask<String> futureTask=new FutureTask<>(new Test2("test"));
    new Thread(futureTask).start();
    while (true){
        if (futureTask.isDone()){
            break;
        }else{
            System.out.println("2333");
        }
    }
    System.out.println(futureTask.get());
}

@Override
public T call() throws Exception {
    Thread.sleep(10000);
    System.out.println(this.data);
    return (T) "success";
}
```

```
    }  
}
```

ScheduledThreadPoolExecutor 实现定时器和延时加载功能

1、定时器：

- ScheduledThreadPoolExecutor 有个 scheduleAtFixedRate(command, initialDelay, period, unit) ;方法
 - command: 执行的线程 (可自己 New 一个)
 - initialDelay: 初始化执行的延时时间
 - period: 时间间隔
 - unit : 时间类型 (如 TimeUnit.SECONDS: 秒的方式执行, TimeUnit.DAYS : 天数的方式执行)

2、延时

- ScheduledThreadPoolExecutor 有个 schedule(callable, delay, unit) ; 方法
 - callable: 回调方法
 - delay: 延时时间
 - unit: 时间类型, 同定时器的 unit 一样

线程池在不同 CPU 下，线程池线程数的配置方法

CPU 密集型：核心线程数=CPU 核心数(或 核心线程数=CPU 核心数+1)

I/O 密集型：核心线程数=2*CPU 核心数 (或 核心线程数=CPU 核心数/ (1-阻塞系数))

混合型：核心线程数= (线程等待时间/线程 CPU 时间+1) *CPU 核心数

1. `getThreadUserTime` 获得线程在用户态执行的时间，单位为纳秒，但由于硬件及操作系统的限制不保证精确到纳秒
2. `getThreadInfo` 获得 `ThreadInfo` 的实例，并通过 `ThreadInfo` 的 `getBlockedTime` 获得被阻塞的时间（调用 `synchronized` 的语句或函数时），通过 `ThreadInfo` 的 `getWaitedTime` 获得等待时间（调用 `wait/join` 等待的时间）

将 `userTime` 除以 `blockedTime+waitedTime` 即可知道执行时间与阻塞时间的比例。

```
public class Test3 {  
    public static void main(String[] args) throws InterruptedException {  
        new Thread(()->{  
            test();  
        }, "test1").start();  
  
        new Thread(()->{  
            test();  
        }, "test2").start();  
  
        ThreadMXBean tmbean = ManagementFactory.getThreadMXBean();  
        tmbean.setThreadContentionMonitoringEnabled(true);  
  
        while(true){  
            ThreadInfo[] threadInfos = tmbean.dumpAllThreads(false, false);  
            for (ThreadInfo info:threadInfos){  
                System.out.println(tmbean.getThreadCpuTime(info.getThreadId()) + "\t" + info.getBl
```

```
ockedTime() + "\t" + info.getWaitedTime() + "\t" + info.getThreadName());  
}  
  
//获取发生死锁的线程 id 数组  
  
/*long[] deadlockedThreads = tmbean.findDeadlockedThreads();  
  
long[] allThread = tmbean.getAllThreadIds();  
  
for (int i=0;i<allThread.length;i++){  
  
    ThreadInfo info = tmbean.getThreadInfo(allThread[i]);  
  
    System.out.println(tmbean.getThreadCpuTime(allThread[i]) + "\t" + info.getBlockedT  
ime() + "\t" + info.getWaitedTime() + "\t" + info.getThreadName());  
  
}*/  
  
System.out.println();  
  
Thread.sleep(20000);  
}  
}  
  
static void test(){  
try {  
  
    while(true){  
  
        Thread.sleep(1000);  
  
        int k=0;  
  
        for (int i=0;i<10000000;i++){  
  
            k++;
```

```
        }

    }

} catch (InterruptedException e) {

    throw new RuntimeException(e);

}

}

}

// -Xint

public class Test3 {

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

        ExecutorService executorService = Executors.newFixedThreadPool(1);

        executorService.execute(() -> {

            test();

        });

        ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();

        threadMXBean.setThreadContentionMonitoringEnabled(true);

        threadMXBean.setThreadCpuTimeEnabled(true);

        while (true) {

            Thread.sleep(10000);

            executorService.execute(() -> {

                test();

            });

        }

    }

}
```

```
        });

        long[] allThreadIds = threadMXBean.getAllThreadIds();

        for (long id : allThreadIds) {

            ThreadInfo info = threadMXBean.getThreadInfo(id);

            System.out.println(threadMXBean.getThreadCpuTime(id) + "\t" +

info.getBlockedTime() + "\t" + info.getWaitedTime() + "\t" +

info.getThreadName());

        }

        System.out.println();

    }

}

static void test() {

    int k = 0;

    int b = 0;

    for (int i = 0; i < 1000000000; i++) {

        k++;

        b = k;

    }

}

}
```

ThreadLocal 使用及原理详解

1.ThreadLocal 可以实现【资源对象】的线程隔离，让每个线程各用各的【资源对象】，避免争用引发的线程安全问题

2.ThreadLocal 同时实现了线程内的资源共享

3.其原理是，每个线程内有一个 ThreadLocalMap 类型的成员变量，用来存储资源对象

①调用 set 方法，就是以 ThreadLocal 自己作为 key，资源对象作为 value，放入当前线程的 ThreadLocalMap 集合中

②调用 get 方法，就是以 ThreadLocal 自己作为 key，到当前线程中查找关联的资源值

③调用 remove 方法，就是以 ThreadLocal 自己作为 key，移除当前线程关联的资源值

4.为什么 ThreadLocalMap 中的 key（即 ThreadLocal）要设计为弱引用？

①Thread 可能需要长时间运行（如线程池中的线程），如果 key 不再使用，需要在内存不足（GC）时释放其占用的内存

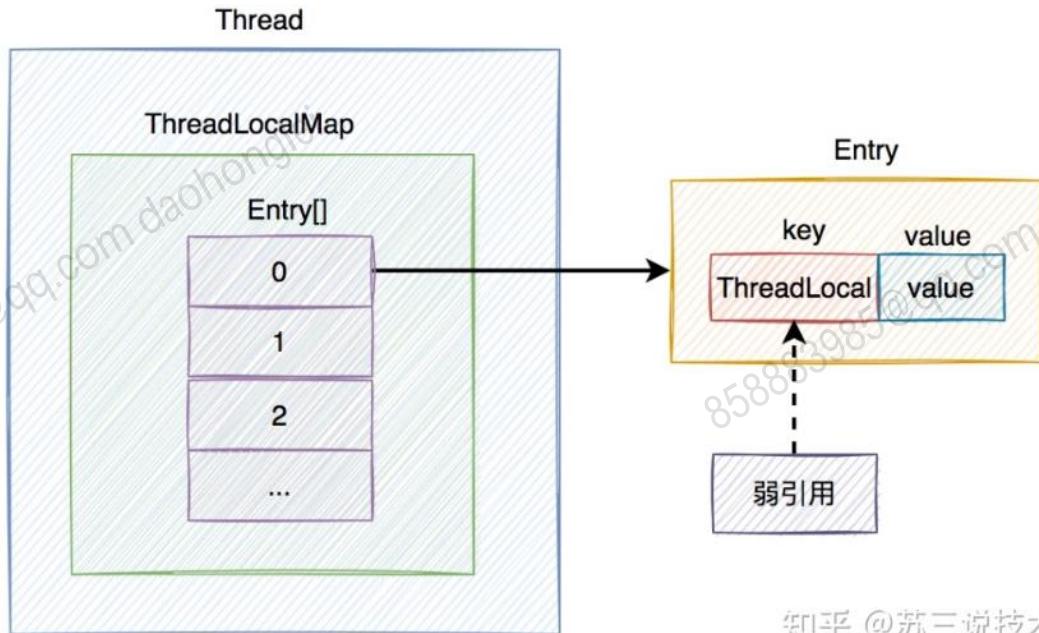
②但 GC 仅是让 key 的内存释放，后续还要根据 key 是否为 null 来进一步释放值的内存，释放时机有

a) 获取时发现 key 为 null

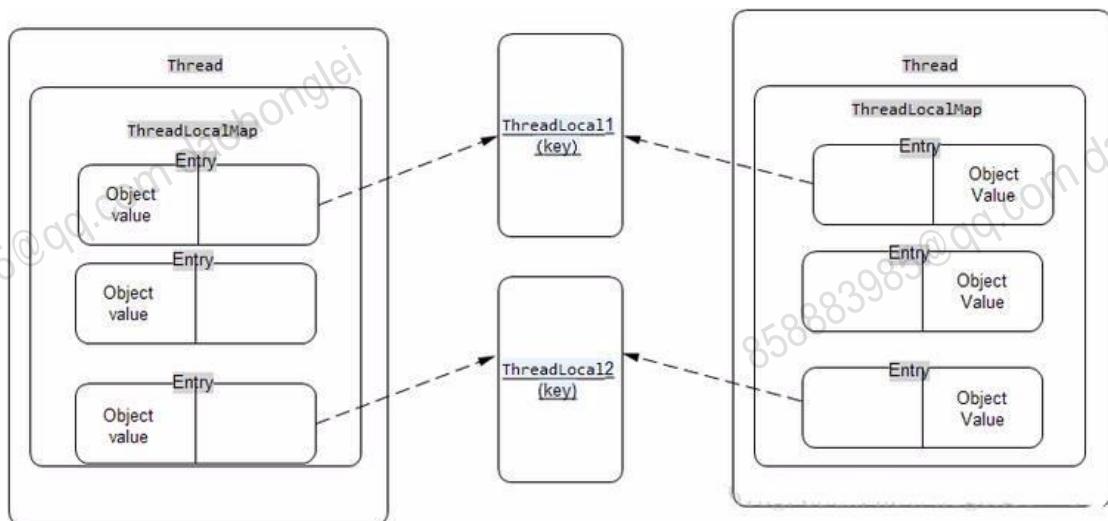
b) set key 时，会使用启发式扫描，清除临近的 null key，启发次数与元素个数，是否发现 nullkey 有关

c) remove 时（推荐），因为一般使用 ThreadLocal 时都把它作为静态变量，因此 GC 无法回收

每个线程 Thread 都有一个 ThreadLocalMap，ThreadLocal 是这个 Map 的工具类。当我们通过 ThreadLocal 存放数据时，这个 Map 会添加一条记录。这条记录的 key 存放的是这个 ThreadLocal 的引用,value 存的是缓存的数据（多个 ThreadLocal 存储不同的数据，这样这个 Map 就会有很多记录）



知乎 @苏三说技术



```
public class Test3 {
    static ThreadLocal<String> threadLocal=new ThreadLocal<>();
    static ThreadLocal<String> threadLocal2=new ThreadLocal<>();
    public static void main(String[] args) {
```

```
new Thread()->{

    Test3.threadLocal.set("test1");

    Test3.threadLocal2.set("test2");

    System.out.println(threadLocal.get());

    System.out.println(threadLocal2.get());

}).start();

new Thread()->{

    System.out.println(Test3.threadLocal.get());

    System.out.println(Test3.threadLocal2.get());

}).start();

}

}
```

什么是线程死锁？死锁如何产生？如何避免线程死锁？

死锁的介绍：

线程死锁是指由于两个或者多个线程互相持有对方所需要的资源，导致这些线程处于等待状态，无法前往执行。当线程进入对象的 synchronized 代码块时，便占有了资源，直到它退出该代码块或者调用 wait 方法，才释放资源，在此期间，其他线程将不能进入该代码块。当线程互相持有对方所需要的资源时，会互相等待对方释放资源，如果线程都不主动释放所占有的资源，将产生死锁。

死锁的产生的一些特定条件：

1、互斥条件：进程对于所分配到的资源具有排它性，即一个资源只能被一个进程占用，直到被该进程释放。

2、请求和保持条件：一个进程因请求被占用资源而发生阻塞时，对已获得的资源保持不放。

3、不剥夺条件：任何一个资源在没被该进程释放之前，任何其他进程都无法对他剥夺占用。

4、循环等待条件：当发生死锁时，所等待的进程必定会形成一个环路（类似于死循环），造成永久阻塞。

如何避免：

1、加锁顺序：

当多个线程需要相同的一些锁，但是按照不同的顺序加锁，死锁就很容易发生。如果能确保所有的线程都是按照相同的顺序获得锁，那么死锁就不会发生。当然这种方式需要你事先知道所有可能会用到的锁，然而总有些时候是无法预知的。

2、加锁时限：

加上一个超时时间，若一个线程没有在给定的时限内成功获得所有需要的锁，则会进行回退并释放所有已经获得的锁，然后等待一段随机的时间再重试。但是如果有很多的线程同一时间去竞争同一批资源，就算有超时和回退机制，还是可能会导致这些线程重复地尝试但却始终得不到锁。

3、死锁检测：

死锁检测即每当一个线程获得了锁，会在线程和锁相关的数据结构中（map、graph 等等）将其记下。除此之外，每当有线程请求锁，也需要记

录在这个数据结构中。死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。

怎么检测一个线程是否拥有锁？

在 `java.lang.Thread` 中有一个静态方法叫 `holdsLock()`，它返回 `true` 表示当且仅当当前线程拥有某个具体对象的锁。

```
public class Test4 {  
  
    public static final void main(String[] args) {  
  
        Thread thread = Thread.currentThread();  
  
        Thread.State state = thread.getState();  
  
        System.out.println(state);  
  
  
        synchronized (Test4.class){  
  
            System.out.println(Thread.holdsLock(Test4.class));  
  
        }  
  
        System.out.println(Thread.holdsLock(Test4.class));  
  
    }  
  
}
```

RUNNABLE

true

false

Synchronized 优化后的锁机制简单介绍一下，包括自旋锁、偏向锁、轻量级锁、重量级锁？

自旋锁：线程自旋说白了就是让 CPU 在做无用功，比如：可以执行几次 for 循环，可以执行几条空的汇编指令，目的是占着 CPU 不放，等待获取锁的机会。如果旋转的时间过长会影响整体性能，时间过短又达不到延迟阻塞的目的。

偏向锁：偏向锁就是一旦线程第一次获得了监视对象，之后让监视对象“偏向”这个线程，之后的多次调用则可以避免 CAS 操作，说白了就是置个变量，如果发现为 true 则无需再走各种加锁/解锁流程。

轻量级锁：轻量级锁是由偏向锁升级来的，偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁争用的时候，偏向锁就会升级为轻量级锁；

重量级锁：重量锁在 JVM 中又叫对象监视器（Monitor），它很像 C 中的 Mutex，除了具备 Mutex(0|1) 互斥的功能，它还负责实现了 Semaphore(信号量) 的功能，也就是说它至少包含一个竞争锁的队列，和一个信号阻塞队列（wait 队列），前者负责做互斥，后一个用于做线程同步。

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 锁升级过程

每一个线程在准备获取共享资源时： 第一步，检查 MarkWord 里面是不是放的自己的 ThreadId，如果是，表示当前线程是处于“偏向锁”。 第二步，如果 MarkWord 不是自己的 ThreadId，锁升级，这时候，用 CAS 来执行切换，新的线程根据 MarkWord 里面现有的 ThreadId，通知之前线程暂停，之前线程将 Markword 的内容置为空。 第三步，两个线程都把对象的 hashCode 复制到自己新建的用于存储锁的记录空间，接着开始通过 CAS 操作，把共享对象的 Markword 的内容修改为自己新建的记录空间的地址的方式竞争 MarkWord，第四步，第三步中成功执行 CAS 的获得资源，失败的则进入自旋。 第五步，自旋的线程在自旋过程中，成功获得资源（即之前获的资源的线程执行完成并释放了共享资源），则整个状态依然处于轻量级锁的状态，如果自旋失败。 第六步，进入重量级锁的状态，这个时候，自旋的线程进行阻塞，等待之前线程执行完成并唤醒自己。

synchronized 底层实现原理

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。

Java 中每一个对象都可以作为锁，这是 synchronized 实现同步的基础：

- 普通同步方法，锁是当前实例对象
- 静态同步方法，锁是当前类的 class 对象
- 同步方法块，锁是括号里面的对象

synchronized 和 volatile 的区别是什么？

- volatile 本质是在告诉 jvm 当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
- volatile 仅能使用在变量级别； synchronized 则可以使用在变量、方法、和类级别的。

- volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。
- volatile 不会造成线程的阻塞； synchronized 可能会造成线程的阻塞。
- volatile 标记的变量不会被编译器优化。

重入锁的 wait 穿透

```
public class Test7 {  
    public static void main(String[] args) throws InterruptedException {  
        Object object = new Object();  
        new Thread(()->{  
            try {  
                synchronized (object){  
                    synchronized (object){  
                        System.out.println(Thread.currentThread().getName() + " wait  
before");  
                        object.wait();  
                        System.out.println(Thread.currentThread().getName() + " wait after");  
                    }  
                }  
            } catch (InterruptedException e) {  
                throw new RuntimeException(e);  
            }  
        }  
    }  
}
```

```
    }, "1").start();

    Thread.sleep(1000);

new Thread(() -> {

    synchronized (object) {

        System.out.println(Thread.currentThread().getName() + " wait test");

        object.notify();

    }

}, "2").start();

}

}
```

1 wait before

2 wait test

1 wait after

volatile 变量和 atomic 变量有什么不同？

volatile 变量和 atomic 变量看起来很像，但功能却不一样。Volatile 变量可以确保先行关系，即写操作会发生在后续的读操作之前，但它并不能保证原子性。

例如用 volatile 修饰 count 变量那么 count++ 操作就不是原子性的。而 AtomicInteger 类提供的 atomic 方法可以让这种操作具有原子性如 getAndIncrement() 方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

synchronized 和 lock 的区别

区别如下：

类别	synchronized	Lock
存在层次	Java的关键字，在jvm层面上	是一个类
锁的释放	1、以获取锁的线程执行完同步代码，释放锁 2、线程执行发生异常，jvm会让线程释放锁	在finally中必须释放锁，不然容易造成线程死锁
锁的获取	假设A线程获得锁，B线程等待。如果A线程阻塞，B线程会一直等待	分情况而定，Lock有多个锁获取的方式，具体下面会说道，大致就是可以尝试获得锁，线程可以不用一直等待
锁状态	无法判断	可以判断
锁类型	可重入 不可中断 非公平	可重入 可判断 可公平（两者皆可）
性能	少量同步	大量同步 https://log.csdn.net/hefenglian

1. 来源：lock 是一个接口，而 synchronized 是 java 的一个关键字，synchronized 是内置的语言实现；
2. 异常是否释放锁：synchronized 在发生异常时候会自动释放占有的锁，因此不会出现死锁；而 lock 发生异常时候，不会主动释放占有的锁，必须手动 unlock 来释放锁，可能引起死锁的发生。（所以最好将同步代码块用 try catch 包起来，finally 中写入 unlock，避免死锁的发生。）
3. 是否响应中断 lock 等待锁过程中可以用 interrupt 来中断等待，而 synchronized 只能等待锁的释放，不能响应中断；
4. 是否尝试获取锁，Lock 可以通过 trylock 来尝试获取锁，获取到了执行获取到的业务，获取不到时执行获取不到的业务，而 synchronized 不能，只等阻塞获取；
5. synchronized 使用 Object 对象本身的 wait、notify、notifyAll 调度机制，而 Lock 可以使用 Condition 进行线程之间的调度，
6. Lock 可以提高多个线程进行读操作的效率。（可以通过 readritelock 实现读写分离）

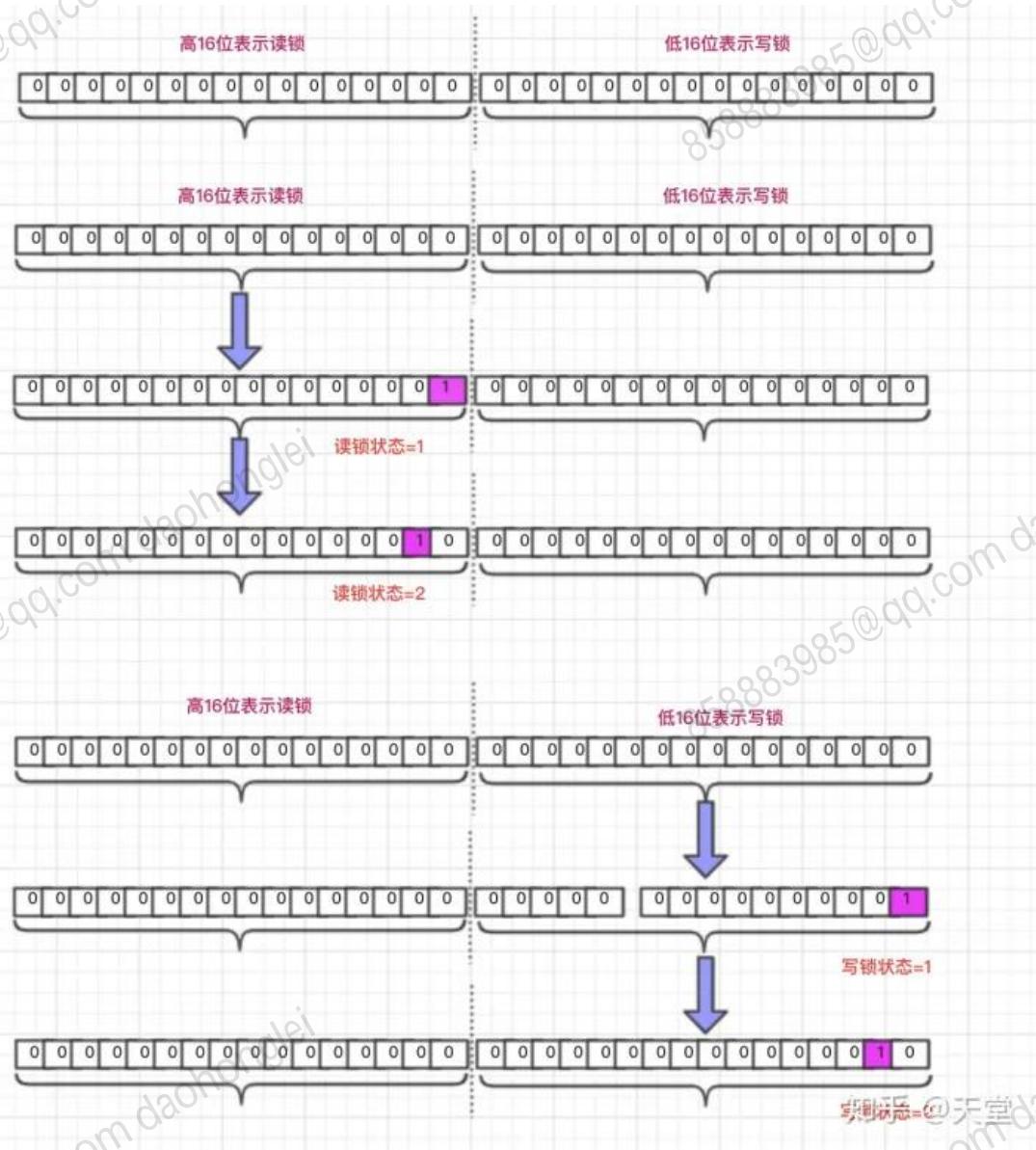
7. 在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时 Lock 的性能要远远优于 synchronized。所以说，在具体使用时要根据适当情况选择。

相同点

- 1.多线程通过 CAS 竞争锁，当某个线程竞争成功，其他线程都是先自旋，如果失败，然后入队列
- 2.竞争锁成功的线程，都会被记录在一个变量中，这个变量返回的就是当前竞争锁成功的线程对象；
- 3.可重入锁状态都是通过一个变量记录的；

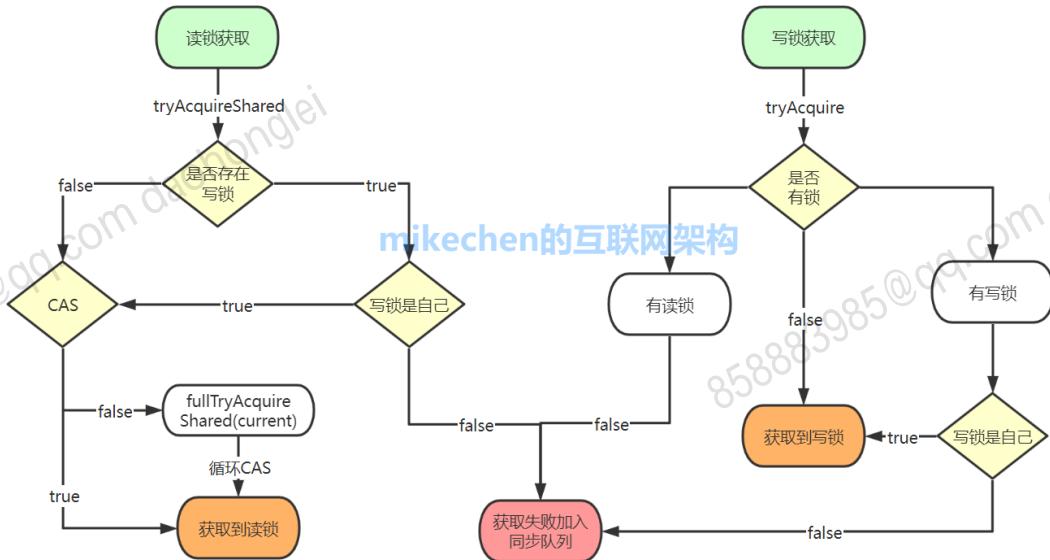
ReentrantReadWriteLock 实现原理

- 在 AQS 中，通过 int 类型的全局变量 state 来表示同步状态，即用 state 来表示锁。ReentrantReadWriteLock 也是通过 AQS 来实现锁的，但是 ReentrantReadWriteLock 有两把锁：读锁和写锁，它们保护的都是同一个资源，那么如何用一个共享变量来区分锁是写锁还是读锁呢？答案就是按位拆分。
- 由于 state 是 int 类型的变量，在内存中占用 4 个字节，也就是 32 位。将其拆分为两部分：高 16 位和低 16 位，其中高 16 位用来表示读锁状态，低 16 位用来表示写锁状态。当设置读锁成功时，就将高 16 位加 1，释放读锁时，将高 16 位减 1；当设置写锁成功时，就将低 16 位加 1，释放写锁时，将第 16 位减 1。如下图所示。



- 那么如何根据 state 的值来判断当前锁的状态时写锁还是读锁呢？
- 假设锁当前的状态值为 S，将 S 和 16 进制数 0x0000FFFF 进行与运算，即 $S \& 0x0000FFFF$ ，运算时会将高 16 位全置为 0，将运算结果记为 c，那么 c 表示的就是写锁的数量。如果 c 等于 0 就表示还没有线程获取锁；如果 c 不等于 0，就表示有线程获取到了锁，c 等于几就代表写锁重入了几次。
- 当成功获取到写锁时，令 $S + 1$ 即表示写锁状态+1；释放写锁时，就进行 $S - 1$ 运算。
- 将 S 无符号右移 16 位 ($S >> 16$)，得到的结果就是读锁的数量。当 $S >> 16$ 得到的结果不等于 0，且 c 也不等于 0 时，就表示当前线程既持有了写锁，也持有了读锁。

- 当成功获取到读锁时，如何对读锁进行加 1 呢？ $S + (1 << 16)$ 得到的结果，就是将对锁加 1。释放读锁是，就进行 $S - (1 << 16)$ 运算。
- 由于读锁和写锁的状态值都只占用 16 位，所以读锁的最大数量为 $2^{16}-1$ ，写锁可被重入的最大次数为 $2^{16}-1$ 。



Java 中 `++` 操作符是线程安全的吗？

不是线程安全的操作。它涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过程可能会出现多个线程交差。

CAS 如何保证原子性

原子性是指一个或者多个操作在 CPU 执行的过程中不被中断的特性，要么执行，要不执行，不能执行到一半（不可被中断的一个或一系列操作）。

为了保证 C A S 的原子性，CPU 提供了**总线锁定、缓存锁定**两种方式

总线锁定

总线（BUS）是计算机组件间的传输数据方式，也就是说 CPU 与其他组件连接传输数

据，就是靠总线完成的，比如 CPU 对内存的读写。

总线锁定是指 CPU 使用了总线锁，所谓总线锁就是使用 CPU 提供的 LOCK#信号，当 CPU 在总线上输出 LOCK#信号时，其他 CPU 的总线请求将被阻塞。

缓存锁定

总线锁定方式虽然保证了原子性，但是在锁定期间，会导致大量阻塞，增加系统的性能开销，所以现代 CPU 为了提升性能，通过锁定范围缩小的思想设计出了缓存行锁定（缓存行是 CPU 高速缓存存储的最小单位）。

所谓缓存锁定是指 CPU 对缓存行进行锁定，当缓存行中的共享变量回写到内存时，其他 CPU 会通过总线嗅探机制感知该共享变量是否发生变化，如果发生变化，让自己对应的共享变量缓存行失效，重新从内存读取最新的数据，缓存锁定是基于缓存一致性机制来实现的，因为缓存一致性机制会阻止两个以上 CPU 同时修改同一个共享变量（现代 CPU 基本都支持和使用缓存锁定机制）。

总线锁定 or 缓存锁定

对于早期的 CPU，总是采用的是锁总线的方式。具体方法是，一旦遇到了 Lock 指令，就由仲裁器选择一个核心独占总线。其余的 CPU 核心不能再通过总线与内存通讯。从而达到“原子性”的目的。具体做法是，某一个核心触发总线的“Lock#”那根线，让总线仲裁器工作，把总线完全分给某个核心。

这种方式的确能解决问题，但是非常不高效。为了个原子性结果搞得其他 CPU 都不能干活了。因此从 Intel P6 CPU 开始就做了一个优化，改用 Ringbus + MESI 协议，也就是文档里说的 cache coherence 机制。这种技术被 Intel 称为“Cache Locking”。

根据文档原文：如果是 P6 后的 CPU，并且数据已经被 CPU 缓存了，并且是要写回到主存的，则可以用 cache locking 处理问题。**否则还是得锁总线**。因此，lock 到底用锁总线，还是用 cache locking，完全是看当时的情况。当然能用后者的就肯定用后者。

Intel P6 是 Intel 第 6 代架构的 CPU，其实也很老了，差不多 1995 年出的……比如 Pentium Pro, Pentium II, Pentium III 都隶属于 P6 架构。

MESI 大致的意思是：若干个 CPU 核心通过 ringbus 连到一起。每个核心都维护自己的 Cache 的状态。如果对于同一份内存数据在多个核里都有 cache，则状态都为 S (shared)。一旦有一核心改了这个数据（状态变成了 M），其他核心就能瞬间通过 ringbus 感知到这个修改，从而把自己的 cache 状态变成 I (Invalid)，并且从标记为 M 的 cache 中读过来。同时，这个数据会被原子的写回到主存。最终，cache 的状态又会变为 S。

这相当于给 cache 本身单独做了一套总线（要不怎么叫 ring bus），避免了真的锁总线。

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt)(JNIEnv *env,
    jobject unsafe, jobject obj, jlong offset, jint e, jint x) // CAS
UnsafeWrapper("Unsafe_CompareAndSwapInt");
oop p = JNIHandles::resolve(obj);
jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
```

程序员囧辉
@51CTO博客

```
// linux_x86
inline jint Atomic::cmpxchg(jint exchange_value,
    volatile jint* dest, jint compare_value) {
    int mp = os::is_MP();
    __asm__ volatile (LOCK_IF_MP(%4) "cmpxchgl %1,(%3)"
        : "=a" (exchange_value)
        : "r" (exchange_value), "a" (compare_value), "r" (dest), "r" (mp)
        : "cc", "memory");
    return exchange_value;
}
```

程序员囧辉
@51CTO博客

LOCK_IF_MP(mp)会根据 mp 的值来决定是否为 cmpxchg 指令添加 lock 前缀，如果通过 mp 判断当前系统是多处理器（即 mp 值为 1），则为 cmpxchg 指令添加 lock 前缀，

否则，不加 lock 前缀。

CAS 原理分析及 ABA 问题详解

什么是 CAS?

