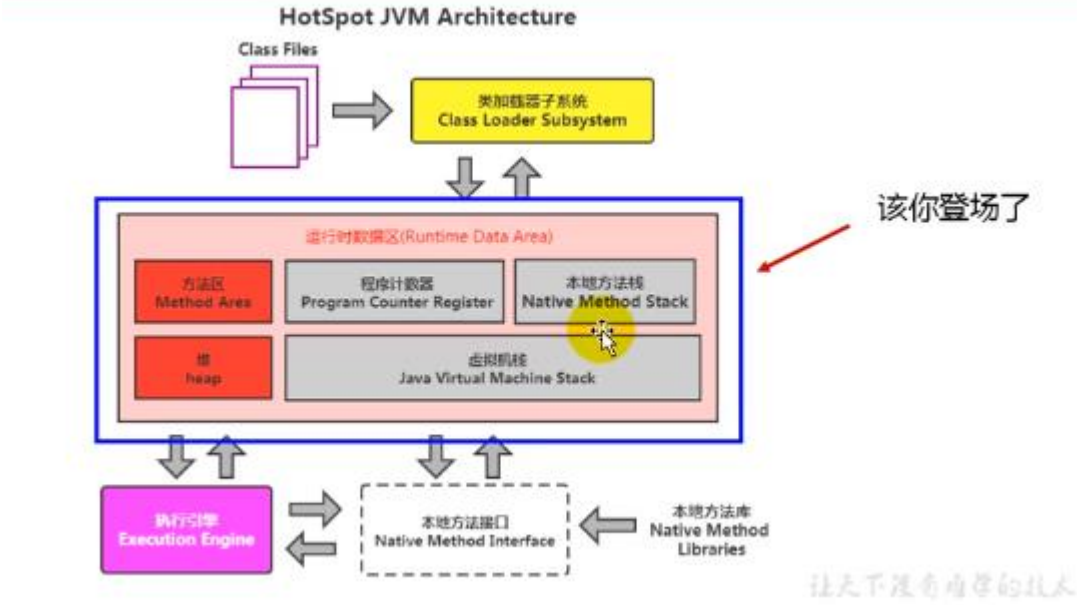
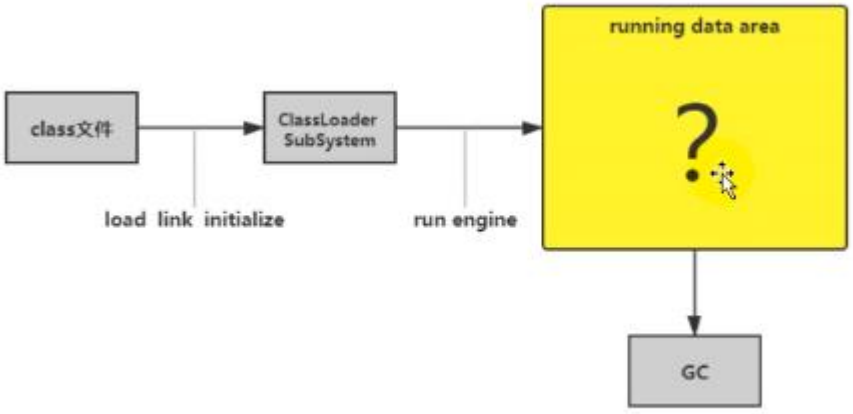
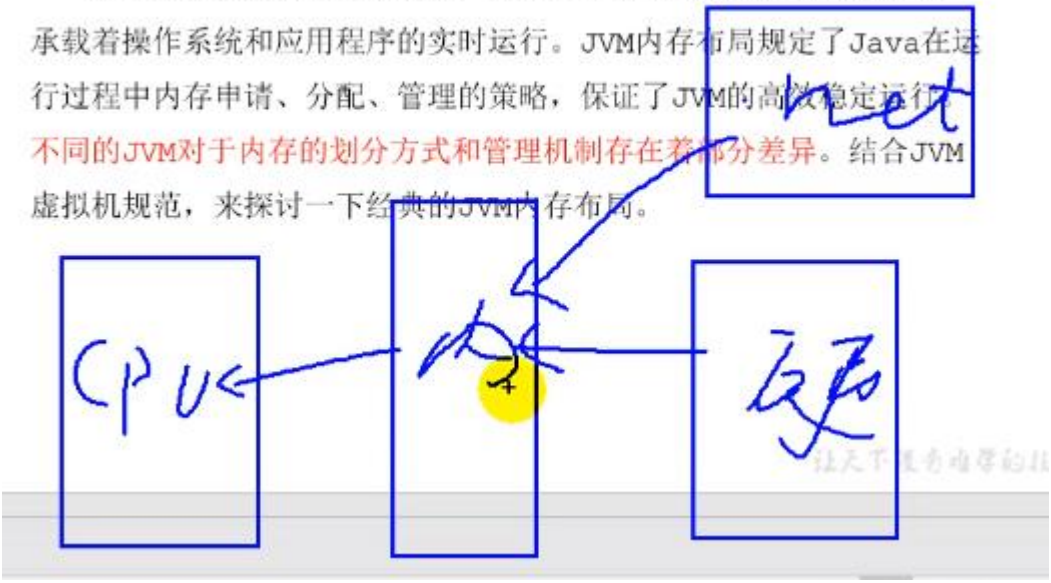
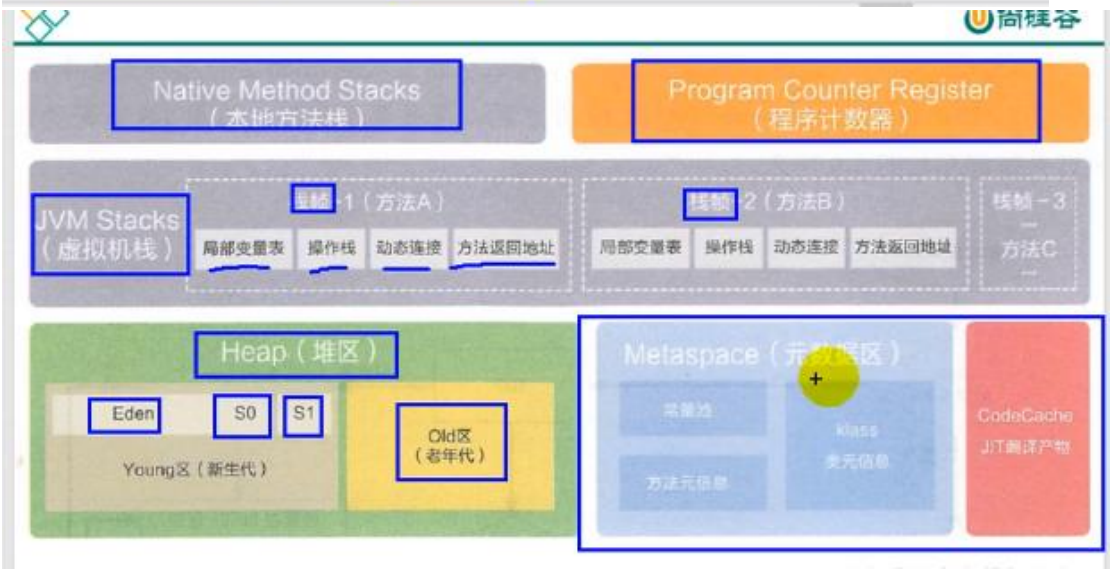
03-运行时数据区概述及线程

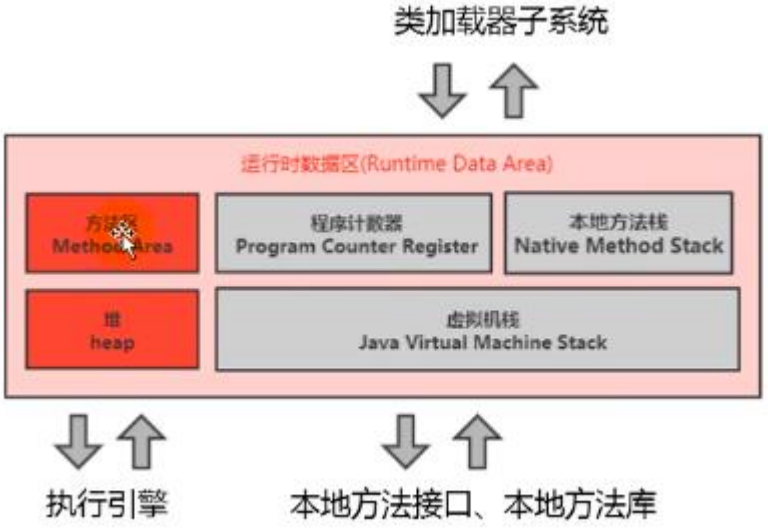


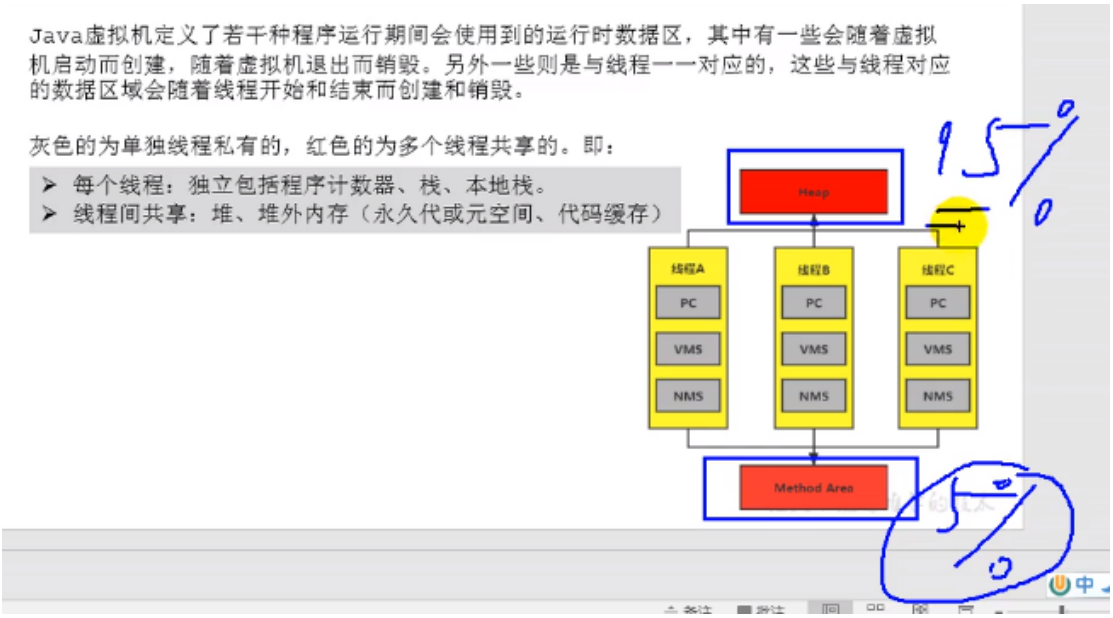


内存是非常重要的系统资源，是硬盘和CPU 的中间仓库及桥梁，承载着操作系统和应用程序的实时运行。JVM内存布局规定了Java在运行过程中内存申请、分配、管理的策略，保证了JVM的高效稳定运行。不同的JVM对于内存的划分方式和管理机制存在着部分差异。结合JVM虚拟机规范，来探讨一下经典的JVM内存布局。









Threads will share the heap and the method area with each other

Class Runtime

java.lang.Object
    java.lang.Runtime

```
public class Runtime
extends Object
```

Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime method.

An application cannot create its own instance of this class.

每个JVM只有一个Runtime实例。即为运行时环境，相当于内存结构的中间的那个框框：运行时环境。

one JVM corresponds to only one Runtime entity

2- 线 程

- 线程是一个程序里的运行单元。JVM允许一个应用有多个线程并行的执行。

- 在Hotspot JVM里，每个线程都与操作系统的本地线程直接映射。
  - 当一个Java线程准备好执行以后，此时一个操作系统的本地线程也同时创建。Java线程执行终止后，本地线程也会回收。

- 操作系统负责所有线程的安排调度到任何一个可用的CPU上。一旦本地线程初始化成功，它就会调用Java线程中的run()方法。

•守护线程、普通线程

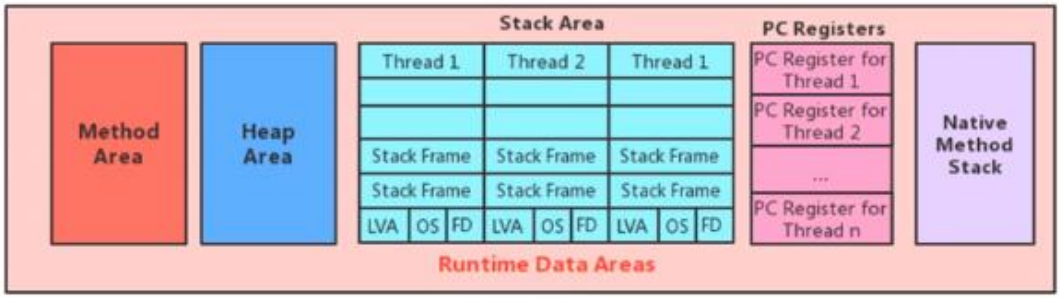
- 如果你使用jconsole或者是任何一个调试工具，都能看到在后台有许多线程在运行。这些后台线程不包括调用public static void main(String[])的main线程以及所有这个main线程自己创建的线程。

- 这些主要的后台系统线程在Hotspot JVM里主要是以下几个：
  - **虚拟机线程**：这种线程的操作是需要JVM达到安全点才会出现。这些操作必须在不同的线程中发生的原因是他们都需要JVM达到安全点，这样堆才不会变化。这种线程的执行类型包括"stop-the-world"的垃圾收集，线程栈收集，线程挂起以及偏向锁撤销。
  - **周期任务线程**：这种线程是时间周期事件的体现(比如中断)，他们一般用于周期性操作的调度执行。
  - **GC线程**：这种线程对在JVM里不同种类的垃圾收集行为提供了支持。
  - **编译线程**：这种线程在运行时会将字节码编译成到本地代码。
  - **信号调度线程**：这种线程接收信号并发送给JVM，在它内部通过调用适当的方法进行处理。

