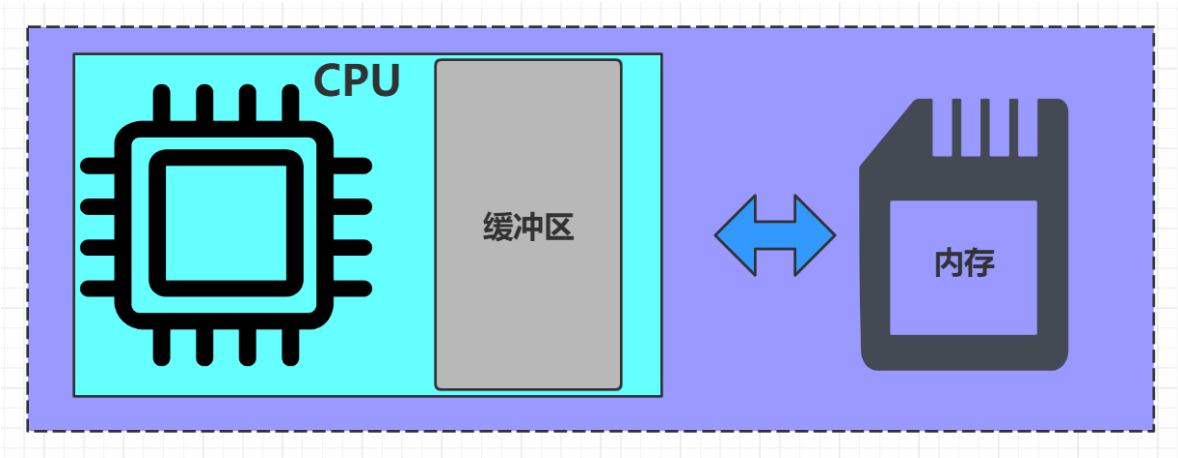
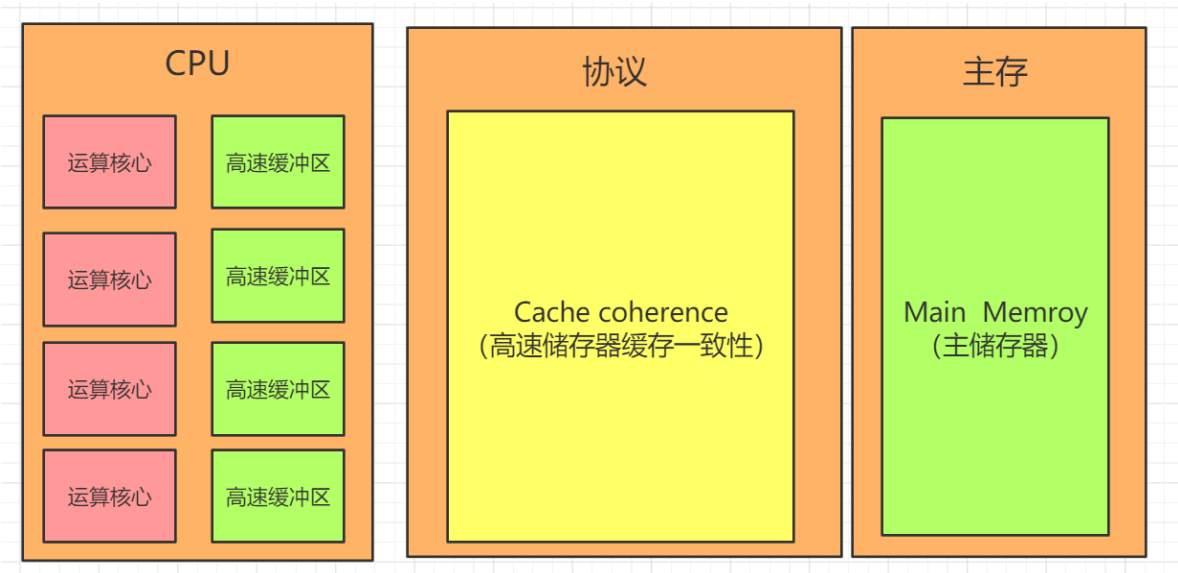
其实，我们知道的，我们的CPU跟内存会有非常频繁的交互，因为如果这个频繁的交互是交给我们的磁盘的话，那么随着我们的CPU运转速度越来越快，那么我们的磁盘的读写性能远远跟不上我们的CPU读写的速度，哪怕是我们现在的SSD，固态硬盘，也仅仅只是减少了我们的寻道时间以及加快了我们的找数据的时间。所以，我们才会在我们磁盘的基础上设计了内存，用来解决我们的单次IO时间过长导致我们CPU的等待成本过大的问题.但是随着我们CPU的发展,我们CPU的性能越来越高,哪怕就算是我们的内存的读写速度都跟不上我们的CPU的读写速度.

因此,这个时候,我们的CPU厂商就想了个办法:在我们的每颗CPU上都加入了高速缓冲区,用来加快我们的读写速度,于是乎,我们的CPU跟我们的内存的交互就演变成为了这样子的一张图片.



但是,根据摩尔定律,我们的IC芯片每隔18个月能容纳的晶体管会翻倍,但是我们的毕竟不可能不限制的增长,单核CPU的主频也有性能瓶颈,想要提升性能,必须增加多个运算核心.所以,随着时间的增长,我们的多核时代来临了.

基于高速缓存的存储交互很好的解决了处理器与内存之间的矛盾，也引入了新的问题：缓存一致性问题。在多处理器系统中，每个处理器有自己的高速缓存，而他们又共享同一块内存（下文成主存，main memory 主要内存），当多个处理器运算都涉及到同一块内存区域的时候，就有可能发生缓存不一致的现象。为了解决这一问题，需要各个处理器运行时都遵循一些协议，在运行时需要通过这些协议保证数据的一致性。比如MSI、MESI、MOSI、Synapse、Firely、DragonProtocol等。那么怎么做的呢?



