Detailed Explanation of Java Memory Areas (Key Points)

Guide Java JVM About 7473 words About 25 minutes

This is a small advertisement that may be useful to you

- **Interview special edition** : Friends who are preparing for Java interviews can consider the interview special edition: **"Java Interview Guide"** (very high quality, specially designed for interviews, best used with JavaGuide).
- **Knowledge Planet** : Technical Column/One-on-one Questions/Resume Modification/Job Hunting Guide/Interview Check-in/Irregular Benefits, welcome to join **the JavaGuide official Knowledge Planet** .

> Unless otherwise specified, all instructions refer to HotSpot virtual machines.
> This article summarizes and supplements the article "In-depth Understanding of Java Virtual Machine: JVM Advanced Features and Best Practices".
> Common interview questions:
>
> - Introducing Java memory areas (runtime data areas)
> - The process of creating a Java object (five steps, it is recommended to be able to write it out silently and know what the virtual machine does at each step)
> - Two ways to access and locate objects (handle and direct pointer)

Preface

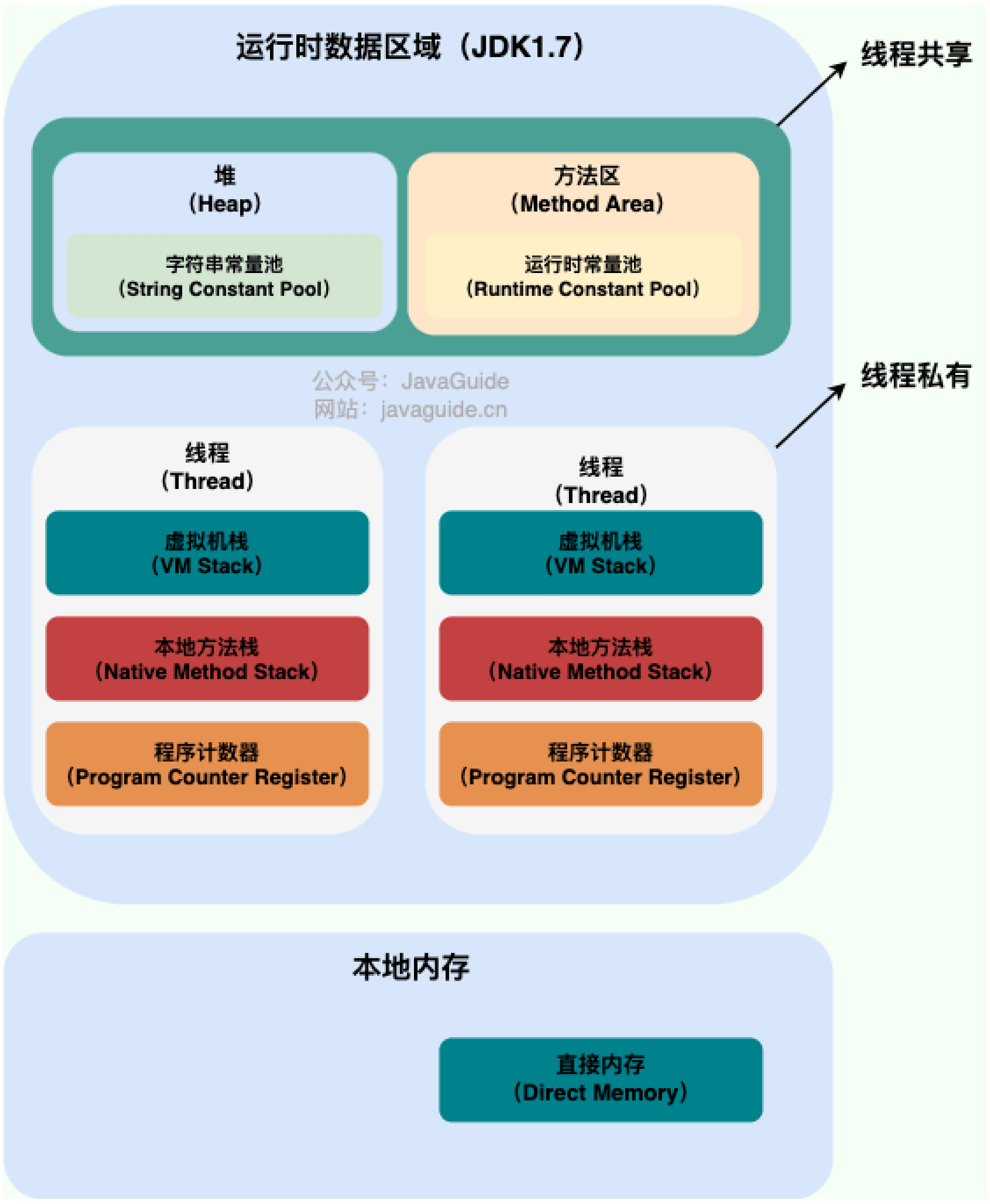
For Java programmers, the virtual machine's automatic memory management mechanism eliminates the need to write corresponding delete/free operations for every new operation, as C/C++ programmers do. This reduces the risk of memory leaks and overflows. Because Java programmers hand over memory control to the Java virtual machine, troubleshooting memory leaks and overflows can be a daunting task if you don't understand how the virtual machine uses memory.

Runtime data area

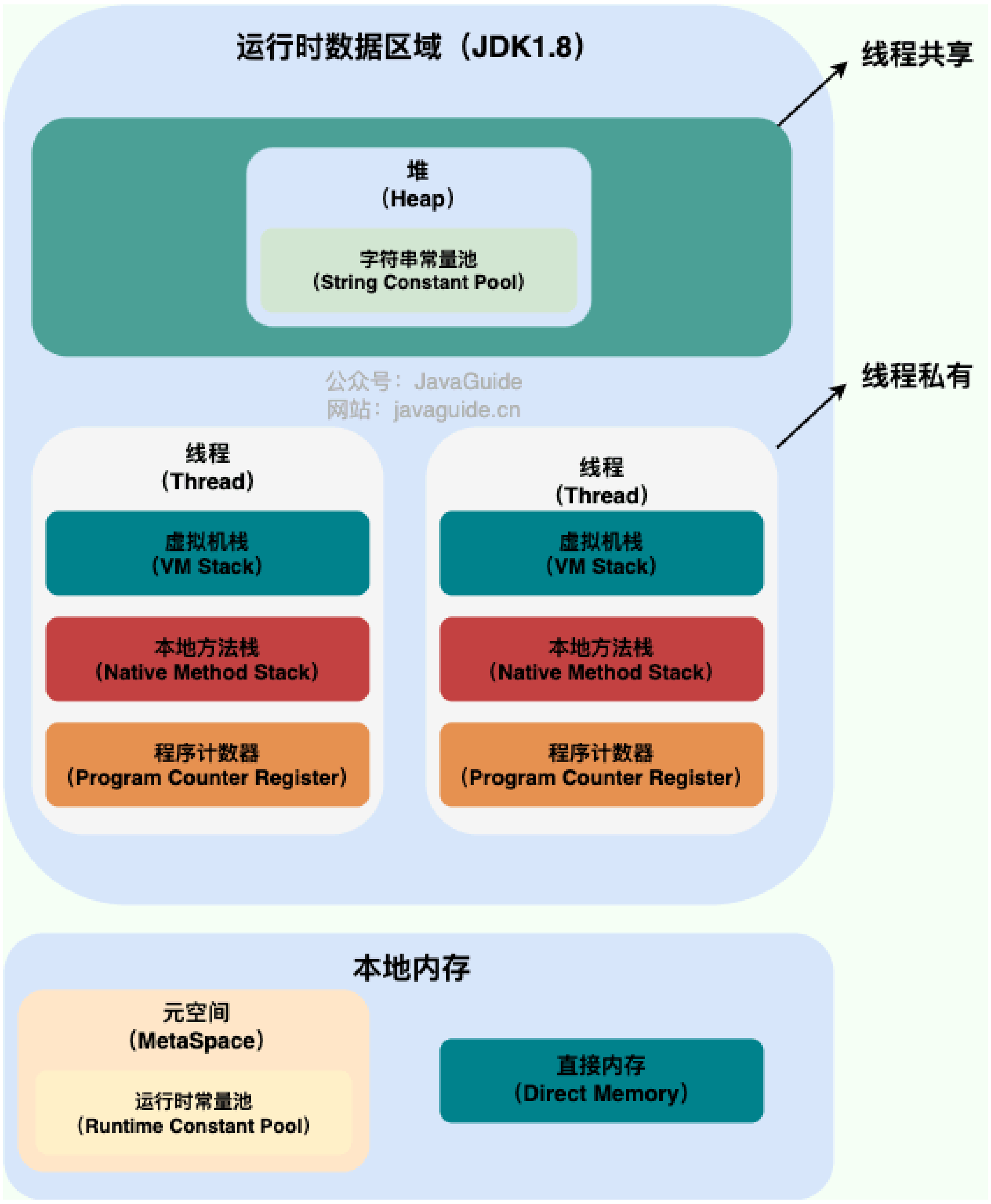
When executing a Java program, the Java virtual machine divides the memory it manages into several different data areas.

