

课程主题

直接内存、Java虚拟机栈、本地方法栈、程序计数器、方法执行

课程目标

课程回顾

课程内容

一、直接内存

概述

直接内存又叫堆外内存。它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。

在JDK1.4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堆中的[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。

直接内存与堆内存比较

1. 直接内存申请空间耗费更高的性能，当频繁申请到一定量时尤为明显
2. 直接内存IO读写的性能要优于普通的堆内存，在多次读写操作的情况下差异明显
3. 本机直接内存的分配不会受到Java 堆大小的限制，受到本机总内存大小限制
4. 配置虚拟机参数时，不要忽略直接内存 防止出现OutOfMemoryError异常

直接内存的实现

Java中分配堆外内存的方式有两种：

- 一是通过 `ByteBuffer.java#allocateDirect` 得到一个 `DirectByteBuffer` 对象
- 二是直接调用 `Unsafe.java#allocateMemory` 分配内存，但 `Unsafe` 只能在JDK的代码中调用，一般不会直接使用该方法分配内存。

其中 `DirectByteBuffer` 也是用 `Unsafe` 去实现内存分配的，对堆内存的分配、读写、回收都做了封装。

我们从堆外内存的分配回收、读写两个角度去分析 `DirectByteBuffer`。

ByteBuffer源码

```
package java.nio;

public abstract class ByteBuffer
```

```

    extends Buffer
    implements Comparable<ByteBuffer>
{

    /**
     * Allocates a new direct byte buffer.
     *
     * <p> The new buffer's position will be zero, its
    limit will be its
     * capacity, its mark will be undefined, and each of
    its elements will be
     * initialized to zero. Whether or not it has a
     * {@link #hasArray backing array} is unspecified.
     *
     * @param capacity
     *     The new buffer's capacity, in bytes
     * @return The new byte buffer
     *
     * @throws IllegalArgumentException
     *     If the <tt>capacity</tt> is a negative
    integer
     */
    public static ByteBuffer allocateDirect(int capacity)
    {
        return new DirectByteBuffer(capacity);
    }
}

```

`ByteBuffer#allocateDirect` 中仅仅是创建了一个 `DirectByteBuffer` 对象，重点在 `DirectByteBuffer` 的构造方法中。

```
DirectByteBuffer(int cap) { // package-private
private
    //主要是调用ByteBuffer的构造方法，为字段赋值
    super(-1, 0, cap, cap);
    //如果是按页对齐，则还要加一个Page的大小；我们分析只pa为false的情况就好了
    boolean pa = VM.isDirectMemoryPageAligned();
    int ps = Bits.pageSize();
    long size = Math.max(1L, (long)cap + (pa ? ps : 0));
    //预分配内存
    Bits.reserveMemory(size, cap);

    long base = 0;
    try {
        //分配内存
        base = unsafe.allocateMemory(size);
    } catch (OutOfMemoryError x) {
        Bits.unreserveMemory(size, cap);
        throw x;
    }
    //将分配的内存的所有值赋值为0
    unsafe.setMemory(base, size, (byte) 0);
    //为address赋值，address就是分配内存的起始地址，之后的数据读写都是以它作为基准
    if (pa && (base % ps != 0)) {
        // Round up to page boundary
        address = base + ps - (base & (ps - 1));
    } else {
        //pa为false的情况，address==base
        address = base;
    }
    //创建一个Cleaner，将this和一个Deallocator对象传进去
```

```
        cleaner = Cleaner.create(this, new Deallocator(base,
size, cap));
        att = null;

    }
```

DirectByteBuffer构造方法中还做了挺多事情的，总的来说分为几个步骤：

1. 预分配内存
2. 分配内存
3. 将刚分配的内存空间初始化为0
4. 创建一个cleaner对象，Cleaner对象的作用是当DirectByteBuffer对象被回收时，释放其对应的堆外内存。

Java的堆外内存回收设计是这样的：当GC发现DirectByteBuffer对象变成垃圾时，会调用 `Cleaner#clean` 回收对应的堆外内存，一定程度上防止了内存泄露。当然，也可以手动的调用该方法，对堆外内存进行提前回收。

Cleaner的实现

我们先看下 `Cleaner#clean` 的实现：

```
public class Cleaner extends PhantomReference<Object> {
    // ...
    private Cleaner(Object referent, Runnable thunk) {
        super(referent, dummyQueue);
        this.thunk = thunk;
    }
    public void clean() {
        if (remove(this)) {
```

```
        try {
            //thunk是一个Deallocator对象
            this.thunk.run();
        } catch (final Throwable var2) {
            ...
        }
    }
}
```

```
private static class Deallocator
    implements Runnable
{
    private static Unsafe unsafe = Unsafe.getUnsafe();

    private long address;
    private long size;
    private int capacity;

    private Deallocator(long address, long size, int
capacity) {
        assert (address != 0);
        this.address = address;
        this.size = size;
        this.capacity = capacity;
    }

    public void run() {
```

```
        if (address == 0) {
            // Paranoia
            return;
        }
        unsafe.freeMemory(address);
        address = 0;
        Bits.unreserveMemory(size, capacity);
    }
```

Cleaner继承自PhantomReference

简单的说，就是当字段referent(也就是DirectByteBuffer对象)被回收时，会调用到Cleaner#clean方法，最终会调用到Deallocator#run进行堆外内存的回收。

Cleaner是虚引用在JDK中的一个典型应用场景。

预分配内存源码

然后再看下DirectByteBuffer构造方法中的第二步，reserveMemory

```
static void reserveMemory(long size, int cap) {
    //maxMemory代表最大堆外内存，也就是-
    XX:MaxDirectMemorySize指定的值
    if (!memoryLimitSet && VM.isBooted()) {
        maxMemory = VM.maxDirectMemory();
        memoryLimitSet = true;
    }

    //1.如果堆外内存还有空间，则直接返回
```

```
    if (tryReserveMemory(size, cap)) {  
        return;  
    }  
    //走到这里说明堆外内存剩余空间已经不足了  
    final JavaLangRefAccess jlra =  
SharedSecrets.getJavaLangRefAccess();
```

//2.堆外内存进行回收，最终会调用到Cleaner#clean的方法。如果目前没有堆外内存可以回收则跳过该循环

```
    while (jlra.tryHandlePendingReference()) {  
        //如果空闲的内存足够了，则return  
        if (tryReserveMemory(size, cap)) {  
            return;  
        }  
    }  
}
```

//3.主动触发一次GC，目的是触发老年代GC

```
System.gc();
```

//4.重复上面的过程

```
boolean interrupted = false;  
try {  
    long sleepTime = 1;  
    int sleeps = 0;  
    while (true) {  
        if (tryReserveMemory(size, cap)) {  
            return;  
        }  
        if (sleeps >= MAX_SLEEPS) {  
            break;  
        }  
        if (!jlra.tryHandlePendingReference()) {  
            try {
```



```
        Thread.sleep(sleepTime);
        sleepTime <= 1;
        sleeps++;
    } catch (InterruptedException e) {
        interrupted = true;
    }
}
}
```

//5.超出指定的次数后，还是没有足够内存，则抛异常

```
throw new OutOfMemoryError("Direct buffer
memory");
```

```
    } finally {
        if (interrupted) {
            // don't swallow interrupts
            Thread.currentThread().interrupt();
        }
    }
}
```

```
private static boolean tryReserveMemory(long size, int
cap) {
    //size和cap主要是page对齐的区别，这里我们把这两个值看作是
    相等的
    long totalCap;
    //totalCapacity代表通过DirectByteBuffer分配的堆外内存
    的大小
    //当已分配大小<=还剩下的堆外内存大小时，更新totalCapacity
    的值返回true
    while (cap <= maxMemory - (totalCap =
totalCapacity.get())) {
```

```
        if (totalCapacity.compareAndSet(totalCap,
totalCap + cap)) {
            reservedMemory.addAndGet(size);
            count.incrementAndGet();
            return true;
        }
    }
    //堆外内存不足, 返回false
    return false;
}
```

在创建一个新的DirecByteBuffer时，会先确认有没有足够的内存，如果没有的话，会通过一些手段回收一部分堆外内存，直到可用内存大于需要分配的内存。具体步骤如下：

1. 如果可用堆外内存足够，则直接返回
2. 调用 `tryHandlePendingReference` 方法回收已经变成垃圾的 DirectByteBuffer 对象对应的堆外内存，直到可用内存足够，或目前没有垃圾 DirectByteBuffer 对象
3. 触发一次 full gc，其主要目的是为了防止‘冰山现象’：一个 DirectByteBuffer 对象本身占用的内存很小，但是它可能引用了一块很大的堆外内存。如果 DirectByteBuffer 对象进入了老年代之后变成了垃圾，因为老年代 GC 一直没有触发，导致这块堆外内存也一直没有被回收。需要注意的是如果使用参数 `-XX:+DisableExplicitGC`，那 `System.gc()` 是无效的
4. 重复 1，2 步骤的流程，直到可用内存大于需要分配的内存
5. 如果超出指定次数还没有回收到足够内存，则 OOM

详细分析下第2步是如何回收垃圾的：`tryHandlePendingReference`最终调用到的是`Reference#tryHandlePending`方法

```
static boolean tryHandlePending(boolean waitForNotify) {
    Reference<Object> r;
    Cleaner c;
    try {
        synchronized (lock) {
            //pending由jvm gc时设置
            if (pending != null) {
                r = pending;
                // 如果是cleaner对象，则记录下来
                c = r instanceof Cleaner ? (Cleaner) r
: null;

                // unlink 'r' from 'pending' chain
                pending = r.discovered;
                r.discovered = null;
            } else {
                // waitForNotify传入的值为false
                if (waitForNotify) {
                    lock.wait();
                }
                // 如果没有待回收的Reference对象，则返回
                false

                return waitForNotify;
            }
        }
    } catch (OutOfMemoryError x) {
        ...
    } catch (InterruptedException x) {
        ...
    }
}
```

```

        // Fast path for cleaners
        if (c != null) {
            //调用clean方法
            c.clean();
            return true;
        }

        ...
        return true;
    }
}

```

可以看到，`tryHandlePendingReference`的最终效果就是：如果有垃圾`DirectByteBuffer`对象，则调用对应的`Cleaner#clean`方法进行回收。`clean`方法在上面已经分析过了。

堆外内存的读写API

```

public ByteBuffer put(byte x) {
    unsafe.putByte(ix(nextPutIndex()), ((x)));
    return this;
}

final int nextPutIndex() {
    if (position >= limit)
        throw new BufferOverflowException();
    return position++;
}

private long ix(int i) {
    return address + ((long)i << 0);
}

```

```

public byte get() {
    return ((unsafe.getBytes(ix(nextGetIndex()))));
}

final int nextGetIndex() {                                     //
package-private
    if (position >= limit)
        throw new BufferUnderflowException();
    return position++;
}

```

读写的逻辑也比较简单，address就是构造方法中分配的native内存的起始地址。Unsafe的putByte/getByte都是native方法，就是写入值到某个地址/获取某个地址的值。



案例演示

代码

```

import java.nio.ByteBuffer;

/**
 * 直接内存 与 堆内存的比较
 */
public class ByteBufferCompare {
    public static void main(String[] args) {
        allocateCompare();    //分配比较
        operateCompare();     //读写比较
    }
}

```

```

/**
 * 直接内存 和 堆内存的 分配空间比较
 *
 * 结论： 在数据量提升时，直接内存相比非直接内存的申请，有很严重的性能
问题
 *
 */
public static void allocateCompare(){
    int time = 10000000;    //操作次数
    long st = System.currentTimeMillis();
    for (int i = 0; i < time; i++) {

        //ByteBuffer.allocate(int capacity)    分配一个新的字节
缓冲区。
        ByteBuffer buffer = ByteBuffer.allocate(2);    //非
直接内存分配申请
    }
    long et = System.currentTimeMillis();

    System.out.println("在进行"+time+"次分配操作时，堆内存 分配
耗时：" + (et-st) + "ms" );

    long st_heap = System.currentTimeMillis();
    for (int i = 0; i < time; i++) {
        //ByteBuffer.allocateDirect(int capacity) 分配新的直接
字节缓冲区。
        ByteBuffer buffer = ByteBuffer.allocateDirect(2); //
直接内存分配申请
    }
    long et_direct = System.currentTimeMillis();

```

```
System.out.println("在进行"+time+"次分配操作时，直接内存 分  
配耗时：" + (et_direct-st_heap) +"ms" );  
  
}  
  
/**  
* 直接内存 和 堆内存的 读写性能比较  
*  
* 结论：直接内存在直接的IO 操作上，在频繁的读写时 会有显著的性能提升  
*  
*/  
public static void operateCompare(){  
    int time = 1000000000;  
  
    ByteBuffer buffer = ByteBuffer.allocate(2*time);  
    long st = System.currentTimeMillis();  
    for (int i = 0; i < time; i++) {  
        // putChar(char value) 用来写入 char 值的相对 put 方法  
        buffer.putChar('a');  
    }  
    buffer.flip();  
    for (int i = 0; i < time; i++) {  
        buffer.getChar();  
    }  
    long et = System.currentTimeMillis();  
  
    System.out.println("在进行"+time+"次读写操作时，非直接内存读  
写耗时：" + (et-st) +"ms");  
  
    ByteBuffer buffer_d =  
    ByteBuffer.allocateDirect(2*time);  
    long st_direct = System.currentTimeMillis();
```

```

    for (int i = 0; i < time; i++) {

        // putChar(char value) 用来写入 char 值的相对 put 方法
        buffer_d.putChar('a');
    }
    buffer_d.flip();
    for (int i = 0; i < time; i++) {
        buffer_d.getChar();
    }
    long et_direct = System.currentTimeMillis();

    System.out.println("在进行"+time+"次读写操作时，直接内存读写
耗时：" + (et_direct - st_direct) + "ms");
}
}

```

输出结果

在进行10000000次分配操作时，堆内存 分配耗时:12ms

在进行10000000次分配操作时，直接内存 分配耗时:8233ms

在进行1000000000次读写操作时，非直接内存读写耗时：4055ms

在进行1000000000次读写操作时，直接内存读写耗时:745ms

可以自己设置不同的time 值进行比较

分析

从数据流的角度，来看

堆内存作用链:

本地IO -> 直接内存 -> 堆内存 -> 直接内存 -> 本地IO

直接内存作用链:

本地IO -> 直接内存 -> 本地IO

直接内存使用场景

有很大的数据需要存储，它的生命周期很长

- 适合频繁的IO操作，例如网络并发场景
- 适合长期存在或能复用的场景

堆外内存分配回收也是有开销的，所以适合长期存在的对象

- 适合注重稳定的场景

堆外内存能有效避免因GC导致的暂停问题。

- 适合简单对象的存储

因为堆外内存只能存储字节数组，所以对于复杂的DTO对象，每次存储/读取都需要序列化/反序列化，

- 适合注重IO效率的场景

用堆外内存读写文件性能更好

文件IO

关于堆外内存IO为什么有更好的性能这点展开一下。

BIO

BIO的文件写 `FileOutputStream#write` 最终会调用到native层的 `io_util.c#writeBytes` 方法

```
void
writeBytes(JNIEnv *env, jobject this, jbyteArray bytes,
           jint off, jint len, jboolean append, jfieldID
fid)
```

```

{
    jint n;
    char stackBuf[BUF_SIZE];
    char *buf = NULL;
    FD fd;

    ...

    // 如果写入长度为0，直接返回0
    if (len == 0) {
        return;
    } else if (len > BUF_SIZE) {
        // 如果写入长度大于BUF_SIZE (8192)，无法使用栈空间
buffer
        // 需要调用malloc在堆空间申请buffer
        buf = malloc(len);
        if (buf == NULL) {
            JNU_ThrowOutOfMemoryError(env, NULL);
            return;
        }
    } else {
        buf = stackBuf;
    }

    // 复制Java传入的byte数组数据到C空间的buffer中
    (*env)->GetByteArrayRegion(env, bytes, off, len,
(jbyte *)buf);

    if (!(*env)->ExceptionOccurred(env)) {
        off = 0;
        while (len > 0) {
            fd = GET_FD(this, fid);
            if (fd == -1) {

```

```

        JNU_ThrowIOException(env, "Stream
Closed");

        break;
    }
    //写入到文件，这里传递的数组是我们新创建的buf
    if (append == JNI_TRUE) {
        n = (jint)IO_Append(fd, buf+off, len);
    } else {
        n = (jint)IO_Write(fd, buf+off, len);
    }
    if (n == JVM_IO_ERR) {
        JNU_ThrowIOExceptionWithLastError(env,
"Write error");
        break;
    } else if (n == JVM_IO_INTR) {
        JNU_ThrowByName(env,
"java/io/InterruptedIOException", NULL);
        break;
    }
    off += n;
    len -= n;
}
}
}

```

`GetByteArrayRegion` 其实就是对数组进行了一份拷贝，该函数的实现在 `jni.cpp` 宏定义中，找了很久才找到

```

//jni.cpp
JNI_ENTRY(void, \
jni_Get##Result##ArrayRegion(JNIEnv *env,
ElementType##Array array, jsize start, \
        jsize len, ElementType *buf)) \
...
    int sc = TypeArrayKlass::cast(src->klass())->
log2_element_size(); \
    //内存拷贝
    memcpy((u_char*) buf, \
        (u_char*) src->Tag##_at_addr(start), \
        len << sc); \
...
} \
JNI_END

```

可以看到，传统的BIO，在native层真正写文件前，会在堆外内存（c分配的内存）中对字节数组拷贝一份，之后真正IO时，使用的是堆外的数组。要这样做的原因是

- 1.底层通过write、read、pwrite，pread函数进行系统调用时，需要传入buffer的起始地址和buffer count作为参数。如果使用java heap的话，我们知道jvm中buffer往往以byte[] 的形式存在，这是一个特殊的对象，由于java heap GC的存在，这里对象在堆中的位置往往会发生移动，移动后我们传入系统函数的地址参数就不是真正的buffer地址了，这样的话无论读写都会发生出错。而C Heap仅仅受Full GC的影响，相对来说地址稳定。

- 2.JVM规范中没有要求Java的byte[] 必须是连续的内存空间，它往往受宿主语言的类型约束；而C Heap中我们分配的虚拟地址空间是可以连续的，而上述的系统调用要求我们使用连续的地址空间作为buffer。

BIO的文件读也一样，这里就不分析了。

NIO

NIO的文件写最终会调用到 `IOUtil#write`

```
static int write(FileDescriptor fd, ByteBuffer src, long
position,
                NativeDispatcher nd, Object lock)
    throws IOException
{
    //如果是堆外内存，则直接写
    if (src instanceof DirectBuffer)
        return writeFromNativeBuffer(fd, src,
position, nd, lock);

    // Substitute a native buffer
    int pos = src.position();
    int lim = src.limit();
    assert (pos <= lim);
    int rem = (pos <= lim ? lim - pos : 0);
    //创建一块堆外内存，并将数据赋值到堆外内存中去
    ByteBuffer bb =
Util.getTemporaryDirectBuffer(rem);
    try {
        bb.put(src);
        bb.flip();
        // Do not update src until we see how many
bytes were written
        src.position(pos);

        int n = writeFromNativeBuffer(fd, bb,
position, nd, lock);
        if (n > 0) {
            // now update src
```

```

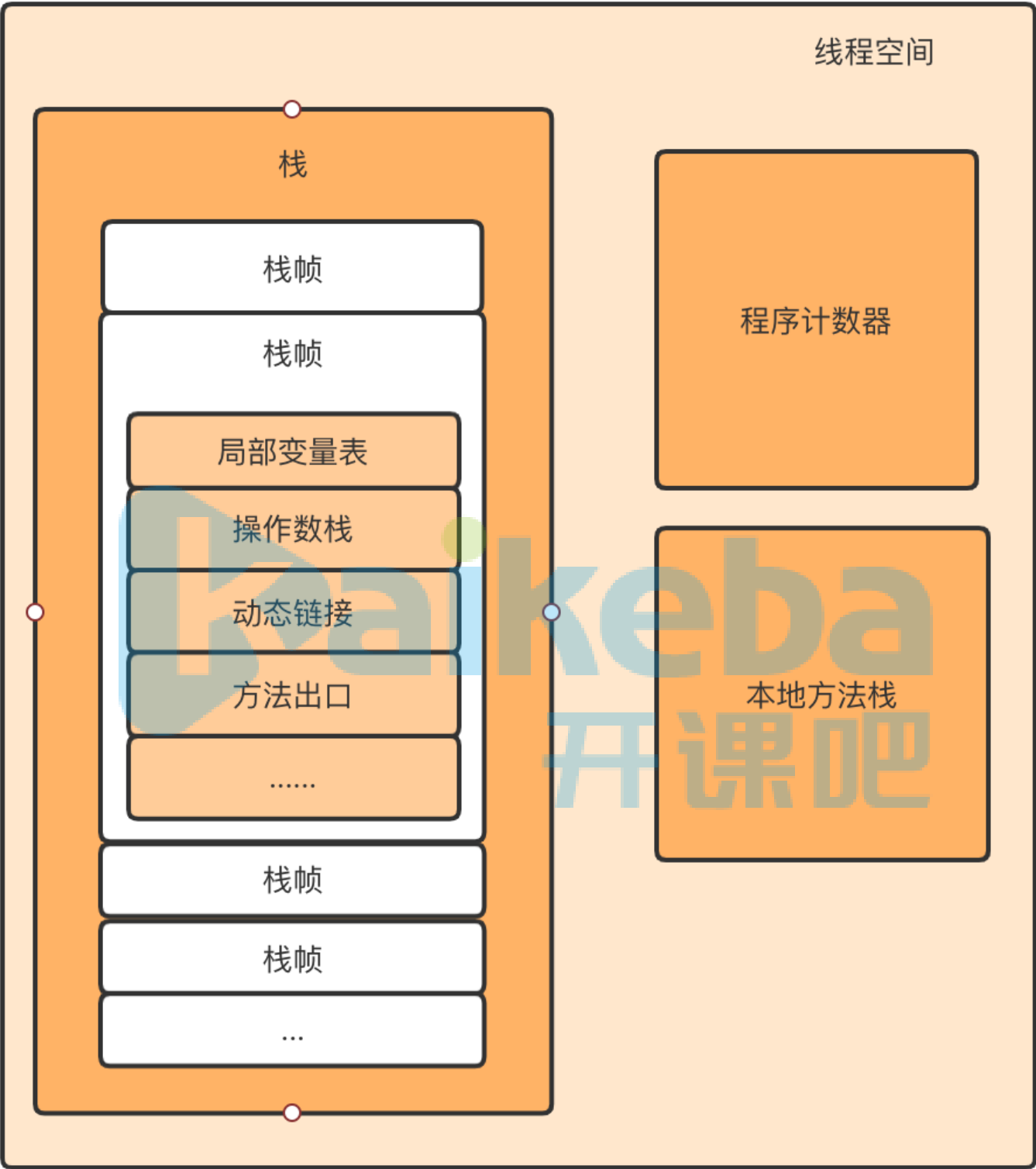
        src.position(pos + n);
    }
    return n;
} finally {
    Util.offerFirstTemporaryDirectBuffer(bb);
}
}

/**
 * 分配一片堆外内存
 */
static ByteBuffer getTemporaryDirectBuffer(int size) {
    BufferCache cache = bufferCache.get();
    ByteBuffer buf = cache.get(size);
    if (buf != null) {
        return buf;
    } else {
        // No suitable buffer in the cache so we need
to allocate a new
        // one. To avoid the cache growing then we
remove the first
        // buffer from the cache and free it.
        if (!cache.isEmpty()) {
            buf = cache.removeFirst();
            free(buf);
        }
        return ByteBuffer.allocateDirect(size);
    }
}
}

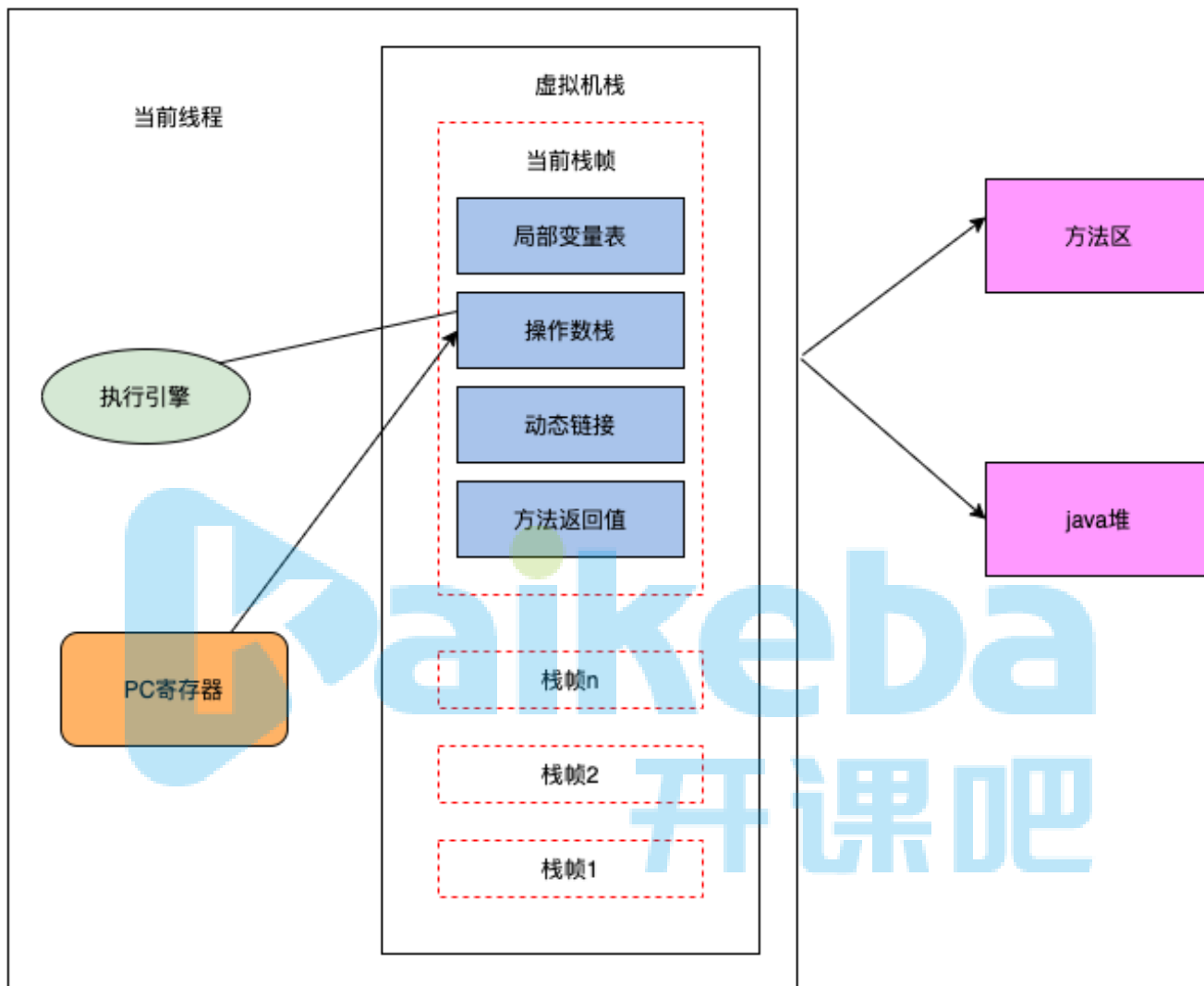
```