而我们的运行时数据区其实也保有了这样子的一种设计.

其实参照这种设置,我们已经能够推到出我们的JVM是如何跟我们的内存还有我们的CPU交互的了.

java中使用的是多线程机制,那么必然会有多个任务同时执行,这个时候类比了我们的CPU运算核心,那么必然会有一块区域或者一种操作能够保证我们数据的一致性,那么我们的JVM内存中数据存放的部分必然会是所有线程都能够获取到的,那么就可以称之为线程共享,而每个线程又有自己单独的工作内存,当我们线程进行运作时,数据肯定会从JVM主存拷贝到线程自己的工作内存,然后再进行操作.

常量池

常量池分为我们前面所说过的静态常量池，运行时常量池，还有字符串常量池，那么其实我们的运行时常量池又是什么呢?

静态常量池

> 其实储存的就是字面量以及符号引用

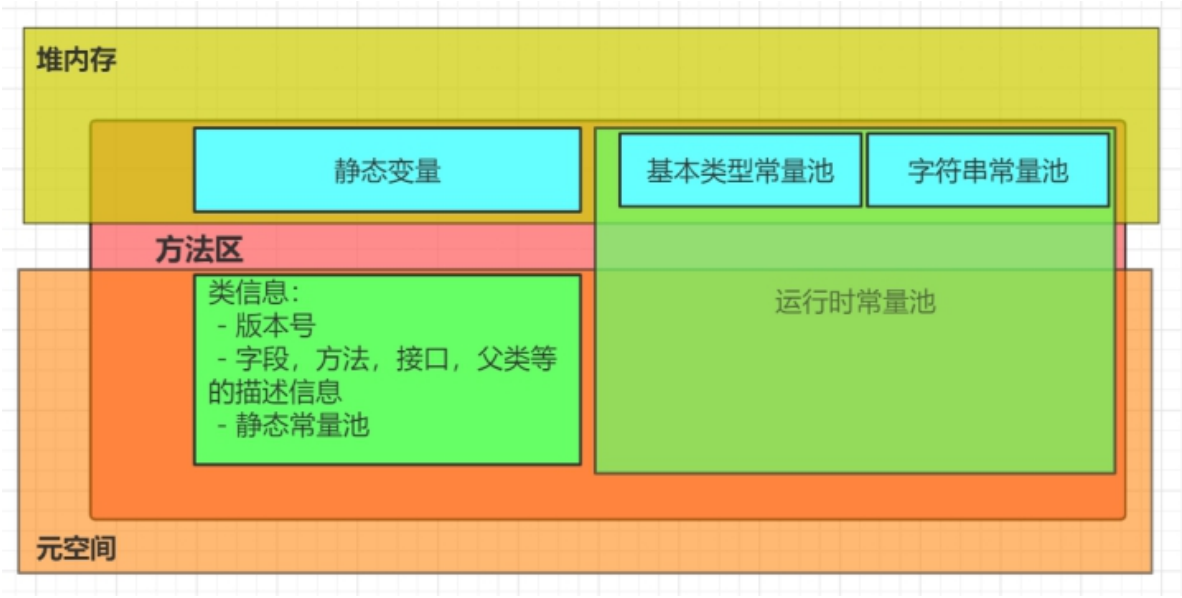
运行时常量池

> 运行时常量池就是我们的每个类以及每个接口在我们的JVM进行run的过程中所在内存中开辟出来的一块用来储存我们静态常量池部分数据的一块特殊区域。

字符串常量池

> 包含在动态常量池里

JDK1.8中各常量池在内存中的划分



2.3 运行时数据区(Run-Time Data Areas)

> 在装载阶段的第(2),(3)步可以发现有运行时数据，堆，方法区等名词
> (2)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
> (3)在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口
> 说白了就是类文件被类装载器装载进来之后，类中的内容(比如变量，常量，方法，对象等这些数据得要有个去处，也就是要存储起来，存储的位置肯定是在JVM中有对应的空间)

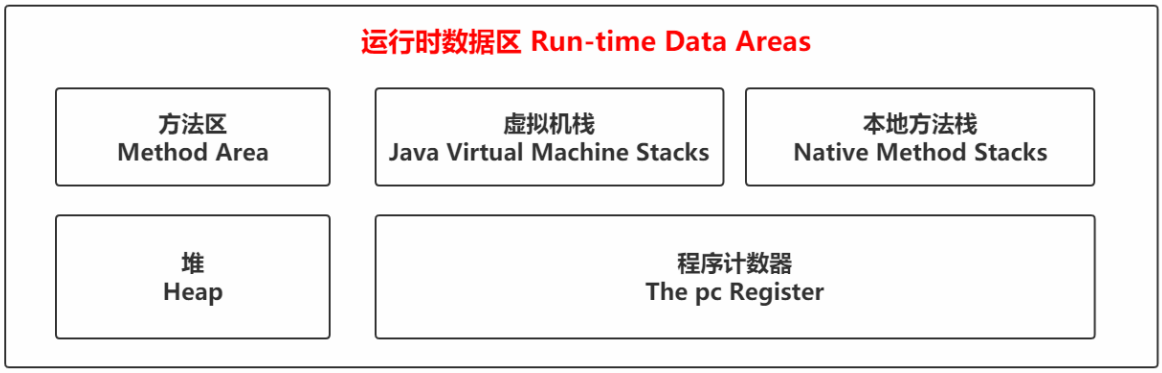
2.3.1 官网概括

官网：https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

```
The Java Virtual Machine defines various run-time data areas that are used
during execution of a program. Some of these data areas are created on Java
Virtual Machine start-up and are destroyed only when the Java Virtual Machine
exits. Other data areas are per thread. Per-thread data areas are created when a
thread is created and destroyed when the thread exits.
```

2.3.2 图解

```
Each run-time constant pool is allocated from the Java Virtual Machine's method
area (§2.5.4).s
```