CAS (Compare-and-Swap, 比较并替换) ,CAS 需要有 3 个操作数：内存地址 V, 旧的预期值 A, 新值 B;其作用是当且仅当内存地址 V 的值与预期值 A 相等时，将内存地址 V 的值修改为 B, 否则就什么都不做。整个比较并替换的操作是一个原子操作。Java 是无法直接访问操作系统底层的 API 的（原因是 Java 的跨平台性限制了 Java 不能和操作系统耦合），所以 Java 通过 JNI (Java Native Interface, Java 本地接口) 调用 C/C++ 来实现 CAS 操作。

CAS 的缺点

1、ABA 问题

ABA 问题是指在 CAS 操作时，其他线程将变量值 A 改为了 B，但是又被改回了 A，等到本线程使用期望值 A 与当前变量进行比较时，发现变量 A 没有变，于是 CAS 就将 A 值进行了交换操作，但是实际上该值已经被其他线程改变过。

解决办法：在变量前面加上版本号，每次变量更新的时候变量的版本号都+1，即 A->B->A 就变成了 1A->2B->3A。只要变量被某一线程修改过，变量对应的版本号就会发生递增变化，从而解决了 ABA 问题。

2、循环时间长开销大

自旋 CAS (也就是不成功就一直循环执行直到成功) 如果长时间不成功，会给 CPU 带来非常大的执行开销。在 Java 中有很多的并发框架都使用了自旋 CAS 来获取相应的锁，会一直循环直到获取到相应的锁后，然后执行相应的操作。那么当其自旋时 CAS，

会一直占用 CPU 的资源。如果自旋 CAS 长时间不成功，会给 CPU 带来非常大的执行开销。

3、只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5 开始，提供了 `AtomicReference` 类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作。所以我们可以使用锁或者利用 `AtomicReference` 类把多个共享变量合并成一个共享变量来操作。

CAS 与 `synchronized` 的使用情景

简单的来说 CAS 适用于写比较少的情况下（多读场景，冲突一般较少），
`synchronized` 适用于写比较多的情况下（多写场景，冲突一般较多）

对于资源竞争较少（线程冲突较轻）的情况，使用 `synchronized` 同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗 cpu 资源；而 CAS 基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。

对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 `synchronized`。

补充：Java 并发编程这个领域中 `synchronized` 关键字一直都是元老级的角色，很久之前很多人都会称它为“重量级锁”。但是，在 JavaSE 1.6 之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁 和 轻量级锁 以及其它各种优化之后变得在某些情况下并不是那么重了。`synchronized` 的底层实现主要依靠 Lock-Free 的队列，基本思路是 自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。在线程冲突较少的情况下，可以获得和 CAS 类似的性能；而线程冲突严重的情况下，性能远高于 CAS。

Unsafe 测试

```
public class UnsafeTest2 {  
    //获得 Unsafe 的一个实例  
  
    static final Unsafe unsafe;  
  
    static final long stateOffset;  
  
    private volatile long state = 0;  
  
    static {  
        try {  
            //使用反射获取 Unsafe 的成员变量 theUnsafe  
            Field field = Unsafe.class.getDeclaredField("theUnsafe");  
  
            //设置为可存取  
            field.setAccessible(true);  
  
            //获取该变量的值  
            unsafe = (Unsafe) field.get(null);  
  
            //获取 UnsafeTest 类里面的 state 变量, 在 UnsafeTest 对象里面的内存偏移量  
            //地址并将其保存到 stateOffset 变量中。  
            //getDeclaredFields(): 获得某个类的所有声明的字段  
  
            stateOffset =  
  
            unsafe.objectFieldOffset(UnsafeTest2.class.getDeclaredField("state"));  
  
        } catch (Exception e) {  
            System.out.println(e.getLocalizedMessage());  
  
            throw new Error(e);  
        }  
    }  
}
```

```
    }

}

public static void main(String[] args){

    UnsafeTest2 unsafeTest = new UnsafeTest2();

    //如果 test 对象中 内存偏移量为 stateOffset 的 state 变量的值为 0，则更新该值为
    1

    boolean b = unsafe.compareAndSwapInt(unsafeTest, stateOffset, 0, 1);

    System.out.println(b);//true

}

}
```

volatile 的实现原理

当 volatile 修饰的变量进行写操作的时候，JVM 就会向 CPU 发送 LOCK#前缀指令，此时当前处理器的缓存行就会被锁定，通过缓存一致性机制确保修改的原子性，然后更新对应的主存地址的数据。

处理器会使用嗅探技术保证在当前处理器缓存行，主存和其他处理器缓存行的数据的在总线上保持一致。在 JVM 通过 LOCK 前缀指令更新了当前处理器的数据之后，其他处理器就会嗅探到数据不一致，从而使当前缓存行失效，当需要用到该数据时直接去内存中读取，保证读取到的数据时修改后的值。

volatile 线程可见性

```
// 可见性例子
```

```
// -Xint 解释 -Xcomp 编译 -Xmixed 混合

public class ForeverLoop {

    static volatile boolean stop = false;

    public static void main(String[] args) {

        new Thread(() -> {

            try {

                Thread.sleep(100);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

            stop = true;

            get().debug("modify stop to true...");

        }).start();

        foo();
    }

    static void foo() {

        int i = 0;

        while (!stop) {

            i++;

        }

        get().debug("stopped... c:{}", i);
    }
}
```

```
}
```

DCL (Double Check Lock) 单例需不需要加 volatile

```
public class Mgr06 {
    private static volatile Mgr06 INSTANCE; //JIT

    private Mgr06() {
    }

    public static Mgr06 getInstance() {
        //业务逻辑代码省略
        if (INSTANCE == null) { //Double Check Lock
            //双重检查
            synchronized (Mgr06.class) {
                if (INSTANCE == null) {
                    try {
                        Thread.sleep( millis: 1 );
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    INSTANCE = new Mgr06();
                }
            }
        }
        return INSTANCE;
    }
}
```

<https://blog.csdn.net/he181026546>

我们可以发现 synchronized 前后都进行单例是非为空的判断，这就是 DCL，那为什么要加 volatile 呢？我们先来看看 new 一个实例的过程：

```
源码:  
class T {  
    int m = 8;  
}  
  
T t = new T();
```

```
汇编码:  
0 new #2 <T>  
3 dup  
4 invokespecial #3 <T.<init>>  
7 astore_1  
8 return
```

<https://blog.csdn.net/he1810368646>

我们通过 new 实例的字节码看到，其实 new 实例有五步：

```
0 new #2 <T>  
4 invokespecial #3 <T.<init>>  
7 astore_1
```

<https://blog.csdn.net/he1810368646>

- 一、按照对象大小分配一个内存，里面有成员变量，赋给成员变量一个默认的初始值，这一步也可称之为对象实例的半初始化。例如:int 类型的最小值是 0，就将 0 赋给 m；
- 二、是汇编中的，Java 不深究；
- 三、这里是调用该对象的构造方法。也就是将 m=8 执行；
- 四、将符号引用，指向堆内的实际内存地址
- 五、不用解释了。

我们把主要的关键三步拿出来分析，以为 CPU 会有指令重排序的现象，这是第一个线

程如果后两步发生了指令重排序，这时候实例 t 指向了半初始化对象，还未执行构造方法的时候。这时，**第二个线程来啦，先判断 t 是否为空，但是 t 不为空**，就会直接返回该对象。那么第二个线程使用的就是个半初始化状态的对象，所记录的就是没有赋值的 m。举个例子，订单原本有一百单，结果第二个线程发现没有，那后面的逻辑代码就会乱套啦。

所以我们对单例的对象用上 volatile 关键字，就是用到了它的特性：禁止指令重排序！！！

Java 中 sleep() 和 wait() 的区别

1. 这两个方法来自不同的类分别是，sleep 来自 Thread 类，和 wait 来自 Object 类。
2. 最主要是 sleep 方法没有释放锁，而 wait 方法释放了锁，使得其他线程可以使用同步控制块或者方法。
3. 使用范围：wait, notify 和 notifyAll 只能在同步控制方法或者同步控制块里面使用，而 sleep 可以在任何地方使用。
4. sleep 和 wait 必须捕获 **InterruptedException** 异常，而 notify 和 notifyAll 不需要捕获异常

Java 多线程学习之 wait、notify/notifyAll 详解

- 1、wait()、notify/notifyAll() 方法是 Object 的本地 final 方法，无法被重写。
- 2、wait() 使当前线程阻塞，前提是必须先获得锁，一般配合 synchronized 关键字使用，即，一般在 synchronized 同步代码块里使用 wait()、notify/notifyAll() 方法。
- 3、由于 wait()、notify/notifyAll() 在 synchronized 代码块执行，说明当前线程一定是获取了锁的。

当线程执行 wait()方法时候，会释放当前的锁，然后让出 CPU，进入等待状态。

只有当 notify/notifyAll() 被执行时候，才会唤醒一个或多个正处于等待状态的线程，然后继续往下执行，直到执行完 synchronized 代码块的代码或是中途遇到 wait()，再次释放锁。

也就是说，notify/notifyAll() 的执行只是唤醒沉睡的线程，而不会立即释放锁，锁的释放要看代码块的具体执行情况。所以在编程中，尽量在使用了 notify/notifyAll() 后立即退出临界区，以唤醒其他线程让其获得锁

4、wait() 需要被 try catch 包围，以便发生异常中断也可以使 wait 等待的线程唤醒。
5、notify 和 wait 的顺序不能错，如果 A 线程先执行 notify 方法，B 线程在执行 wait 方法，那么 B 线程是无法被唤醒的。

6、notify 和 notifyAll 的区别

notify 方法只唤醒一个等待（对象的）线程并使该线程开始执行。所以如果有多个线程等待一个对象，这个方法只会唤醒其中一个线程，选择哪个线程取决于操作系统对多线程管理的实现。notifyAll 会唤醒所有等待(对象的)线程，尽管哪一个线程将会第一个处理取决于操作系统的实现。如果当前情况下有多个线程需要被唤醒，推荐使用 notifyAll 方法。比如在生产者-消费者里面的使用，每次都需要唤醒所有的消费者或是生产者，以判断程序是否可以继续往下执行。

7、在多线程中要测试某个条件的变化，使用 if 还是 while？

要注意，notify 唤醒沉睡的线程后，线程会接着上次的执行继续往下执行。所以在进行条件判断时候，可以把 wait 语句忽略不计来进行考虑；显然，要确保程序一定要执行，并且要保证程序直到满足一定的条件再执行，要使用 while 进行等待，直到满足条件才继续往下执行。

强引用，软引用和弱引用的区别

强引用：只有这个引用被释放之后，对象才会被释放掉，只要引用存在，垃圾回收器永远不会回收，这是最常见的 New 出来的对象。

软引用：内存溢出之前通过代码回收的引用。软引用主要用户实现类似缓存的功能，在内存足够的情况下直接通过软引用取值，无需从繁忙的真实来源查询数据，提升速度；当内存不足时，自动删除这部分缓存数据，从真正的来源查询这些数据。

弱引用：第二次垃圾回收时回收的引用，短时间内通过弱引用取对应的数据，可以取到，当执行过第二次垃圾回收时，将返回 null。弱引用主要用于监控对象是否已经被垃圾回收器标记为即将回收的垃圾，可以通过弱引用的 isEnqueued 方法返回对象是否被垃圾回收器标记。

虚引用：虚引用的主要目的是在一个对象所占的内存被实际回收之前得到通知，从而可以进行一些相关的清理工作。

引用类型	被垃圾回收时间	用途	生存时间
强引用	从来不会	对象的一般状态	JVM 停止运行时终止
软引用	当内存不足时	对象缓存	内存不足时终止
弱引用	正常垃圾回收时	对象缓存	垃圾回收后终止
虚引用	正常垃圾回收时	跟踪对象的垃圾回收	垃圾回收后终止

```
public class Test6 {  
  
    public static void main(String[] args) {  
  
        //JVM 宁愿抛出 OutOfMemory 错误也不会回收这种对象  
  
        Object object = new Object();  
    }  
}
```

```
Object[] objects = new Object[100];
```

```
//只有在内存不足的时候 JVM 才会回收该对象。
```

```
SoftReference<String> sr = new SoftReference<>(new String("hello"));
```

```
System.out.println(sr.get());
```

```
System.gc();
```

```
System.out.println(sr.get());
```

```
//JVM 进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象
```

```
WeakReference<String> wr = new WeakReference<>(new String("hello2"));
```

```
System.out.println(wr.get());
```

```
System.gc();
```

```
System.out.println(wr.get());
```

```
//如果一个对象与虚引用关联，则跟没有引用与之关联一样，在任何时候都可能被垃圾回收器回收
```

```
/**
```

* 虚引用和前面的软引用、弱引用不同，它并不影响对象的生命周期。在 java 中用 `java.lang.ref.PhantomReference` 类表示。

* 如果一个对象与虚引用关联，则跟没有引用与之关联一样，在任何时候都可能被垃圾回收器回收。

* 要注意的是，虚引用必须和引用队列关联使用，当垃圾回收器准备回收一个对象时，

```
* 如果发现它还有虚引用，就会把这个虚引用加入到与之 关联的引用队列中。  
* 程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将  
要被垃圾回收。  
* 如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内  
存被回收之前采取必要的行动。  
  
*/  
  
ReferenceQueue<String> queue = new ReferenceQueue<>();  
  
PhantomReference<String> pr = new PhantomReference<>(new  
String("hello3"), queue);  
  
System.out.println(pr.get());  
  
System.gc();  
  
System.out.println(pr.get());  
  
}  
}
```

对象的内存布局了解吗？

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。

对象头占 12B，包括对象标记和类型指针。对象标记存储对象自身的运行时数据，如哈希码、GC 分代年龄、锁标志、偏向线程 ID 等，这部分占 8B，称为 Mark Word。Mark Word 被设计为动态数据结构，以便在极小的空间存储更多数据，根据对象状态复用存储空间。

类型指针 是对象指向它的类型元数据的指针，占 4B。JVM 通过该指针来确定对象是

哪个类的实例。

实例数据 是对象真正存储的有效信息，即本类对象的实例成员变量和所有可见的父类成员变量。存储顺序会受到虚拟机分配策略参数和字段在源码中定义顺序的影响。相同宽度的字段总是被分配到一起存放，在满足该前提条件的情况下父类中定义的变量会出现在子类之前。

对齐填充 不是必然存在的，仅起占位符作用。虚拟机的自动内存管理系统要求任何对象的大小必须是 8B 的倍数，对象头已被设为 8B 的 1 或 2 倍，如果对象实例数据部分没有对齐，需要对齐填充补全。

锁状态	32bit				
	25bit		4bit	1bit	2bit
	23bit	2bit		偏向模式	标志位
未锁定	对象哈希码		分代年龄	0	01
轻量级锁定	指向调用栈中锁记录的指针				00
重量级锁定 (锁膨胀)	指向重量级锁的指针				10
GC 标记	空				11
可偏向	线程 ID	Epoch	分代年龄	1	01

biased_lock	lock	状态
0	01	无锁
1	01	偏向锁
0	00	轻量级锁
0	10	重量级锁
0	11	

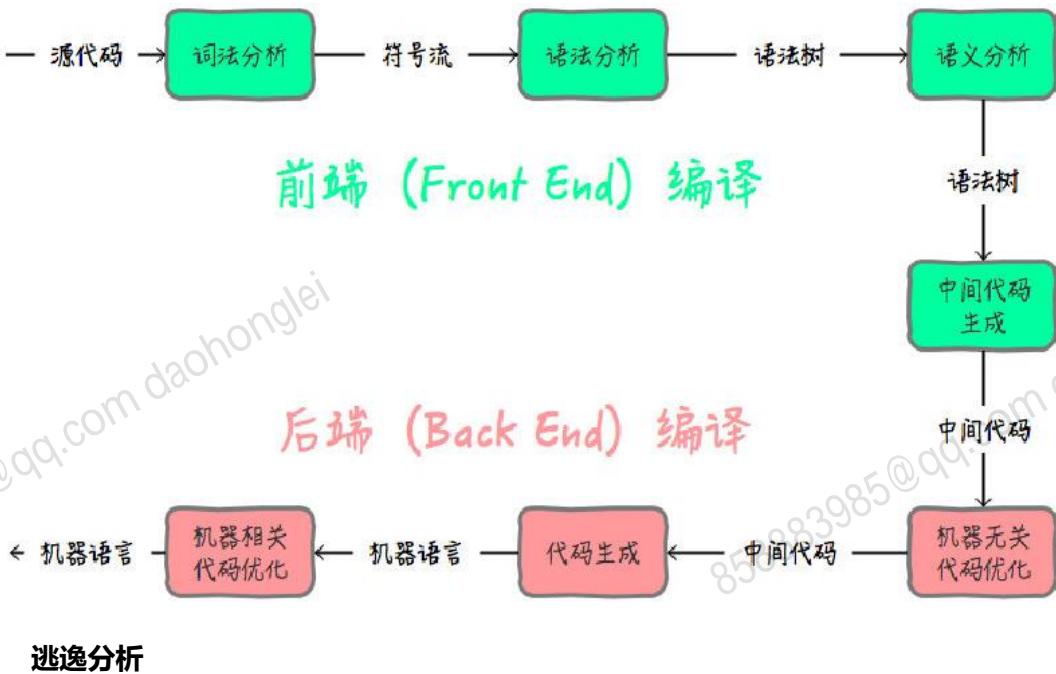
Java 的程序编译和代码优化

Java 语言的“编译期”有不同的解释，例如它可能是指把*.java 文件转变成*.class 文件的过程，也可能是指把*.class 文件转变成机器码的过程。

第一个过程的编译器被称为“前端编译器”，例如 javac

第二个过程的编译器被称为“JIT 编译器，Just In Time Compiler”，例如 HotSpot 的

C1、C2 编译器。



逃逸分析的基本行为就是分析对象动态作用域：当一个对象在方法中被定义后，它可能被外部方法所引用，称为方法逃逸。甚至还有可能被外部线程访问到，譬如赋值给类变量或可以在其他线程中访问的实例变量，称为线程逃逸。如果证明一个对象不会逃逸到方法或线程外，那么就能进行一些非常高效的优化策略：

栈上分配

栈上分配就是把方法中的变量和对象分配到栈上，方法执行完后自动销毁，而不需要垃圾回收的介入，从而提高系统性能。

同步消除

线程同步本身比较耗时，如果确定一个对象不会逃逸出线程，无法被其它线程访问到，那该对象的读写就不会存在竞争，对这个变量的同步措施就可以消除掉。单线程中是没有锁竞争。（锁和锁块内的对象不会逃逸出线程就可以把这个同步块取消）

标量替换

Java 虚拟机中的原始数据类型 (int, long 等数值类型以及 reference 类型等) 都不能再进一步分解, 它们就可以称为标量。相对的, 如果一个数据可以继续分解, 那它称为聚合量, Java 中最典型的聚合量是对象。如果逃逸分析证明一个对象不会被外部访问, 并且这个对象是可分解的, 那程序真正执行的时候将可能不创建这个对象, 而改为直接创建它的若干个被这个方法使用到的成员变量来代替。拆散后的变量便可以被单独分析与优化, 以各自分别在栈帧或寄存器上分配空间, 原本的对象就无需整体分配空间了。

JVM 的类加载机制是什么? 有哪些实现方式?

类加载机制:

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, 用来封装在方法区内的数据结构。类的加载最终是在堆区内的 Class 对象, Class 对象封装了类在方法区内的数据结构, 并且向 Java 程序员提供了访问方法区内的数据结构的接口。

类加载有三种方式:

- 1) 命令行启动应用时由 JVM 初始化加载
- 2) 通过 Class.forName () 方法动态加载
- 3) 通过 ClassLoader.loadClass () 方法动态加载

类的加载过程

一个 java 文件从被加载到被卸载这个生命过程, 总共要经历 5 个阶段,

JVM 将类加载过程分为:

加载->链接（验证+准备+解析）->初始化（使用前的准备）->使用->

卸载

(1) 加载

首先通过一个类的全限定名来获取此类的二进制字节流；其次将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构；最后在 java 堆中生成一个代表这个类的 Class 对象，作为方法区这些数据的访问入口。总的来说就是查找并加载类的二进制数据。

(2) 链接：

验证：确保被加载类的正确性；

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；

(3) 为类的静态变量赋予正确的初始值

类的初始化

(1) 类什么时候才被初始化

1) 创建类的实例，也就是 new 一个对象

2) 访问某个类或接口的静态变量，或者对该静态变量赋值

3) 调用类的静态方法

4) 反射 (Class.forName("com.lyj.load"))

5) 初始化一个类的子类（会首先初始化子类的父类）

6) JVM 启动时标明的启动类，即文件名和类名相同的那个类

(2) 类的初始化顺序

1) 如果这个类还没有被加载和链接，那先进行加载和链接

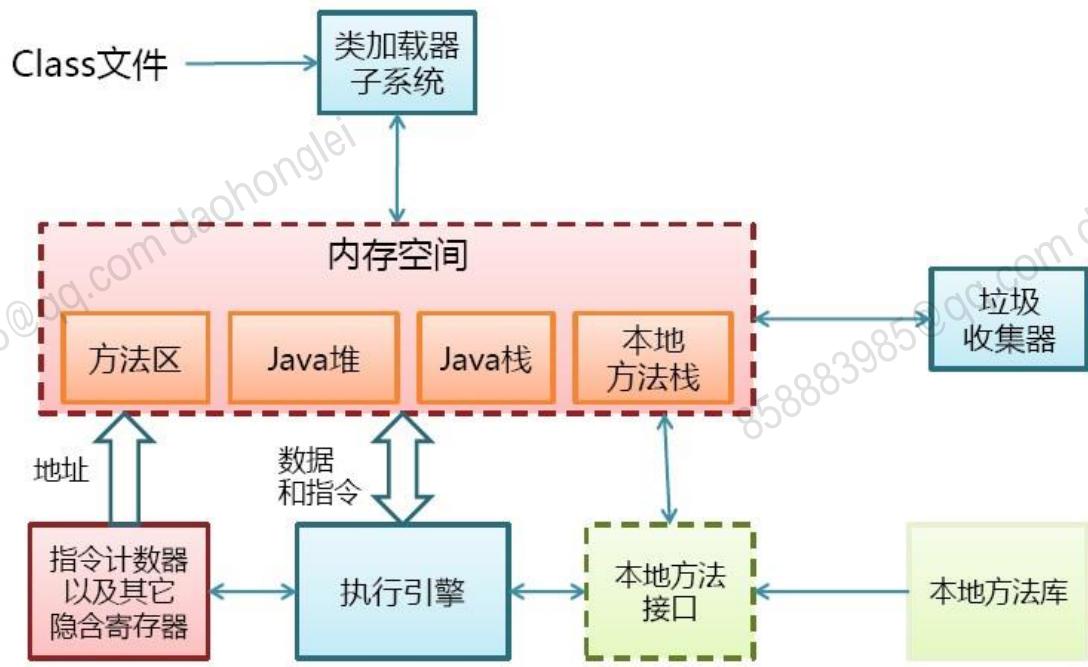
2) 假如这个类存在直接父类，并且这个类还没有被初始化（注意：在一个类加载器中，类只能初始化一次），那就初始化直接的父类（不适用于接口）

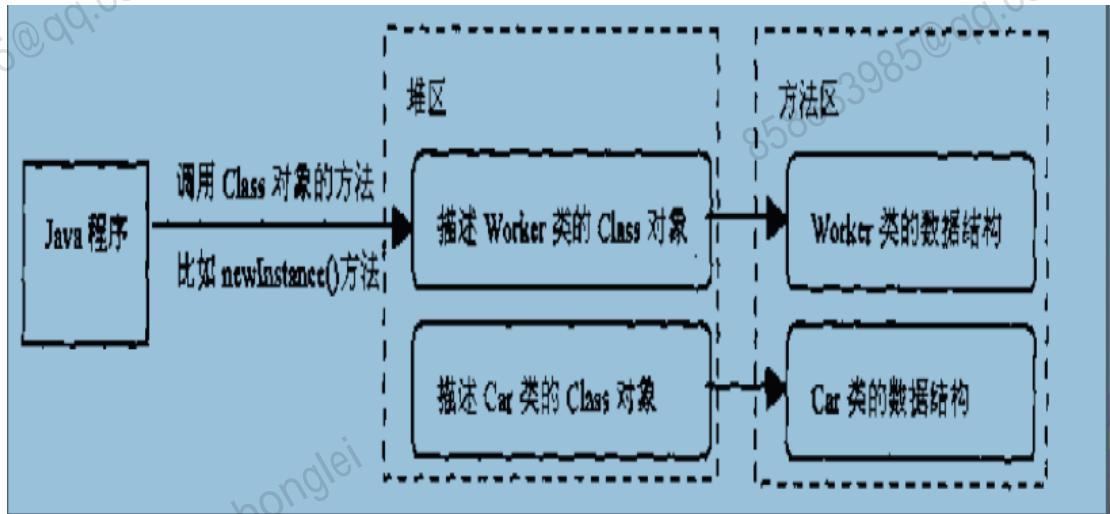
3) 加入类中存在初始化语句（如 static 变量和 static 块），那就依次执行这些初始化语句。

4) 总的来说，初始化顺序依次是：（静态变量、静态初始化块）->（变量、初始化块）-> 构造器；如果有父类，则顺序是：父类 static 方法-> 子类 static 方法-> 父类构造方法--> 子类构造方法

类的加载

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这个类的 java.lang.Class 对象，用来封装类在方法区类的对象。如：





类的加载的最终产品是位于堆区中的 Class 对象。Class 对象封装了类在方法区内
的数据结构，并且向 Java 程序员提供了访问方法区内的数据结构的接口。加载类
的方式有以下几种：

- 1) 从本地系统直接加载
- 2) 通过网络下载.class 文件
- 3) 从 zip, jar 等归档文件中加载.class 文件
- 4) 从专有数据库中提取.class 文件
- 5) 将 Java 源文件动态编译为.class 文件 (服务器)

ClassLoader 有 4 种具体的分类：

- 1、BootstrapClassLoader
- 2、ExtClassLoader
- 3、AppClassLoader
- 4、自定义 ClassLoader

BootstrapClassLoader：用来加载 Java 的核心类库，即存放在 java.* 包中
的字节码文件，如 java.util.List、java.io.InputStream、java.lang.Integer
等。

ExtClassLoader：用来加载 Java 的扩展类库，即存放在 javax.* 包中的字节
码文件，如 javax.sql.DataSource、javax.net.SocketFactory 等。

AppClassLoader：用来加载当前程序所在目录的类，即开发者自己编写的 Java 文件对应的字节码文件。

自定义的 ClassLoader：指开发者根据具体需求编写的类加载器，可以实现定制化加载。

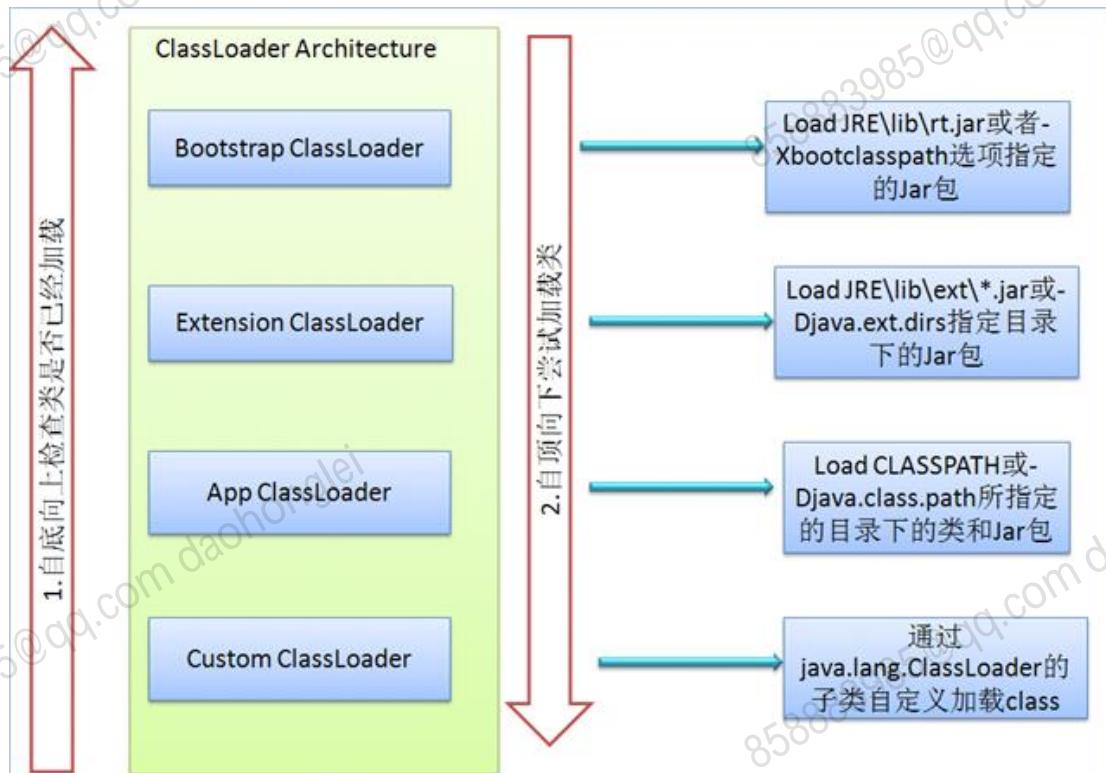
类加载器的顺序

1) 加载过程中会先检查类是否被已加载，检查顺序是自底向上，从 Custom

ClassLoader 到 BootStrap ClassLoader 逐层检查，只要某个 classloader 已加载就视为已加载此类，保证此类只所有 ClassLoader 加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

2) 在加载类时，每个类加载器会将加载任务上交给其父，如果其父找不到，再由自己去加载。

3) Bootstrap Loader (启动类加载器) 是最顶级的类加载器了，其父加载器为 null。



一道错误的面试题解答

能不能自己写个类叫java.lang.System?

答案：通常不可以，但可以采取另类方法达到这个需求。

解释：为了不让我们写System类，类加载采用委托机制，这样可以保证爸爸们优先，爸爸们能找到的类，儿子就没有机会加载。而System类是Bootstrap加载器加载的，就算自己重写，也总是使用Java系统提供的System，自己写的System类根本没有机会得到加载。

但是，我们可以自己定义一个类加载器来达到这个目的，为了避免双亲委派机制，这个类加载器也必须是特殊的。由于系统自带的三个类加载器都加载特定目录下的类，如果我们自己的类加载器放在一个特殊的目录，那么系统的加载器就无法加载，也就是最终还是由我们自己的加载器加载。

错在哪了？

自己编写类加载器就能加载一个假冒的 java.lang.System 吗？

不行。

- 假设你自己的类加载器用双亲委派，那么优先由启动类加载器加载真正的 java.lang.System，自然不会加载假冒的
- 假设你自己的类加载器不用双亲委派，那么你的类加载器加载假冒的 java.lang.System 时，它需要先加载父类 java.lang.Object，而你没有用委派，找不

到 `java.lang.Object` 所以加载会失败

- 以上也仅仅是假设。实际操作你就会发现自定义类加载器加载以 `java.` 打头的类时，会抛安全异常，在 jdk9 以上版本这些特殊包名都与模块进行了绑定，更连编译都过不了

双亲委派的目的有两点

1. 让上级类加载器中的类对下级共享（反之不行），即能让你的类能依赖到 jdk 提供的核心类
2. 让类的加载有优先次序，保证核心类优先加载

类初始化

类初始化是类加载过程的最后一个阶段，到初始化阶段，才真正开始执行类中的 Java 程序代码。虚拟机规范严格规定了有且只有四种情况必须立即对类进行初始化：

- 遇到 `new`、`getstatic`、`putstatic`、`invokestatic` 这四条字节码指令时，如果类还没有进行过初始化，则需要先触发其初始化。生成这四条指令最常见的 Java 代码场景是：使用 `new` 关键字实例化对象时、读取或设置一个类的静态字段（`static`）时（被 `static` 修饰又被 `final` 修饰的，已在编译期把结果放入常量池的静态字段除外）、以及调用一个类的静态方法时。
- 使用 `Java.lang.reflect` 包的方法对类进行反射调用时，如果类还没有进行过初始化，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，虚拟机会先执行该主类。

- static final 修饰的引用类型也会初始化，在初始化时赋值

虚拟机规定只有这四种情况才会触发类的初始化，称为对一个类进行主动引用，除此之外所有引用类的方式都不会触发其初始化，称为被动引用。

初始化数据块

什么是初始化数据块？

初始化数据块——当创建对象或加载类时运行的代码。

有两种类型的初始化数据块：

静态初始化器：加载类时运行的代码 不支持继承

实例初始化器：创建新对象时运行的代码 创建子类时也会调用父类的实力初始化器

```
public class Test3 {  
  
    static {  
  
        System.out.println("static");  
  
    }  
  
    {  
  
        System.out.println("object");  
  
    }  
  
    public static void main(String[] args) {  
  
        new Test3();  
  
        new Test3();  
  
        new Test3();  
  
    }  
}
```

```
}

static class Test33 extends Test3{

}

}
```

static

object

object

object

类初始化和实例化的区别

类的初始化：是完成程序执行前的准备工作。在这个阶段，静态的（变量，方法，代码块）会被执行。同时会开辟一块存储空间用来存放静态的数据。初始化只在类加载的时候执行一次。

类的实例化：是指创建一个对象的过程。这个过程中会在堆中开辟内存，将一些非静态的方法，变量存放在里面。在程序执行的过程中，可以创建多个对象，既多次实例化。每次实例化都会开辟一块新的内存。

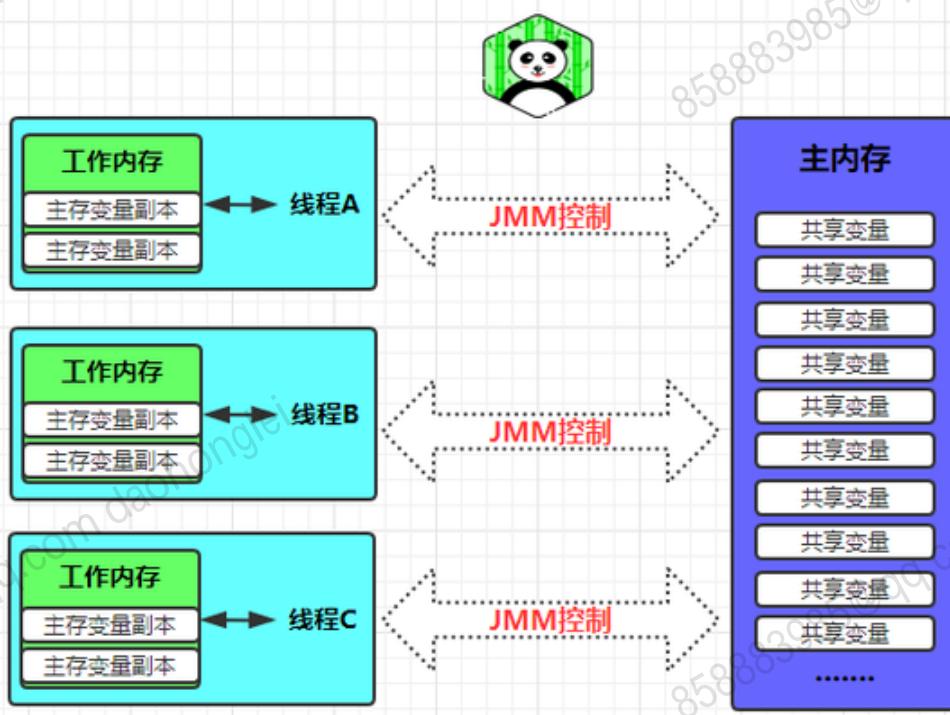
主要区别

	执行次数	触发机制	是否执行构造方法	生命周期	执行内容
初始化 (类级别)	1	Class.forName(), new, main方法的类，通过子类加载父类静态成员导致父类的初始化	否	第五步	为静态成员赋值，执行静态代码块
实例化 (实例对象)	N	new, Class.newInstance()	是	第六步	在堆区分配内存空间，执行实例对象初始化，设置引用变量a指向刚分配的内存地址