JDK 1.8 is slightly different from previous versions. Here we use JDK 1.7 and JDK 1.8 as examples.

JDK 1.7 :


运行时数据区域（JDK1.7）
线程共享
堆
(Heap)
字符串常量池
(String Constant Pool)
方法区
(Method Area)
运行时常量池
(Runtime Constant Pool)
公众号：JavaGuide
网站：javaguide.cn
线程私有
线程
(Thread)
虚拟机栈
(VM Stack)
本地方法栈
(Native Method Stack)
程序计数器
(Program Counter Register)
线程
(Thread)
虚拟机栈
(VM Stack)
本地方法栈
(Native Method Stack)
程序计数器
(Program Counter Register)
本地内存
直接内存
(Direct Memory)

JDK 1.8 :



Thread-private:

- Program Counter
- Virtual Machine Stack
- Native Method Stack

Threads shared:

- heap
- Method Area
- Direct memory (part of the non-runtime data area)

The Java Virtual Machine specification is quite liberal with regard to runtime data areas. For example, the heap can be contiguous or discontiguous. The heap size can be fixed or expandable at runtime. Virtual machine implementers can use any garbage collection algorithm to manage the heap, or even opt out of garbage collection altogether.

Program Counter

The program counter is a small memory location that can be thought of as an indicator of the line number of the bytecode being executed by the current thread. The bytecode interpreter selects the next bytecode instruction to execute by changing the value of this counter. Functions such as branching, looping, jumping, exception handling, and thread recovery all rely on this counter.

In addition, in order to restore to the correct execution position after thread switching, each thread needs to have an independent program counter. The counters of each thread do not affect each other and are stored independently. We call this type of memory area "thread-private" memory.

From the above introduction, we know that the program counter has two main functions:

- The bytecode interpreter reads instructions in sequence by changing the program counter, thereby implementing code flow control, such as sequential execution, selection, looping, and exception handling.
- In the case of multiple threads, the program counter is used to record the location of the current thread execution, so that when the thread is switched back, it can know where the thread last ran.

⚠️ Note: The program counter is the only `OutOfMemoryError` memory area that does not appear. Its life cycle is created when the thread is created and dies when the thread ends.

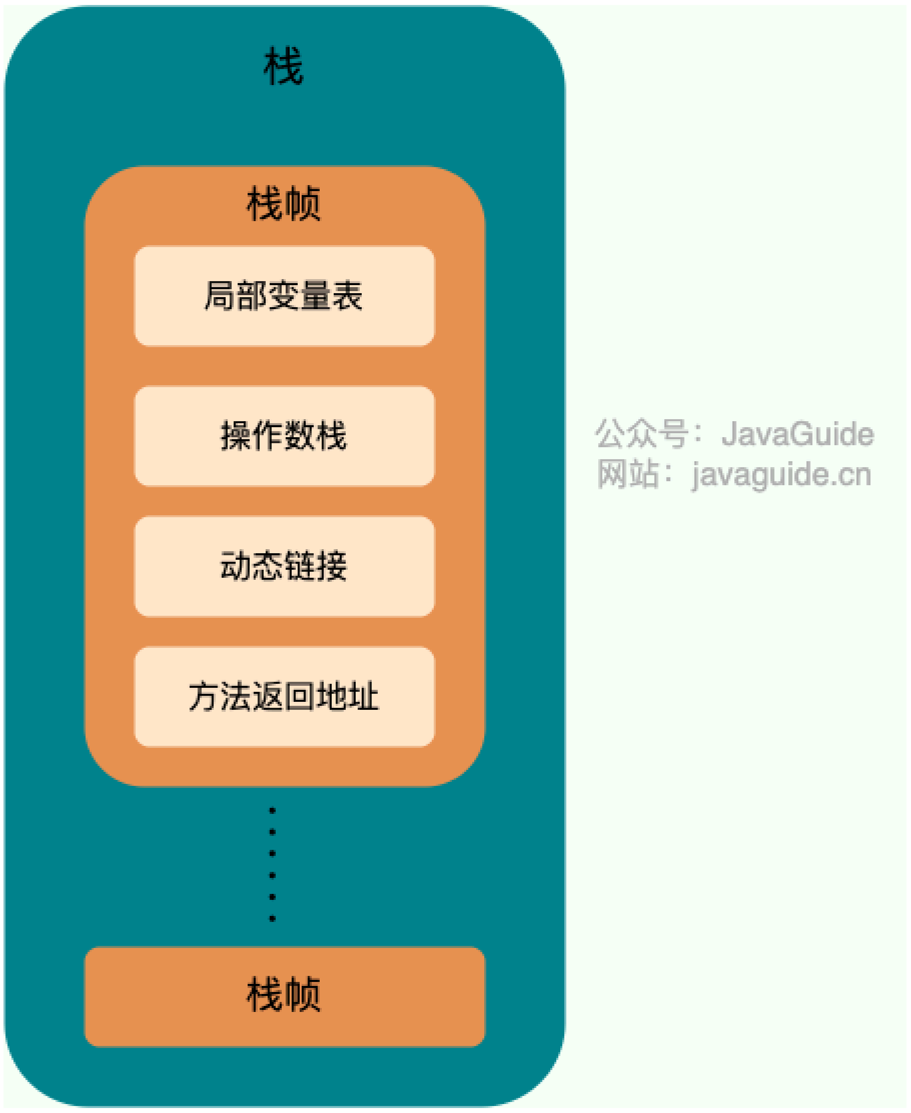
Java Virtual Machine Stack

Like the program counter, the Java virtual machine stack (hereinafter referred to as the stack) is also thread-private. Its life cycle is the same as that of the thread. It is created when the thread is created and dies when the thread dies.

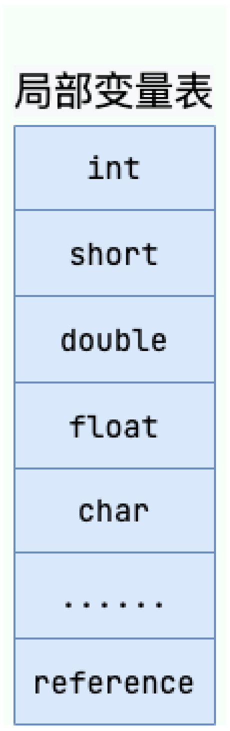
The stack is definitely a core part of the JVM runtime data area. Except for some native method calls that are implemented through the local method stack (mentioned later) other Java method calls are implemented through the stack (which also needs to be coordinated with other runtime data areas such as the program counter).

The data of method calls needs to be passed through the stack. For each method call, a corresponding stack frame will be pushed into the stack. After each method call, a stack frame will be popped out.

The stack is composed of stack frames, each of which contains: a local variable table, an operand stack, dynamic links, and method return addresses. Similar to the stack in data structures, both are first-in-last-out data structures, supporting only two operations: pop and push.