2.3.3 初步认识

2.3.3.1 Method Area(方法区)

（1）方法区是各个线程共享的内存区域，在虚拟机启动时创建

```
The Java Virtual Machine has a method area that is shared among all Java Virtual
Machine threads.
The method area is created on virtual machine start-up.
```

（2）虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却又一个别名叫做Non-Heap(非堆)，目的是与Java堆区分开来

```
Although the method area is logically part of the heap,......
```

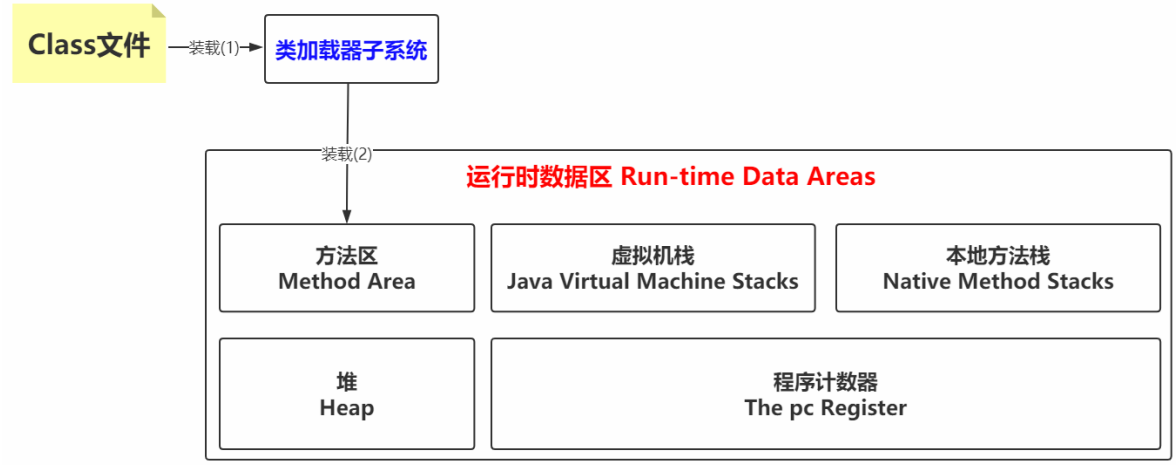
（3）用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据

```
It stores per-class structures such as the run-time constant pool, field and
method data, and the code for methods and constructors, including the special
methods (§2.9) used in class and instance initialization and interface
initialization.
```

（4）当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常

```
If memory in the method area cannot be made available to satisfy an allocation
request, the Java Virtual Machine throws an OutOfMemoryError.
```

> 此时回看装载阶段的第2步，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
>
> 如果这时候把从Class文件到装载的第(1)和(2)步合并起来理解的话，可以画个图



> **值得说明的**
>
> JVM运行时数据区是一种规范，真正的实现
> 在JDK 8中就是Metaspace，在JDK6或7中就是Perm Space

2.3.3.2 Heap(堆)

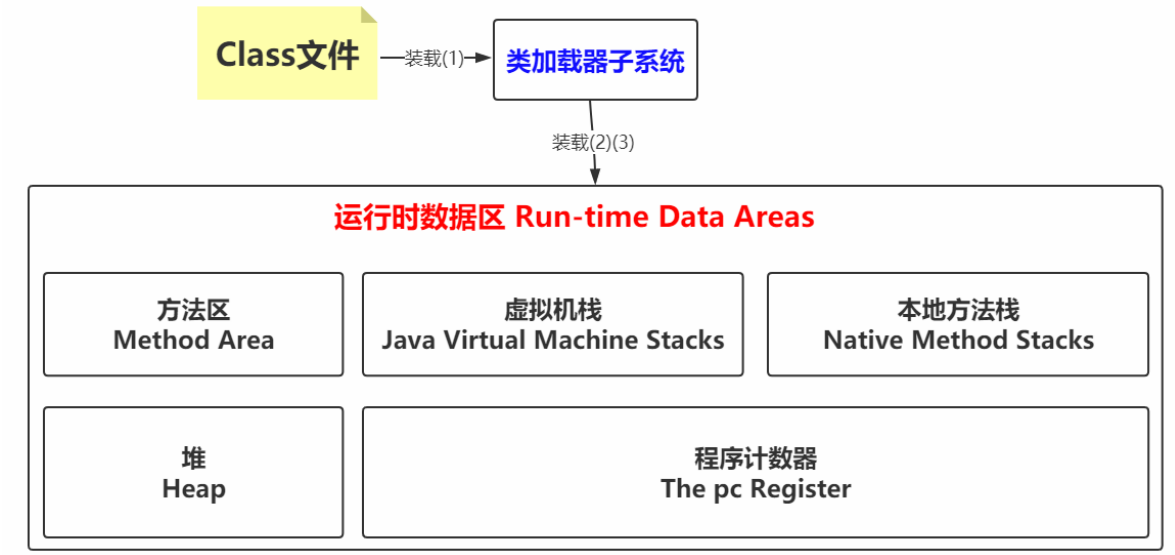
（1）Java堆是Java虚拟机所管理内存中最大的一块，在虚拟机启动时创建，被所有线程共享。

（2）Java对象实例以及数组都在堆上分配。

```
The Java Virtual Machine has a heap that is shared among all Java Virtual
Machine threads. The heap is the run-time data area from which memory for all
class instances and arrays is allocated.
The heap is created on virtual machine start-up.
```

> 此时回看装载阶段的第3步，在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口

此时装载(1)(2)(3)的图可以改动一下



2.3.3.3 Java Virtual Machine Stacks(虚拟机栈)

> 经过上面的分析，类加载机制的装载过程已经完成，后续的链接，初始化也会相应的生效。

> 假如目前的阶段是初始化完成了，后续做啥呢？肯定是Use使用咯，不用的话这样折腾来折腾去有什么意义？那怎样才能被使用到？换句话说里面内容怎样才能被执行？比如通过主函数main调用其他方法，这种方式实际上是main线程执行之后调用的方法，即要想使用里面的各种内容，得要以线程为单位，执行相应的方法才行。**那一个线程执行的状态如何维护？一个线程可以执行多少个方法？这样的关系怎么维护呢？**