04-程序计数器(PC寄存器)

PC Register介绍

尚硅谷

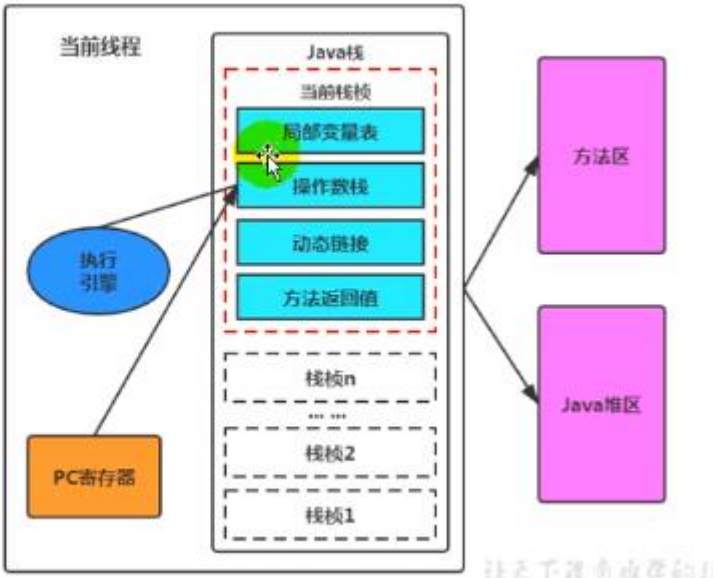


JVM中的程序计数寄存器（Program Counter Register）中， Register 的命名源于CPU的寄存器，寄存器存储指令相关的现场信息。 CPU只有把数据装载到寄存器才能够运行。

这里，并非是广义上所指的物理寄存器，或许将其翻译为PC计数器（或指令计数器）会更加贴切(也称为程序钩子) ，并且也不容易引起一些不必要的误会。JVM中的PC寄存器是对物理PC寄存器的一种抽象模拟。

作用：

PC寄存器用来存储指向下一条指令的地址，也即将要执行的指令代码。由执行引擎读取下一条指令。



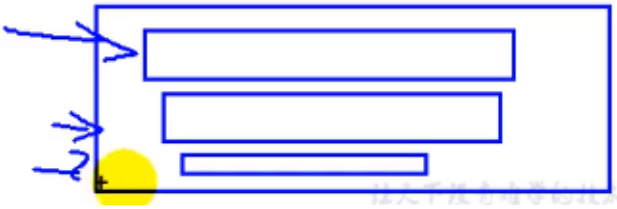
one stack frame represents one function

- 它是一块很小的内存空间，几乎可以忽略不记。也是运行速度最快的存储区域。
- 在JVM规范中，每个线程都有它自己的程序计数器，是线程私有的，生命周期与线程的生命周期保持一致。
- 任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。程序计数器会存储当前线程正在执行的Java方法的JVM指令地址；或者，如果是在执行native方法，则是未指定值（undefned）。

PC Register介绍

尚硅谷

- 它是程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。
- 字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。
- 它是唯一一个在Java 虚拟机规范中没有规定任何OutOtMemoryError情况的区域。



Next feature of PC is about OOM and GC.

Method area\heap have GC,but PC register\JVM stack have no GC.

PC doesn’t have the warning of out of memory,others except PC may overflow,but they have OOM

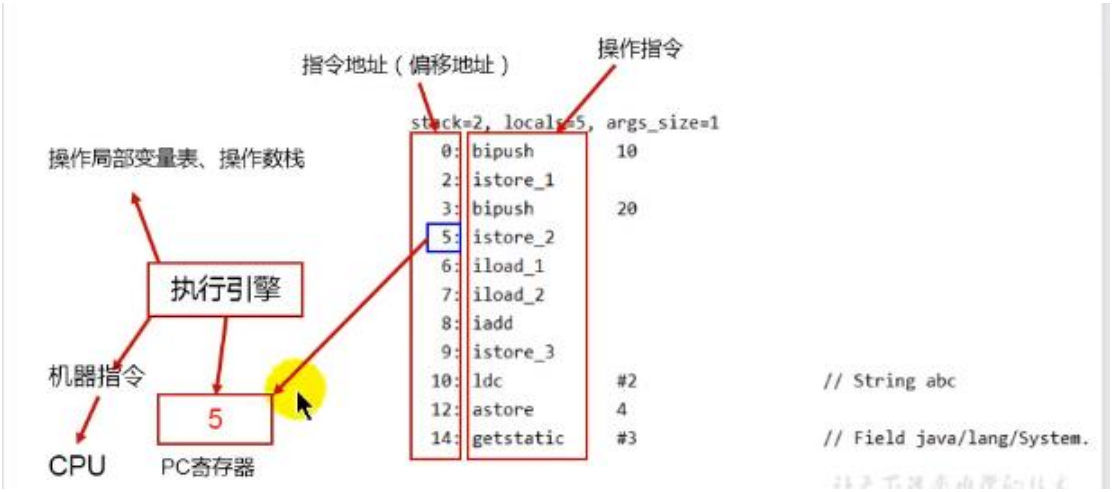
2- 举例说明

举例说明

```
public int minus(){
    int c = 3;
    int d = 4;
    return c - d;
}
```

字节码文件

```
public int minus();
  descriptor: ()I
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=1
       0: iconst_3
       1: istore_1
       2: iconst_4
       3: istore_2
       4: iload_1
       5: iload_2
       6: isub
       7: ireturn
    LineNumberTable:
      line 15: 0
      line 16: 2
      line 17: 4
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       8     0  this   Lcom/atguigu/java/PCRegisterTest;
          2       6     1     c   I
```



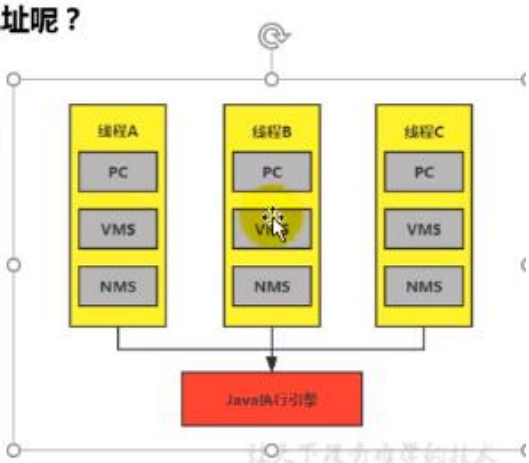
两个常见问题

使用PC寄存器存储字节码指令地址有什么用呢？

为什么使用PC寄存器记录当前线程的执行地址呢？

因为CPU需要不停的切换各个线程，这时候切换回来以后，就得知道接着从哪开始继续执行。

JVM的字节码解释器就需要通过改变PC寄存器的值来明确下一条应该执行什么样的字节码指令。

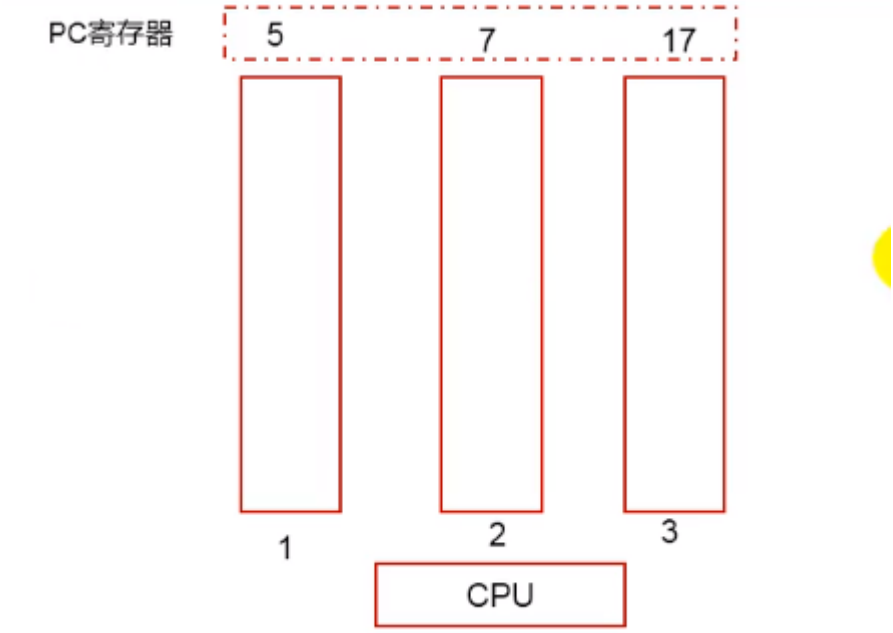


PC寄存器为什么会被设定为线程私有？

我们都知道所谓的多线程在一个特定的时间段内只会执行其中某一个线程的方法，CPU会不停地做任务切换，这样必然导致经常中断或恢复，如何保证分毫无差呢？为了能够准确地记录各个线程正在执行的当前字节码指令地址，最好的办法自然是为每一个线程都分配一个PC寄存器，这样一来各个线程之间便可以进行独立计算，从而不会出现相互干扰的情况。

由于CPU时间片轮限制，众多线程在并发执行过程中，任何一个确定的时刻，一个处理器或者多核处理器中的一个内核，只会执行某个线程中的一条指令。

这样必然导致经常中断或恢复，如何保证分毫无差呢？每个线程在创建后，都会产生自己的程序计数器和栈帧，程序计数器在各个线程之间互不影响。



CPU 时间片

尚硅谷

CPU 时间片即 CPU 分配给各个程序的时间，每个线程被分配一个时间段，称作它的时间片。

在宏观上：我们可以同时打开多个应用程序，每个程序并行不悖，同时运行。

但在微观上：由于只有一个 CPU，一次只能处理程序要求的一部分，如何处理公平，一种方法就是引入时间片，每个程序轮流执行。

并行：vs 串行
并发：

05-虚拟机栈

1. 虚拟机栈概述
2. 栈的存储单位
3. 局部变量表
4. 操作数栈
5. 代码追踪
6. 栈顶缓存技术
7. 动态链接
8. 方法的调用：解析与分派
9. 方法返回地址
10. 一些附加信息
11. 栈的相关面试题

由于跨平台性的设计，Java的指令都是根据栈来设计的。不同平台CPU架构不同，所以不能设计为基于寄存器的。

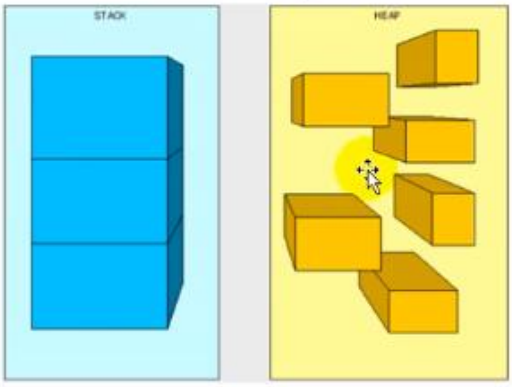
优点是跨平台，指令集小，编译器容易实现，缺点是性能下降，实现同样的功能需要更多的指令。

内存中的栈与堆

尚硅谷

栈是运行时的单位，而堆是存储的单位。

即：栈解决程序的运行问题，即程序如何执行，或者说如何处理数据。堆解决的是数据存储的问题，即数据怎么放、放在哪儿。



usually we put object in heap,as for some other datas like referencing addresses or something like local variables or interim variables,we put them in stack

虚拟机栈基本内容

尚硅谷

- **Java虚拟机栈是什么？**

  Java虚拟机栈（Java Virtual Machine Stack），早期也叫Java栈。每个线程在创建时都会创建一个虚拟机栈，其内部保存一个个的栈帧（Stack Frame），对应着一次次的Java方法调用。

  - 是线程私有的

- **生命周期**

  生命周期和线程一致。

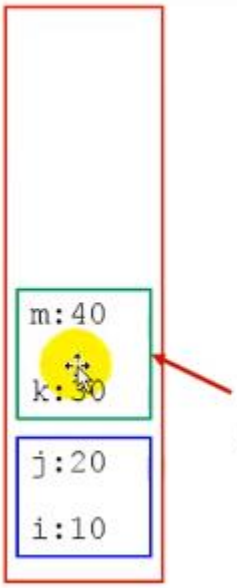
- **作用**

  主管Java程序的运行，它保存方法的局部变量(8种基本数据类型、对象的引用地址)、部分结果，并参与方法的调用和返回。

  - 局部变量 vs 成员变量(或属性)
  - 基本数据变量 vs 引用类型变量（类、数组、接口）

stack frame is the basic unit to save data. One stack frame corresponds to one function.

