字节代码的组织形式：

8 bipush 100 // Push int constant 100

8：当前命令的地址偏移量（8是经过编译器优化后的，易于人使用的）

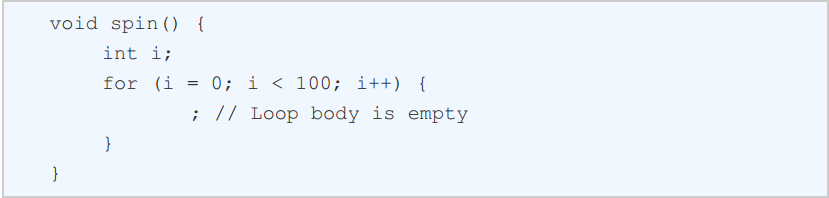
Bipush：字节代码指令

100 ：参数

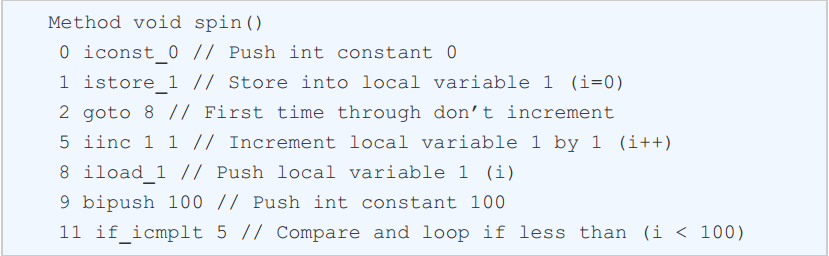
在每一行中，在表示运行时常量池索引的操作数前，会井号（’#’）开头

astore\_1 1表示的是变量的索引

spin()是一个很简单的方法，它进行了 100 次空循环：



编译之后的字节代码：





（体会一下什么是基于栈的架构设计和基于寄存器的架构设计）

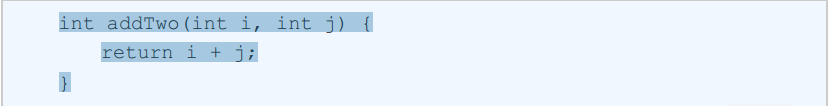
Java 虚拟机是基于栈架构设计的， 它的大多数操作是从当前栈帧的操作数栈取出 1 个或多个操作数，或将结果压入操作数栈中。每调用一个方法，都会创建一个新的栈帧，并创建对应方法所需的操作数栈和局部变量表（参见§2.6 “栈帧”） 。每条线程在运行时的任意时刻，都会包含若

干个由不同方法嵌套调用而产生的栈帧，当然也包括了若干个栈帧内部的操作数栈， 但是只有当前栈帧中的操作数栈才是活动的。

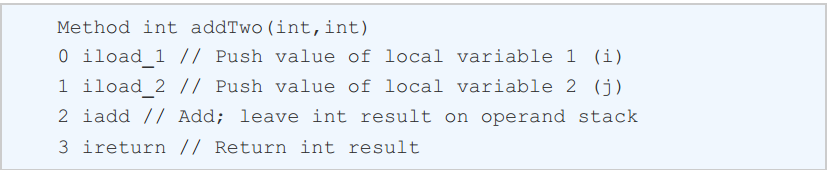
在 Java 虚拟机中，缺乏对 byte、char 和 short 类型数据直接操作的支持所带来的问题并不大，因为这些类型的值都在编译过程中就自动被转换为 int 类型（byte 和 short 带符号扩展为 int 类型，char 零位扩展为 int 类型） 。因此对于 byte、char 和 short 类型的数据均可以用 int 的指令操作。唯一额外的代价是将它们长度扩展至 4 字节。