（1）虚拟机栈是一个线程执行的区域，保存着一个线程中方法的调用状态。换句话说，一个Java线程的运行状态，由一个虚拟机栈来保存，所以虚拟机栈肯定是线程私有的，独有的，随着线程的创建而创建。

```
Each Java Virtual Machine thread has a private Java Virtual Machine stack,
created at the same time as the thread.
```

（2）每一个被线程执行的方法，为该栈中的栈帧，即每个方法对应一个栈帧。

调用一个方法，就会向栈中压入一个栈帧；一个方法调用完成，就会把该栈帧从栈中弹出。

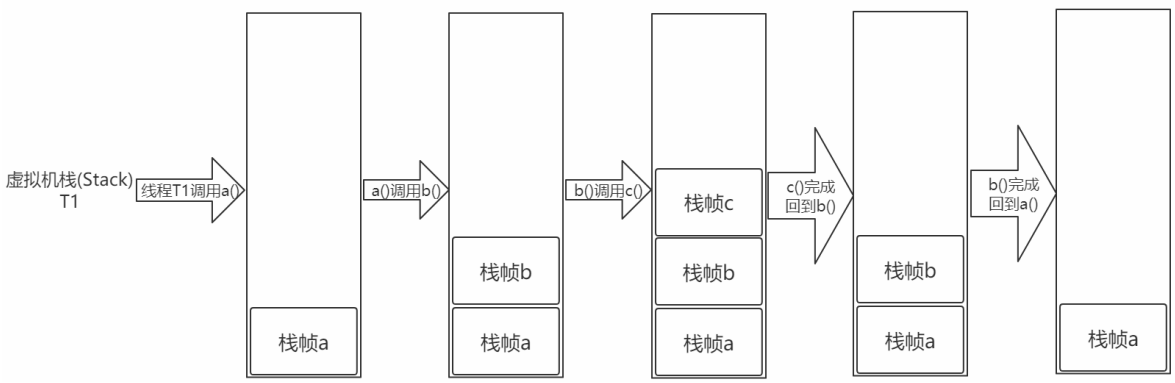
```
 A Java Virtual Machine stack stores frames (§2.6).
```

```
A new frame is created each time a method is invoked. A frame is destroyed when
its method invocation completes.
```

- **图解栈和栈帧**

```
void a(){
    b();
}
void b(){
    c();
}
void c(){

}
```



- **栈帧**

官网：https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.6

栈帧：每个栈帧对应一个被调用的方法，可以理解为一个方法的运行空间。

每个栈帧中包括局部变量表(Local Variables)、操作数栈(Operand Stack)、指向运行时常量池的引用(A reference to the run-time constant pool)、方法返回地址(Return Address)和附加信息。

局部变量表:方法中定义的局部变量以及方法的参数存放在这张表中
局部变量表中的变量不可直接使用，如需要使用的话，必须通过相关指令将其加载至操作数栈中作为操作数使用。