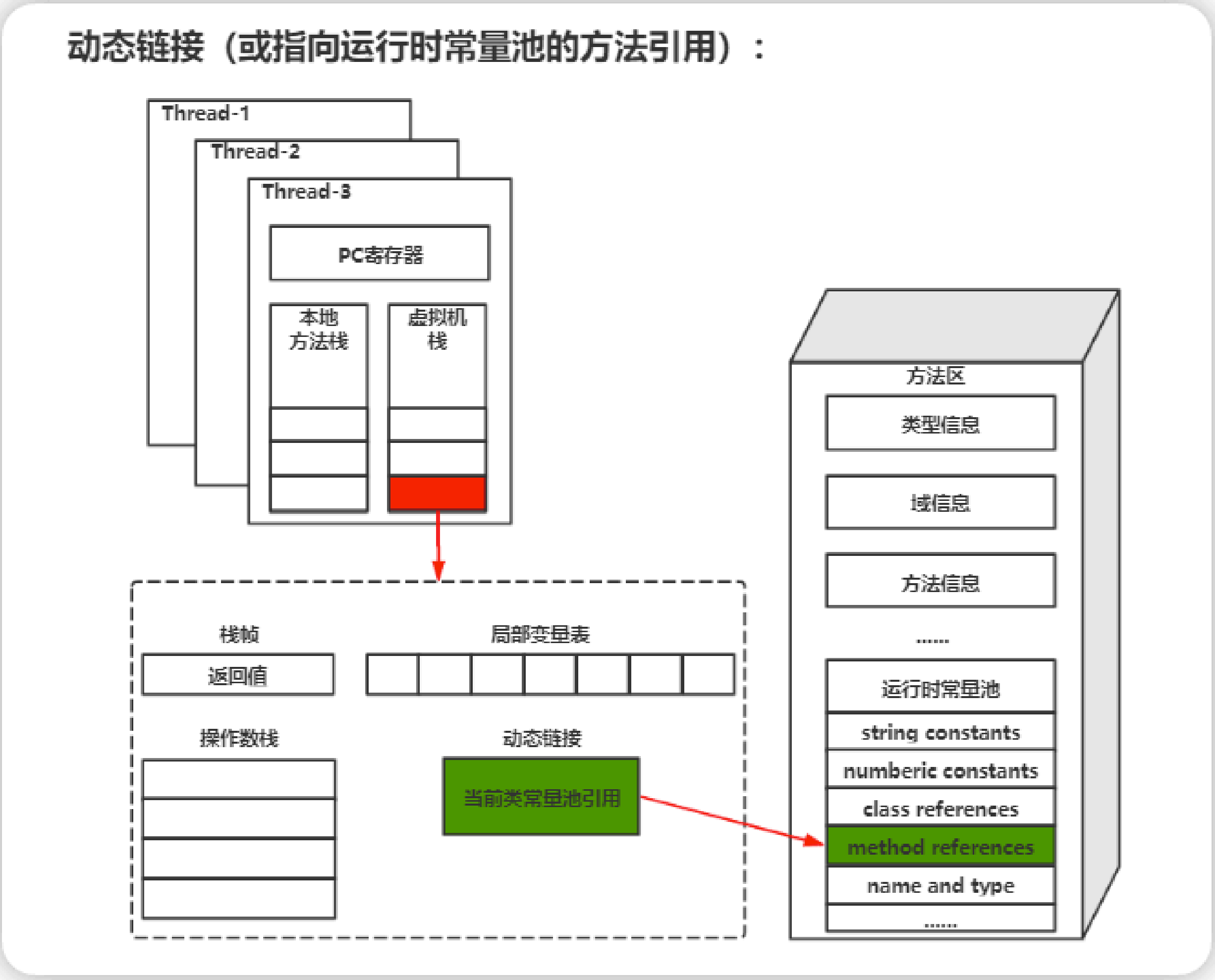


The local variable table mainly stores various data types known at compile time (boolean, byte, char, short, int, float, long, double) and object references (reference type, which is different from the object itself, may be a reference pointer pointing to the starting address of the object, or it may point to a handle representing the object or other locations related to this object).



The operand stack is primarily used as a transit point for method calls, storing intermediate calculation results generated during method execution. Temporary variables generated during the calculation process are also placed on the operand stack.

Dynamic linking primarily serves scenarios where a method needs to call other methods. The constant pool of a class file stores a large number of symbolic references, such as symbolic references to method references. When a method calls another method, the symbolic references to the method in the constant pool must be converted into direct references to the method's memory address. Dynamic linking is the process of converting symbolic references into direct references to the method being called. This process is also known as **dynamic linking** .



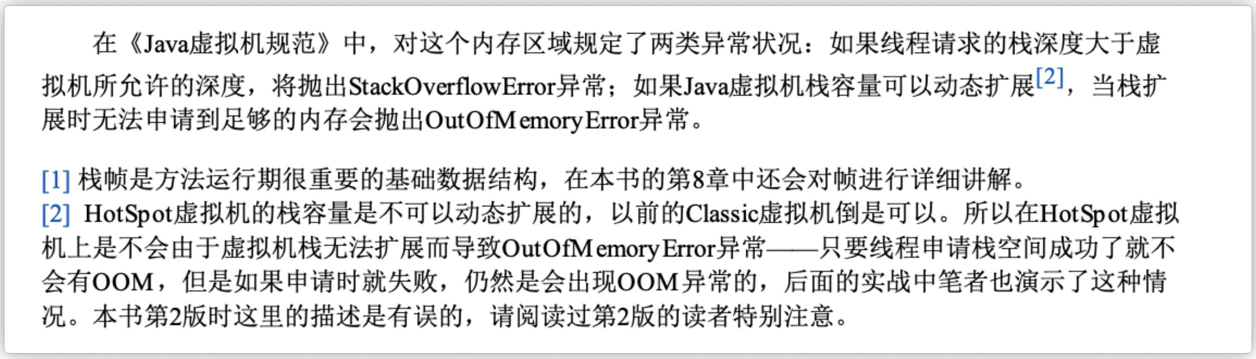
While stack space isn't infinite, it generally doesn't present a problem during normal calls. However, if a function call falls into an infinite loop, too many frames can be pushed onto the stack, occupying too much space and causing the stack to become too deep. If the thread's requested stack depth exceeds the maximum Java Virtual Machine stack depth, `StackOverFlowError` an error is thrown.

Java methods have two ways to return: a normal return statement and an exception. Regardless of the return method, the stack frame is popped. In other words, **the stack frame is created when the method is called and destroyed when the method ends. Whether a method completes normally or with an exception, it is considered a method completion.**

In addition `StackOverFlowError` to errors, stack errors may also occur `OutOfMemoryError` . This is because if the memory size of the stack can be dynamically expanded, the virtual machine cannot apply for sufficient memory space when dynam expanding the stack, and `OutOfMemoryError` an exception is thrown.

To briefly summarize, there are two possible errors that may occur in the stack during program execution:

- **`StackOverFlowError`** **:** If the stack memory size does not allow dynamic expansion, then when the thread requests the stack depth to exceed the maximum depth of the current Java virtual machine stack, `StackOverFlowError` an error is thrown.
- **`OutOfMemoryError`** **:** If the memory size of the stack can be expanded dynamically, then when the virtual machine cannot apply for sufficient memory space when dynamically expanding the stack, `OutOfMemoryError` an exception is thrown.


在《Java虚拟机规范》中，对这个内存区域规定了两类异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如果Java虚拟机栈容量可以动态扩展[2]，当栈扩展时无法申请到足够的内存会抛出OutOfMemoryError异常。

[1] 栈帧是方法运行期很重要的基础数据结构，在本书的第8章中还会对帧进行详细讲解。
[2] HotSpot虚拟机的栈容量是不可以动态扩展的，以前的Classic虚拟机倒是可以。所以在HotSpot虚拟机上是不会由于虚拟机栈无法扩展而导致OutOfMemoryError异常——只要线程申请栈空间成功了就不会有OOM，但是如果申请时就失败，仍然是会出现OOM异常的，后面的实战中笔者也演示了这种情况。本书第2版时这里的描述是有误的，请阅读过第2版的读者特别注意。

Native Method Stack

The native method stack is very similar to the virtual machine stack, but the difference is that **the virtual machine stack serves the virtual machine to execute Java methods (that is, bytecode), while the native method stack serves the native methods used by the virtual machine.** In the HotSpot virtual machine, the native method stack and the Java virtual machine stack are combined into one.

When a native method is executed, a stack frame is also created on the native method stack to store the local variable table, operand stack, dynamic link, and exit information of the native method.

After the method is executed, the corresponding stack frame will be popped out and the memory space will be released. Two errors, `StackOverFlowError` and , may also occur. `OutOfMemoryError`

heap

The Java heap is the largest memory area managed by the Java Virtual Machine. It is memory area shared by all threads and is created when the virtual machine starts. **This memory area's sole purpose is to store object instances; almost all object**

instances and arrays are allocated here.

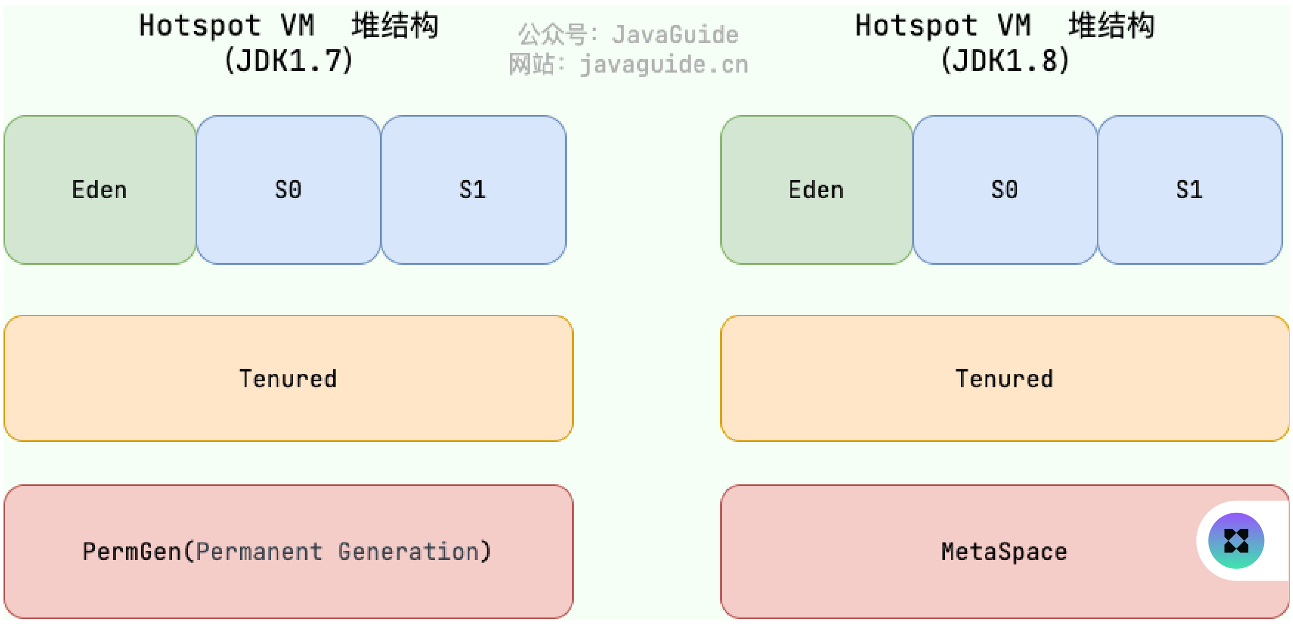
In the Java world, almost all objects are allocated on the heap. However, with the development of JIT compilers and the maturation of escape analysis technology, stack allocation and scalar substitution optimization techniques are leading to subtle changes, making the heap allocation of all objects less absolute. Starting with JDK 1.7, escape analysis has been enabled by default. If object references in certain methods are not returned or used externally (that is, they do not escape), then the objects can be allocated directly on the stack.