Java 内存模型 JMM 概述

Java 内存模型(即 Java Memory Model, 简称 JMM)本身是一种抽象的概念，并不真实存在，它描述的是一组规则或规范，通过这组规范定义了程序中各个变量（包括实例字段，静态字段和构成数组对象的元素）的访问方式。由于 JVM 运行程序的实体是线程，而每个线程创建时 JVM 都会为其创建一个工作内存(有些地方称为栈空间)，用于存储线程私有的数据，而 Java 内存模型中规定所有变量都存储在主内存，主内存是共享内存区域，所有线程都可以访问，但线程如果想要对一个变量读取赋值等操作那么必须在工作内存中进行，所以线程想操作变量时首先要将变量从主内存拷贝到自己的工作内存空间，然后对变量进行操作，操作完成后再将变量刷写回主内存，不能直接操作主内存中的变量，工作内存中存储着主内存中的变量副本拷贝 (PS：有些小伙伴可能会疑惑，Java 中线程在执行一个方法时就算里面引用或者创建了对象，他不是也存在堆中吗？栈内存储的不仅仅只是对象的引用地址吗？这里简单说一下，当线程真正运行到这一行时会根据局部表中的对象引用地址去找到主存中的真实对象，然后会将对象拷贝到自己的工作内存再操作……，但是当所操作的对象是一个大对象时 (1MB+) 并不会完全拷贝，而是将自己操作和需要的那部分成员拷贝），前面说过，工作内存是每个线程的私有数据区域，因此不同的线程间无法访问对方的工作内存，线程间的通信(传值)必须通过主内存来完成，其简要访问过程如下图：

JMM



重点注意！！！JMM 与 JVM 内存区域的划分是不同的概念层次，在理解 JMM 的时候不要带着 JVM 的内存模型去理解，更恰当说 JMM 描述的是一组规则，通过这组规则控制 Java 序中各个变量在共享数据区域和私有数据区域的访问方式，JMM 是围绕**原子性**，**有序性、可见性**拓展延伸的。JMM 与 Java 内存区域唯一相似点，都存在共享数据区域和私有数据区域，在 JMM 中主内存属于共享数据区域，从某个程度上讲应该包括了堆和方法区，而工作内存数据线程私有数据区域，从某个程度上讲则应该包括程序计数器、虚拟机栈以及本地方法栈。或许在某些地方，我们可能会看见主内存被描述为堆内存，工作内存被称为线程栈，实际上他们表达的都是同一个含义。

关于 JMM 中的主内存和工作内存说明如下：

主内存：主要存储的是 Java 实例对象，所有线程创建的实例对象都存放在主内存中（除开开启了逃逸分析和标量替换的栈上分配和 TLAB 分配），不管该实例对象是成员变量还是方法中的本地变量(也称局部变量)，当然也包括了共享的类信息、常量、静态变量。由于是共享数据区域，多条线程对同一个变量进行非原子性操作时可能会发现线程安全问

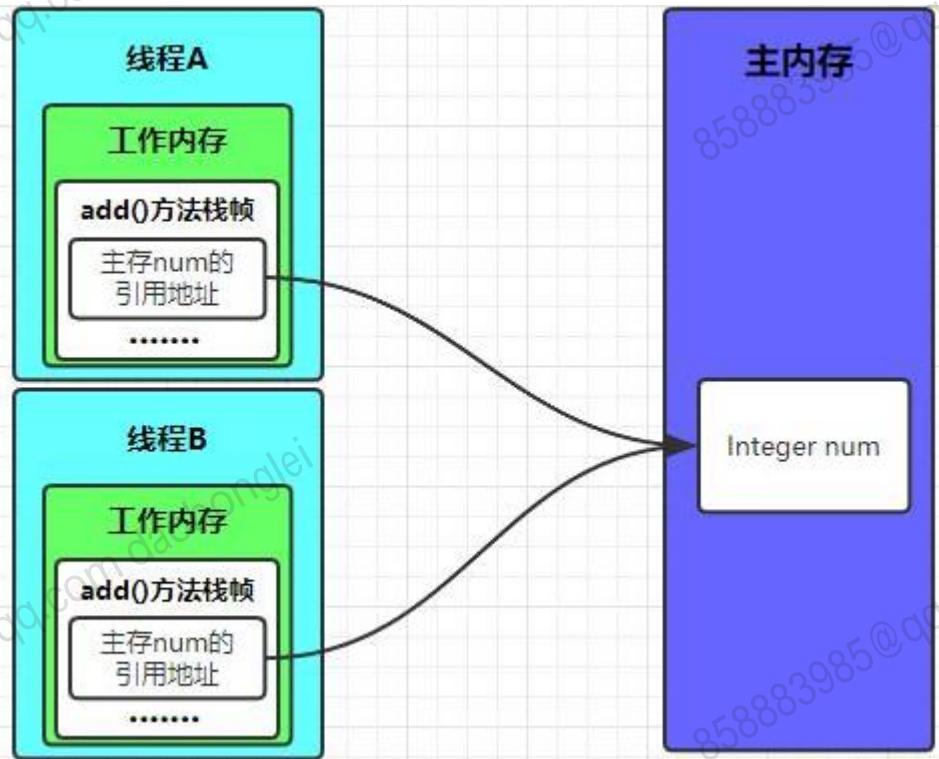
题。

工作内存：主要存储当前方法的所有本地变量信息(工作内存中存储着主内存中的变量副本拷贝)，每个线程只能访问自己的工作内存，即线程中的本地变量对其它线程是不可见的，就算是两个线程执行的是同一段代码，它们也会各自在自己的工作内存中创建属于当前线程的本地变量，当然也包括了字节码行号指示器、相关 Native 方法的信息。注意由于工作内存是每个线程的私有数据，线程间无法相互访问工作内存，线程之间的通讯还是需要依赖于主存，因此存储在工作内存的数据不存在线程安全问题。

弄清楚主内存和工作内存后，接了解一下主内存与工作内存的数据存储类型以及操作方式，根据虚拟机规范，对于一个实例对象中的成员方法而言，如果方法中包含本地变量是基本数据类型 (boolean,byte,short,char,int,long,float,double) ，将直接存储在工作内存的帧栈结构中的局部变量表，但倘若本地变量是引用类型，那么该对象的内存中的具体引用地址将会被存储在工作内存的帧栈结构中的局部变量表，而对象实例将存储在主内存(共享数据区域，堆)中。但对于实例对象的成员变量，不管它是基本数据类型或者包装类型(Integer、Double 等)还是引用类型，都会被存储到堆区 (栈上分配与 TLAB 分配除外)。至于 static 变量以及类本身相关信息将会存储在主内存中。需要注意的是，在主内存中的实例对象可以被多线程共享，倘若两条线程同时调用了同一个类的同一个方法，那么两条线程会将要操作的数据拷贝一份到自己的工作内存中，执行完成操作后才刷新到主内存，简单示意图如下所示：

```
Integer num = new Integer( value: 100 );
```

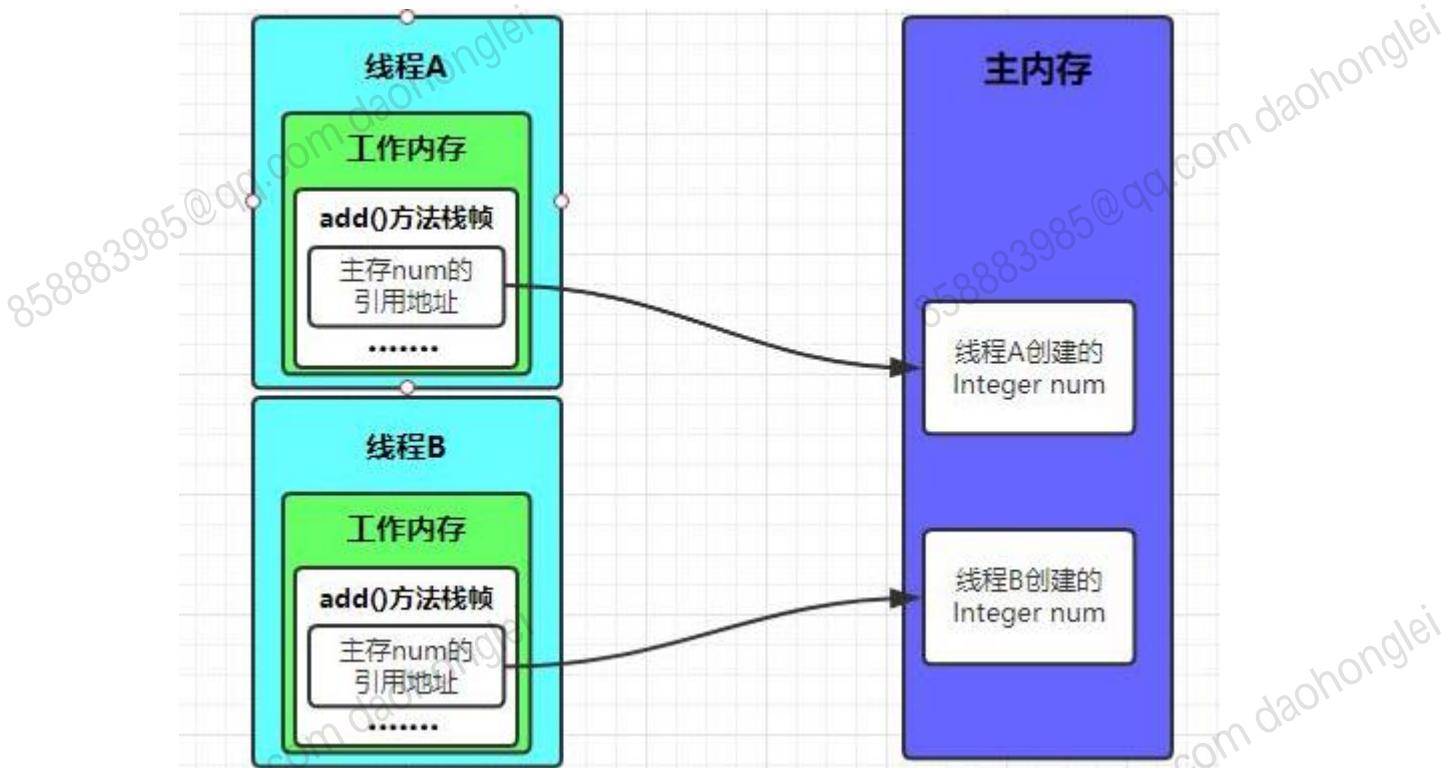
```
public void add() {  
    num++;  
}
```



```

public void add() {
    Integer num = new Integer( value: 100 );
    num++;
}

```

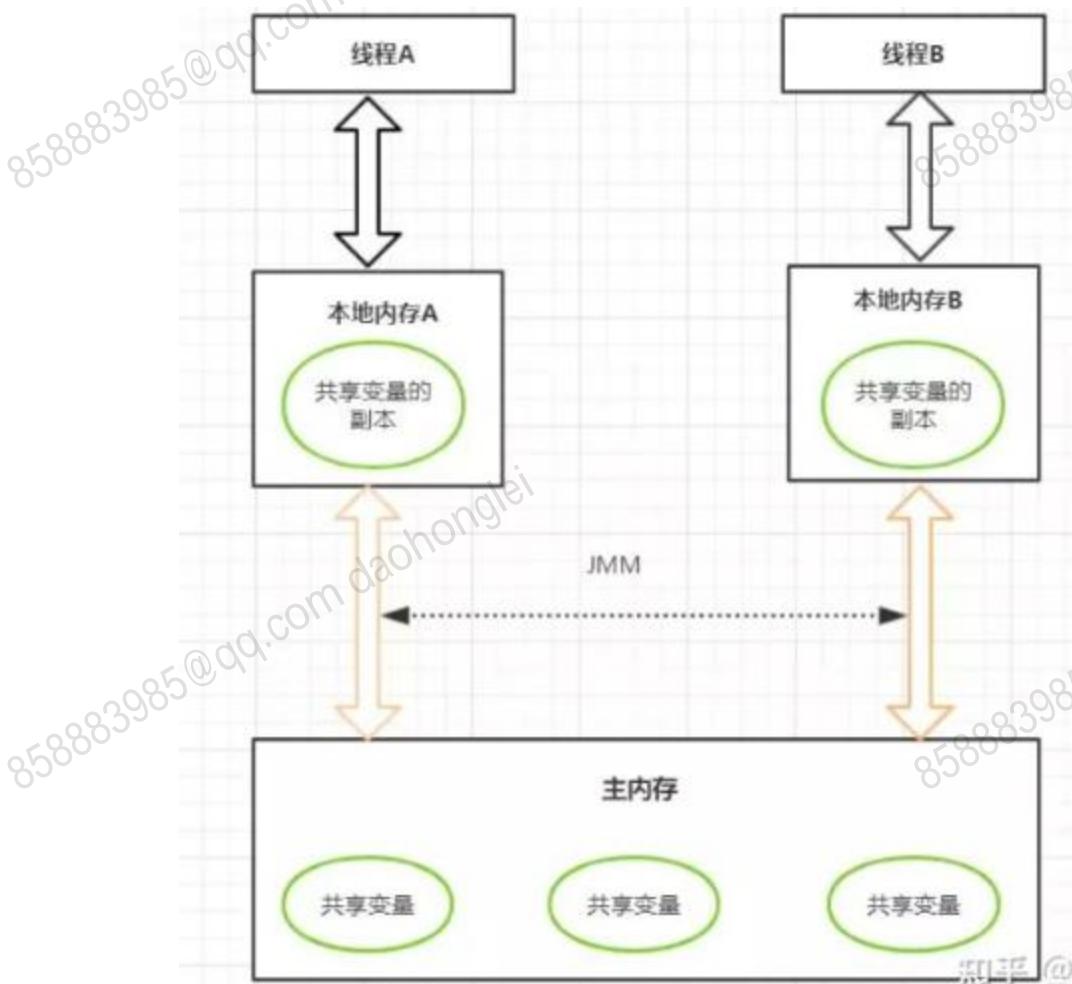


JVM 内存模型的相关知识了解多少，比如重排序，内存屏障，
happen-before，主内存，工作内存。

思路：先画出 Java 内存模型图，结合例子 volatile，说明什么是重排序，内存屏障，最好能给面试官写以下 demo 说明。

我的答案：

1) Java 内存模型图：



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了该线程中是用到的变量的主内存副本拷贝，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。

2) 指令重排序。

在这里，先看一段代码

```
1 public class PossibleReordering {
2     static int x = 0, y = 0;
3     static int a = 0, b = 0;
4
5     public static void main(String[] args) throws InterruptedException {
6         Thread one = new Thread(new Runnable() {
7             public void run() {
8                 a = 1;
9                 x = b;
10            });
11        );
12
13        Thread other = new Thread(new Runnable() {
14            public void run() {
15                b = 1;
16                y = a;
17            });
18        );
19        one.start();other.start();
20        one.join();other.join();
21        System.out.println("(" + x + "," + y + ")");
22    }
}
```

知乎 @superstar001

运行结果可能为(1,0)、(0,1)或(1,1)，也可能是(0,0)。因为，在实际运行时，代码指令可能并不是严格按照代码语句顺序执行的。大多数现代微处理器都会采用将指令乱序执行(out-of-order execution，简称 OoOE 或 OOE) 的方法，在条件允许的情况下，直接运行当前有能力立即执行的后续指令，避开获取下一条指令所需数据时造成的等待。通过乱序执行的技术，处理器可以大大提高执行效率。而这就是指令重排。

3) 内存屏障 **volatile 向前加写屏障(前面写完才能写 volatile 变量)，向后加读屏障(volatile 变量读完才能读后面变量)**

内存屏障，也叫内存栅栏，是一种 CPU 指令，用于控制特定条件下的重排序和内存可见性问题。

LoadLoad 屏障：对于这样的语句 Load1; LoadLoad; Load2，在 Load2 及后续读取操作要读取的数据被访问前，保证 Load1 要读取的数据被读取完毕。

StoreStore 屏障：对于这样的语句 Store1; StoreStore; Store2，在 Store2 及后续写入操作执行前，保证 Store1 的写入操作对其他处理器可见。

LoadStore 屏障：对于这样的语句 Load1; LoadStore; Store2，在 Store2 及后续写入操作被刷出前，保证 Load1 要读取的数据被读取完毕。

StoreLoad 屏障：对于这样的语句 Store1; StoreLoad; Load2，在 Load2 及后续所有读取操作执行前，保证 Store1 的写入对所有处理器可见。它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能。

4) happen-before 原则

单线程 happen-before 原则：在同一个线程中，书写的前面的操作 happen-before 后面的操作。锁的 happen-before 原则：同一个锁的 unlock 操作 happen-before 此锁的 lock 操作。

volatile 的 happen-before 原则：对一个 volatile 变量的写操作 happen-before 对此变量的任意操作(当然也包括写操作了)。

happen-before 的传递性原则：如果 A 操作 happen-before B 操作，B 操作 happen-before C 操作，那么 A 操作 happen-before C 操作。

线程启动的 happen-before 原则：同一个线程的 start 方法 happen-before 此线程的其它方法。

线程中断的 happen-before 原则：对线程 interrupt 方法的调用 happen-before 被中断线程的检测到中断发送的代码。

线程终结的 happen-before 原则：线程中的所有操作都 happen-before 线程的终止检测。

对象创建的 happen-before 原则：一个对象的初始化完成先于他的 finalize 方法调用。

内存溢出和内存泄漏的区别？

内存溢出 OutOfMemory，指程序在申请内存时，没有足够的内存空间供其使用。

内存泄露 Memory Leak，指程序在申请内存后，无法释放已申请的内存空间，内存泄漏

最终将导致内存溢出。

Java 内存溢出异常及其处理

一、在 Java 语言中，对象访问是如何进行的？

即使是最简单的访问，也会涉及 Java 栈、Java 堆、方法区这三个最重要内存区域之间的
的关联关系。比如下面这行代码：

```
Object obj = new Object();
```

简单分析一下这行看似普通的代码：

首先，“Object obj” 这部分的语义将会反映到 Java 栈的本地变量表中，作为一个
reference 类型数据出现。

其次，“new Object()” 这部分语义将会反映到 Java 堆中，形成一块存储了 Object 类型
所有实例数据值的结构化内存。

另外，在 Java 堆中还必须包含能查到此对象类型数据（如对象类型、父类、实现的接口、
方法等）的地址信息，这些类型的数据类型存储在方法区中。

而且，不同虚拟机实现的对象访问方式会有所不同，主流的访问方式有两种：使用句柄
和直接指针。这两种方式应用都十分广泛。

句柄：Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄
地址，而句柄中包含了对象实例数据和类型数据各自的具体地址信息。优点是：reference

中存储的是稳定的句柄地址只移动指针，不会修改 reference 本身。

直接指针：Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，

reference 中直接存储的就是对象地址。优点是：速度快，节省开销。

二、有关于 OutOfMemoryError 异常的实战处理

目的：

通过代码验证 Java 虚拟机规范中描述的各个运行时区域储存的内容。

希望开发者能根据异常的信息快速判断是哪个区域的内存溢出，知道怎样的代码可能会导

致这些区域的内存溢出，以及出现这些异常后该如何处理。

1. Java 堆溢出

Java 堆用于存储对象实例，我们只要不断地创建对象，并且保证 GC Roots 到对象之间有可达路径来避免垃圾回收机制来清除这些对象，就会在对象数量到达最大堆的容量限制后产生内存溢出异常。

首先，限制 Java 堆的大小为 20MB，不可扩展（将堆的最小值-Xms 参数与最大值-Xmx 参数设置为一样即可避免堆自动扩展），通过参数-XX：

+HeapDumpOnOutOfMemoryError 可以让虚拟机在出现内存溢出异常时 Dump 出当前的内存堆转储快照以便事后进行分析。以下是在 Mac 下使用 idea 设置虚拟机参数：

第一步：打开 “Run->Edit Configurations” 菜单

第二步：选择 “VM Options” 选项，输入你要设置的 VM 参数

```
-verbose: gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -  
XX:SurvivorRatio=8
```

```
package com.OOM;
```

```
import java.util.ArrayList;  
  
import java.util.List;  
  
public class HeapOOM {  
  
    static class OOMObject {  
    }  
  
    public static void main(String[] args) {  
  
        List<OOMObject> list = new ArrayList<>();  
  
        while (true) {  
  
            list.add(new OOMObject());  
  
        }  
  
    }  
}
```

以上代码的运行结果：不光给出了错误原因是 OutOfMemoryError，而且指明了是在堆内存中发生的。下面还有堆中各个分区的信息，相比于 eclipse，IDEA 人性化了很多。

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space  
  
at java.util.Arrays.copyOf((Arrays.java:3210)  
  
at java.util.Arrays.copyOf((Arrays.java:3181)  
  
at java.util.ArrayList.grow(ArrayList.java:265)  
  
at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:239)  
  
at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:231)
```

```
at java.util.ArrayList.add(ArrayList.java:462)  
at com.OOM.HeapOOM.main(HeapOOM.java:22)
```

Heap

```
PSYoungGen    total 9216K, used 7786K [0x00000007bf600000,  
0x00000007c0000000, 0x00000007c0000000)  
    eden space 8192K, 95% used  
[0x00000007bf600000,0x00000007bfd9aa80,0x00000007bfe00000)  
    from space 1024K, 0% used  
[0x00000007bff00000,0x00000007bff00000,0x00000007c0000000)  
    to   space 1024K, 0% used  
[0x00000007bfe00000,0x00000007bfe00000,0x00000007bff00000)  
ParOldGen    total 10240K, used 8783K [0x00000007bec00000,  
0x00000007bf600000, 0x00000007bf600000)  
    object space 10240K, 85% used  
[0x00000007bec00000,0x00000007bf493f08,0x00000007bf600000)  
Metaspace    used 3122K, capacity 4500K, committed 4864K, reserved  
1056768K  
    class space   used 341K, capacity 388K, committed 512K, reserved 1048576K
```

那么，应该如何解决这个异常呢？以下是解决方向：

- 通过内存映像分析功能根据对 dump 出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，也就是要先分清楚到底是出现了内存泄露还是内存溢出。

- 如果是内存泄露，可进一步通过工具查看泄露对象到 GC Roots 的引用链。
- 如果不存在泄露，那就应当检查虚拟机的堆参数（-Xmx 与-Xms），与机器物理内存对比看是否还可以调大，从代码上检查是否存在某些对象的生命周期过长，持有状态时间过长的情况，尝试减少程序运行期的内存消耗。

2. 虚拟机栈和本地方法栈溢出

由于在 HotSpot 虚拟机中并不区分虚拟机栈和本地方法栈，因此对于 HotSpot 来说，
-Xoss 参数（设置本地方法栈大小）虽然存在，但实际上无效的，栈容量只由-Xss 参数

设定。关于虚拟机栈和本地方法栈，在 Java 虚拟机规范中描述了两种异常：

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError 异常

如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常

- 使用-Xss 参数减少栈内存容量。结果：抛出 StackOverflowError 异常，异常出现时输出的栈深度相应缩小。
- 定义了大量的本地变量，增加此方法帧中本地变量表的长度。结果：抛出 StackOverflowError 异常时输出的栈深度相应缩小。

```
package com.OOM;

public class JavaVMstackSOF {

    private int stackLength = 1;

    public void stackLeak() {

        stackLength++;

        stackLeak();
    }
}
```

```
public static void main(String[] args) {  
    JavaVMstackSOF oom = new JavaVMstackSOF();  
  
    try {  
  
        oom.stackLeak();  
  
    } catch (Throwable throwable) {  
  
        System.out.println("stack length:" + oom.stackLength);  
  
        throw throwable;  
  
    }  
  
}  
}
```

```
stack length:20898  
  
Exception in thread "main" java.lang.StackOverflowError  
  
at com.OOM.JavaVMstackSOF.stackLeak(JavaVMstackSOF.java:13)
```

实验结果表明：

- 在单个线程下，无论是由于栈帧太大，还是虚拟机栈容量太小，当内存无法分配的时候，虚拟机抛出的都是 StackOverflowError 异常。
- 但是，如果是建立过多线程导致的内存溢出，在不能减少线程数或者更换 64 位虚拟机的情况下，就只能通过减少最大堆和减少栈容量来换取更多的线程。

```
package com.OOM;  
  
public class JavaVMStackOOM {
```

```
private void dontStop() {  
    while (true) {  
        }  
    }  
  
private void stackLeakByThread() {  
    while (true) {  
        Thread thread = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                dontStop();  
            }  
        });  
        thread.start();  
    }  
}  
  
public static void main(String[] args) {  
    JavaVMStackOOM oom = new JavaVMStackOOM();  
    oom.stackLeakByThread();  
}
```

```
}
```

注意：在 windows 平台的虚拟机执行以上代码有较大风险，可能会造成操作系统假死。

运行结果：

```
Exception in thread "main" java.lang.OutOfMemoryError:unable to create new  
native thread
```

3.运行时常量池溢出

如果要向运行时常量池中添加内容，最简单的做法就是使用 String.intern()这个 Native 方法。该方法的作用是：如果池中已经包含一个等于此 String 对象的字符串，则返回代表池中这个字符串的 String 对象的引用。由于常量池分配在方法区内，我们可以通过-XX:PermSize 和-XX:MaxPermSize 限制方法区的大小，从而间接限制其中常量池的容量。

下面将展示代码实例，在运行代码之前同样需要设置虚拟机参数，如下：

```
-XX:PermSize=10 -XX:MaxPermSize=10 -XX:MetaspaceSize
```

```
package com.OOM;  
  
import java.util.ArrayList;  
  
import java.util.List;  
  
  
  
public class RuntimeConstantPoolOOM {  
  
    public static void main(String[] args) {  
  
        // 使用 list 保持着常量池引用，避免 FULL GC 回收常量池行为  
  
        List<String> list = new ArrayList<String>();
```

```
// 10MB 的 PermSize 在 integer 范围内足够产生 OOM 了

int i = 0;

while (true) {

    System.out.println("i:" + i);

    list.add(String.valueOf(i++).intern());

}

}
```

运行结果：

```
Exception in thread "main" java.lang.OutOfMemoryError:PermGen space
```

结论：

运行结果表示，产生 OOM 的部分是 PermGen space，说明运行时常量池属于方法区 (HotSpot 虚拟机中的永久代) 的一部分。

4.方法区溢出

方法区用于存放 Class 的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。

以下代码是模拟方法区溢出时的情形，在 Spring 和 hibernate 对类进行增强时也会用到 CGLib 这类字节码技术，增强的类越多，就需要越大的方法区来保证动态生成的 Class 可以加载入内存。

```
package com.OOM;

import net.sf.cglib.proxy.Enhancer;
```

```
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;

public class JavaMethodAreaOOM {

    public static void main(String[] args) {
        while (true) {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(OOMObject.class);
            enhancer.setUseCache(false);
            enhancer.setCallback(new MethodInterceptor() {
                @Override
                public Object intercept(Object o, Method method, Object[] objects,
                        MethodProxy methodProxy) throws Throwable {
                    return methodProxy.invokeSuper(o, objects);
                }
            });
            enhancer.create();
        }
    }

    static class OOMObject {
    }
}
```

```
}
```

运行结果：

```
Caused by:java.lang.OutOfMemoryError:PermGen space
```

需要注意的是，如果你的 JDK 版本是 1.8.那么控制台会提示你：

```
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=10M;
```

```
support was removed in 8.0
```

```
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=10M;
```

```
support was removed in 8.0
```

也就是说 JDK1.8 版本不支持这两个参数配置。

5.本机直接内存溢出

DirectMemory 容量可通过-XX: MaxDirectMemorySize 指定，如果不指定，则默认与 Java 堆的最大值 (-Xms 指定) 一样。

代码如下：

```
package com.OOM;

import sun.misc.Unsafe;
import java.lang.reflect.Field;

public class DirectMemoryOOM {

    private static final int _1MB = 1024 * 1024;

    public static void main(String[] args) throws Exception {
        Field unsafedField = Unsafe.class.getDeclaredFields()[0];
```

```
        unsafedField.setAccessible(true);

        Unsafe unsafe = (Unsafe) unsafedField.get(null);

        while (true) {

            unsafe.allocateMemory(_1MB);