One stack corresponds to one thread

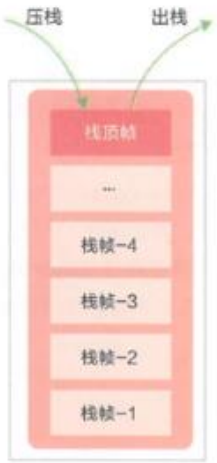


代表一个栈桢，一个栈桢就对应着一个方法。

```java
public void methodA() {
    int i = 10;
    int j = 20;

    methodB();
}

public void methodB(){
    int k = 30;
    int m = 40;
}
```

栈的特点（优点）

- 栈是一种快速有效的分配存储方式，访问速度仅次于程序计数器。
- JVM直接对Java栈的操作只有两个：
  - 每个方法执行，伴随着进栈(入栈、压栈)
  - 执行结束后的出栈工作
- 对于栈来说不存在垃圾回收问题
  - GC ； OOM



虚拟机栈基本内容

栈中可能出现的异常

- Java 虚拟机规范允许Java栈的大小是动态的或者是固定不变的。
  - 如果采用固定大小的Java虚拟机栈，那每一个线程的Java虚拟机栈容量可以在线程创建的时候独立选定。如果线程请求分配的栈容量超过Java虚拟机栈允许的最大容量，Java虚拟机将会抛出一个 StackOverflowError 异常。
  - 如果Java虚拟机栈可以动态扩展，并且在尝试扩展的时候无法申请到足够的内存，或者在创建新的线程时没有足够的内存去创建对应的虚拟机栈，那Java虚拟机将会抛出一个 OutOfMemoryError 异常。

```java
/**
 * 演示栈中的异常:StackOverflowError
 * @author shkstart
 * @create 2020 下午 9:08
 */
public class StackErrorTest {

    public static void main(String[] args) {
        main(args);
    }

}
```

虚拟机栈基本内容

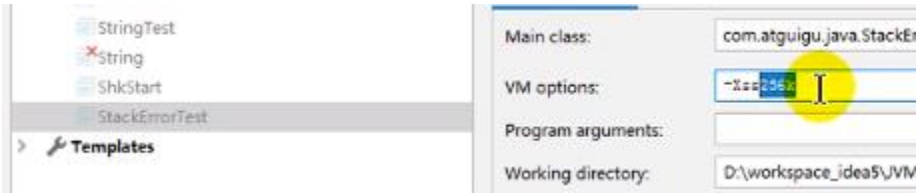
尚硅谷

- 设置栈内存大小

我们可以使用参数-Xss 选项来设置线程的最大栈空间，栈的大小直接决定了函数调用的最大可达深度。

```java
public class StackDeepTest {
        private static int count = 0;
        public static void recursion() {
                count++;
                recursion();
        }
        public static void main(String args[]) {
                try {
                        recursion();
                } catch (Throwable e) {
                        System.out.println("deep of calling = " + count);
                        e.printStackTrace();
                }
        }
}
```

```java
package com.atguigu.java;

/**
 * 演示栈中的异常:StackOverflowError
 * @author shkstart
 * @create 2020 下午 9:08
 *
 * 默认情况下：count : 11420
 * 设置栈的大小：-Xss256k : count : 2465
 */
public class StackErrorTest {
    private static int count = 1;
    public static void main(String[] args) {
        System.out.println(count);
        count++;
        main(args);
    }

}
```



2- 栈的存储单位

栈中存储什么？

- 每个线程都有自己的栈，栈中的数据都是以栈帧（Stack Frame）的格式存在。
- 在这个线程上正在执行的每个方法都各自对应一个栈帧（Stack Frame）。
- 栈帧是一个内存区块，是一个数据集，维系着方法执行过程中的各种数据信息。

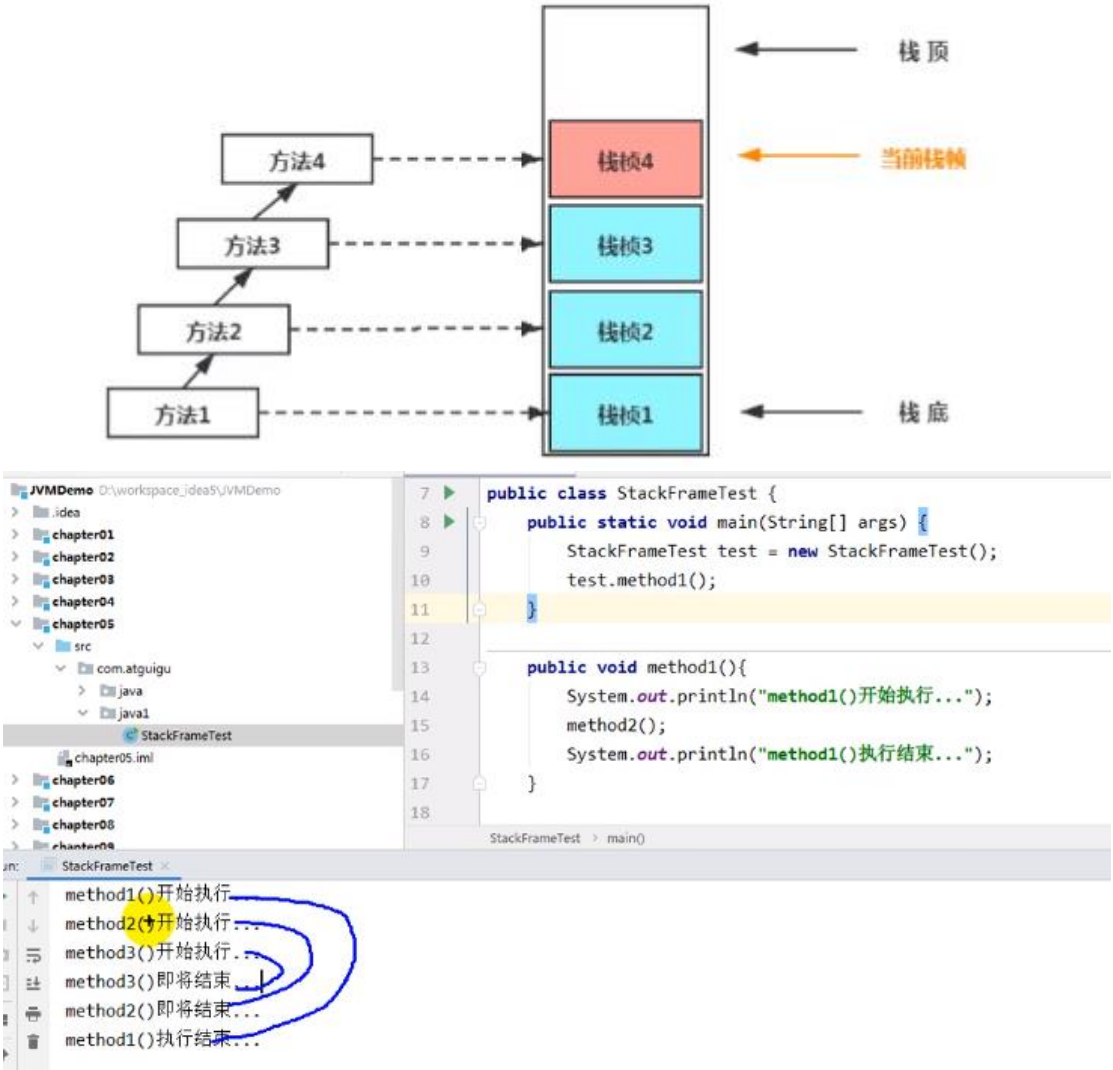
复习：
- OOP的基本概念：类、对象
- 类中基本结构：field（属性、字段、域）、method

栈运行原理

尚硅谷

- JVM直接对Java栈的操作只有两个，就是对栈帧的压栈和出栈，遵循“先进后出”/“后进先出”原则。
- 在一条活动线程中，一个时间点上，只会有一个活动的栈帧。即只有当前正在执行的方法的栈帧（栈顶栈帧）是有效的，这个栈帧被称为当前栈帧（Current Frame），与当前栈帧相对应的方法就是当前方法（Current Method），定义这个方法的类就是当前类（Current Class）。
- 执行引擎运行的所有字节码指令只针对当前栈帧进行操作。
- 如果在该方法中调用了其他方法，对应的新的栈帧会被创建出来，放在栈的顶端，成为新的当前帧。

栈运行原理


栈顶
当前栈帧
栈底
方法4
方法3
方法2
方法1
栈帧4
栈帧3
栈帧2
栈帧1
JVMDemo D:\workspace_idea5\JVMDemo
.idea
chapter01
chapter02
chapter03
chapter04
chapter05
src
com.atguigu
java
java1
StackFrameTest
chapter05.iml
chapter06
chapter07
chapter08
public class StackFrameTest {
public static void main(String[] args) {
StackFrameTest test = new StackFrameTest();
test.method1();
}
public void method1(){
System.out.println("method1()开始执行...");
method2();
System.out.println("method1()执行结束...");
}
StackFrameTest > main()
StackFrameTest
method1()开始执行...
method2()开始执行...
method3()开始执行...
method3()即将结束...
method2()即将结束...
method1()执行结束...

栈运行原理

- 不同线程中所包含的栈帧是不允许存在相互引用的，即不可能在一个栈帧之中引用另外一个线程的栈帧。
- 如果当前方法调用了其他方法，方法返回之际，当前栈帧会传回此方法的执行结果给前一个栈帧，接着，虚拟机会丢弃当前栈帧，使得前一个栈帧重新成为当前栈帧。
- Java方法有两种返回函数的方式，一种是正常的函数返回，使用return指令；另外一种是抛出异常。不管使用哪种方式，都会导致栈帧被弹出。

```
public class CurrentFrameTest{
        public void methodA() {
                System.out.println("当前栈帧对应的方法->methodA");
                methodB();
                System.out.println("当前栈帧对应的方法->methodA");
        }
        public void methodB() {
                System.out.println("当前栈帧对应的方法->methodB");
        }
}
```

threads are isolated,so stack frame in thread can not refer to each other

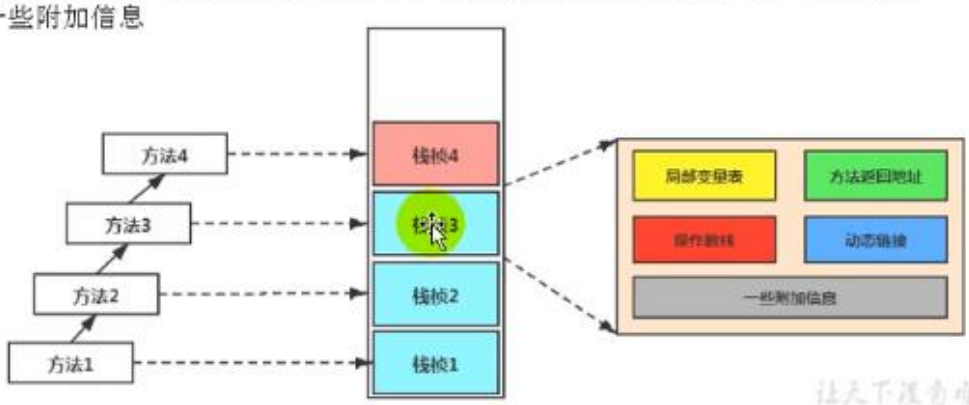
Ali uses outer space to share data among instances

```
 *
 * 方法的结束方式分为两种：① 正常结束，以return为代表  ② 方法执行中出现未捕获处理的异常，以抛出异常的
 *
 */
public class StackFrameTest {
    public static void main(String[] args) {
        try {
            StackFrameTest test = new StackFrameTest();
            test.method1();

        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("main()正常结束");
```

栈桢的内部结构

每个栈帧中存储着：

- 局部变量表（Local Variables）
- 操作数栈（Operand Stack）（或表达式栈）
- 动态链接(Dynamic Linking)（或指向运行时常量池的方法引用）
- 方法返回地址（Return Address）（或方法正常退出或者异常退出的定义）
- 一些附加信息



栈桢的内部结构

尚硅谷

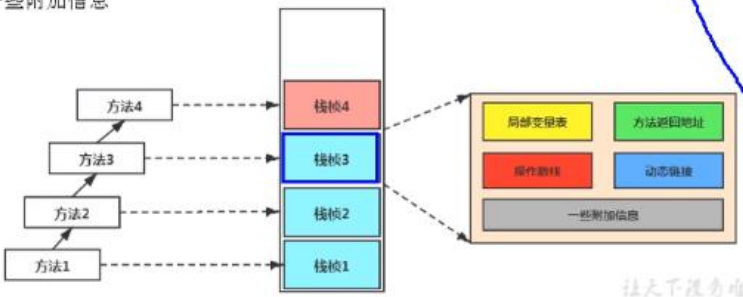
每个栈帧中存储着：

- 局部变量表（Local Variables）
- 操作数栈（Operand Stack）（或表达式栈）
- 动态链接(Dynamic Linking)（或指向运行时常量池的方法引用）
- 方法返回地址（Return Address）（或方法正常退出或者异常退出的定义）
- 一些附加信息

LV

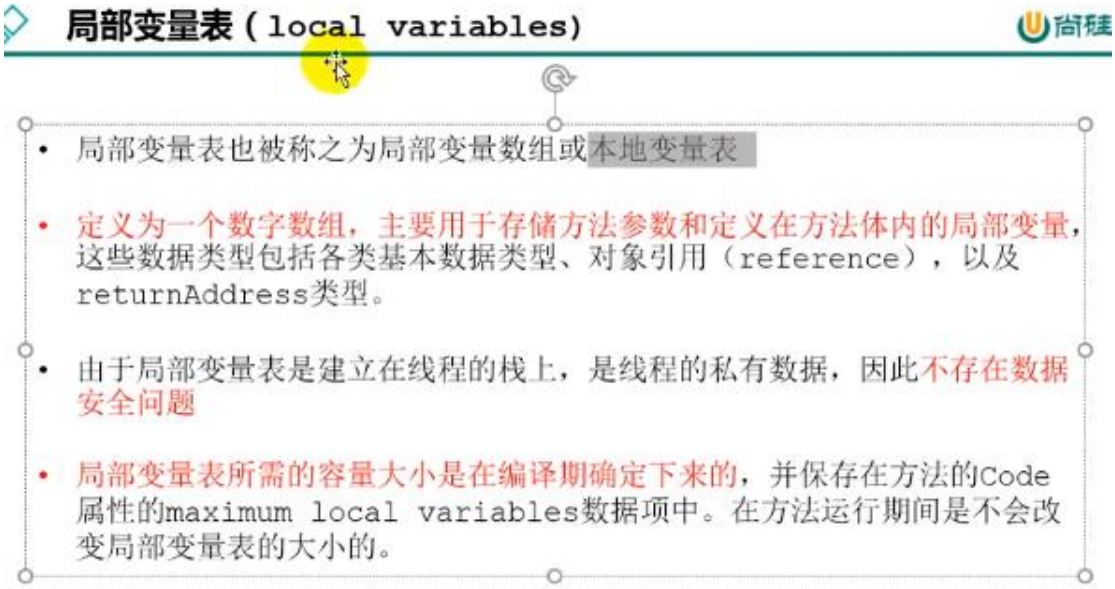
static{

}



it is similar between static block and dynamic linking , just two different perspectives.

