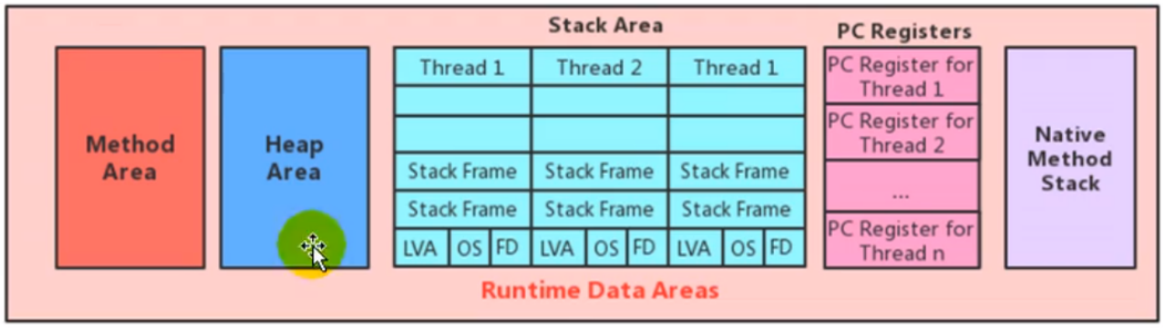
四、程序计数器（PC寄存器）

1、PC Register介绍

1.1 官方文档

- https://docs.oracle.com/javase/specs/index.html

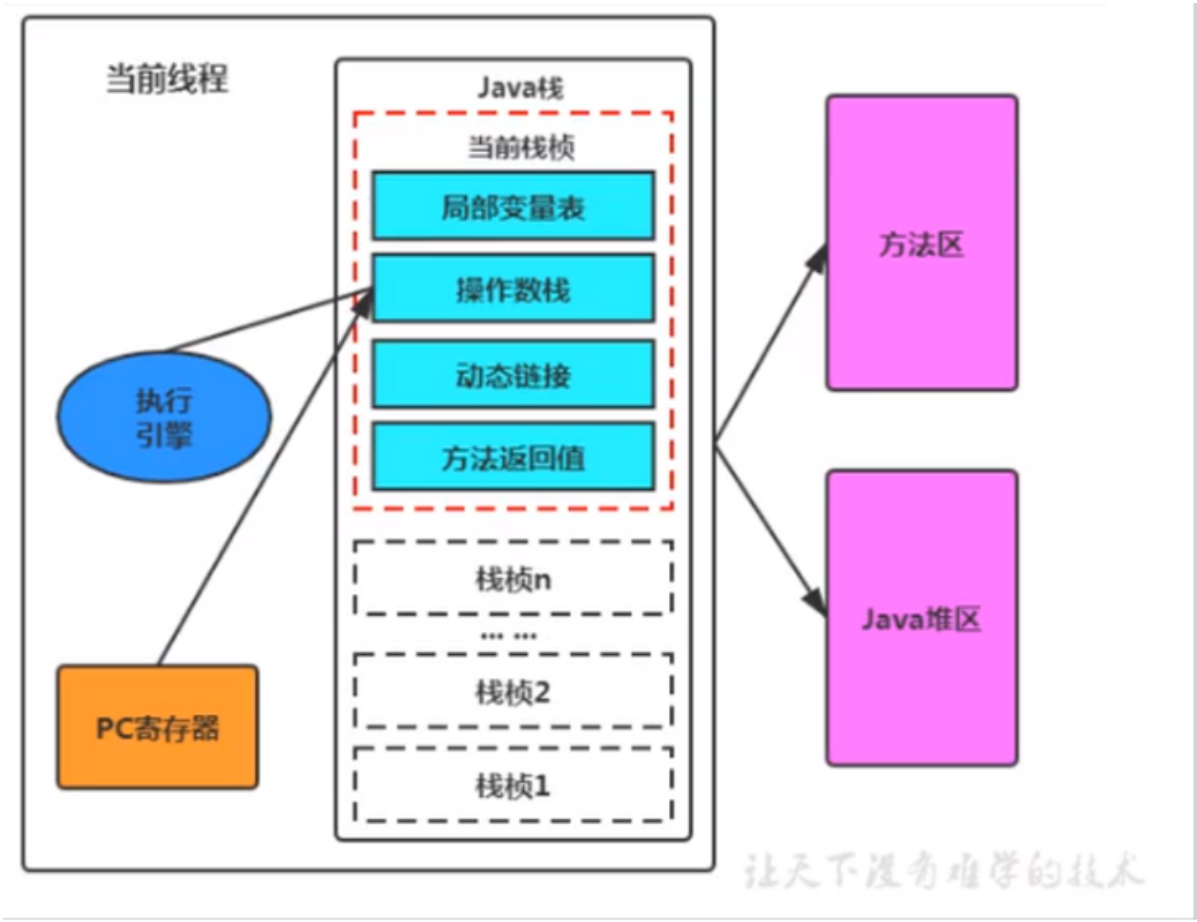
1.2 PC Register介绍



JVM中的程序计数寄存器(Program Counter Register，直译：程序计数寄存器，或简称PC寄存器、程序计数器)中，Register的命名源于CPU的寄存器（JVM是对物理CPU寄存器的抽象模拟，是一个软件），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，程序理解成线程中执行的一行行代码，钩子专门用来勾线程中一行行运行的代码，也可以称为行号指示器，即上一行代码执行完了，下一行代码该执行哪一个，由程序计数器来记录)，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。

1.2 PC寄存器的作用

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。



PC寄存器是每一个线程都有一份，相应的指令程序会被分配到栈的栈帧中，一个栈帧对应一个方法，其中具体的指令都会有一个行号的标识，PC寄存器是指向下一个需要执行指令的行号标识，执行引擎根据指向的行号获取指令，并执行，执行完后再去PC寄存器中取下一条指令。

1.3 特点

- 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)
- 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0utOtMemoryError（OOM）情况的区域
- 它也没有GC（垃圾回收机制）

