**1、 CPU和内存的交互**

在计算机中，cpu和内存的交互最为频繁，相比内存，磁盘读写太慢，内存相当于高速的缓冲区。

但是随着cpu的发展，内存的读写速度也远远赶不上cpu。因此cpu厂商在每颗cpu上加上高速缓存，用于缓解这种情况。现在cpu和内存的交互大致如下。

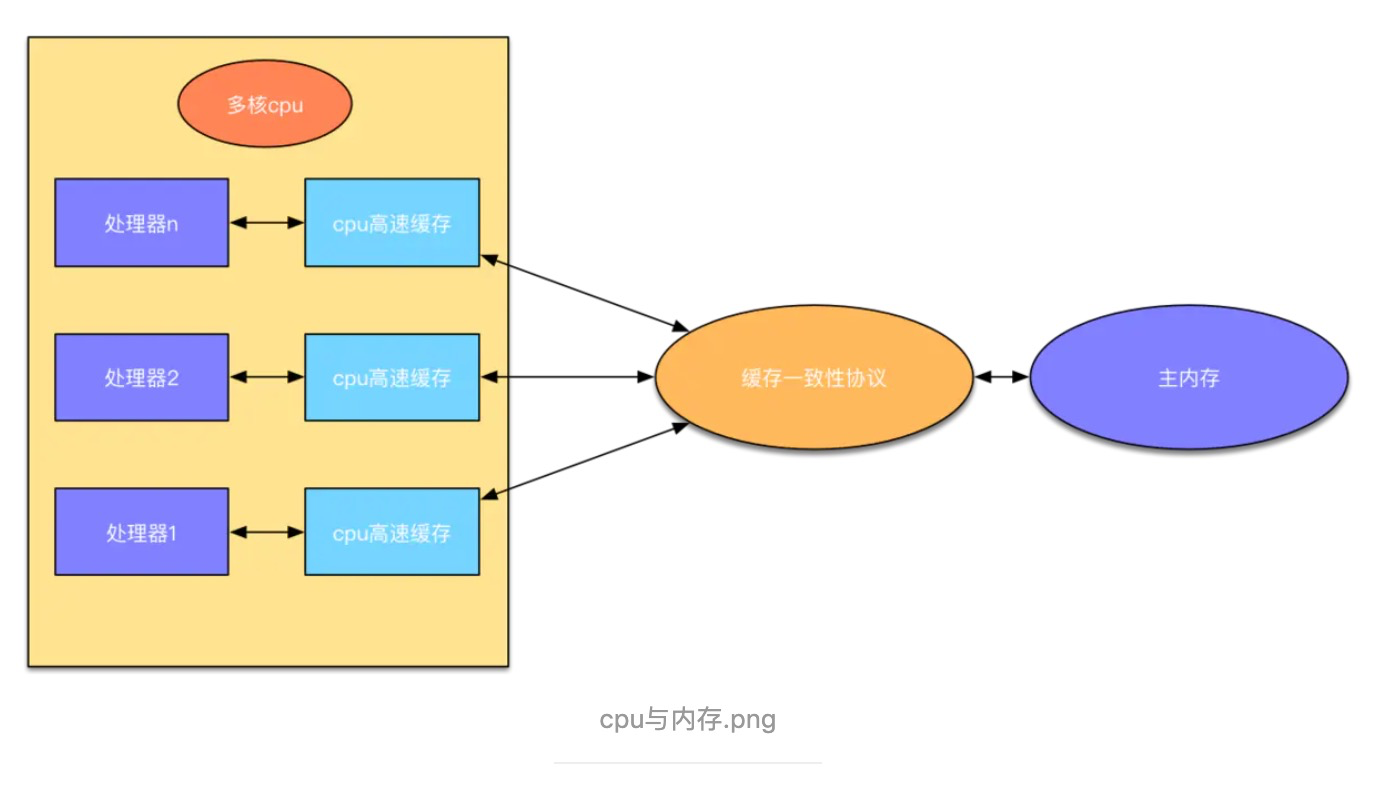


在多核cpu中，每个处理器都有各自的高速缓存(L1,L2,L3)，而主内存确只有一个 。虽然加入高速缓存解决了处理器和内存的矛盾(一快一慢)，但是引来了新的问题 - **缓存一致性**