        }

    }

}
```

运行结果：

```
# 引用原书中的运行结果

Exception in thread "main" java.lang.OutOfMemoryError
```

对象分配内存是否线程安全？

对象创建十分频繁，即使修改一个指针的位置在并发下也不是线程安全的，可能正给对象 A 分配内存，指针还没来得及修改，对象 B 又使用了指针来分配内存。

解决方法：① CAS 加失败重试保证更新原子性。② 把内存分配按线程划分在不同空间，即每个线程在 Java 堆中预先分配一小块内存，叫做本地线程分配缓冲 TLAB，哪个线程要分配内存就在对应的 TLAB 分配，TLAB 用完了再进行同步。

栈上分配和 TLAB 分配

TLAB，全称 Thread Local Allocation Buffer

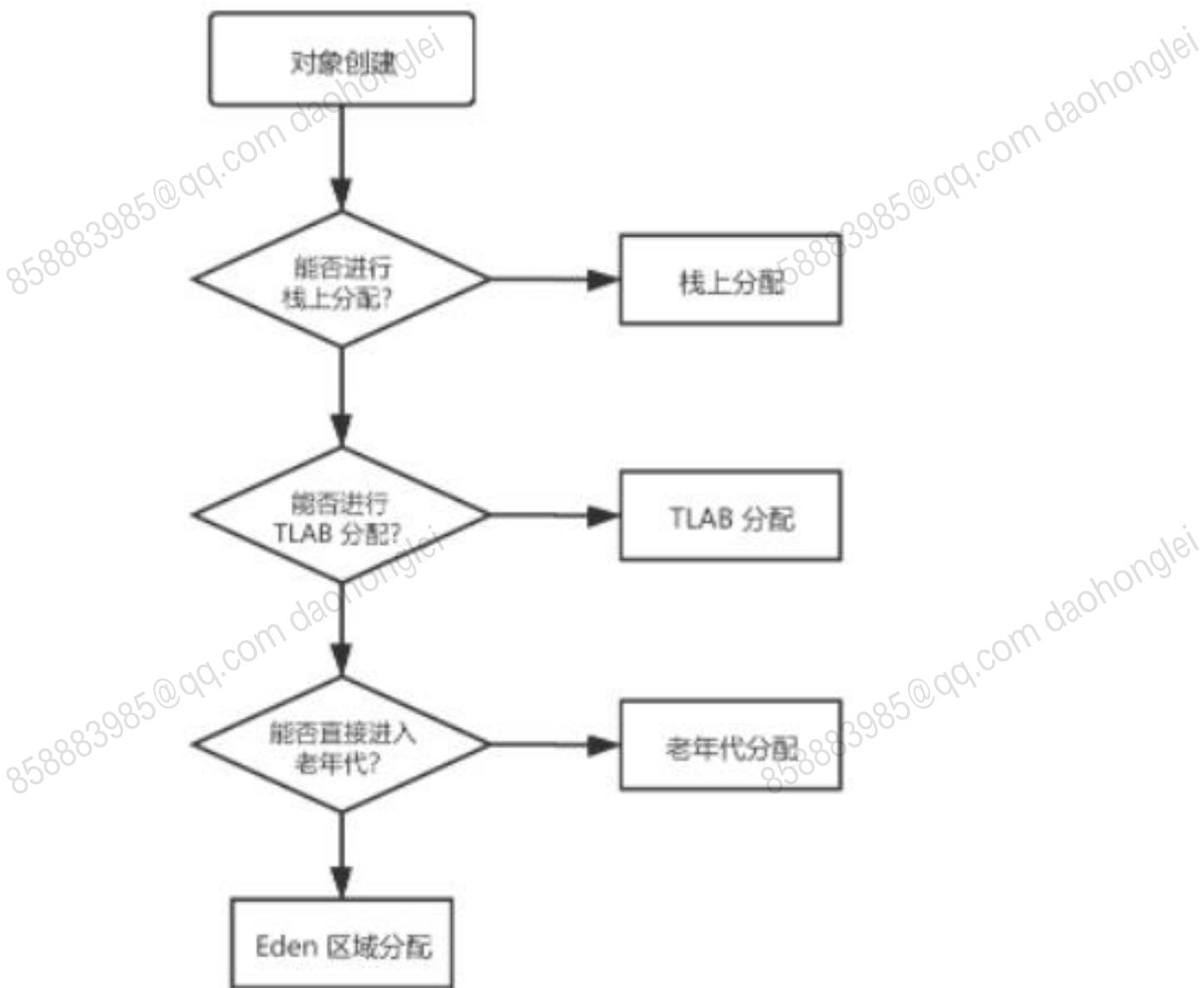
内存区域

- 栈上分配使用的是栈来进行对象内存的分配。

- TLAB 分配使用的是 **Eden** 区域进行内存分配，实际还是属于堆内存。

优先级

- 栈上分配优先于 TLAB 分配进行，逃逸分析中若可进行栈上分配优化，会优先进行对
象栈上直接分配内存。
- 当无法进行栈上直接分配时，则会进行 TLAB 分配。

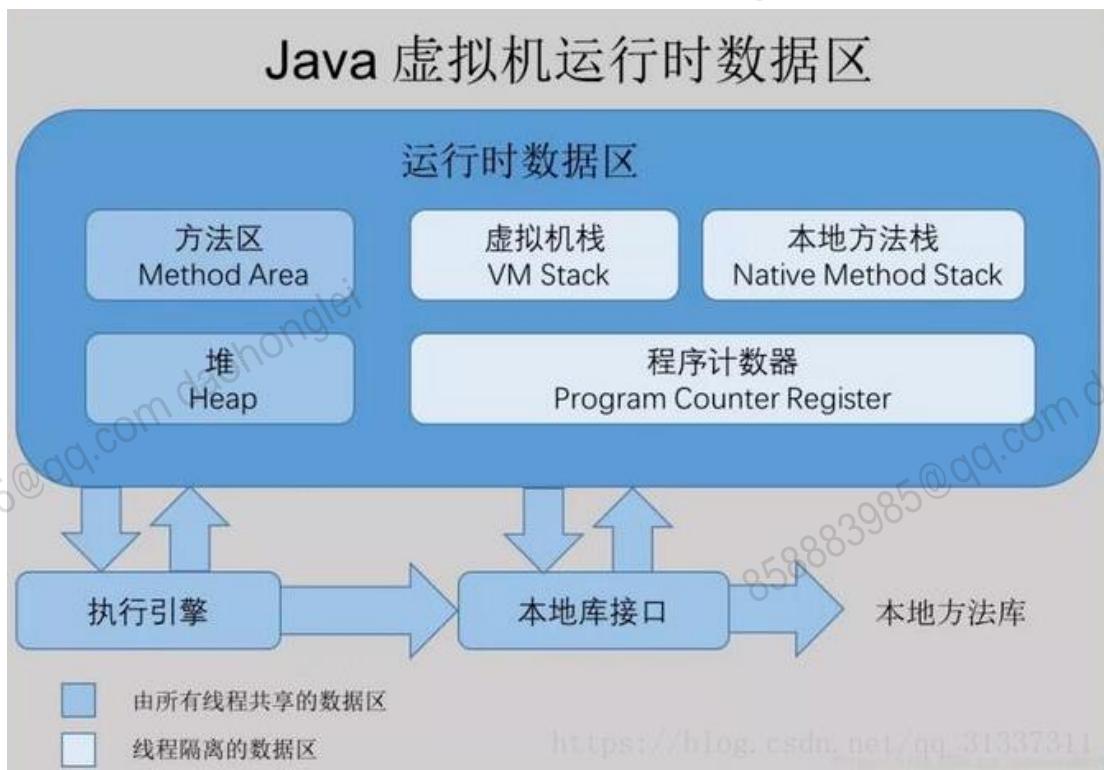


Java8 虚拟机内存模型

1. Java 虚拟机运行时数据区

在 JDK1.8 之前，JVM 运行时数据区分为堆、虚拟机栈、本地方法栈、方法区、

程序计数器。如下图所示：



虚拟机栈：线程私有，随线程创建而创建。栈里面是一个一个“栈帧”，每个栈帧对应一次方法调用。栈帧中存放了**局部变量表**（基本数据类型变量和对象引用）、**操作数栈**、**动态连接**、**返回地址**等信息。当栈调用深度大于 JVM 所允许的范围，会抛出 **StackOverflowError** 的错误。

本地方法栈：线程私有，这部分主要与虚拟机用到的 Native 方法相关，一般情况下，并不需要关心这部分的内容。

程序计数器：也叫 PC 寄存器，JVM 支持多个线程同时运行，每个线程都有自己
的程序计数器。**倘若当前执行的是 JVM 的方法，则该寄存器中保存当前执行指令的地
址**；倘若执行的是 native 方法，则 PC 寄存器中为空。**(PS：线程执行过程中并不都
是一口气执行完，有可能在一个 CPU 时钟周期内没有执行完，由于时间片用完了，所
以不得不暂停执行，当下一次获得 CPU 资源时，通过程序计数器就知道该从什么地方**

开始执行)

方法区: 方法区存放类的信息 (包括类的字节码, 类的结构) 、常量、静态变量等。**字符串常量池就是在方法区中**。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分, 但是它却有一个别名叫做 Non-Heap (非堆), 目的是与 Java 堆区分开来。很多人都更愿意把方法区称为 “永久代” (Permanent Generation)。从 jdk1.7 已经开始准备 “去永久代”的规划, **jdk1.7 的 HotSpot 中, 已经把原本放在方法区中的静态变量、字符串常量池等移到堆内存中。**

堆: 堆中存放的是数组 (**PS: 数组也是对象**) 和对象。当申请不到空间时会抛出 **OutOfMemoryError**。

2. PermGen (永久代)

“方法区”是 JVM 的规范, 而 “永久代”是方法区的一种实现, 并且只有 HotSpot 才有 “PermGen space”, 而对于其他类型的虚拟机并没有 “PermGen space”。

在 JDK1.8 中, HotSpot 已经没有 “PermGen space” 这个区间了, 取而代之是 Metaspace (元空间)

3. Metaspace (元空间)

在 JDK1.8 中, 永久代已经不存在, **存储的类信息、编译后的代码数据等已经移动到了 MetaSpace (元空间) 中**, 元空间并没有处于堆内存上, 而是直接占用的本地内存 (NativeMemory)。

元空间的本质和永久代类似, 都是对 JVM 规范中方法区的实现。

不过元空间与永久代之间最大的区别在于: **元空间并不在虚拟机中, 而是使用本地内存**。

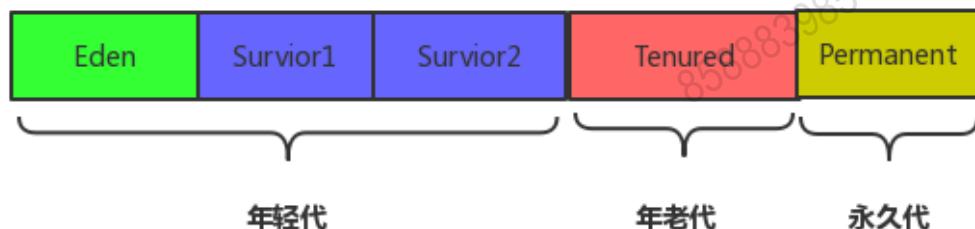
元空间的大小仅受本地内存限制，可以通过以下参数来指定元空间大小：

- `-XX:MetaspaceSize`, 初始空间大小，达到该值就会触发垃圾收集进行类型卸载，同时 GC 会对该值进行调整：如果释放了大量的空间，就适当降低该值；如果释放了很少的空间，那么在不超过 `MaxMetaspaceSize` 时，适当提高该值
- `-XX:MaxMetaspaceSize`, 最大空间，默认是没有限制的
- `-XX:MinMetaspaceFreeRatio`, 在 GC 之后，最小的 Metaspace 剩余空间容量的百分比，减少为分配空间所导致的垃圾收集
- `-XX:MaxMetaspaceFreeRatio`, 在 GC 之后，最大的 Metaspace 剩余空间容量的百分比，减少为释放空间所导致的垃圾收集

4. 堆内存划分

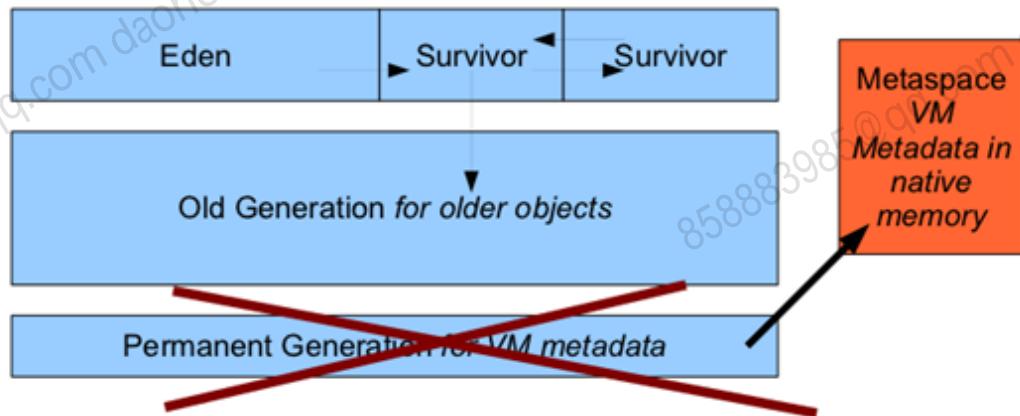
在 JDK1.7 以及其前期的 JDK 版本中，堆内存通常被分为三块区域：Young

Generation、Old Generation、Permanent Generation for VM Matedata



在 JDK1.8 中把存放元数据中的永久内存从堆内存中移到了本地内存中，JDK1.8

中 JVM 堆内存结构就变成了如下：



永久代转移到元空间原因：

- 1、字符串存在永久代中，容易出现性能问题和内存溢出。

- 2、类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。
- 3、永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。
- 4、Oracle 可能会将 HotSpot 与 JRockit 合二为一。

为什么要把堆和栈区分出来呢

- 1.从软件设计的角度来看，栈代表了处理逻辑，而堆代表了数据，这样分离使得处理逻辑更为清晰。这种隔离、模块化的思想在软件设计的方方面面都有体现。
- 2.堆与栈的分离，使得堆中的内容可以被多个栈共享。这种共享有很多好处，一方面提供了一种有效的数据交互方式（如内存共享），另一方面，节省了内存空间。
- 3.栈因为运行时的需要（如保存系统运行的上下文），需要进行址段的划分。由于栈只能向上增长，因此会限制住栈存储内容的能力。而堆不同，堆的大小可以根据需要动态增长。因此，堆与栈的分离，使得动态增长成为可能，相应栈中只需要记录堆中的一个地址即可。
- 4.堆和栈的完美结合就是面向对象的一个实例。其实，面向对象的程序与以前结构化的程序在执行上没有任何区别，但是面向对象的引入使得对待问题的思考方式发生了改变，是更接近于自然的思考方式。当把对象拆开会发现，对象的属性其实就是数据，存放在堆中，而对象的方法就是处理逻辑，存放在栈中。我们编写对象的时候，其实即编写了数据结构，也编写了处理数据的逻辑。
总结：栈主要用来执行程序，堆主要用来存放对象，为栈提供数据存储服务。也正是因为堆与栈分离的思想才使得 JVM 的垃圾回收成为可能。

直接内存是什么？

直接内存不属于运行时数据区，也不是虚拟机规范定义的内存区域，但这部分内存被频繁使用，而且可能导致内存溢出。

JDK1.4 中新加入了 NIO 这种基于通道与缓冲区的 IO，它可以使用 Native 函数库直接分配堆外内存，通过一个堆里的 DirectByteBuffer 对象作为内存的引用进行操作，避免了在 Java 堆和 Native 堆来回复制数据。

直接内存的分配不受 Java 堆大小的限制，但还是会受到本机总内存及处理器寻址空间限制，一般配置虚拟机参数时会根据实际内存设置 -Xmx 等参数信息，但经常忽略直接内存，使内存区域总和大于物理内存限制，导致动态扩展时出现 OOM。

由直接内存导致的内存溢出，一个明显的特征是在 Heap Dump 文件中不会看见明显的异常，如果发现内存溢出后产生的 Dump 文件很小，而程序中又直接或间接使用了直接内存（典型的间接使用就是 NIO），那么就可以考虑检查直接内存方面的原因。

对象的访问定位

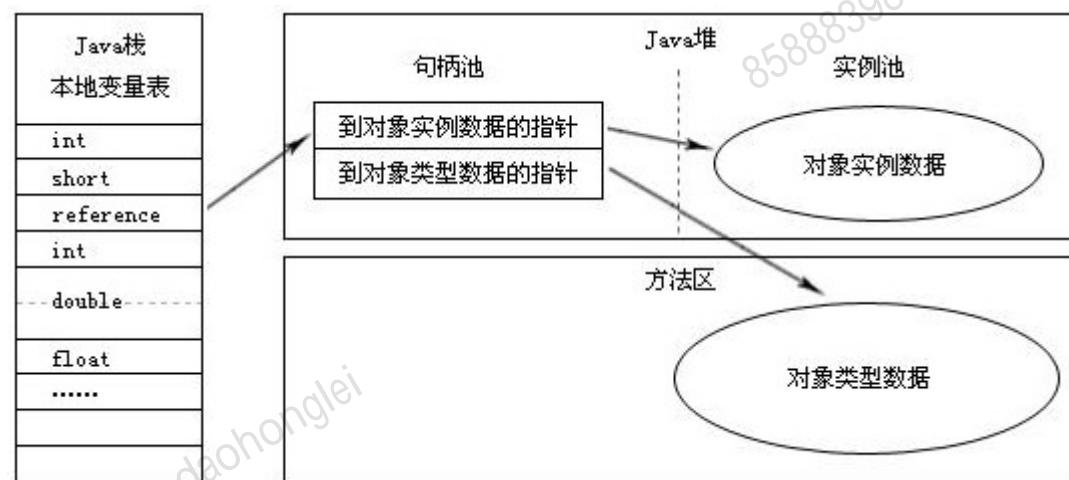
我们的 Java 程序需要通过栈上的对象引用（reference）数据（存储在栈上的局部变量表中）来操作堆上的具体对象。

由于 reference 类型在 Java 虚拟机规范里面也只规定了一个指向对象的引用，并没有定义这个引用的具体实现，对象访问方式也是取决于虚拟机实现而定的。主流的访问方式有使用句柄和直接指针两种。

1、使用句柄访问

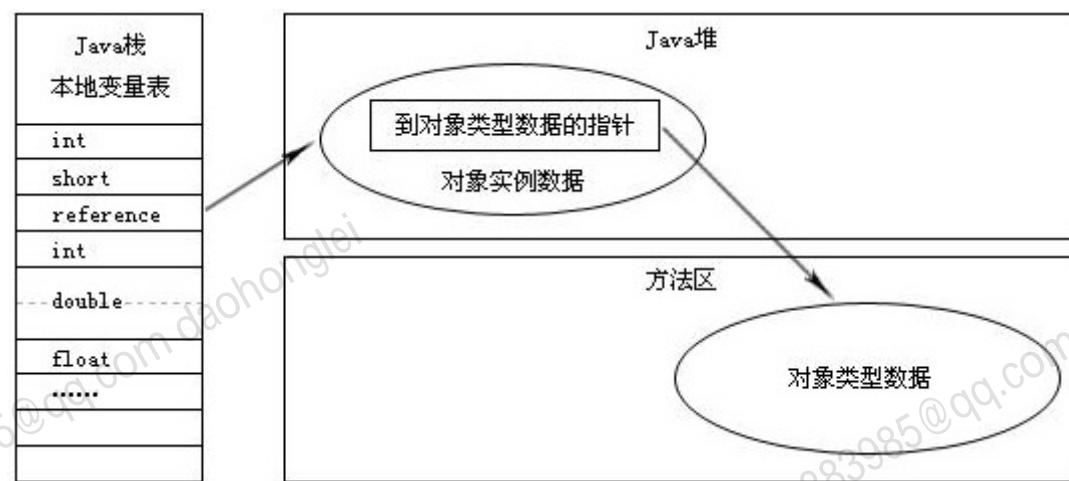
如果使用句柄访问的话，Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的地址

信息。如下图所示：



2、使用直接指针访问

如果使用直接指针访问的话，Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，reference 中存储的直接就是对象地址，如下图所示：



这两种对象访问方式各有优势，下面分别来谈一谈：

(1) 句柄

使用句柄访问的最大好处就是 reference 中存储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而 reference 本身不需要被修改。

(2) 直接指针

使用直接指针来访问最大的好处就是速度更快，它节省了一次指针定位的时间开销，由于对象访问的在 Java 中非常频繁，因此这类开销积小成多也是一项非常可观的执行成本。

从上一部分讲解的对象内存布局可以看出，HotSpot 是使用直接指针进行对象访问的，不过在整个软件开发的范围来看，各种语言、框架中使用句柄来访问的情况也十分常见。

如何判断对象是否是垃圾？

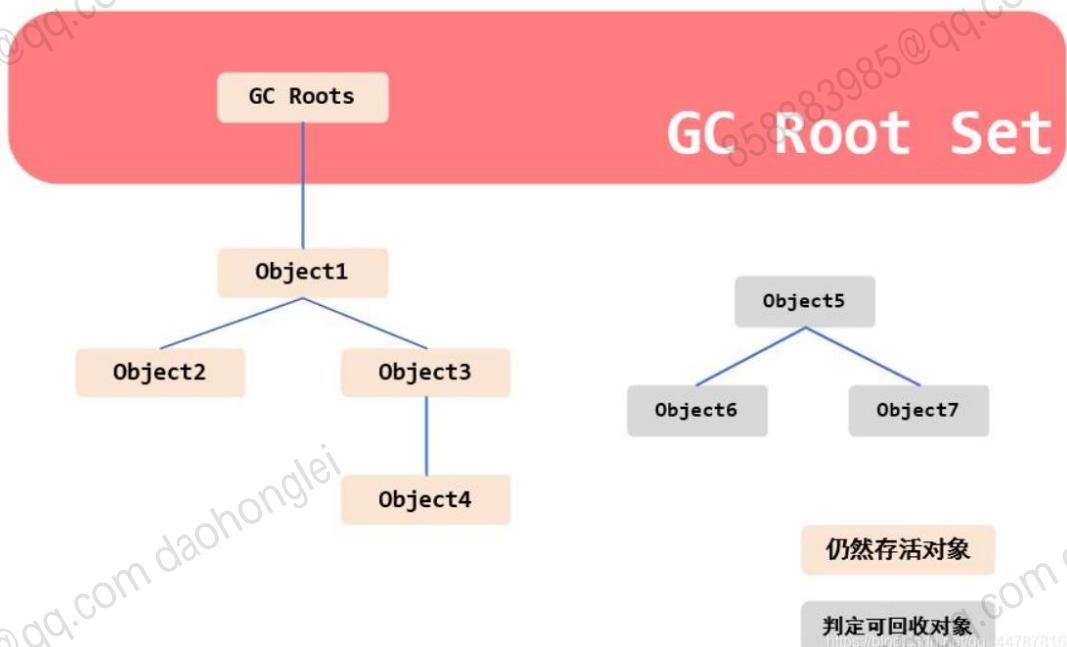
引用计数：在对象中添加一个引用计数器，如果被引用计数器加 1，引用失效时计数器减 1，如果计数器为 0 则被标记为垃圾。原理简单，效率高，但是在 Java 中很少使用，因为存在对象间循环引用的问题，导致计数器无法清零。

可达性分析：主流语言的内存管理都使用可达性分析判断对象是否存活。基本思路是通过一系列称为 GC Roots 的根对象作为起始节点集，从这些节点开始，根据引用关系向下搜索，搜索过程走过的路径称为引用链，如果某个对象到 GC Roots 没有任何引用链相连，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。

什么是 GC root，GC root 原理

1、GC root 原理

GC root 原理：通过对枚举 GCroot 对象做引用可达性分析，即从 GC root 对象开始，向下搜索，形成的路径称之为引用链。如果一个对象到 GC roots 对象没有任何引用，没有形成引用链，那么该对象等待 GC 回收。



2、GC root 对象是什么？

Java 中可以作为 GC Roots 的对象

- 1、虚拟机栈 (javaStack) (栈帧中的局部变量区，也叫做局部变量表) 中引用的对象。
- 2、本地方法栈中 JNI(Native 方法)引用的对象。
- 3、类的静态属性引用的对象。
- 4、常量池中常量引用的对象。
- 5、所有被同步锁持有的对象
- 6、反应 Java 虚拟机内部情况的 JMXBean, JVMTI 中注册的回调，本地代码缓存等
- 7、Java 虚拟机内部的引用，如基本数据类型对应的 Class 对象，一些常驻的异常对象 (NPE)，类加载器等等

OopMap

OopMap 用于枚举 GC Roots，记录栈中引用数据类型的位置

一个线程为一个栈，一个栈由多个栈桢组成，一个栈桢对应一个方法，一个方法有多个安全点。GC 发生时，程序首先运行到最近的一个安全点停下来，然后更新自己的 OopMap，记录栈上哪些位置代表着引用。枚举根节点时，递归遍历每个栈桢的 OopMap，通过栈中记录的被引用的对象内存地址，即可找到这些对象（GC Roots）

安全点

在虚拟机在进行可达性分析时，HotSpot 虚拟机会在特定的位置记录在哪有引用，这些特定的位置就叫做安全点。

通过 oopMap 可以快速进行 GCROOT 枚举，但是随着而来的又有一个问题，就是在方法执行的过程中，可能会导致引用关系发生变化，那么保存的 OopMap 就要随着变化。如果每次引用关系发生了变化都要去修改 OopMap 的话，这又是一件成本很高的事情。所以这里就引入了安全点的概念。

并不需要一发生改变就去更新这个映射表。只要这个更新在 GC 发生之前就可以了。所以 OopMap 只需要在预先选定的一些位置上记录变化的 OopMap 就行了。这些特定的点就是 SafePoint（安全点）。由此也可以知道，程序并不是在所有的位置上都可以进行 GC 的，只有在达到这样的安全点才能暂停下来进行 GC。

如何让程序在要进行 GC 的时候都跑到最近的安全点上停顿下来，1. 抢断式中断，2. 主动式中断，目前都是主动式，设置一个标志，当程序运行到安全点时就去轮训该位置，发现该位置被设置为真时就自己中断挂起为什么呢？

因为可达性分析算法必须是在一个确保一致性的内存快照中进行。如果在分析的过程中对

象引用关系还在不断变化，分析结果的准确性就不能保证。

而 JVM 中对于安全点的定义主要有如下几种：

- 循环结束的末尾段
- 方法调用之后
- 抛出异常的位置
- 方法返回之前

安全区域

安全点的使用似乎解决了 OopMap 计算的效率的问题，但是这里还有一个问题。安全点需要程序自己跑过去，那么对于那些已经停在路边休息或者看风景的程序（比如那些处在 Sleep 或者 Blocked 状态的线程），他们可能并不会在很短的时间内跑到安全点去。所以这里为了解决这个问题，又引入了安全区域的概念。

安全区域很好理解，就是在程序的一段代码片段中并不会导致引用关系发生变化，也就不用去更新 OopMap 表了，那么在这段代码区域内任何地方进行 GC 都是没有问题的。这段区域就称之为安全区域。线程执

行的过程中，如果进入到安全区域内，就会标志自己已经进行到安全区域了。那么虚拟机要进行 GC 的时候，就不会管这些已经运行到安全区域的线程，当线程要脱离安全区域的时候，要自己检查系统是否已经完成了 GC 或者根节点枚举（这个跟 GC 的算法有关），如果完成了就继续执行，如果未完成，它就必须等待收到可以安全离开安全区域的 Safe Region 的信号为止。

JVM 何时会 Stop the world

1. 定时进入 SafePoint：每经过-XX:GuaranteedSafepointInterval 配置的时间，都会

让所有线程进入 Safepoint，一旦所有线程都进入，立刻从 Safepoint 恢复。这个定时主要是为了一些没必要立刻 Stop the world 的任务执行，可以设置-XX:GuaranteedSafepointInterval=0 关闭这个定时，我推荐是关闭。

2. 由于 jstack, jmap 和 jstat 等命令，也就是 **Signal Dispatcher** 线程要处理的大部分命令，都会导致 Stop the world：这种命令都需要采集堆栈信息，所以需要所有线程进入 Safepoint 并暂停。
3. 偏向锁取消（这个不一定会引发整体的 Stop the world，参考 JEP 312: Thread-Local Handshakes）：Java 认为，锁大部分情况是没有竞争的（某个同步块大多数情况都不会出现多线程同时竞争锁），所以可以通过偏向来提高性能。即在无竞争时，之前获得锁的线程再次获得锁时，会判断是否偏向锁指向我，那么该线程将不用再次获得锁，直接就可以进入同步块。但是高并发的情况下，偏向锁会经常失效，导致需要取消偏向锁，取消偏向锁的时候，需要 Stop the world，**因为要获取每个线程使用锁的状态以及运行状态。**
4. Java Instrument 导致的 Agent 加载以及类的重定义：由于涉及到类重定义，需要修改栈上和这个类相关的信息，所以需要 Stop the world，JVMTI
5. Java Code Cache 相关：**当发生 JIT 编译优化或者去优化，需要 OSR 或者 Bailout** 或者清理代码缓存的时候，由于需要读取线程执行的方法以及改变线程执行的方法，所以需要 Stop the world
6. GC：这个由于需要每个线程的对象使用信息，以及回收一些对象，释放某些堆内存或者直接内存，所以需要 Stop the world
7. JFR 的一些事件：如果开启了 JFR 的 OldObject 采集，这个是定时采集一些存活时间比较久的对象，所以需要 Stop the world。同时，JFR 在 dump 的时候，由于每个

线程都有一个 JFR 事件的 buffer，需要将 buffer 中的事件采集出来，所以需要 Stop the world。

8. 其他的事件，不经常遇到，可以参考 JVM 源码 vmOperations.hpp:

```
# 打开 safepoint 日志，日志会输出到 jvm 进程的标准输出里  
-XX:+PrintSafepointStatistics -XX:PrintSafepointStatisticsCount=1  
  
# 当有线程进入 Safepoint 超过 2000 毫秒时，会认为进入 Safepoint 超时了，这时会打  
印哪些线程没有进入 Safepoint  
-XX:+SafepointTimeout -XX:SafepointTimeoutDelay=2000  
  
# 没有这个选项，JIT 编译器可能会优化掉 for 循环处的 safepoint，那么直到循环结束线  
程才能进入 safepoint，而加了这个参数后，每次 for 循环都能进入 safepoint 了，建议加  
上此选项  
-XX:+UseCountedLoopSafepoints  
  
# 在高并发应用中，偏向锁并不能带来性能提升，反而会触发很多没必要的 Safepoint，  
建议加上此选项关闭偏向锁  
-XX:-UseBiasedLocking  
  
# 避免 jvm 定时进入 safepoint，就如 safepoint 中的 no vm operation 操作就是 jvm 定  
时触发的 safepoint  
-XX:+UnlockDiagnosticVMOptions -XX:GuaranteedSafepointInterval=0
```

偏向锁撤销导致 stw

1. 现象

某次上线公司内部系统时候，发现系统反应很慢，停顿时间过长。

2.问题排查

遇到这种情况，推测是频繁 GC 导致，首先是查看日志（配置参数-

XX:+PrintGCAplicationStoppedTime），结果如下：

```
Total time for which application threads were stopped: 0.0050077 seconds,
```

```
Stopping threads took: 0.0009924
```

```
Total time for which application threads were stopped: 0.0048003 seconds,
```

```
Stopping threads took: 0.000722
```

```
Total time for which application threads were stopped: 0.0042130 seconds,
```

```
Stopping threads took: 0.0006124
```

```
Total time for which application threads were stopped: 0.0167637 seconds,
```

```
Stopping threads took: 0.0002226
```

```
Total time for which application threads were stopped: 0.0041316 seconds,
```

```
stopping threads took: 0.0003492
```

```
Total time for which application threads were stopped: 0.0116966 seconds,
```

```
Stopping threads took: 0.0004518
```

```
Total time for which application threads were stopped: 0.0031024 seconds,
```

Stopping threads took: 0.0002000

Total time for which application threads were stopped: 0.0031730 seconds,

Stopping threads took: 0. 0001507

Total time for which application threads were stopped: 0.0643528 seconds,

Stopping threads took: 0.0005159

Total time for which application threads were stopped: 0.0051813 seconds,

Stopping threads took: 0.001691

Total time for which application threads were stopped: 0.0041419 seconds,

Stopping threads took: 0. 0005032

Total time for which application threads were stopped: 0. 0036222 seconds,

Stopping threads took: 0. 0005483

可以看出这是 stop the world 过于频繁导致，在测试环境复现此情况，配置下面 JVM 参

数打印出 stop the world 的原因。

-XX:+PrintSafepointStatistics

-XX:+PrintSafepointStatisticsCount=1

```
-XX:+SafepointTimeout  
  
-XX:SafepointTimeoutDelay=200  
  
-XX:+UnlockDiagnosticVMOptions  
  
-XX:-DisplayVMOutput  
  
-XX:+LogVMOutput  
  
-XX:LogFile=/XXXX.log
```

最终在日志中发现是频繁的 RevokeBias (偏向锁撤销) 导致的 STW 时间过长，从而导致系统停顿。

3.问题解决

1) 为什么频繁的偏向锁撤销会导致 STW 时间增加呢？阅读偏向锁源码可以知道：偏向锁的撤销需要等待全局安全点（safe point），暂停持有偏向锁的线程，检查持有偏向锁的线程状态。首先遍历当前 JVM 的所有线程，如果能找到偏向线程，则说明偏向的线程还存活，此时检查线程是否在执行同步代码块中的代码，如果是，则升级为轻量级锁，进行 CAS 竞争锁。可以看出撤销偏向锁的时候会导致 stop the word。

```
static BiasedLocking::Condition revoke_bias(oop obj, bool allow_rebias, bool  
is_bulk, JavaThread* requesting_thread) {
```

```
markOop mark = obj->mark();

// 如果对象不是偏向锁，直接返回 NOT_BIASED

if (!mark->has_bias_pattern()) {

    ...

return BiasedLocking::NOT_BIASED;

}

uint age = mark->age();

markOop biased_prototype = markOopDesc::biased_locking_prototype()-
>set_age(age);//匿名偏向模式 (101)

markOop unbiased_prototype = markOopDesc::prototype()->set_age(age);//无锁
模式 (001)

...
```

```
JavaThread* biased_thread = mark->biased_locker();

if (biased_thread == NULL) {// 匿名偏向。

if (!allow_rebias) {

obj->set_mark(unbiased_prototype);// 如果不允许重偏向，则将对象的 mark word 设
置为无锁模式

}

...
return BiasedLocking::BIAS_REVOKED;
}

/*判断偏向线程是否还存活*/

bool thread_is_alive = false;

if (requesting_thread == biased_thread) {
```

```
// 如果当前线程就是偏向线程

thread_is_alive = true;

}

} else {

// 当前线程不是偏向线程，遍历当前 jvm 的所有线程，如果能找到，则说明偏向的线程还
存活

for (JavaThread* cur_thread = Threads::first(); cur_thread != NULL; cur_thread =
cur_thread->next()) {

if (cur_thread == biased_thread) {

thread_is_alive = true;

break;

}

}

}
```

```
}

// 如果偏向的线程已经不存活了

if (!thread_is_alive) {

    // 如果允许重偏向，则将对象 mark word 设置为匿名偏向状态，否则设置为无锁状态

    if (allow_rebias) {

        obj->set_mark(biased_prototype);

    } else {

        obj->set_mark(unbiased_prototype);

    }

    ...

    return BiasedLocking::BIAS_REVOKED;
}
```

```
}
```

```
/*线程还存活则遍历线程栈中所有的 lock record*/
```

```
GrowableArray<MonitorInfo*>* cached_monitor_info =  
get_or_compute_monitor_info(biased_thread);
```

```
BasicLock* highest_lock = NULL;
```

```
for (int i = 0; i < cached_monitor_info->length(); i++) {
```

```
MonitorInfo* mon_info = cached_monitor_info->at(i);
```

```
// 如果能找到对应的 lock record, 说明偏向所有者正在持有锁
```

```
if (mon_info->owner() == obj) {
```

```
...
```

```
/*升级为轻量级锁, 修改栈中所有关联该锁的 lock record,
```

先处理所有锁重入的情况，轻量级锁的 displaced mark word 为 NULL，表示锁重入*/

```
markOop mark = markOopDesc::encode((BasicLock*) NULL);
```

```
highest_lock = mon_info->lock();
```

```
highest_lock->set_displaced_header(mark);
```

```
} else {
```

```
...
```

```
}
```

```
}
```

/* highest_lock 如果非空，则它是最早关联该锁的 lock record，这个 lock record 是线程彻底退出该锁的最后一个 lock record，所以要设置 lock record 的 displaced mark word 为无锁状态的 mark word，并让锁对象的 mark word 指向当前 lock record*/

```
if (highest_lock != NULL) {
```

```
highest_lock->set_displaced_header(unbiased_prototype);

obj->release_set_mark(markOopDesc::encode(highest_lock));

...

