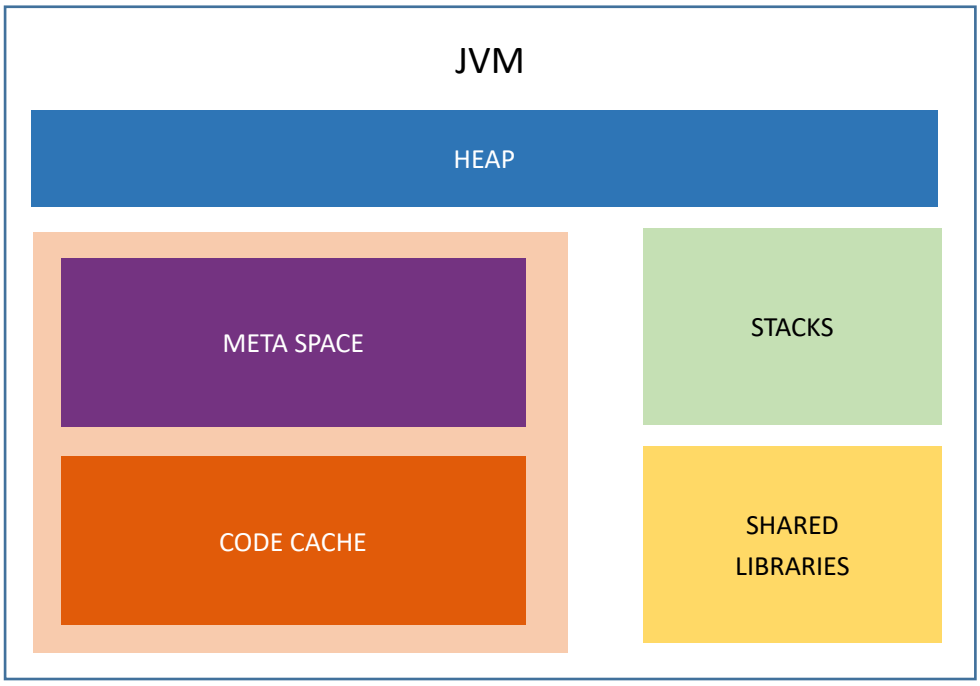
Java 8.0 内存模型

当 JVM 进程启动时，操作系统会为该进程分配内存空间，包括：

- 堆（Heap）
- 元空间（Meta Space）
- JIT 代码缓存（JIT Code Cache）
- 线程堆栈（Thread Stack）
- 共享库（Shared Library）



我们将这些合称为“本地内存（Native Memory）”。下面分别介绍它们的含义

Heap

JVM 用来存储对象实例的地方。这个区域被分为“青年代”（Young Generation Space）和“终身代”（Tenured Space）两部分：

Young Generation：“青年代”又被分为“Eden Space”和“Survivor Space”两部分。前者用于分配新创建的对象，后者存放的是微量垃圾回收（Minor Garbage Collection）之后存留下来的对象，这个区域又被均分为 S0 区和 S1 区。

微量垃圾回收：针对 EDEN/SURVIVOR 区域执行的垃圾回收。当 EDEN 区域没有足够的空间创建新的对象时，就会触发一次微量垃圾回收。微量垃圾回收只会清理 EDEN 和 SURVIVOR 区域的对象。

如果要按照清理范围来对分垃圾回收行为进行分类的话，那么可以分为两大类：1）只对 Young Generation 进行清理；2）清理范围扩大到 Tenured Space，或者说整个 Heap。

Tenured Space：在经过某个指定阈值的垃圾回收次数之后仍然得以存活的对象，最后就会被放入“终身代”区域。

因此，Heap 当中分块的原则可以说实际上是依据垃圾回收的范围来划分的。

Meta Space

元空间在堆内存以外。早期的 Java 虚拟机中存在一个叫 Perm Gen Space 的内存区域，用作存储从类加载器加载来的类定义。由于 Perm Gen Space 设计的不灵活性容易带来内存溢出错误，Java 8 定义了元空间来代替它。理论上元空间没有最大内存限制（若一定需要限制的话，Java 提供相应的启动参数），因此避免了内存溢出错误；不过如果它无限制的增加，甚至令操作系统不得不使用虚拟内存，那也会带来严重的性能问题。

Code Cache

我们知道 Java 编译器编译出来的 .class 文件包含的是字节码。而 JVM 执行字节码时，又会需要将其转换成操作系统本地指令（不同的操作系统本地指令不一样）。早期的 JVM 是直接将字节码在虚拟机中执行，效率较低；后来引进了一个叫 JIT 编译器的概念，即在运行时将经常重复运行的字节码编译成本地指令，并缓存起来，这样执行时就有很大机会运行本地指令而不是字节码，执行效率大大提高。Code Cache 内存区就是用来缓存 JIT 本地指令的。