可以看到，NIO的文件写，对于堆内内存来说也是会有一次额外的内存拷贝的。

二、程序计数器



作用



示例

```
//javac PCRegisterTest.java
//java PCRegisterTest
//javap -verbose PCRegisterTest

package com.kkb.jvmdemo.example;
```

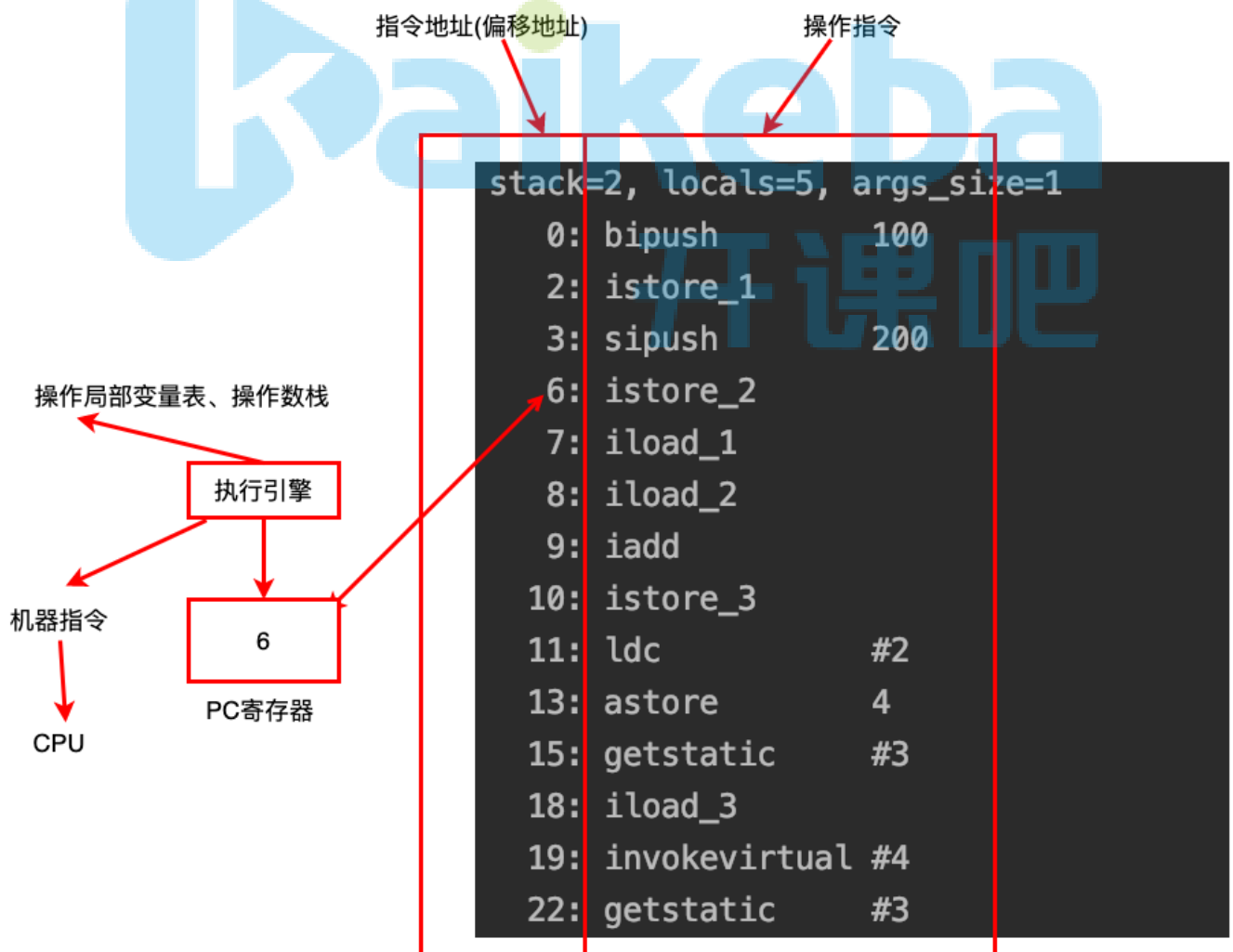


```

public class PCRegisterTest {
    public static void main(String[] args){
        int i = 100;
        int j = 200;
        int m = i + j;

        String str = "a";
        System.out.println(m);
        System.out.println(str);
    }
}

```



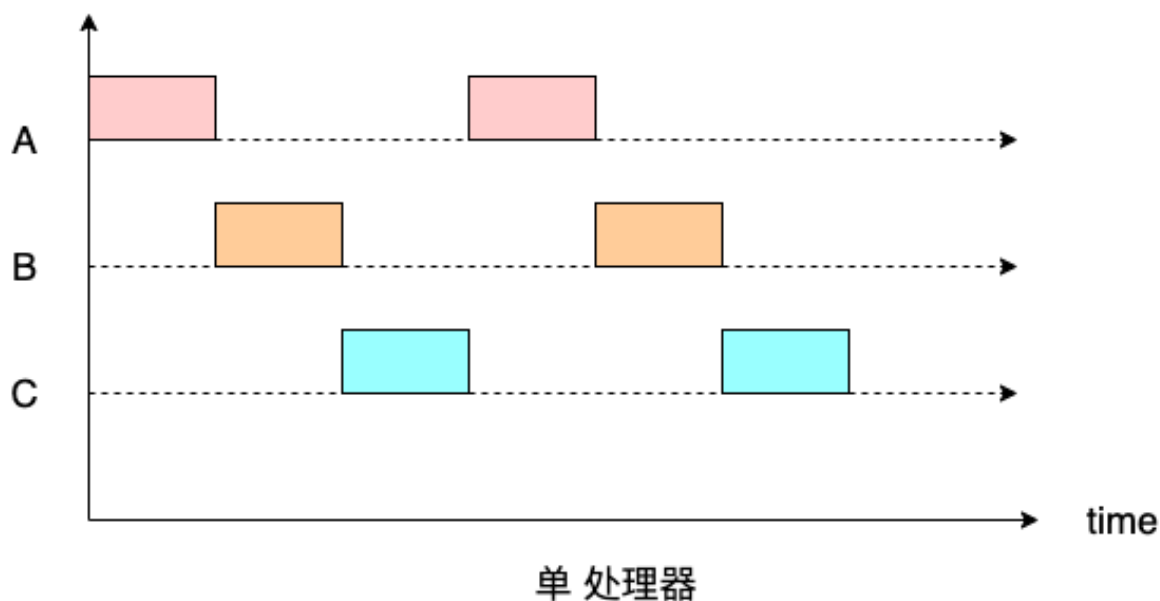
特点

- 它是一块很小的内存空间，几乎可以忽略不计。也是运行速度最快的存储区域。
- 在JVM规范中，每个线程都有它的程序计数器，是线程私有的，它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JVM指令地址；或者，如果是执行native方法，则是未指定值(undefined)。
- 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读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OutOfMemoryError(OOM)情况的区域。

面试问题

1、并发和并行

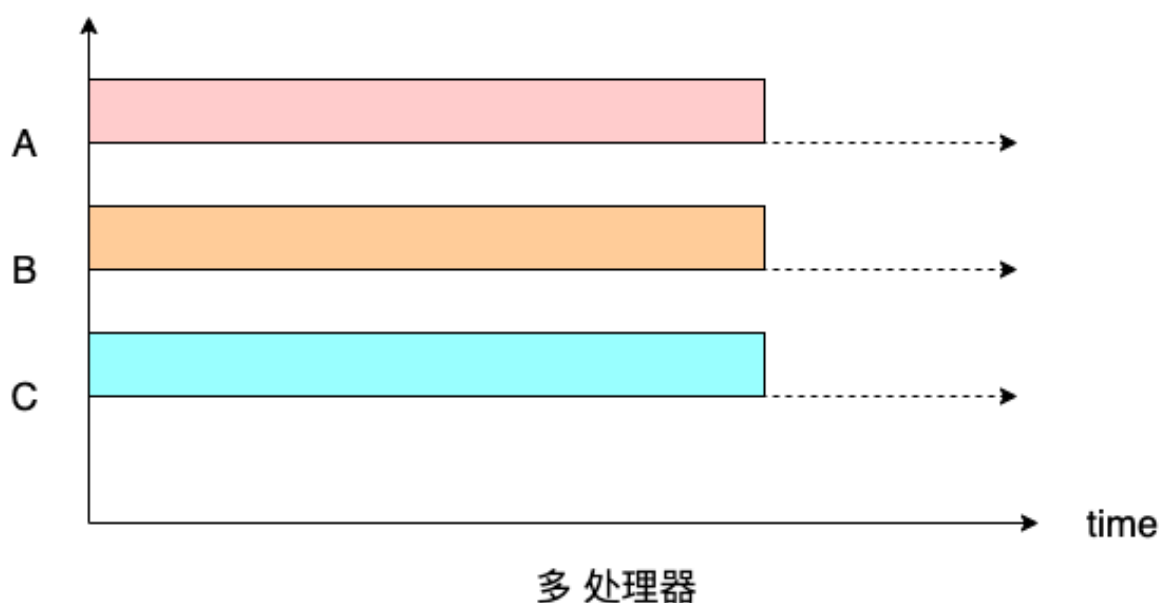
单核并发



在单核机器上，“多进程”并不是真正的多个进程在同时执行，而是通过CPU时间分片，操作系统快速在进程间切换而模拟出来的多进程。我们通常把这种情况成为并发。

单核的多个进程，不是“一并发生”的，是cpu高速切换让我们看起来像“一并发生”而已。

多核并行



我们使用的计算机基本上都搭载了多核CPU，这时，我们能真正的实现多个进程并行执行，这种情况叫做并行（一并进行）。

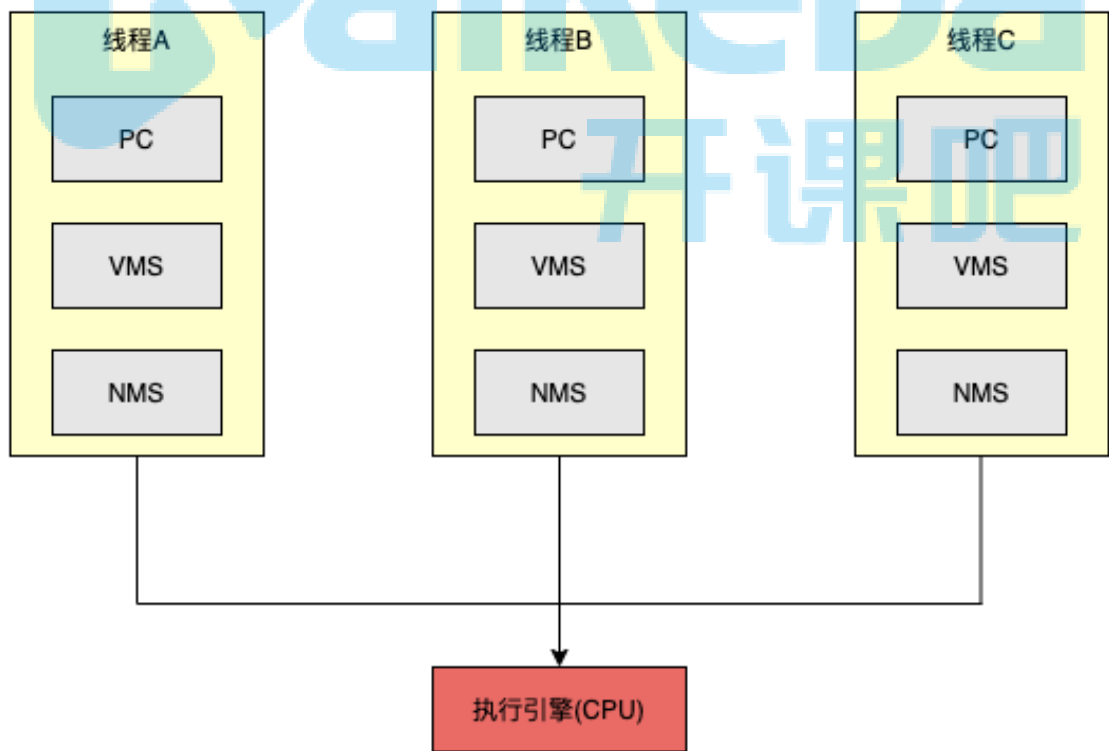
多核cpu让多进程的并发有可能变成了并行。

所以我们说得并发，有可能是是并行，也可能是并发，这跟cpu的核心数有关系。

在多核机器上，我们的多个线程可以并行执行在多个核上，进一步提升效率。

2、为什么使用PC寄存器记录当前线程的执行地址？

因为CPU需要不停地切换线程，这时候切换回来以后，线程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、PC寄存器为什么被设定为线程私有

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独立计算，从而不会出现相互干扰的情况。

由于CPU时间片轮换限制，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，如何保证分毫不差呢？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。

三、Java虚拟机栈（Java方法）

如何设置栈的大小

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。JDK5.0以后每个线程栈大小为1M，以前每个线程栈大小为256K。根据应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右

```
* 演示栈中的异常
*
* 默认情况下: count 10823
* 设置栈的大小: -Xss256k count 1874
*/

public class StackErrorTest {
    private static int count = 1;
    public static void main(String[] args) {
        System.out.println(count);
        count++;
        main(args);
    }
}
```

虚拟机栈存储哪些数据

栈帧是什么

栈帧(Stack Frame)是用于支持虚拟机进行方法执行的数据结构。

栈帧存储了方法的**局部变量表**、**操作数栈**、**动态连接和方法返回地址**等信息。每一个方法从调用至执行完成的过程，都对应着一个栈帧在虚拟机栈里从入栈到出栈的过程。

当前栈帧

一个线程中方法的调用链可能会很长，所以会有很多栈帧。只有位于JVM虚拟机栈栈顶的元素才是有效的，即称为**当前栈帧**，与这个栈帧相关连的方法称为**当前方法**，定义这个方法的类叫做**当前类**。

执行引擎运行的所有字节码指令都只针对当前栈帧进行操作。如果当前方法调用了其他方法，或者当前方法执行结束，那这个方法的栈帧就不再是当前栈帧了。

什么时候创建栈帧

调用新的方法时，新的栈帧也会随之创建。并且随着程序控制权转移到新方法，新的栈帧成为了当前栈帧。方法返回之际，原栈帧会返回方法的执行结果给之前的栈帧(返回给方法调用者)，随后虚拟机将会丢弃此栈帧。

局部变量表

静态方法的局部变量表：下标为0需要特殊留给某个引用

非静态方法的局部变量表：下标为0永远留给this引

存储内容

局部变量表(Local Variable Table)是一组变量值存储空间，用于存放方法参数和方法内定义的局部变量。

一个局部变量可以保存一个类型为boolean、byte、char、short、int、float、reference和returnAddress类型的数据。reference类型表示对一个对象实例的引用。。

局部变量表中的存储顺序：

- this引用（实例对象都需要维护的一个变量，而且在局部变量表中始终处于第一个位置，也就是下标为0的位置）
- 方法参数
- 方法内声明的变量

存储容量

局部变量表的容量以[变量槽\(Variable Slot\)](#)为最小单位，Java虚拟机规范并没有定义一个槽所应该占用内存空间的大小，但是规定了一个槽应该可以存放一个32位以内的数据类型。

在Java程序编译为Class文件时，就在方法的Code属性中的[max_locals](#)数据项中确定了该方法所需分配的局部变量表的最大容量。(最大Slot数量)

[double\long这种8字节类型的数据，都需要两个slot来存储。](#)

其他

虚拟机通过索引定位的方法查找相应的局部变量，索引的范围是从0~[局部变量表最大容量](#)。如果Slot是32位的，则遇到一个64位数据类型的变量(如long或double型)时，会连续使用两个连续的Slot来存储。

操作数栈

作用

[操作数栈\(Operand Stack\)](#)也常称为操作栈，它是一个[后入先出栈\(LIFO\)](#)。

JVM的解释引擎是基于栈（操作数栈）的方式去执行的。（另外还有一种是基于寄存器的方式）

使用做饭的案例去分析操作数栈的作用？

当一个方法刚刚开始执行时，其操作数栈是空的，随着方法执行和字节码指令的执行，会从局部变量表或对象实例的字段（成员变量）中复制常量或变量写入到操作数栈，再随着计算的进行将栈中元素出栈到局部变量表或者返回给方法调用者，也就是出栈/入栈操作。一个完整的方法执行期间往往包含多个这样出栈/入栈的过程。

举例：比如两数相加，需要将两个数字取到操作数栈里面，再进行计算。

存储内容

操作数栈的每一个元素可以是任意Java数据类型，32位的数据类型占一个栈容量，64位的数据类型占2个栈容量。

存储容量

同局部变量表一样，操作数栈的最大深度也在编译的时候写入到方法的Code属性的max_stacks数据项中。且在方法执行的任意时刻，操作数栈的深度都不会超过max_stacks中设置的最大值。

结论

一个线程的执行过程中，需要进行两个栈的入栈出栈操作，一个是JVM栈（栈帧的出栈和入栈），一个是操作数栈（参与计算的值进行出栈和入栈）

动态链接 (Dynamic Linking)

在一个class文件中，一个方法要调用其他方法，需要将这些方法的[符号引用](#)转化为其在内存地址中的[直接引用](#)，而[符号引用存在于方法区中的运行时常量池](#)。

- 符号引用：

23: invokevirtual #13

26: invokevirtual #14

29: sipush 128

32: invokestatic #5

35: astore 4

37: sipush 128

40: invokestatic #5

43: astore 5

- 直接引用：

就是对应方法的内存地址

Java虚拟机栈中，[每个栈帧都包含一个指向运行时常量池中该栈所属方法的符号引用](#)，持有这个引用的目的是为了支持方法调用过程中的[动态连接 \(Dynamic Linking\)](#)。

这些[符号引用](#)一部分会在[类加载阶段](#)或者[第一次使用](#)时就[直接转化为直接引用](#)，这类转化称为[静态解析](#)。另一部分将在每次运行期间转化为直接引用，这类转化称为[动态连接](#)。

方法返回

当一个方法开始执行时，可能有两种方式退出该方法：

- 正常完成出口
- 异常完成出口

正常完成出口是指方法正常完成并退出，没有抛出任何异常(包括Java虚拟机异常以及执行时通过throw语句显示抛出的异常)。如果当前方法正常完成，则根据当前方法返回的字节码指令，这时有可能会有返回值传递给方法调用者(调用它的方法)，或者无返回值。[具体是否有返回值以及返回值的数据类型将根据该方法返回的字节码指令确定。](#)