The Java heap is the primary area managed by the garbage collector, and is therefore also called **the GC heap** . From a garbage collection perspective, since most modern collectors use a generational garbage collection algorithm, the Java heap can be further subdivided into the Young and Old generations; and more specifically, into the Eden, Survivor, and Old spaces. This further division aims to improve memory recovery or speed up memory allocation.

In JDK 7 and prior versions, the heap memory is usually divided into the following three parts:

1. Young Generation Memory
2. Old Generation
3. Permanent Generation

The Eden area and the two Survivor areas S0 and S1 shown in the figure below all belong to the new generation, the middle layer belongs to the old generation, and the bottom layer belongs to the permanent generation.


Hotspot VM 堆结构 (JDK1.7)
公众号：JavaGuide
网站：javaguide.cn
Hotspot VM 堆结构 (JDK1.8)
Eden
S0
S1
Eden
S0
S1
Tenured
Tenured
PermGen(Permanent Generation)
MetaSpace

Since JDK 8, PermGen has been replaced by Metaspace, which uses local memory. (I'll explain this in detail in the Method Area section.)

In most cases, objects are first allocated in the Eden area. After a new generation garbage collection, if the object is still alive, it will be moved to S0 or S1, and its age will be incremented by 1 (after moving from Eden to Survivor, the object's initial age becomes 1). When its age reaches a certain level (15 by default), it will be promoted to the old generation. The age threshold for object promotion to the old generation can be `-XX:MaxTenuringThreshold` set using the parameter . However, the value should be between 0 and 15, otherwise the following error will occur:

```bash
MaxTenuringThreshold of 20 is invalid; must be between 0 and 15
```

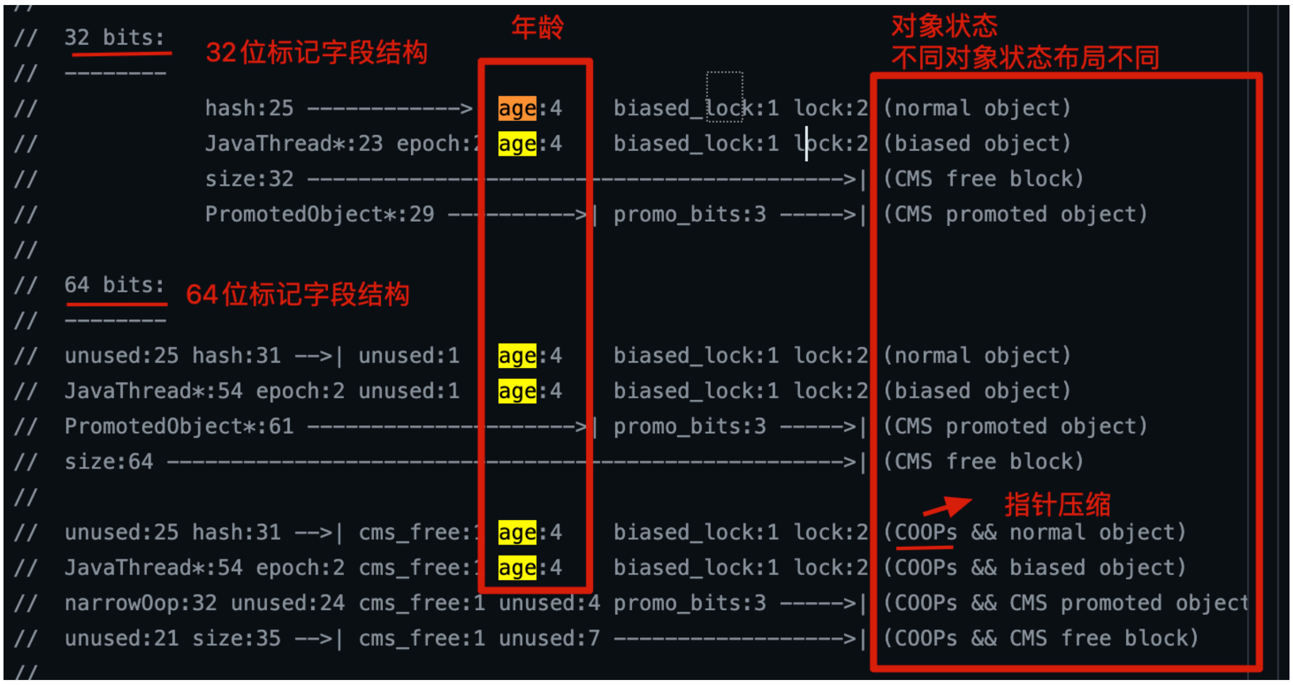
Why can the age range only be 0-15?

Because the area recording the age is in the object header, the size of this area is usually 4 bits. The maximum binary number that can be represented by these 4 bits is 1111, which is 15 in decimal. Therefore, the age of an object is limited to 0 to 15.

Here we briefly introduce it in detail in combination with object layout.

In the HotSpot VM, the memory layout of objects can be divided into three areas: the object header, instance data, and alignment padding. The object header consists of two parts: the mark word and the type pointer. A detailed description of the object memory layout will be provided later, so I won't repeat it here.

This age information is stored in the mark field (the mark field also stores other information about the object itself, such as hash code, lock status information, etc.). `markOop.hpp` The structure of the mark word is defined as follows:



```
//  32 bits:
//  --------
//           hash:25 ------------>| age:4    biased_lock:1 lock:2 (normal object)
//           JavaThread*:23 epoch:2 age:4    biased_lock:1 lock:2 (biased object)
//           size:32 ------------------------------------------>| (CMS free block)
//           PromotedObject*:29 ---------->| promo_bits:3 ----->| (CMS promoted object)
//
//  64 bits:
//  --------
//  unused:25 hash:31 -->| unused:1   age:4    biased_lock:1 lock:2 (normal object)
//  JavaThread*:54 epoch:2 unused:1   age:4    biased_lock:1 lock:2 (biased object)
//  PromotedObject*:61 --------------------->| promo_bits:3 ----->| (CMS promoted object)
//  size:64 ----------------------------------------------------->| (CMS free block)
//
//  unused:25 hash:31 -->| cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && normal object)
//  JavaThread*:54 epoch:2 cms_free:1 age:4    biased_lock:1 lock:2 (COOPs && biased object)
//  narrowOop:32 unused:24 cms_free:1 unused:4 promo_bits:3 ----->| (COOPs && CMS promoted object
//  unused:21 size:35 -->| cms_free:1 unused:7 ------------------>| (COOPs && CMS free block)
```

You can see that the size occupied by the object age is indeed 4 bits.

> **Correction (see: issue552)** : "When Hotspot traverses all objects, it accumulates the size they occupy from young to large according to their age. When the accumulated size exceeds half of the Survivor area when it reaches a certain age, the smaller `MaxTenuringThreshold` value of this age and is taken as the new promotion age threshold."
> **The code for dynamic age calculation is as follows**

```c++
uint ageTable::compute_tenuring_threshold(size_t
survivor_capacity) {
 //survivor_capacity是survivor空间的大小
size_t desired_survivor_size = (size_t)((((double)
survivor_capacity)*TargetSurvivorRatio)/100);//TargetSurvivorRatio
为50
size_t total = 0;
uint age = 1;
while (age < table_size) {
total += sizes[age];//sizes数组是每个年龄段对象大小
if (total > desired_survivor_size) break;
age++;
```

```
    }
    uint result = age < MaxTenuringThreshold ? age :
    MaxTenuringThreshold;
      ...
    }
```

The most common error here is `OutOfMemoryError` the error, and there are several manifestations of this error, such as:

1. **`java.lang.OutOfMemoryError: GC Overhead Limit Exceeded`** : This error occurs when the JVM spends too much time performing garbage collection and can only reclaim little heap space.
2. **`java.lang.OutOfMemoryError: Java heap space`** This error occurs when there is insufficient heap space to store a new object. (This is related to the configured maximum heap size and is subject to the physical memory available. The maximum heap size can be `-Xmx` configured via a parameter; if not specified, the default value will be used. See Default Java 8 max heap size for details.)
3. ...