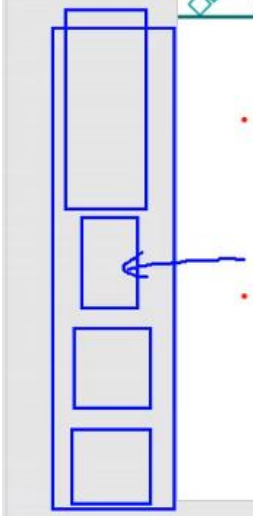
How many stack frame there will be depends on the size of each stack frame which depends on the size of local variables and operand stack



关于Slot的理解

- 参数值的存放总是在局部变量数组的index0开始，到数组长度-1的索引结束。
- 局部变量表，最基本的存储单元是Slot（变量槽）
- 局部变量表中存放编译期可知的各种基本数据类型(8种)，引用类型(reference)，returnAddress类型。
- 在局部变量表里，32位以内的类型只占用一个slot（包括returnAddress类型），64位的类型（long和double)占用两个slot。
  - byte 、short 、char 在存储前被转换为int，boolean 也被转换为int，0 表示false ，非0 表示true。
  - long 和double 则占据两个Slot。

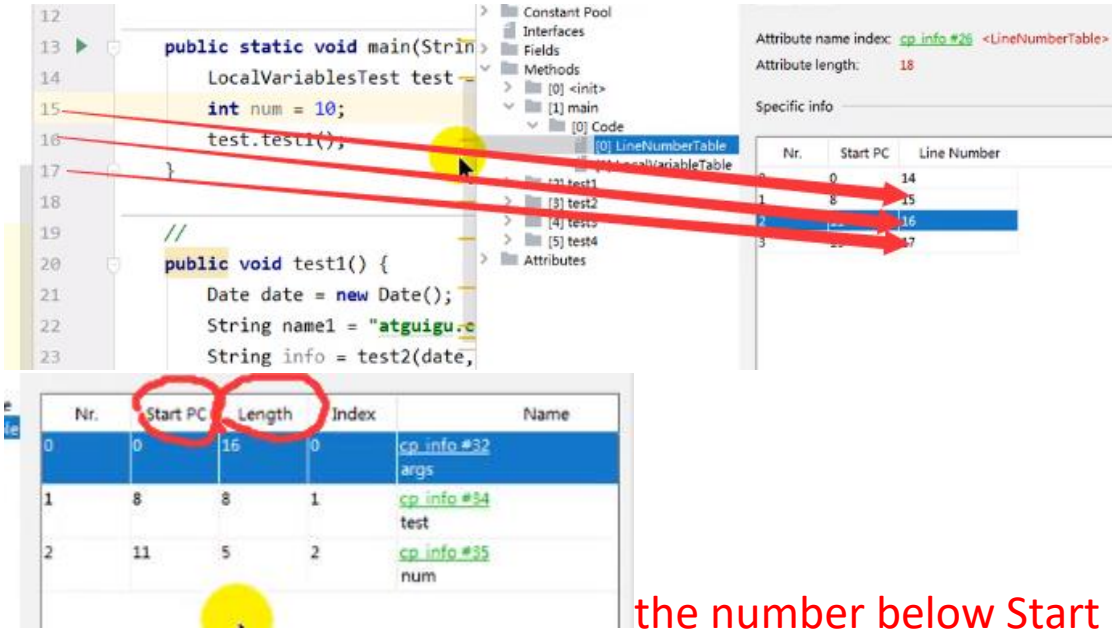
局部变量表（local variables）



- 方法嵌套调用的次数由栈的大小决定。一般来说，栈越大，方法嵌套调用次数越多。对一个函数而言，它的参数和局部变量越多，使得局部变量表膨胀，它的栈帧就越大，以满足方法调用所需传递的信息增大的需求。进而函数调用就会占用更多的栈空间，导致其嵌套调用次数就会减少。
- 局部变量表中的变量只在当前方法调用中有效。在方法执行时，虚拟机通过使用局部变量表完成参数值到参数变量列表的传递过程。当方法调用结束后，随着方法栈帧的销毁，局部变量表也会随之销毁。

this “L” means reference

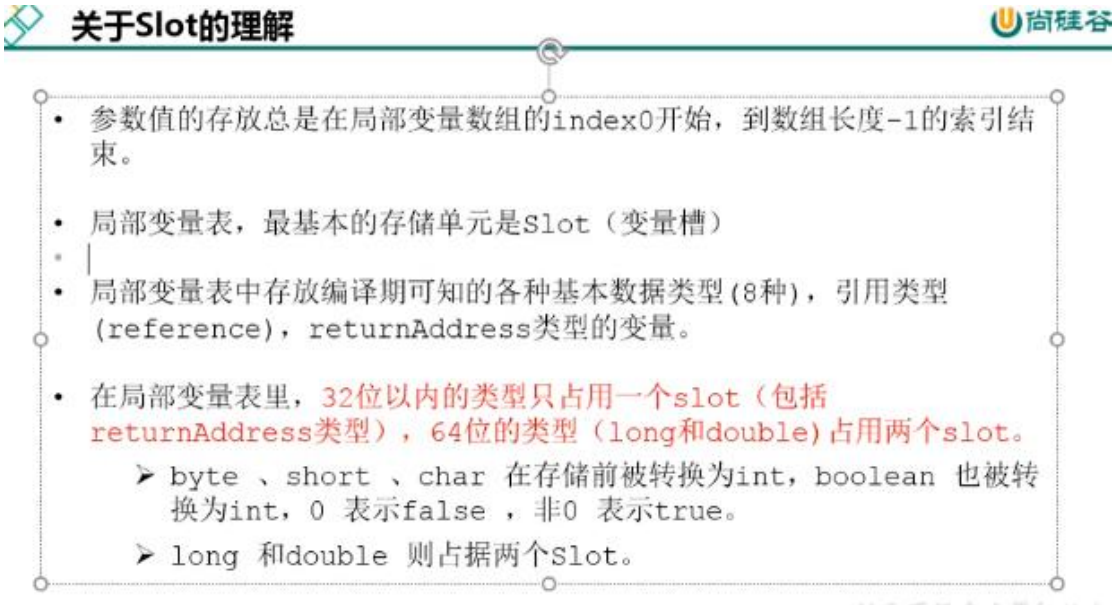
Code length means the length of bytecode

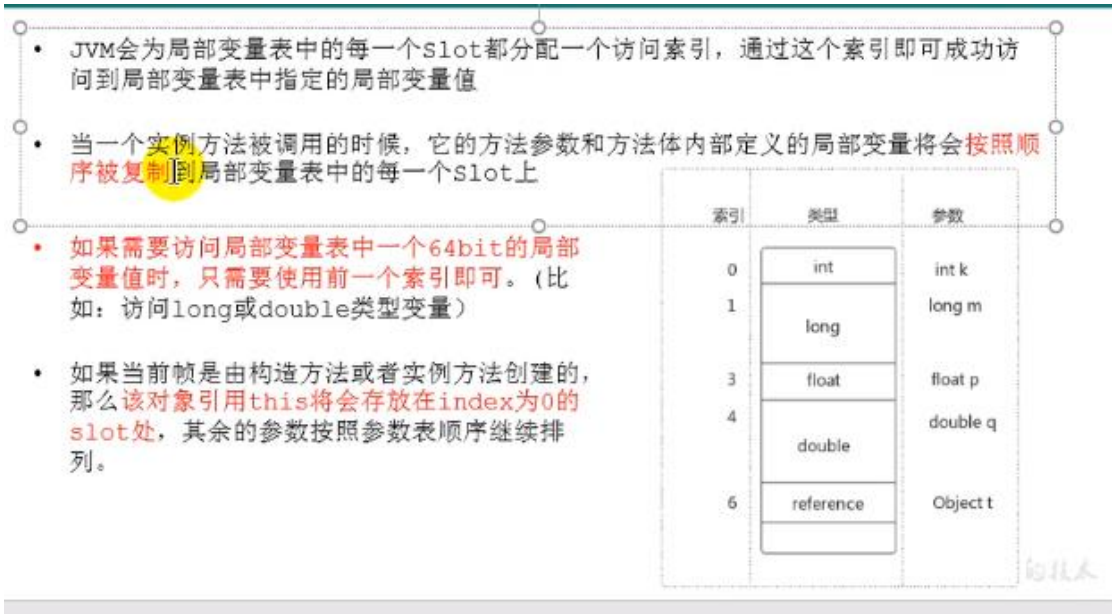


Nr.	Start PC	Length	Index	Name
0	0	16	0	cp_info #32 args
1	8	8	1	cp_info #34 test
2	11	5	2	cp_info #35 num

the number below Start PC means the variable starts from this number-line to the end of the whole function field.

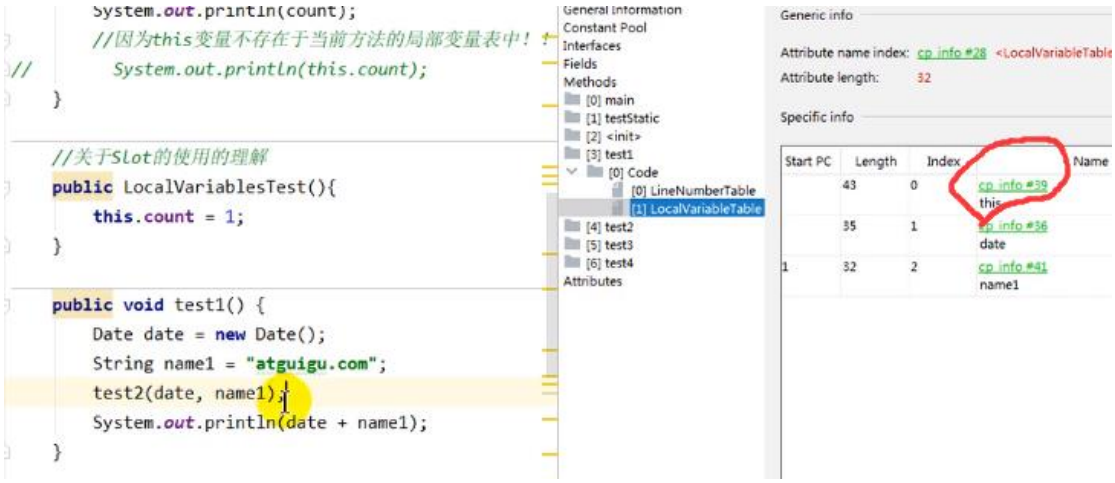
The length means the variable's effective field length.





```
//练习：
public static void testStatic(){
    LocalVariablesTest test = new LocalVariablesTest();
    Date date = new Date();
    int count = 10;
    System.out.println(count);
    //因为this变量不存在于当前方法的局部变量表中！！
    System.out.println(this.count);
}
```

as for function which is not static,there will be “this” putting at the top of the slot at first and its index is 0

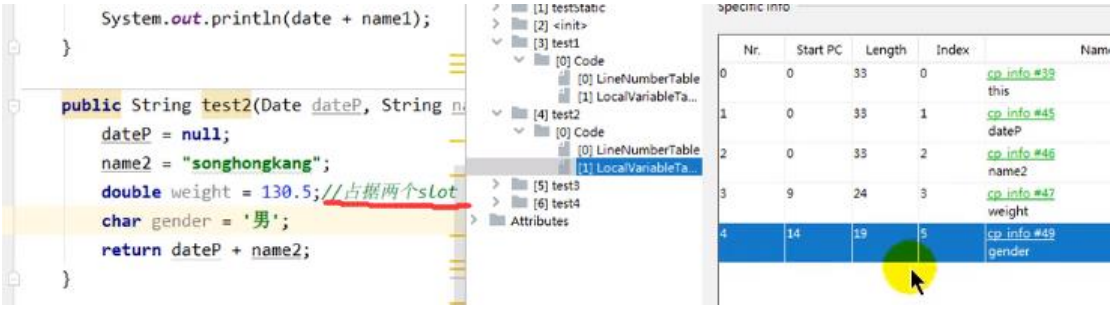


```
    System.out.println(count);
    //因为this变量不存在于当前方法的局部变量表中！！
//      System.out.println(this.count);
}

//关于Slot的使用的理解
public LocalVariablesTest(){
    this.count = 1;
}

public void test1() {
    Date date = new Date();
    String name1 = "atguigu.com";
    test2(date, name1);
    System.out.println(date + name1);
}
```

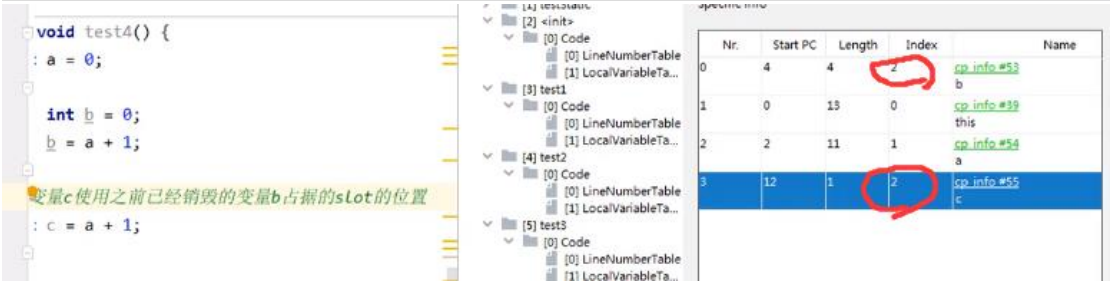

typical feature “this”.