**1.1、缓存一致性**

如何保证多个处理器运算涉及到同一个内存区域时，保证运行时数据一致性？

为了解决这个问题，各个处理器需遵循一些协议保证一致性。【如MSI，MESI协议等等】



**1.2、内存屏障（Memory Barrier）**

CPU中，每个CPU又有多级缓存【上图统一定义为高速缓存】，一般分为L1,L2,L3，因为这些缓存的出现，提高了数据访问性能，避免每次都向内存索取，但是弊端也很明显，不能实时的和内存发生信息交换，分**在不同CPU执行的不同线程对同一个变量的缓存值不同。**而且由于操作系统可能存在重排序，导致读取到错误的数据，因此，操作系统提供了一些**内存屏障**以解决这种问题。

硬件层的内存屏障分为两种：Load Barrier和Store Barrier，即读屏障和写屏障。【内存屏障是硬件层的】

内存屏障作用

（1）阻止屏障两侧指令重排序

（2）强制把写缓冲区/高速缓存中的脏数据等写回主内存，让缓存中相应的数据失效。

**1.3、volatile型变量**

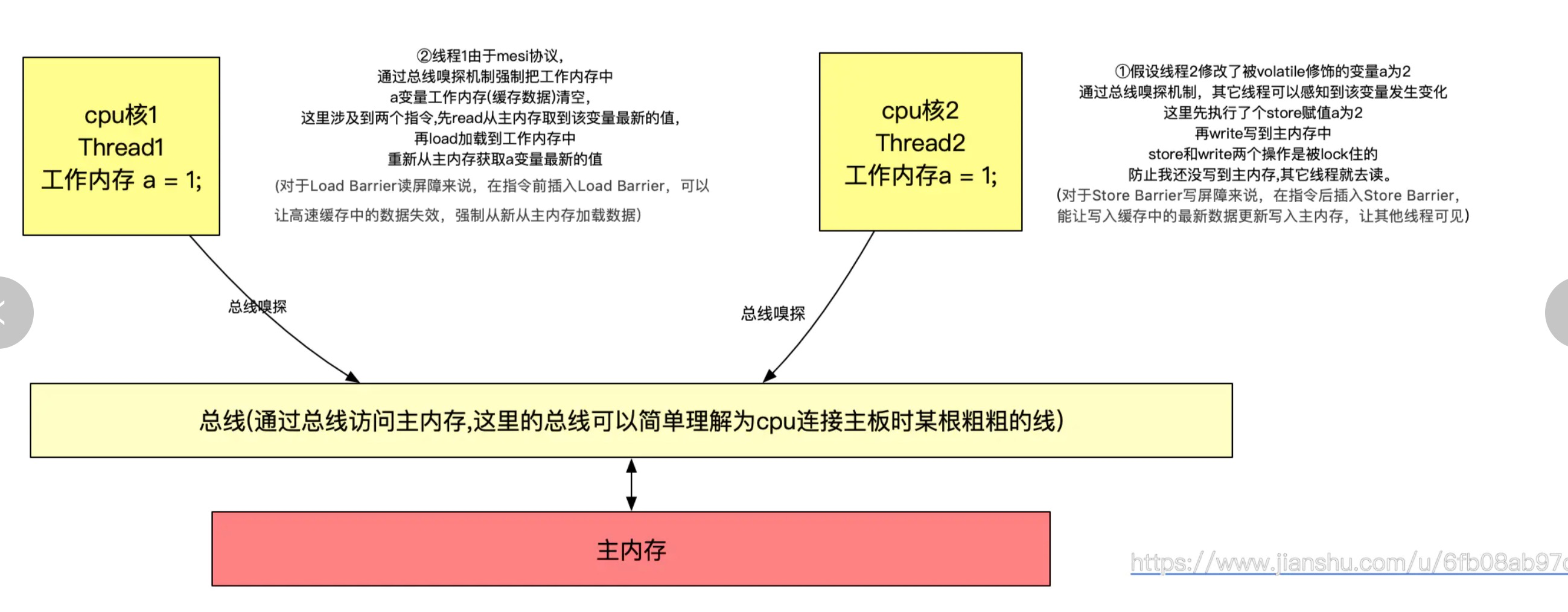
不同硬件对内存屏障的实现方式不一样。java屏蔽掉这些差异，声明某个变量为volatile修饰时，通过jvm生成读写内存屏障的指令。对于读屏障:在指令前插入读屏障，可以让高速缓存中的数据失效，强制从主内存取。

volatile有两个作用：

（1）**可见性**（对于一个该变量的读，一定能看到读之前最后的写入）

（2）**防止指令重排序**（执行代码时,为了提高执行效率,如果没有声明volatile，会在不影响最后结果的前提下对指令进行重新排序）

至于volatile底层是怎么实现保证不同线程可见性的，这里涉及到的就是硬件上的，被volatile修饰的变量在进行写操作时，会生成一个特殊的汇编指令，该指令会触发mesi协议，会存在一个总线嗅探机制的东西，简单来说就是这个cpu会不停检测总线中该变量的变化，如果该变量一旦变化了，由于这个嗅探机制，其它cpu会立马将该变量的cpu缓存数据清空掉，重新的去从主内存拿到这个数据。简单画了个图。

