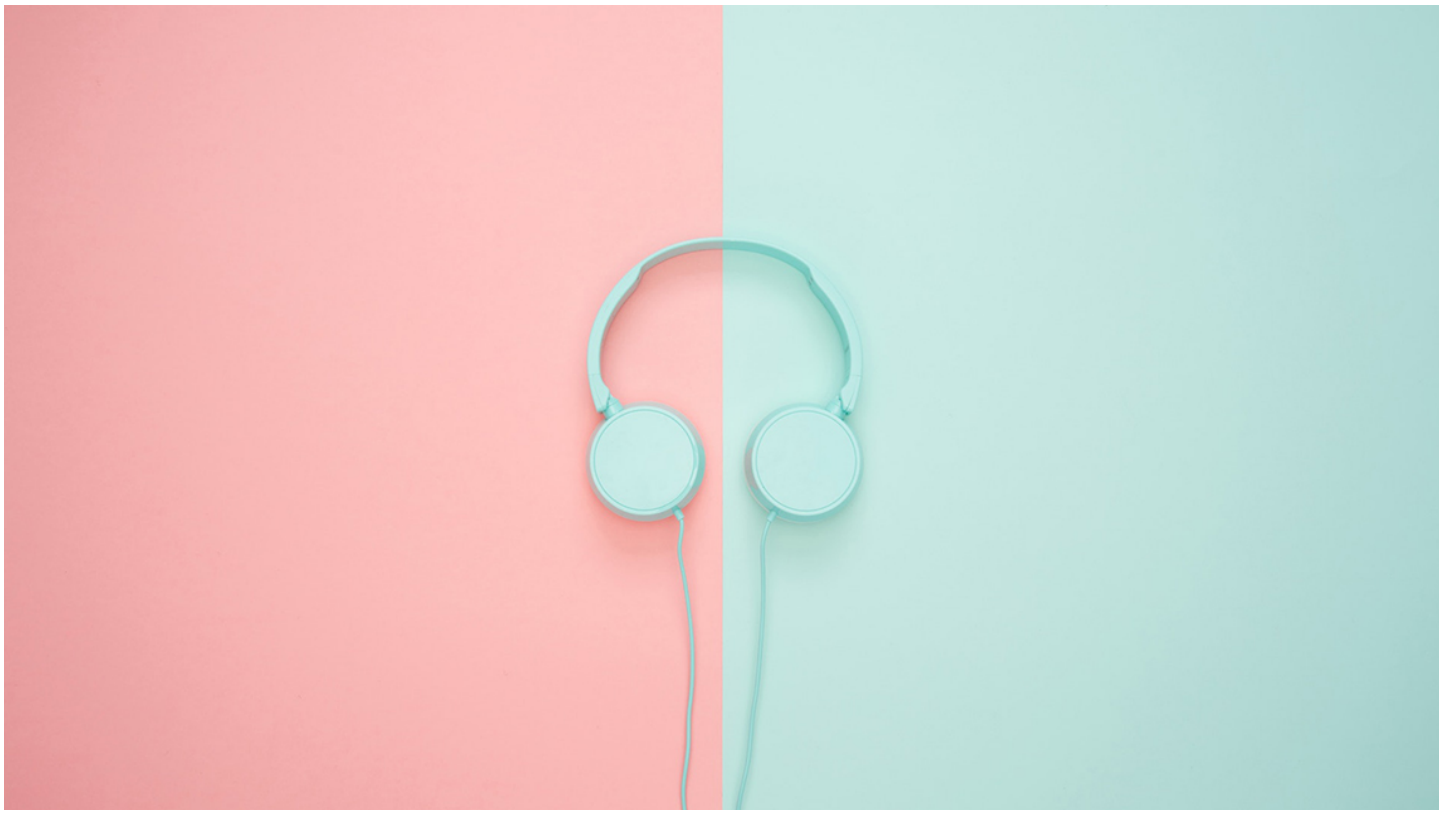
转载请注明链接: https://blog.csdn.net/feather_wch/article/details/52254111

Java对象的内存布局

版本号:2018/09/21-1(1:25)



对象

1、Java中创建对象有几种方式?(5)

> 1. new
> 2. 反射机制
> 3. Object.clone
> 4. 反序列化
> 5. Unsafe.allocateInstance()

2、创建对象的多种方法的特点?