操作数栈:以压栈和出栈的方式存储操作数的

动态链接:每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接(Dynamic Linking)。

方法返回地址:当一个方法开始执行后,只有两种方式可以退出，一种是遇到方法返回的字节码指令；一种是遇见异常，并且这个异常没有在方法体内得到处理。

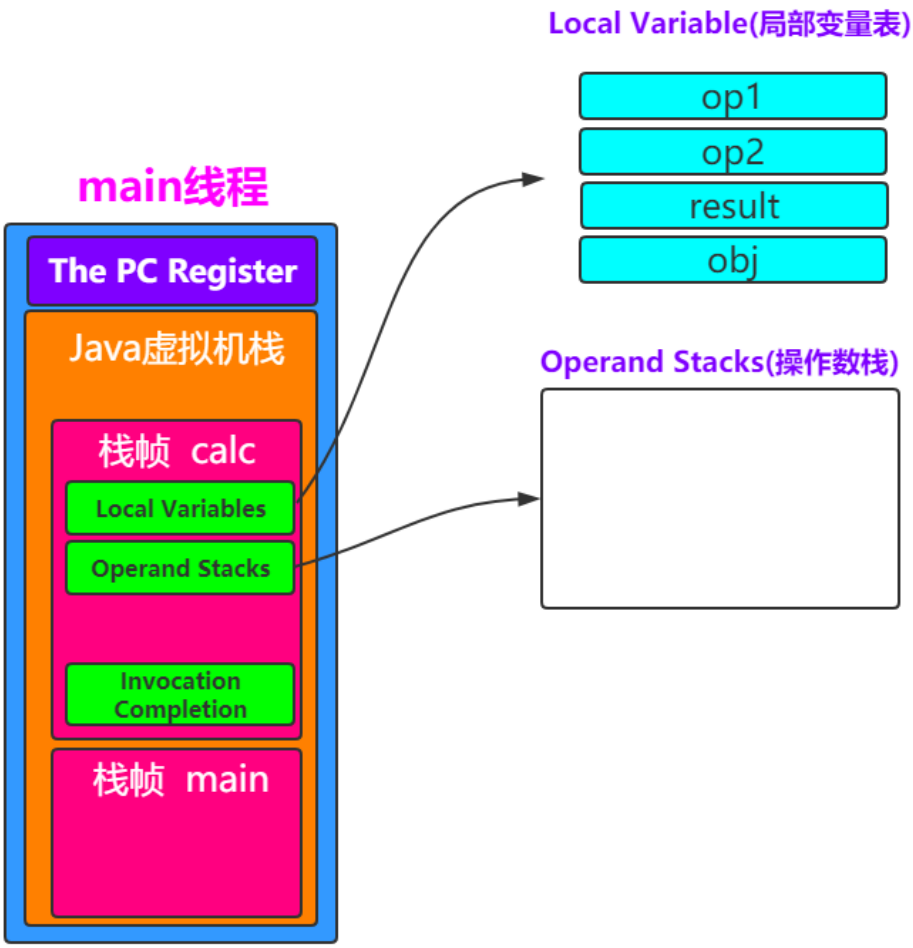
- **结合字节码指令理解栈帧**

> javap -c Person.class > Person.txt

```
Compiled from "Person.java"
class Person {
...
 public static int calc(int, int);
  Code:
   0: iconst_3   //将int类型常量3压入[操作数栈]
   1: istore_0   //将int类型值存入[局部变量0]
   2: iload_0    //从[局部变量0]中装载int类型值入栈
   3: iload_1    //从[局部变量1]中装载int类型值入栈
   4: iadd     //将栈顶元素弹出栈，执行int类型的加法，结果入栈
   5: istore_2   //将栈顶int类型值保存到[局部变量2]中
   6: iload_2    //从[局部变量2]中装载int类型值入栈
   7: ireturn    //从方法中返回int类型的数据
...
}
```

思考：index的值是0还是1

```
On class method invocation, any parameters are passed in consecutive local
variables starting from local variable 0. On instance method invocation, local
variable 0 is always used to pass a reference to the object on which the
instance method is being invoked (this in the Java programming language). Any
parameters are subsequently passed in consecutive local variables starting from
local variable 1.
```



2.3.3.4 The pc Register(程序计数器)

> 我们都知道一个JVM进程中有多个线程在执行，而线程中的内容是否能够拥有执行权，是根据CPU调度来的。
>
> 假如线程A正在执行到某个地方，突然失去了CPU的执行权，切换到线程B了，然后当线程A再获得CPU执行权的时候，怎么能继续执行呢？这就是需要在线程中维护一个变量，记录线程执行到的位置。

如果线程正在执行Java方法，则计数器记录的是正在执行的虚拟机字节码指令的地址；

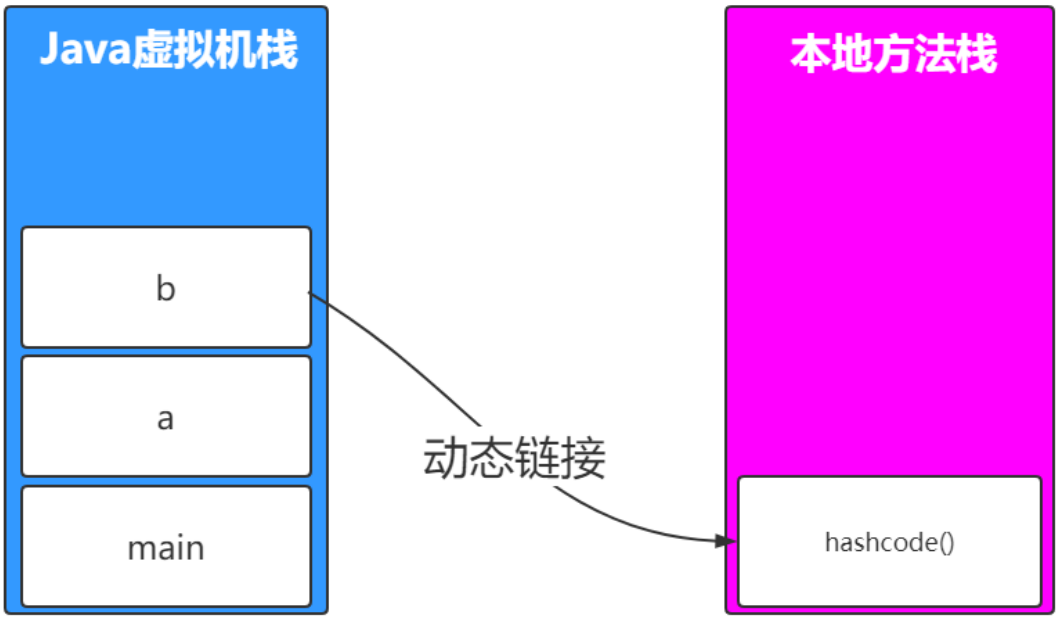
如果正在执行的是Native方法，则这个计数器为空。

```
The Java Virtual Machine can support many threads of execution at once (JLS §17). Each Java Virtual Machine thread has its own pc (program counter) register. At any point, each Java Virtual Machine thread is executing the code of a single method, namely the current method (§2.6) for that thread. If that method is not native, the pc register contains the address of the Java Virtual Machine instruction currently being executed. If the method currently being executed by the thread is native, the value of the Java Virtual Machine's pc register is undefined. The Java Virtual Machine's pc register is wide enough to hold a returnAddress or a native pointer on the specific platform.
```

2.3.3.5 Native Method Stacks(本地方法栈)

如果当前线程执行的方法是Native类型的，这些方法就会在本地方法栈中执行。

那如果在Java方法执行的时候调用native的方法呢?



除了上面五块内存之外,其实我们的JVM还会使用到其他两块内存

- 直接内存（Direct Memory）

  并不是虚拟机运行时数据区的一部分，也不是JVM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 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。