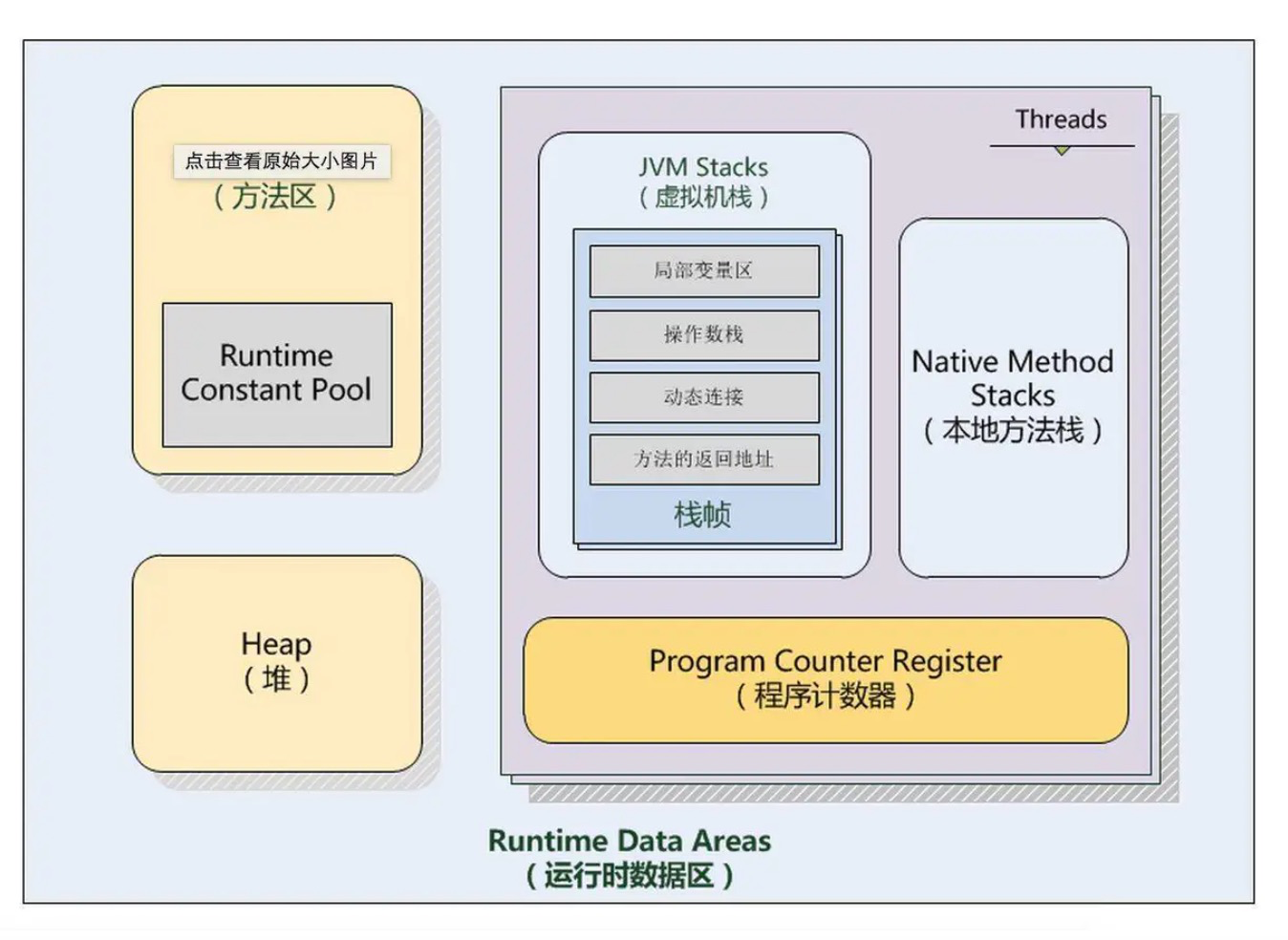


**2、Java内存模型**

Java内存模型（Java Memory Model ,JMM）就是一种符合计算机内存模型规范的，屏蔽了各种硬件和操作系统的访问差异的，保证了Java程序在各种平台下对内存的访问都能保证效果一致的机制及规范。