> 1. Object.clone()、反序列化: 直接复制已有的数据, 来初始化对象的实例字段
> 2. new、反射机制: 通过调用构造器来初始化实例字段
> 3. Unsafe.allocateInstance(): 创建对象，但不会初始化实例字段

3、Unsafe.allocateInstance()有什么用?

> 1. 能突破限制创建实例
> 2. 通过allocateInstance()方法，可以创建一个类的实例，但是不需要调用其的构造函数、初始化代码、各种JVM安全检查以及底层的内容。
> 3. 即使构造函数是私有，也可以通过这个方法创建它的实例。

4、new语句生成的字节码

> 1. 首先是new指令，用于请求内存。
> 2. 其次是invokespecial指令，用于执行构造器。

```
People people = new People();
```

```
        0: new           #2                  // class People
        3: dup
        4: invokespecial #3                  // Method People."<init>":()V
        7: astore_1
        8: return
```

5、Java对象构造器的约束

> 1-一个类没有定义任何构造器，Java 编译器会自动添加一个无参数的构造器。

```
public class People {
    // 默认具有无参构造器
}
```

> 2-如果父类存在无参数构造器，子类构造器会隐式调用父类无参构造器。

```
public class People {
}
public class Student extends People{
    public Student(){
        //默认隐式调用父类无参数构造器
    }
    public Student(int age){
        //默认隐式调用父类无参数构造器
    }
}
```

> 3-如果父类没有无参数构造，子类构造器需要显式地调用父类带参数构造器。

```
public class People {
    public People(String name){
    }
}
// 错误形式
public class Student extends People{
    public Student(){
        // 报错!
    }
    public Student(String name){
        // 报错!
    }
}
// 正确形式
public class Student extends People{
    public Student(){
        super("");
    }
    public Student(String name){
        super(name);
    }
}
```

6、子类显式地调用父类构造器的方法?

> 1. 一是直接使用“super”关键字调用父类构造器
> 2. 二是使用“this”关键字调用同一个类中的其他构造器,间接调用父类构造器。

7、调用父类构造器必须作为构造器的第一条语句?

> 1. 以便优先初始化继承而来的父类字段。
> 2. 不然编译器会报错
> 3. 这个限制可以通过字节码注入绕开

```
public Student(){
    // 会报错
    System.out.println("我先执行");
    super("");
}
```

8、子类会层层调用父类的构造器?

> 1. 父类的构造器会作为子类构造器的第一个语句执行
> 2. 子类的实例对象会层层调用构造器，直到Object类。

9、父类的private字段，子类是无法继承的，因此并没有给这些字段分配空间?被子类的实例字段隐藏的父类实例字段呢?

> 错误!
>
> 1. 层层调用父类构造器，一定会给所有父类中的实例字段分配内存空间。
> 2. 子类的同名实例字段会隐藏掉父类的同名实例字段(父类该字段无法被子类对象访问)，但依然是分配了内存空间的。

对象的组成

10、对象有几个组成部分?

> 1. 对象头-Header
> 2. 对象实例-Instance Data
> 3. 对象填充-Padding

12、Java对象的对象头(Object Header)是什么?由哪几部分组成?

> 1. 标记字段---存储对象自身的运行时数据：
>   1. HashCode
>   2. GC分代年龄
>   3. 锁状态标志
>   4. 线程持有锁
>   5. 偏向线程ID
>   6. 偏向时间戳
> 2. 类型指针
>   1. 对象指向它的类元数据的指针，也就是该对象指向它的类。

13、对象实例存储的是什么?

> 1. 对象真正存储的有效信息
> 2. 定义的各种字段(父类、子类)

14、对齐填充的作用?

1. JVM的内存管理要求对象的 起始地址 都必须是 8字节的整数倍

字段在内存中的分布

1、对象头的大小是多少?