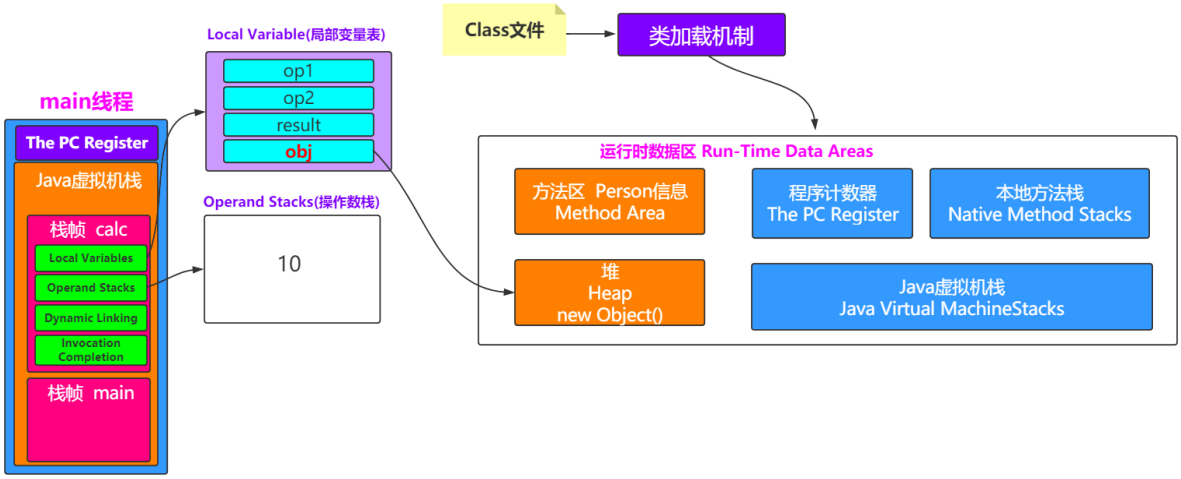
  本机直接内存的分配不会受到Java 堆大小的限制，但是，既然是内存，则肯定还是会受到本机总内存的大小及处理器寻址空间的限制。因此在分配JVM空间的时候应该考虑直接内存所带来的影响，特别是应用到NIO的场景。

- 其他内存:
  - Code Cache：**JVM本身是个本地程序，还需要其他的内存去完成各种基本任务，比如，JIT编译器在运行时对热点方法进行编译，就会将编译后的方法储存在Code Cache里面；GC等功能。需要运行在本地线程之中，类似部分都需要占用内存空间。这些是实现JVM JIT等功能的需要，但规范中并不涉及。

2.3.4 折腾一下

2.3.4.1 栈指向堆

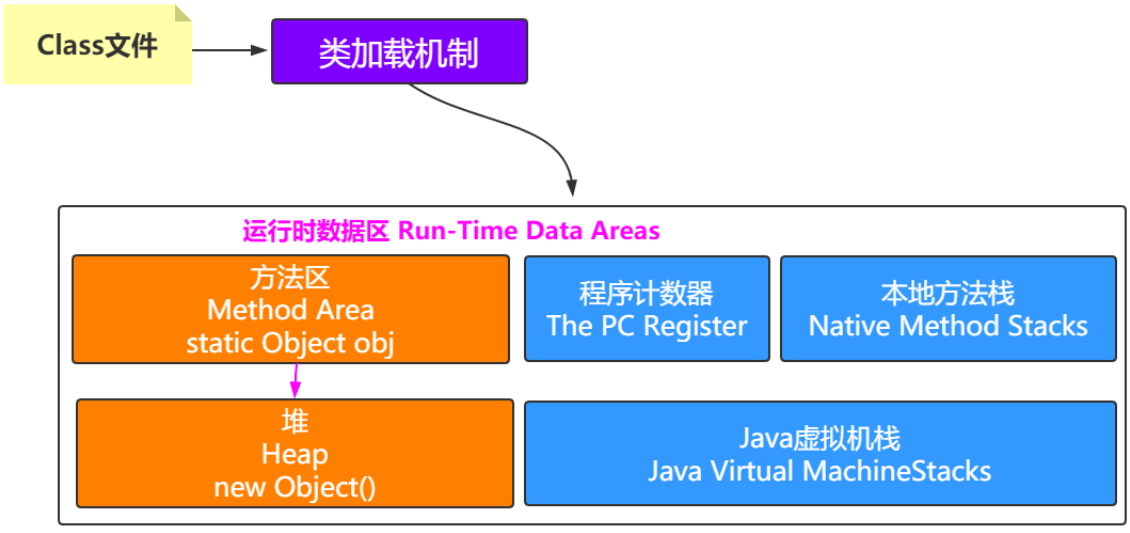
如果在栈帧中有一个变量，类型为引用类型，比如Object obj=new Object()，这时候就是典型的栈中元素指向堆中的对象。



2.3.4.2 方法区指向堆

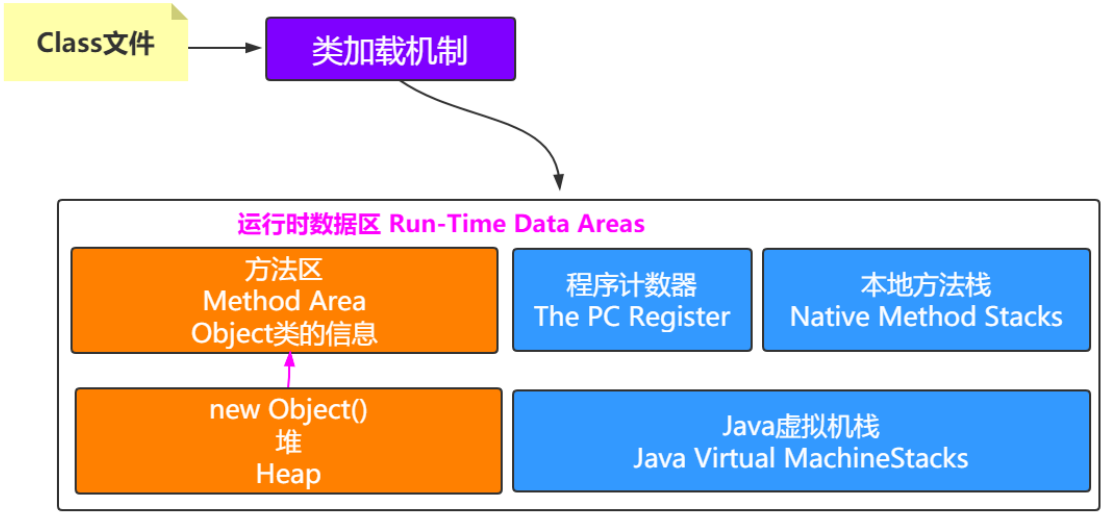
方法区中会存放静态变量，常量等数据。如果是下面这种情况，就是典型的方法区中元素指向堆中的对象。