} else {

// 偏向所有者没有在持有锁

...

if (allow_rebias) {

obj->set_mark(biased_prototype);// 设置为匿名偏向状态

}

} else {

obj->set_mark(unbiased_prototype);// 将 mark word 设置为无锁状态

}
```

```
    }

    return BiasedLocking::BIAS_REVOKED;

}
```

2) 定位到问题了，此时可以有两种思路：第一种，减少竞争锁的线程数量，第二种，关闭偏向锁，JVM 参数-XX:-UseBiasedLocking。经过修改代码，系统停顿时间显著减少。

小结一下：我们需要结合自身应用并发情况，来评估偏向锁带来的收益。偏向锁主要影响重启后短时间内的负载尖刺，平滑流量场景影响不大。

JVM 的安全点学习与代码测试

监控安全点（打印 JVM 停顿时间，不止 GC，处理毛刺）：

```
-XX:+PrintGC -XX:+PrintGCAApplicationStoppedTime
```

取消偏向锁：

```
-XX:-UseBiasedLocking
```

1.Code

```
public class SafepointTest {

    static double sum = 0;
```

```
public static void foo(){

    for (int i = 0 ; i < 0x77777777 ; i++) {

        sum += Math.sqrt(i);

    }

}

public static void bar() {

    for (int i = 0 ; i < 50_000_000 ; i++) {

        new Object().hashCode();

    }

}

public static void main(String[] args) {

    new Thread(SafepointTest::foo).start();

    new Thread(SafepointTest::bar).start();

}
```

2. 安全点进入测试 VM 参数设置

安全点进入，日志详情查看：-XX:+PrintSafepointStatistics

返回结果：

vmop	[threads: total initially_running wait_to_block]	[time: spin block sync cleanup vmop] page_trap_count
0.352: ParallelGCFailedAllocation	[13 1 1]	[3969 0 3969 0 2] 0
4.415: ParallelGCFailedAllocation	[12 0 0]	[0 0 0 0 1] 0
4.508: ParallelGCFailedAllocation	[12 0 0]	[0 0 0 0 1] 0
4.601: ParallelGCFailedAllocation	[12 0 0]	[0 0 0 0 1] 0
4.695: ParallelGCFailedAllocation	[12 0 0]	[0 0 0 0 1] 0
4.788: ParallelGCFailedAllocation	[12 0 0]	[0 0 0 0 1] 0
4.988: ParallelGCFailedAllocation	[12 0 0]	[0 0 0 0 1] 0
5.172: ParallelGCFailedAllocation	[12 0 0]	[0 0 0 0 0] 0
5.353: ParallelGCFailedAllocation	[12 0 0]	[0 0 0 0 0] 0
5.389: no vm operation	[10 0 1]	[0 0 0 0 473] 0

结果分析：

JVM-GC 日志解析：

此日志分为二段，第一段是时间戳，VM Operation 的类型，以及线程概况

total：安全点里的总线程数

initially_running：安全点时开始时正在运行状态的线程数

wait_to_block：在 VM Operation 开始前需要等待其暂停的线程数

第二段是到达安全点时的各个阶段以及执行操作所花的时间，其中最重要的是 vmop

spin：等待线程响应 safepoint 号召的时间

block：暂停所有线程所用的时间

sync：等于 spin + block，这是从开始到进入安全点所耗的时间，可用于判断

进入安全点耗时

cleanup：清理所有时间

vmop：真正执行 VM Operation 的时间

3. 生产上安全点日志记录问题

生产上需要将安全点日志打印到独立文件，VM 配置如下：

```
-XX:+UnlockDiagnosticVMOptions  
-XX:-DisplayVMOutput  
-XX:+LogVMOutput  
-XX:LogFile=/dev/nya/vm.log
```

打开 Diagnostic (只是开放了更多的 flag 可选，不会主动激活某个 flag) , 关掉输出 VM 日志到 stdout, 输出到独立文件, /dev/nya 目录 (内存文件系统) 。

记忆集合 卡表

在新生代做 GCROOTs 进行可达性分析的时候会碰到跨代引用的对象，这种如果又对老年代再去扫描效率很低，为此，在新生代可以引入记忆集 (Remember Set) 的数据结构（记录从非收集区到收集区的指针集合），避免把整个老年代加入 GCroots 扫描的范围，事实上并不是只是新生代、老年代之间才有跨代引用的问题，所有涉及部分区域收集的垃圾收集器，典型的 G1、ZGC 和 shenandoah 收集器，都会面临相同的问题。

在垃圾收集过程中，收集器通过记忆集判断某一块非收集区域是否指向收集区域的指针即可，无需了解跨代引用指针的全部细节。

hotspot 使用叫做卡表 cartable 的方式实现记忆集：卡表使用一个字节数组实现 CARD_TABLE[], 每个元素对应这个其标识内存区域一块特定大小的内存块，称为卡页。一个卡页中可以包含多个对象，只要一个对象存在跨代指针，其作用的元素表示就变成 1，表示该元素变脏，否则为 0

GC 时候，只要筛选本收集区的卡表变脏的元素加入 GCRoots 里边。

卡表的维护 卡表变脏上面已经说了，但是需要知道如何让卡表变脏，即发生引用字段赋值时，如何更新卡表对应的标识为 1。 Hotspot 使用写屏障维护卡表状态。

JVM 的常见垃圾回收算法？

- 1) 标记-清楚算法：前后线标记处所有需要回收的对象，在标记完成后统一回收有被标记的对象。
- 2) 标记-复制算法：将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当一块内存用完了，将其存在另外一块上面，然后再把已使用过的内存空间一次清理掉。
- 3) 标记-整理算法：标记过程与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所一端移动，然后直接清理掉端边界以外的内存。
- 4) 分代收集算法：一般是把 Java 堆分为新生代和老年代，根据各个年代的特点采用最适当的收集算法。新生代都发现有大批对象死去，选用复制算法。老年代中因为对象存活率高，必须使用“标记-清理”或“标记-整理”算法来进行回收。

JVM 调优的常见命令行工具有哪些？JVM 常见的调优参数有哪些？

(1) JVM 调优的常见命令工具包括：

- 1) jps 命令用于查询正在运行的 JVM 进程，
- 2) jstat 可以实时显示本地或远程 JVM 进程中类装载、内存、垃圾收集、JIT 编译等数据
- 3) jinfo 用于查询当前运行这的 JVM 属性和参数的值。
- 4) jmap 用于显示当前 Java 堆和永久代的详细信息
- 5) jhat 用于分析使用 jmap 生成的 dump 文件，是 JDK 自带的工具
- 6) jstack 用于生成当前 JVM 的所有线程快照，线程快照是虚拟机每一条线

程正在执行的方法,目的是定位线程出现长时间停顿的原因。

(2) JVM 常见的调优参数包括:

-Xmx: 指定 java 程序的最大堆内存, 使用 `java -Xmx5000M -version` 判断

当前系统能分配的最大堆内存

-Xms: 指定最小堆内存, 通常设置成跟最大堆内存一样, 减少 GC

-Xmn: 设置年轻代大小。整个堆大小=年轻代大小 + 年老代大小。所以增大年轻代后, 将会减小年老代大小。此值对系统性能影响较大, Sun 官方推荐配置为整个堆的 3/8。

-Xss: 指定线程的最大栈空间, 此参数决定了 java 函数调用的深度, 值越大调用深度越深, 若值太小则容易出栈溢出错误(StackOverflowError)

-XX:PermSize 指定方法区(永久区)的初始值, 默认是物理内存的 1/64, 在 Java8 永久区移除, 代之的是元数据区, 由-XX:MetaspaceSize 指定

-XX:MaxPermSize 指定方法区的最大值, 默认是物理内存的 1/4,

-XX:MaxMetaspaceSize(在 java8 中由) 指定元数据区的大小

-XX:NewRatio=n 年老代与年轻代的比值, -XX:NewRatio=2, 表示年老代与年轻代的比值为 2:1

-XX:SurvivorRatio=n Eden 区与 Survivor 区的大小比值, -

XX:SurvivorRatio=8 表示 Eden 区与 Survivor 区的大小比值是 8:1:1, 因为 Survivor 区有两个(from, to)

如果你发现每次 GC 后, Heap 的剩余空间会是总空间的 50%, 这表示你的 Heap 处于健康状态。许多 Server 端的 Java 程序每次 GC 后最好能有 65%的剩余空间。

Server 端 JVM 最好将-Xms 和-Xmx 设为相同值。为了优化 GC, 最好让-Xmn 值约

等于-Xmx 的 1/3 或 3/8。

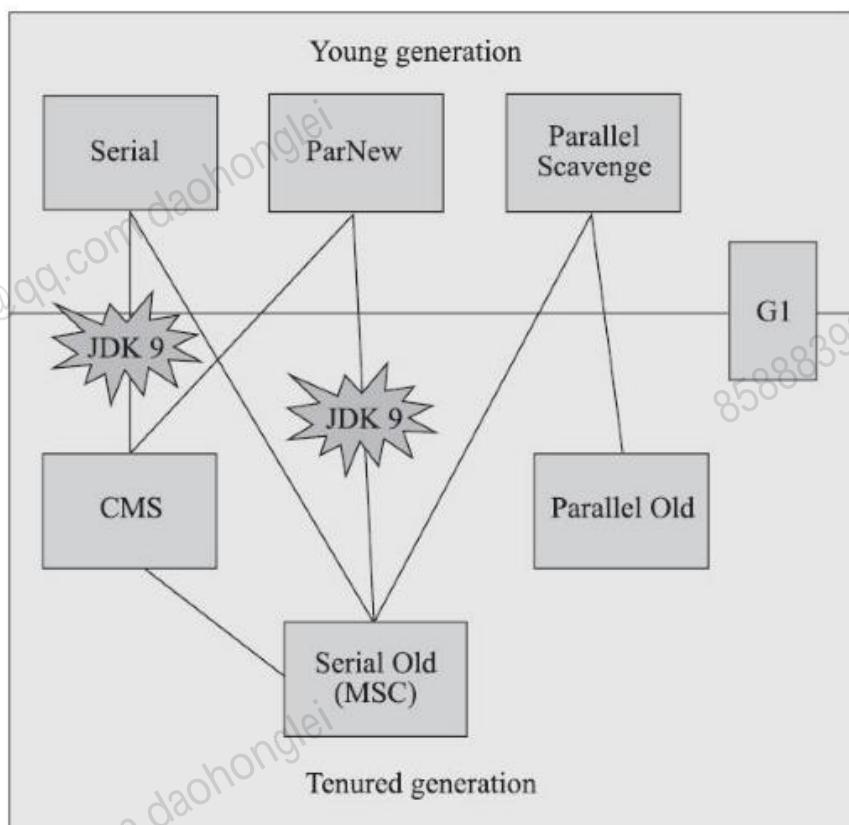
一个 GUI 程序最好是每 10 到 20 秒间运行一次 GC，每次在半秒之内完成。

在 GC 中，如果对象在 Survivor 区复制一次，年龄增加 1。当对象达到设定的阈值时，将会晋升到老年带。默认情况下，并行 GC 的年龄阈值为 15，并发 GC 的年龄阈值为 6。

由于 age 只有 4 位，所以最大值为 15，这就是-XX:MaxTenuringThreshold 选项最大值为 15 的原因。-XX:MaxTenuringThreshold，这个值在 CMS 下默认为 6，G1 下默认为 15

-XX:PretenureSizeThreshold=2m 缺点：只对 Serial 和 ParNew 两个新生代收集器有用

java 垃圾收集器



Serial

最基础的收集器，使用复制[算法](#)、单线程工作，只用一个处理器或一条线程完成垃圾收集，进行垃圾收集时必须暂停其他所有工作线程。

Serial 是虚拟机在客户端模式的默认新生代收集器，简单高效，对于内存受限的环境它是所有收集器中额外内存消耗最小的，对于处理器核心较少的环境，Serial 由于没有线程交互开销，可获得最高的单线程收集效率。

Parallel Scavenge

新生代收集器，基于复制[算法](#)，是可并行的多线程收集器，与 ParNew 类似。特点是它的关注点与其他收集器不同，Parallel Scavenge 的目标是达到一个可控制的吞吐量，吞吐量就是处理器用于运行用户代码的时间与处理器消耗总时间的比值。

ParNew

Serial 的多线程版本，除了使用多线程进行垃圾收集外其余行为完全一致。ParNew 是虚拟机在服务端模式的默认新生代收集器，一个重要原因是除了 Serial 外只有它能与 CMS 配合。自从 JDK 9 开始，ParNew 加 CMS 不再是官方推荐的解决方案，官方希望它被 G1 取代。

Serial Old

Serial 的老年代版本，单线程工作，使用标记-整理[算法](#)。Serial Old 是虚拟机在客户端模式的默认老年代收集器，用于服务端有两种用途：① JDK5 及之前与 Parallel Scavenge 搭配。② 作为 CMS 失败预案。

Parallel Old

Parallel Scavenge 的老年代版本，支持多线程，基于标记-整理[算法](#)。JDK6 提供，注重吞吐量可考虑 Parallel Scavenge 加 Parallel Old。

CMS

以获取最短回收停顿时间为为目标，基于标记-清除算法，过程相对复杂，分为四个步骤：

初始标记、并发标记、重新标记、并发清除。

初始标记和重新标记需要 STW (Stop The World, 系统停顿)，初始标记仅是标记 GC Roots 能直接关联的对象，速度很快。并发标记从 GC Roots 的直接关联对象开始遍历整个对象图，耗时较长但不需要停顿用户线程。重新标记则是为了修正并发标记期间因用户程序运作而导致标记产生变动的那部分记录。并发清除清理标记阶段判断的已死亡对象，不需要移动存活对象，该阶段也可与用户线程并发。

缺点：① 对处理器资源敏感，并发阶段虽然不会导致用户线程暂停，但会降低吞吐量。
② 无法处理浮动垃圾，有可能出现并发失败而导致 Full GC。
③ 基于标记-清除算法，产生空间碎片。

G1

开创了收集器面向局部收集的设计思路和基于 Region 的内存布局，主要面向服务端，最初设计目标是替换 CMS。

G1 之前的收集器，垃圾收集目标要么是整个新生代，要么是整个老年代或整个堆。而 G1 可面向堆任何部分来组成回收集进行回收，衡量标准不再是分代，而是哪块内存中存放的垃圾数量最多，回收受益最大。

跟踪各 Region 里垃圾的价值，价值即回收所获空间大小以及回收所需时间经验值，在后台维护一个优先级列表，每次根据用户设定允许的收集停顿时间优先处理回收价值最大的 Region。这种方式保证了 G1 在有限时间内获取尽可能高的收集效率。

G1 运作过程：

- **初始标记：**标记 GC Roots 能直接关联到的对象，让下一阶段用户线程并发运行时能

正确地在可用 Region 中分配新对象。需要 STW 但耗时很短，在 Minor GC 时同步

完成。

- **并发标记：**从 GC Roots 开始对堆中对象进行可达性分析，递归扫描整个堆的对象图。

耗时长但可与用户线程并发，扫描完成后要重新处理 SATB 记录的在并发时有变动的

对象。

- **最终标记：**对用户线程做短暂暂停，处理并发阶段结束后仍遗留下来的少量 SATB 记录。

- **筛选回收：**对各 Region 的回收价值[排序](), 根据用户期望停顿时间制定回收计划。

必须暂停用户线程，由多条收集线程并行完成。

可由用户指定期望停顿时间是 G1 的一个强大功能，但该值不能设得太低，一般设置为

100~300 ms。

```
java -XX:+PrintCommandLineFlags
```

```
java -XX:+PrintFlagsFinal
```

```
java -XX:+PrintFlagsInitial
```

- **常用：**

- -XX:+PrintFlagsFinal
 - 设置值（最终生效值）
- -XX:+PrintFlagsInitial
 - 默认值
- -XX:+PrintCommandLineFlags
 - 命令行参数

收集器	串行、并行or并发	新生代/老年代	算法	目标	适用场景
Serial	串行	新生代	复制算法	响应速度优先	单CPU环境下的Client模式
Serial Old	串行	老年代	标记-整理	响应速度优先	单CPU环境下的Client模式、CMS的后备预案
ParNew	并行	新生代	复制算法	响应速度优先	多CPU环境时在Server模式下与CMS配合
Parallel Scavenge	并行	新生代	复制算法	吞吐量优先	在后台运算而不需要太多交互的任务
Parallel Old	并行	老年代	标记-整理	吞吐量优先	在后台运算而不需要太多交互的任务
CMS	并发	老年代	标记-清除	响应速度优先	集中在互联网站或B/S系统服务端上的Java应用
G1	并发	both	标记-整理+复制算法	响应速度优先	面向服务端应用，将来替换CMS

CSDN @Leo Han

cms 收集器的缺点?

CMS 是一款优秀的收集器， 它最主要的优点在名字上已经体现出来： 并发收集、低停顿，一些官方公开文档里面 也称之为 “并发低停顿收集器” (Concurrent Low Pause Collector) 。 CMS 收集器是 HotSpot 虚拟机追求低停顿的第一次成功尝试，但是它还远达不到完美的程度，至少有以下三个明显的缺点：

首先， CMS 收集器对处理器资源非常敏感。事实上，面向并发设计的程序都对处理器资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但却会因为占用了一部分线程（或者说处理器的计算能力）而导致应用程序变慢，降低总吞吐量。 CMS 默认启动的回收线程数是 $(\text{处理器核心数量} + 3) / 4$ ，也就是说，如果处理器核心数在四个或以上，并发回收时垃圾收集线程只占用不超过 25% 的处理器运算资源，并且会随着处理器核心数量的增加而下降。但是当处理器核心数量不足四个时，CMS 对用户程序的影响就可能变得很大。如果应用本来的处理器负载就很高，还要分出一半的运算能力去执行收集器线程，就可能导致用户程序的执行速度忽然大幅降低。为了缓解这种情况，虚拟机提供了一种称为 “增量式并发收集器” (Incremental Concurrent Mark Sweep/i-CMS) 的 CMS 收集器变种，所做的事情和以前单核处理器年代 PC 机操作系统靠抢占式多任务来模拟多核并行多任务的思想一样，是在并发标记、清理的时候让收集器线程、用户线程交替运行，尽量减少垃圾收集线程的独占资源的时间，这样整个垃圾收集的过程

程会更长， 但对用户程序的影响就会显得较少一些， 直观感受是速度变慢的时间更多了， 但速度下降幅度就没有那么明显。实践证明增量式的 CMS 收集器效果很一般，从 JDK 7 开始，i-CMS 模式已经被声明为 “deprecated” ， 即已过时不再提倡用户使用， 到 JDK 9 发布后 i-CMS 模式被完全废弃。

然后，由于 CMS 收集器无法处理 “浮动垃圾” (Floating Garbage) ， 有可能出现 “Con-current Mode Failure” 失败进而导致另一次完全 “Stop The World” 的 Full GC 的产生。在 CMS 的并发标记和并发清理阶段， 用户线程是还在继续运行的， 程序在运行自然就还会伴随有新的垃圾对象不断产生， 但这一部分垃圾对象是出现在标记过程结束以后， CMS 无法在当次收集中处理掉它们， 只好留待下一次垃圾收集时再清理掉。这一部分垃圾就称为 “浮动垃圾” 。同样也是由于在垃圾收集阶段用户线程还需要持续运行， 那就还需要预留足够内存空间提供给用户线程使用， 因此 CMS 收集器不能像其他收集器那样等待到老年代几乎完全被填满了再进行收集， 必须预留一部分空间供并发收集时的程序运作使用。在 JDK 5 的默认设置下， CMS 收集器当老年代使用了 68% 的空间后就会被激活， 这是一个偏保守的设置， 如果在实际应用中老年代增长并不是太快， 可以适当调高参数-XX: CMSInitiatingOccupancyFraction 的值来提高 CMS 的触发百分比，降低内存回收频率， 获取更好的性能。到了 JDK 6 时， CMS 收集器的启动阈值就已经默认提升至 92%。但这又会更容易面临另一种风险：要是 CMS 运行期间预留的内存无法满足程序分配新对象的需要， 就会出现一次 “并发失败” (Concurrent Mode Failure) ， 这时候虚拟机将不得不启动后备预案：冻结用户线程的执行， 临时启用 Serial Old 收集器来重新进行老年代的垃圾收集， 但这样停顿时间就很长了。所以参数-XX: CMSInitiatingOccupancyFraction 设置得太高将会很容易导致大量的并发失败产生， 性能反而降低， 用户应在生产环境中根据实际应用情况来权衡设置。

还有最后一个缺点，在本节的开头曾提到，CMS 是一款基于“标记-清除”算法实现的收集器，如果对前面对垃圾回收器算法讲解还有印象的话，就可能想到这意味着收集结束时会有大量空间碎片产生。空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很多剩余空间，但就是无法找到足够大的连续空间来分配当前对象，而不得不提前触发一次 Full GC 的情况。为了解决这个问题，CMS 收集器提供了一个-XX: +UseCMS-CompactAtFullCollection 开关参数（默认是开启的，此参数从 JDK 9 开始废弃），用于在 CMS 收集器不得不进行 Full GC 时开启内存碎片的合并整理过程，由于这个内存整理必须移动存活对象，（在 Shenandoah 和 ZGC 出现前）是无法并发的。这样空间碎片问题是解决了，但停顿时间又会变长，因此虚拟机设计者们还提供了另外一个参数-XX: CMSFullGCsBefore-Compaction（此参数从 JDK 9 开始废弃），这个参数的作用是要求 CMS 收集器在执行过若干次（数量由参数值决定）不整理空间的 Full GC 之后，下一次进入 Full GC 前会先进行碎片整理（默认值为 0，表示每次进入 Full GC 时都进行碎片整理）。只能预防，不能根治。

G1 收集器简介？以及它的内存划分怎么样的？

(1) 简介：

Garbage-First (G1，垃圾优先) 收集器是服务类型的收集器，目标是多处理器机器、大内存机器。它高度符合垃圾收集暂停时间的目标，同时实现高吞吐量。

Oracle JDK 7 update 4 以及更新发布版完全支持 G1 垃圾收集器

(2) G1 的内存划分方式：

它是将堆内存被划分为多个大小相等的 heap 区，每个 heap 区都是逻辑上连续的一段内存(virtual memory)。其中一部分区域被当成老一代收集器相同的角色

(eden, survivor, old), 但每个角色的区域个数都不是固定的。这在内存使用上提供了更多的灵活性

G1 参数设置

- -XX:+UserG1GC 手动指定使用 G1 收集器执行内存回收任务。
- -XX: +G1HeapRegionSize 设置每个 Region 的大小，值是 2 的幂，范围是 1MB 到 32MB，默认是堆内存的 1/2000，目标是根据最小的 Java 堆大小划分出约 2048 个区域。
- -XX: MaxGCPauseMillis 设置期望达到的最大 GC 停顿时间指标（JVM 会尽力实现，但不保证达到）。默认值是 200ms
- -XX:ParallelGCThread 设置 STW 时 GC 线程数的值，最多设置为 8
- -XX: ConcGCThread 设置并发标记的线程数，将 n 设置为并行垃圾回收线程数 (ParallelGCThreads) 的 1/4 左右
- -XX: InitiatingHeapOccupancyPercent 设置出发并发 GC 周期 Java 堆占用率的阈值。超过此值，就出发 GC，默认值是 45.

ZGC 了解吗？

JDK11 中加入的具有实验性质的低延迟垃圾收集器，目标是尽可能在不影响吞吐量的前提下，实现在任意堆内存大小都可以把停顿时间限制在 10ms 以内的低延迟。

基于 Region 内存布局，不设分代，使用了读屏障、染色指针和内存多重映射等技术实现可并发的标记-整理，以低延迟为首要目标。

ZGC 的 Region 具有动态性，是动态创建和销毁的，并且容量大小也是动态变化的。

触发主 GC 的条件

JVM 进行次 GC 的频率很高,但因为这种 GC 占用时间极短,所以对系统产生的影响不大。更值得关注的是主 GC 的触发条件,因为它对系统影响很明显。总的来说,有两个条件会触发主 GC:

- 1)当应用程序空闲时,即没有应用线程在运行时,GC 会被调用。因为 GC 在优先级最低的线程中进行,所以当应用忙时,GC 线程就不会被调用,但以下条件除外。
2)Java 堆内存不足时,GC 会被调用。当应用线程在运行,并在运行过程中创建新对象,若这时内存空间不足,JVM 就会强制地调用 GC 线程,以便回收内存用于新的分配。若 GC 一次之后仍不能满足内存分配的要求,JVM 会再进行两次 GC 作进一步的尝试,若仍无法满足要求,则 JVM 将报 “out of memory” 的错误,Java 应用将停止。

由于是否进行主 GC 由 JVM 根据系统环境决定,而系统环境在不断的变化当中,所以主 GC 的运行具有不确定性,无法预计它何时必然出现,但可以确定的是对一个长期运行的应用来说,其主 GC 是反复进行的。

减少 GC 开销的措施

根据上述 GC 的机制,程序的运行会直接影响系统环境的变化,从而影响 GC 的触发。若不针对 GC 的特点进行设计和编码,就会出现内存驻留等一系列负面影响。为了避免这些影响,基本的原则就是尽可能地减少垃圾和减少 GC 过程中的开销。具体措施包括以下几个方面:

(1)不要显式调用 System.gc()

此函数建议 JVM 进行主 GC,虽然只是建议而非一定,但很多情况下它会触发主 GC,从而增加主 GC 的频率,也即增加了间歇性停顿的次数。

(2)尽量减少临时对象的使用

临时对象在跳出函数调用后,会成为垃圾,少用临时变量就相当于减少了垃圾的产生,从而延长了出现上述第二个触发条件出现的时间,减少了主 GC 的机会。

(3)对象不用时最好显式置为 Null

一般而言,为 Null 的对象都会被作为垃圾处理,所以将不用的对象显式地设为 Null,有利于 GC 收集器判定垃圾,从而提高了 GC 的效率。

(4)尽量使用 StringBuffer,而不用 String 来累加字符串

由于 String 是固定长的字符串对象,累加 String 对象时,并非在一个 String 对象中扩增,而是重新创建新的 String 对象,如 Str5=Str1+Str2+Str3+Str4,这条语句执行过程中会产生多个垃圾对象,因为每次作 “+” 操作时都必须创建新的 String 对象,但这些过渡对象对系统来说是没有实际意义的,只会增加更多的垃圾。避免这种情况可以改用 StringBuffer 来累加字符串,因 StringBuffer 是可变长的,它在原有基础上进行扩增,不会产生中间对象。

(5)能用基本类型如 int,long,就不用 Integer,Long 对象

基本类型变量占用的内存资源比相应用对象占用的少得多,如果没有必要,最好使用基本变量。

(6)尽量少用静态对象变量

静态变量属于全局变量,不会被 GC 回收,它们会一直占用内存。

(7)分散对象创建或删除的时间

集中在短时间内大量创建新对象,特别是大对象,会导致突然需要大量内存,JVM 在面临这种情况时,只能进行主 GC,以回收内存或整合内存碎片,从而增加主 GC 的频率。集中删除对象,道理也是一样的。它使得突然出现了大量的垃圾对象,空闲空间必然减少,从而大大增加了下一次创建新对象时强制主 GC 的机会。

调优目标:

- 堆内存使用率 <= 70%;
- 老年代内存使用率 <= 70%;
- avg pause <= 1 秒;
- Full GC 次数 0 或 avg pause interval >= 24 小时 ;
- 创建更多的线程

什么时候 JVM 调优?

- 系统吞吐量与响应性能不高或下降;
- Heap 内存 (老年代) 持续上涨达到设置的最大内存值;
- Full GC 次数频繁;
- GC 停顿时间过长 (超过 1 秒) ;
- 应用出现 OutOfMemory 等内存异常;
- 应用中有使用本地缓存且占用大量内存空间;

JVM 常用配置参数

-Xms10m -Xmx256m -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -

Xloggc:D:\gc.txt -XX:+UseG1GC -XX:+HeapDumpOnOutOfMemoryError -
XX:HeapDumpPath=D:\a.dump

- 栈

- -Xss 设置每个线程可使用的内存大小，即栈的大小。在相同物理内存下，减小这个值能生成更多的线程，当然操作系统对一个进程内的线程数还是有限制的，不能无限生成。线程栈的大小是个双刃剑，如果设置过小，可能会出现栈溢出，特别是在该线程内有递归、大的循环时出现溢出的可能性更大，如果该值设置过大，就有影响到创建栈的数量，如果是多线程的应用，就会出现内存溢出的错误。

- 堆设置

- -Xms 堆内存的初始大小，默认为物理内存的 1/64
- -Xmx 堆内存的最大大小，默认为物理内存的 1/4
- -Xmn 堆内新生代的大小。通过这个值也可以得到老生代的大小：-Xmx 减去-Xmn
- -XX:NewRatio:设置新生代和老年代的比值。如：为 3，表示年轻代与老年代比值为 1: 3
- -XX:SurvivorRatio:新生代中 Eden 区与两个 Survivor 区的比值。注意 Survivor 区有两个。如：为 3，表示 Eden: Survivor=3: 2，一个 Survivor 区占整个新生代的 1/5
- -XX:MaxTenuringThreshold:设置转入老年代的存活次数。如果是 0，则直接越过新生代进入老年代
- -XX:PermSize、-XX:MaxPermSize:分别设置永久代最小大小与最大大小
(Java8 以前)
- -XX:MetaspaceSize、-XX:MaxMetaspaceSize:分别设置元空间最小大小与最大大小 (Java8 以后)
- -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=: 内存异常自动转到指定路径

- 收集器设置

- -XX:+UseSerialGC:设置串行收集器
- -XX:+UseParallelGC:设置并行收集器
- -XX:+UseParalledOldGC:设置并行老年代收集器
- -XX:+UseConcMarkSweepGC:设置并发收集器
- -XX:+UseG1GC 使用 G1 收集器

- 垃圾回收统计信息 jinfo

- -XX:+PrintGC
- -XX:+PrintGCDetails
- -XX:+PrintGCTimeStamps
- -Xloggc:filename

- 并行收集器设置

- -XX:ParallelGCThreads=n:设置并行收集器收集时使用的 CPU 数。并行收集线程数。
- -XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间
- -XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为
$$1/(1+n)$$

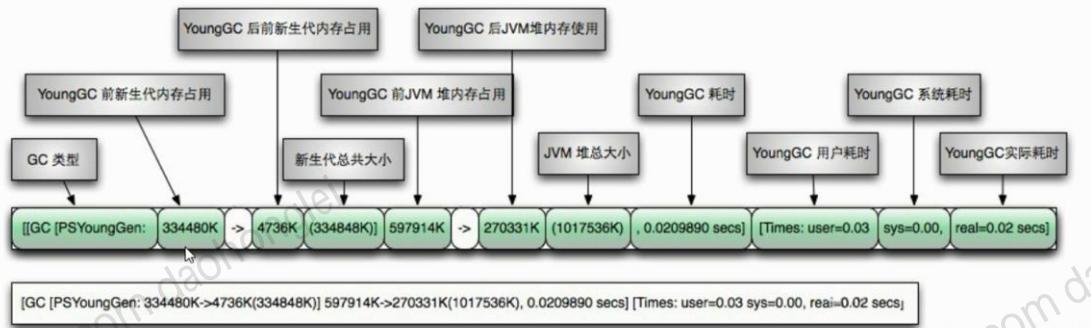
- 并发收集器设置

- -XX:+CMSIncrementalMode:设置为增量模式。适用于单 CPU 情况。
- -XX:ParallelGCThreads=n:设置并发收集器新生代收集方式为并行收集时，使用的 CPU 数。并行收集线程数

-XX:+PrintGCDetails 参数使用

1. YoungGC 日志的详解

下图为 GC 日志的详解. 解释了每一个参数的含义.



以上面控制台打印的的 GC 打印日志举例解读.

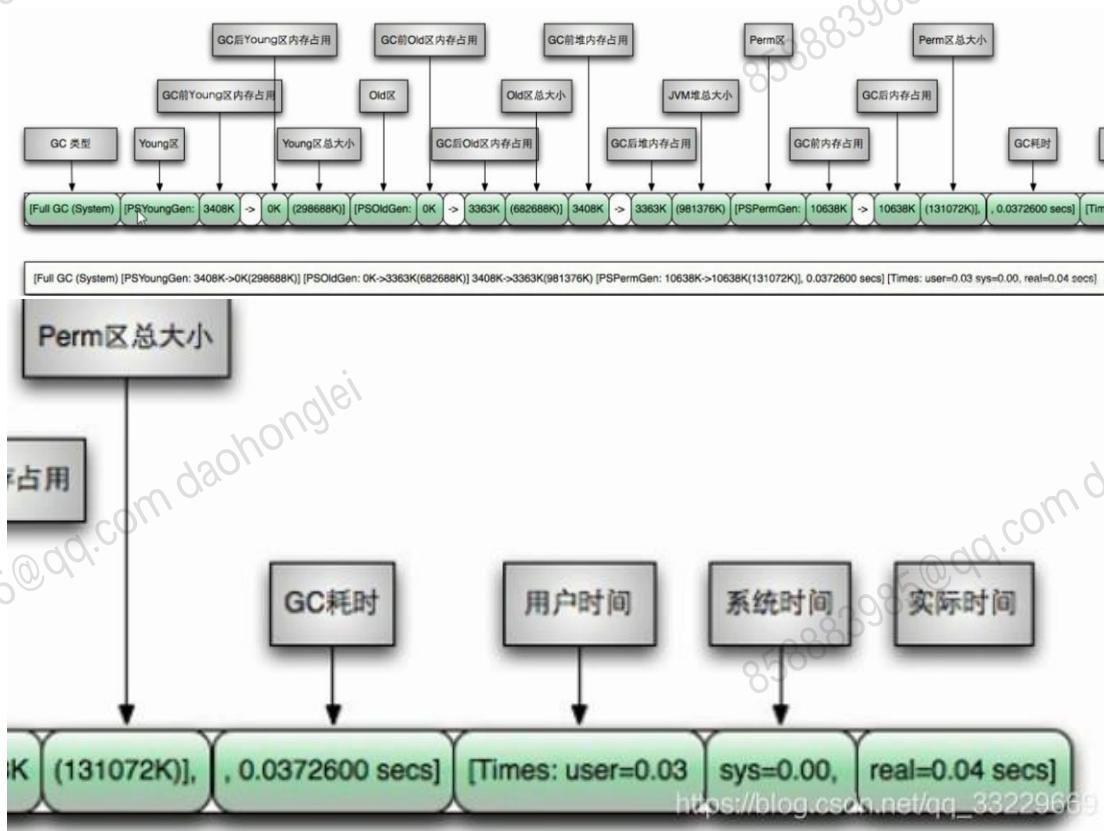
[PSYoungGen: 1912K->496K(2560K) 1912K->736K(9728K), 0.0017018 secs]

[Times: user=0.00 sys=0.00, real=0.00 secs]

- PSYoungGen 代表的是新生区的 GC
- 1912K 代表的 YoungGC 前, 新生代占用 1912k 的内存
- 496K 代表 YoungGC 后, 新生代占用 496K 的内存
- 2560K 代表 新生代总大小为 2560K
- 1912K 代表 YoungGC 前, JVM 堆内存的占用大小
- 736K 代表 YoungGC 后, JVM 堆内存的占用大小
- 9728K 代表 JVM 堆内存的总的大小. (大约为 10m, 即项目启动时, 设置的 10m 的堆内存大小.)
- 0.0017018 secs 代表 YoungGC 的耗时时间, 单位为秒
- Times: user=0.00 sys=0.00, real=0.00 secs 分别为三个时间(了解) 分别为 YoungGC 用户耗时时间, 系统耗时时间, 实际耗时时间. 单位为秒 .

2. Full GC 日志的详解

下2张图 为 Full GC 的详细解读



根据上面控制台打印的 GC 日志详细解读每一个参数的含义:

[Full GC (Allocation Failure) [PSYoungGen: 480K->0K(2560K)] [ParOldGen: 248K->659K(7168K)] 728K->659K(9728K), [Metaspace: 3031K->3031K(1056768K)], 0.0071285 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]

- Full GC 代表 GC 类型为 Full GC 括号中的 Allocation Failure ,代表分配空间失败.
- PSYoungGen 代表新生代区
- 480K 代表 GC 前 Young 区 ,内存占用的大小
- 0K 代表 GC 后 Young 区 ,内存占用的大小. 把 Young 区的内存全部回收了
- 2560K 代表 Young 区 的总的大小.
- ParOldGen 代表老年代区
- 248K 代表 GC 前 Old 区内存占用大小

- 659K 代表 GC 后 Old 区内存占用大小
- 7168K 代表 Old 区总的大小
- 728K 代表 GC 前堆内存占用大小
- 659K 代表 GC 后堆内存占用大小
- 9728K 代表堆的总大小
- Metaspace 代表元空间. 图片中写的是 PSPermGen 永久代为 jdk7 的版本, jdk8 为元空间
- 3031K 代表 GC 前 元空间占用的大小
- 3031K 代表 GC 后 元空间占用的大小 (Full GC 前后, 元空间的大小不变.)
- 1056768K 代表 元空间的总大小.
- 0.0071285 secs 代表 Full GC 的消耗时间. 单位为秒
- Times: user=0.03 sys=0.00, real=0.01 secs 分别为三个时间 分别为 Full GC 用户耗时时间, 系统耗时时间, 实际耗时时间. 单位为秒 . 用于分析日志的性能

通过上面的分析可以看到, 在老年代 GC 前为 248K , GC 后 Old 区内存占用大小为 659K .

代表 代码中 new byte[50 * 1024 * 1024];太大了, 无法被回收, 老年代都扛不住了, 所以报了 Java heap space 堆空间溢出的错误.

3. 参数解读规律

无论是 Full GC 还是 Young GC ,解读日志都有如下的规律.

首先是该区域的名称, 接着是 GC 前的大小. -> 是 GC 后的大小. 接着是总的大小. 如果没有标明名称, 代表为堆的大小. 即堆的 GC 前 -> GC 后大小, 总大小.

规律：



你知道哪些故障处理工具？

-Xms5M -Xmx5M -Xmn3M -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:SurvivorRatio=8

jps: 虚拟机进程状况工具

功能和 ps 命令类似：可以列出正在运行的虚拟机进程，显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一 ID (LVMID)。LVMID 与操作系统的进程 ID (PID) 一致，使用 Windows 的任务管理器或 UNIX 的 ps 命令也可以查询到虚拟机进程的 LVMID，但如果同时启动了多个虚拟机进程，必须依赖 jps 命令。

- -q 不输出类名、Jar 名和传入 main 方法的参数
- -m 输出传入 main 方法的参数
- -l 输出 main 类或 Jar 的全限名
- -v 输出传入 JVM 的参数
- Joption：传递参数到 vm,例如:-J-Xms512m

jstat: 虚拟机统计信息监视工具

<https://blog.csdn.net/javalingyu/article/details/124800644>

<https://docs.oracle.com/javase/1.5.0/docs/tooldocs/share/jstat.html>

用于监视虚拟机各种运行状态信息。可以显示本地或远程虚拟机进程中的类加载、内存、垃圾收集、即时编译器等运行时数据，在没有 GUI 界面的服务器上是运行期定位虚拟

机性能问题的常用工具。

参数含义：S0 和 S1 表示两个 Survivor，E 表示新生代，O 表示老年代，YGC 表示 Young GC 次数，YGCT 表示 Young GC 耗时，FGC 表示 Full GC 次数，FGCT 表示 Full GC 耗时，GCT 表示 GC 总耗时。

功能描述：

Jstat 是 JDK 自带的一个轻量级小工具。它位于 java 的 bin 目录下，主要利用 JVM 内建的指令对 Java 应用程序的资源和性能进行实时的命令行的监控，包括了对 Heap size 和垃圾回收状况的监控。

命令用法：jstat [-命令选项] [vmid] [间隔时间/毫秒] [查询次数]

注意：使用的 jdk 版本是 jdk8。