1. 64位的JVM中，标记字段占64位，类型指针占64位。(64+64)/8=16个字节。
2. 也就是说，每一个Java对象在内存中的额外开销就是16个字节。
3. 例如Integer类，仅有一个int类型的私有字段，占4个字节。所以一个Integer对象的额外内存开销至少是 400%。
4. 这也是Java要引入基本数据类型的原因之一。

2、Java为什么要引入基本数据类型?

1. Integer等包装类会有大量的额外内存开销。
2. 原始数据类型数组，数据在内存上是连续的。但是对象数组中的对象分散在堆中，无法利用CPU的缓存机制。
3. 原始数据类型数组，能直接在内存地址中取出数值。但是Integer等对象数组，需要先找到目标地址，才能读取数据。

压缩指针

3、压缩指针是什么?

1. 为了减少对象的内存使用量，64位JVM引入了 压缩指针
2. JVM参数是: `-XX:+UseCompressedOops` ，默认开启。
3. 将堆中原本64位的Java对象指针压缩成32位。
4. 对象头中的类型指针也会被压缩成32位，使得对象头的大小从16字节降至12字节。
5. 此外压缩指针还可以作用于引用类型的字段和引用类型的数组。

4、压缩指针的原理?

1. 场景: 停车场停放房车，每个房车会占据2个车位。
2. 原本的内存寻址, 使用的是车位号，值为6的指针代表第6个车位。最大的车位是2^{32}(4GB个车位)，最大的车号是$2$31(2GB辆车)。
3. 采用压缩指针，使用的是车号，值为6的指针代表第6个车。其具体的车位是 `10`、`11` 。这样最大的车位是2^{33}(8GB个车位)，最大的车号是$2$32(4GB辆车)。
4. 假设一个房车占8个车位(8个字节)，最大的车号依然是2^{32}(4GB辆车)。最大的车位(内存地址)是$2$35(32GB个车位)
5. 此外具有的前提是，每辆车必须从偶数号车位开始停车。(内存对齐)

5、什么是内存对齐?

> 1. Java中对象的起始地址需要是8的整数倍
> 2. 能提高系统性能(例如如果放置在奇数内存位置上，CPU读取出数据需要读取两次，放到偶数内存位置上，CPU只需要读取一次。)
> 3. JVM是的参数 `-XX:ObjectAlignmentInBytes`，默认值为8。
> 4. 但是通过内存对齐选项进一步提升寻址范围，可能会导致压缩指针没有达到原本节省空间的效果。(一些数据浪费的填空空间过大，整体得不偿失)

6、如果一个对象用不到8的整数倍字节，该怎么办?(对象填充)

> 1. 那些空白的部分空间会被浪费
> 2. 这些浪费掉的空间称之为对象间的填充（padding）

7、JVM中压缩指针可以寻址到多少个字节?

> 1. 在默认情况下，Java 虚拟机中的 32 位压缩指针可以寻址到 2 的 35 次方个字节。
> 2. 也就是 32GB 的地址空间（超过 32GB 则会关闭压缩指针）。
> 3. 在对压缩指针解引用时，将其左移 3 位，再加上一个固定偏移量，便可以得到能够寻址32GB地址空间的伪64位指针。

8、关闭了压缩指针，Java虚拟机还是会进行内存对齐

9、内存对齐仅仅存在于对象与对象之间?

> 错误!
>
> 1. 也存在于对象中的字段之间。
> 2. 比如JVM要求 long 字段、double 字段，以及非压缩指针状态下的引用字段的地址为8的倍数。

10、字段为什么要进行内存对齐?

> 1. 是为了让字段只出现在CPU的同一个缓存行中。
> 2. 如果字段不是对齐的，那么就有可能出现跨缓存行的字段。
> 3. 也就是说，该字段的读取可能需要替换两个缓存行，而该字段的存储也会同时污染两个缓存行。
> 4. 这两种情况都会影响程序的执行效率。

字段重排列

11、什么是字段重排列?

> 1. JVM会重新分配字段的先后顺序
> 2. 目的是进行内存对齐

12、JVM有几种排列方法? 对应的虚拟机选项是什么?