异常完成出口是指方法执行过程中遇到异常，并且这个异常在方法体内部没有得到处理，导致方法退出。

无论是Java虚拟机抛出的异常还是代码中使用athrow指令产生的异常，只要在本方法的异常表中没有搜索到相应的异常处理器，就会导致方法退出。

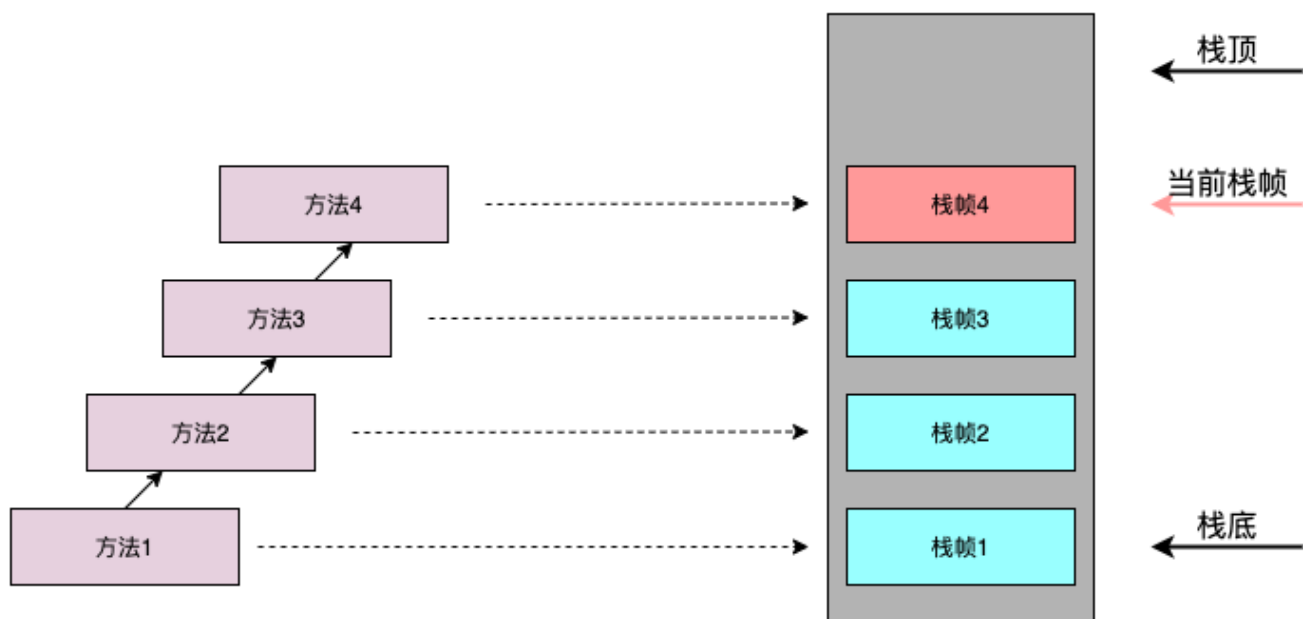
无论方法采用何种方式退出，在方法退出后都需要返回到方法被调用的位置，程序才能继续执行，方法返回时可能需要在当前栈帧中保存一些信息，用来帮他恢复它的上层方法执行状态。

A() ---> B()，当B方法执行完返回时，发生以下操作：

方法退出过程实际上就等同于把当前栈帧出栈，因此退出可以执行的操作有：恢复上层方法的局部变量表和操作数栈，把返回值(如果有的话)压入调用者的操作数栈中，调整PC计数器的值以指向方法调用指令后的下一条指令。

一般来说，方法正常退出时，调用者的PC计数值可以作为返回地址，栈帧中可能保存此计数值。而方法异常退出时，返回地址是通过异常处理器表确定的，栈帧中一般不会保存此部分信息。

栈运行原理



```
public class StackFrameTest {  
    public static void main(String[] args) {  
        StackFrameTest test = new StackFrameTest();  
        test.method1();  
        //输出 method1 () 和method2 () 都作为当前栈帧出现了两次,  
method3 () 一次  
        //      method1()开始执行。。。开课吧  
        //      method2()开始执行。。。开课吧  
        //      method3()开始执行。。。开课吧  
        //      method3()执行结束。。。开课吧  
        //      method2()执行结束。。。开课吧  
        //      method1()执行结束。。。开课吧  
    }  
  
    public void method1(){  
        System.out.println("method1()开始执行。。。");  
        method2();  
        System.out.println("method1()执行结束。。。");  
        //return 可以省略  
    }  
}
```

```
public int method2(){
    System.out.println("method2()开始执行。。。");
    int i = 10;
    int m = (int) method3();
    System.out.println("method2()执行结束。。。");
    return i+m;
}

public double method3(){
    System.out.println("method3()开始执行。。。");
    double j = 20.0;
    System.out.println("method3()执行结束。。。");
    return j;
}
}
```

栈异常

虚拟机栈的深度，在编译的时候，就已经确定了

Java虚拟机规范中，对该区域规定了这两种异常情况：

1. 如果线程请求的栈深度大于虚拟机所允许的深度，将会抛出 **StackOverflowError** 异常（-Xss）；
2. 虚拟机栈可以动态拓展，当扩展时无法申请到足够的内存，就会抛出 **OutOfMemoryError** 异常。

```
package com.kkb.test.memory;

public class StackErrorMock {
    private static int index = 1;
```

```
public void call(){
    index++;
    call();
}

public static void main(String[] args) {
    StackErrorMock mock = new StackErrorMock();
    try {
        mock.call();
    } catch (Throwable e){
        System.out.println("Stack deep : "+index);
        e.printStackTrace();
    }
}
```

相关面试题

1.举例栈溢出的情况? (StackOverflowError)

- 递归调用等, 通过-Xss设置栈的大小;

2.调整栈的大小, 就能保证不出现溢出么?

- 不能 如递归无限次数肯定会溢出, 调整栈大小只能保证溢出的时间晚一些, 极限情况会导致OOM内存溢出 (Out Of Memory Error) 注意是Error

3.分配的栈内存越大越好么?

- 不是 会挤占其他线程的空间

4.垃圾回收是否会涉及到虚拟机栈?

- 不会

5.方法中定义的局部变量是否线程安全?

具体情况具体分析

```
/**
 * 面试题：
 * 方法中定义的局部变量是否线程安全？ 具体情况具体分析
 *
 * 何为线程安全？
 *     如果只有一个线程可以操作此数据，则必定是线程安全的。
 *     如果有多个线程操作此数据，则此数据是共享数据。如果不考虑同步机
制的话，会存在线程安全问题
 *
 * 我们知道StringBuffer是线程安全的源码中实现synchronized,
StringBuilder源码未实现synchronized,在多线程情况下是不安全的
 * 二者均继承自AbstractStringBuilder
 *
 */
public class StringBuilderTest {

    //s1的声明方式是线程安全的，s1在方法method1内部消亡了
    public static void method1(){
        StringBuilder s1 = new StringBuilder();
        s1.append("a");
        s1.append("b");
    }

    //stringBuilder的操作过程：是不安全的，因为method2可以被多个
线程调用
```



```
public static void method2(StringBuilder  
stringBuilder){  
    stringBuilder.append("a");  
    stringBuilder.append("b");  
}
```

//s1的操作：是线程不安全的 有返回值，可能被其他线程共享

```
public static StringBuilder method3(){  
    StringBuilder s1 = new StringBuilder();  
    s1.append("a");  
    s1.append("b");  
    return s1;  
}
```

//s1的操作：是线程安全的，StringBuilder的toString方法是创建了一个新的String，s1在内部消亡了

```
public static String method4(){  
    StringBuilder s1 = new StringBuilder();  
    s1.append("a");  
    s1.append("b");  
    return s1.toString();  
}
```

```
public static void main(String[] args) {  
    StringBuilder s = new StringBuilder();  
    new Thread(()->{  
        s.append("a");  
        s.append("b");  
    }).start();  
  
    method2(s);  
}
```

```
}
```

四、本地方法栈（本地方法）

什么是本地方法

本地方法栈和虚拟机栈相似，区别就是虚拟机栈为虚拟机执行Java服务（字节码服务），而本地方法栈为虚拟机使用到的Native方法（比如C++方法）服务。

简单地讲，一个Native Method就是一个java调用非java代码的接口。一个Native Method是这样一个java的方法：该方法的实现由非java语言实现，比如C。

在定义一个native method时，并不提供实现体（有些像定义一个java interface），因为其实体是由非java语言在外面实现的。下面给了一个示例：

```
public class IHaveNatives
{
    native public void Native1( int x ) ;
    native static public long Native2() ;
    native synchronized private float Native3( Object o
) ;
    native void Native4( int[] ary ) throws Exception ;
}
```

本地接口的作用是融合不同的编程语言为java所用，它的初衷是融合C/C++程序。

为什么要使用本地方法

java使用起来非常方便，然而有些层次的任务用java实现起来不容易，或者我们对程序的效率很在意时，问题就来了。

本地方法非常有用，因为它有效地扩充了jvm。

事实上，我们所写的java代码已经用到了本地方法，在sun的java的并发（多线程）的机制实现中，许多与操作系统的接触点都用到了本地方法，这使得java程序能够超越java运行时的界限。有了本地方法，java程序可以做任何应用层次的任务。

有时java应用需要与java外面的环境交互。这是本地方法存在的主要原因，你可以想想java需要与一些底层系统如操作系统或某些硬件交换信息时的情况。

本地方法正是这样一种交流机制：它为我们提供了一个非常简洁的接口，而且我们无需去了解java应用之外的繁琐的细节。

java使用起来非常方便，然而有些层次的任务用java实现起来不容易，或者我们对程序的效率很在意时，问题就来了。

与java环境外交互：

有时java应用需要与java外面的环境交互。这是本地方法存在的主要原因，你可以想想java需要与一些底层系统如操作系统或某些硬件交换信息时的情况。本地方法正是这样一种交流机制：它为我们提供了一个非常简洁的接口，而且我们无需去了解java应用之外的繁琐的细节。

与操作系统交互：

JVM支持着java语言本身和运行时库，它是java程序赖以生存的平台，它由一个解释器（解释字节码）和一些连接到本地代码的库组成。然而不管怎样，它毕竟不是一个完整的系统，它经常依赖于一些底层（underneath 在下面的）系统的支持。这些底层系统常常是强大的操作系统。通过使用本地方法，我们得以用java实现了jre的与底层系统的交互，甚至JVM的一部分就是用C写的。还有，如果我们要使用一些java语言本身没有提供封装的操作系统特性时，我们也需要使用本地方法。

Sun's Java

Sun的解释器是用C实现的，这使得它能像一些普通的C一样与外部交互。jre大部分是用java实现的，它也通过一些本地方法与外界交互。例如：类java.lang.Thread 的 setPriority()方法是用java实现的，但是它实现调用的是该类里的本地方法setPriority0()。这个本地方法是用C实现的，并被植入JVM内部，在Windows 95的平台上，这个本地方法最终将调用Win32 SetPriority() API。这是一个本地方法的具体实现由JVM直接提供，更多的情况是本地方法由外部的动态链接库（external dynamic link library）提供，然后被JVM调用。

```
package java.lang;

public class Object {

    private static native void registerNatives();
    static {
        registerNatives();
    }

    public final native Class<?> getClass();
    .....
}
```

```
package java.lang;

public
class Thread implements Runnable {
    .....
    /* Some private helper methods */
    private native void setPriority0(int newPriority);
    private native void stop0(Object o);
    private native void suspend0();
    private native void resume0();
    private native void interrupt0();
    private native void setNativeName(String name);
}
```

现状

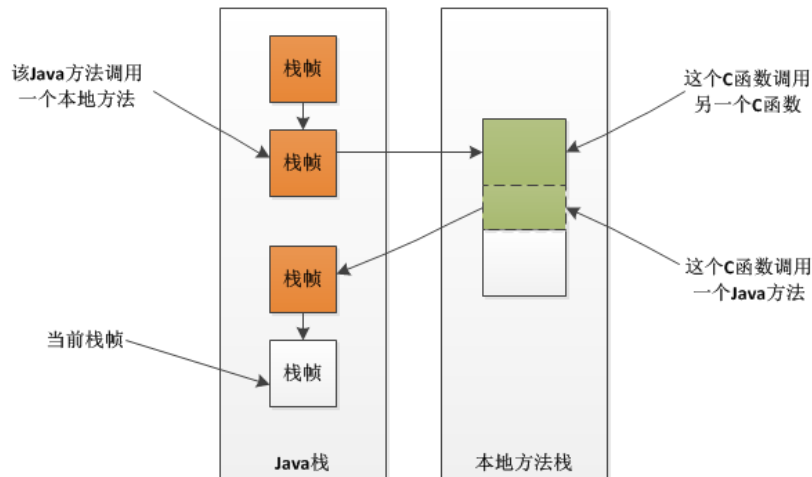
目前该方法使用的越来越少了，除非是与硬件有关的应用，比如通过java程序驱动打印机或者java系统管理生产设备，在企业应用中已经比较少见。因为现状的异构领域间的通信很发达，比如可以使用socket通信，也可以使用web service 等等，这里不做过多介绍。

本地方法栈的使用流程

下图描绘了这样一个情景，就是当一个线程调用一个本地方法时，本地方法又回调虚拟机中的另一个Java方法。

[这幅图展示了JAVA虚拟机内部线程运行的全景图](#)。一个线程可能在整个生命周期中都执行Java方法，操作它的Java栈；或者它可能毫无障碍地在Java栈和本地方法栈之间跳转。

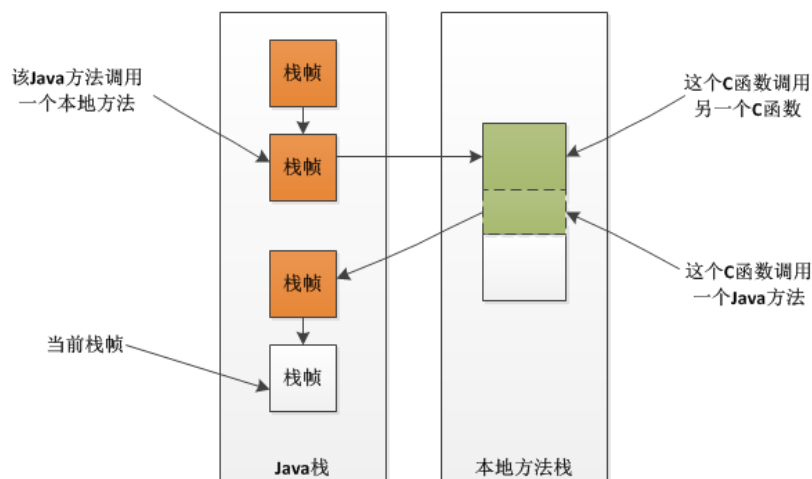
一个线程调用JAVA方法和本地方法时的栈



该线程首先调用了两个Java方法，而第二个Java方法又调用了一个本地方法，这样导致虚拟机使用了一个本地方法栈。假设这是一个C语言栈，其间有两个C函数，第一个C函数被第二个Java方法当做本地方法调用，而这个C函数又调用了第二个C函数。之后第二个C函数又通过本地方法接口回调了一个Java方法（第三个Java方法），最终这个Java方法又调用了一个Java方法（它成为图中的当前方法）。

本地方法栈的理解

一个线程调用JAVA方法和本地方法时的栈



本地方法栈(Native Method Stack)

java虚拟机用于管理java方法的调用，而本地方法栈用于管理本地方法的调用。

本地方法栈也是线程私用的。

本地方法是使用C语言实现的。

当某个线程调用一个本地方法时，它就进入了一个全新的并且不受虚拟机限制的世界。它和虚拟机拥有同样的权限。

- 本地方法可以通过本地接口来访问虚拟机内部的运行时数据区。
- 它甚至可以直接使用本地处理器中的寄存器
- 直接从本地内存的堆中分配任意数量的内存

允许本地方法栈的大小是固定的或者可动态扩展的内存大小。

- 如果线程请求分配的栈容量超过本地方法栈允许的最大容量,Java虚拟机将会抛出一个StackOverflowError异常
- 如果本地方法栈可以动态扩展,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本地方法栈,那Java虚拟机将会抛出一个OutOfMemoryError异常。

并不是所有的JVM都支持本地方法。因为java虚拟机规范并没有明确要求本地方法栈的使用语言、具体实现方式、数据结构等。如果JVM产品不打算支持native方法，也可以无需实现本地方法栈。

在Hotspot JVM中，直接将本地方法栈和虚拟机栈合二为一。

五、方法执行

字节码指令集(字典)

概述

Java虚拟机的指令由一个字节长度的

- 代表着某种特定操作含义的数字（称为操作码，Opcode）
- 跟随其后的零至多个代表此操作所需参数（称为操作数，Operands）而构成。

Opcode+操作数

- `iconst_0` 操作码
- `bipush 10` 操作码+操作数

比如：

字节码	助记符	指令含义
0x00	<code>nop</code>	什么都不做
0x01	<code>aconst_null</code>	将 <code>null</code> 推送至栈顶
0x02	<code>iconst_m1</code>	将 <code>int</code> 型 <code>-1</code> 推送至栈顶
0x03	<code>iconst_0</code>	将 <code>int</code> 型 <code>0</code> 推送至栈顶
0x04	<code>iconst_1</code>	将 <code>int</code> 型 <code>1</code> 推送至栈顶
0x05	<code>iconst_2</code>	将 <code>int</code> 型 <code>2</code> 推送至栈顶
0x06	<code>iconst_3</code>	将 <code>int</code> 型 <code>3</code> 推送至栈顶
0x07	<code>iconst_4</code>	将 <code>int</code> 型 <code>4</code> 推送至栈顶

0x08	iconst_5	将 int 型 5 推送至栈顶
0x09	lconst_0	将 long 型 0 推送至栈顶
0x0a	lconst_1	将 long 型 1 推送至栈顶
0x0b	fconst_0	将 float 型 0 推送至栈顶
0x0c	fconst_1	将 float 型 1 推送至栈顶
0x0d	fconst_2	将 float 型 2 推送至栈顶
0x0e	dconst_0	将 double 型 0 推送至栈顶
0x0f	dconst_1	将 double 型 1 推送至栈顶
0x10	bipush	将单字节的常量值 (Byte.MIN_VALUE ~ Byte.MAX_VALUE, 即 -128 ~ 127) 推送至栈顶
0x11	sipush	将短整型的常量值 (Short.MIN_VALUE ~ Short.MAX_VALUE, 即 -32768 ~ 32767) 推送至栈顶
0x12	ldc	将 int、float 或 String 型常量值从常量池中推送至栈顶
0x13	ldc_w	将 int、float 或 String 型常量值从常量池中推送至栈顶 (宽索引)
0x14	ldc2_w	将 long 或 double 型常量值从常量池中推送至栈顶 (宽索引)
0x15	iload	将指定的 int 型局部变量推送至栈顶
0x16	lload	将指定的 long 型局部变量推送至栈顶
0x17	fload	将指定的 float 型局部变量推送至栈顶

0x18	dload	将指定的 double 型局部变量推送至栈顶
0x19	aload	将指定的 引用 型局部变量推送至栈顶
0x1a	iload_0	将第一个 int 型局部变量推送至栈顶
0x1b	iload_1	将第二个 int 型局部变量推送至栈顶
0x1c	iload_2	将第三个 int 型局部变量推送至栈顶
0x1d	iload_3	将第四个 int 型局部变量推送至栈顶
0x1e	lload_0	将第一个 long 型局部变量推送至栈顶
0x1f	lload_1	将第二个 long 型局部变量推送至栈顶
0x20	lload_2	将第三个 long 型局部变量推送至栈顶
0x21	lload_3	将第四个 long 型局部变量推送至栈顶
0x22	fload_0	将第一个 float 型局部变量推送至栈顶
0x23	fload_1	将第二个 float 型局部变量推送至栈顶
0x24	fload_2	将第三个 float 型局部变量推送至栈顶
0x25	fload_3	将第四个 float 型局部变量推送至栈顶
0x26	dload_0	将第一个 double 型局部变量推送至栈顶
0x27	dload_1	将第二个 double 型局部变量推送至栈顶
0x28	dload_2	将第三个 double 型局部变量推送至栈顶
0x29	dload_3	将第四个 double 型局部变量推送至栈顶
0x2a	aload_0	将第一个 引用 型局部变量推送至栈顶
0x2b	aload_1	将第二个 引用 型局部变量推送至栈顶
0x2c	aload_2	将第三个 引用 型局部变量推送至栈顶

0x2d	aload_3	将第四个 引用 型局部变量推送至栈顶
0x2e	iaload	将 int 型数组指定索引的值推送至栈顶
0x2f	laload	将 long 型数组指定索引的值推送至栈顶
0x30	faload	将 float 型数组指定索引的值推送至栈顶
0x31	daload	将 double 型数组指定索引的值推送至栈顶
0x32	aaload	将 引用 型数组指定索引的值推送至栈顶
0x33	baload	将 boolean 或 byte 型数组指定索引的值推送至栈顶
0x34	caload	将 char 型数组指定索引的值推送至栈顶
0x35	saload	将 short 型数组指定索引的值推送至栈顶
0x36	istore	将栈顶 int 型数值存入指定局部变量
0x37	lstore	将栈顶 long 型数值存入指定局部变量
0x38	fstore	将栈顶 float 型数值存入指定局部变量
0x39	dstore	将栈顶 double 型数值存入指定局部变量
0x3a	astore	将栈顶 引用 型数值存入指定局部变量
0x3b	istore_0	将栈顶 int 型数值存入第一个局部变量
0x3c	istore_1	将栈顶 int 型数值存入第二个局部变量
0x3d	istore_2	将栈顶 int 型数值存入第三个局部变量
0x3e	istore_3	将栈顶 int 型数值存入第四个局部变量
0x3f	lstore_0	将栈顶 long 型数值存入第一个局部变量
0x40	lstore_1	将栈顶 long 型数值存入第二个局部变量

0x41	lstore_2	将栈顶 long 型数值存入第三个局部变量
0x42	lstore_3	将栈顶 long 型数值存入第四个局部变量
0x43	fstore_0	将栈顶 float 型数值存入第一个局部变量
0x44	fstore_1	将栈顶 float 型数值存入第二个局部变量
0x45	fstore_2	将栈顶 float 型数值存入第三个局部变量
0x46	fstore_3	将栈顶 float 型数值存入第四个局部变量
0x47	dstore_0	将栈顶 double 型数值存入第一个局部变量
0x48	dstore_1	将栈顶 double 型数值存入第二个局部变量
0x49	dstore_2	将栈顶 double 型数值存入第三个局部变量
0x4a	dstore_3	将栈顶 double 型数值存入第四个局部变量
0x4b	astore_0	将栈顶 引用 型数值存入第一个局部变量
0x4c	astore_1	将栈顶 引用 型数值存入第二个局部变量
0x4d	astore_2	将栈顶 引用 型数值存入第三个局部变量
0x4e	astore_3	将栈顶 引用 型数值存入第四个局部变量
0x4f	iastore	将栈顶 int 型数值存入指定数组的指定索引位置
0x50	lastore	将栈顶 long 型数值存入指定数组的指定索引位置
0x51	fastore	将栈顶 float 型数值存入指定数组的指定索引位置
0x52	dastore	将栈顶 double 型数值存入指定数组的指定索引位置