```
C:\Users\Administrator>jstat -help

Usage: jstat -help|-options

jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]

Definitions:

<option>    An option reported by the -options option

<vmid>      Virtual Machine Identifier. A vmid takes the following form:

<lvmid>[@<hostname>[:<port>]]

Where <lvmid> is the local vm identifier for the target

Java virtual machine, typically a process id; <hostname> is

the name of the host running the target Java virtual machine;

and <port> is the port number for the rmiregistry on the

target host. See the jvmstat documentation for a more complete
```

description of the Virtual Machine Identifier.

<lines> Number of samples between header lines.

<interval> Sampling interval. The following forms are allowed:

<n>["ms"|"s"]

Where <n> is an integer and the suffix specifies the units as

milliseconds("ms") or seconds("s"). The default units are "ms".

<count> Number of samples to take before terminating.

-J<flag> Pass <flag> directly to the runtime system.

- **option: 参数选项**
- **-t: 可以在打印的列加上 Timestamp 列，用于显示系统运行的时间**
- **-h: 可以在周期性数据输出的时候，指定输出多少行以后输出一次表头**
- **vmid: Virtual Machine ID (进程的 pid)**
- **interval: 执行每次的间隔时间，单位为毫秒**
- **count: 用于指定输出多少次记录，缺省则会一直打印**

option 可以从下面参数中选择

jstat -options

- **-class 用于查看类加载情况的统计**
- **-compiler 用于查看 HotSpot 中即时编译器编译情况的统计**
- **-gc 用于查看 JVM 中堆的垃圾收集情况的统计**
- **-gccapacity 用于查看新生代、老生代及持久代的存储容量情况**
- **-gcmetacapacity 显示 metaspace 的大小**
- **-gcnew 用于查看新生代垃圾收集的情况**

- **-gcnewcapacity** 用于查看新生代存储容量的情况
- **-gcold** 用于查看老生代及持久代垃圾收集的情况
- **-gcoldcapacity** 用于查看老生代的容量
- **-gcutil** 显示垃圾收集信息
- **-gccause** 显示垃圾回收的相关信息（通过-gcutil），同时显示最后一次仅当前正在发生
的垃圾收集的原因
- **-printcompilation** 输出 JIT 编译的方法信息

示例：

1.-class 类加载统计

```
[root@hadoop ~]# jps #先通过 jps 获取到 java 进程号 (这里是一个 zookeeper 进程)
3346 QuorumPeerMain

7063 Jps

[root@hadoop ~]# jstat -class 3346 #统计 JVM 中加载的类的数量与 size
Loaded  Bytes  Unloaded  Bytes   Time
1527  2842.7    0    0.0    1.02
```

- **Loaded:** 加载类的数量
- **Bytes:** 加载类的 size，单位为 Byte
- **Unloaded:** 卸载类的数目
- **Bytes:** 卸载类的 size，单位为 Byte
- **Time:** 加载与卸载类花费的时间

2.-compiler 编译统计

```
[root@hadoop ~]# jstat -compiler 3346 #用于查看 HotSpot 中即时编译器编译情况的
```

统计

Compiled Failed Invalid Time FailedType FailedMethod

404 0 0 0.19 0

- **Compiled:** 编译任务执行数量
- **Failed:** 编译任务执行失败数量
- **Invalid:** 编译任务执行失效数量
- **Time:** 编译任务消耗时间
- **FailedType:** 最后一个编译失败任务的类型
- **FailedMethod:** 最后一个编译失败任务所在的类及方法

3.-gc 垃圾回收统计

[root@hadoop ~]# jstat -gc 3346 #用于查看 JVM 中堆的垃圾收集情况的统计

SOC	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	CCSC
CCSU	YGC	YGCT	FGC	FGCT	GCT					
128.0	128.0	0.0	128.0	1024.0	919.8	15104.0	2042.4	8448.0	8130.4	
1024.0	996.0	7	0.019	0	0.000	0.019				

- **SOC:** 年轻代中第一个 survivor (幸存区) 的容量 (字节)
- **S1C:** 年轻代中第二个 survivor (幸存区) 的容量 (字节)
- **S0U:** 年轻代中第一个 survivor (幸存区) 目前已使用空间 (字节)
- **S1U:** 年轻代中第二个 survivor (幸存区) 目前已使用空间 (字节)
- **EC:** 年轻代中 Eden (伊甸园) 的容量 (字节)
- **EU:** 年轻代中 Eden (伊甸园) 目前已使用空间 (字节)
- **OC:** Old 代的容量 (字节)

- **OU:** Old 代目前已使用空间 (字节)
- **MC:** metaspace(元空间)的容量 (字节)
- **MU:** metaspace(元空间)目前已使用空间 (字节)
- **CCSC:** 当前压缩类空间的容量 (字节)
- **CCSU:** 当前压缩类空间目前已使用空间 (字节)
- **YGC:** 从应用程序启动到采样时年轻代中 gc 次数
- **YGCT:** 从应用程序启动到采样时年轻代中 gc 所用时间(s)
- **FGC:** 从应用程序启动到采样时 old 代(全 gc)gc 次数
- **FGCT:** 从应用程序启动到采样时 old 代(全 gc)gc 所用时间(s)
- **GCT:** 从应用程序启动到采样时 gc 用的总时间(s)

4.-gccapacity 堆内存统计

```
jstat -gccapacity -h5 3346 1000      # -h5: 每 5 行显示一次表头      1000: 每 1 秒钟
```

显示一次，单位为毫秒

```
[root@hadoop ~]# jstat -gccapacity 3346 #用于查看新生代、老生代及持久代的存储容量情况

NGCMN  NGCMX  NGC  S0C  S1C    EC    OGCMN  OGC MXN  OGC
OC     MCMN   MCMX   MC   CCSMN  CCSMX  CCSC   YGC   FGC
1280.0 83264.0 1280.0 128.0 128.0 1024.0 15104.0 166592.0 15104.0
15104.0 0.0 1056768.0 8448.0 0.0 1048576.0 1024.0 7 0
```

```
[root@hadoop ~]# jstat -gccapacity -h5 3346 1000 #-h5: 每 5 行显示一次表头
1000: 每 1 秒钟显示一次，单位为毫秒
```

NGCMN	NGCMX	NGC	S0C	S1C	EC	OGCMN	OGCMX	OGC
OC	MCMN	MCMX	MC	CCSMN	CCSMX	CCSC	YGC	FGC
1280.0	83264.0	1280.0	128.0	128.0	1024.0	15104.0	166592.0	15104.0
15104.0	0.0	1056768.0	8448.0	0.0	1048576.0	1024.0	7	0

OC	MCMN	MCMX	MC	CCSMN	CCSMX	CCSC	YGC	FGC
1280.0	83264.0	1280.0	128.0	128.0	1024.0	15104.0	166592.0	15104.0
15104.0	0.0	1056768.0	8448.0	0.0	1048576.0	1024.0	7	0
1280.0	83264.0	1280.0	128.0	128.0	1024.0	15104.0	166592.0	15104.0
15104.0	0.0	1056768.0	8448.0	0.0	1048576.0	1024.0	7	0
1280.0	83264.0	1280.0	128.0	128.0	1024.0	15104.0	166592.0	15104.0
15104.0	0.0	1056768.0	8448.0	0.0	1048576.0	1024.0	7	0
1280.0	83264.0	1280.0	128.0	128.0	1024.0	15104.0	166592.0	15104.0
15104.0	0.0	1056768.0	8448.0	0.0	1048576.0	1024.0	7	0
1280.0	83264.0	1280.0	128.0	128.0	1024.0	15104.0	166592.0	15104.0
NGCMN	NGCMX	NGC	S0C	S1C	EC	OGCMN	OGCMX	OGC
OC	MCMN	MCMX	MC	CCSMN	CCSMX	CCSC	YGC	FGC
1280.0	83264.0	1280.0	128.0	128.0	1024.0	15104.0	166592.0	15104.0
15104.0	0.0	1056768.0	8448.0	0.0	1048576.0	1024.0	7	0
1280.0	83264.0	1280.0	128.0	128.0	1024.0	15104.0	166592.0	15104.0
15104.0	0.0	1056768.0	8448.0	0.0	1048576.0	1024.0	7	0
1280.0	83264.0	1280.0	128.0	128.0	1024.0	15104.0	166592.0	15104.0
15104.0	0.0	1056768.0	8448.0	0.0	1048576.0	1024.0	7	0
1280.0	83264.0	1280.0	128.0	128.0	1024.0	15104.0	166592.0	15104.0
15104.0	0.0	1056768.0	8448.0	0.0	1048576.0	1024.0	7	0

- **NGCMN:** 年轻代(young)中初始化(最小)的大小(字节)

- NGCMX: 年轻代(young)的最大容量 (字节)
- NGC: 年轻代(young)中当前的容量 (字节)
- S0C: 年轻代中第一个 survivor (幸存区) 的容量 (字节)
- S1C: 年轻代中第二个 survivor (幸存区) 的容量 (字节)
- EC: 年轻代中 Eden (伊甸园) 的容量 (字节)
- OGCMN: old 代中初始化(最小)的大小 (字节)
- OGCMX: old 代的最大容量(字节)
- OGC: old 代当前新生成的容量 (字节)
- OC: Old 代的容量 (字节)
- MCMN: metaspace(元空间)中初始化(最小)的大小 (字节)
- MCMX: metaspace(元空间)的最大容量 (字节)
- MC: metaspace(元空间)当前新生成的容量 (字节)
- CCSMN: 最小压缩类空间大小
- CCSMX: 最大压缩类空间大小
- CCSC: 当前压缩类空间大小
- YGC: 从应用程序启动到采样时年轻代中 gc 次数
- FGC: 从应用程序启动到采样时 old 代(全 gc)gc 次数

5.-gcmetacapacity 元数据空间统计

```
[root@hadoop ~]# jstat -gcmetacapacity 3346 #显示元数据空间的大小  
  
MCMN MCMX MC CCSMN CCSMX CCSC YGC FGC FGCT GCT  
  
0.0 1056768.0 8448.0 0.0 1048576.0 1024.0 8 0 0.000 0.020
```

- MCMN: 最小元数据容量

- **MCMX**: 最大元数据容量
- **MC**: 当前元数据空间大小
- **CCSMN**: 最小压缩类空间大小
- **CCSMX**: 最大压缩类空间大小
- **CCSC**: 当前压缩类空间大小
- **YGC**: 从应用程序启动到采样时年轻代中 gc 次数
- **FGC**: 从应用程序启动到采样时 old 代(全 gc)gc 次数
- **FGCT**: 从应用程序启动到采样时 old 代(全 gc)gc 所用时间(s)
- **GCT**: 从应用程序启动到采样时 gc 用的总时间(s)

6.-gcnew 新生代垃圾回收统计

```
[root@hadoop ~]# jstat -gcnew 3346 #用于查看新生代垃圾收集的情况
```

```
S0C S1C S0U S1U TT MTT DSS EC EU YGC YGCT
```

```
128.0 128.0 67.8 0.0 1 15 64.0 1024.0 362.2 8 0.020
```

- **S0C**: 年轻代中第一个 survivor (幸存区) 的容量 (字节)
- **S1C**: 年轻代中第二个 survivor (幸存区) 的容量 (字节)
- **S0U**: 年轻代中第一个 survivor (幸存区) 目前已使用空间 (字节)
- **S1U**: 年轻代中第二个 survivor (幸存区) 目前已使用空间 (字节)
- **TT**: 持有次数限制
- **MTT**: 最大持有次数限制
- **DSS**: 期望的幸存区大小
- **EC**: 年轻代中 Eden (伊甸园) 的容量 (字节)
- **EU**: 年轻代中 Eden (伊甸园) 目前已使用空间 (字节)

- YGC: 从应用程序启动到采样时年轻代中 gc 次数
- YGCT: 从应用程序启动到采样时年轻代中 gc 所用时间(s)

7.-gcnewcapacity 新生代内存统计

```
[root@hadoop ~]# jstat -gcnewcapacity 3346 #用于查看新生代存储容量的情况
```

```
NGCMN NGCMX NGC S0CMX S0C S1CMX S1C ECMX EC YGC FGC
```

```
1280.0 83264.0 1280.0 8320.0 128.0 8320.0 128.0 66624.0 1024.0 8 0
```

- NGCMN: 年轻代(young)中初始化(最小)的大小(字节)
- NGCMX: 年轻代(young)的最大容量 (字节)
- NGC: 年轻代(young)中当前的容量 (字节)
- S0CMX: 年轻代中第一个 survivor (幸存区) 的最大容量 (字节)
- S0C: 年轻代中第一个 survivor (幸存区) 的容量 (字节)
- S1CMX: 年轻代中第二个 survivor (幸存区) 的最大容量 (字节)
- S1C: 年轻代中第二个 survivor (幸存区) 的容量 (字节)
- ECMX: 年轻代中 Eden (伊甸园) 的最大容量 (字节)
- EC: 年轻代中 Eden (伊甸园) 的容量 (字节)
- YGC: 从应用程序启动到采样时年轻代中 gc 次数
- FGC: 从应用程序启动到采样时 old 代(全 gc)gc 次数

8.-gcold 老年代垃圾回收统计

```
[root@hadoop ~]# jstat -gcold 3346 #用于查看老年代及持久代垃圾收集的情况
```

```
MC MU CCSC CCSU OC OU YGC FGC FGCT GCT
```

```
8448.0 8227.5 1024.0 1003.7 15104.0 2102.2 8 0 0.000 0.020
```

- MC: metaspace(元空间)的容量 (字节)

- MU: metaspace(元空间)目前已使用空间 (字节)
- CCSC: 压缩类空间大小
- CCSU: 压缩类空间使用大小
- OC: Old 代的容量 (字节)
- OU: Old 代目前已使用空间 (字节)
- YGC: 从应用程序启动到采样时年轻代中 gc 次数
- FGC: 从应用程序启动到采样时 old 代(全 gc)gc 次数
- FGCT: 从应用程序启动到采样时 old 代(全 gc)gc 所用时间(s)
- GCT: 从应用程序启动到采样时 gc 用的总时间(s)

9.-gcoldcapacity 老年代内存统计

```
[root@hadoop ~]# jstat -gcoldcapacity 3346 #用于查看老年代的容量  
OGCMN OGCMX OGC OC YGC FGC FGCT GCT  
15104.0 166592.0 15104.0 15104.0 8 0 0.000 0.020
```

- OGCMN: old 代中初始化(最小)的大小 (字节)
- OGCMX: old 代的最大容量(字节)
- OGC: old 代当前新生成的容量 (字节)
- OC: Old 代的容量 (字节)
- YGC: 从应用程序启动到采样时年轻代中 gc 次数
- FGC: 从应用程序启动到采样时 old 代(全 gc)gc 次数
- FGCT: 从应用程序启动到采样时 old 代(全 gc)gc 所用时间(s)
- GCT: 从应用程序启动到采样时 gc 用的总时间(s)

10.-gcutil 垃圾回收统计

```
[root@hadoop ~]# jstat -gcutil 3346 #显示垃圾收集信息
```

```
S0 S1 E O M CCS YGC YGCT FGC FGCT GCT
```

```
52.97 0.00 42.10 13.92 97.39 98.02 8 0.020 0 0.000 0.020
```

- **S0**: 年轻代中第一个 survivor (幸存区) 已使用的占当前容量百分比
- **S1**: 年轻代中第二个 survivor (幸存区) 已使用的占当前容量百分比
- **E**: 年轻代中 Eden (伊甸园) 已使用的占当前容量百分比
- **O**: old 代已使用的占当前容量百分比
- **M**: 元数据区已使用的占当前容量百分比
- **CCS**: 压缩类空间已使用的占当前容量百分比
- **YGC** : 从应用程序启动到采样时年轻代中 gc 次数
- **YGCT** : 从应用程序启动到采样时年轻代中 gc 所用时间(s)
- **FGC** : 从应用程序启动到采样时 old 代(全 gc)gc 次数
- **FGCT** : 从应用程序启动到采样时 old 代(全 gc)gc 所用时间(s)
- **GCT**: 从应用程序启动到采样时 gc 用的总时间(s)

11.-gccause

```
[root@hadoop ~]# jstat -gccause 3346 #显示垃圾回收的相关信息(通-gcutil) ,同时
```

```
显示最后一次或当前正在发生的垃圾回收的诱因
```

```
S0 S1 E O M CCS YGC YGCT FGC FGCT GCT LGCC GCC
```

```
52.97 0.00 46.09 13.92 97.39 98.02 8 0.020 0 0.000 0.020 Allocation Failure No GC
```

- **LGCC**: 最后一次 GC 原因
- **GCC**: 当前 GC 原因 (No GC 为当前没有执行 GC)

12.-printcompilation JVM 编译方法统计

```
[root@hadoop ~]# jstat -printcompilation 3346 #输出 JIT 编译的方法信息
```

Compiled Size Type Method

```
421 60 1 sun/nio/ch/Util$2 clear
```

- **Compiled:** 编译任务的数目
- **Size:** 方法生成的字节码的大小
- **Type:** 编译类型
- **Method:** 类名和方法名用来标识编译的方法。类名使用/做为一个命名空间分隔符。

方法名是给定类中的方法。上述格式是由-XX:+PrintComplation 选项进行设置的

远程监控

与 jps 一样, jstat 也支持远程监控, 同样也需要开启安全授权, 方法参照 jps。

```
C:\Users\Administrator>jps 192.168.146.128
```

```
3346 QuorumPeerMain
```

```
3475 Jstatd
```

```
C:\Users\Administrator>jstat -gcutil 3346@192.168.146.128
```

S0	S1	E	O	M	CCS	YGC	YGCT	FGC	FGCT	GCT
52.97	0.00	65.15	13.92	97.39	98.02	8	0.020	0	0.000	0.020

jinfo: Java 配置信息工具

https://blog.csdn.net/K_520_W/article/details/121572228

实时查看和调整虚拟机各项参数, 使用 jps 的 -v 参数可以查看虚拟机启动时显式指定的参数, 但如果想知道未显式指定的参数值只能使用 jinfo 的 -flag 查询。

1、简介

jinfo(Configuration Info for Java) 查看虚拟机配置参数信息, 也可用于调整虚拟机的配

置参数。

在很多情况下，Java 应用程序不会指定所有的 Java 虚拟机参数。而此时，开发人员可能不知道某一个具体的 Java 虚拟机参数的默认值。在这种情况下，可能需要通过查找文档获取某个参数的默认值。这个查找过程可能是非常艰难的。但有了 jinfo 工具，开发人员可以很方便地找到 Java 虚拟机参数的当前值。

jinfo 不仅可以查看运行时某一个 Java 虚拟机参数的实际取值，甚至可以在运行时修改部分参数，并使之立即生效。但是，并非所有参数都支持动态修改。参数只有被标记 manageable 的 flag 可以被实时修改。其实，这个修改能力是极其有限的。

2、jinfo -sysprops pid：查看该进程的全部配置信息

描述：输出当前 jvm 进行的全部的系统属性

2.1、启动如下程序

```
package com.kgf.kgfjavalearning2021.jvm;

import java.util.ArrayList;
/**
 * 测试 GC 的频率:
 * 设置内存大小: -Xmx60m -XX:SurvivorRatio=8
 *
 * -Xmx60m: 表示是最大堆的大小, 也就是当你的虚拟机启动后, 就会分配这么大的堆内存给你
 * -XX:SurvivorRatio=8:
```

```
* 设置新生代中 eden 和 S0/S1 空间的比例

* 默认

* -XX:SurvivorRatio=8,Eden:S0:S1=8:1:1

* 假如

* -XX:SurvivorRatio=4,Eden:S0:S1=4:1:1

* SurvivorRatio 值就是设置 Eden 区的比例占多少,S0/S1 相同

*

*/



public class GCTest {

    public static void main(String[] args) {

        ArrayList<byte[]> list = new ArrayList<>();

        for (int i = 0; i < 1000; i++) {

            byte[] arr = new byte[1024*100];//每次向 list 中存放 100kb 的字节数组

            list.add(arr);

            try {

                Thread.sleep(100);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}
```

```
    }
}

}
```

在 idea 中设置堆内存：

2.2、使用 jinfo 命令查看, 查看到的参数其实在我们的 java 中是可以通过
system.getProperties()参数进行获取到的，其实就是环境变量

命令：jinfo -sysprops pid

```
C:\Users\86136>jps
10864 GCTest
11248 RemoteMavenServer
2052
8616 Jps
4972 Launcher

C:\Users\86136>jinfo -sysprops 10864
Attaching to process ID 10864, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.51-b03
java.runtime.name = Java(TM) SE Runtime Environment
java.vm.version = 25.51-b03
sun.boot.library.path = D:\jdk1.8.0_51\jre\bin
java.vendor.url = http://java.oracle.com/
java.vm.vendor = Oracle Corporation
path.separator =
file.encoding.pkg = sun.io
java.vm.name = Java HotSpot(TM) 64-Bit Server VM
sun.os.patch.level =
sun.java.launcher = SUN_STANDARD
user.script =
user.country = CN
user.dir = E:\WorkSpace\kgf-java-learning2021
java.vm.specification.name = Java Virtual Machine Specification
java.runtime.version = 1.8.0_51-b16
java.awt.graphicsenv = sun.awt.Win32GraphicsEnvironment
os.arch = amd64
```

CSDN @爱上口袋的天空

```
C:\Users\86136>jps

10864 GCTest

11248 RemoteMavenServer

2052

8616 Jps
```

4972 Launcher

C:\Users\86136>jinfo -sysprops 10864

Attaching to process ID 10864, please wait...

Debugger attached successfully.

Server compiler detected.

JVM version is 25.51-b03

java.runtime.name = Java(TM) SE Runtime Environment

java.vm.version = 25.51-b03

sun.boot.library.path = D:\jdk1.8.0_51\jre\bin

java.vendor.url = http://java.oracle.com/

java.vm.vendor = Oracle Corporation

path.separator = ;

file.encoding.pkg = sun.io

java.vm.name = Java HotSpot(TM) 64-Bit Server VM

sun.os.patch.level =

sun.java.launcher = SUN_STANDARD

user.script =

user.country = CN

user.dir = E:\WorkSpace\kgf-java-learning2021

java.vm.specification.name = Java Virtual Machine Specification

java.runtime.version = 1.8.0_51-b16

java.awt.graphicsenv = sun.awt.Win32GraphicsEnvironment

```
os.arch = amd64

java.endorsed.dirs = D:\jdk1.8.0_51\jre\lib\endorsed

line.separator =

java.io.tmpdir = C:\Users\86136\AppData\Local\Temp

java.vm.specification.vendor = Oracle Corporation

user.variant =

os.name = Windows 8.1

sun.jnu.encoding = GBK

java.library.path =

D:\jdk1.8.0_51\bin;C:\WINDOWS\Sun\Java\bin;C:\WINDOWS\system32;C:\WINDO
WS;E:\WorkSpace\software\Python37\Scripts;E:\WorkSpace\software\Python37;D:
\oracle12c\product\12.2.0\dbhome_1\bin;C:\Program Files (x86)\Intel\Intel(R)
Management Engine Components\iCLS;C:\Program Files\Intel\Intel(R)
Management Engine

Components\iCLS;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbe
m;C:\Windows\System32\WindowsPowerShell\v1.0;C:\Windows\System32\OpenS
SH;C:\Program Files (x86)\Intel\Intel(R) Management Engine

Components\DAL;C:\Program Files\Intel\Intel(R) Management Engine

Components\DAL;C:\Program Files (x86)\Intel\Intel(R) Management Engine

Components\IPT;C:\Program Files\Intel\Intel(R) Management Engine

Components\IPT;C:\Program Files (x86)\NVIDIA
Corporation\PhysX\Common;D:\jdk1.8.0_51\bin;D:\apache-maven-
```

3.5.2\bin;D:\TortoiseSVN\bin;D:\oracleClient\instantclient_12_2;D:\mysql-5.7.24-win64\bin;D:\zookeeper-3.4.10\bin;d:\Git\cmd;D:\Redis-x64-3.2.100;D:\hadoopWorkSpace\hadoop-2.7.2\bin;C:\Program Files (x86)\Common Files\Thunder Network\KanKan\Codecs;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0;C:\WINDOWS\System32\OpenSSH;D:\scala-2.11.8\bin;D:\gradle-6.7\bin;D:\nodejs;D:\OpenSSL-Win64\bin;D:\software\Vagrant\bin;C:\Ruby27-x64\bin;C:\Users\86136\AppData\Local\Microsoft\WindowsApps;;D:\Microsoft VS Code\bin;C:\Users\86136\AppData\Roaming\npm;

java.specification.name = Java Platform API Specification

java.class.version = 52.0

sun.management.compiler = HotSpot 64-Bit Tiered Compilers

os.version = 6.3

user.home = C:\Users\86136

user.timezone = Asia/Shanghai

java.awt.printerjob = sun.awt.windows.WPrinterJob

file.encoding = UTF-8

java.specification.version = 1.8

user.name = 86136

java.class.path =

D:\jdk1.8.0_51\jre\lib\charsets.jar;D:\jdk1.8.0_51\jre\lib\deploy.jar;D:\jdk1.8.0_51\jr

e\lib\ext\access-bridge-
64.jar;D:\jdk1.8.0_51\jre\lib\ext\cldrdata.jar;D:\jdk1.8.0_51\jre\lib\ext\dnsns.jar;D:\j
dk1.8.0_51\jre\lib\ext\jaccess.jar;D:\jdk1.8.0_51\jre\lib\ext\jfxrt.jar;D:\jdk1.8.0_51\jr
e\lib\ext\localizedata.jar;D:\jdk1.8.0_51\jre\lib\ext\nashorn.jar;D:\jdk1.8.0_51\jre\lib\lib
ext\sunec.jar;D:\jdk1.8.0_51\jre\lib\ext\sunjce_provider.jar;D:\jdk1.8.0_51\jre\lib\ext
\sunmscapi.jar;D:\jdk1.8.0_51\jre\lib\ext\sunpkcs11.jar;D:\jdk1.8.0_51\jre\lib\ext\zi
pfs.jar;D:\jdk1.8.0_51\jre\lib\javaws.jar;D:\jdk1.8.0_51\jre\lib\jce.jar;D:\jdk1.8.0_51\jr
e\lib\jfr.jar;D:\jdk1.8.0_51\jre\lib\jfxswt.jar;D:\jdk1.8.0_51\jre\lib\jsse.jar;D:\jdk1.8.0_
51\jre\lib\management-
agent.jar;D:\jdk1.8.0_51\jre\lib\plugin.jar;D:\jdk1.8.0_51\jre\lib\resources.jar;D:\jdk
1.8.0_51\jre\lib\rt.jar;E:\WorkSpace\kgf-java-
learning2021\target\classes;D:\repository\org\springframework\boot\spring-
boot-starter-web\2.5.6\spring-boot-starter-web-
2.5.6.jar;D:\repository\org\springframework\boot\spring-boot-
starter\2.5.6\spring-boot-starter-
2.5.6.jar;D:\repository\org\springframework\boot\spring-boot-starter-
logging\2.5.6\spring-boot-starter-logging-
2.5.6.jar;D:\repository\ch\qos\logback\logback-classic\1.2.6\logback-classic-
1.2.6.jar;D:\repository\ch\qos\logback\logback-core\1.2.6\logback-core-
1.2.6.jar;D:\repository\org\apache\logging\log4j\log4j-to-slf4j\2.14.1\log4j-to-
slf4j-2.14.1.jar;D:\repository\org\apache\logging\log4j\log4j-api\2.14.1\log4j-api-
2.14.1.jar;D:\repository\org\slf4j\jul-to-slf4j\1.7.32\jul-to-slf4j-

1.7.32.jar;D:\repository\jakarta\annotation\jakarta.annotation-api\1.3.5\jakarta.annotation-api-1.3.5.jar;D:\repository\org\yaml\snakeyaml\1.28\snakeyaml-1.28.jar;D:\repository\org\springframework\boot\spring-boot-starter-json\2.5.6\spring-boot-starter-json-2.5.6.jar;D:\repository\com\fasterxml\jackson\core\jackson-databind\2.12.5\jackson-databind-2.12.5.jar;D:\repository\com\fasterxml\jackson\core\jackson-annotations\2.12.5\jackson-annotations-2.12.5.jar;D:\repository\com\fasterxml\jackson\core\jackson-core\2.12.5\jackson-core-2.12.5.jar;D:\repository\com\fasterxml\jackson\datatype\jackson-datatype-jdk8\2.12.5\jackson-datatype-jdk8-2.12.5.jar;D:\repository\com\fasterxml\jackson\datatype\jackson-datatype-jsr310\2.12.5\jackson-datatype-jsr310-2.12.5.jar;D:\repository\com\fasterxml\jackson\module\jackson-module-parameter-names\2.12.5\jackson-module-parameter-names-2.12.5.jar;D:\repository\org\springframework\boot\spring-boot-starter-tomcat\2.5.6\spring-boot-starter-tomcat-2.5.6.jar;D:\repository\org\apache\tomcat\embed\tomcat-embed-core\9.0.54\tomcat-embed-core-9.0.54.jar;D:\repository\org\apache\tomcat\embed\tomcat-embed-el\9.0.54\tomcat-embed-el-

9.0.54.jar;D:\repository\org\apache\tomcat\embed\tomcat-embed-websocket\9.0.54\tomcat-embed-websocket-9.0.54.jar;D:\repository\org\springframework\spring-web\5.3.12\spring-web-5.3.12.jar;D:\repository\org\springframework\spring-beans\5.3.12\spring-beans-5.3.12.jar;D:\repository\org\springframework\spring-webmvc\5.3.12\spring-webmvc-5.3.12.jar;D:\repository\org\springframework\spring-aop\5.3.12\spring-aop-5.3.12.jar;D:\repository\org\springframework\spring-context\5.3.12\spring-context-5.3.12.jar;D:\repository\org\springframework\spring-expression\5.3.12\spring-expression-5.3.12.jar;D:\repository\org\springframework\boot\spring-boot-devtools\2.5.6\spring-boot-devtools-2.5.6.jar;D:\repository\org\springframework\boot\spring-boot\2.5.6\spring-boot-2.5.6.jar;D:\repository\org\springframework\boot\spring-boot-autoconfigure\2.5.6\spring-boot-autoconfigure-2.5.6.jar;D:\repository\org\projectlombok\lombok\1.18.16\lombok-1.18.16.jar;D:\repository\org\slf4j\slf4j-api\1.7.32\slf4j-api-1.7.32.jar;D:\repository\org\springframework\spring-core\5.3.12\spring-core-5.3.12.jar;D:\repository\org\springframework\spring-jcl\5.3.12\spring-jcl-5.3.12.jar;D:\repository\com\google\guava\guava\20.0\guava-20.0.jar;D:\repository\org\apache\commons\commons-lang3\3.12.0\commons-lang3-3.12.0.jar;D:\repository\com\alibaba\fastjson\1.2.75\fastjson-1.2.75.jar;D:\repository\cn\hutool\hutool-all\5.5.1\hutool-all-5.5.1.jar

5.5.1.jar;D:\repository\org\apache\poi\poi-ooxml\4.1.2\poi-ooxml-
4.1.2.jar;D:\repository\org\apache\commons\commons-compress\1.19\commons-
compress-1.19.jar;D:\repository\com\github\virtuald\curvesapi\1.06\curvesapi-
1.06.jar;D:\repository\org\apache\poi\poi-ooxml-schemas\4.1.2\poi-ooxml-
schemas-4.1.2.jar;D:\repository\org\apache\xmlbeans\xmlbeans\3.1.0\xmlbeans-
3.1.0.jar;D:\repository\org\apache\poi\poi\4.1.2\poi-
4.1.2.jar;D:\repository\commons-codec\commons-codec\1.15\commons-codec-
1.15.jar;D:\repository\org\apache\commons\commons-
collections4\4.4\commons-collections4-
4.4.jar;D:\repository\org\apache\commons\commons-math3\3.6.1\commons-
math3-3.6.1.jar;D:\repository\com\zaxxer\SparseBitSet\1.2\SparseBitSet-
1.2.jar;D:\JetBrains\idea20190303\lib\idea_rt.jar

java.vm.specification.version = 1.8

sun.arch.data.model = 64

sun.java.command = com.kgf.kgfjavalearning2021.jvm.GCTest

java.home = D:\jdk1.8.0_51\jre

user.language = zh

java.specification.vendor = Oracle Corporation

awt.toolkit = sun.awt.windows.WToolkit

java.vm.info = mixed mode

java.version = 1.8.0_51

java.ext.dirs = D:\jdk1.8.0_51\jre\lib\ext;C:\WINDOWS\Sun\Java\lib\ext