接收参数

如果传递了 n 个参数给某个实例方法，则当前栈帧会按照约定的顺序接收这些参数，将它们保存为方法的第 1 个至第 n 个局部变量之中。譬如：

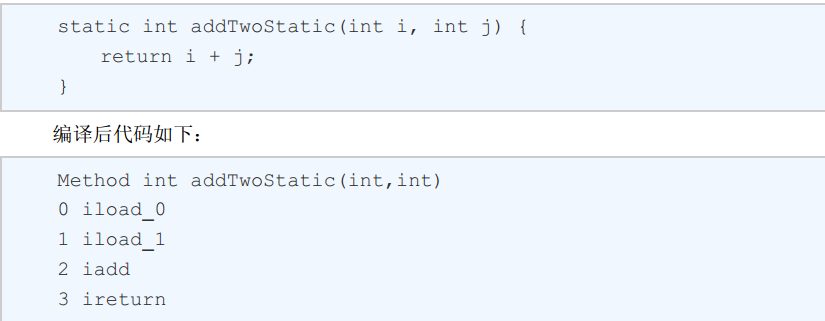


编译后代码如下：

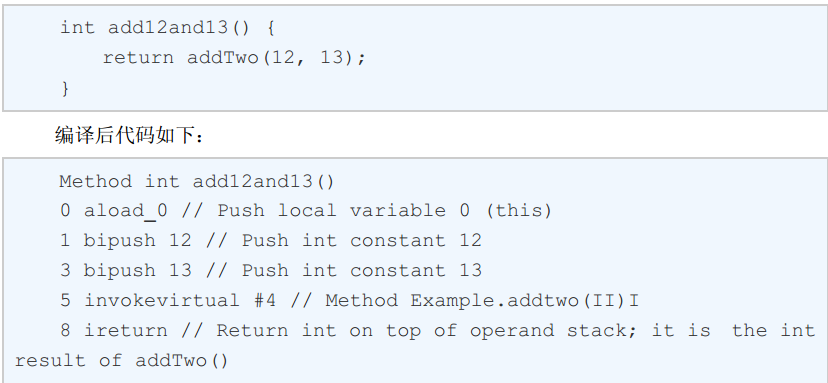


其中iload中的1 时表示局部变量表中的第一个（第0个用来存储this（当前对象自身的引用））

改为static之后的区别（iload之后的数组）

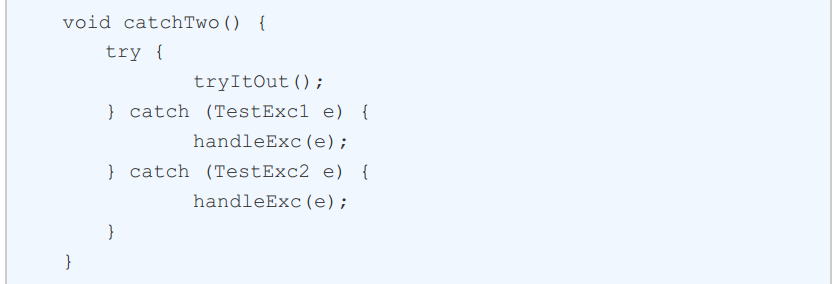


方法调用

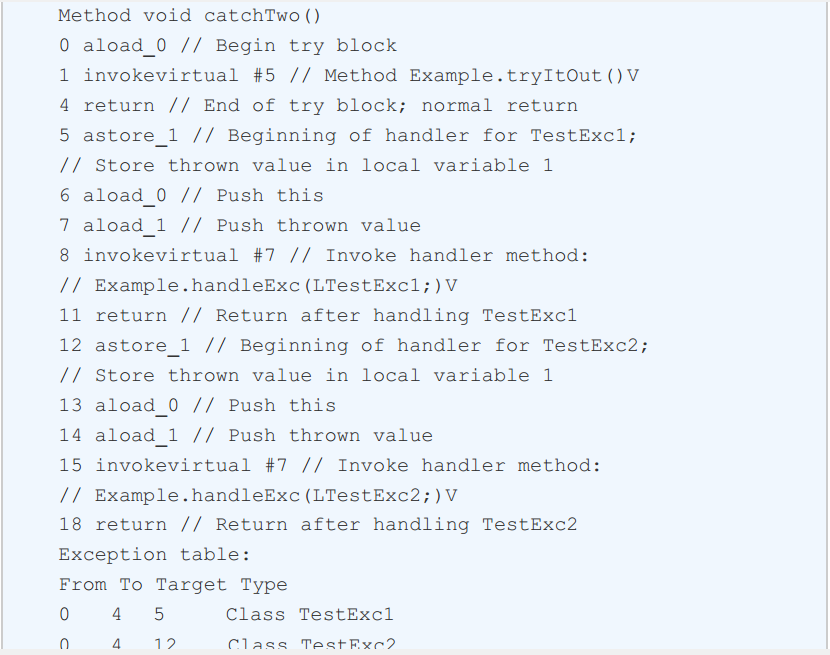


方法调用过程的第一步是将当前实例的自身引用“this”压入到操作数栈中。传递给方法的参数，int 值 12 和 13 随后入栈。当调用 addTwo()方法时，Java 虚拟机会创建一个新的栈帧，传递给 addTwo()方法的参数作为新的帧的对应局部变量的初始值。即 this 和两个传递给的addTwo()方法的参数 12 和 13 被作为 addTwo()方法栈帧的第 0、1、2 个局部变量。

虚拟机的异常处理：



编译之后为：



最后的Exception table 为一场表（在上面的方法中可能出现的异常），如果陈旭运行时不出现异常，则可以当try语句块不存在，finally语句的实现是通过：9 jsr 14 // Call finally block实现的