根据java虚拟机规范，java虚拟机管理的内存将分为下面五大区域：

其中方法区和堆是所有线程共享的，栈、本地方法栈和程序计数器则为线程私有的



**2.1、五大内存区域**

**2.1.1、程序计数器**

程序计数器是一块很小的内存空间，它是线程私有的，可以认作为当前线程的行号指示器。

我们知道对于一个处理器(如果是多核cpu那就是一核)，在一个确定的时刻都只会执行一条线程中的指令，一条线程中有多个指令，为了线程切换可以恢复到正确执行位置，每个线程都需有独立的一个程序计数器，不同线程之间的程序计数器互不影响，独立存储。

**注意：**如果线程执行的是个java方法，那么计数器记录字节码指令的地址。如果为native【底层方法】，那么计数器为空。这块内存区域是虚拟机规范中唯一没有OutOfMemoryError的区域。

**2.1.2、Java栈（虚拟机栈）**

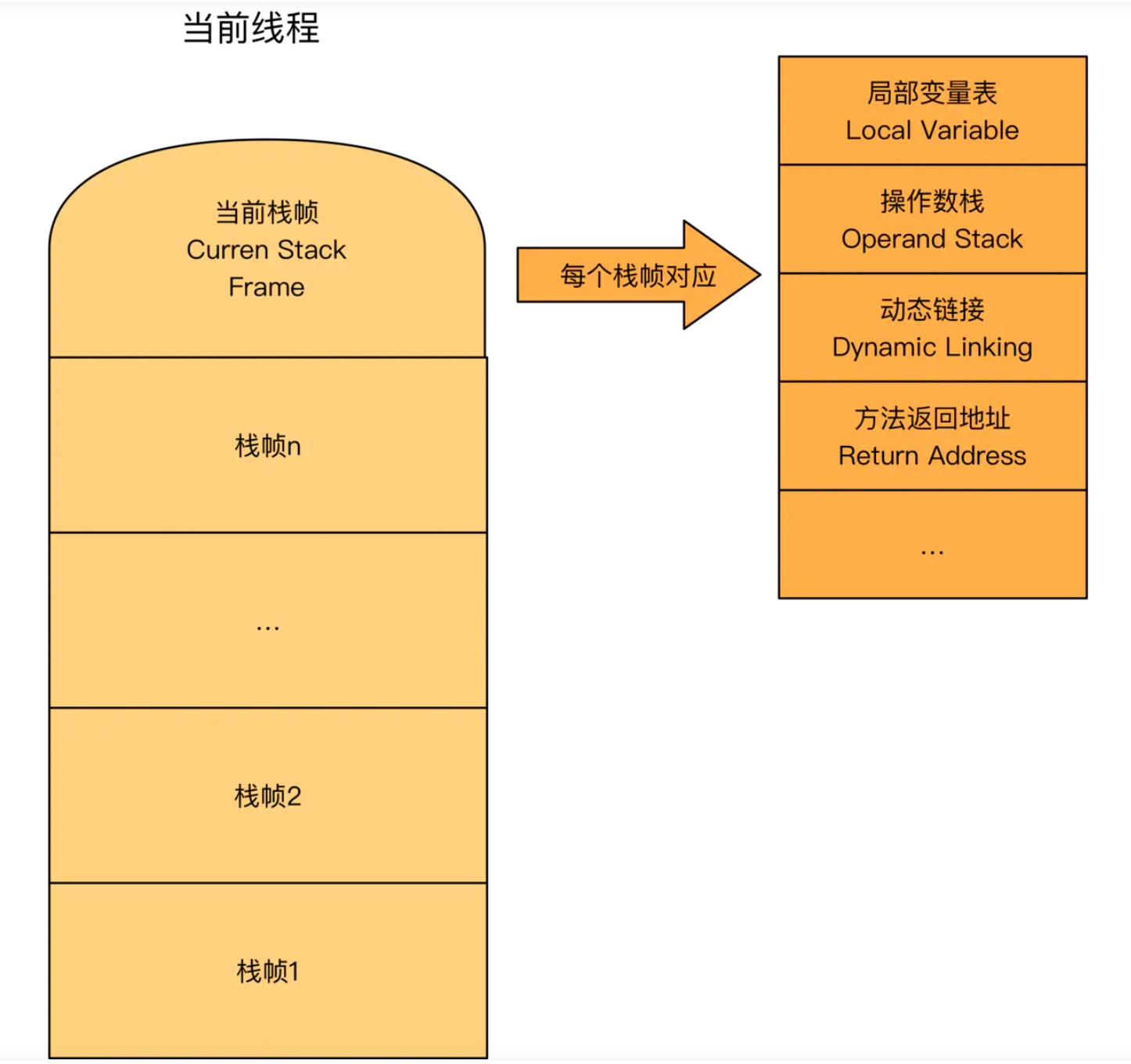
栈描述的是Java方法执行的内存模型，是线程私有的，配置java运行时的参数-Xss2m，来设置栈的最大内存为2m