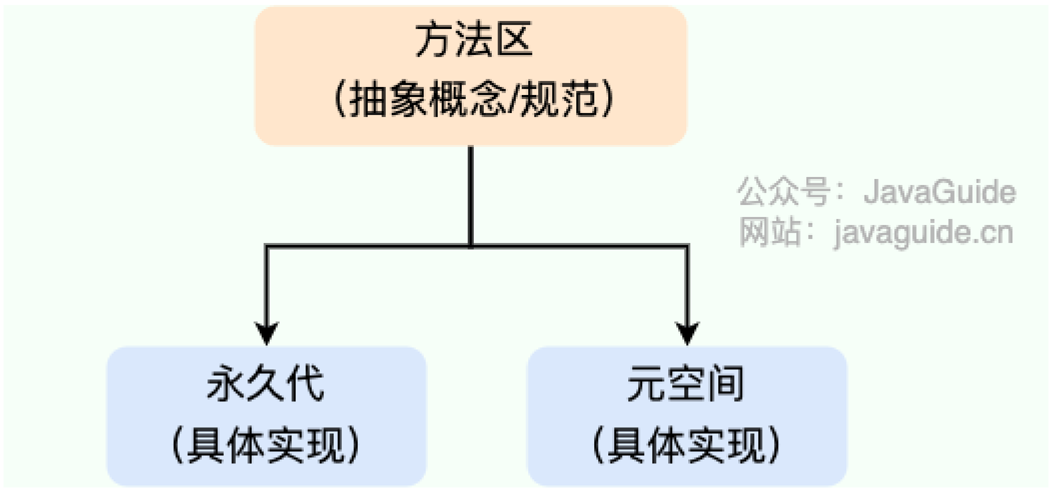
Method Area

The method area is a logical area of the JVM runtime data area and is a memory area shared by all threads.

The Java Virtual Machine Specification only defines the concept of a method area and its purpose. Exactly how the method area is implemented is a matter for the virtual machine itself. This means that different virtual machine implementations implement the method area differently.

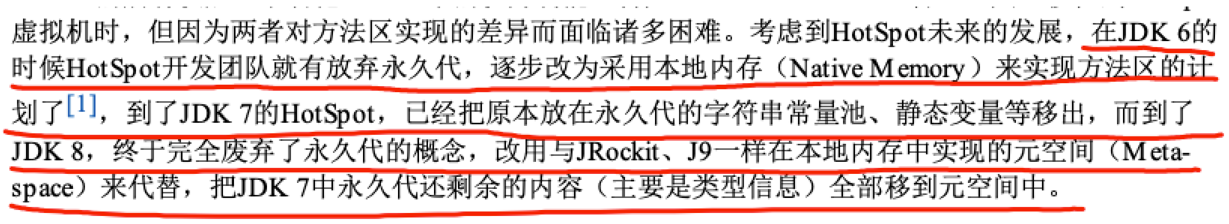
When the virtual machine wants to use a class, it reads and parses the class file to obtain relevant information and then stores this information in the method area. The method area stores **data such as class information loaded by the virtual machine, field information, method information, constants, static variables, and the code cache compiled by the just-in-time compiler** .

What is the relationship between the method area, the permanent generation, and the metaspace? The relationship between the method area, the permanent generation, and the metaspace is very similar to the relationship between interfaces a classes in Java. A class implements an interface, and the class can be considered the permanent generation and the metaspace, while the interface can be considered the method area. In other words, the permanent generation and the metaspace are two implementations of the method area in the HotSpot VM specification. Furthermore, the permanent generation was the implementation of the method area before JDK 1.8, while the method area was implemented in the metaspace in JDK 1.8 and later.



Why should we replace PermGen with MetaSpace?

The following figure is from "In-depth understanding of Java Virtual Machine" 3rd edition 2.2.5



虚拟机时，但因为两者对方法区实现的差异而面临诸多困难。考虑到HotSpot未来的发展，在JDK 6的时候HotSpot开发团队就有放弃永久代，逐步改为采用本地内存（Native Memory）来实现方法区的计划了[1]，到了JDK 7的HotSpot，已经把原本放在永久代的字符串常量池、静态变量等移出，而到了JDK 8，终于完全废弃了永久代的概念，改用与JRockit、J9一样在本地内存中实现的元空间（Meta-space）来代替，把JDK 7中永久代还剩余的内容（主要是类型信息）全部移到元空间中。

1. The entire permanent generation has a fixed size limit set by the JVM itself and cannot be adjusted (that is, it is limited by the JVM memory). The metaspace uses local memory and is limited by the available memory of the local machine. Although the metaspace may still overflow, the probability of overflow is smaller than before.

> When the metaspace overflows, you will get the following error: `java.lang.OutOfMemoryError: MetaSpace`

You can `-XX: MaxMetaspaceSize` set the maximum Metaspace size using the flag. The default value is unlimited, which means it is only limited by the system memory. `-XX: MetaspaceSize` The resize flag defines the initial size of the Metaspace. If this flag is not specified, the Metaspace will be dynamically resized based on the application demand runtime.

2. The metaspace stores the metadata of the class. In this way, the amount of metadata loaded is not `MaxPermSize` controlled by the system, but by the actual available space of the system, so that more classes can be loaded.

3. In JDK8, when the HotSpot and JRockit codes were merged, JRockit never had a permanent generation. After the merger, there was no need to set up such a permanent generation.

4. The permanent generation brings unnecessary complexity to GC and the recovery efficiency is low.

What are the commonly used parameters in the method area?

Before JDK 1.8, when the permanent generation had not yet been completely removed, the following parameters were usually used to adjust the method area size.

```java
-XX:PermSize=N //方法区 (永久代) 初始大小
-XX:MaxPermSize=N //方法区 (永久代) 最大大小,超过这个值将会抛出
OutOfMemoryError 异常:java.lang.OutOfMemoryError: PermGen
```

Relatively speaking, garbage collection rarely occurs in this area, but it does not mean that the data will "exist forever" after entering the method area.

In JDK 1.8, the method area (HotSpot's permanent generation) was completely removed (this was already the case in JDK 1.7), and replaced by the Metaspace, which uses local memory. The following are some common parameters:

```java
-XX:MetaspaceSize=N //设置 Metaspace 的初始 (和最小大小)
-XX:MaxMetaspaceSize=N //设置 Metaspace 的最大大小
```

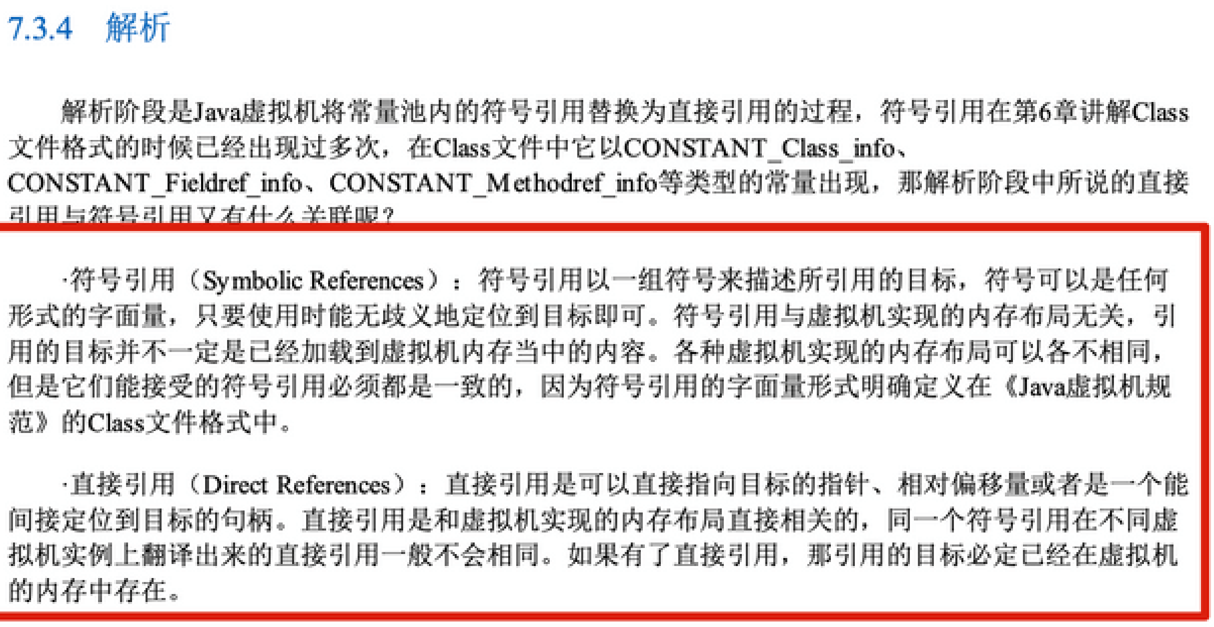
A big difference from the permanent generation is that if you do not specify a size, the virtual machine will use up all available system memory as more classes are created.