1. 三种排列方法
2. 虚拟机选项 -XX:FieldsAllocationStyle(默认值是1)

13、字段重排列必然遵守的两个规则

1. 如果一个字段占据 C 个字节，那么该字段的偏移量需要对齐至 NC。这里偏移量指的是字 段地址与对象的起始地址差值。
    1. 比如字段占据8个字节，那么偏移量就要是0、8、16、24...
    2. long 类为例，它仅有一个long类型的实例字段。在使用了压缩指针的64位JVM中，尽管对象头的大小为12个字节，该 long 类型字段的偏移量也只能是 16，会浪费12~16这4个字节。
2. 子类所继承字段的偏移量，需要与父类对应字段的偏移量保持一致。
    1. JVM还会对齐子类字段的起始位置。对于使用了压缩指针的64位虚拟机，子类第一个字段需要对齐至 4N；而对于关闭了压缩指针的 64 位虚拟机，子类第一个字段 则需要对齐至 8N。

14、如下的A类和B类，B类中的字段分布是怎样的(启用压缩指针)?

```
class A {
  long l;
  int i;
}

class B extends A {
  long l;
  int i;
}
```

> 启用压缩指针时，B 类的字段分布:

```
# 启用压缩指针时，B 类的字段分布
B object internals:
 OFFSET  SIZE   TYPE DESCRIPTION
 0     4        (object header)
 4     4        (object header)
 8     4        (object header)
 12     4    int A.i                                          0
 16     8   long A.l                                          0
 24     8   long B.l                                          0
 32     4    int B.i                                          0
 36     4        (loss due to the next object alignment)
```

1. 因为对象需要对齐至8N，因此末尾会有4字节的空间浪费

15、不启用压缩指针时，B类的字段又是如何分布的?

```
# 关闭压缩指针时，B 类的字段分布
B object internals:
  OFFSET  SIZE   TYPE DESCRIPTION
  0     4        (object header)
  4     4        (object header)
  8     4        (object header)
  12     4        (object header)
  16     8   long A.l
  24     4    int A.i
  28     4        (alignment/padding gap)
  32     8   long B.l
  40     4    int B.i
  44     4        (loss due to the next object alignment)
```

> 1. 不启用压缩指针，会导致字段会以8N去对齐。
> 2. 但是是否可以让B.i放置到OffSet为28的空间去，从而节省后面的8个字节呢?是可以的。之所以HotSpot采用现在这种方式，是因为代码年久失修，按道理是需要这样进行优化的。(历史遗留问题)

16、自动内存管理系统为什么要求对象的大小必须是8字节的整数倍？即内存对齐的根本原因在于?

> 1. 在某些体系架构上，内存不对齐，在内存读写时会报错。
> 2. 在X86_64上，是为了让字段也能对齐，这样就不会出现字段横跨两个缓存行的情况
> 3. 另一个原因是对象地址后三位一直是0，JVM利用这个特性来实现压缩指针，也可以用这三位来记录一些额外信息

17、 @Contended注释是什么？有什么用?

> 1. Java 8 引入的新注释
> 2. 用来解决对象字段之间的虚共享问题(false sharing)
> 3. 这个注释也会影响到字段的排列。

18、虚共享是什么?

> 1. 假设两个线程分别访问同一对象中不同的 volatile 字段，逻辑上它们并没 有共享内容，因此不需要同步。
> 2. 然而，如果这两个字段恰好在同一个缓存行中，那么对这些字段的写操作会导致缓存行的写回， 也就造成了实质上的共享。
> 3. Java 虚拟机会让不同的 @Contended 字段处于独立的缓存行中，因此会看到大量的空间被浪费。
> 4. 具体的分布算法属于实现细节，随着Java版本的变动也比较大。

19、Contended 字段的内存布局如何查看(用什么工具)?

> 1. 使用JOL
> 2. 注意使用虚拟机选项 `-XX:-RestrictContended`。

> 3. 如果在Java9以上版本，在使用javac时 需要添加
>
> ```
> --add-exports java.base/jdk.internal.vm.annotation=ALL-UNNAME
> ```

20、什么是缓存行?