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



2.3.4.3 堆指向方法区

What？堆还能指向方法区？
注意，方法区中会包含类的信息，堆中会有对象，那怎么知道对象是哪个类创建的呢？

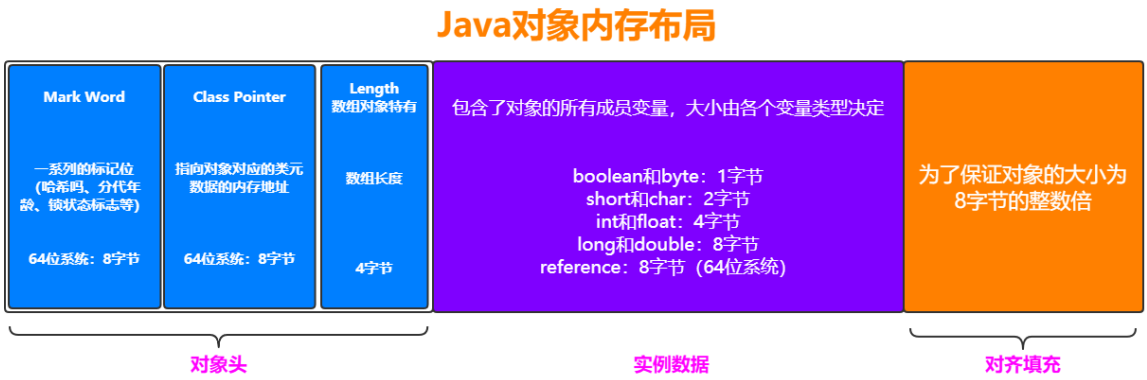


思考：

一个对象怎么知道它是由哪个类创建出来的？怎么记录？这就需要了解一个Java对象的具体信息咯。

2.3.4.4 Java对象内存模型

> 一个Java对象在内存中包括3个部分：对象头、实例数据和对齐填充



验证hashCode的储存方式

> 使用到我们的jol工具

依赖：

```
<dependency>
            <groupId>org.openjdk.jol</groupId>
            <artifactId>jol-core</artifactId>
            <version>RELEASE</version>
</dependency>
```

实体类：

```
package com.example.jvmcase.domain;


import lombok.Data;
import org.openjdk.jol.info.ClassLayout;
```

```
public class Worker {
    private Integer id;
    private String username;
    private String password;

    public Integer getId() {
        return id;
    }

    public String getPassword() {
        return password;
    }

    public String getUsername() {
        return username;
    }


    public void setUsername(String username) {
        this.username = username;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    @Override
    public String toString() {
        return super.toString();
    }

    public static void printf(Worker p) {
        // 查看对象的整体结构信息
        // JOL工具类
        System.out.println(ClassLayout.parseInstance(p).toPrintable());
    }
}
```

测试代码：

```
package com.example.jvmcase.basic;

import com.example.jvmcase.domain.Worker;


public class Test {
    public static void main(String[] args) {

        Worker work = new Worker();
        System.out.println(work);
        Worker.printf(work);
        System.out.println(work.hashCode());
```

```
    }
}
```

测试结果：

```
com.example.jvmcase.domain.Worker@6acbcfc0
com.example.jvmcase.domain.Worker object internals:
 OFFSET  SIZE               TYPE DESCRIPTION                               VALUE
      0     4                    (object header)                           01
c0 cf cb (00000001 11000000 11001111 11001011) (-875577343)
      4     4                    (object header)                           6a
00 00 00 (01101010 00000000 00000000 00000000) (106)
      8     4                    (object header)                           43
c1 00 f8 (01000011 11000001 00000000 11111000) (-134168253)
     12     4   java.lang.Integer Worker.id                                 null
     16     4    java.lang.String Worker.username                           null
     20     4    java.lang.String Worker.password                           null
Instance size: 24 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

1791741888
```