0x53	aastore	将栈顶 引用 型数值存入指定数组的指定索引位置
0x54	bastore	将栈顶 boolean 或 byte 型数值存入指定数组的指定索引位置
0x55	castore	将栈顶 char 型数值存入指定数组的指定索引位置
0x56	sastore	将栈顶 short 型数值存入指定数组的指定索引位置
0x57	pop	将栈顶数值弹出（数值不能是 long 或 double 类型的）
0x58	pop2	将栈顶的一个（对于 long 或 double 类型）或两个数值（对于非 long 或 double 的其他类型）弹出
0x59	dup	复制栈顶数值并将复制值压入栈顶
0x5a	dup_x1	复制栈顶数值并将两个复制值压入栈顶
0x5b	dup_x2	复制栈顶数值并将三个（或两个）复制值压入栈顶
0x5c	dup2	复制栈顶一个（对于 long 或 double 类型）或两个数值（对于非 long 或 double 的其他类型）并将复制值压入栈顶
0x5d	dup2_x1	dup_x1 指令的双倍版本
0x5e	dup2_x2	dup_x2 指令的双倍版本
0x5f	swap	将栈最顶端的两个数值互换（数值不能是 long 或 double 类型）

0x60	iadd	将栈顶两 int 型数值相加并将结果压入栈顶
0x61	ladd	将栈顶两 long 型数值相加并将结果压入栈顶
0x62	fadd	将栈顶两 float 型数值相加并将结果压入栈顶
0x63	dadd	将栈顶两 double 型数值相加并将结果压入栈顶
0x64	isub	将栈顶两 int 型数值相减并将结果压入栈顶
0x65	lsub	将栈顶两 long 型数值相减并将结果压入栈顶
0x66	fsub	将栈顶两 float 型数值相减并将结果压入栈顶
0x67	dsub	将栈顶两 double 型数值相减并将结果压入栈顶
0x68	imul	将栈顶两 int 型数值相乘并将结果压入栈顶
0x69	lmul	将栈顶两 long 型数值相乘并将结果压入栈顶
0x6a	fmul	将栈顶两 float 型数值相乘并将结果压入栈顶
0x6b	dmul	将栈顶两 double 型数值相乘并将结果压入栈顶
0x6c	idiv	将栈顶两 int 型数值相除并将结果压入栈顶
0x6d	ldiv	将栈顶两 long 型数值相除并将结果压入栈顶
0x6e	fdiv	将栈顶两 float 型数值相除并将结果压入栈顶
0x6f	ddiv	将栈顶两 double 型数值相除并将结果压入栈顶
0x70	irem	将栈顶两 int 型数值作取模运算并将结果压入栈顶
0x71	lrem	将栈顶两 long 型数值作取模运算并将结果压入栈顶
0x72	frem	将栈顶两 float 型数值作取模运算并将结果压入栈顶

0x73	drem	将栈顶两 double 型数值作取模运算并将结果压入栈顶
0x74	ineg	将栈顶两 int 型数值取负并将结果压入栈顶
0x75	lneg	将栈顶两 long 型数值取负并将结果压入栈顶
0x76	fneg	将栈顶两 float 型数值取负并将结果压入栈顶
0x77	dneg	将栈顶两 double 型数值取负并将结果压入栈顶
0x78	ishl	将 int 型数值左移指定位数并将结果压入栈顶
0x79	lshl	将 long 型数值左移指定位数并将结果压入栈顶
0x7a	ishr	将 int 型数值右（带符号）移指定位数并将结果压入栈顶
0x7b	lshr	将 long 型数值右（带符号）移指定位数并将结果压入栈顶
0x7c	iushr	将 int 型数值右（无符号）移指定位数并将结果压入栈顶
0x7d	lushr	将 long 型数值右（无符号）移指定位数并将结果压入栈顶
0x7e	iand	将栈顶两 int 型数值作“按位与”并将结果压入栈顶
0x7f	land	将栈顶两 long 型数值作“按位与”并将结果压入栈顶
0x80	ior	将栈顶两 int 型数值作“按位或”并将结果压入栈顶

0x81	lor	将栈顶两 long 型数值作“按位或”并将结果压入栈顶
0x82	ixor	将栈顶两 int 型数值作“按位异或”并将结果压入栈顶
0x83	lxor	将栈顶两 long 型数值作“按位异或”并将结果压入栈顶
0x84	iinc M N	(M 为非负整数, N 为整数) 将局部变量数组的第 M 个单元中的 int 值增加 N, 常用于 for 循环中自增量的更新
0x85	i2l	将栈顶 int 型数值强制转换成 long 型数值, 并将结果压入栈顶
0x86	i2f	将栈顶 int 型数值强制转换成 float 型数值, 并将结果压入栈顶
0x87	i2d	将栈顶 int 型数值强制转换成 double 型数值, 并将结果压入栈顶
0x88	l2i	将栈顶 long 型数值强制转换成 int 型数值, 并将结果压入栈顶
0x89	l2f	将栈顶 long 型数值强制转换成 float 型数值, 并将结果压入栈顶
0x8a	l2d	将栈顶 long 型数值强制转换成 double 型数值, 并将结果压入栈顶
0x8b	f2i	将栈顶 float 型数值强制转换成 int 型数值, 并将结果压入栈顶
0x8c	f2l	将栈顶 float 型数值强制转换成 long 型数值, 并将结果压入栈顶

0x8d	f2d	将栈顶 float 型数值强制转换成 double 型数值，并将结果压入栈顶
0x8e	d2i	将栈顶 double 型数值强制转换成 int 型数值，并将结果压入栈顶
0x8f	d2l	将栈顶 double 型数值强制转换成 long 型数值，并将结果压入栈顶
0x90	d2f	将栈顶 double 型数值强制转换成 float 型数值，并将结果压入栈顶
0x91	i2b	将栈顶 int 型数值强制转换成 byte 型数值，并将结果压入栈顶
0x92	i2c	将栈顶 int 型数值强制转换成 char 型数值，并将结果压入栈顶
0x93	i2s	将栈顶 int 型数值强制转换成 short 型数值，并将结果压入栈顶
0x94	lcmp	比较栈顶两 long 型数值的大小，并将结果（1、0 或 -1）压入栈顶
0x95	fcmpl	比较栈顶两 float 型数值的大小，并将结果（1、0 或 -1）压入栈顶；当其中一个数值为“NaN”时，将 -1 压入栈顶
0x96	fcmpg	比较栈顶两 float 型数值的大小，并将结果（1、0 或 -1）压入栈顶；当其中一个数值为“NaN”时，将 1 压入栈顶
0x97	dcmpl	比较栈顶两 double 型数值的大小，并将结果（1、0 或 -1）压入栈顶；当其中一个数值为“NaN”时，将 -1 压入栈顶

0x98	dcmpg	比较栈顶两 double 型数值的大小，并将结果（1、0 或 -1）压入栈顶；当其中一个数值为“NaN”时，将 1 压入栈顶
0x99	ifeq	当栈顶 int 型数值等于 0 时跳转
0x9a	ifne	当栈顶 int 型数值不等于 0 时跳转
0x9b	iflt	当栈顶 int 型数值小于 0 时跳转
0x9c	ifge	当栈顶 int 型数值大于或等于 0 时跳转
0x9d	ifgt	当栈顶 int 型数值大于 0 时跳转
0x9e	ifle	当栈顶 int 型数值小于或等于 0 时跳转
0x9f	if_icmpeq	比较栈顶两 int 型数值的大小，当结果等于 0 时跳转
0xa0	if_icmpne	比较栈顶两 int 型数值的大小，当结果不等于 0 时跳转
0xa1	if_icmplt	比较栈顶两 int 型数值的大小，当结果小于 0 时跳转
0xa2	if_icmpge	比较栈顶两 int 型数值的大小，当结果大于或等于 0 时跳转
0xa3	if_icmpgt	比较栈顶两 int 型数值的大小，当结果大于 0 时跳转
0xa4	if_icmple	比较栈顶两 int 型数值的大小，当结果小于或等于 0 时跳转
0xa5	if_acmpeq	比较栈顶两 引用 型数值，当结果相等时跳转
		比较栈顶两 引用 型数值，当结果不相等时跳

0xa6	if_acmpne	转
0xa7	goto	无条件跳转
0xa8	jsr	跳转至指定的 16 位 offset 位置，并将 jsr 的下一条指令地址压入栈顶
0xa9	ret	返回至局部变量指定的 index 的指令位置（一般与 jsr 或 jsr_w 联合使用）
0xaa	tableswitch	用于 switch 条件跳转，case 值连续（可变长度指令）
0xab	lookupswitch	用于 switch 条件跳转，case 值不连续（可变长度指令）
0xac	ireturn	从当前方法返回 int
0xad	lreturn	从当前方法返回 long
0xae	freturn	从当前方法返回 float
0xaf	dreturn	从当前方法返回 double
0xb0	areturn	从当前方法返回对象引用
0xb1	return	从当前方法返回 void
0xb2	getstatic	获取指定类的静态字段，并将其压入栈顶
0xb3	putstatic	为指定类的静态字段赋值
0xb4	getfield	获取指定类的实例字段，并将其压入栈顶
0xb5	putfield	为指定类的实例字段赋值
0xb6	invokevirtual	调用实例方法

0xb7	invokespecial	调用超类构造方法，实例初始化方法，私有方法
0xb8	invokestatic	调用静态方法
0xb9	invokeinterface	调用接口方法
0xba	--	无此指令
0xbb	new	创建一个对象，并将其引用值压入栈顶
0xbc	newarray	创建一个指定的原始类型（如 int、float、char 等）的数组，并将其引用值压入栈顶
0xbd	anewarray	创建一个引用型（如类、接口、数组 等）的数组，并将其引用值压入栈顶
0xbe	arraylength	获得数组的长度值并将其压入栈顶
0xbf	athrow	将栈顶的异常抛出
0xc0	checkcast	校验类型转换，校验未通过将抛出 ClassCastException
0xc1	instanceof	校验对象是否是指定的类的实例，如果是则将 1 压入栈顶，否则将 0 压入栈顶
0xc2	monitorenter	获得对象的锁，用于同步方法或同步块
0xc3	monitorexit	释放对象的锁，用于同步方法或同步块
0xc4	wide	扩展局部变量的宽度
0xc5	multianewarray	创建指定类型和指定维度的多维数组（执行该指定时，操作数栈中必须包含各维度的长度），并将其引用值压入栈顶

0xc6	ifnull	为 null 时跳转
0xc7	ifnonnull	不为 null 时跳转
0xc8	goto_w	无条件跳转（宽索引）
0xc9	jsr_w	跳转至指定的 32 位 offset 位置，并将 jsr_w 的下一条指令地址压入栈顶

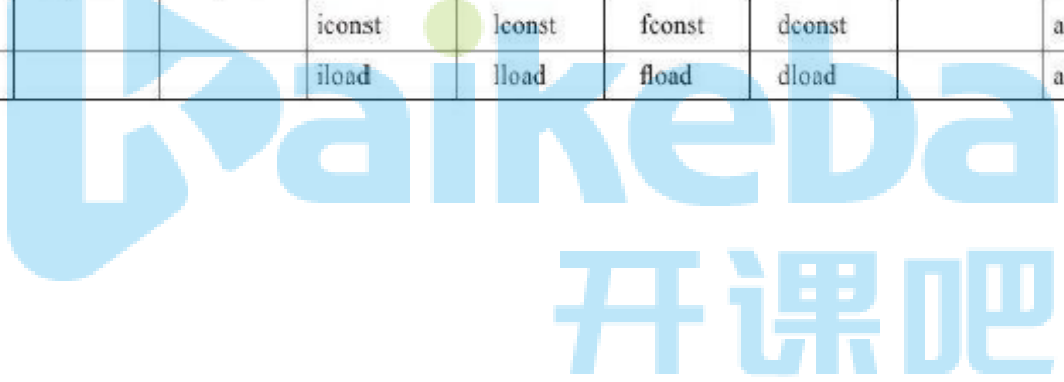


基本数据类型

- 1、除了long和double类型外，每个变量都占局部变量区中的一个变量槽(slot)，而long及double会占用两个连续的变量槽。
- 2、大多数对于boolean、byte、short和char类型数据的操作，都使用相应的int类型作为运算类型。

表 6-31 Java 虚拟机指令集所支持的数据类型

opcode	byte	short	int	long	float	double	char	reference
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload



(续)

opcode	byte	short	int	long	float	double	char	reference
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aaload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl				
Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				
i2T	i2b	i2s		i2l	i2f	i2d		
l2T			l2i		l2f	l2d		
f2T			f2i	f2l		f2d		
d2T			d2i	d2l	d2f			
Tcmp				lcmp				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		
if_TempOP			if_icmpOP					if_acmpOP
Treturn			ireturn	lreturn	freturn	dreturn		areturn

一、加载和存储指令（总）

1、将一个【局部变量表】加载到【操作数栈】：

```
iload、iload_<n>、lload、lload_<n>、fload、fload_<n>、
dload、dload_<n>、aload、aload_<n>
```

2、将一个数值从【操作数栈】存储到【局部变量表】：

```
istore、istore_<n>、lstore、lstore_<n>、fstore、fstore_<n>、dstore、dstore_<n>、astore、astore_<n>
```

3、将一个【常量】加载到【操作数栈】：

```
bipush、sipush、  
ldc、ldc_w、ldc2_w、  
aconst_null、iconst_m1、iconst_<i>、lconst_<l>、fconst_<f>、dconst_<d>
```

4、扩充局部变量表的访问索引的指令：

```
wide_<n>:_0、_1、_2、_3,
```

存储数据的操作数栈和局部变量表主要就是由加载和存储指令进行操作，除此之外，还有少量指令，如访问对象的字段或数组元素的指令也会向操作数栈传输数据。

二、const系列（小数值）

该系列命令主要负责把【简单的数值类型】送到【操作数栈栈顶】。该系列命令不带参数。注意只把简单的数值类型送到栈顶时，才使用如下的命令。

比如对应int型，该方式只能把-1, 0, 1, 2, 3, 4, 5（分别采用iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5）送到栈顶。

对于int型，其他的数值请使用push系列命令（比如bipush）。

指令码	助记符	说明
0x02 栈顶	iconst_m1	将int型(-1)推送至栈
0x03 栈顶	iconst_0	将int型(0)推送至
0x04 栈顶	iconst_1	将int型(1)推送至
0x05 栈顶	iconst_2	将int型(2)推送至
0x06 栈顶	iconst_3	将int型(3)推送至
0x07 栈顶	iconst_4	将int型(4)推送至
0x08 栈顶	iconst_5	将int型(5)推送至
0x09 栈顶	lconst_0	将long型(0)推送至
0x0a 栈顶	lconst_1	将long型(1)推送至
0x0b 至栈顶	fconst_0	将float型(0)推送

0x0c 至栈顶	fconst_1	将float型(1)推送
0x0d 至栈顶	fconst_2	将float型(2)推送
0x0e 至栈顶	dconst_0	将double型(0)推送
0x0f 至栈顶	dconst_1	将double型(1)推送

三、push系列（中数值）

该系列命令负责把一个【整形数字（长度比较小）】送到到【操作数栈栈顶】。该系列命令【有一个参数】，用于指定要送到栈顶的数字。

注意该系列命令只能操作一定范围内的整形数值，超出该范围的使用将使用【ldc命令】系列。

指令码	助记符	说明
0x10 顶	bipush	将单字节的常量值(-128~127)推送至栈
0x11 送至栈顶	sipush	将一个短整型常量值(-32768~32767)推送至栈顶

四、ldc系列（大数值或字符串常量）

该系列命令负责把【长度较长的数值常量】或【String常量值】从【常量池中】推送至【操作数栈栈顶】。该命令后面需要给一个表示常量在常量池中位置(编号)的参数，

哪些常量是放在常量池呢？比如：

```
final static int id=32768;  
final static float double=6.5
```

对于const系列命令和push系列命令操作范围之外的数值类型常量，都放在常量池中。

另外，所有不是通过new创建的String都是放在常量池中的。

指令码	助记符	说明
0x12	ldc	将int, float或String型常量值从常量池中推送至栈顶
0x13	ldc_w	将int, float或String型常量值从常量池中推送至栈顶（宽索引）
0x14	ldc2_w	将long或double型常量值从常量池中推送至栈顶（宽索引）

五、load系列

5.1、load系列A

该系列命令负责把【本地变量表中的值】送到【操作数栈栈顶】。这里的本地变量不仅可以是数值类型，还可以是引用类型。

- 对于前四个本地变量可以采用`iload_0`, `iload_1`, `iload_2`, `iload_3`(它们分别表示第0, 1, 2, 3个整形变量)这种不带参数的简化命令形式。
- 对于第4以上的本地变量将使用`iload`命令这种形式，在它后面给一参数，以表示是对第几个(从0开始)本类型的本地变量进行操作。对本地变量所进行的编号，是对所有类型的本地变量进行的（并不按照类型分类）。
- 对于非静态函数（虚方法，实例方法），第一变量是this，即其对应的操作是`aload_0`。
- 还有函数传入参数也算本地变量，在进行编号时，它是先于函数体的本地变量的。

指令码	助记符	说明
0x15 本地变量推送至栈顶	iload	将指定的int型本地变量推送至栈顶
0x16 本地变量推送至栈顶	lload	将指定的long型本地变量推送至栈顶
0x17 本地变量推送至栈顶	fload	将指定的float型本地变量推送至栈顶
0x18 本地变量推送至栈顶	dload	将指定的double型本地变量推送至栈顶

0x19	aload	将指定的引用类型 本地变量推送至栈顶
0x1a	iload_0	将第一个int型本地 变量推送至栈顶
0x1b	iload_1	将第二个int型本地 变量推送至栈顶
0x1c	iload_2	将第三个int型本地 变量推送至栈顶
0x1d	iload_3	将第四个int型本地 变量推送至栈顶
0x1e	lload_0	将第一个long型本地 变量推送至栈顶
0x1f	lload_1	将第二个long型本 地变量推送至栈顶
0x20	lload_2	将第三个long型本地 变量推送至栈顶
0x21	lload_3	将第四个long型本地 变量推送至栈顶
0x22	fload_0	将第一个float型本地 变量推送至栈顶
0x23	fload_1	将第二个float型本地 变量推送至栈顶

0x24 变量推送至栈顶	fload_2	将第三个float型本地
0x25 变量推送至栈顶	fload_3	将第四个float型本地
0x26 变量推送至栈顶	dload_0	将第一个double型本地
0x27 变量推送至栈顶	dload_1	将第二个double型本地
0x28 变量推送至栈顶	dload_2	将第三个double型本地
0x29 变量推送至栈顶	dload_3	将第四个double型本地
0x2a 变量推送至栈顶	aload_0	将第一个引用类型本地
0x2b 变量推送至栈顶	aload_1	将第二个引用类型本地
0x2c 变量推送至栈顶	aload_2	将第三个引用类型本地
0x2d 变量推送至栈顶	aload_3	将第四个引用类型本地

5.2、load系列B

该系列命令负责把数组的某项送到栈顶。该命令根据栈里内容来确定对哪个数组的哪项进行操作。

比如，如果有成员变量：

```
final String names[]={"robin", "hb"};
```

那么这句话：

```
String str=names[0];
```

对应的指令为

```
17: aload_0      //将this引用推送至栈顶，即压入栈。
18: getfield #5; //Field names:[Ljava/lang/String;
    //将栈顶的指定的对象的第5个实例域（Field）的值（这个值可能是引用，这里就是引用）压入栈顶
21: iconst_0      //数组的索引值（下标）推至栈顶，即压入栈
22: aaload        //根据栈里内容来把name数组的第一项的值推至栈顶
23: astore 5      //把栈顶的值存到str变量里。因为str在我的程序中是其所在非静态函数的第5个变量(从0开始计数),
```

指令码	助记符	说明
0x2e	iaload	将int型数组指定索引的值推送至栈顶

0x2f	laload	将long型数组指定索引的值推送至栈顶
0x30	faload	将float型数组指定索引的值推送至栈顶
0x31	daload	将double型数组指定索引的值推送至栈顶
0x32	aaload	将引用型数组指定索引的值推送至栈顶
0x33	baload	将boolean或byte型数组指定索引的值推送至栈顶
0x34	caload	将char型数组指定索引的值推送至栈顶
0x35	saload	将short型数组指定索引的值推送至栈顶

六、store系列

6.1、store系列A

该系列命令负责把【操作数栈栈顶的值】存入【本地变量表】。这里的本地变量不仅可以是数值类型，还可以是引用类型。

- 如果是把栈顶的值存入到前四个本地变量的话，采用的是`istore_0`、`istore_1`、`istore_2`、`istore_3`(它们分别表示第0，1，2，3个本地整型变量)这种不带参数的简化命令形式。
- 如果是把栈顶的值存入到第四个以上本地变量的话，将使用`istore`命

令这种形式，在它后面给一参数，以表示是把栈顶的值存入到第几个(从0开始)本地变量中。对本地变量所进行的编号，是对所有类型的本地变量进行的（并不按照类型分类）。

- 对于非静态函数，第一变量是this，它是只读的。
- 还有函数传入参数也算本地变量，在进行编号时，它是先于函数体的本地变量的。

指令码	助记符	说明
0x36 本地变量	istore	将栈顶int型数值存入指定
0x37 本地变量	lstore	将栈顶long型数值存入指
0x38 指定本地变量	fstore	将栈顶float型数值存入
0x39 指定本地变量	dstore	将栈顶double型数值存入
0x3a 本地变量	astore	将栈顶引用型数值存入指定
0x3b 本地变量	istore_0	将栈顶int型数值存入第一个
0x3c 本地变量	istore_1	将栈顶int型数值存入第二个
0x3d 本地变量	istore_2	将栈顶int型数值存入第三个

0x3e 本地变量	istore_3	将栈顶int型数值存入第四个
0x3f 个本地变量	lstore_0	将栈顶long型数值存入第一
0x40 个本地变量	lstore_1	将栈顶long型数值存入第二
0x41 个本地变量	lstore_2	将栈顶long型数值存入第三
0x42 个本地变量	lstore_3	将栈顶long型数值存入第四
0x43 个本地变量	fstore_0	将栈顶float型数值存入第一
0x44 个本地变量	fstore_1	将栈顶float型数值存入第二
0x45 个本地变量	fstore_2	将栈顶float型数值存入第三
0x46 个本地变量	fstore_3	将栈顶float型数值存入第四
0x47 个本地变量	dstore_0	将栈顶double型数值存入第一
0x48 个本地变量	dstore_1	将栈顶double型数值存入第二

0x49 个本地变量	dstore_2	将栈顶double型数值存入第三个本地变量
0x4a 个本地变量	dstore_3	将栈顶double型数值存入第四个本地变量
0x4b 本地变量	astore_0	将栈顶引用型数值存入第一个本地变量
0x4c 本地变量	astore_1	将栈顶引用型数值存入第二个本地变量
0x4d 本地变量	astore_2	将栈顶引用型数值存入第三个本地变量
0x4e 本地变量	astore_3	将栈顶引用型数值存入第四个本地变量

6.2、store系列B

该系列命令负责把栈顶项的值存到数组里。该命令根据栈里内容来确定对哪个数组的哪项进行操作。

比如，如下代码：

```
int moneys[] = new int[5];

moneys[1] = 100;
```

其对应的指令为：

```
49: iconst_5

50: newarray int
```

```
52: astore 11
```

```
54: aload 11
```

```
56: iconst_1
```

```
57: bipush 100
```

```
59: iastore
```

```
60: lload 6          //因为str在我的程序中是其所非静态在函数的第6  
个变量(从0开始计数)。
```