2、举例说明

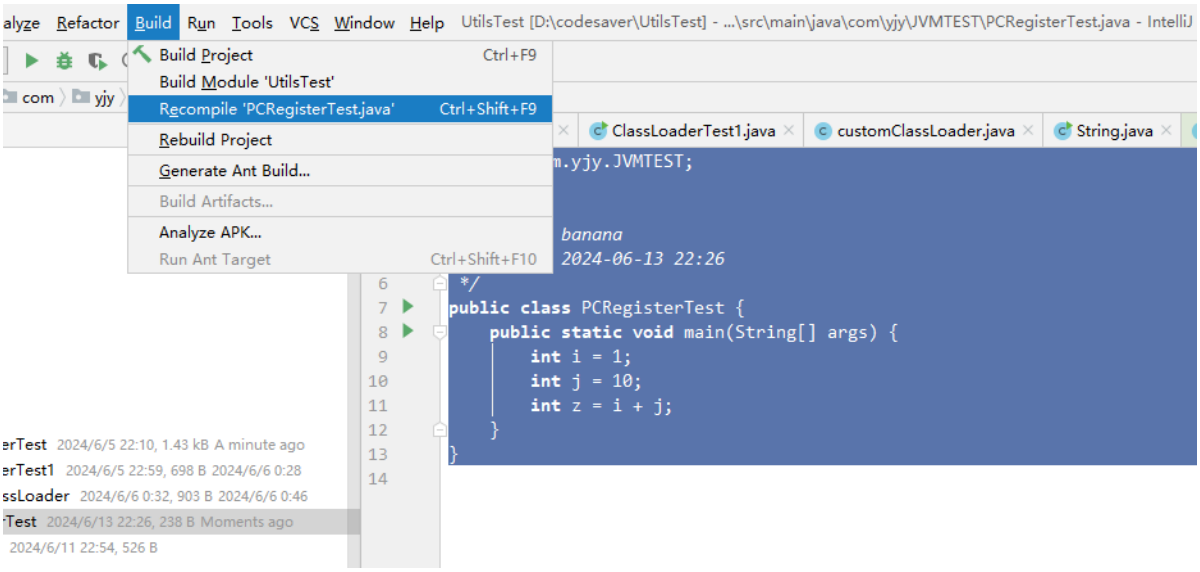
2.1 举例一

编写一个测试代码

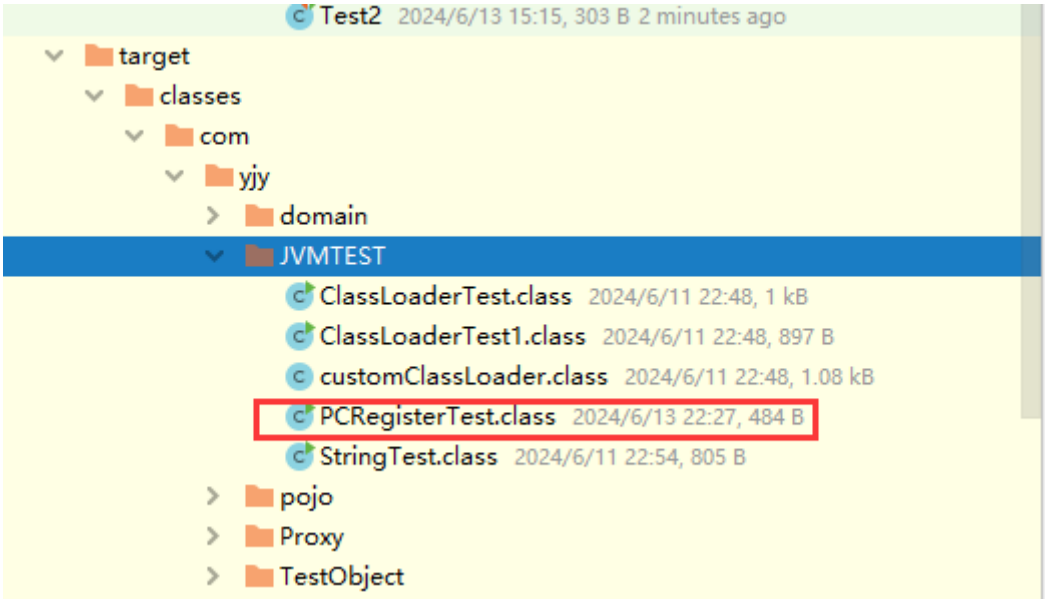
```
package com.yjy.JVMTEST;

/**
```

```
 * @author banana
 * @create 2024-06-13 22:26
 */
public class PCRegisterTest {
    public static void main(String[] args) {
        int i = 1;
        int j = 10;
        int z = i + j;
    }
}
```