If we are careful enough,we can find that the number below Start PC and the number under Length in each

line add up to the same which is 43



double type and long type occupy two slots



please pay attention to “b” and the index of “c” which is equal to “b”’s because of recycling mechanism

举例：静态变量与局部变量的对比

- 参数表分配完毕之后，再根据方法体内定义的变量的顺序和作用域分配。
- 我们知道类变量表有两次初始化的机会，第一次是在“准备阶段”，执行系统初始化，对类变量设置零值，另一次则是在“初始化”阶段，赋予程序员在代码中定义的初始值。
- 和类变量初始化不同的是，局部变量表不存在系统初始化的过程，这意味着一旦定义了局部变量则必须人为的初始化，否则无法使用。

```
public void test() {
    int i;
    System.out.println(i);
}
```

这样的代码是错误的，没有赋值不能够使用。

```
基本数据类型  ② 引用数据类型
置分：① 成员变量：在使用前，都经历过默认初始化赋值
            类变量： linking的prepare阶段：给类变量默认赋值  ---> initial阶段：给类变量显式赋值即静态代码块赋值
            实例变量：
        ② 局部变量
/*
变量的分类：按照数据类型分：① 基本数据类型  ② 引用数据类型
          按照在类中声明的位置分：① 成员变量：在使用前，都经历过默认初始化赋值
                                  类变量： linking的prepare阶段：给类变量默认赋值  ---> initial阶段：
                                  实例变量：随着对象的创建，会在堆空间中分配实例变量空间，并进行默认赋
                                ② 局部变量：在使用前，必须要进行显式赋值的！否则，编译不通过
        }
        //变量c使用之前已经销毁的变量b占据的slot的位置
        int c = a + 1;
    }

    /*
    变量的分类：按照数据类型分：① 基本数据类型  ② 引用数据类型
              按照在类中声明的位置分：① 成员变量：在使用前，都经历过默认初始化赋值
                                    类变量： linking的prepare阶段：给类变量默认赋值  ---> initial阶段：
                                    实例变量：随着对象的创建，会在堆空间中分配实例变量空间，并进行默认赋
                                  ② 局部变量：在使用前，必须要进行显式赋值的！否则，编译不通过
     */
    public void test5Temp(){
        int num = 1;
        System.out.println(num);//错误信息：变量num未进行初始化
    }

}
```

if the variable in variable table no longer exists,then there will be no pointer from this variable,as a consequence the object which is indicated by this pointer in heap will be collected by garbage collector.

the operand stack here is made by array,so it can get all features inherited from array

操作数栈

- 每一个独立的栈帧中除了包含局部变量表以外，还包含一个后进先出（Last-In-First-Out）的操作数栈，也可以称之为表达式栈（Expression Stack）。
- 操作数栈，在方法执行过程中，根据字节码指令，往栈中写入数据或提取数据，即入栈（push）/出栈（pop）。
  - 某些字节码指令将值压入操作数栈，其余的字节码指令将操作数取出栈。使用它们后再把结果压入栈。
  - 比如：执行复制、交换、求和等操作

操作数栈

- 如果被调用的方法带有返回值的话，其返回值将会被压入当前栈帧的操作数栈中，并更新PC寄存器中下一条需要执行的字节码指令。
- 操作数栈中元素的数据类型必须与字节码指令的序列严格匹配，这由编译器在编译器期间进行验证，同时在类加载过程中的类检验阶段的数据流分析阶段要再次验证。
- 另外，我们说Java虚拟机的解释引擎是基于栈的执行引擎，其中的栈指的就是操作数栈。

操作数栈

- 每一个独立的栈帧中除了包含局部变量表以外，还包含一个后进先出（Last-In-First-Out）的操作数栈，也可以称之为表达式栈（Expression Stack）。
- 操作数栈，在方法执行过程中，根据字节码指令，往栈中写入数据或提取数据，即入栈（push）/出栈（pop）。
  - 某些字节码指令将值压入操作数栈，其余的字节码指令将操作数取出栈。使用它们后再把结果压入栈。
  - 比如：执行复制、交换、求和等操作

操作数栈

尚硅谷

代码举例

```
public void testAddOperation() {
    byte i = 15;
    int j = 8;
    int k = i + j;
}
```

字节码指令信息

```
public void testAddOperation();
    Code:
      0: bipush        15
      2: istore_1
      3: bipush        8
      5: istore_2
      6: iload_1
      7: iload_2
      8: iadd
      9: istore_3
      10: return
```

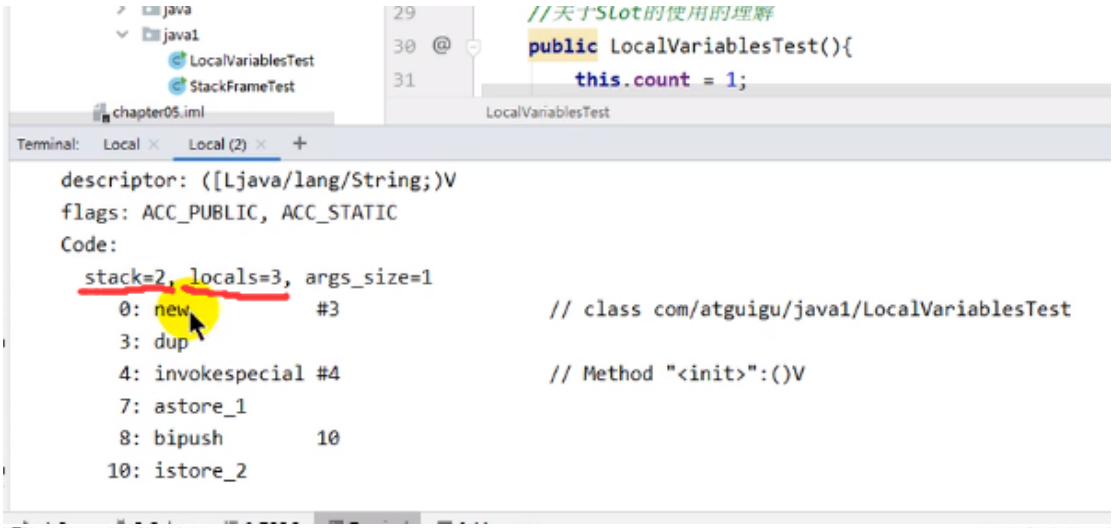
操作数栈

- 操作数栈，主要用于保存计算过程的中间结果，同时作为计算过程中变量临时的存储空间。
- 操作数栈就是JVM执行引擎的一个工作区，当一个方法刚开始执行的时候，一个新的栈帧也会随之被创建出来，这个方法的操作数栈是空的。
- 每一个操作数栈都会拥有一个明确的栈深度用于存储数值，其所需的最大深度在编译期就定义好了，保存在方法的Code属性中，为max_stack的值。
- 栈中的任何一个元素都是可以任意的Java数据类型。
  - 32bit的类型占用一个栈单位深度
  - 64bit的类型占用两个栈单位深度
- 操作数栈并非采用访问索引的方式来进行数据访问的，而是只能通过标准的入栈（push）和出栈（pop）操作来完成一次数据访问。

数组，一旦创建，其长度就是确定的！

like local variable table,once an array is created,its length is determined.but these two structures are totally different.

Although operand stack is made by array,it can not use index to access data.



操作数栈

- 如果被调用的方法带有返回值的话，其返回值将会被压入当前栈帧的操作数栈中，并更新PC寄存器中下一条需要执行的字节码指令。
- 操作数栈中元素的数据类型必须与字节码指令的序列严格匹配，这由编译器在编译器期间进行验证，同时在类加载过程中的类检验阶段的数据流分析阶段要再次验证。
- 另外，我们说Java虚拟机的解释引擎是基于栈的执行引擎，其中的栈指的就是操作数栈。

5- 代码追踪

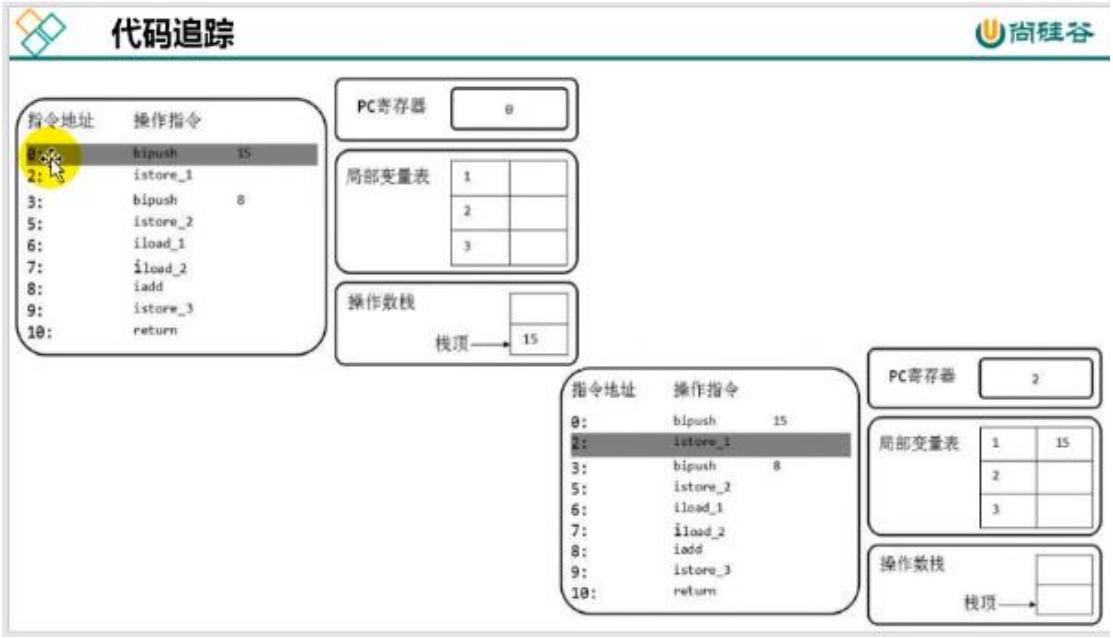
代码追踪

```
public void testAddOperation(){
    byte i = 15;
    int j = 8;
    int k = i + j;
}
```

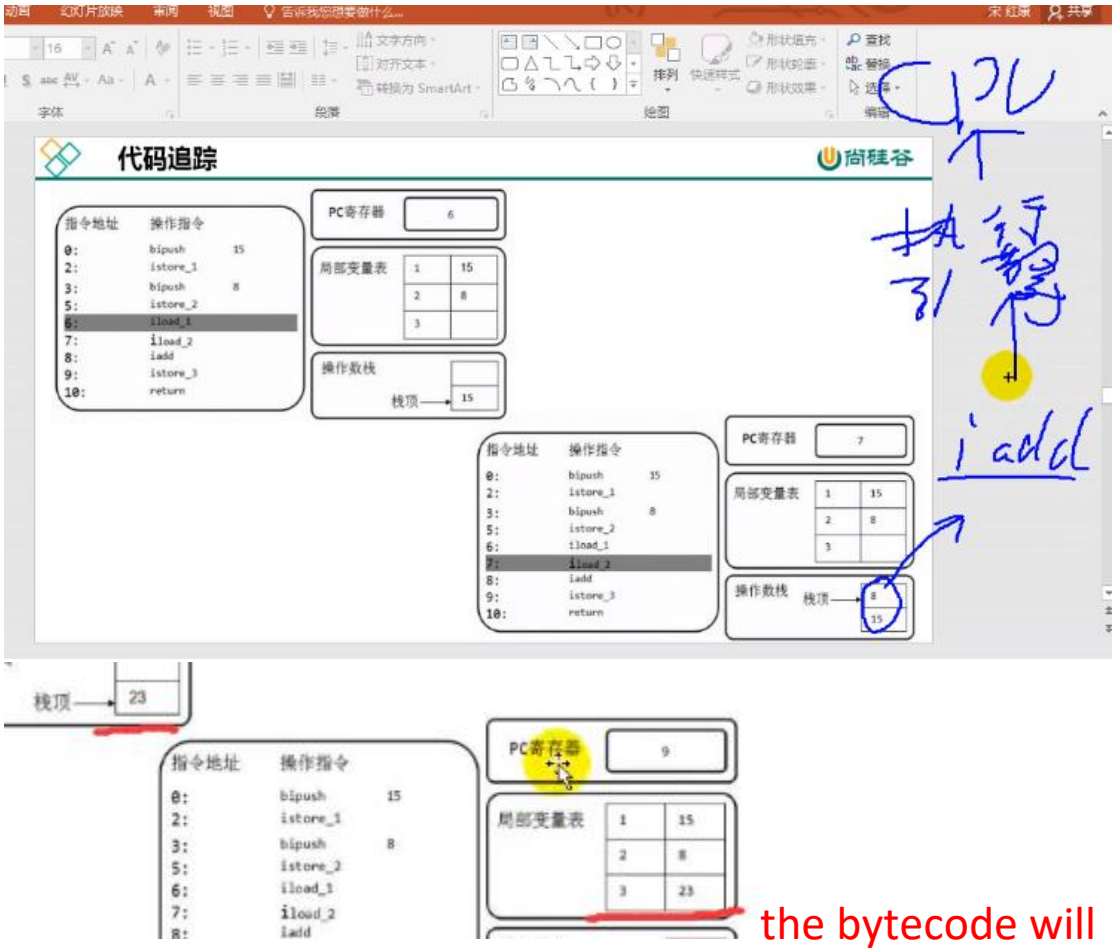
使用javap命令反编译class文件： javap -v 类名.class

```
public void testAddOperation();
    Code:
      0: bipush        15
      2: istore_1
      3: bipush        8
      5: istore_2
      6: iload_1
      7: iload_2
      8: iadd
      9: istore_3
     10: return
```

```
lic void testAddOperation() {
//byte、short、char、boolean：都以int型来保存
byte i = 15;
```



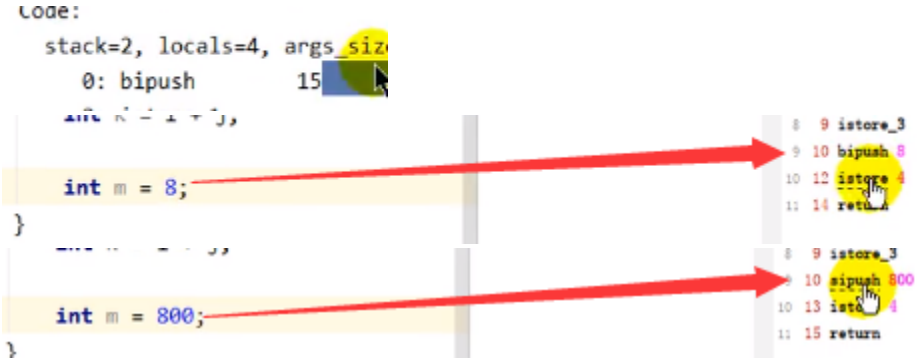
we can see that there is no index which is number 0,cause index 0 has been taken by “this”.