```
sun.boot.class.path =  
D:\jdk1.8.0_51\jre\lib\resources.jar;D:\jdk1.8.0_51\jre\lib\rt.jar;D:\jdk1.8.0_51\jre\lib  
\sunrsasign.jar;D:\jdk1.8.0_51\jre\lib\jsse.jar;D:\jdk1.8.0_51\jre\lib\jce.jar;D:\jdk1.8.0  
_51\jre\lib\charsets.jar;D:\jdk1.8.0_51\jre\lib\jfr.jar;D:\jdk1.8.0_51\jre\classes  
  
java.vendor = Oracle Corporation  
  
file.separator =  
  
java.vendor.url.bug = http://bugreport.sun.com/bugreport/  
  
sun.io.unicode.encoding = UnicodeLittle  
  
sun.cpu.endian = little  
  
sun.desktop = windows  
  
sun.cpu.isalist = amd64  
  
C:\Users\86136>
```

3、jinfo -flags pid: 查看曾经赋过值的参数值

命令：jinfo -flags 10696

```
C:\Users\86136>jps  
11248 RemoteMavenServer  
4992 Launcher  
2052  
10696 GCTest  
13596 Jps  
  
C:\Users\86136>jinfo -flags 10696  
Attaching to process ID 10696, please wait...  
Debugger attached successfully.  
Server compiler detected.  
VM build: 51-b03  
Non-default VM flags: -XX:CICompilerCount=4 -XX:InitialHeapSize=62914560 -XX:MaxHeapSize=62914560 -XX:MaxNewSize=20971520 -XX:MinHeapDeltaBytes=524288 -XX:NewSize=209715  
20 -XX:OldSize=41943040 -XX:SurvivorRatio=8 -XX:UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseFastUnorderedTimeStamps -XX:+UseLargePagesIndividualAllocation  
-XX:+UseParallelGC  
Command line: -Xmx60m -XX:SurvivorRatio=8 -javaagent:D:\JetBrains\idea20190303\lib\idea_rt.jar=1036:D:\JetBrains\idea20190303\bin -Dfile.encoding=UTF-8  
  
C:\Users\86136>
```

可以看到这个是我们在idea中设置过的参数

CSDN @爱上口袋的天空

```
C:\Users\86136>jps  
  
11248 RemoteMavenServer  
  
4992 Launcher  
  
2052  
  
10696 GCTest
```

13596 Jps

C:\Users\86136>jinfo -flags 10696

Attaching to process ID 10696, please wait...

Debugger attached successfully.

Server compiler detected.

JVM version is 25.51-b03

Non-default VM flags: -XX:CICompilerCount=4 -XX:InitialHeapSize=62914560 -

XX:MaxHeapSize=62914560 -XX:MaxNewSize=20971520 -

XX:MinHeapDeltaBytes=524288 -XX:NewSize=20971520 -XX:OldSize=41943040 -

XX:SurvivorRatio=8 -XX:+UseCompressedClassPointers -

XX:+UseCompressedOops -XX:+UseFastUnorderedTimeStamps -XX:-

UseLargePagesIndividualAllocation -XX:+UseParallelGC

Command line: -Xmx60m -XX:SurvivorRatio=8 -

javaagent:D:\JetBrains\idea20190303\lib\idea_rt.jar=1036:D:\JetBrains\idea201903

03\bin -Dfile.encoding=UTF-8

C:\Users\86136>

4、jinfo -flag <具体参数> pid: 查看具体参数的值

```
C:\Users\86136>jps  
11248 RemoteMavenServer  
1876 Jps  
2052  
12652 GCTest  
3308 Launcher  
  
C:\Users\86136>jinfo -flag MaxHeapSize 12652  
-XX:MaxHeapSize=62914560
```

```
C:\Users\86136>
```

CSDN @爱上口袋的天空

上面我们在程序里面设置了最大的堆内存是 60M,这里的内存刚好是 60M

5、使用 jinfo 进行参数修改

jinfo 不仅可以查看运行时某一个 Java 虚拟机参数的实际取值，甚至可以在运行时修改部分参数，并使之立即生效。但是，并非所有参数都支持动态修改。参数只有被标记 manageable 的 flag 可以被实时修改。其实，这个修改能力是极其有限的

可以查看被标记为manageable的参数

```
java -XX:+PrintFlagsFinal -version | grep manageable
```

intx CMSAbortablePrecleanWaitMillis	= 100	{manageable}
intx CMSWaitDuration	= 2000	{manageable}
bool HeapDumpAfterFullGC	= false	{manageable}
bool HeapDumpBeforeFullGC	= false	{manageable}
bool HeapDumpOnOutOfMemoryError	= false	{manageable}
ccstr HeapDumpPath	=	{manageable}
uintx MaxHeapFreeRatio	= 100	{manageable}
uintx MinHeapFreeRatio	= 0	{manageable}
bool PrintClassHistogram	= false	{manageable}
bool PrintClassHistogramAfterFullGC	= false	{manageable}
bool PrintClassHistogramBeforeFullGC	= false	{manageable}
bool PrintConcurrentLocks	= false	{manageable}
bool PrintGC	= false	{manageable}
bool PrintGCDates	= false	{manageable}
bool PrintGCDetails	= false	{manageable}
bool PrintGCTimeStamps	= false	{manageable}

5.1、布尔类型: jinfo -flag +-参数 pid

5.2、非布尔类型: jinfo -flag 参数名=参数值 pid

6、查看 JVM 启动参数值

- java -XX: -PrintFlagsInitial 查看所有 JVM 参数启动的初始值
- java -XX: +PrintFlagsFinal 查看所有 JVM 参数的最终值(这个用的更多一些)
- java -XX: +PrintCommandLineFlags 查看那些已经被用户或者 JVM 设置过的详细的 XX 参数的名称和值

上面前两个用的比较多一些，特别是第二个

6.1、我们使用 java -XX: +PrintFlagsFinal 输出到一个文件中

命令：java -XX: +PrintFlagsFinal >1.txt

```
C:\Users\86136>java -XX:+PrintFlagsFinal >1.txt
用法: java [-options] class [args...]
          (执行类)
或   java [-options] -jar jarfile [args...]
          (执行 jar 文件)

其中选项包括:
-d32           使用 32 位数据模型 (如果可用)
-d64           使用 64 位数据模型 (如果可用)
-server        选择 "server" VM
               默认 VM 是 server.

-cp <目录和 zip/jar 文件的类搜索路径>
-classpath <目录和 zip/jar 文件的类搜索路径>
           用 ; 分隔的目录, JAR 档案
           和 ZIP 档案列表, 用于搜索类文件。
-D<名称>=<值>      设置系统属性
-verbose:[class|gc|jni]
               安全性级别
```

CSDN @爱上口袋的天空

1.txt 内容如下：

下面我们查看 JVM 的最大堆内存：

```
bool IgnoreEmptyClassPaths          = false
bool IgnoreUnrecognizedVMOptions    = false
uintx IncreaseFirstTierCompileThresholdAt = 50
bool IncrementalInline              = true
uintx InitialBootClassLoaderMetaspaceSize = 4194304
uintx InitialCodeCacheSize          = 2555904
uintx InitialHeapSize               := 266338304
uintx InitialRAMFraction            = 64
uintx InitialSurvivorRatio          = 8
uintx InitialTenuringThreshold     = 7
uintx InitiatingHeapOccupancyPercent = 45
bool Inline                         = true
                                         {product}
                                         {product}
                                         {product}
                                         {product}
                                         {C2 product}
                                         {product}
                                         {product}
                                         {pd product}
                                         {product}
                                         {product}
                                         {product}
                                         {product}
                                         {product}
                                         {product}
```

可以发现最大堆内存为 4G, 初始化堆内存为 256M, 安装下面的计算时正确的，因为我的

电脑是 16G 的内存

默认情况下：

- 初始内存大小：物理电脑内存大小/64
- 最大内存大小：物理电脑内存大小/4

6.3、java -XX: +PrintCommandLineFlags

```
C:\Users\86136>java -XX:+PrintCommandLineFlags -version  
-XX:InitialHeapSize=266275648 -XX:MaxHeapSize=4260410368 -XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers -XX:  
+UseCompressedOops -XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC  
java version "1.8.0_51"  
Java(TM) SE Runtime Environment (build 1.8.0_51-b16)  
Java HotSpot(TM) 64-Bit Server VM (build 25.51-b03, mixed mode)  
C:\Users\86136> CSDN @爱上口袋的天空
```

```
C:\Users\86136>java -XX:+PrintCommandLineFlags -version  
-XX:InitialHeapSize=266275648 -XX:MaxHeapSize=4260410368 -  
XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers -  
XX:+UseCompressedOops -XX:-UseLargePagesIndividualAllocation -  
XX:+UseParallelGC  
java version "1.8.0_51"  
Java(TM) SE Runtime Environment (build 1.8.0_51-b16)  
Java HotSpot(TM) 64-Bit Server VM (build 25.51-b03, mixed mode)
```

jmap：Java 内存映像工具

https://blog.csdn.net/weixin_42272869/article/details/124190855

用于生成堆转储快照，还可以查询 finalize 执行队列、Java 堆和方法区的详细信息，如空间使用率，当前使用的是哪种收集器等。和 jinfo 一样，部分功能在 Windows 受限，除了生成堆转储快照的 -dump 和查看每个类实例的 -histo 外，其余选项只能在 Linux 使用。

jmap 相关命令：

可通过 jmap --help 查看

- jmap [option] : 连接运行的进程，分析内存情况
- jmap [option] <executable> : 分析 core dump 文件
- jmap [option] [server_id@] : 连接到远程的服务器进行分析
- jmap -help / java -h : 查看帮助信息

1、jmap [option]

jmap [option]

- pid: 可通过 jps 或者 ps 获得

示例如上面示例图一样，用 ps 查询到 java 的进程号 25488，然后 jmap 25488 查看到当前进程的相关信息

2、jmap [option] <executable>

jmap [option] <executable>

- executable: 生成核心转储的 Java 可执行文件。
- core: 要打印配置信息的核心文件

3、jmap [option] [server_id@]

- server-id: 当很多 debug 服务在远程跑的时候，这个服务的唯一 ID
- remote-hostname-or-IP: 远程服务的 IP 或者 hostname

4、[option]

- -dump: 生成 Java 堆快储快照，如下
- live: 仅将存活对象 dump 出
- format: 编码格式
- file: 生成得文件名称

- **-finalizerinfo** : 显示出等待执行 finalize 方法得对象
- **-heap** : 显示 Java 堆详细信息
- **-histo[:live]** : 显示堆中对象的详细信息, 如果加了 live, 只显示存活得对象信息
- **-clstats** : 显示出加载器静态对象

选 项	作 用
-dump	生成 Java 堆转储快照。格式为: -dump[:live,]format=b, file=<filename>, 其中 live 子参数说明是否只 dump 出存活的对象
-finalizerinfo	显示在 F-Queue 中等待 Finalizer 线程执行 finalize 方法的对象。只在 Linux / Solaris 平台下有效
-heap	显示 Java 堆详细信息, 如使用哪种回收器、参数配置、分代状况等。只在 Linux / Solaris 平台下有效
-histo	显示堆中对象统计信息, 包括类、实例数量、合计容量
-permstat	以 ClassLoader 为统计口径显示永久代内存状态。只在 Linux / Solaris 平台下有效
-F	当虚拟机进程对 -dump 选项没有响应时, 可使用这个选项强制生成 dump 快照。只在 Linux / Solaris 平台下有效 CSDN @pan_mlpan

常用命令:

```
jmap [pid] # 查看具体情况

jmap -dump:live,format=b,file=xxx.xxx [pid] # 将当前 Java 进程的内存占用情况导出来

jmap -histo:live [pid] >a.log # 显示存活得对象信息

jmap -finalizerinfo [pid] # 查看 等待执行 finalize 方法的数量

jmap -heap [pid] # 堆摘要信息
```

示例: (这里有一个 Java 应用, pid 为 25488)

1、jmap -heap [pid] : 显示堆的详细信息

执行:

```
jmap -heap 25488
```

执行情况, 显示二个主要的信息

- 堆的配置 (Heap Configuration)
- 堆使用 (Heap Usage) (新生代、老年代)

```
lms@weplib:~$ jmap -heap 25488
Attaching to process ID 25488, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.40-b25

using thread-local object allocation.
Parallel GC with 2 thread(s)

Heap Configuration:
  MinHeapFreeRatio      = 0
  MaxHeapFreeRatio      = 100
  MaxHeapSize           = 2092957696 (1996.0MB)
  NewSize               = 44040192 (42.0MB)
  MaxNewSize             = 697303040 (665.0MB)
  OldSize               = 88080384 (84.0MB)
  NewRatio              = 2
  SurvivorRatio          = 8
  MetaspaceSize          = 21807104 (20.796875MB)
  CompressedClassSpaceSize = 1073741824 (1024.0MB)
  MaxMetaspaceSize       = 17592186044415 MB
  G1HeapRegionSize        = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
  capacity = 651689984 (621.5MB)
  used     = 457513928 (436.3192825317383MB)
  free     = 194176056 (185.18071746826172MB)
  70.2042288684445% used
From Space:
  capacity = 17825792 (17.0MB)
  used     = 5208424 (4.967140197753906MB)
  free     = 12617368 (12.032859802246094MB)
  29.218471751493567% used
To Space:
  capacity = 18350080 (17.5MB)
  used     = 0 (0.0MB)
  free     = 18350080 (17.5MB)
  0.0% used
PS Old Generation
  capacity = 282591232 (269.5MB)
  used     = 226835016 (216.32672882080078MB)
  free     = 55756216 (53.17327117919922MB)
  80.26965818953647% used

39393 interned Strings occupying 3980672 bytes.
lms@weplib:~$
```

CSDN @pan_mlpan

2、jmap -finalizerinfo [pid] : 查看 等待执行 finalize 方法的数量

执行：

```
jmap -finalizerinfo 25488
```

执行情况，显示待执行 finalize 方法的对象数量

```
lms@weplib:~$ jmap -finalizerinfo 25488
Attaching to process ID 25488, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.40-b25
Number of objects pending for finalization: 0
```

3、**jmap -dump:live,format=b,file=xxx.xxx [pid]** #将当前 Java 进程的内存占用情况导出来

执行：

```
jmap -dump:live,format=b,file=/home/lms/a.txt 25488
```

执行情况，在指定路径下生成指定快照文件 a.txt，生成的文件可以用 jhat 工具来进行分析

```
lms@weplib:~$ jmap -dump:live,format=b,file=/home/lms/a.txt 25488
Dumping heap to /home/lms/a.txt ...
Heap dump file created
```

生成的文件用 jhat 来分析，默认开启端口 7000

```
lms@weplib:~$ jhat /home/lms/a.txt
Reading from /home/lms/a.txt...
Dump file created Fri Apr 15 11:10:01 CST 2022
Snapshot read, resolving...
Resolving 2926799 objects...
Chasing references, expect 585 dots.
.
.
.
Eliminating duplicate references...
.
.
.

Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

CSDN @pan_mlpan

可以根据 ip:7000 来查看：



Package <Arrays>

```
class [Lantlr.Token; [0x87e0cf98]
class [Lantlr.collections.AST; [0x87e0cf30]
class [Lch.qos.logback.classic.spi.IThrowableProxy; [0x87f0c2a8]
class [Lch.qos.logback.classic.spi.StackTraceElementProxy; [0x87f0c1d8]
class [Lch.qos.logback.classic.spi.ThrowableProxy; [0x87f0c240]
class [Lch.qos.logback.core.Appender; [0x83dc8eb0]
class [Lch.qos.logback.core.filter.Filter; [0x83dc8fe8]
class [Lch.qos.logback.core.joran.spi.ConsoleTarget; [0x83dc8f80]
class [Lch.qos.logback.core.pattern.parser	TokenName$TokenizerState; [0x83dc7c68]
class [Lch.qos.logback.core.spi.ContextAware; [0x83dc9120]
class [Lch.qos.logback.core.spi.ContextAwareBase; [0x83dc90b8]
class [Lch.qos.logback.core.spi.FilterAttachable; [0x83dc8f18]
class [Lch.qos.logback.core.spi.FilterReply; [0x83dc8bd8]
class [Lch.qos.logback.core.spi.LifeCycle; [0x83dc9050]
class [Lch.qos.logback.core.subst.Node$type; [0x83dc7e08]
class [Lch.qos.logback.core.subst.Token$type; [0x83dc7e70]
class [Lch.qos.logback.core.subst.Tokenizer$TokenizerState; [0x83dc7ed8]
class [Lcom.alibaba.druid.pool.DruidConnectionHolder; [0x842ebe80]
class [Lcom.alibaba.druid.pool.PreparedStatementPool$MethodType; [0x87e0cc58]
class [Lcom.dcampus.common.util.ResponseCodeUtil; [0x83e9f7b8]
class [Lcom.dcampus.lms.annotation.ParamValid; [0x83e97400]
class [Lcom.dcampus.lms.enums.AIServiceEnum; [0x87e0bdb8]
class [Lcom.dcampus.lms.enums.CourseStatusEnum; [0x87e0c438]
class [Lcom.dcampus.lms.enums.CourseTypeEnum; [0x87e0c3d0]
class [Lcom.dcampus.lms.enums.FileDataParamNameEnum; [0x88081ae0]
class [Lcom.dcampus.lms.enums.LoginResultEnum; [0x87e0c7e0]
class [Lcom.fasterxml.classmate.AnnotationInclusion; [0x847358f8]
class [Lcom.fasterxml.classmate.Annotations; [0x847357b8]
class [Lcom.fasterxml.classmate.ResolvedType; [0x83dc8488]
class [Lcom.fasterxml.classmate.members.HierarcicType; [0x84735d08]
class [Lcom.fasterxml.classmate.members.RawConstructor; [0x83dc83b8]
class [Lcom.fasterxml.classmate.members.RawField; [0x83dc8350]
class [Lcom.fasterxml.classmate.members.RawMember; [0x83dc8420]
class [Lcom.fasterxml.classmate.members.RawMethod; [0x83dc82e8]
class [Lcom.fasterxml.classmate.members.ResolvedConstructor; [0x84735960]
class [Lcom.fasterxml.classmate.members.ResolvedField; [0x84735a30]
```

4、jmap -histo:live [pid] >a.log #将当前 Java 进程的内存占用情况导出来

执行：

```
jmap -histo:live 25488 >/home/lms/a.log
```

执行情况，在指定路径下生成指定导出文件 a.log

```
lms@weplib:~$ jmap -histo:live 25488 >/home/lms/a.log  
lms@weplib:~$
```

查看该文件内容：

num	#instances	#bytes	class name
1:	312906	60517352	[C
2:	30493	17768296	[B
3:	310546	7453104	java.lang.String
4:	191842	6138944	java.util.concurrent.ConcurrentHashMap\$Node
5:	47925	4217400	java.lang.reflect.Method
6:	129164	4133248	java.util.HashMap\$Node
7:	43053	3314504	[Ljava.util.HashMap\$Node;
8:	28288	3150488	java.lang.Class
9:	55675	2243752	[Ljava.lang.Object;
10:	45142	1805680	java.util.LinkedHashMap\$Entry
11:	1226	1780032	[Ljava.util.concurrent.ConcurrentHashMap\$Node;
12:	69178	1660272	org.hibernate.engine.spi.TypedValue
13:	69178	1660272	org.hibernate.internal.util.ValueHolder
14:	58024	1652600	[I
15:	30579	1467792	java.util.HashMap
16:	19344	1392768	org.ehcache.impl.internal.store.heap.holders.SerializedOnHeapValue
17:	49196	1282016	[Ljava.lang.String;
18:	38200	1222400	org.hibernate.engine.jdbc.Size
19:	21440	1200640	java.util.LinkedHashMap
20:	71808	1148928	java.lang.Integer
21:	69178	1106848	org.hibernate.engine.spi.TypedValue\$1
22:	19035	1065960	org.hibernate.cache.spi.QueryKey
23:	19344	928512	java.nio.HeapByteBufferR
24:	17979	862992	org.aspectj.weaver.reflect.ShadowMatchImpl
25:	17794	854112	org.aspectj.weaver.reflect.ShadowMatchImpl
26:	17671	848208	org.ehcache.impl.internal.concurrent.ConcurrentHashMap\$TreeNode
27:	26020	732584	[Z
28:	27365	656760	org.apache.catalina.loader.ResourceEntry
29:	19254	616128	org.hibernate.type.IntegerType
30:	19035	609120	org.hibernate.transform.CachableResultTransformer
31:	18768	600576	org.hibernate.type.TimestampType
32:	17979	575328	org.aspectj.weaver.patterns.ExposedState
33:	23871	572904	org.springframework.core.MethodClassKey
34:	17794	569408	org.aspectj.weaver.patterns.ExposedState
35:	23682	568368	org.springframework.core.MethodClassKey
36:	35081	561296	java.lang.Object
37:	23071	553704	java.util.ArrayList
38:	10685	512880	org.apache.tomcat.util.buf.ByteChunk
39:	15498	495936	java.lang.ref.WeakReference
40:	10081	483888	org.apache.tomcat.util.buf.CharChunk
41:	21450	480016	[Ljava.lang.Class;
42:	9931	476688	org.apache.tomcat.util.buf.MessageBytes
43:	19344	464256	org.ehcache.impl.internal.store.heap.holders.CopiedOnHeapKey
44:	19254	462096	org.hibernate.type.descriptor.java.IntegerTypeDescriptor
45:	11370	454800	java.util.WeakHashMap\$Entry
46:	18908	453792	java.util.Date
47:	18768	450432	org.hibernate.type.descriptor.java.JdbcTimestampTypeDescriptor
48:	5690	409680	java.lang.reflect.Field
49:	8182	392736	java.util.concurrent.locks.ReentrantReadWriteLock\$NonfairSync
50:	19674	314784	java.util.HashMap\$EntrySet
51:	19276	313624	[Lorg.hibernate.type.Type;
52:	19326	309216	org.hibernate.internal.util.compare.ComparableComparator
53:	19326	309216	org.hibernate.type.descriptor.java.ImmutableMutabilityPlan
54:	19254	308064	org.hibernate.type.descriptor.sql.IntegerTypeDescriptor
55:	18768	300288	org.hibernate.type.descriptor.java.JdbcTimestampTypeDescriptor\$Type
56:	18768	300288	org.hibernate.type.descriptor.sql.TimestampTypeDescriptor
57:	5566	222640	java.lang.ref.SoftReference

CSDN @pan_mlpan

常用 JVM 参数：

-Xms: 初始堆大小，默认为物理内存的 1/64(<1GB)；默认(MinHeapFreeRatio 参数可以调整)空余堆内存小于 40%时，JVM 就会增大堆直到-Xmx 的最大限制

-Xmx: 最大堆大小，默认(MaxHeapFreeRatio 参数可以调整)空余堆内存大于 70%时，

JVM 会减少堆直到 -Xms 的最小限制

-Xmn: 新生代的内存空间大小, 注意: 此处的大小是 (eden+ 2 survivor space)。与 jmap -heap 中显示的 New gen 是不同的。整个堆大小=新生代大小 + 老生代大小 + 永久代大小。在保证堆大小不变的情况下, 增大新生代后, 将会减小老生代大小。此值对系统性能影响较大,Sun 官方推荐配置为整个堆的 3/8。

-XX:SurvivorRatio: 新生代中 Eden 区域与 Survivor 区域的容量比值, 默认值为 8。两个 Survivor 区与一个 Eden 区的比值为 2:8, 一个 Survivor 区占整个年轻代的 1/10。

-Xss: 每个线程的堆栈大小。JDK5.0 以后每个线程堆栈大小为 1M, 以前每个线程堆栈大小为 256K。应根据应用的线程所需内存大小进行适当调整。在相同物理内存下, 减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的, 不能无限生成, 经验值在 3000~5000 左右。一般小的应用, 如果栈不是很深, 应该是 128k 够用的, 大的应用建议使用 256k。这个选项对性能影响比较大, 需要严格的测试。和 threadstacksize 选项解释很类似, 官方文档似乎没有解释, 在论坛中有这样一句话: "-Xss is translated in a VM flag named ThreadStackSize" 一般设置这个值就可以了。

-XX:PermSize: 设置永久代(perm gen)初始值。默认值为物理内存的 1/64。

-XX:MaxPermSize: 设置持久代最大值。物理内存的 1/4。

jhat: 虚拟机堆转储快照分析工具

JDK 提供 jhat 与 jmap 搭配使用分析 jmap 生成的堆转储快照。jhat 内置了一个微型的 HTTP/Web 服务器, 生成堆转储快照的分析结果后可以在浏览器查看。

使用 jhat 命令, 会启动一个 http 服务, 默认端口 7000。

注: jhat 命令在 JDK9、JDK10 中已经被删除, 官方建议用 VisualVM 代替。

jhat 使用

```
[root@bogon opt]# jhat /opt/1.hprof  
Reading from /opt/1.hprof...  
Dump file created Tue Mar 22 04:10:08 CST 2022  
Snapshot read, resolving...  
Resolving 6770 objects...  
Chasing references, expect 1 dots.  
Eliminating duplicate references.  
Snapshot resolved.  
Started HTTP server on port 7000  
Server is ready.
```

CSDN @不见长安见晨雾

jhat /opt/1.hprof

← → ⚡ 不安全 192.168.56.100:7000

All Classes (excluding platform)

Package com.studio

[class com.studio.ParamTest](#) [0xff867af0]

Other Queries

- [All classes including platform](#)
- [Show all members of the rootset](#)
- [Show instance counts for all classes \(including platform\)](#)
- [Show instance counts for all classes \(excluding platform\)](#)
- [Show heap histogram](#)
- [Show finalizer summary](#)
- [Execute Object Query Language \(OQL\) query](#)

CSDN @不见长安见晨雾

← → ⚡ 不安全 | 192.168.56.100:7000/allClassesWithPlatform/

All Classes (including platform)

Package <Arrays>

和使用 `jmap -histo 进程号` 输出的结果是一样的

```
class [B [0xff800558]  
class [C [0xff800420]  
class [D [0xff8004f0]  
class [E [0xff800488]  
class [I [0xff800628]  
class [J [0xff800690]  
class [Ljava.io.File$PathStatus; [0xff82c7a8]  
class [Ljava.io.File; [0xff830198]  
class [Ljava.io.ObjectStreamField; [0xff805470]  
class [Ljava.io.Serializable; [0xff804248]  
class [Ljava.lang.CharSequence; [0xff808198]  
class [Ljava.lang.Class; [0xff8042b0]  
class [Ljava.lang.ClassValue$Entry; [0xff84e1a0]  
class [Ljava.lang.Comparable; [0xff805408]  
class [Ljava.lang.Enum; [0xff82c740]  
..
```

CSDN @不见长安见晨雾

Object Query Language (OQL) query

[All Classes \(excluding platform\)](#) [OQL Help](#)

```
select s from java.lang.String s where s.value.length>100
```

可以使用OQL定位一些大对象

```
java.lang.String@0xff8444f8  
java.lang.String@0xff843c80  
java.lang.String@0xff807aa8  
java.lang.String@0xff817db8
```

CSDN @不见长安见晨雾

```
select s from java.lang.String s where s.value.length>100
```

```
select a from [I a where a.length > 256
```

```
select classof(cl).name from instanceof java.lang.ClassLoader cl
```

```
select v.elementAtData.length from java.util.Vector v
```

```
select v.elementAtData.length , v.elementAtData from java.util.Vector v where  
v.elementAtData.length>9
```

```
select list.elementAtData from java.util.ArrayList list
```

```
select list.elementAtData from java.util.ArrayList list where list.elementAtData.length>9
```

```
select user.age from com.test4.User user where user.age>18
```

jhat 指定端口

```
[root@bogon opt]# jhat -port 6666 /opt/1.hprof
Reading from /opt/1.hprof...
Dump file created Tue Mar 22 04:10:08 CST 2022
Snapshot read, resolving...
Resolving 6770 objects...
Chasing references, expect 1 dots.
Eliminating duplicate references.
Snapshot resolved.
Started HTTP server on port 6666
Server is ready.
```

CSDN @不见长安见晨雾

jstack: Java 堆栈跟踪工具

https://blog.csdn.net/weixin_44588186/article/details/124680586

用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的目的通常是定位线程出现长时间停顿的原因，**如线程间死锁、死循环、请求外部资源导致的长时间挂起等**。线程出现停顿时通过 jstack 查看各个线程的调用堆栈，可以获知没有响应的线程在后台做什么或等什么资源。

常用指令

jstack [-option] <pid> // 打印某个进程的堆栈信息

其他常用指令如下：

```
C:\Users\Admin-Zhuang>jstack
Usage:
  jstack [-l] <pid>
    (to connect to running process)
  jstack -F [-m] [-l] <pid>
    (to connect to a hung process)
  jstack [-m] [-l] <executable> <core>
    (to connect to a core file)
  jstack [-m] [-l] [<server_id@>]<remote server IP or hostname>
    (to connect to a remote debug server)

Options:
  -F  to force a thread dump. Use when jstack <pid> does not respond (process is hung)
  -m  to print both java and native frames (mixed mode)
  -l  long listing. Prints additional information about locks
  -h or -help to print this help message
```

CSDN @哦...吼吼吼

指令说明

- -F 当 jstack 指令无响应时，强制打印一个堆栈信息

- -m 打印包含 Java 和 C/C++ 帧的混合模式堆栈跟踪
- -l 打印关于锁的其他信息，比如拥有 java.util.concurrent.ownable 同步器的列表
- -h / -help 打印帮助信息

Jstack 实战

案例一：jstack 分析死锁

```
public class DeadLock {  
  
    private static Object lockA = new Object();  
  
    private static Object lockB = new Object();  
  
    public static void main(String[] args) {  
  
        new Thread(() -> {  
  
            synchronized (lockA) {  
  
                try {  
  
                    System.out.println("线程 1 开始运行=====");  
  
                    Thread.sleep(2000);  
  
                } catch (InterruptedException e) {  
  
                }  
  
                synchronized (lockB) {  
  
                    System.out.println("线程 1 运行结束=====");  
  
                }  
  
            }  
  
        }).start();  
  
        new Thread(() -> {  
  
    }
```

```
synchronized (lockB) {  
  
    try {  
  
        System.out.println("线程 2 开始运行=====");  
  
        Thread.sleep(2000);  
  
    } catch (InterruptedException e) {  
  
    }  
  
    synchronized (lockA) {  
  
        System.out.println("线程 2 结束运行=====");  
  
    }  
  
}  
  
}).start();  
  
  
System.out.println("主线程运行结束=====");  
}  
}
```