Runtime constant pool

In addition to the description information of the class version, fields, methods, interfaces, etc., the Class file also contains a constant pool table (Constant Pool Table) for storing various literals and symbolic references generated during compilation .

A literal is a fixed value representation in source code; its meaning is clearly defined. Literals include integers, floating-point numbers, and string literals. Common symbol references include class symbol references, field symbol references, method symbol references, and interface method symbol references.

Section 7.34 of the third edition of "In-depth Understanding of the Java Virtual Machine" explains symbolic references and direct references as follows:



7.3.4 解析

解析阶段是Java虚拟机将常量池内的符号引用替换为直接引用的过程，符号引用在第6章讲解Class文件格式的时候已经出现过多次，在Class文件中它以CONSTANT_Class_info、CONSTANT_Fieldref_info、CONSTANT_Methodref_info等类型的常量出现，那解析阶段中所说的直接引用与符号引用又有什么关联呢？

·符号引用（Symbolic References）：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定是已经加载到虚拟机内存当中的内容。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须都是一致的，因为符号引用的字面量形式明确定义在《Java虚拟机规范》的Class文件格式中。

·直接引用（Direct References）：直接引用是可以直接指向目标的指针、相对偏移量或者是一个能间接定位到目标的句柄。直接引用是和虚拟机实现的内存布局直接相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经在虚拟机的内存中存在。

The constant pool table will be stored in the runtime constant pool of the method area after the class is loaded.

The runtime constant pool functions similarly to the symbol table of a traditional programming language, although it contains a wider range of data than a typical symbol table.

Since the runtime constant pool is part of the method area, it is naturally limited by the method area memory. When the constant pool can no longer apply for memory, `OutOfMemoryError` an error will be thrown.

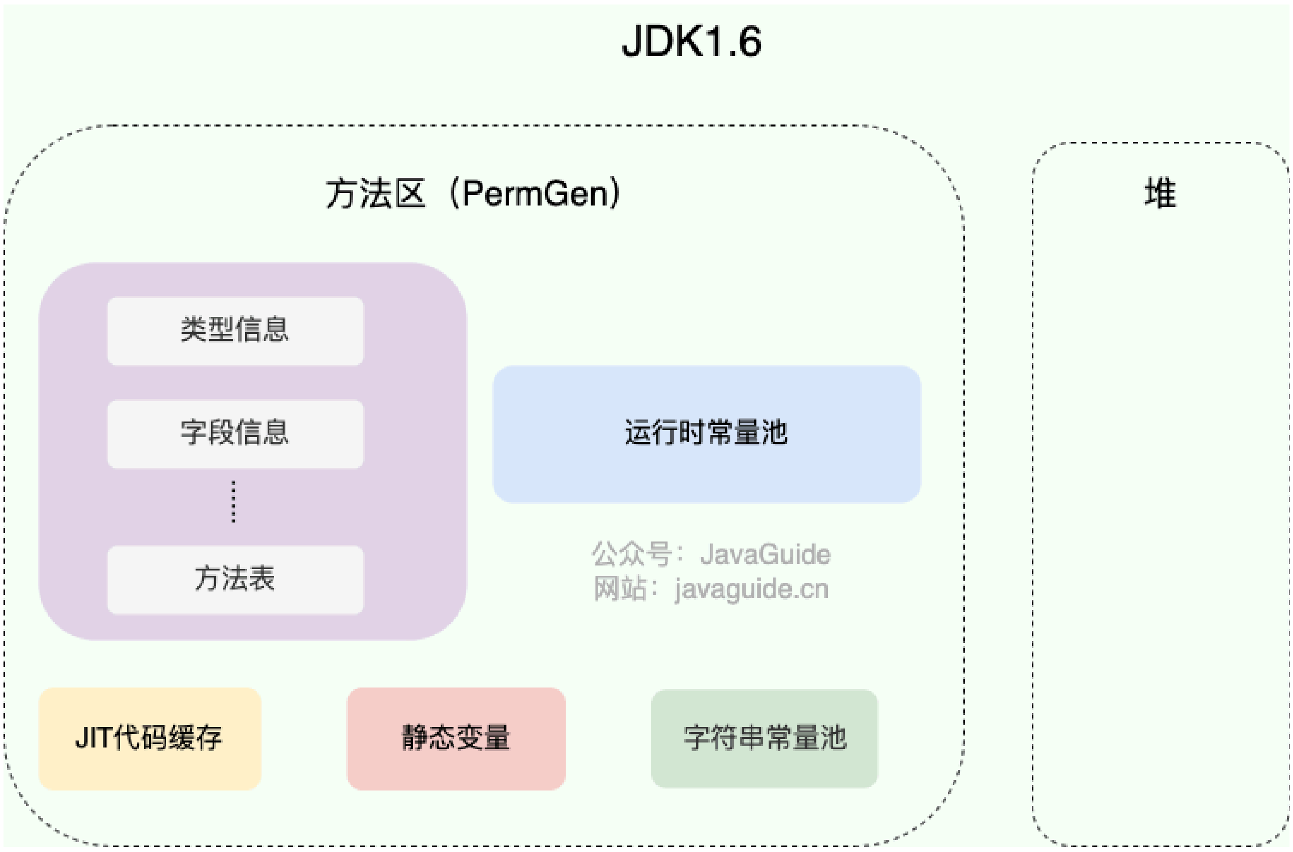
String constant pool

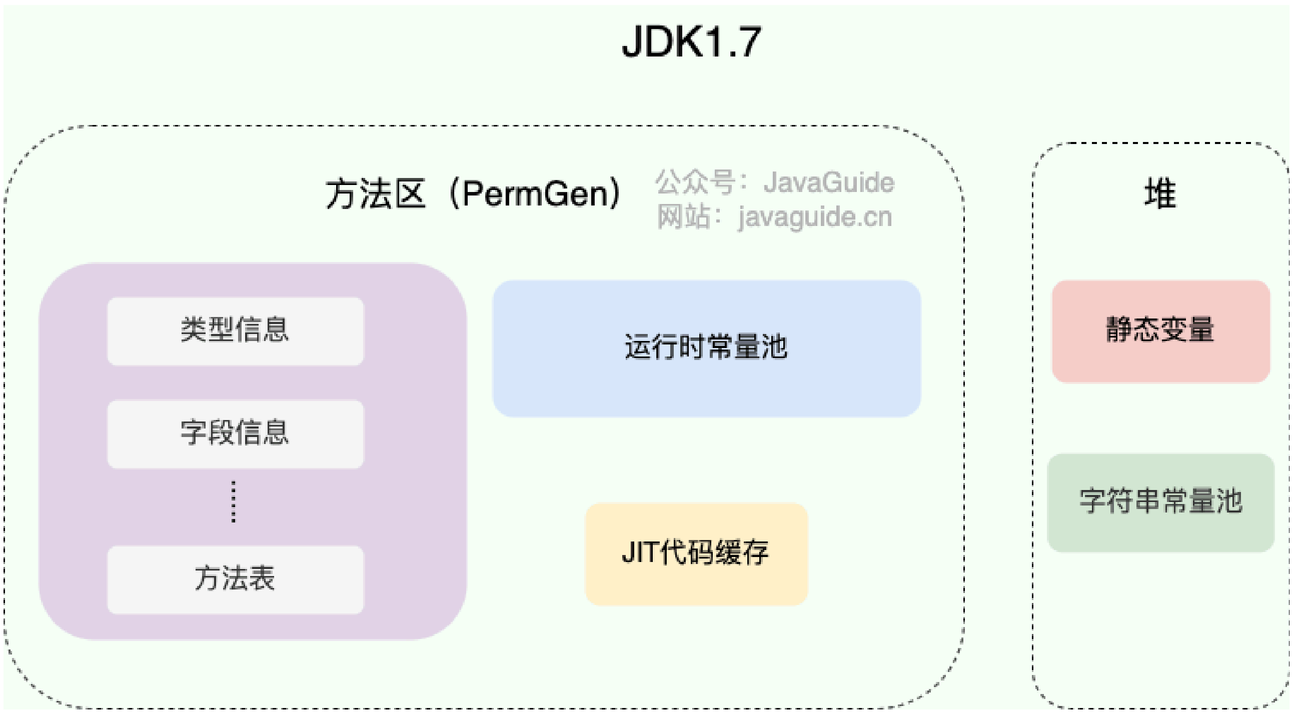
The string constant pool is an area specially opened up by the JVM for strings (String class) in order to improve performance and reduce memory consumption. The main purpose is to avoid repeated creation of strings.

```java
// 在字符串常量池中创建字符串对象 ”ab“
// 将字符串对象 ”ab“ 的引用赋值给给 aa
String aa = "ab";
// 直接返回字符串常量池中字符串对象 ”ab“，赋值给引用 bb
String bb = "ab";
System.out.println(aa==bb); // true
```

The implementation of the string constant pool in the HotSpot virtual machine `src/hotspot/share/classfile/stringTable.cpp` can `StringTable` be simply understood as a fixed-size constant pool `HashTable` with a capacity `StringTableSize` (which can `-XX:StringTableSize` be set through a parameter). It stores the mapping relationship between strings (keys) and references to string objects (values). The references to string objects point to string objects in the heap.