> 1791741888这个数字是我们的HashCode值，转换成16进制可得6a cb cf c0，经过对比。

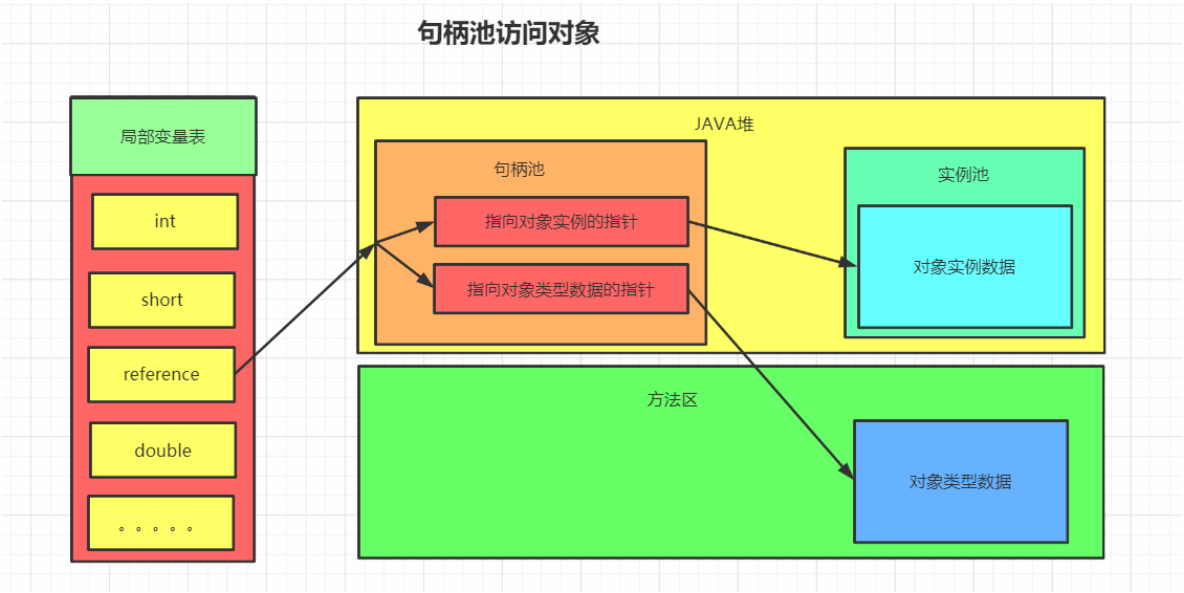
由此可得，我们的哈希码使用的大端储存。

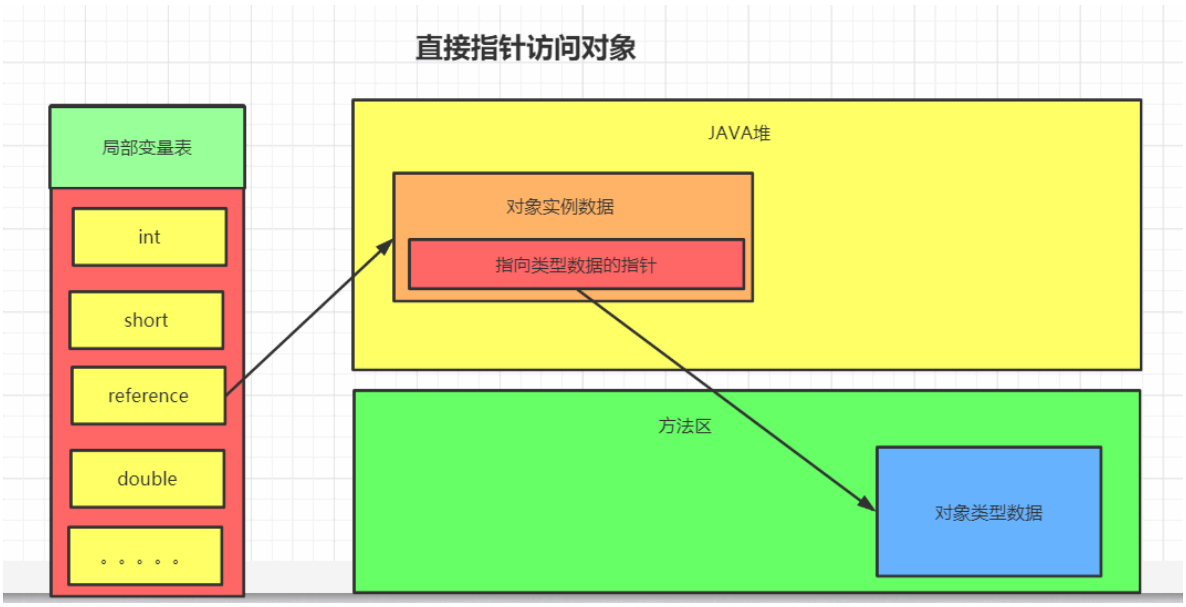
```
例如：
十进制数9877，如果用小端存储表示则为：
高地址 <- - - - - - - - 低地址
10010101`[高序字节]` 00100110`[低序字节]`
用大端存储表示则为：
高地址 <- - - - - - - - 低地址
00100110`[低序字节]` 10010101`[高序字节]`
```

> 小端存储：便于数据之间的类型转换，例如：long类型转换为int类型时，高地址部分的数据可以直接截掉。
> 大端存储：便于数据类型的符号判断，因为最低地址位数据即为符号位，可以直接判断数据的正负号。

Class Pointer

> 引用定位到对象的方式有两种，一种叫句柄池访问，一种叫直接访问





区别：

句柄池：

使用句柄访问对象，会在堆中开辟一块内存作为句柄池，句柄中储存了对象实例数据（属性值结构体）的内存地址，访问类型数据的内存地址（类信息，方法类型信息），对象实例数据一般也在heap中开辟，类型数据一般储存在方法区中。

优点：reference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本身不需要改变。

缺点：增加了一次指针定位的时间开销。

直接访问：

直接指针访问方式指reference中直接储存对象在heap中的内存地址，但对应的类型数据访问地址需要在实例中存储。

优点：节省了一次指针定位的开销。

缺点：在对象被移动时(如进行GC后的内存重新排列)，reference本身需要被修改。

指针压缩：

在32位系统中，类型指针为4字节32位，在64位系统中类型指针为8字节64位，**但是JVM会默认的进行指针压缩**，所以我们上图输出结果中类型指针也是4字节32位。如果我们关闭指针压缩的话，就可以看到64位的类型指针了，所以我们通常在部署服务时，JVM内存不要超过32G，因为超过32G就无法开启指针压缩了

> 关闭指针压缩： -XX:+UseCompressedOops

对齐填充

> 没有对齐填充就可能会存在数据跨内存地址区域存储的情况

在没有对齐填充的情况下，内存地址存放情况如下：

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	
boolean	int				char		long								

因为处理器只能0x00-0x07，0x08-0x0F这样读取数据，所以当我们想获取这个long型的数据时，处理器必须要读两次内存，第一次(0x00-0x07)，第二次(0x08-0x0F)，然后将两次的结果才能获得真正的数值。

那么在有对齐填充的情况下，内存地址存放情况是这样的：

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
0	0	0	0	1	0	1		0	0	0	0	0	0	0	1
boolean	int				char			long							

现在处理器只需要直接一次读取(0x08-0x0F)的内存地址就可以获得我们想要的数据了。

对齐填充存在的意义就是为了提高CPU访问数据的效率，这是一种以空间换时间的做法；正如我们所见，虽然访问效率提高了（减少了内存访问次数），但是在0x07处产生了1bit的空间浪费。