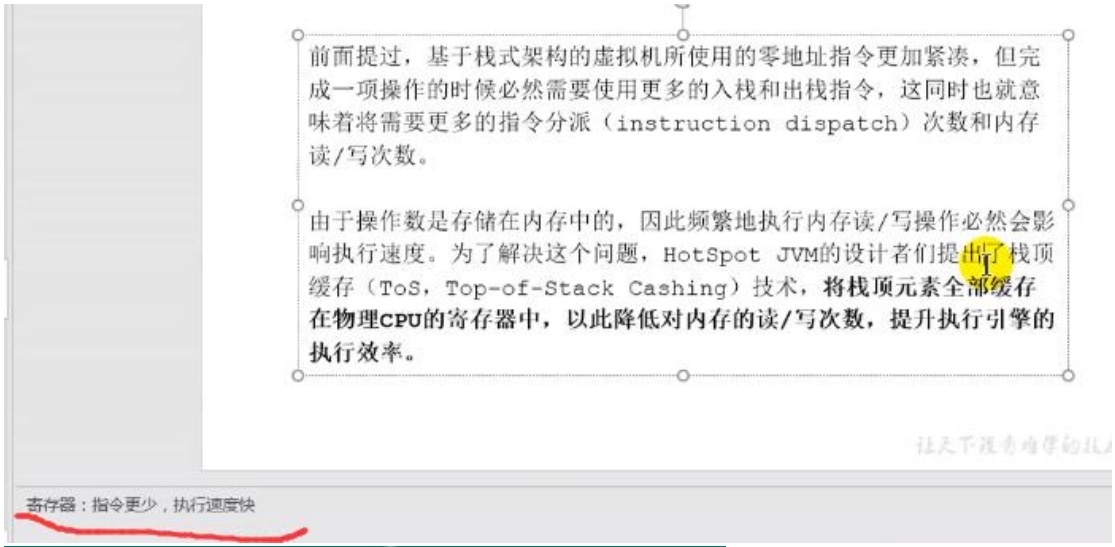


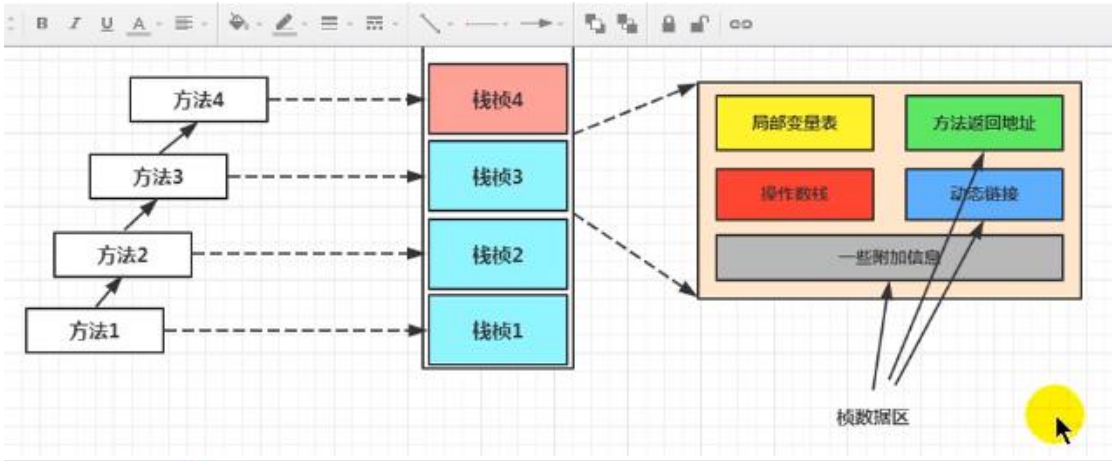
the bytecode will

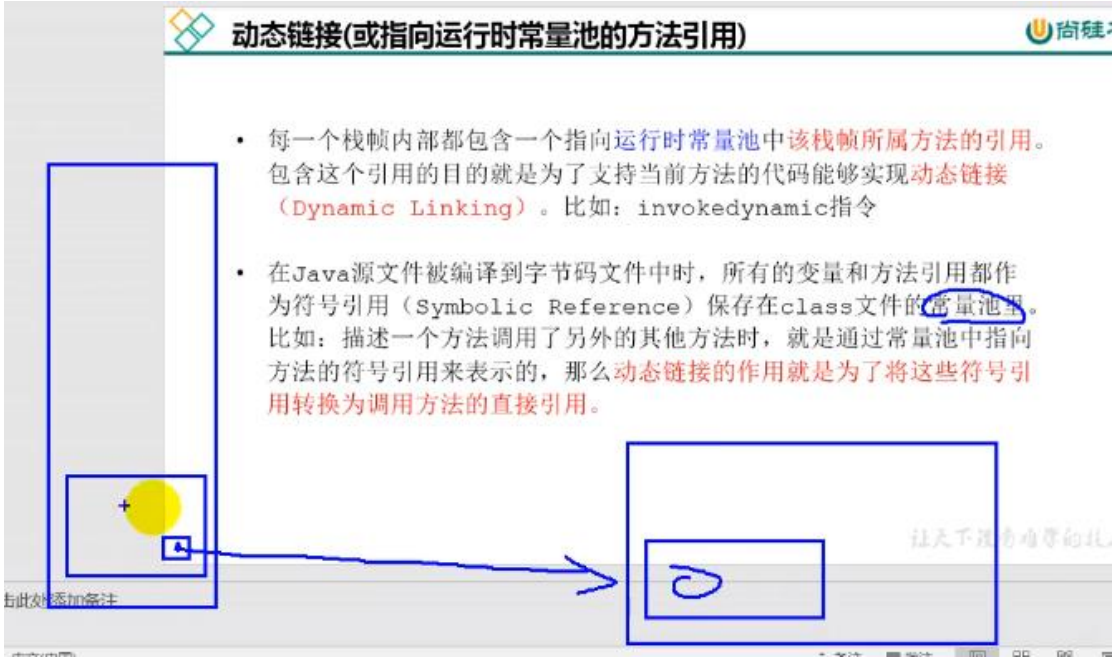
be translated to machine-code by execution engine,then the code will be sent to cpu to execute .

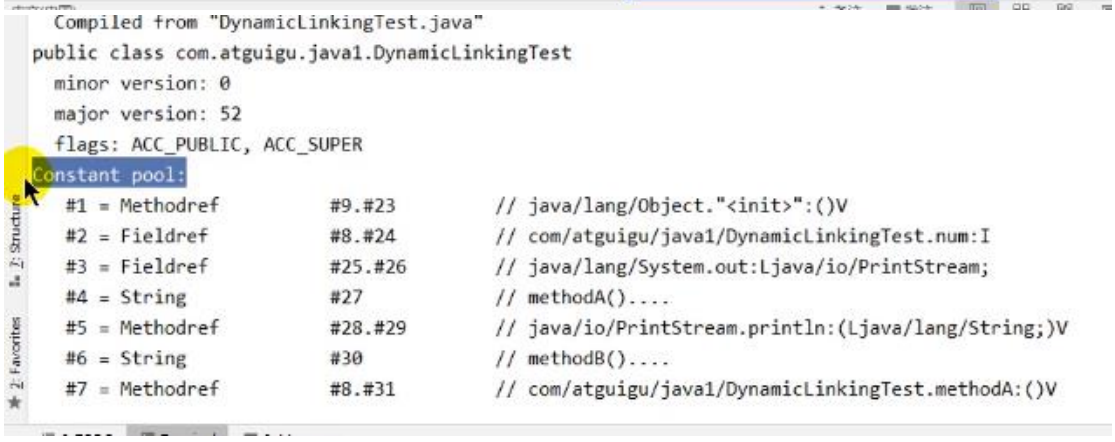
we can see the picture above,it’s obvious that the depth of the operand is two and the size of local variables is four.and we can contrast it to this pic:







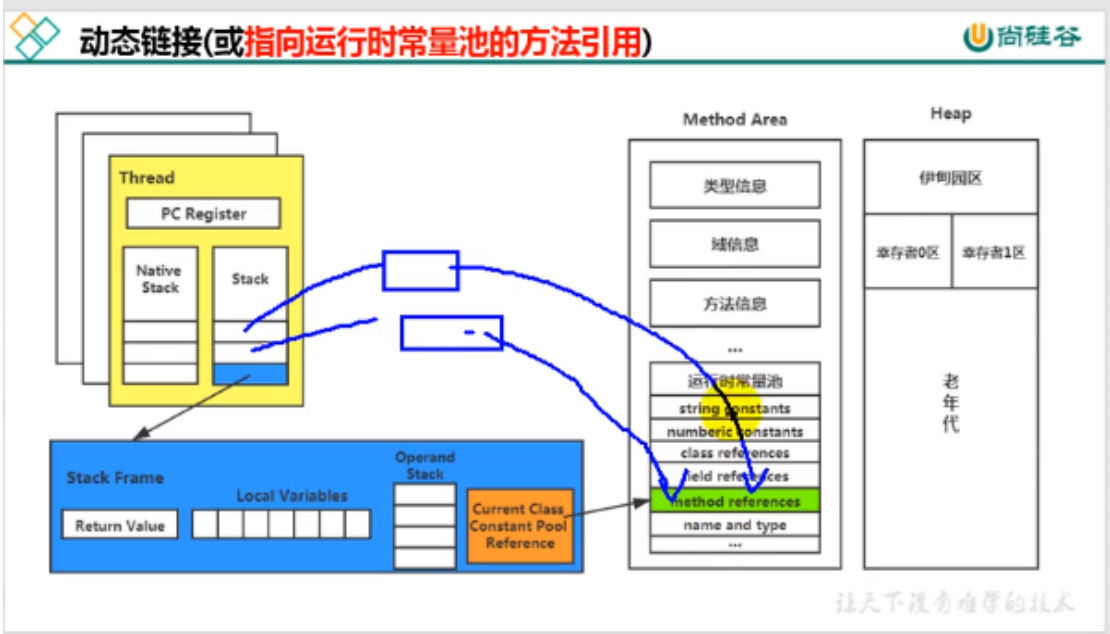




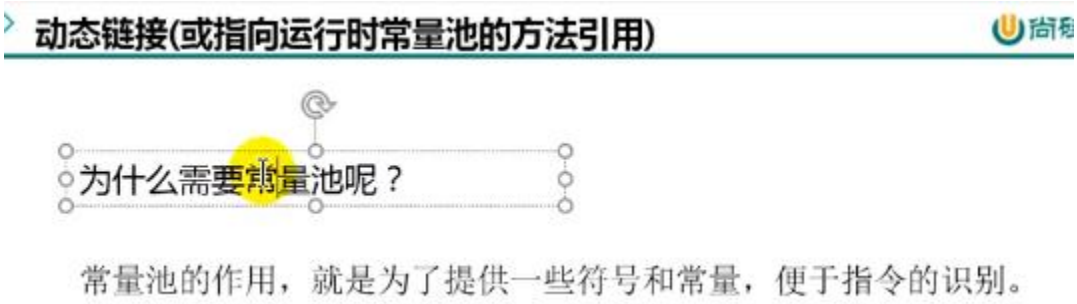
```
Compiled from "DynamicLinkingTest.java"
public class com.atguigu.java1.DynamicLinkingTest
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #9.#23         // java/lang/Object."<init>":()V
   #2 = Fieldref           #8.#24         // com/atguigu/java1/DynamicLinkingTest.num:I
   #3 = Fieldref           #25.#26        // java/lang/System.out:Ljava/io/PrintStream;
   #4 = String             #27            // methodA()....
   #5 = Methodref          #28.#29        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #6 = String             #30            // methodB()....
   #7 = Methodref          #8.#31         // com/atguigu/java1/DynamicLinkingTest.methodA:()V
```

if you notice the annotations behind each line,you will find that there are symbols like “()V”which means the funtion return void and there is no parameters.

Every variable in constant pool will only appear once,and every machine-code are able to use the variables in this pool when they need to refer.



let me explain the pic above,firstly,the Thread is made up by PC register,Native stack and stack.The Current Class Constant Pool Reference in stack is going to have a pointer to running-time constant pool.The whole process is called dynamic linking.



方法的调用

在JVM中，将符号引用转换为调用方法的直接引用与方法的绑定机制相关。

- 静态链接：

当一个字节码文件被装载进JVM内部时，如果被调用的目标方法在编译期可知，且运行期保持不变时。这种情况下将调用方法的符号引用转换为直接引用的过程称之为静态链接。

- 动态链接：

如果被调用的方法在编译期无法被确定下来，也就是说，只能够在程序运行期将调用方法的符号引用转换为直接引用，由于这种引用转换过程具备动态性，因此也就被称之为动态链接。

方法的调用

对应的方法的绑定机制为：早期绑定（Early Binding）和晚期绑定（Late Binding）。绑定是一个字段、方法或者类在符号引用被替换为直接引用的过程，这仅仅发生一次。

- 早期绑定：

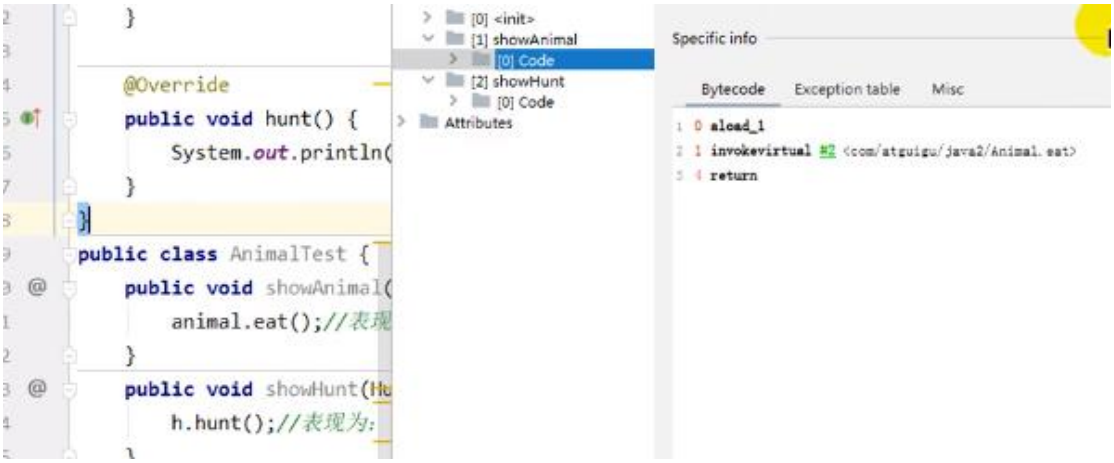
早期绑定就是指被调用的目标方法如果在编译期可知，且运行期保持不变时，即可将这个方法与所属的类型进行绑定，这样一来，由于明确了被调用的目标方法究竟是哪一个，因此也就可以使用静态链接的方式将符号引用转换为直接引用。