指令码	助记符	说明
0x4f 的指定索引位置	iastore	将栈顶int型数值存入指定数组
0x50 的指定索引位置	lastore	将栈顶long型数值存入指定数组
0x51 组的指定索引位置	fastore	将栈顶float型数值存入指定数
0x52 组的指定索引位置	dastore	将栈顶double型数值存入指定数
0x53 指定索引位置	aastore	将栈顶引用型数值存入指定数组的

0x54	bastore	将栈顶boolean或byte型数值存入指定数组的指定索引位置
0x55	castore	将栈顶char型数值存入指定数组的指定索引位置
0x56	sastore	将栈顶short型数值存入指定数组的指定索引位置

七、pop系列

[该系列命令似乎只是简单对栈顶进行操作](#)，更多详情待补充。

指令码	助记符	说明
0x57	pop	将栈顶数值弹出（数值不能是long或double类型的）
0x58	pop2	将栈顶的一个（long或double类型的）或两个数值弹出（其它）
0x59	dup	复制栈顶数值（数值不能是long或double类型的）并将复制值压入栈顶
0x5a	dup_x1	复制栈顶数值（数值不能是long或double类型的）并将两个复制值压入栈顶
0x5b	dup_x2	复制栈顶数值（数值不能是long或double类型的）并将三个（或两个）复制值压入栈顶
0x5c	dup2	复制栈顶一个（long或double类型的）或两个（其它）数值并将复制值压入栈顶

0x5d dup2_x1 复制栈顶数值 (long或double类型的) 并将两个复制值压入栈顶

0x5e dup2_x2 复制栈顶数值 (long或double类型的) 并将三个 (或两个) 复制值压入栈顶

八、栈顶元素数学操作及移位操作系列

该系列命令用于对栈顶元素行数学操作，和对数值进行移位操作。移位操作的操作数和要移位的数都是从栈里取得。

比如对于代码：

```
int k=100;k=k>>1;
```

其对应的JVM指令为：

```
60: bipush 100
```

```
62: istore 12 // 因为k在我的程序中是其所在非静态函数的第12个变量 (从0开始计数)。
```

```
64: iload 12
```

```
66: iconst_1
```

```
67: ishr
```

```
68: istore 12
```

指令码	助记符	说明
0x5f 值不能是long或double类型的)	swap	将栈最顶端的两个数值互换(数
0x60 结果压入栈顶	iadd	将栈顶两int型数值相加并将
0x61 结果压入栈顶	ladd	将栈顶两long型数值相加并将
0x62 结果压入栈顶	fadd	将栈顶两float型数值相加并将
0x63 结果压入栈顶	dadd	将栈顶两double型数值相加并将
0x64 果压入栈顶	isub	将栈顶两int型数值相减并将结
0x65 果压入栈顶	lsub	将栈顶两long型数值相减并将结
0x66 结果压入栈顶	fsub	将栈顶两float型数值相减并将
0x67 结果压入栈顶	dsub	将栈顶两double型数值相减并将
0x68 压入栈顶	imul	将栈顶两int型数值相乘并将结果

0x69 结果压入栈顶	lmul	将栈顶两long型数值相乘并将结果压入栈顶
0x6a 结果压入栈顶	fmul	将栈顶两float型数值相乘并将结果压入栈顶
0x6b 结果压入栈顶	dmul	将栈顶两double型数值相乘并将结果压入栈顶
0x6c 结果压入栈顶	idiv	将栈顶两int型数值相除并将结果压入栈顶
0x6d 结果压入栈顶	ldiv	将栈顶两long型数值相除并将结果压入栈顶
0x6e 结果压入栈顶	fdiv	将栈顶两float型数值相除并将结果压入栈顶
0x6f 将结果压入栈顶	ddiv	将栈顶两double型数值相除并将结果压入栈顶
0x70 将结果压入栈顶	irem	将栈顶两int型数值作取模运算并将结果压入栈顶
0x71 并将结果压入栈顶	lrem	将栈顶两long型数值作取模运算并将结果压入栈顶
0x72 算并将结果压入栈顶	frem	将栈顶两float型数值作取模运算并将结果压入栈顶
0x73 算并将结果压入栈顶	drem	将栈顶两double型数值作取模运算并将结果压入栈顶

0x74 入栈顶	ineg	将栈顶int型数值取负并将结果压
0x75 压入栈顶	lneg	将栈顶long型数值取负并将结果
0x76 压入栈顶	fneg	将栈顶float型数值取负并将结果
0x77 压入栈顶	dneg	将栈顶double型数值取负并将结果
0x78 将结果压入栈顶	ishl	将int型数值左移位指定位数并
0x79 并将结果压入栈顶	lshl	将long型数值左移位指定位数
0x7a 定位数并将结果压入栈顶	ishr	将int型数值右（符号）移位指
0x7b 指定位数并将结果压入栈顶	lshr	将long型数值右（符号）移位
0x7c 定位数并将结果压入栈顶	iushr	将int型数值右（无符号）移位指
0x7d 指定位数并将结果压入栈顶	lushr	将long型数值右（无符号）移位
0x7e 并将结果压入栈顶	iand	将栈顶两int型数值作“按位与”

0x7f land
与"并将结果压入栈顶

将栈顶两long型数值作"按位

0x80 ior
或"并将结果压入栈顶

将栈顶两int型数值作"按位

0x81 lor
或"并将结果压入栈顶

将栈顶两long型数值作"按位

0x82 ixor
或"并将结果压入栈顶

将栈顶两int型数值作"按位异

0x83 lxor
或"并将结果压入栈顶

将栈顶两long型数值作"按位异



运算指令

- 1、运算或算术指令用于对两个操作数栈上的值进行某种特定运算，并把结果重新存入到操作栈顶。
- 2、算术指令分为两种：[整型运算的指令和浮点型运算的指令](#)。
- 3、无论是哪种算术指令，都使用Java虚拟机的数据类型，[由于没有直接支持byte、short、char和boolean类型的算术指令，使用操作int类型的指令代替](#)。

加法指令：iadd、ladd、fadd、dadd。
减法指令：isub、lsub、fsub、dsub。
乘法指令：imul、lmul、fmul、dmul。
除法指令：idiv、ldiv、fdiv、ddiv。
求余指令：irem、lrem、frem、drem。
取反指令：ineg、lneg、fneg、dneg。
位移指令：ishl、ishr、iushr、lshl、lshr、lushr。
按位或指令：ior、lor。
按位与指令：iand、land。
按位异或指令：ixor、lxor。
局部变量自增指令：iinc。
比较指令：dcmpg、dcmpl、fcmpg、fcmpl、lcmp。

类型转换指令

- 1、类型转换指令可以将两种不同的数值类型进行相互转换。
- 2、这些转换操作一般用于实现用户代码中的显式类型转换操作，或者用来处理字节码指令集中数据类型相关指令无法与数据类型一一对应的问题。

宽化类型转换

int类型到long、float或者double类型。

long类型到float、double类型。

float类型到double类型。

i2l、f2b、l2f、l2d、f2d。

窄化类型转换

i2b、i2c、i2s、l2i、f2i、f2l、d2i、d2l和d2f。

对象创建与访问指令

创建类实例的指令：[new](#)。

创建数组的指令：[newarray](#)、[anewarray](#)、[multianewarray](#)。

访问类字段（static字段，或者称为类变量）和实例字段（非static字段，或者称为实例变量）的指令：[getfield](#)、[putfield](#)、[getstatic](#)、[putstatic](#)。

把一个数组元素加载到操作数栈的指令：[baload](#)、[caload](#)、[saload](#)、[iaload](#)、[laload](#)、[faload](#)、[daload](#)、[aaload](#)。

将一个操作数栈的值存储到数组元素中的指令：[bastore](#)、[castore](#)、[sastore](#)、[iastore](#)、[fastore](#)、[dastore](#)、[aastore](#)。

取数组长度的指令：[arraylength](#)。

检查类实例类型的指令：[instanceof](#)、[checkcast](#)。

操作数栈管理指令

直接操作操作数栈的指令：

将操作数栈的栈顶一个或两个元素出栈：[pop](#)、[pop2](#)。

复制栈顶一个或两个数值并将复制值或双份的复制值重新压入栈顶：[dup](#)、[dup2](#)、[dup_x1](#)、[dup2_x1](#)、[dup_x2](#)、[dup2_x2](#)。

将栈最顶端的两个数值互换：[swap](#)。

控制转移指令

1、控制转移指令可以让Java虚拟机有条件或无条件地从指定的位置指令而不是控制转移指令的下一条指令继续执行程序。

2、从概念模型上理解，可以认为控制转移指令就是在有条件或无条件地修改PC寄存器的值。

条件分支：[ifeq](#)、[iflt](#)、[ifle](#)、[ifne](#)、[ifgt](#)、[ifge](#)、[ifnull](#)、[ifnonnull](#)、[if_icmpeq](#)、[if_icmpne](#)、[if_icmplt](#)、[if_icmpgt](#)、[if_icmple](#)、[if_icmpge](#)、[if_acmpeq](#)和[if_acmpne](#)。

复合条件分支：[tableswitch](#)、[lookupswitch](#)。

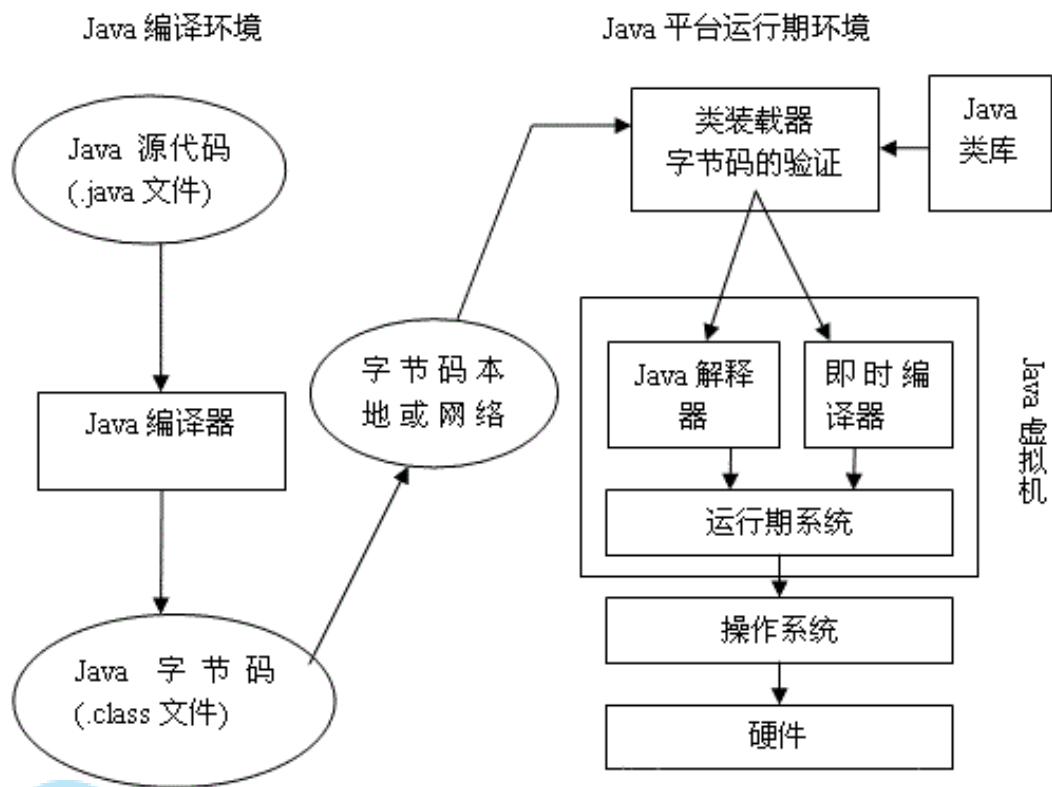
无条件分支：[goto](#)、[goto_w](#)、[jsr](#)、[jsr_w](#)、[ret](#)。

在Java虚拟机中有专门的指令集用来处理int和reference类型的条件分支比较操作，为了可以无须明显标识一个实体值是否null，也有专门的指令用来检测null值。

JVM程序执行流程

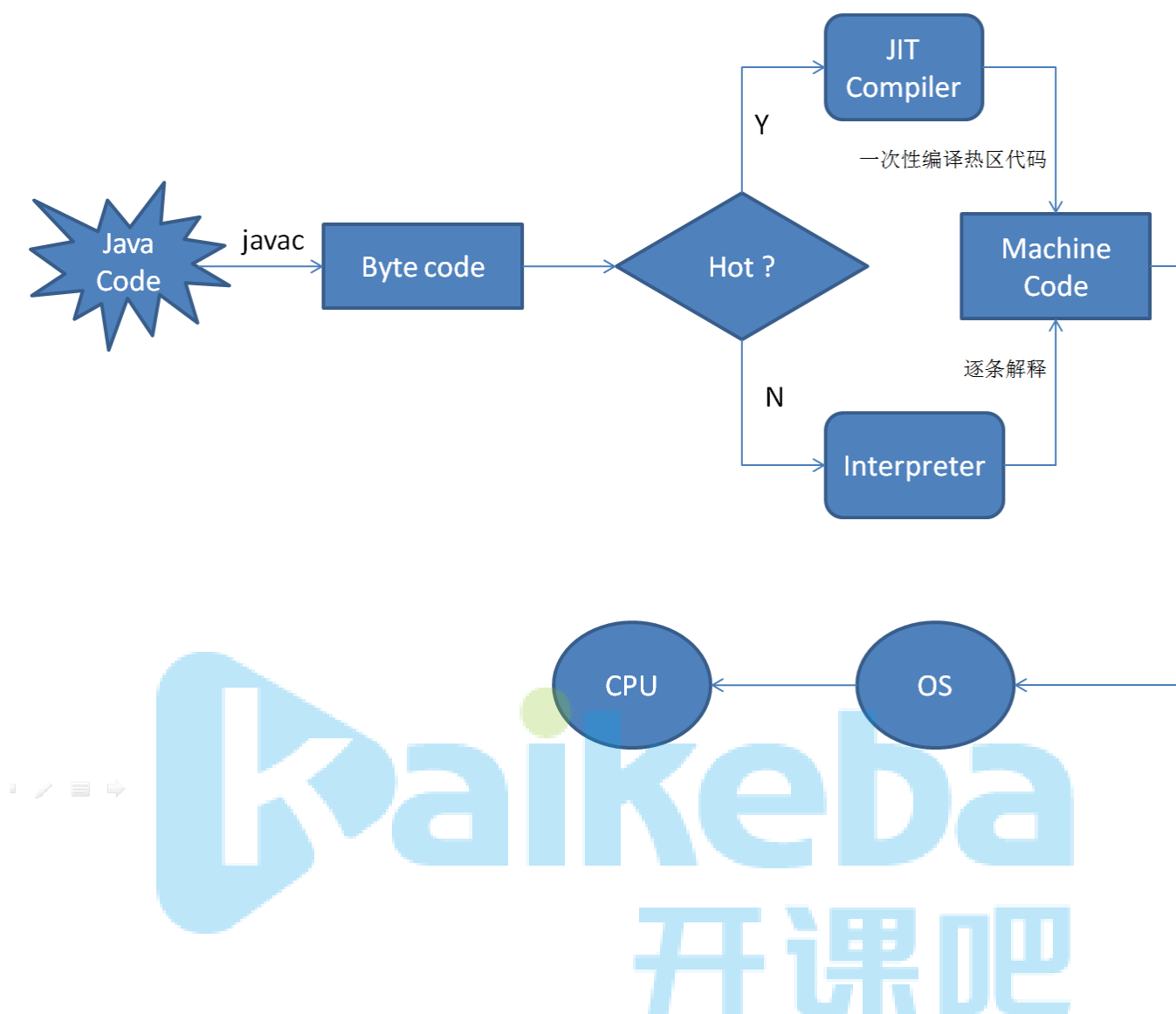
执行流程图

Java编译成字节码、动态编译和解释为机器码的过程分析：



编译器和解释器的协调工作流程：





在部分商用虚拟机中（如HotSpot），Java程序最初是通过解释器（Interpreter）进行解释执行的，当虚拟机发现某个方法或代码块的运行特别频繁时，就会把这些代码认定为“热点代码”。为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，完成这个任务的编译器称为即时编译器（[Just In Time Compiler](#)，下文统称JIT编译器）。

由于Java虚拟机规范并没有具体的约束规则去限制即使编译器应该如何实现，所以这部分功能完全是与虚拟机具体实现相关的内容，**如无特殊说明，我们提到的编译器、即时编译器都是指Hotspot虚拟机内的即时编译器，虚拟机也是特指HotSpot虚拟机。**

我们的JIT是属于动态编译方式的，动态编译（dynamic compilation）指的是“在运行时进行编译”；与之相对的是事前编译（[ahead-of-time compilation](#)，简称AOT），也叫静态编译（static compilation）。

JIT编译（just-in-time compilation）狭义来说是当某段代码即将第一次被执行时进行编译，因而叫“即时编译”。JIT编译是动态编译的一种特例。JIT编译一词后来被泛化，时常与动态编译等价；但要注意广义与狭义的JIT编译所指的区别。

热点代码

程序中的代码只有是热点代码时，才会编译为本地代码，那么什么是热点代码呢？

运行过程中会被即时编译器编译的“热点代码”有两类：

1. [被多次调用的方法。](#)
2. [被多次执行的循环体。](#)

[两种情况，编译器都是以整个方法作为编译对象。](#)这种编译方法因为编译发生在方法执行过程之中，因此形象的称之为栈上替换（On Stack Replacement, OSR），即方法栈帧还在栈上，方法就被替换了。

热点检测方式

要知道方法或一段代码是不是热点代码，是不是需要触发即时编译，需要进行Hot Spot Detection（热点探测）。

目前主要的热点探测方式有以下两种：

- [基于采样的热点探测](#)

采用这种方法的虚拟机会周期性地检查各个线程的栈顶，如果发现某些方法经常出现在栈顶，那这个方法就是“热点方法”。这种探测方法的好处是实现简单高效，还可以很容易地获取方法调用关系（将调用堆栈展开即可），缺点是很难精确地确认一个方法的热度，容易因为受到线程阻塞或别的外界因素的影响而扰乱热点探测。

- [基于计数器的热点探测---采用这种](#)

采用这种方法的虚拟机会为每个方法（甚至是代码块）建立计数器，统计方法的执行次数，如果执行次数超过一定的阈值，就认为它是“热点方法”。这种统计方法实现复杂一些，需要为每个方法建立并维护计数器，而且不能直接获取到方法的调用关系，但是它的统计结果相对更加精确严谨。

在HotSpot虚拟机中使用的是第二种——基于计数器的热点探测方法，因此它为每个方法准备了两个计数器：方法调用计数器和回边计数器。在确定虚拟机运行参数的前提下，这两个计数器都有一个确定的阈值，当计数器超过阈值溢出了，就会触发JIT编译。

方法调用计数器

顾名思义，这个计数器用于统计方法被调用的次数。

在JVM client模式下的阈值是1500次，Server是10 000次。可以通过虚拟机参数：-XX: CompileThreshold设置。但是JVM还存在热度衰减，时间段内调用方法的次数较少，计数器就减小。

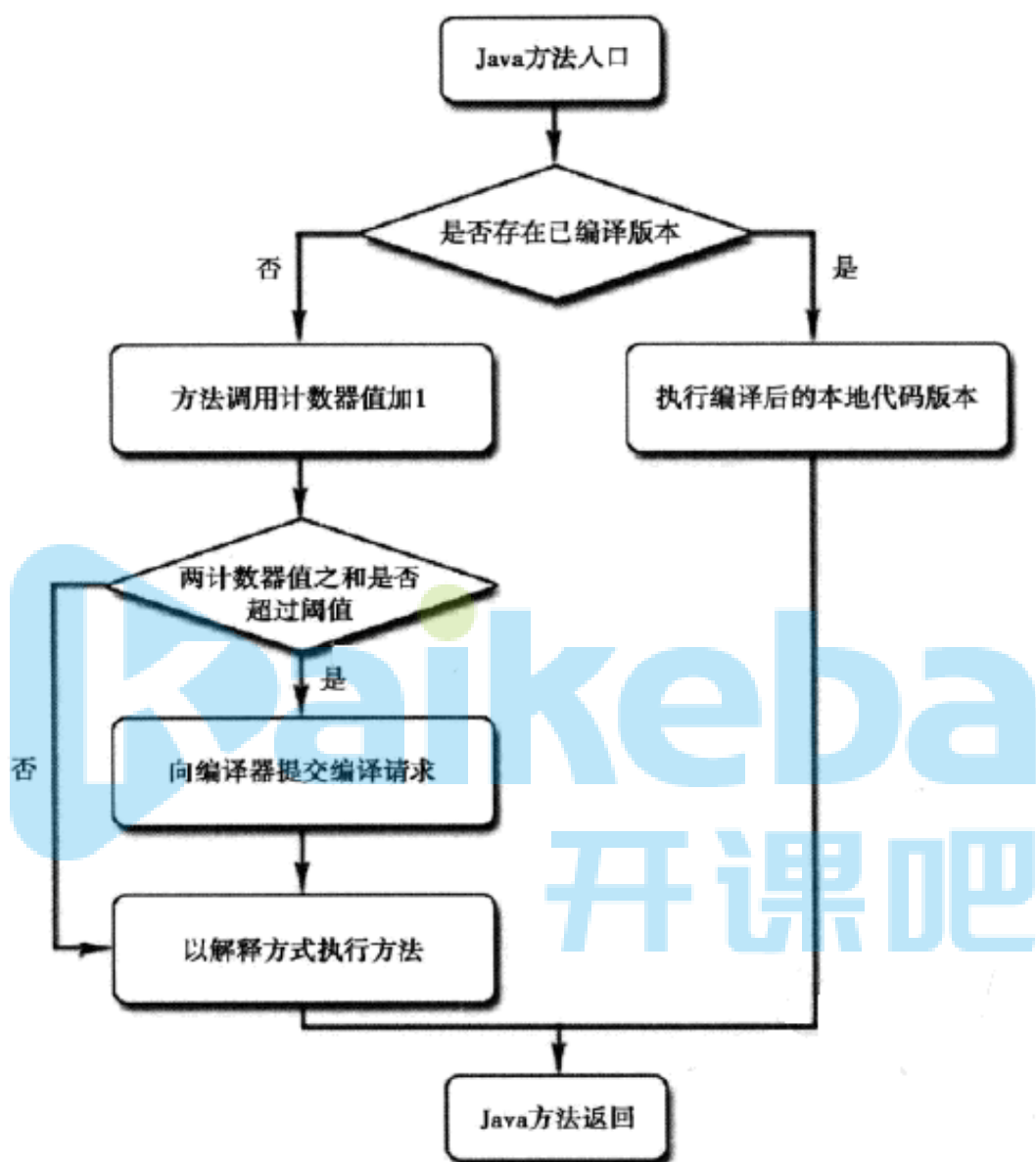


图 11-2 方法调用计数器触发即时编译

回边计数器

它的作用就是统计一个方法中循环体代码执行的次数，在字节码中遇到控制流向后跳转的指令称为“回边”。

解释器方法执行

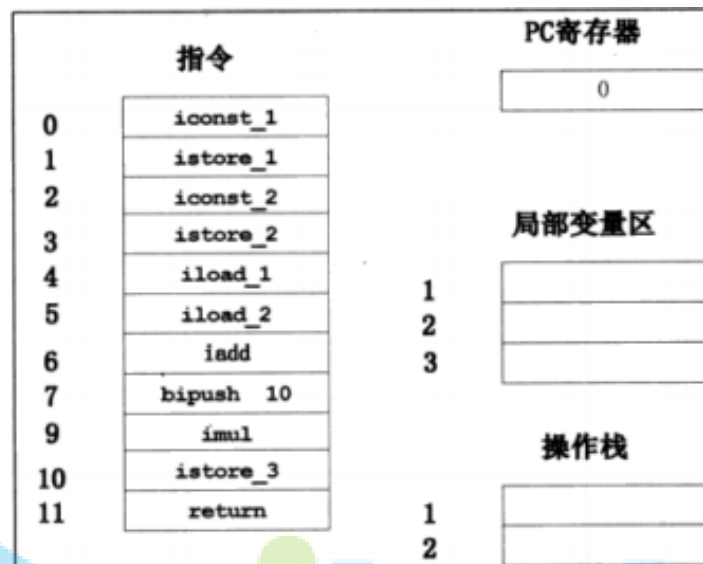
以下面代码为例看一下执行引擎是如何将一段代码在执行部件上执行的，如下一段【Java源代码】：

```
public class Math{  
    public static void main(String[] args){  
        int a = 1 ;  
        int b = 2;  
        int c = (a+b)*10;  
    }  
}
```