运行结果如下：

线程 1 开始运行=====

主线程运行结束=====

线程 2 开始运行=====

可见，线程 1 和线程 2 都没有运行完成，两个线程在抢夺资源过程中陷入了死循环

排查思路

使用 jps 指令查看当前运行中的 java 程序

使用 jstack -l < pid > , 输出线程及锁详情

进行死锁分析

动手实践

那么首先使用 jps:

```
C:\Users\Admin-Zhuang>jps  
9360 Launcher  
596 Jps  
6852  
15276 RemoteMavenServer36  
5084 DeadLock CSDN @哦...吼吼吼
```

使用 jstack 指令:

```
C:\Users\Admin-Zhuang>jstack 5084  
CSDN @哦...吼吼吼  
  
Found one Java-level deadlock:  
=====  
"Thread-1":  
  waiting to lock monitor 0x000000001d290de8 (object 0x000000076b19b890, a java.lang.Object),  
  which is held by "Thread-0"  
"Thread-0":  
  waiting to lock monitor 0x000000001d293678 (object 0x000000076b19b8a0, a java.lang.Object),  
  which is held by "Thread-1"  
  
Java stack information for the threads listed above:  
=====  
"Thread-1":  
  at com.zzy.jvm.c7_jvm.DeadLock.lambda$main$1(DeadLock.java:31)  
  - waiting to lock <0x000000076b19b890> (a java.lang.Object)  
  - locked <0x000000076b19b8a0> (a java.lang.Object)  
  at com.zzy.jvm.c7_jvm.DeadLock$$Lambda$2/1342443276.run(Unknown Source)  
  at java.lang.Thread.run(Thread.java:745)  
"Thread-0":  
  at com.zzy.jvm.c7_jvm.DeadLock.lambda$main$0(DeadLock.java:18)  
  - waiting to lock <0x000000076b19b8a0> (a java.lang.Object)  
  - locked <0x000000076b19b890> (a java.lang.Object)  
  at com.zzy.jvm.c7_jvm.DeadLock$$Lambda$1/1706234378.run(Unknown Source)  
  at java.lang.Thread.run(Thread.java:745)  
  
Found 1 deadlock.
```

CSDN @哦...吼吼吼

通过上图，可以得到死锁的信息：

```
"Thread-1":  
  
  waiting to lock monitor 0x000000001d290de8 (object 0x000000076b19b890, a  
  java.lang.Object),  
  which is held by "Thread-0"  
  
"Thread-0":
```

```
waiting to lock monitor 0x000000001d293678 (object 0x000000076b19b8a0, a  
java.lang.Object),  
which is held by "Thread-1"
```

并且可以定位到对应代码位置，是不是非常的方便~

案例二：jstack 分析 CPU 过高

实例代码如下：

```
public class CpuProblem {  
  
    private static ExecutorService executorService =  
        Executors.newFixedThreadPool(5);  
  
    public static void main(String[] args) {  
  
        Task task1 = new Task();  
  
        Task task2 = new Task();  
  
        executorService.execute(task1);  
  
        executorService.execute(task2);  
  
    }  
  
    public static Object lock = new Object();  
  
    static class Task implements Runnable {  
  
        public void run() {  
  
            synchronized (lock) {  
  
                long sum = 0L;  
  
                while (true) {
```

```
        sum += 1;

    }

}

}

}

}
```

排查思路

1. top
2. top -Hp <pid>
3. 获取 cpu 最高的进程号并转换为 16 进制<pid_16>
4. jstack <pid>|grep -A 10 <pid_16>
5. 查看对应问题的代码

动手实践

1. 使用命令 top -p , 显示你的 java 进程的内存情况, pid 是你的 java 进程号, 比如

19663

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
19663	root	20	0	2709764	21584	10620	S	99.0	0.8	8:17.30	java

2. 按 H(获取 top -Hp pid 指令), 获取每个线程的内存情况

```

top - 23:36:24 up 1:13, 5 users, load average: 1.77, 1.49, 0.85
Threads: 11 total, 1 running, 10 sleeping, 0 stopped, 0 zombie
%Cpu(s): 99.7 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.3 si, 0.0 st
KiB Mem : 2869804 total, 1962896 free, 416208 used, 490700 buff/cache
KiB Swap: 2097148 total, 2097148 free, 0 used. 2174036 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
19664 root 20 0 2709764 21584 10620 R 99.0 0.8 8:53.66 java
19663 root 20 0 2709764 21584 10620 S 0.0 0.8 0:00.00 java
19665 root 20 0 2709764 21584 10620 S 0.0 0.8 0:00.01 java
19666 root 20 0 2709764 21584 10620 S 0.0 0.8 0:00.00 java
19667 root 20 0 2709764 21584 10620 S 0.0 0.8 0:00.00 java
19668 root 20 0 2709764 21584 10620 S 0.0 0.8 0:00.00 java
19669 root 20 0 2709764 21584 10620 S 0.0 0.8 0:00.00 java
19670 root 20 0 2709764 21584 10620 S 0.0 0.8 0:00.00 java
19671 root 20 0 2709764 21584 10620 S 0.0 0.8 0:00.00 java
19672 root 20 0 2709764 21584 10620 S 0.0 0.8 0:00.19 java
19777 root 20 0 2709764 21584 10620 S 0.0 0.8 0:00.00 java

```

CSDN @哦...吼吼吼

3. 找到内存和 cpu 占用最高的线程 tid, 比如 19664
4. 转为十六进制得到 0x4cd0, 此为线程 id 的十六进制表示
5. 执行 jstack 19663|grep -A 10 4cd0, 得到线程堆栈信息中 4cd0 这个线程所在行的

后面 10 行, 从堆栈中可以发现导致 cpu 飙高的调用方法

```

[root@localhost ~]# jstack 19663|grep -A 10 4cd0
"main" #1 prio=5 os_prio=0 tid=0x00007fbb30009800 nid=0x4cd0 runnable [0x00007fbb380e5000]
    java.lang.Thread.State: RUNNABLE
        at com.tuling.jvm.Math.main(Math.java:22)

"VM Thread" os_prio=0 tid=0x00007fbb3006e000 nid=0x4cd1 runnable
"VM Periodic Task Thread" os_prio=0 tid=0x00007fbb300b7000 nid=0x4cd8 waiting on condition
JNI global references: 5

```

CSDN @哦...吼吼吼

6. 查看对应的堆栈信息找出可能存在问题的代码

finalize 的理解

一般的回答是：它是 Object 中的一个方法，子类重写它，垃圾回收时此方法会被调用，

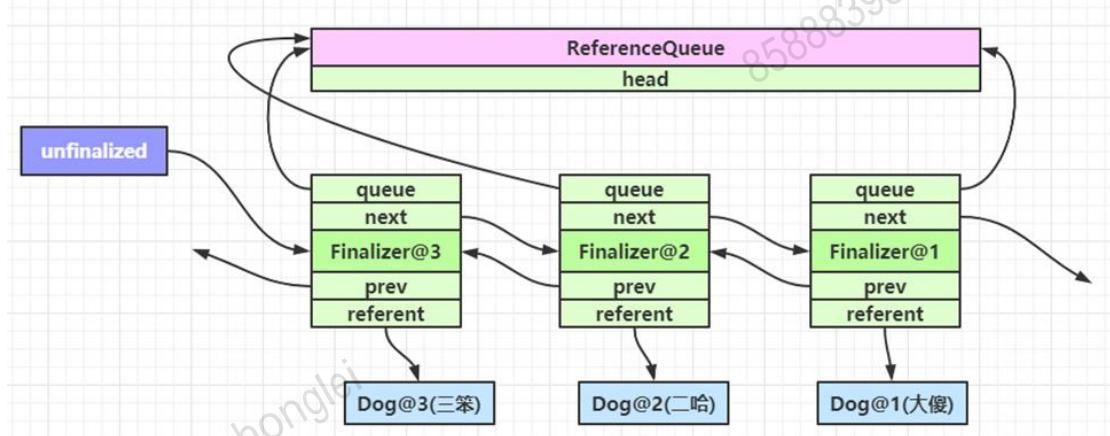
可以在其中进行一些资源释放和清理工作

优秀的回答是：将资源释放和清理放在 finalize 方法中非常不好，非常影响性能，严重时甚至会引起 OOM，从 Java9 开始就被标注为 @Deprecated，不建议被使用了

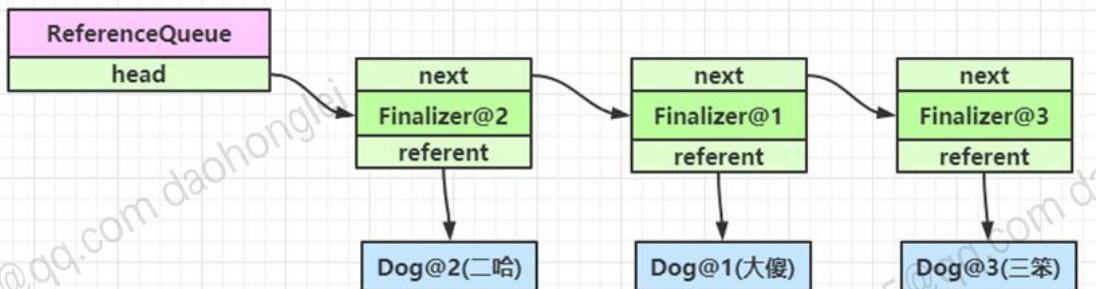
两个重要队列

unfinalized 队列：当重写了 finalize 方法的对象，在构造方法调用之时，JVM 都会将其包

装成一个 Finalizer 对象，并加入 unfinalized 队列中（静态成员变量、双向链表结构）



ReferenceQueue 队列：第二个重要的队列，也是 Finalizer 类中一个静态成员变量，名为 queue (是一个单向链表结构)，刚开始它是空的。当狗对象可以被当作垃圾回收时，就会把这些狗对象对应的 Finalizer 对象加入这个队列

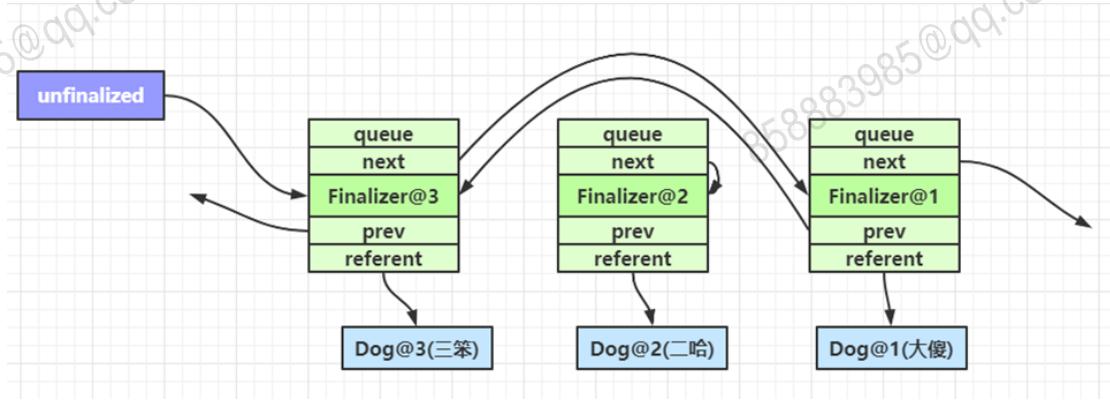


真正回收时机

即使 Dog 对象没人引用，垃圾回收时也没法立刻回收它，因为 Finalizer 还在引用它嘛，

为的是【先别着急回收啊，等我调完 finalize 方法，再回收】

查看 FinalizerThread 线程内的代码，这个线程从 ReferenceQueue 中逐一取出每个 Finalizer 对象，把它们从链表断开，这样没谁能引用到它，以及其对应的狗对象，所以下次 gc 时就可以被回收了



为什么 finalize 方法非常不好，非常影响性能

非常不好

- FinalizerThread 是守护线程，代码很有可能没来得及执行完，线程就结束了，造成资源没有正确释放
- 异常被吞掉这个就太糟了，你甚至不能判断有没有在释放资源时发生错误

影响性能

- 重写了 finalize 方法的对象在第一次被 gc 时，并不能及时释放它占用的内存，因为要等着 FinalizerThread 调用完 finalize，把它从第一个 unfinalized 队列移除后，第二次 gc 时才能真正释放内存
- 可以想象 gc 本就因为内存不足引起，finalize 调用又很慢（两个队列的移除操作，都是串行执行的，用来释放连接类的资源也应该不快），不能及时释放内存，对象释放不及时就会逐渐移入老年代，老年代垃圾积累过多就会容易 full gc，full gc 后释放速度如果仍跟不上创建新对象的速度，就会 OOM

质疑

- 有的文章提到【Finalizer 线程会和我们的主线程进行竞争，不过由于它的优先级较低，获取到的 CPU 时间较少，因此它永远也赶不上主线程的步伐】这个显然是错误的，FinalizerThread 的优先级较普通线程更高，赶不上步伐的原因应该是

finalize 执行慢等原因综合导致

关于 Java 的 JIT(即时编译器)

前言

之前在整理终于搞懂了！字符串拼接的各种姿势以及底层的小知识这篇文章的时候提到了锁消除，然后了解到逃逸分析，之后又去学习了相关知识，了解到 Java 逃逸分析只出现在 JIT（即时编译器）进行。这时候随着深入了解，出现了以下几个问题。

一、JIT(即时编译器)

了解 Java 逃逸分析只出现在 JIT（Just In Time）进行，就不禁想 JIT 是什么。

1.1 解释执行和编译执行的区别

解释执行：解释执行是采用匹配执行解释器（解释器是个黑盒，通常也有编译器的组成部分）内部已经编译好的机器码，不是生成新的机器码（也有说法是逐条翻译成机器码？）。 - 由于逐条翻译，程序启动快，但是执行效率不高。

编译执行：运行期间，通过将字节码编译成对应的新的机器码（会将其缓存起来，通过参数-XX:ReservedCodeCacheSize），然后执行。 - 需要先编译出新的机器指令，所以程序启动较慢，但是执行效率高（因为执行的是机器指令）。

1.2 Java 代码编译过程

在讲 JIT 之前，简单过一遍刚开始时，Java 执行的过程

Java 将源代码翻译为字节码 -> JVM 逐条解释为机器码 -> 执行 -> 执行结果

后来因为解释执行必然比执行编译好的机器指令的执行效率低，所以引入 JIT（即时编译器）。在执行时，JIT 会把翻译过的机器码保存起来，已备下次使用，因此从理论上来说，采用 JIT 技术能够在执行效率上，接近曾经纯编译技术。

1.3 JIT 是什么

JIT 编译器 (just in time 即时编译器) , 当虚拟机发现某个方法或代码块运行特别频繁时, 就会把这些代码认定为(Hot Spot Code) 热点代码, 为了提高热点代码的执行效率, 在运行时, 虚拟机将会把这些代码编译成与本地平台相关的机器码, 并进行各层次的优化。

那么, 虚拟机是怎么知道哪些是热点代码或者运行频繁呢, 那就要说说 HotSpot 了。

二、HotSpot 是什么

现在主流的商用虚拟机 (HotSpot (Oracle) 、J9 VM (IBM)) 中几乎都同时包含解释器和编译器。

二者在其中各有优势: **当程序需要迅速启动和执行时, 解释器可以首先发挥作用, 省去编译的时间, 立即执行;** 当程序运行后, 随着时间的推移, **编译器逐渐会返回作用, 把越来越多的代码编译成本地代码后, 可以获取更高的执行效率。** 解释执行可以节约内存, 而编译执行可以提升效率。

2.1 说 JIT 比解释快, 其实说的是“执行编译后的代码”比“解释器解释执行”要快

并不是说“编译”这个动作比“解释”这个动作快。JIT 编译再怎么快, 至少也比解释执行一次略慢一些, 而要得到最后的执行结果还得再经过一个“执行编译后的代码”的过程。所以, 对“只执行一次”的代码而言, 解释执行其实总是比 JIT 编译执行要快。

怎么算是“**只执行一次的代码**”呢? 粗略说, 下面两个条件同时满足时就是严格的“只执行一次”

1. 只被调用一次, 例如类的构造器 (class initializer, `()`)
2. 没有循环

2.2 对于为何 JIT 只对“热点代码”进行即时编译

大概分为两个情况考虑

1. 时间

对只执行一次的代码做 JIT 编译再执行，可以说是得不偿失。**对只执行少量次数的代码，JIT 编译带来的执行速度的提升也未必能抵消掉最初编译带来的开销。只有对频繁执行的代码，JIT 编译才能保证有正面的收益。**

2. 空间

对一般的 Java 方法而言，编译后代码的大小相对于字节码的大小，膨胀比达到 10x 是很正常的。同上面说的时间开销一样，这里的空间开销也是，只有对执行频繁的代码才值得编译，**如果把所有代码都编译则会显著增加代码所占空间，导致“代码爆炸”**。这也就解释了为什么有些 JVM 会选择不总是做 JIT 编译，而是选择用解释器+JIT 编译器的混合执行引擎。

2.3 热点探测技术 (Hot Spot Detection)

程序中的代码只有是热点代码时，才会编译为本地代码，那么什么是热点代码呢？运行过程中会被即时编译器编译的“**热点代码**”有两类：

1. 被多次调用的方法。
2. 被多次执行的循环体

热点探测技术主要有以下两种：

2.3.1 基于采样的热点探测

采用这种方法的虚拟机会周期性地检查各个线程的栈顶，如果发现某些方法经常出现在栈顶，那这个方法就是“**热点方法**”。这种探测方法的**好处是实现简单高效，还可以很容易地获取方法调用关系**（将调用堆栈展开即可），**缺点是很难精确地确认一个方法的热度，容易因为受到线程阻塞或别的外界因素的影响而扰乱热点探测。**

2.3.2 基于计数器的热点探测

采用这种方法的虚拟机会为每个方法（甚至是代码块）建立计数器，统计方法的执行次数，如果执行次数超过一定的阈值，就认为它是“热点方法”。这种统计方法实现复杂一些，需要为每个方法建立并维护计数器，而且不能直接获取到方法的调用关系，但是它的统计结果相对更加精确严谨。

在这个方法下，JVM 会为每个方法或者代码块保留一个调用计数，以预定义的编译阈值 (-XX:CompileThreshold, client 模式下默认是 1500, server 模式默认是 10000) 开始，

这里听到两种情况，一种是计数器用减的，到 0 即时编译，一种是开始 0，执行一次加一，超过阈值即时编译。（结合之后的热点衰减，我这里就先认为是加的了）

两种情况，编译器都是以整个方法作为编译对象。这种编译方法因为编译发生在方法执行过程之中，因此形象的称之为栈上替换（On Stack Replacement, OSR），即方法栈帧还在栈上，方法就被替换了

2.3.3 Hotspot 选择的热点探测技术

在 HotSpot 虚拟机中使用的是第二种——基于计数器的热点探测方法，因此它为每个方法准备了两个计数器：方法调用计数器和回边计数器。在确定虚拟机运行参数的前提下，这两个计数器都有一个确定的阈值，当计数器超过阈值溢出了，就会触发 JIT 编译。编译阈值 (-XX:CompileThreshold, client 模式下默认是 1500, server 模式默认是 10000)

2.3.3.1 方法计数器

当一个方法被调用时，会先检查该方法是否存在被 JIT 编译过的版本，如果存在，则优先使用编译后的本地代码来执行。如果不存在已被编译过的版本，则将此方法的调用计

数器值加 1，然后判断方法调用计数器与回边计数器值之和是否超过方法调用计数器的阈值。如果已超过阈值，那么将会向即时编译器提交一个该方法的代码编译请求。

2.3.3.2 回边计数器

它的作用就是统计一个方法中循环体代码执行的次数，在字节码中遇到控制流向后跳转的指令称为“回边”。

作用是在遇到回边（可以狭义的理解为循环，但并非所有循环都是回边）时，虚拟机检测改代码是否存在已编译版本 - 是就执行，否加调用计数器+1，并检测是否超过阈值。

2.3.4 栈上替换(On Stack Replacement, OSR)

方法触发标准编译，循环体触发 OSR 栈上替换（在解释执行过程中直接切换到本地代码执行）。

在方法调用计数器种，如果不做任何设置，执行引擎并不会同步等待编译请求完成，而是继续进行解释器按照解释方式执行字节码，直到提交的请求被编译器编译完成。当编译工作完成之后，这个方法的调用入口地址就会系统自动改写成新的，下一次调用该方法时就会使用已编译的版本。

但在实际情况中并不总是有“下一次调用”的机会。假如有一个包含了千万次的循环方法，方法只执行一次，此时如果等待方法执行完成再进行编译，由于方法只调用一次，编译器将没有机会使用编译后的代码。

为了防止编译器做这种无用功，需要一种技术在解释执行循环期间将代码替换为编译后的代码，即循环的第 N 次使用解释执行，第 N+1 次使用编译后的代码，这样就能将“下一次调用”缩小到“下一次循环”。这种技术叫作栈上替换(On Stack Replacement, OSR)。 OSR 机制类似协程切换，它将解释器栈的数据打包到 OSR

buffer，然后在编译后的代码里面提取 OSR buffer 的数据放入编译后的执行栈，再继续执行。

2.4 热点衰减

如果不做任何设置，方法调用计数器统计的并不是方法被调用的绝对次数，而是一个相对的执行频率，即一段时间之内方法被调用的次数。当超过一定的时间限度，如果方法的调用次数仍然不足以让它提交给即时编译器编译，那这个方法的调用计数器就会被减少一半，这个过程称为方法调用计数器热度的衰减（Counter Decay），而这段时期就称为此方法统计的半衰周期（Counter Half Life Time）。半衰周期是化学中的概念，比如出土的文物通过查看 C60 来获得文物的年龄。进行热度衰减的动作是在虚拟机进行垃圾收集时顺便进行的，可以使用虚拟机参数 `-XX:-UseCounterDecay` 来关闭热度衰减，让方法计数器统计方法调用的绝对次数，这样，只要系统运行时间足够长，绝大部分方法都会被编译成本地代码。另外，可以使用 `-XX:CounterHalfLifeTime` 参数设置半衰周期的时间，单位是秒。

三、静态编译、动态编译和即时编译器到底是什么关系

之前一直分不清楚，这几个概念之间的关系

1. 动态编译（dynamic compilation）：在运行时进行编译，如 JIT 编译（just-in-time compilation），自适应动态编译（adaptive dynamic compilation）。
2. 静态编译（static compilation）：运行前已经编译好了，如 AOT（ahead-of-time compilation）。

JIT 编译（just-in-time compilation）狭义来说是当某段代码即将第一次被执行时进行编译，因而叫“即时编译”。**JIT 编译是动态编译的一种特例。JIT 编译一词后来被泛化，时常与动态编译等价；但要注意广义与狭义的 JIT 编译所指的区别。**

自适应动态编译 (adaptive dynamic compilation) 也是一种动态编译，但它通常执行的时机比 JIT 编译迟，**先让程序“以某种式”先运行起来，收集一些信息之后再做动态编译**。这样的编译可以更加优化。

四、JIT 编译器分类

在 HotSpot 虚拟机中，内置了两种 JIT，分别为 C1 编译器和 C2 编译器，这两个编译器的编译过程是不一样的。

4.1 C1 编译器

C1 编译器是一个简单快速的编译器，主要的关注点在于局部性的优化，适用于执行时间较短或对启动性能有要求的程序，也称为 Client Compiler，例如，GUI 应用对界面启动速度就有一定要求。

4.2 C2 编译器

C2 编译器是为长期运行的服务器端应用程序做性能调优的编译器，适用于执行时间较长或对峰值性能有要求的程序，也称为 Server Compiler，例如，服务器上长期运行的 Java 应用对稳定运行就有一定的要求。

4.3 混合模式

在 Java7 之前，需要根据程序的特性来选择对应的 JIT，虚拟机默认采用解释器和其中一个编译器配合工作。

HotSpot 虚拟机会根据自身版本与计算机的硬件性能自动选择运行模式，用户也可以使用 -client 和 -server 参数强制指定虚拟机运行在 Client 模式或者 Server 模式。这种配合使用的方式称为**“混合模式”（Mixed Mode）**，用户可以使用参数 -Xint 强制虚拟机运行于**“解释模式”（Interpreted Mode）**，这时候编译器完全不介入工作。另外，使用 -Xcomp 强制虚拟机运行于**“编译模式”（Compiled Mode）**，这时候将优先采

用编译方式执行，但是解释器仍然要在编译无法进行的情况下接入执行过程。通过虚拟机 -version 命令可以查看当前默认的运行模式。

```
C:\Users\Administrator>java -version  
  
java version "1.8.0_144"  
  
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)  
  
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)  
  
``shell  
  
C:\Users\Administrator>java -version  
  
java version "1.8.0_144"  
  
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)  
  
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
```

4.4 分层编译

Java7 引入了分层编译，这种方式综合了 C1 的启动性能优势和 C2 的峰值性能优势，我们也可以通过参数 -client 或者 -server 强制指定虚拟机的即时编译模式。

分层编译将 JVM 的执行状态分为了 5 个层次：

- 第 0 层：程序解释执行，默认开启性能监控功能（Profiling），如果不开启，可触发第二层编译；
- 第 1 层：可称为 C1 编译，将字节码编译为本地代码，进行简单、可靠的优化，不开启 Profiling；
- 第 2 层：也称为 C1 编译，开启 Profiling，仅执行带方法调用次数和循环回边执行次数 profiling 的 C1 编译；
- 第 3 层：也称为 C1 编译，执行所有带 Profiling 的 C1 编译；

- 第 4 层：可称为 C2 编译，也是将字节码编译为本地代码，但是会启用一些编译耗时较长的优化，甚至会根据性能监控信息进行一些不可靠的激进优化。

对于 C1 的三种状态，按执行效率从高至低：第 1 层、第 2 层、第 3 层。

通常情况下，C2 的执行效率比 C1 高出 30% 以上。

在 Java8 中，默认开启分层编译，-client 和 -server 的设置已经是无效的了。如果只想开启 C2，可以关闭分层编译（-XX:-TieredCompilation），如果只想用 C1，可以在打开分层编译的同时，使用参数：-XX:TieredStopAtLevel=1。

你可以通过 `java -version` 命令行可以直接查看到当前系统使用的编译模式：

```
C:\Users\Administrator>java -Xint -version  
  
java version "1.8.0_45"  
  
Java(TM) SE Runtime Environment (build 1.8.0_45-b14)  
  
Java HotSpot(TM) 64-Bit Server VM (build 25.45-b02, interpreted  
mode)  
C:\Users\Administrator>java -Xcomp -version  
  
java version "1.8.0_45"  
  
Java(TM) SE Runtime Environment (build 1.8.0_45-b14)  
  
Java HotSpot(TM) 64-Bit Server VM (build 25.45-b02, compiled mode)
```

五、JIT 编译优化技术

JIT 编译运用了一些经典的编译优化技术来实现代码的优化，即通过一些例行检查优化，可以智能地编译出运行时的最优性能代码。主要有两种：方法内联、逃逸分析。

5.1 逃逸分析

其中，逃逸分析我们已经在终于搞懂了！字符串拼接的各种姿势以及底层的小知识中讲过了，这次主要说下另一种，方法内联

5.2 方法内联

5.2.1 方法内联是什么

调用一个方法通常要经历压栈和出栈。调用方法是将程序执行顺序转移到存储该方法的内存地址，将方法的内容执行完后，再返回到执行该方法前的位置。

这种执行操作要求在执行前保护现场并记忆执行的地址，执行后要恢复现场，并按原来保存的地址继续执行。因此，方法调用会产生一定的时间和空间方面的开销（其实可以理解为一种资源开销小于上下文切换的类似动作）。

那么对于那些方法体代码不是很大，又频繁调用的方法来说，这个时间和空间的消耗会很大。

方法内联的优化行为就是把目标方法的代码复制到发起调用的方法之中，避免发生真实的方法调用。

5.2.2 方法内联优化的触发条件

JVM 会自动识别热点方法，并对它们使用方法内联进行优化。我们可以通过-XX:CompileThreshold 来设置热点方法的阈值。但要强调一点，**热点方法不一定会被 JVM 做内联优化，如果这个方法体太大了，JVM 将不执行内联操作**。而方法体的大小阈值，我们也可以通过参数设置来优化：

经常执行的方法，默认情况下，方法体大小小于 325 字节的都会进行内联，我们可以通过-XX:MaxFreqInlineSize=N 来设置大小值；

不是经常执行的方法，默认情况下，方法大小小于 35 字节才会进行内联，我们也可以通过-XX:MaxInlineSize=N 来重置大小值。

之后我们就可以通过配置 JVM 参数来查看到方法被内联的情况：

```
// 在控制台打印编译过程信息
```

```
-XX:+PrintCompilation  
  
// 解锁对 JVM 进行诊断的选项参数。默认是关闭的，开启后支持一些特定参数对 JVM 进行诊断  
  
-XX:+UnlockDiagnosticVMOptions  
  
// 将内联方法打印出来  
  
-XX:+PrintInlining
```

5.2.3 提高方法内联的方法

热点方法的优化可以有效提高系统性能，一般我们可以通过以下几种方式来提高方法内联：

1. 通过设置 JVM 参数来减小热点阈值或增加方法体阈值，以便更多的方法可以进行内联，但这种方法意味着需要占用更多地内存；
2. 在编程中，避免在一个方法中写大量代码，习惯使用小方法体；
3. 尽量使用 final、private、static 关键字修饰方法，编码方法因为继承，会需要额外的类型检查。

5.2.4 总结

一个方法中的内容越少，当该方法经常被执行时，则容易进行方法内联，从而优化性能。

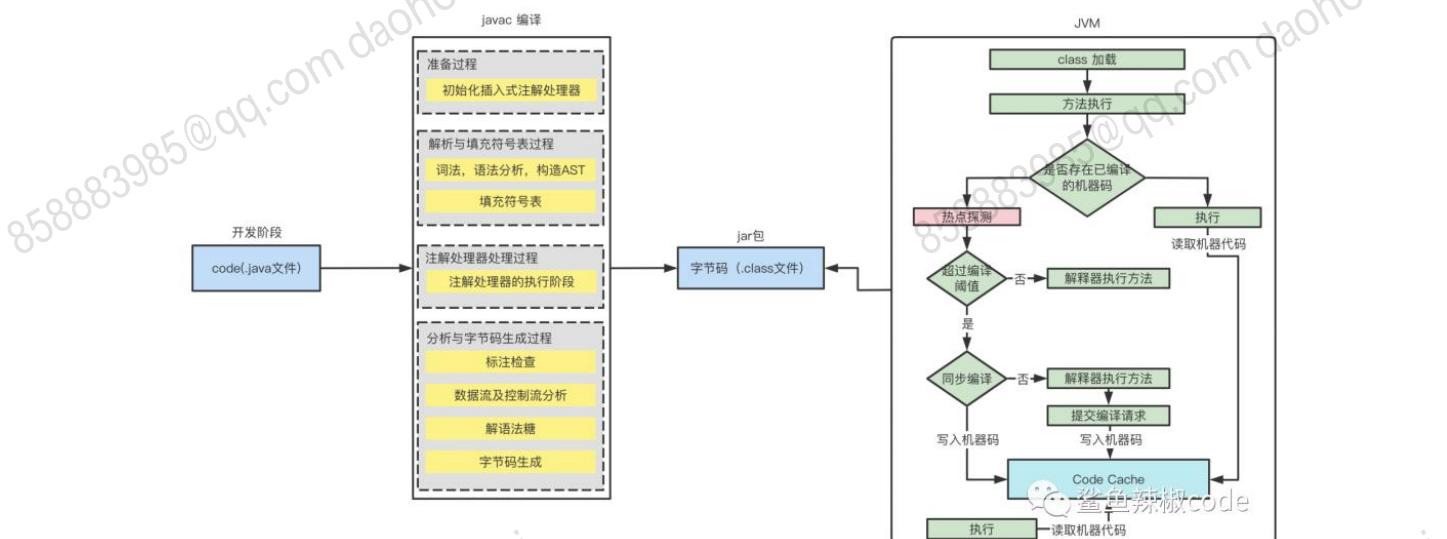
5.3 退优化

在前面分层编译说到，**不同的层级通过收集不同数量的信息，可以进行不同层级的优化，虚拟机执行方法或循环的次数越多，它知道的代码的额外信息越多，代码优化效率越高。**

假设虚拟机执行了很多次 obj.equals()发现 obj 的类型都是 String，那么虚拟机可以乐观地认为 obj 就是 String 类型，继而直接调用 String.equals，省去了查询 obj 虚函数

表的开销。但是如果后续变量 obj 接收到其他类型的对象，**虚拟机也必须有处理这种少数情况的能力，这种处理少数情况的行为即退优化。**

除了上述这个例子外，编译器优化还会做很多乐观的假设，**它广泛使用 fast/slow 惯例，乐观地认为大部分情况程序都是走快速路径 fast，而只有极少数情况走慢速路径 slow。**当极少数情况发生时，虚拟机将执行退优化，使用慢速路径作为后备方案。退优化可以认为是栈上替换的逆操作。



查看及分析即时编译结果

-XX:+PrintCompilation
-XX:+UnlockDiagnosticVMOptions
-XX:+PrintInlining
-XX:+PrintAssembly

```
public class Test {
```

```
    public static final int NUM = 15000;
```

```
    public static int doubleValue(int i) {
```

```
        for (int j = 0; j < 100000; j++) ;//空循环用于演示即时编译优化
```

```
        return i * 2;

    }

public static long calcSum() {

    long sum = 0;

    for (int i = 1; i <= 100; i++) {

        sum += doubleValue(i);

    }

    return sum;

}

public static void main(String[] args) {

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

        calcSum();

    }

}
```

<https://github.com/daohonglei/javaStereotypedWriting>

<https://gitee.com/daohonglei/javaStereotypedWriting>