每个方法被执行的时候都会创建一个栈帧用于存储局部变量表，操作栈，动态链接，方法出口等信息。每一个方法被调用的过程就对应一个栈帧在虚拟机栈中从入栈到出栈的过程。【栈先进后出】

**局部变量表:**一片连续的内存空间，用来存放方法参数，以及方法内定义的局部变量，存放着编译期间已知的数据类型(八大基本类型和对象引用(reference类型),returnAddress类型。最小的局部变量表空间单位为Slot，虚拟机没有指明Slot的大小，但在jvm中，long和double类型数据明确规定为64位，这两个类型占2个Slot，其它基本类型固定占用1个Slot。

**reference类型:**与基本类型不同的是它不等同本身，即使是String，内部也是char数组组成，它可能是指向一个对象起始位置指针，也可能指向一个代表对象的句柄或其他与该对象有关的位置。

**returnAddress类型**:指向一条字节码指令的地址



需要注意的是，局部变量表所需要的内存空间在编译期完成分配，当进入一个方法时，这个方法在栈中需要分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表大小。

Java虚拟机栈可能出现两种类型的异常：

（1）线程请求的栈深度大于虚拟机允许的栈深度，将抛出StackOverflowError异常。

（2）虚拟机栈空间可以动态扩展，当动态扩展无法申请到足够的空间时，将抛出OutOfMemory异常。

**2.1.3、本地方法栈**

本地方法栈和虚拟机栈发挥的作用十分相似，也是线程私有，区别是虚拟机栈执行的是Java方法(也就是字节码)服务，而本地方法栈则为虚拟机使用到的native方法服务，底层调用的c或者c++文件。

**2.1.4、Java堆**

对于大多数应用来说，堆是java虚拟机管理内存最大的一块内存区域，因为堆存放的对象是线程共享的，所以多线程的时候也需要同步机制。通过-Xms=-Xmx=4G，来设置java堆的内存大小

**java虚拟机规范对这块的描述是：**所有对象实例及数组都要在堆上分配内存，但随着JIT编译器（即时编译）的发展和逃逸分析技术（可以分配在栈上）的成熟，这个说法也不是那么绝对，但是大多数情况都是这样的。

**即时编译器：**可以把把Java的字节码，包括需要被解释的指令的程序转换成可以直接发送给处理器的指令的程序)

**逃逸分析：**通过逃逸分析来决定某些实例或者变量是否要在堆中进行分配，如果开启了逃逸分析，即可将这些变量直接在栈上进行分配，而非堆上进行分配。这些变量的指针可以被全局所引用，或者其其它线程所引用。

**注意：**它是所有线程共享的，它的目的是存放对象实例或者数组实例。同时它也是GC所管理的主要区域，因此常被称为GC堆，又由于现在收集器常使用分代算法，Java堆中还可以细分为新生代和老年代。

**2.1.5、方法区（JDK8后废弃，采用元空间）**

方法区（永久代）同Java堆一样，是所有线程共享的内存区域。

用于存储已被虚拟机加载的类信息、常量、静态变量、编译后的代码、运行时常量池中各种字面量和符号引用。

**2.1.6、堆外内存**

堆外内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用，而且也可能导致OutOfMemoryError 异常出现，所以我们放到这里一起讲解。在JDK 1.4 中新加入了NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O 方式，它可以使用Native 函数库直接分配堆外内存，然后通过一个存储在Java 堆里面的DirectByteBuffer（allocateDirect方法） 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java 堆和Native 堆中来回复制数据。

通过JVM参数可以指定堆外内存大小，-XX:MaxDirectMemorySize=512m

堆外内存的回收：

1、自动回收（受GC管控）

一旦DirectByteBuffer在堆内的弱引用对象被GC回收，那么就会由ReferenceHandler线程通过DirectByteBuffer中的Cleaner的clean方法（底层还是调用Unsafe.freeMemery）来释放堆外内存

2、手动回收（底层还是调用Unsafe.freeMemery）

（1）DirectByteBuffer.Cleaner.clean()

（2）system.gc()（触发Full-GC，触发GC的自动堆外内存回收）

（3）利用反射获取Unsafe实例，使用freeMemery来直接回收