21、64位JVM中对象间需要对其到多少个字节？32位JVM呢?

> 1. 64位: 8字节
> 2. 32为: 4字节

补充题

1、为什么一个子类即使无法访问父类的私有实例字段，或者子类实例字段隐藏了父类的同名实例字段，子类的实例还是会为这些父类实例字段分配内存呢?

2、HotSpot的对象头的源码哪里看?

> 1. HotSpot的对象头由mark word(标记字段)和类型指针组成。
> 2. 对象头的源代码在OpenJDK源代码目录下，src/hotspot/share/oops/oop.hpp里的class oopDesc。
> 3. 数组对象头的源代码在同目录下的arrayOop.hpp里的class arrayOopDesc

3、String字面量(Literal)存放在哪里?

> 1. String Literal指向的对象存放在JVM的String pool里

4、内存屏障是什么?

> 1. 底层的内存屏障，比如说mfence，lock指令，是用来防止指令重排布的。
> 2. Java里的内存屏障还会限制即时编译器对内存访问的重排序。

问题汇总

1. Java中创建对象有几种方式？(5)
2. 每种创建对象方法的特点?
3. 哪种创建方法只会创建对象，不会初始化实例字段?
4. 哪种创建方法会通过调用构造器来初始化实例字段?
5. 哪种创建方法会通过直接复制已有的数据,来初始化对象的实例字段?
6. Unsafe.allocateInstance()有什么用?
7. new语句生成的字节码中new指令和invokespecial指令的作用?new指令就是用来创建对象并且执行构造器的?
8. “一个类没有定义任何构造器，Java 编译器会自动添加一个无参数的构造器”这句话是否正确?

9. “如果父类没有无参数构造器，子类的构造器必须要显式地调用父类带参数构造器”这句话是否正确?
10. 子类如何显式地调用父类构造器?
11. 需要显式的调用父类构造器的场景中，调用父类构造器必须在子类构造器中作为第一条执行的语句?
12. 能否避免这个限制?
13. 子类会层层调用父类、父类的父类的构造器?
14. 父类的private字段，子类是无法继承的，因此并没有给这些字段分配空间?被子类的实例字段隐藏的父类实例字段是否会分配空间?
15. 对象有几个组成部分?
16. Java对象的对象头(Object Header)是什么?由哪几部分组成?
17. 对象实例存储的是什么?
18. 对齐填充的作用?
19. 对象头的大小是多少?
20. Java为什么要引入基本数据类型?
21. 压缩指针是什么?
22. 压缩指针的原理?
23. 什么是内存对齐?
24. 如果一个对象用不到8的整数倍字节，该怎么办?
25. 32位JVM中压缩指针可以寻址到多少个字节?
26. 关闭了压缩指针，Java虚拟机还是会进行内存对齐?
27. 内存对齐仅仅存在于对象与对象之间?
28. 字段为什么要进行内存对齐?
29. 什么是字段重排列?
30. JVM有几种排列方法? 对应的虚拟机选项是什么?
31. 字段重排列必然遵守的两个规则
32. 如下的A类和B类，B类中的字段分布是怎样的(启用压缩指针)?

```
class A {
 long l;
 int i;
}

class B extends A {
 long l;
 int i;
}
```

33. 不启用压缩指针时，B类的字段又是如何分布的?
34. 自动内存管理系统为什么要求对象的大小必须是8字节的整数倍？即内存对齐的根本原因在于?
35. @Contended注释是什么？有什么用?
36. 虚共享是什么?
37. Contended 字段的内存布局如何查看(用什么工具)?
38. 什么是缓存行?

39. 64位JVM中对象间需要对其到多少个字节？32位JVM呢？
40. 为什么一个子类即使无法访问父类的私有实例字段，或者子类实例字段隐藏了父类的同名实例字段，子类的实例还是会为这些父类实例字段分配内存呢？
41. HotSpot的对象头的源码哪里看？
42. String字面量(Literal)存放在哪里？
43. 内存屏障是什么？

参考资料

1. 10|Java对象的内存布局
2. jvm-volatile之缓存行