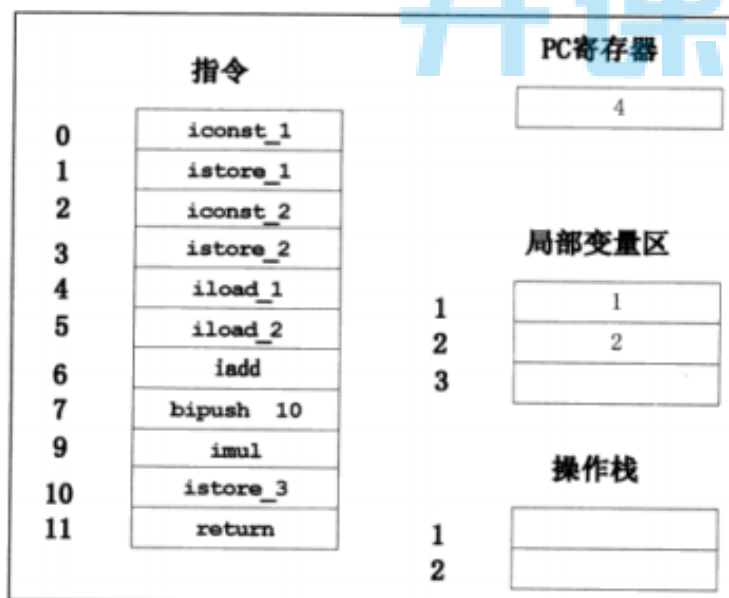
其中main的【class字节码】指令如下：

偏移量	指令	说明
0:	iconst_1	常数1入栈
1:	istore_1	将栈顶元素移入本地变量1存储
2:	iconst_2	常数2入栈
3:	istore_2	将栈顶元素移入本地变量2存储
4:	iload_1	本地变量1入栈
5:	iload_2	本地变量2入栈
6:	iadd	弹出栈顶两个元素相加
7:	bipush 10	将10入栈
9:	imul	栈顶两个元素相乘
10:	istore_3	栈顶元素移入本地变量3存储

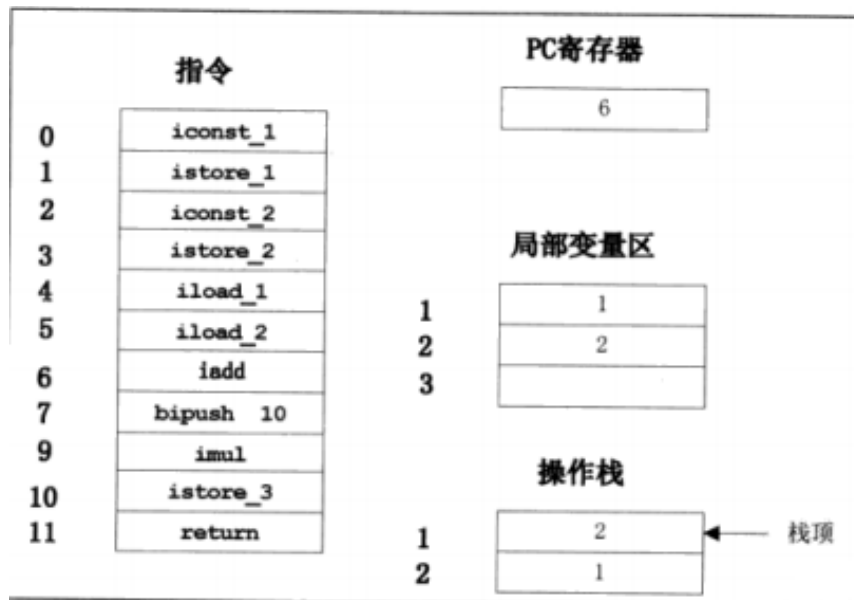
对应到执行引擎的各执行部件如图：



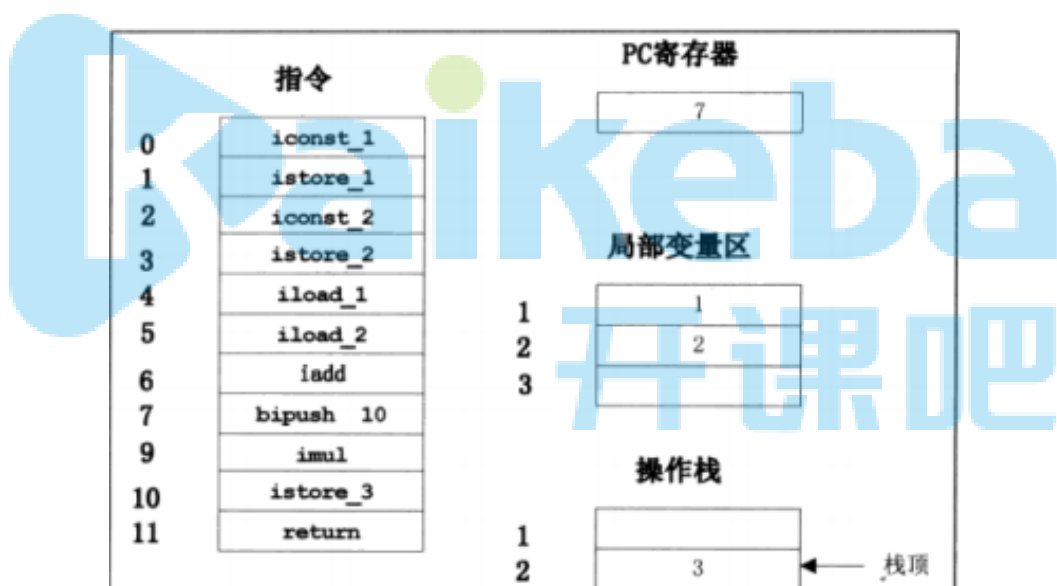
在开始执行方法之前，PC寄存器存储的指针是第1条指令的地址，局部变量区和操作栈都没有数据。从第1条到第4条指令分别将a、b两个本地变量赋值，对应到局部变量区就是1和2分别存储常数1和2，如图：



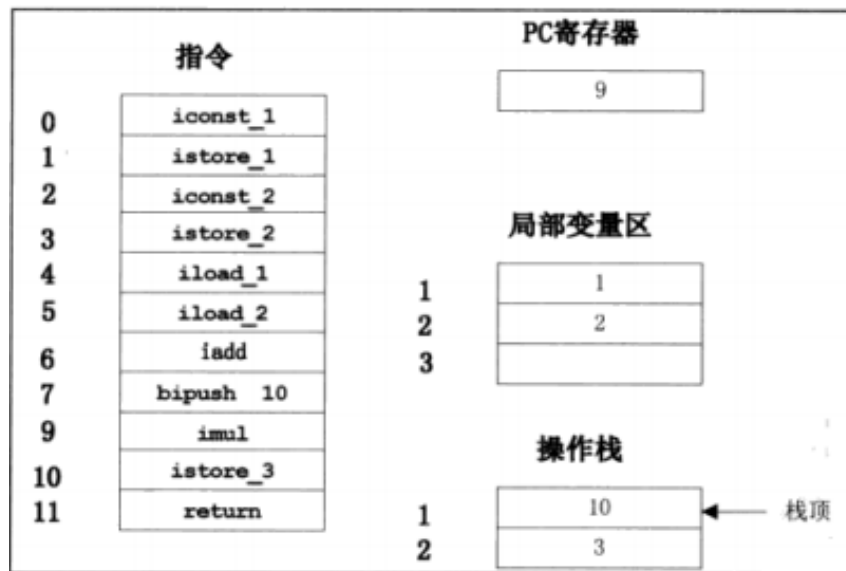
第5条和第6条指令分别是将两个局部变量入栈，然后相加，如图：



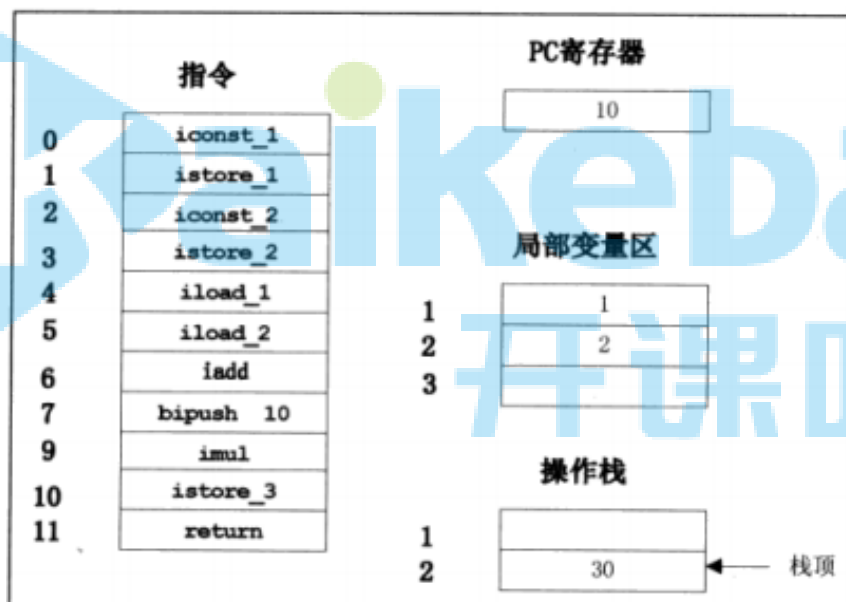
1先入栈2后入栈，栈顶元素是2，第7条指令是将栈顶的两个元素弹出后相加，结果再入栈，如图：



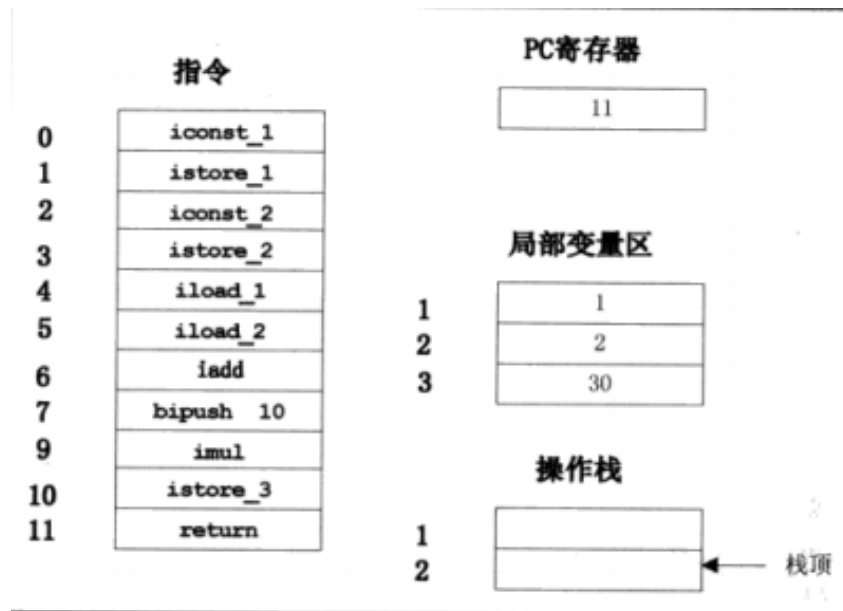
可以看出，变量a和b相加的结果3存在当前栈顶中，接下来第8条指令将10入栈，如图：



当前PC寄存器执行的地址是9，下一个操作是将当前栈的两个操作数弹出进行相乘并把结果压入栈中，如图：



第10条指令是将当前的栈顶元素存入局部变量3中，如图：



第10条指令执行完后栈中元素出栈，出栈的元素存储在局部变量区3中，对应的是变量c的值。最后一条指令是`return`，这条指令执行完后当前的这个方法对应的这些部件会被JVM回收，局部变量区的所有值将全部释放，PC寄存器会被销毁，在Java栈中与这个方法对应的栈帧将消失。

JIT使用

- 为何HotSpot需要使用解释器和编译器并存的架构？
- JVM为什么要实现两个不同的即时编译器？
- 程序何时会使用解释器执行？何时会使用编译器执行？
- 哪些程序代码会被编译成为本地代码？如何编译？
- JAVA代码的执行效率就一定比C，C++静态执行的执行差？JAVA代码解析执行有何优势？

为什么要使用解释器与编译器并存的架构

尽管并不是所有的Java虚拟机都采用解释器与编译器并存的架构，但许多主流的商用虚拟机（如HotSpot），都同时包含解释器和编译器。

解释器与编译器特点

- 当程序需要迅速启动和执行的时候，解释器可以首先发挥作用，省去编译的时间，立即执行。在程序运行后，随着时间的推移，编译器逐渐发挥作用，把越来越多的代码编译成本地代码之后，可以获取更高的执行效率。
- 当程序运行环境中内存资源限制较大（如部分嵌入式系统中），可以使用解释器执行节约内存，反之可以使用编译执行来提升效率。

编译的时间开销

解释器的执行，抽象的看是这样的：

输入的代码 -> [解释器 解释执行] -> 执行结果

而要JIT编译然后再执行的话，抽象的看则是：

输入的代码 -> [编译器 编译] -> 编译后的代码 -> [执行] -> 执行结果

说JIT比解释快，其实说的是“执行编译后的代码”比“解释器解释执行”要快，并不是说“编译”这个动作比“解释”这个动作快。JIT编译再怎么快，至少也比解释执行一次略慢一些，而要得到最后的执行结果还得再经过一个“执行编译后的代码”的过程。所以，对“只执行一次”的代码而言，解释执行其实总是比JIT编译执行要快。

怎么算是“只执行一次的代码”呢？粗略说，下面两个条件同时满足时就是严格的“只执行一次”

1、只被调用一次，例如类的构造器（class initializer, ()）

2、没有循环

对只执行一次的代码做JIT编译再执行，可以说是得不偿失。

对只执行少量次数的代码，JIT编译带来的执行速度的提升也未必能抵消掉最初编译带来的开销。

只有对频繁执行的代码，JIT编译才能保证有正面的收益。

编译的空间开销

对一般的Java方法而言，编译后代码的大小相对于字节码的大小，膨胀比达到10x是很正常的。同上面说的时间开销一样，这里的空间开销也是，只有对执行频繁的代码才值得编译，如果把所有代码都编译则会显著增加代码所占空间，导致“代码爆炸”。

这也就解释了为什么有些JVM会选择不总是做JIT编译，而是选择用解释器+JIT编译器的混合执行引擎。

为何要实现两个不同的即时编译器

HotSpot虚拟机中内置了两个即时编译器：[Client Compiler](#)和[Server Compiler](#)，简称为C1、C2编译器，分别用在客户端和服务端。

目前主流的HotSpot虚拟机中默认是采用解释器与其中一个编译器直接配合的方式工作。程序使用哪个编译器，取决于虚拟机运行的模式。HotSpot虚拟机会根据自身版本与宿主机器的硬件性能自动选择运行模式，用户也可以使用“-client”或“-server”参数去强制指定虚拟机运行在Client模式或Server

模式。

用Client Compiler获取更高的编译速度，用Server Compiler 来获取更好的编译质量。为什么提供多个即时编译器与为什么提供多个垃圾收集器类似，都是为了适应不同的应用场景。

如何编译为本地代码?

Server Compiler和Client Compiler两个编译器的编译过程是不一样的。

对Client Compiler来说，它是一个简单快速的编译器，主要关注点在于局部优化，而放弃许多耗时较长的全局优化手段。

而Server Compiler则是专门面向服务器端的，并为服务端的性能配置特别调整过的编译器，是一个充分优化过的高级编译器。

JIT优化

HotSpot 虚拟机使用了很多种优化技术，这里只简单介绍其中的几种，完整的优化技术介绍可以参考官网内容。

公共子表达式的消除

公共子表达式消除是一个普遍应用于各种编译器的经典优化技术，他的含义是：如果一个表达式E已经计算过了，并且从先前的计算到现在E中所有变量的值都没有发生变化，那么E的这次出现就成为了公共子表达式。对于这种表达式，没有必要花时间再对他进行计算，只需要直接用前面计算过的表达式结果代替E就可以了。

如果这种优化仅限于程序的基本块内，便称为局部公共子表达式消除 (Local Common Subexpression Elimination)

如果这种优化范围涵盖了多个基本块，那就称为全局公共子表达式消除 (Global Common Subexpression Elimination) 。

举个简单的例子来说明他的优化过程，假设存在如下代码：

```
int d = (c*b)*12+a+(a+b*c);
```

如果这段代码交给Javac编译器则不会进行任何优化，那生成的代码如下所示，是完全遵照Java源码的写法直译而成的。

```
iload_2 // b
imul // 计算b*c
bipush 12 // 推入12
imul // 计算(c*b)*12
iload_1 // a
iadd // 计算(c*b)*12+a
iload_1 // a
iload_2 // b
iload_3 // c
imul // 计算b*c
iadd // 计算a+b*c
iadd // 计算(c*b)*12+a+(a+b*c)
istore 4
```

当这段代码进入到虚拟机即时编译器后，他将进行如下优化：[编译器检测到“cb”与“bc”是一样的表达式，而且在计算期间b与c的值是不变的](#)。因此，这条表达式就可能被视为：

```
int d = E*12+a+(a+E);
```

这时，编译器还可能（取决于哪种虚拟机的编译器以及具体的上下文而定）进行另外一种优化：[代数化简 \(Algebraic Simplification\)](#)，把表达式变为：

```
int d = E*13+a*2;
```

表达式进行变换之后，再计算起来就可以节省一些时间了。

方法内联

在使用JIT进行即时编译时，将方法调用直接使用方法体中的代码进行替换，这就是方法内联，减少了方法调用过程中压栈与入栈的开销。同时为之后的一些优化手段提供条件。如果JVM监测到一些小方法被频繁的执行，它会把方法的调用替换成方法体本身。

比如说下面这个：

```
private int add4(int x1, int x2, int x3, int x4) {  
    return add2(x1, x2) + add2(x3, x4);  
}  
private int add2(int x1, int x2) {  
    return x1 + x2;  
}
```

可以肯定的是运行一段时间后JVM会把add2方法去掉，并把你的代码翻译成：

```
private int add4(int x1, int x2, int x3, int x4) {  
    return x1 + x2 + x3 + x4;  
}
```

方法逃逸分析

逃逸分析(Escape Analysis)是目前Java虚拟机中比较前沿的优化技术。这是一种可以有效减少Java 程序中同步负载和内存堆分配压力的跨函数全局数据流分析算法。通过逃逸分析,Java Hotspot编译器能够分析出一个新的对象的引用的使用范围从而决定是否要将这个对象分配到堆上。

逃逸分析的基本行为就是分析对象动态作用域：当一个对象在方法中被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他地方中，称为方法逃逸。

逃逸分析包括：

- 全局变量赋值逃逸
- 方法返回值逃逸
- 实例引用发生逃逸
- 线程逃逸:赋值给类变量或可以在其他线程中访问的实例变量

例如：

```
public class EscapeAnalysis {  
  
    //全局变量  
    public static Object object;  
  
    //public Object object;  
  
    public void globalVariableEscape(){//全局变量赋值逃逸  
        object = new Object();  
    }  
  
    public Object methodEscape(){ //方法返回值逃逸
```

```

        return new Object();
    }

    public void instancePassEscape(){ //实例引用发生逃逸
        EscapeAnalysis escapeAnalysis = null;
        this.speak(escapeAnalysis);
        //下面就可以获取escapeAnalysis的引用
    }

    public void speak(EscapeAnalysis escapeAnalysis){
        escapeAnalysis = new Object();
        System.out.println("Escape Hello");
    }
}

```



使用方法逃逸的案例进行分析：

```

public static StringBuffer craeteStringBuffer(String s1,
String s2) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb;
}

```

StringBuffer sb是一个方法内部变量，上述代码中直接将sb返回，这样这个StringBuffer有可能被其他方法所改变，这样它的作用域就不只是在方法内部，虽然它是一个局部变量，称其逃逸到了方法外部。甚至还有可能被外部线程访问到，譬如赋值给类变量或可以在其他线程中访问的实

例变量，称为线程逃逸。

上述代码如果想要StringBuffer sb不逃出方法，可以这样写：

```
public static String createStringBuffer(String s1, String
s2) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb.toString();
}
```

不直接返回 StringBuffer，那么StringBuffer将不会逃逸出方法。

使用逃逸分析，编译器可以对代码做如下优化：

- 一、同步省略。如果一个对象被发现只能从一个线程被访问到，那么对于这个对象的操作可以不考虑同步。
- 二、将堆分配转化为栈分配。如果一个对象在子程序中被分配，要使指向该对象的指针永远不会逃逸，对象可能是栈分配的候选，而不是堆分配。
- 三、分离对象或标量替换。有的对象可能不需要作为一个连续的内存结构存在也可以被访问到，那么对象的部分（或全部）可以不存储在内存，而是存储在CPU寄存器中。

在Java代码运行时，通过JVM参数可指定是否开启逃逸分析，

- XX:+DoEscapeAnalysis : 表示开启逃逸分析
- XX:-DoEscapeAnalysis : 表示关闭逃逸分析

从jdk 1.7开始已经默认开始逃逸分析，如需关闭，需要指定 `-XX:-DoEscapeAnalysis`

对象的栈上内存分配

我们知道，在一般情况下，对象和数组元素的内存分配是在堆内存上进行的。但是随着JIT编译器的日渐成熟，很多优化使这种分配策略并不绝对。JIT编译器就可以在编译期间根据逃逸分析的结果，来决定是否可以将对象的内存分配从堆转化为栈。

我们来看以下代码：

```
public class EscapeAnalysisTest {
    public static void main(String[] args) {
        long a1 = System.currentTimeMillis();
        for (int i = 0; i < 1000000; i++) {
            alloc();
        }
        // 查看执行时间
        long a2 = System.currentTimeMillis();
        System.out.println("cost " + (a2 - a1) + " ms");
        // 为了方便查看堆内存中对象个数，线程sleep
        try {
            Thread.sleep(100000);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
    }

    // 分析该方法中的User对象是否发生方法逃逸
    private static void alloc() {
        User user = new User();
    }
}
```



```
}

static class User {
}
}
```