Before JDK 1.7, the string constant pool was stored in the permanent generation. In JDK 1.7, the string constant pool and static variables were moved from the permanent generation to the Java heap.





Why did JDK 1.7 move the string constant pool to the heap?

This is primarily due to the low GC efficiency of the permanent generation (implemented in the method area). GC is only performed during a full heap collection (Full GC). Java programs often have a large number of strings waiting to be collected. Placing the string constant pool in the heap allows for more efficient and timely reclaiming of string memory.

Related question: Does the JVM constant pool store objects or references? - RednaxelaFX - Zhihu

Finally, let me share what Professor Zhou Zhiming said in issue #112 of the sample code & errata GitHub repository in "In-depth Understanding of the Java Virtual Machine (3rd Edition)" :

> **The runtime constant pool, method area, and string constant pool are logical concepts that do not change with the virtual machine implementation and are public and abstract. Metaspace and Heap are physical concepts related to a specific virtual machine implementation and are private and specific.**

Direct Memory

Direct memory is a special memory buffer that is not allocated in the Java heap or me area, but is allocated on the native memory through JNI.

Direct memory is not part of the virtual machine's runtime data area, nor is it a memory area defined in the virtual machine specification. However, this part of memory is frequently used and may cause `OutOfMemoryError` errors.

NIO (Non-Blocking I/O, also known as New I/O) , newly added in JDK 1.4 , introduces an I/O method based on **channels and buffers . It can directly allocate off-heap memory using the native library and then reference this memory using a DirectByteBuffer object stored in the Java heap. This can significantly improve performance in some scenarios by avoiding data copying back and forth between the Java heap and the native heap** .

Direct memory allocation is not limited by the Java heap, but since it is memory, it is limited by the total memory size of the machine and the processor addressing space.

A similar concept is **off-heap memory** . Some articles equate direct memory with off-heap memory, but I personally think this is not entirely accurate.

Off-heap memory is memory that allocates memory objects outside the heap. This memory is directly managed by the operating system (rather than the virtual machine). As a result, the impact of garbage collection on the application can be reduced to a certain extent.

Exploring HotSpot Virtual Machine Objects

The above introduction gives us a general understanding of the memory situation of the virtual machine. Now let's take a detailed look at the entire process of object allocation, layout, and access in the Java heap of the HotSpot virtual machine.

Object creation

I suggest that you should be able to write out the Java object creation process and understand what each step is doing.

Step 1: Class loading check

When the virtual machine encounters a new instruction, it first checks whether the instruction's argument can locate a symbolic reference to the class in the constant pool, and then checks whether the class represented by the symbolic reference has been loa parsed, and initialized. If not, the corresponding class loading process must be perform first.

Step 2: Allocate memory

After **the class loading check passes, the virtual machine allocates memory** for the newly created object . The memory size required for the object is determined after class loading is complete. Allocating space for the object is equivalent to allocating a block of memory of a specific size from the Java heap. There are two **allocation methods : "pointer bump"** and **"free list." The choice of allocation method is determined by the regularity of the Java heap, which in turn is determined by whether the garbage collector used has compaction capabilities** .

Two ways of memory allocation (supplementary content, need to be mastered):

- Pointer collision:
  - Applicable occasions: when the heap memory is regular (that is, there is no memory fragmentation).
  - Principle: All used memory is integrated into one side, and unused memory is placed on the other side. There is a demarcation pointer in the middle. You only need to move the pointer toward the unused memory by the object memory size.
  - GC collector using this allocation method: Serial, ParNew
- Free List:
  - Applicable occasions: when the heap memory is irregular.
  - Principle: The virtual machine maintains a list that records which memory blocks are available. When allocating, it finds a large enough memory block to allocate to the object instance and finally updates the list record.
  - GC collector using this allocation method: CMS

The choice of either of these two methods depends on whether the Java heap memory is well-organized. This, in turn, depends on whether the GC collector uses a "mark-and-sweep" or "mark-and-compact" (also known as "mark-and-compact") algorithm. It's worth noting that the copying algorithm also produces well-organized memory.

Memory allocation concurrency issues (supplementary content, need to be mastered)

When creating objects, there is a very important issue, which is thread safety. Because in the actual development process, creating objects is very frequent. As a virtual machine, it is necessary to ensure that threads are safe. Generally speaking, virtual machines use tv methods to ensure thread safety:

- **CAS + Failure Retry:** CAS is an implementation of optimistic locking. Optimistic locking assumes no conflicts and completes an operation without locking. If a conflict fails, the operation is retried until it succeeds. **Virtual machines use CAS with failure retry to ensure the atomicity of update operations.**
- **TLAB:** Allocate a block of memory in the Eden area for each thread in advance. When the JVM allocates memory to objects in the thread, it first allocates it in TLAB. When the object is larger than the remaining memory in TLAB or the memory in TLAB is exhausted, the above-mentioned CAS is used for memory allocation.

Step 3: Initialize zero value

After the memory allocation is completed, the virtual machine needs to initialize the allocated memory space to zero values (excluding the object header). This step ensures that the instance fields of the object can be used directly in the Java code without assigning initial values, and the program can access the zero values corresponding to the data types of these fields.

Step 4: Set the object header

After initializing to zero values, **the VM performs necessary configuration on the object** , including information such as which class the object is an instance of, how to find the class metadata, the object's hash code, and the object's GC generation age. **This information is stored in the object header.** Furthermore, the object header configuration varies depending on the VM's current state, such as whether biased locking is enabled.

Step 5: Execute the init method

After all of the above steps are completed, from the virtual machine's perspective, a new object has been created. However, from the Java program's perspective, object creation has just begun; `<init>` methods haven't yet been executed, and all fields are still zero. Therefore, generally speaking, executing the new instruction is followed by executing `<init>` methods, initializing the object according to the programmer's wishes. Only then is a truly usable object considered fully created.

Memory layout of the object

In the Hotspot virtual machine, the layout of objects in memory can be divided into three areas: **object header** , **instance data** , **and padding** .

The object header includes two parts of information:

1. Mark Word: used to store the runtime data of the object itself, such as hash code, GC generation age, lock status flag, lock held by thread, biased thread ID, biased timestamp, etc.
2. Type pointer (Klass pointer): The object points to its class metadata. The virtual machine uses this pointer to determine which class the object is an instance of.

The instance data part is the valid information actually stored in the object , and is also the content of various types of fields defined in the program.