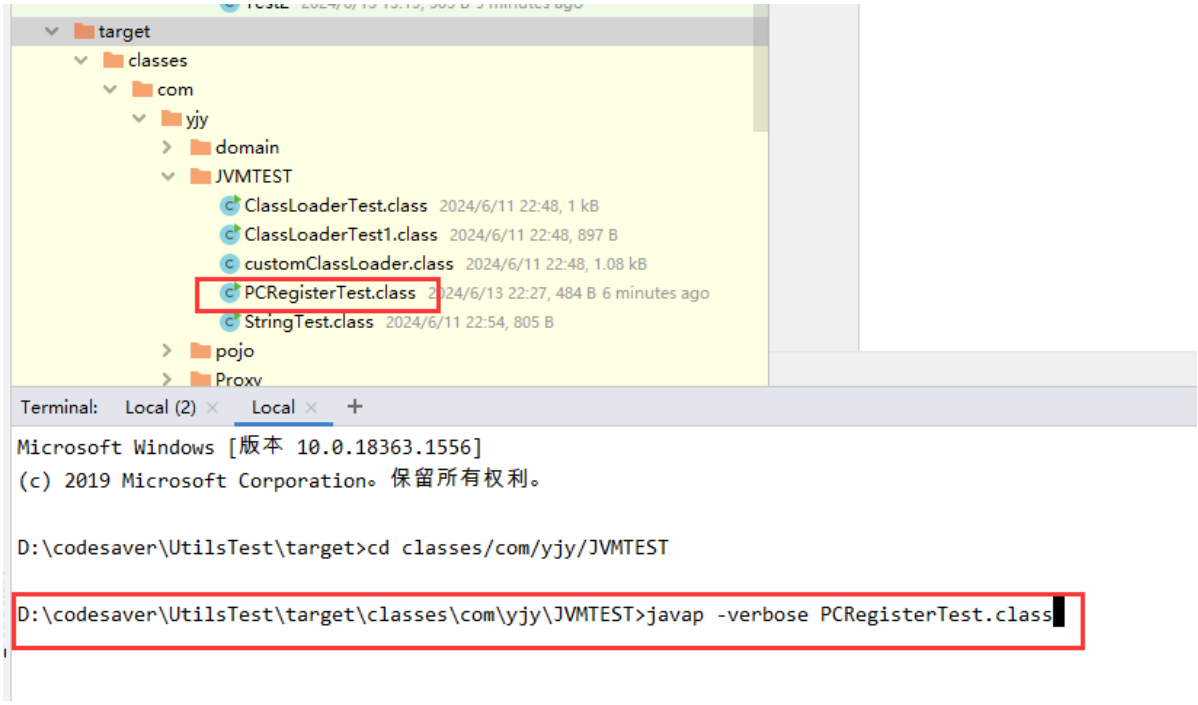
编译这个代码



刷新后即可看到变异后的字节码文件



进入指定目录，反编译字节码文件



其中内容如下所示

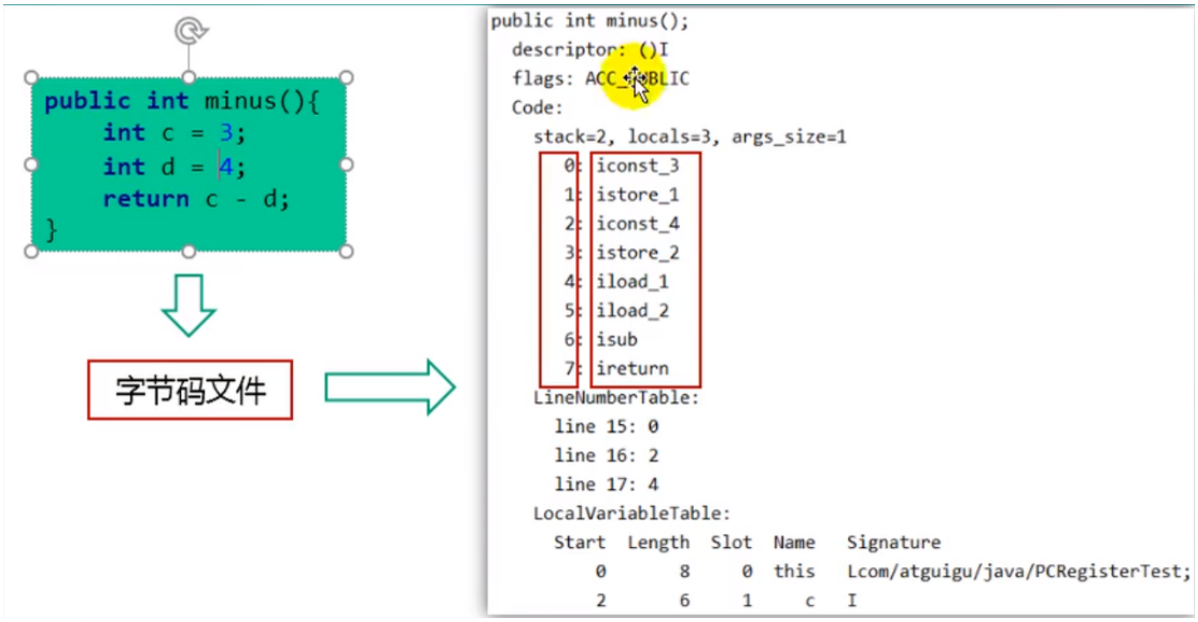
```
Classfile
/D:/codesaver/UtilsTest/target/classes/com/yjy/JVMTEST/PCRegisterTest.class
  Last modified 2024-6-13; size 484 bytes
  MD5 checksum 6e9b42eaebd139ba2a8bcdcca512e48d
  Compiled from "PCRegisterTest.java"
public class com.yjy.JVMTEST.PCRegisterTest
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #3.#21         // java/lang/Object."<init>":()V
   #2 = Class              #22            // com/yjy/JVMTEST/PCRegisterTest
   #3 = Class              #23            // java/lang/Object
   #4 = Utf8               <init>
   #5 = Utf8               ()V
   #6 = Utf8               Code
   #7 = Utf8               LineNumberTable
   #8 = Utf8               LocalVariableTable
   #9 = Utf8               this
  #10 = Utf8               Lcom/yjy/JVMTEST/PCRegisterTest;
  #11 = Utf8               main
  #12 = Utf8               ([Ljava/lang/String;)V
  #13 = Utf8               args
  #14 = Utf8               [Ljava/lang/String;
  #15 = Utf8               i
  #16 = Utf8               I
  #17 = Utf8               j
  #18 = Utf8               z
  #19 = Utf8               SourceFile
  #20 = Utf8               PCRegisterTest.java
  #21 = NameAndType        #4:#5          // "<init>":()V
  #22 = Utf8               com/yjy/JVMTEST/PCRegisterTest
  #23 = Utf8               java/lang/Object
{
```

```
  public com.yjy.JVMTEST.PCRegisterTest();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."
<init>":()V
         4: return
      LineNumberTable:
        line 7: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Lcom/yjy/JVMTEST/PCRegisterTest;

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=4, args_size=1
         0: iconst_1            // 取数1
         1: istore_1            // 将其保存到索引为1的位置
         2: bipush        10    // 取数10
         4: istore_2            // 将其保存到索引为2的位置
         5: iload_1             // 取出索引为1位置的内容
         6: iload_2             // 取出索引为2位置的内容
         7: iadd                // 将其求和
         8: istore_3            // 将求和结果放到索引为3的位置
         9: return              // maian方法结束
      LineNumberTable:
        line 9: 0
        line 10: 2
        line 11: 5
        line 12: 9
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      10     0  args   [Ljava/lang/String;
            2       8     1     i   I
            5       5     2     j   I
            9       1     3     z   I
}
SourceFile: "PCRegisterTest.java"
```