- 晚期绑定：

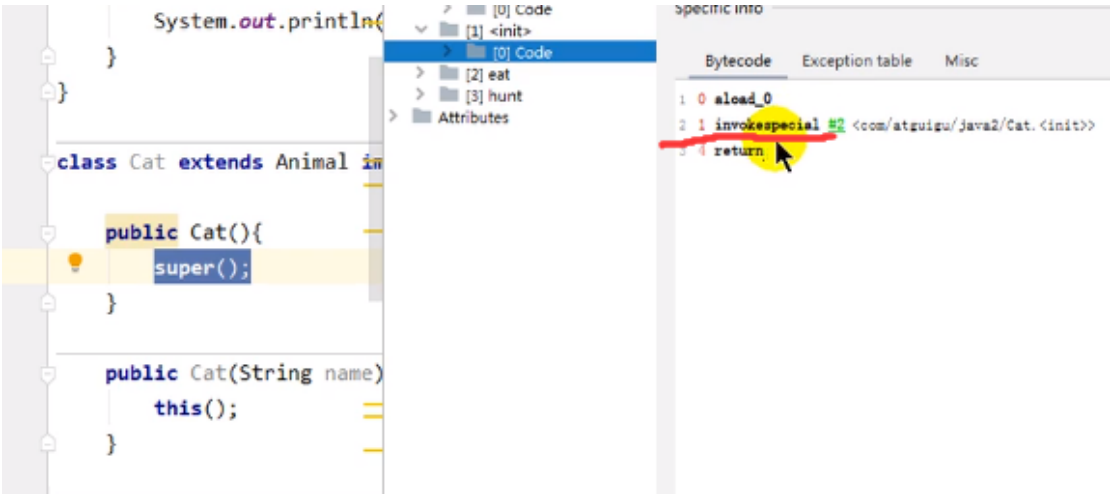
如果被调用的方法在编译期无法被确定下来，只能够在程序运行期根据实际的类型绑定相关的方法，这种绑定方式也就被称之为晚期绑定。

static linking corresponds to early binding,while dynamic lining corresponds to late binding

```
public class AnimalTest {
    public void showAnimal(Animal animal){
        animal.eat();//表现为：晚期绑定
    }
    public void showHunt(Huntable h){
        h.hunt();//表现为：晚期绑定
    }
}
```



the instance above:Huntable is an interface,Animal is a class which implements Huntable.



the instance above:represent early binding

方法的调用

随着高级语言的横空出世，类似于Java一样的基于面向对象的编程语言如今越来越多，尽管这类编程语言在语法风格上存在一定的差别，但是它们彼此之间始终保持着一个共性，那就是都支持封装、继承和多态等面向对象特性，既然这一类的编程语言具备多态特性，那么自然也就具备早期绑定和晚期绑定两种绑定方式。

Java中任何一个普通的方法其实都具备虚函数的特征，它们相当于C++语言中的虚函数（C++中则需要使用关键字virtual来显式定义）。如果在Java程序中不希望某个方法拥有虚函数的特征时，则可以使用关键字final来标记这个方法。

方法的调用：虚方法与非虚方法

非虚方法：

- 如果方法在编译期就确定了具体的调用版本，这个版本在运行时是不可变的。这样的方法称为非虚方法。
- 静态方法、私有方法、final方法、实例构造器、父类方法都是非虚方法。
- 其他方法称为虚方法。

子类对象的多态性的使用前提：① 类的继承关系 ② 方法的重写

方法的调用：虚方法与非虚方法

虚拟机中提供了以下几条方法调用指令：

- 普通调用指令：
  1. invokestatic：调用静态方法，解析阶段确定唯一方法版本
  2. invokespecial：调用<init>方法、私有及父类方法，解析阶段确定唯一方法版本
  3. invokevirtual：调用所有虚方法
  4. invokeinterface：调用接口方法
- 动态调用指令：
  5. invokedynamic：动态解析出需要调用的方法，然后执行

前四条指令固化在虚拟机内部，方法的调用执行不可人为干预，而invokedynamic指令则支持由用户确定方法版本。其中invokestatic指令和invokespecial指令调用的方法称为非虚方法，其余的（final修饰的除外）称为虚方法。

```
//invokevirtual
showFinal();//因为此方法声明有final，不能被子类重写，所以也认为此方法是非虚方法。
```

we

need to be careful with “final function” which is called a virtual function but actually it can’t be rewrite.

And if we put a prefix “super.” before “final function”,it will turn into “invokespecial “

```
//虚方法如下：
//invokevirtual
showCommon();
info();
```

```
MethodInterface in = null;
//invokeinterface
in.methodA();
```

方法的调用：关于invokedynamic指令

尚硅谷

- JVM字节码指令集一直比较稳定，一直到Java7中才增加了一个invokedynamic指令，这是Java为了实现『动态类型语言』支持而做的一种改进。
- 但是在Java7中并没有提供直接生成invokedynamic指令的方法，需要借助ASM这种底层字节码工具来产生invokedynamic指令。直到Java8的Lambda表达式的出现，invokedynamic指令的生成，在Java中才有了直接的生成方式。
- Java7中增加的动态语言类型支持的本质是对Java虚拟机规范的修改，而不是对Java语言规则的修改，这一块相对来讲比较复杂，增加了虚拟机中的方法调用，最直接的受益者就是运行在Java平台的动态语言的编译器。

方法的调用：关于invokedynamic指令

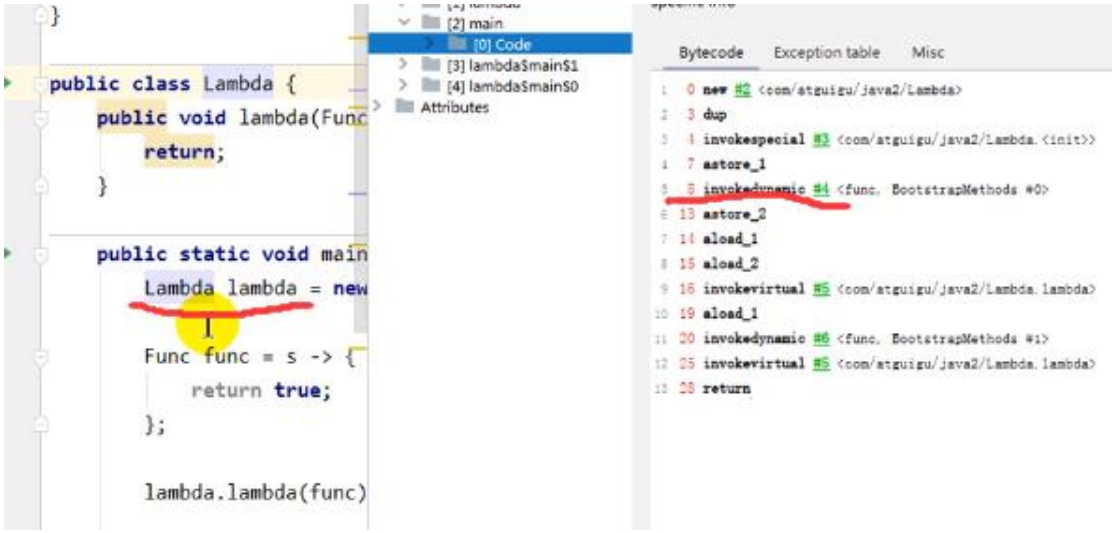
尚硅

动态类型语言和静态类型语言

动态类型语言和静态类型语言两者的区别就在于对类型的检查是在编译期还是在运行期，满足前者就是静态类型语言，反之是动态类型语言。

说的再直白一点就是，静态类型语言是判断变量自身的类型信息；动态类型语言是判断变量值的类型信息，变量没有类型信息，变量值才有类型信息，这是动态语言的一个重要特征。

```
Java: String info = "atguigu"; //info = atguigu;
JS: var name = "shkstart"; var name = 10;
Python: info = 130.5;
```



Java 语言中方法重写的本质：

1. 找到操作数栈顶的第一个元素所执行的对象的实际类型，记作 C。
2. 如果在类型 C 中找到与常量中的描述符合简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；如果不通过，则返回 java.lang.IllegalAccessError 异常。
3. 否则，按照继承关系从下往上依次对 C 的各个父类进行第 2 步的搜索和验证过程。
4. 如果始终没有找到合适的方法，则抛出 java.lang.AbstractMethodError异常。

IllegalAccessError介绍：

程序试图访问或修改一个属性或调用一个方法，这个属性或方法，你没有权限访问。一般的，这个会引起编译器异常。这个错误如果发生在运行时，就说明一个类发生了不兼容的改变。

normally we need to find a proper function between extended class up and down layer by layer when we get the function,Obviously it is too complex,so we need virtual variable table

方法的调用：虚方法表

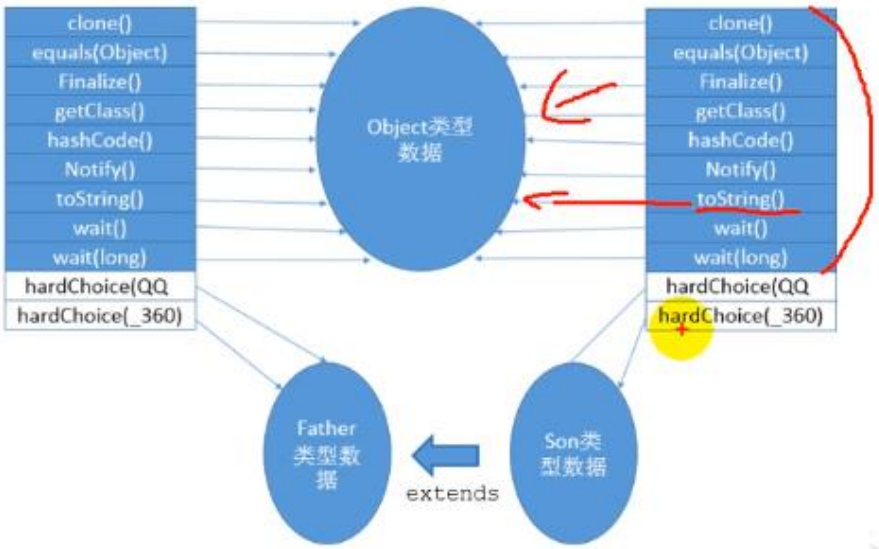
- 在面向对象的编程中，会很频繁的使用到动态分派，如果在每次动态分派的过程中都要重新在类的方法元数据中搜索合适的目标的话就可能影响到执行效率。因此，为了提高性能，JVM采用在类的方法区建立一个虚方法表（virtual method table）(非虚方法不会出现在表中)来实现。使用索引表来代替查找。
- 每个类中都有一个虚方法表，表中存放着各个方法的实际入口。
- 那么虚方法表什么时候被创建？