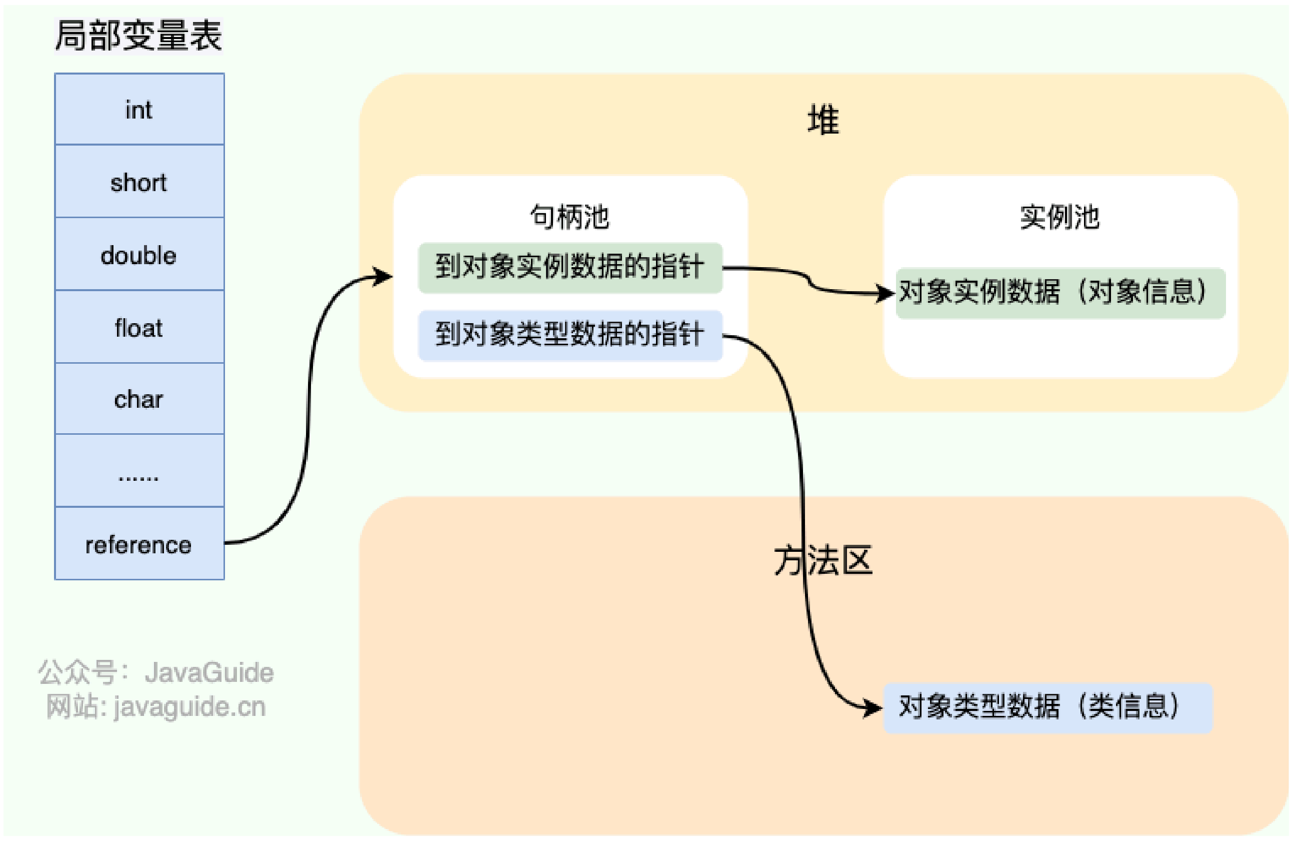
Alignment padding is not mandatory and has no special meaning; it simply serves as a placeholder. The Hotspot VM's automatic memory management system requires that the starting address of an object be an integer multiple of 8 bytes. In other words, the object size must be an integer multiple of 8 bytes. Object headers are multiples of 8 bytes (1 or 2). Therefore, when the object instance data is misaligned, alignment padding is required.

Object access location

The purpose of creating objects is to use them. Java programs use reference data on the stack to manipulate specific objects on the heap. Object access methods are determined by the virtual machine implementation. Currently, the mainstream access methods include **using handles** and **direct pointers** .

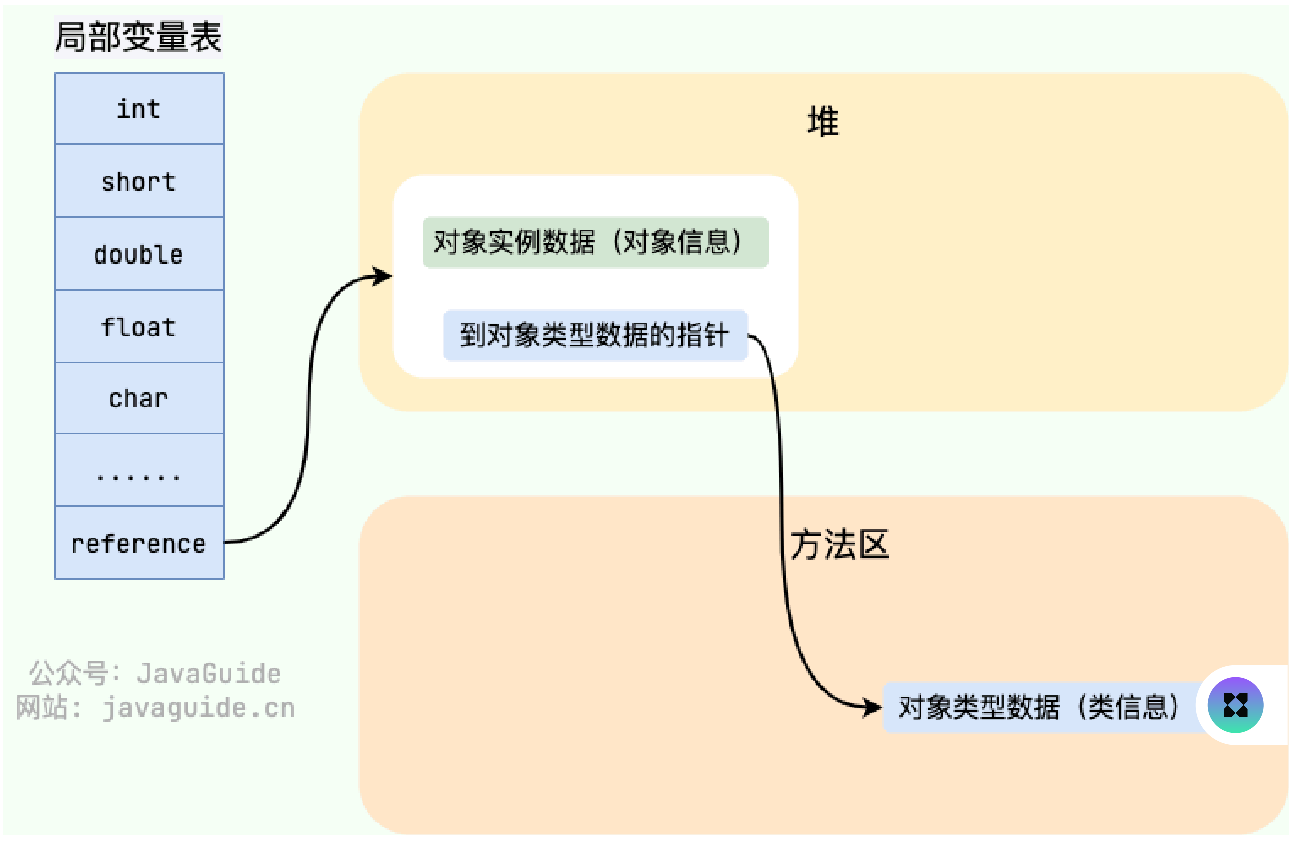
Handle

If a handle is used, a block of memory will be allocated in the Java heap as a handle pool. The reference stores the handle address of the object, and the handle contains the specific address information of the object instance data and the object type data.



Direct pointers

If direct pointer access is used, the address of the object is stored directly in the reference.



Both object access methods have their own advantages. The biggest advantage of using handles is that the reference stores a stable handle address. When the object is moved, only the instance data pointer in the handle changes; the reference itself remains unchanged. The biggest advantage of using direct pointer access is speed, as it saves the time overhead of a pointer positioning.

The HotSpot virtual machine mainly uses this method to access objects.

refer to

- Deep Understanding of Java Virtual Machine: JVM Advanced Features and Best Practices (Second Edition)
- Write Your Own Java Virtual Machine
- Chapter 2. The Structure of the Java Virtual Machine: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html
- JVM stack frame internal structure - dynamic link: https://chenxitag.com/archives/368

- When does the "literal" in new String("literal") in Java enter the string constant pool? - Answer by Mu Girl - Zhihu: https://www.zhihu.com/question/55994121/answer/147296098
- Does the JVM constant pool store objects or references? - Answer by RednaxelaFX - Zhihu: https://www.zhihu.com/question/57109429/answer/151717241
- http://www.pointsoftware.ch/en/under-the-hood-runtime-data-areas-javas-memory-model/
- https://dzone.com/articles/jvm-permgen-–-where-art-thou
- https://stackoverflow.com/questions/9095748/method-area-and-permgen

JavaGuide官方公众号
（微信搜索JavaGuide）

1、公众号后台回复“PDF”获取原创PDF面试手册

2、公众号后台回复“学习路线”获取Java学习路线最新版

3、公众号后台回复“开源”获取优质Java开源项目合集

4、公众号后台回复“八股文”获取Java面试真题+面经

Recently Updated2025/8/6 18:37

Contributors: Snailclimb , Jin Yang , Charles Wu , SnailClimb , JKSAGE , bestgrc , yellowgg , CoderZZ , yidasanqian , LicoCode , Jim Han , Kou Shuang , Shuang Kou , liwenguang , Rocky , shuang.kou , jiayao , ygf , guide , hailong.sha , Ma Zhaoming , TommyMerlin , kaka2634 , 13350063660 , Sheldon7777 , ZhoucpSAMA , huzhuoyu , Gale , sam , geomonlin , Dayu , Guide , Mr.Hope , paigeman , seven17777777 , gyaatrox , qksuki , wunameya , Kisa-Dong , CoteNite , uncle-lv

Copyright © 2025 Guide