其实代码内容很简单，就是使用for循环，在代码中创建100万个User对象。

我们在alloc方法中定义了User对象，但是并没有在方法外部引用他。也就是说，这个对象并不会逃逸到alloc外部。经过JIT的逃逸分析之后，就可以对其内存分配进行优化。

我们指定以下JVM参数并运行：

```
-Xmx4G -Xms4G -XX:-DoEscapeAnalysis -XX:+PrintGCDetails -
XX:+HeapDumpOnOutOfMemoryError
```

在程序打印出 `cost XX ms` 后，代码运行结束之前，我们使用`jmap`命令，来查看下当前堆内存中有多少个User对象：

```

~ jps
2809 StackAllocTest
2810 Jps
~ jmap -histo 2809
num          #instances          #bytes  class name
-----
  1:             524        87282184    [I
  2:        1000000        16000000    StackAllocTest$User
  3:             6806         2093136    [B
  4:             8006         1320872    [C
  5:             4188          100512    java.lang.String
  6:             581           66304    java.lang.Class

```

从上面的jmap执行结果中我们可以看到，堆中共创建了100万个StackAllocTest\$User实例。

在关闭逃避分析的情况下（-XX:-DoEscapeAnalysis），虽然在alloc方法中创建的User对象并没有逃逸到方法外部，但是还是被分配在堆内存中。也就是说，如果没有JIT编译器优化，没有逃逸分析技术，正常情况下就应该是这样的。即所有对象都分配到堆内存中。

接下来，我们开启逃逸分析，再来执行下以上代码。

```

-Xmx4G -Xms4G -XX:+DoEscapeAnalysis -XX:+PrintGCDetails -
XX:+HeapDumpOnOutOfMemoryError

```

在程序打印出 cost XX ms 后，代码运行结束之前，我们使用 jmap 命令，来查看下当前堆内存中有多少个User对象：

```

~ jps
709
2858 Launcher
2859 StackAllocTest

```

```
2860 Jps
~ jmap -histo 2859
num          #instances          #bytes  class name
-----
 1:             524        101944280    [I
 2:            6806         2093136     [B
 3:           83619         1337904  StackAllocTest$User
 4:            8006         1320872     [C
 5:            4188          100512  java.lang.String
 6:             581           66304  java.lang.Class
```

从以上打印结果中可以发现，开启了逃逸分析之后（-XX:+DoEscapeAnalysis），在堆内存中只有8万多个 `StackAllocTest$User` 对象。也就是说在经过JIT优化之后，堆内存中分配的对象数量，从100万降到了8万。

除了以上通过jmap验证对象个数的方法以外，还可以尝试将堆内存调小，然后执行以上代码，根据GC的次数来分析，也能发现，开启了逃逸分析之后，在运行期间，GC次数会明显减少。正是因为很多堆上分配被优化成了栈上分配，所以GC次数有了明显的减少。

总结

所以，如果以后有人问你：是不是所有的对象和数组都会在堆内存分配空间？

那么你可以告诉他：不一定，随着JIT编译器的发展，在编译期间，如果JIT经过逃逸分析，发现有些对象没有逃逸出方法，那么有可能堆内存分配会被优化成栈内存分配。但是这也并不是绝对的。就像我们前面看到的一样，在开启逃逸分析之后，也并不是所有User对象都没有在堆上分配。

标量替换

标量 (Scalar) 是指一个无法再分解成更小的数据的数据。

与之对应的有一个聚合量（对象）。

标量替换，其实指的就是将聚合量拆分为一个个的标量放入栈中。

在JIT阶段，如果经过逃逸分析，发现一个对象不会被外界访问的话，那么经过JIT优化，就会把这个对象拆解成若干个其中包含的若干个成员变量来代替。

```
//有一个类A
public class A{
    public int a=1;
    public int b=2
}

//方法getAB使用类A里面的a, b
private void getAB(){
    A x = new A();
    x.a;
    x.b;
}

//JVM在编译的时候会直接编译成
private void getAB(){
    a = 1;
    b = 2;
}

//这就是标量替换
```

```
// 只要已经创建了对象，那么该对象一定在堆中存储
```

```
// new指令最终一定会产生对象吗?
```

同步锁消除

同样基于逃逸分析，当加锁的变量不会发生逃逸，是线程私有的完全没有必要加锁。在JIT编译时期就可以将同步锁去掉，以减少加锁与解锁造成的资源开销。

```
public class TestLockEliminate {
    public static String getString(String s1, String s2) {
        StringBuffer sb = new StringBuffer();
        sb.append(s1);
        sb.append(s2);
        return sb.toString();
    }

    public static void main(String[] args) {
        long tsStart = System.currentTimeMillis();
        for (int i = 0; i < 10000000; i++) {
            getString("TestLockEliminate ", "Suffix");
        }
        System.out.println("一共耗费: " +
            (System.currentTimeMillis() - tsStart) + " ms");
    }
}
```

getString()方法中的StringBuffer数以函数内部的局部变量，进作用于方法内部，不可能逃逸出该方法，因此他就不可能被多个线程同时访问，也就没有资源的竞争，但是StringBuffer的append操作却需要执行同步操作，代码如下：

```

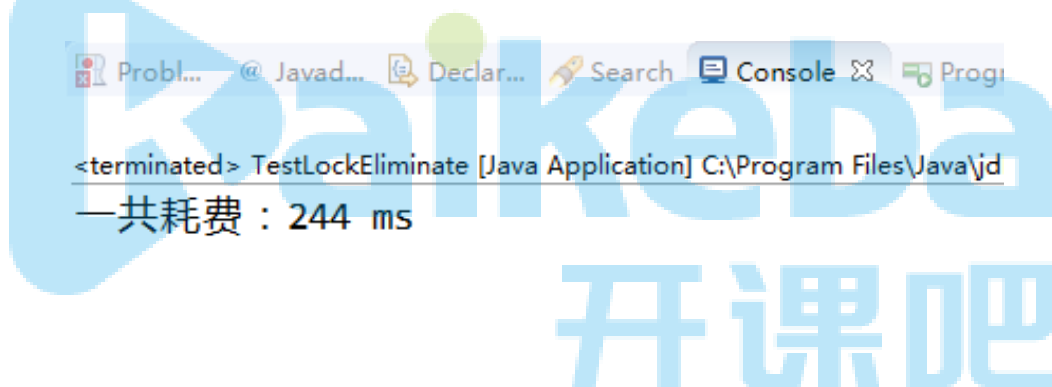
@Override
public synchronized StringBuffer append(String str) {
    toStringCache = null;
    super.append(str);
    return this;
}

```

逃逸分析和锁消除分别可以使用参数 `-XX:+DoEscapeAnalysis` 和 `-XX:+EliminateLocks` (锁消除必须在-server模式下) 开启。使用如下参数运行上面的程序：

```
-XX:+DoEscapeAnalysis -XX:-EliminateLocks
```

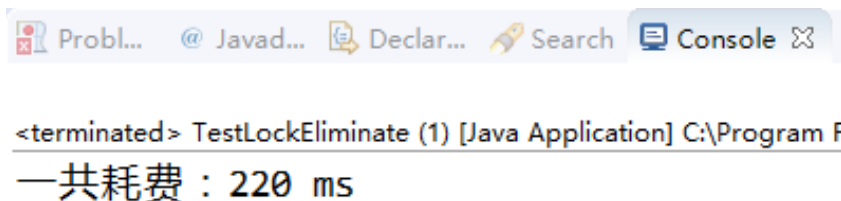
得到如下结果：



使用如下命令运行程序：

```
-XX:+DoEscapeAnalysis -XX:+EliminateLocks
```

得到如下结果：



六、方法调用

学习目标

- 方法是如何被调用的?
 - JVM中方法调用是怎么一个流程?
 - 方法符号引用
 - 方法直接引用
 - 接口方法表
 - 虚方法表
 - 静态方法是如何被调用的?
 - 私有方法是如何被调用的?
 - 构造方法是如何被调用的?
 - 成员方法是如何被调用的?
- 搞清楚什么是虚方法? 什么是非虚方法?
- 搞清楚什么是虚方法表?
- 搞清楚什么是静态绑定? 什么是动态绑定?
- 搞清楚方法重载和重写都发生了什么?
 - 两个类是父子关系, 父类有个static修饰的m1方法, 子类也有个static修饰的m1方法, 问题: 子类m1方法是否会覆盖父类m1方法?
- 方法调用指令有哪些? 分别是干啥的?
- 搞清楚内联缓存和方法内联的区别? 以及内联缓存的作用?

相关知识点

1. 私有方法 ([类和私有方法绑死了](#))
 1. 非虚方法
 2. invokespecial
 3. 静态绑定
2. 构造方法 ([类和构造方法绑死了](#))
 1. 非虚方法
 2. invokespecial
 3. 静态绑定
3. 静态方法 ([类和静态方法绑死了](#))
 1. 非虚方法
 2. invokestatic
 3. 静态绑定
4. 成员方法 (类和成员方法绑死了吗? 思考多态)
 1. 虚方法
 2. invokevirtual
 3. 动态绑定
5. 接口方法 (接口实现类和接口方法绑死了吗?)
 1. 虚方法
 2. invokeinterface
 3. 动态绑定

常见方法调用类型

1. 私有方法（[类和私有方法绑死了](#)）
2. 构造方法（[类和构造方法绑死了](#)）
3. 静态方法（[类和静态方法绑死了](#)）

```
StringUtils.isNotBlank();
```

4. 成员方法（类和成员方法绑死了吗？思考多态）

```
Father f = new Son();
```

```
f.eat(); // 调用的是Father还是Son的eat方法呢？只有在运行的时候才能确定调用是哪个对象的方法
```

5. 接口方法（接口实现类和接口方法绑死了吗？）

```
UserService service = new UserService1();
```

```
UserService service = new UserService2();
```

```
service.save();
```

重载与重写

```
static abstract class Human{}
static class Man extends Human{ }
static class Woman extends Human{ }
```

发生了方法的重载，方法重载会在编译期确定具体调用哪个方法，这个也相当于是一种静态分派

```
public void sayHello(Human guy){
    System.out.println("hello,人类! "); //1
}
public void sayHello(Man guy){
    System.out.println("hello,老铁! "); //2
}
public void sayHello(Woman guy){
    System.out.println("hello,老妹! "); //3
}
```

```
public static void main(String[] args){
```

```
Human h1 = new Man();
Human h2 = new Woman();
```

编译看左边
运行看右边

```
StaticCall02 sd = new StaticCall02();
sd.sayHello(h1);
sd.sayHello(h2);
```

编译时，确定的h1类型是左边的类型，也就是Human类型

```
}
```

虚方法和非虚方法

如果在编译期就可以确定要调用的目标方法，那么该目标方法就被称为非虚方法（静态绑定）。反之都是虚方法（动态绑定）。

非虚方法包括：静态方法、私有方法、final方法、实例构造器、父类方法（super()）

所有非私有实例方法被调用-->编译-->invokevirtual指令。

接口方法调用-->编译-->invokeinterface指令。

这两种指令，均属于Java虚拟机中的[虚方法调用](#)。

方法调用指令

字节码指令

我们再从JVM层面分析下，JVM里面是通过哪里指令来实现方法的调用的：

--非虚方法调用指令

- `invokestatic`:调用静态方法
- `invokespecial`:调用非静态私有方法、构造方法(包括super)

--虚方法调用指令

- `invokeinterface`:调用接口方法([多态]())
- `invokevirtual`:调用非静态非私有方法([多态]())
- `invokedynamic`:动态调用 (Java7引入的，第一次用却是在Java8中，用在了Lambda表达式和默认方法中，它允许调用任意类中的同名方法，注意是任意类，和重载重写不同) (动态 ≠ 多态)

编程语言分为静态语言和动态语言。

- 静态语言：编译期间就确定变量类型。Java是静态语言 (JDK8之后，使用lambda表达式去实现动态语言功能)
- 动态语言：运行请求才能确定变量类型。[JS](#) `var flag = 10; var flag = "kaikeba";`

具体来说，Java字节码指令中与调用相关的指令共有五种：

- [invokevirtual](#)：调用非静态非私有方法([多态](#))

用于[调用对象的实例方法](#)，根据对象的实际类型进行分派（[虚方法分派](#)），这也是Java语言中最常见的方法分派方式。

- [invokeinterface](#)：调用接口方法([多态](#))

用于[调用接口方法](#)，它会在运行时搜索一个实现了这个接口方法的对象，找出适合的方法进行调用。

- [invokespecial](#)：调用非静态私有方法、构造方法(包括super)

用于[调用一些需要特殊处理的实例方法](#)，包括[实例初始化（<init>）方法、私有方法和父类方法](#)。

- [invokestatic](#)：调用静态方法

调用[静态方法（static方法）](#)。

- [invokedynamic](#)：

用于[在运行时动态解析出调用点限定符所引用的方法，并执行该方法](#)，前面4条调用指令的分派逻辑都固化在Java虚拟机内部，而[invokedynamic指令的分派逻辑是由用户所设定的引导方法决定的](#)。动态调用（Java7引入的，第一次用却是在Java8中，用在了Lambda表达式和默认方法中，它允许调用任意类中的同名方法，注意是任意类，和重载重写不同）（动态 ≠ 多态）

```
interface Student {  
    boolean isRecommend();  
}
```

```
class Edu {
    public double youhui (double originPrice, Student stu) {
        return originPrice * 0.7d;
    }
}

class Kaikeba extends Edu {
    @Override
    public double youhui (double originPrice, Student stu) {
        if (stu.isRecommend()) {
            //
invokeinterface
            return originPrice * randomYouhui ();
        }
        //
invokestatic
        return super.youhui(originPrice, stu);
    }
    //
invokespecial
    }
}
private static double randomYouhui () {
    return new Random()
        .nextDouble();
}
//
invokevirtual
}
}
```

静态绑定和动态绑定

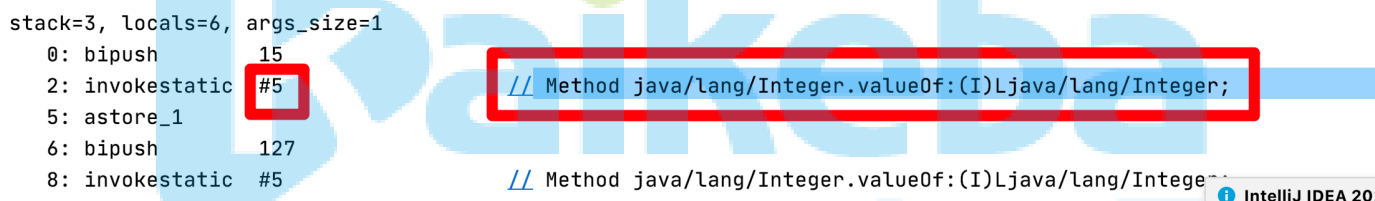
Java 虚拟机识别方法的关键在于类名、方法名以及方法描述符（method descriptor）。

前面两个就不做过多的解释了。至于方法描述符，它是由方法的参数类型以及返回类型所构成。在同一个类中，如果同时出现多个名字相同且描述符也相同的方法，那么 Java 虚拟机会在类的验证阶段报错。

JVM是如何确定要执行的方法是哪一个？

JVM识别方法的关键在于【类名+方法名+方法描述符(参数类型、返回类型)】，这其实就是方法的完整表示方式（符号引用）

```
stack=3, locals=6, args_size=1
 0: bipush      15
 2: invokestatic #5
 5: astore_1
 6: bipush      127
 8: invokestatic #5
// Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
// Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
```



方法和类关联起来，叫做方法的绑定，绑定分为静态绑定和动态绑定。

在JVM中，将符号引用替换为直接引用（目标方法的内存地址）的处理逻辑，与方法的绑定机制有关系。

Java语言具有继承和多态的特性，所以自然而然的也就具备以下两种绑定方式：静态绑定和动态绑定。

当程序中调用了某一个方法时：

- 如果在编译期（直接可以根据编译类型确定内存中的目标方法）就可以确定目标方法的话，并且在运行期间不可变，这其实就是一种静态绑定机制，也叫静态链接，也叫静态分派，也叫前期绑定。
 - 静态方法
 - 私有方法
 - final方法
 - this()
 - super()
- 如果在运行期（不能根据编译类型确定内存中的目标方法，需要根据对象类型去确定内存中的目标方法）间才能根据对象类型确定目标方法的话，这是动态绑定机制，也叫动态链接、也叫动态分派，也叫后期绑定。

静态绑定

Java虚拟机中的静态绑定(static binding)指的是在类加载时的解析阶段便能够直接识别目标方法的情况。

在Java中，final、private、static修饰的方法以及构造函数都是静态绑定的，不需程序运行，不需具体的实例对象就可以知道这个方法的具体内容。

动态绑定

而动态绑定(dynamic binding)则指的是需要在运行过程中根据调用者的动态类型来识别目标方法的情况。

```
Father ft = new Son();  
ft.say(); //say方法是调用的father类的，还是Son类的？
```

虚分配 (Virtual Dispatch)

上面提到了五种方法调用指令，其中最复杂的要属 `invokevirtual` 指令，它涉及到了多态的特性，这里面使用到了 `virtual dispatch` 做方法调用。

`virtual dispatch` 机制会首先从 `receiver`（被调用方法的对象）的类的实现中查找对应的方法，如果没找到，则去父类查找，直到找到函数并实现调用，而不是依赖于引用的类型。

下面是一段有趣的代码。反映了 `virtual dispatch` 机制和一般的 `field` 访问的不同。

```
public class Father {  
    String name = "James";  
    String method1(){  
        return "kaikeba";  
    }  
}  
  
public class Son extends Father {  
    String name = "xiao James";  
    String method1(){
```



```
        return "lao NB le";
    }

    public static void main(String[] args){
        Father f = new Father();
        Father s = new Son();

        System.out.println(f.name + "," +
f.method1()); //James,kaikeba
        System.out.println(s.name + "," +
s.method1()); //James,lao NB le
        System.out.println(((Son)f).name + "," +
((Son)f).method1()); // xiao James,lao NB le
    }
}
```

运行结果

```
James,kaikeba
James,lao NB le
xiao James,lao NB le
```

前两行输出中，对于 `name` 这个属性的访问，直接指向了父类中的变量，因为引用类型为父类。

第二行对于 `method1()` 的方法调用，则是指向了子类中的方法，虽然引用类型也为父类，但这是虚分派的结果，虚分派不管引用类型的，只查被调用对象的类型。

既然虚分派机制是从被调用对象本身的类开始查找，那么对于一个覆盖了父类中某方法的子类的对象，是无法调用父类中那个被覆盖的方法的吗？

在虚分派机制中这确实是不可以的。但却可以通过 `invokespecial` 实现。如下代码：

```
public class Son extends Father {
    String name = "xiao James";

    String method1(){
        return "lao NB le";
    }

    public String execute(){
        return super.method1();
    }

    public static void main(String[] args){
        Father f = new Father();
        Son s = new Son();

        System.out.println(s.execute());
    }
}
```

`execute()`就成功的调用了父类的方法 `method1()`，虽然 `target()`已经被子类重写了。具体的调用细节，从字节码中可以看到：

```
ALOAD 0: this
```

```
INVOKESPECIAL Father.method1() : String
```

```
ARETURN
```

其中使用了 `invokespecial` 指令，而不是施行虚分派策略的 `invokevirtual` 指令。

方法表 (Method Table)

介绍了虚分派，接下来介绍是它的一种实现方式 - 方法表^{**}。类似于 C++ 的虚函数表 `vtbl`。

在有的 JVM 实现中，使用了方法表机制实现虚分派，而有时候，为了节省内存可能不采用方法表的实现。

不要被方法表这个名字迷惑，它并不是记录所有方法的表。它是为虚分派服务，不会记录用 `invokestatic` 调用的静态方法和用 `invokespecial` 调用的构造函数和私有方法。

JVM 会在链接类的过程中，给类分配相应的方法表内存空间。每个类对应一个方法表。这些都是存在于 `method area` 区中的。这里与 C++ 略有不同，C++ 中每个对象的第一个指针就是指向了相应的虚函数表。而 Java 中每个对象索引到对应的类，在对应的类数据中对应一个方法表。

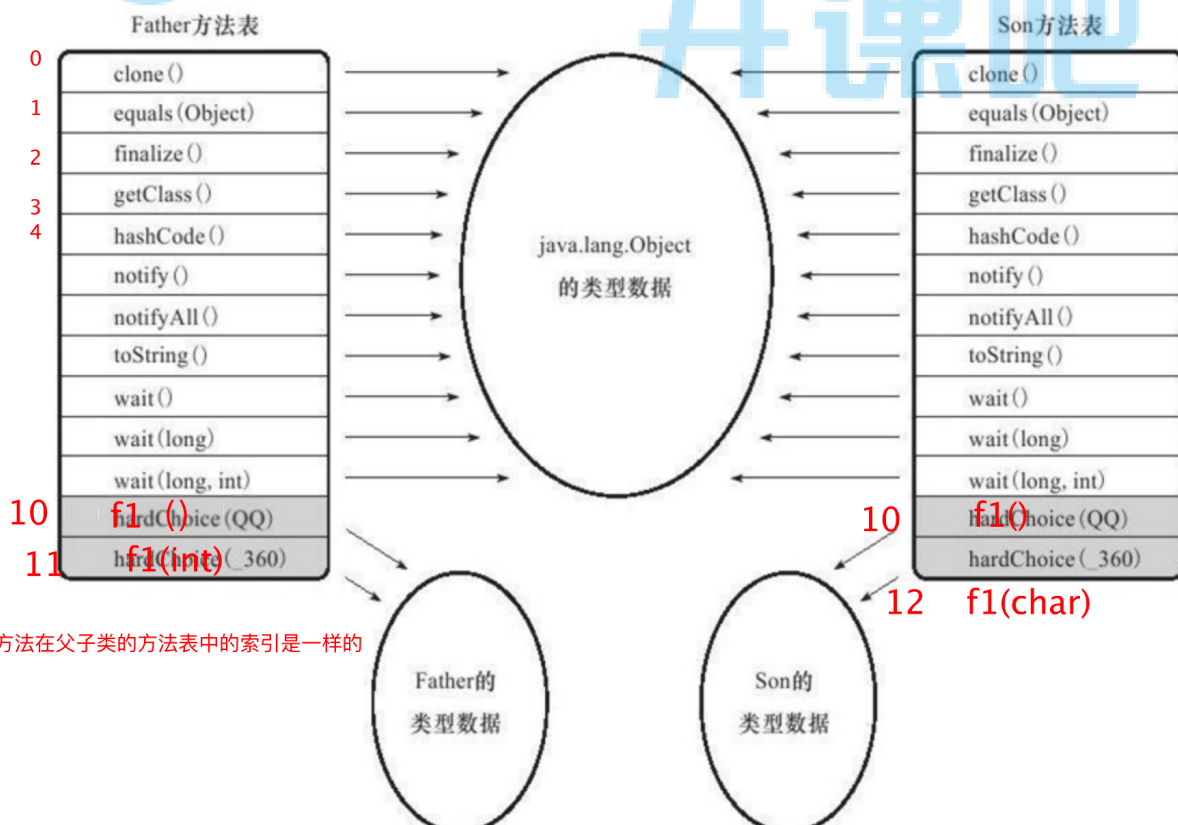
一种虚方法表的实现

父类的方法比子类的方法先得到解析，即父类的方法相比子类的方法位于表的前列。

表中每项对应于一个方法，索引到实际方法的实现代码上。如果子类重写了父类中某个方法的代码，则该方法第一次出现的位置的索引更换到子类的实现代码上，而不会在方法表中出现新的项。