虚方法表会在类加载的链接阶段被创建并开始初始化，类的变量初始值准备完成之后，JVM会把该类的方法表也初始化完毕。

virtual method table will be initialized while resolve phase

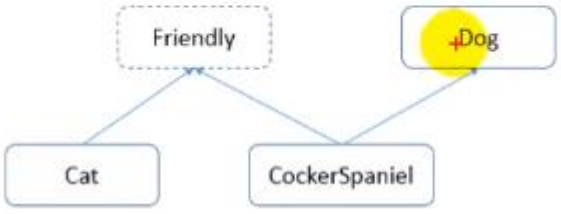
方法的调用：虚方法表

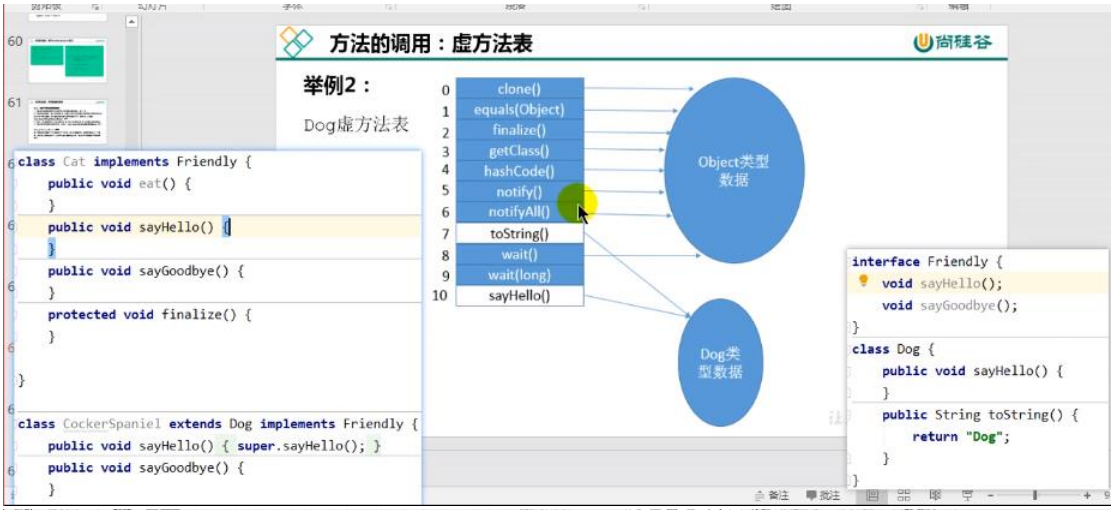
举例1：

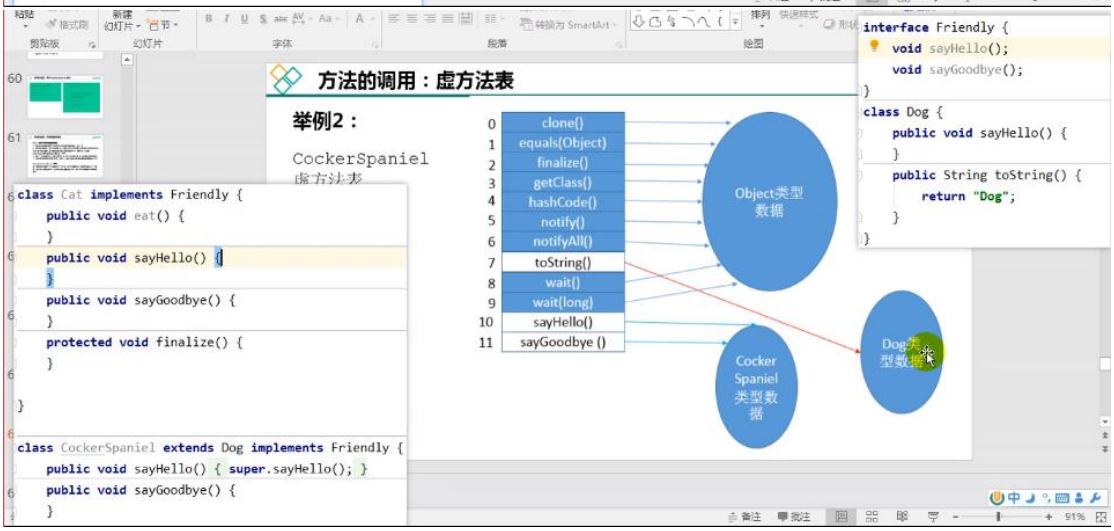


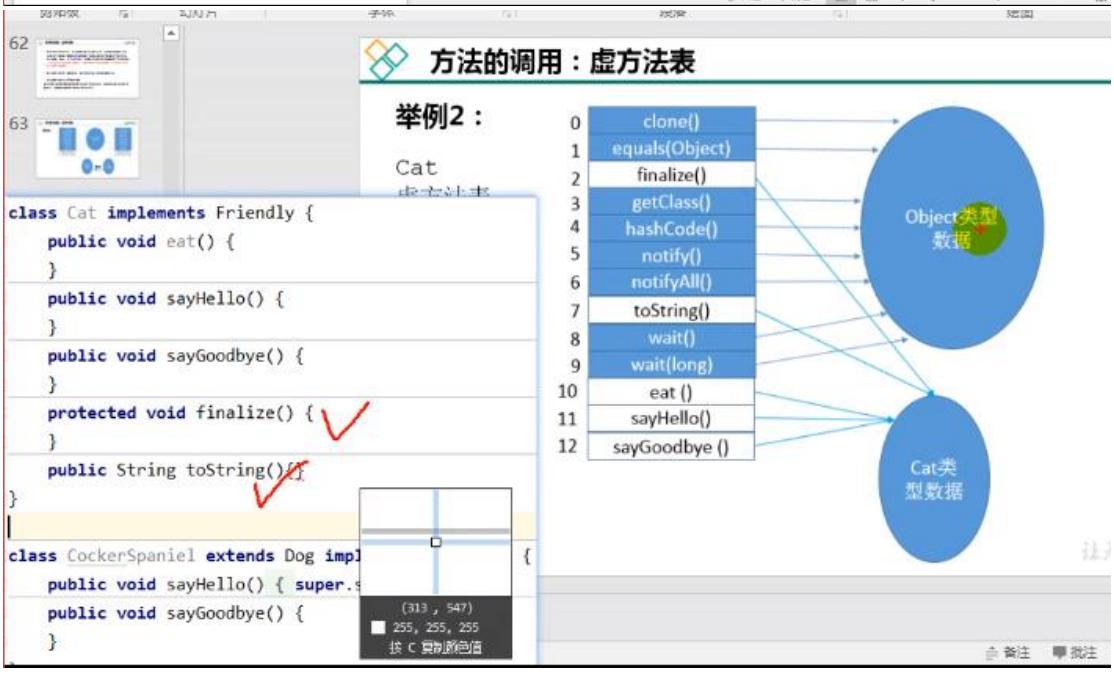
方法的调用：虚方法表

举例2：

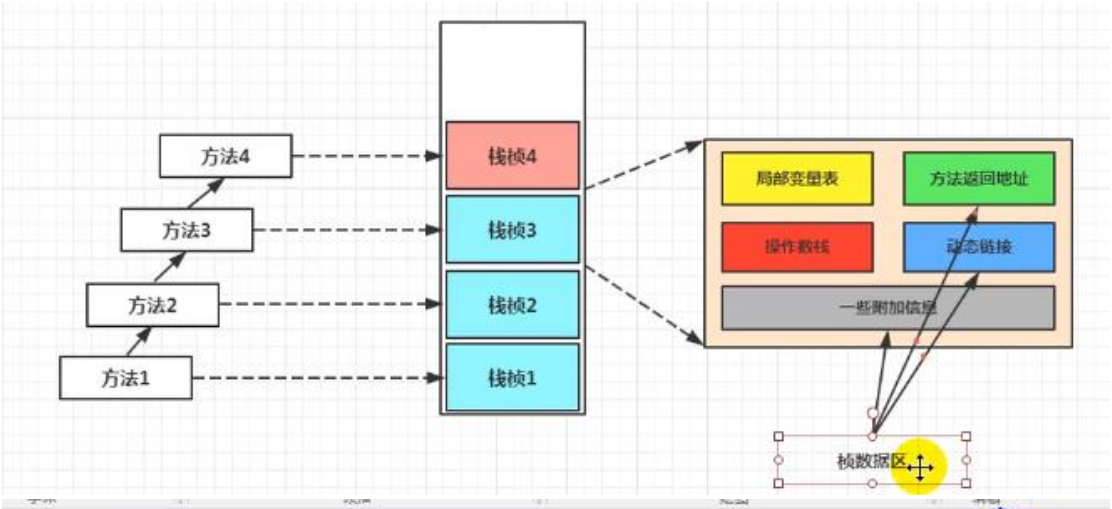



方法的调用：虚方法表
尚硅谷
举例2：
Dog虚方法表
0 clone()
1 equals(Object)
2 finalize()
3 getClass()
4 hashCode()
5 notify()
6 notifyAll()
7 toString()
8 wait()
9 wait(long)
10 sayHello()
Object类型数据
Dog类型数据
class Cat implements Friendly {
public void eat() {
}
public void sayHello() {
}
public void sayGoodbye() {
}
protected void finalize() {
}
}
class CockerSpaniel extends Dog implements Friendly {
public void sayHello() { super.sayHello(); }
public void sayGoodbye() {
}
interface Friendly {
void sayHello();
void sayGoodbye();
}
class Dog {
public void sayHello() {
}
public String toString() {
return "Dog";
}
}


方法的调用：虚方法表
举例2：
CockerSpaniel
虚方法表
0 clone()
1 equals(Object)
2 finalize()
3 getClass()
4 hashCode()
5 notify()
6 notifyAll()
7 toString()
8 wait()
9 wait(long)
10 sayHello()
11 sayGoodbye ()
Object类型数据
Cocker Spaniel类型数据
Dog类型数据
class Cat implements Friendly {
public void eat() {
}
public void sayHello() {
}
public void sayGoodbye() {
}
protected void finalize() {
}
}
class CockerSpaniel extends Dog implements Friendly {
public void sayHello() { super.sayHello(); }
public void sayGoodbye() {
}
interface Friendly {
void sayHello();
void sayGoodbye();
}
class Dog {
public void sayHello() {
}
public String toString() {
return "Dog";
}
}


方法的调用：虚方法表
举例2：
Cat
0 clone()
1 equals(Object)
2 finalize()
3 getClass()
4 hashCode()
5 notify()
6 notifyAll()
7 toString()
8 wait()
9 wait(long)
10 eat ()
11 sayHello()
12 sayGoodbye ()
Object类型数据
Cat类型数据
class Cat implements Friendly {
public void eat() {
}
public void sayHello() {
}
public void sayGoodbye() {
}
protected void finalize() {
}
public String toString(){}
}
class CockerSpaniel extends Dog impl {
public void sayHello() { super.
public void sayGoodbye() {
}
(313 , 547)
255, 255, 255
按 C 复制颜色值

9- 方法返回地址
(Return Address)



方法返回地址(return address)

- 存放调用该方法的pc寄存器的值。
- 一个方法的结束，有两种方式：
  - 正常执行完成
  - 出现未处理的异常，非正常退出
- 无论通过哪种方式退出，在方法退出后都返回到该方法被调用的位置。方法正常退出时，调用者的pc计数器的值作为返回地址，即调用该方法的指令的下一条指令的地址。而通过异常退出的，返回地址是要通过异常表来确定，栈帧中一般不会保存这部分信息。

方法返回地址(return address)

本质上，方法的退出就是当前栈帧出栈的过程。此时，需要恢复上层方法的局部变量表、操作数栈、将返回值压入调用者栈帧的操作数栈、设置PC寄存器值等，让调用者方法继续执行下去。

正常完成出口和异常完成出口的区别在于：通过异常完成出口退出的不会给他的上层调用者产生任何的返回值。

方法返回地址(return address)

当一个方法开始执行后，只有两种方式可以退出这个方法：

1、执行引擎遇到任意一个方法返回的字节码指令（return），会有返回值传递给上层的方法调用者，简称正常完成出口；

- 一个方法在正常调用完成之后究竟需要使用哪一个返回指令还需要根据方法返回值的实际数据类型而定。
- 在字节码指令中，返回指令包含ireturn（当返回值是boolean、byte、char、short和int类型时使用）、lreturn、freturn、dreturn以及areturn，另外还有一个return指令供声明为void的方法、实例初始化方法、类和接口的初始化方法使用。

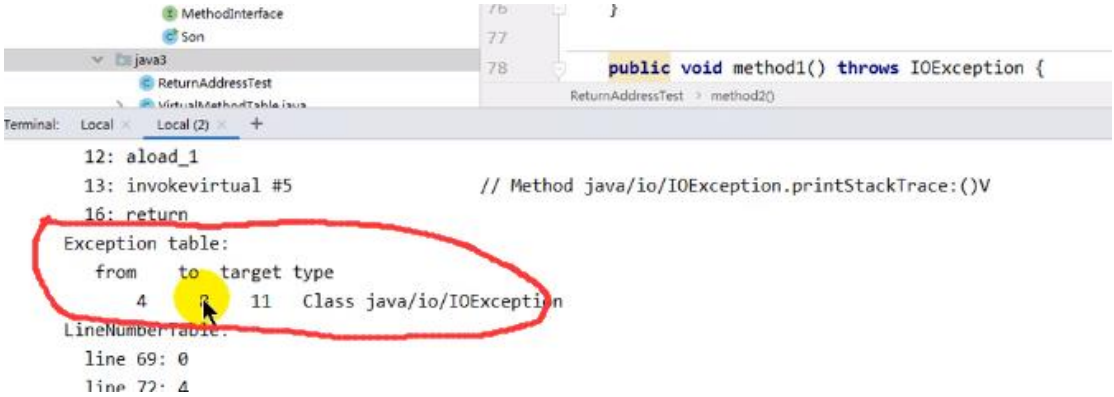
方法返回地址(return address)

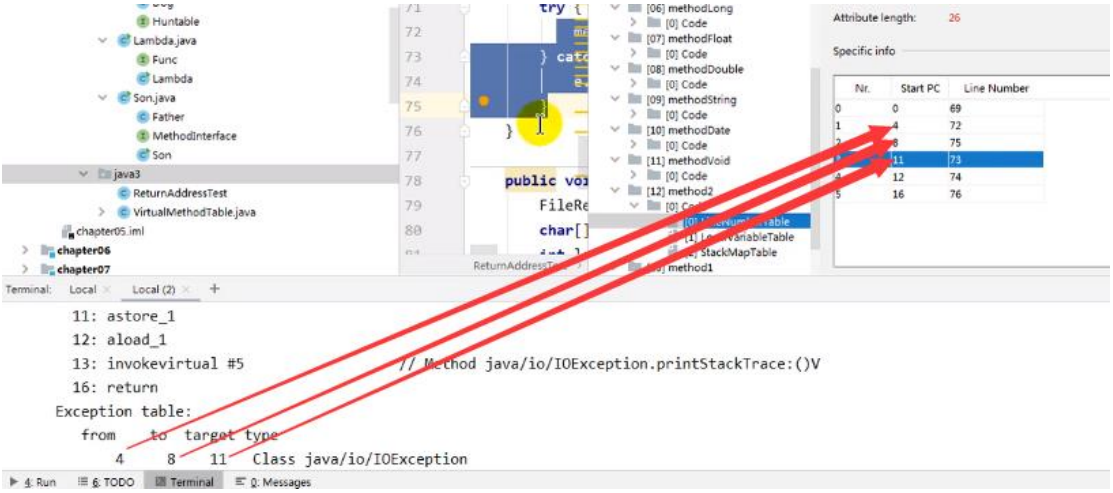
2、在方法执行的过程中遇到了异常（Exception），并且这个异常没有在方法内进行处理，也就是只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出。简称异常完成出口。

方法执行过程中抛出异常时的异常处理，存储在一个异常处理表，方便在发生异常的时候找到处理异常的代码。

```
Exception      table:
from    to      target  type
4       16      19      any
19      21      19      any
```

if the exception happened from 4 to 16,it should be handled by processor in line 19

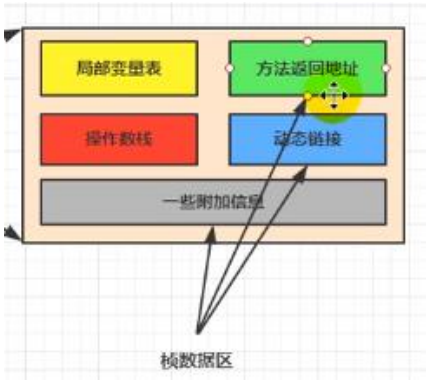




方法返回地址(return address)

本质上，方法的退出就是当前栈帧出栈的过程。此时，需要恢复上层方法的局部变量表、操作数栈、将返回值压入调用者栈帧的操作数栈、设置PC寄存器值等，让调用者方法继续执行下去。

正常完成出口和异常完成出口的区别在于：通过异常完成出口退出的不会给他的上层调用者产生任何的返回值。



10- 一些附加信息

栈帧中还允许携带与Java虚拟机实现相关的一些附加信息。例如，对程序调试提供支持的信息。

11- 栈的相关面试题