关于如下前面的第一列数字就是指令地址或偏移地址（即PC寄存器中存储的数据，执行器会根据PC寄存器中存储的指令地址取出对应的操作指令，然后操作局部变量表、操作数栈、将字节码指令翻译成机器指令，然后让CPU帮我们去执行这些机器指令），第二列则是操作指令

```
Code:
  stack=2, locals=4, args_size=1
     0: iconst_1
     1: istore_1
     2: bipush        10
     4: istore_2
     5: iload_1
     6: iload_2
     7: iadd
     8: istore_3
     9: return
```

概括图

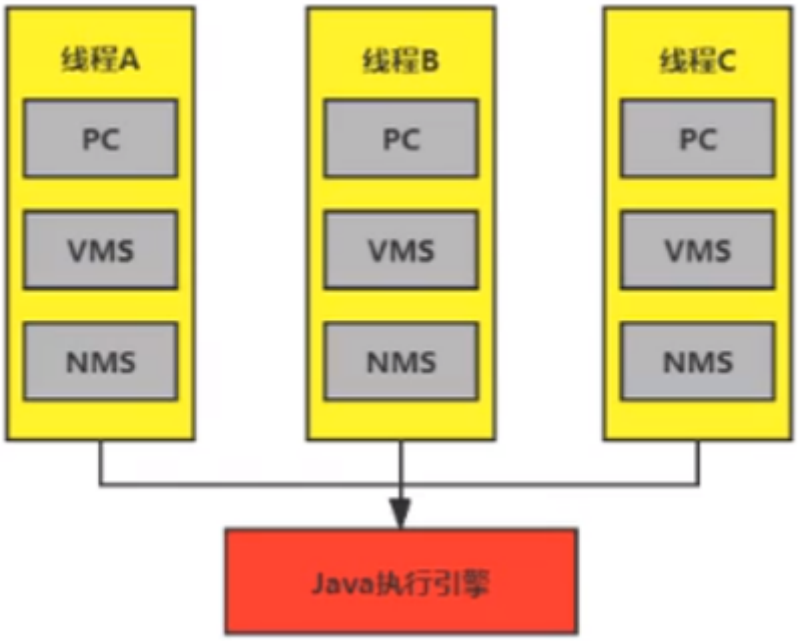


3、两个常见问题

3.1 问题一

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。



3.2 问题二

PC寄存器为什么会被设定为现线程私有的?

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。

3.3 CPU时间片

并行：相对于串行，串行就是一次只能做一件事，并行就是一次可以一起做几件事。

并发：CPU快速切换执行，看上去是并行，其实是串行。

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。