JVM 运行时，当代码索引到一个方法时，是根据它在方法表中的偏移量来实现访问的。（第一次执行到调用指令时，会执行解析，将符号索引替换为对应的直接索引）。

由于 `invokevirtual` 调用的方法在对应的类的方法表中都有固定的位置，直接索引的值可以用偏移量来表示。（符号索引解析的最终目的是完成直接索引：对象方法和对象变量的调用都是用偏移量来表示直接索引的）



invokeinterface 与 invokevirtual 的比较

当使用 invokeinterface 来调用方法时，由于不同的类可以实现同一 interface，我们无法确定在某个类中的 interface 中的方法处在哪个位置。于是，也就无法解析 CONSTANT_intefaceMethodref-info 为直接索引，而必须每次都执行一次在 methodtable 中的搜索了。所以，在这种实现中，通过 invokeinterface 访问方法比通过 invokevirtual 访问明显慢很多。

itable：接口方法表

vtable：虚拟方法表

方法内联

前面课程讲过

内联缓存

内联缓存是一种加快动态绑定的优化技术。

它能够缓存虚方法调用中调用者的动态类型，以及该类型所对应的目标方法。在之后的执行过程中，如果碰到已缓存的类型，内联缓存便会直接调用该类型所对应的目标方法。如果没有碰到已缓存的类型，内联缓存则会退化至使用基于方法表的动态绑定。

在针对多态的优化手段中，我们通常会提及以下三个术语。

- 单态 (monomorphic) 指的是仅有一种状态的情况。
- 多态 (polymorphic) 指的是有限数量种状态的情况。二态 (bimorphic) 是多态的其中一种。
- 超多态 (megamorphic) 指的是更多种状态的情况。通常我们用一个具体数值来区分多态和超多态。在这个数值之下，我们称之为多态。否则，我们称之为超多态。

对于内联缓存来说，我们也有对应的单态内联缓存、多态内联缓存和超多态内联缓存。单态内联缓存，顾名思义，便是只缓存了一种动态类型以及它所对应的目标方法。它的实现非常简单：比较所缓存的动态类型，如果命中，则直接调用对应的目标方法。多态内联缓存则缓存了多个动态类型及其目标方法。它需要逐个将所缓存的动态类型与当前动态类型进行比较，如果命中，则调用对应的目标方法。一般来说，我们会将更加热门的动态类型放在前面。在实践中，大部分的虚方法调用均是单态的，也就是只有一种动态类型。为了节省内存空间，Java 虚拟机只采用单态内联缓存。

前面提到，当内联缓存没有命中的情况下，Java 虚拟机需要重新使用方法表进行动态绑定。对于内联缓存中的内容，我们有两种选择。一是替换单态内联缓存中的纪录。这种做法就好比 CPU 中的数据缓存，它对数据的局部性有要求，即在替换内联缓存之后的一段时间内，方法调用的调用者的动态类型应当保持一致，从而能够有效地利用内联缓存。

因此，在最坏情况下，我们用两种不同类型的调用者，轮流执行该方法调用，那么每次进行方法调用都将替换内联缓存。也就是说，只有写缓存的额外开销，而没有用缓存的性能提升。

另外一种选择则是劣化为超多态状态。这也是Java虚拟机的具体实现方式。处于这种状态下的内联缓存，实际上放弃了优化的机会。它将直接访问方法表，来动态绑定目标方法。与替换内联缓存纪录的做法相比，它牺牲了优化的机会，但是节省了写缓存的额外开销。

为了优化[动态绑定](#)的效率，可以使用内联缓存进行优化：

利用缓存去存储动态类型和方法之间的对应关系。

key：调用者的动态类型(new Son)

value：方法对象

查找方式：

1. 先找根据调用者的类型去内联缓存中查找方法对象
2. 如果没有找到，再去方法区中查找调用者类型对应的方法表，再去方法表中找对应的方法对象。

案例分析

重载方法的查找过程演示

重载方法的查找过程是发生在编译过程中的。重载方法又叫编译时多态。

- 根据编译类型去查找方法名称和方法描述符（参数类型和返回值）
- 如果没有找到，则查找是否有凑合的方法（可以自动类型转换的参数）

- 实在找不到，则报错

方法重载：在同一个类中，方法名称一致，方法参数类型和顺序不一致，不关心方法返回值。也叫编译时多态，选择调用目标方法时，重点就是考虑编译类型。

方法重写：也叫方法覆盖，指的是在[继承关系](#)或者[实现关系](#)下，子类方法和父类方法的描述符（参数和返回值）一致，方法名称一致。

在Java程序里，如果同一个类中出现多个名字相同，并且参数类型相同的方法，那么它无法通过编译。

也就是说，在正常情况下，如果我们想要在同一类中定义名字相同的方法，那么它们的参数类型必须不同。这些方法之间的关系，我们称之为**重载**。

重载的方法在编译过程中即可完成识别。具体到每一个方法调用，Java编译器会根据所传入参数的声明类型（注意与实际类型区分）来选取重载方法。选取的过程共分为三个阶段：

1. 在不考虑对基本类型自动装箱（auto-boxing，auto-unboxing），以及可变长参数的情况下选取重载方法；

2. 如果在第 1 个阶段中没有找到适配的方法，那么在允许自动装拆箱，但不允许可变长参数的情况下选取重载方法；
3. 如果在第 2 个阶段中没有找到适配的方法，那么在允许自动装拆箱以及可变长参数的情况下选取重载方法。

如果 Java 编译器在同一个阶段中找到了多个适配的方法，那么它会在其中选择一个最为贴切的，而决定贴切程度的一个关键就是形式参数类型的继承关系。



```
static abstract class Human{}
static class Man extends Human{ }
static class Woman extends Human{}
```

发生了方法的重载，方法重载会在编译期确定具体调用哪个方法，这个也相当于是一种静态分派

```
public void sayHello(Human guy){
    System.out.println("hello,人类!");//1
}
public void sayHello(Man guy){
    System.out.println("hello,老铁!");//2
}
public void sayHello(Woman guy){
    System.out.println("hello,老妹!");//3
}
```

```
public static void main(String[] args){
```

```
Human h1 = new Man();
Human h2 = new Woman();
```

编译看左边
运行看右边

```
StaticCall02 sd = new StaticCall02();
sd.sayHello(h1);
sd.sayHello(h2);
```

编译时，确定的h1类型是左边的类型，也就是Human类型

运行时多态查找虚方法的过程

```
package com.kkb.test;
```

//调用方法

```
public class AutoCall {
    public static void main(String[] args) {
        Father father = new Son();
        // 多态
        father.f1();
        // 打印结果: Son-f1()
```

```
    char c = 'a';
    father.f1(c);
}

// 被调用的父类
class Father {
    public void f1() {
        System.out.println("father-f1()");
    }

    public void f1(int i) {
        System.out.println("father-f1() para-int " + i);
    }
}

// 被调用的子类
class Son extends Father {
    public void f1() {
        // 覆盖父类的方法
        System.out.println("Son-f1()");
    }

    public void f1(char c) {
        System.out.println("Son-s1() para-char " + c);
    }
}
```

```

// 被调用的父类
class Father {
    public void f1() {
        System.out.println("father-f1()");
    }

    public void f1(int i) {
        System.out.println("father-f1() para-int " + i);
    }
}

// 被调用的子类
class Son extends Father {
    public void f1() {
        // 覆盖父类的方法
        System.out.println("Son-f1()");
    }

    public void f1(char c) {
        System.out.println("Son-f1() para-char " + c);
    }
}

```

```

PS D:\05-workspace\vip-class\jvm\bin\com\kkb\test> javap -
v .\AutoCall
警告: 二进制文件.\AutoCall包含com.kkb.test.AutoCall
Classfile /D:/05-workspace/vip-
class/jvm/bin/com/kkb/test/AutoCall.class
    Last modified 2020-4-11; size 553 bytes
    MD5 checksum bb066cb8ac1c23ea2281bae481f39419
    Compiled from "AutoCall.java"
public class com.kkb.test.AutoCall
    minor version: 0
    major version: 52
    flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
    #1 = Class                #2          //
com/kkb/test/AutoCall

```

```

    #2 = Utf8                               com/kkb/test/AutoCall
    #3 = Class                               #4                                //
java/lang/Object
    #4 = Utf8                               java/lang/Object
    #5 = Utf8                               <init>
    #6 = Utf8                               ()V
    #7 = Utf8                               Code
    #8 = Methodref                          #3.#9                                //
java/lang/Object."<init>":()V
    #9 = NameAndType                        #5:#6                                // "<init>":()V
    #10 = Utf8                              LineNumberTable
    #11 = Utf8                              LocalVariableTable
    #12 = Utf8                              this
    #13 = Utf8                              Lcom/kkb/test/AutoCall;
    #14 = Utf8                              main
    #15 = Utf8                              ([Ljava/lang/String;)V
    #16 = Class                             #17                                //
com/kkb/test/Son
    #17 = Utf8                              com/kkb/test/Son
    #18 = Methodref                          #16.#9                                //
com/kkb/test/Son."<init>":()V
    #19 = Methodref                          #20.#22                                //
com/kkb/test/Father.f1:()V
    #20 = Class                             #21                                //
com/kkb/test/Father
    #21 = Utf8                              com/kkb/test/Father
    #22 = NameAndType                        #23:#6                                // f1:()V
    #23 = Utf8                              f1
    #24 = Utf8                              args
    #25 = Utf8                              [Ljava/lang/String;
    #26 = Utf8                              father
    #27 = Utf8                              Lcom/kkb/test/Father;
    #28 = Utf8                              MethodParameters

```

```

#29 = Utf8                               SourceFile
#30 = Utf8                               AutoCall.java
{
  public com.kkb.test.AutoCall();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
      0: aload_0
      1: invokespecial #8                      // Method
java/lang/Object."<init>":()V
      4: return
   LineNumberTable:
      line 4: 0
    LocalVariableTable:
      Start   Length  Slot  Name   Signature
      0        5      0    this
Lcom/kkb/test/AutoCall;

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=1
      0: new                               #16          // class
com/kkb/test/Son
      3: dup
      4: invokespecial #18          // Method
com/kkb/test/Son."<init>":()V
      7: astore_1
      8: aload_1
      9: invokevirtual #19          // Method
com/kkb/test/Father.f1:()V

```

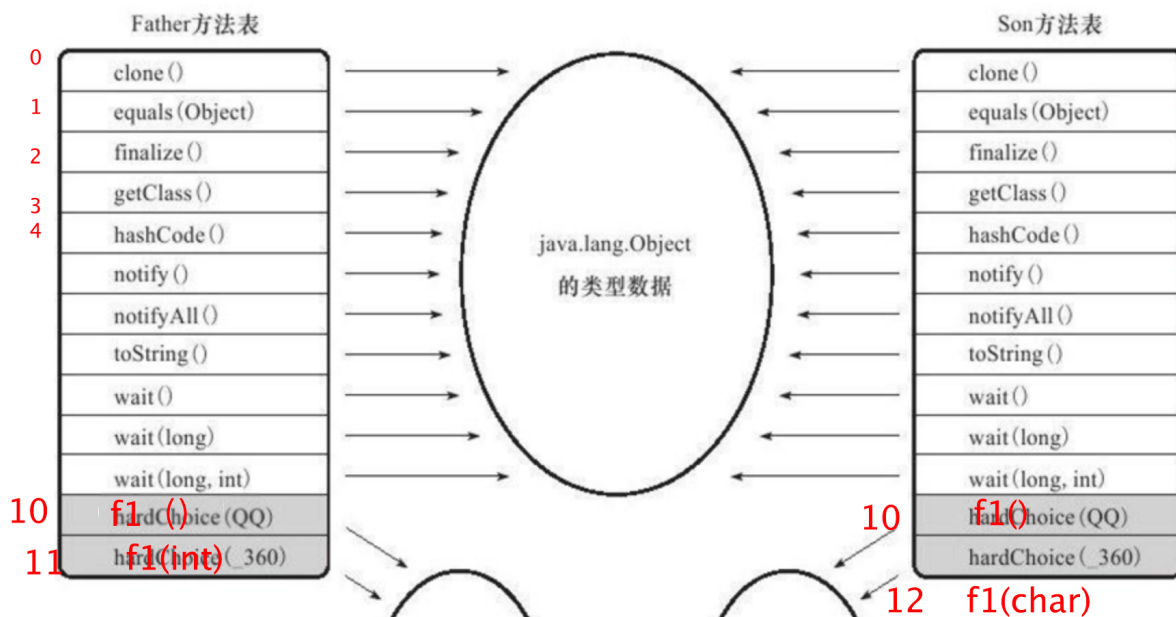
```

    12: return
LineNumberTable:
  line 6: 0
  line 8: 8
  line 10: 12
LocalVariableTable:
  Start    Length  Slot  Name   Signature
      0         13     0  args   [Ljava/lang/String;
      8          5     1 father  Lcom/kkb/test/Father;
MethodParameters:
  Name                      Flags
  args
}
SourceFile: "AutoCall.java"

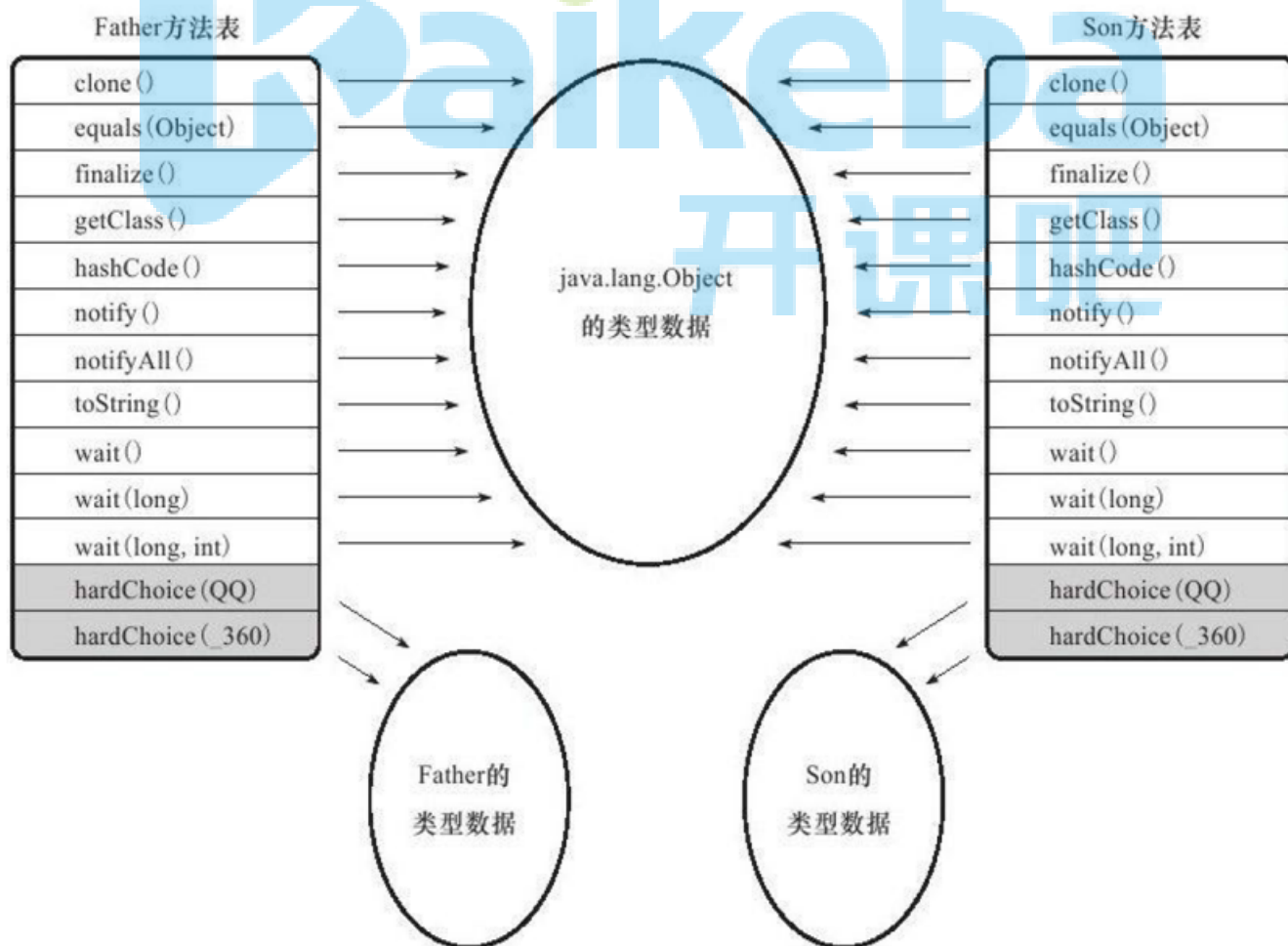
```

上面的源代码中有三个重要的概念：[多态\(polymorphism\)](#)、[方法覆盖](#)、[方法重载](#)。打印的结果大家也都比较清楚，但是JVM是如何知道father.f1()调用的是子类Son中方法而不是Father中的方法呢？在解释这个问题之前，我们首先简单的讲下JVM管理的一个非常重要的数据结构——[虚方法表 \(virtual method table\)](#)。

在JVM加载类的同时，会在方法区中为这个类存放很多信息。其中就有一个数据结构叫虚方法表。它以数组的形式记录了当前类及其所有超类的可见方法字节码在内存中的直接地址。



同一个方法在父子类的方法表中的索引是一样的



虚方法表有两个特点：

(1) 子类方法表中继承了父类的方法，比如Father extends Object。

(2) 相同的方法(相同的方法签名：方法名和参数列表)在所有类的方法表中的索引相同。比如Father方法表中的f1()和Son方法表中的f1()都位于各自方法表的第11项中。

对于上面的源代码，编译器首先会把main方法编译成下面的字节码指令：

```
0: new                #16                // class
com/kkb/test/Son
3: dup
4: invokespecial      #18                // Method
com/kkb/test/Son."<init>":()V
7: astore_1
8: aload_1
9: invokevirtual      #19                // Method
com/kkb/test/Father.f1:()V
12: return
```

其中invokevirtual指令的详细调用过程是这样的：

(1) invokevirtual指令中的[#19](#)指的是AutoCall类的常量池中第19个常量表的索引项。这个常量表(CONSTATN_Methodref_info) 记录的是方法f1信息的符号引用(包括f1所在的类名，方法名和返回类型)。JVM会首先根据这个符号引用找到调用方法f1的类的全限定名: hr.test.Father。这是因为调用方法f1的类的对象father声明为Father类型。

(2) 在Father类型的[虚方法表](#)中查找方法f1，如果找到，则将方法f1在方法表中的索引项11(如上图)记录到AutoCall类的常量池中第15个常量表中(常量池解析)。这里有一点要注意：[如果Father类型方法表中没有方法f1](#)，那么即使Son类型中方法表有，编译的时候也通过不了。因为调

用方法f1的类的对象father的声明为Father类型。

(3) 在调用invokevirtual指令前有一个aload_1指令，它会将开始创建在堆中的Son对象的引用压入操作数栈。然后invokevirtual指令会根据这个Son对象的引用首先找到堆中的Son对象，然后进一步找到Son对象所属类型的方法表。

(4) 这时通过第(2)步中解析完成的#15常量表中的方法表的索引项11，可以定位到Son类型方法表中的方法f1()，然后通过直接地址找到该方法字节码所在的内存空间。

很明显，根据对象(father)的声明类型(Father)还不能够确定调用方法f1的位置，必须根据father在堆中实际创建的对象类型Son来确定f1方法所在的位置。这种在程序运行过程中，通过动态创建的对象的方法表来定位方法的方式，我们叫做 [动态绑定机制](#)。

扩展题

上面的过程很清楚的反映出在方法覆盖的多态调用的情况下，JVM是如何定位到准确的方法的。但是下面的调用方法JVM是如何定位的呢?(仍然使用上面代码中的Father和Son类型)

```
public class AutoCall{
    public static void main(String[] args){
        Father father=new Son();
        char c='a';
        father.f1(c);
    }
}
```

// 被调用的父类

```
class Father {  
    public void f1() {  
        System.out.println("father-f1()");  
    }  
  
    public void f1(int i) {  
        System.out.println("father-f1() para-int " + i);  
    }  
}
```

// 被调用的子类

```
class Son extends Father {  
    public void f1() {  
        // 覆盖父类的方法  
        System.out.println("Son-f1()");  
    }  
  
    public void f1(char c) {  
        System.out.println("Son-s1() para-char " + c);  
    }  
}
```

//打印结果： father-f1() para-int 97

问题是Father类型中并没有方法签名为f1(char)的方法呀。但打印结果显示JVM调用了Father类型中的f1(int)方法，并没有调用到Son类型中的f1(char)方法。

根据上面详细阐述的调用过程，首先可以明确的是：JVM首先是根据对象father声明的类型Father来解析常量池的(也就是用Father方法表中的索引项来代替常量池中的符号引用)。如果Father中没有匹配到“合适”的方法，就无法进行常量池解析，这在编译阶段就通过不了。

那么什么叫“合适”的方法呢？当然，方法签名完全一样的方法自然是合适的。但是如果方法中的参数类型在声明的类型中并不能找到呢？比如上面的代码中调用father.f1(char)，Father类型并没有f1(char)的方法签名。实际上，JVM会找到一种“凑合”的办法，就是通过 [参数的自动转型](#) 来找到“合适”的方法。比如char可以通过自动转型成int，那么Father类中就可以匹配到这个方法了。但是还有一个问题，如果通过自动转型发现可以“凑合”出两个方法的话怎么办？比如下面的代码：

```
class Father{
    public void f1(Object o){
        System.out.println("Object");
    }
    public void f1(double[] d){
        System.out.println("double[]");
    }
}

public class Demo{
    public static void main(String[] args) {
        new Father().f1(null);
    }
}
```

//打印结果： double[]

null可以引用于任何的引用类型，那么JVM如何确定“合适”的方法呢。一个很重要的标准就是：

如果一个方法可以接受传递给另一个方法的任何参数，那么第一个方法就相对不合适。

比如上面的代码：任何传递给f1(double[])方法的参数都可以传递给f1(Object)方法，而反之却不行，那么f1(double[])方法就更合适。因此JVM就会调用这个更合适的方法。

总结

- (1) 所有私有方法、静态方法、构造器及初始化方法都是采用静态绑定机制。在编译器阶段就已经指明了调用方法在常量池中的符号引用，JVM运行的时候只需要进行一次常量池解析即可。
- (2) 类对象方法的调用必须在运行过程中采用动态绑定机制（先找内联缓存[JIT优化项]、通过方法表去查找）。

编译时如果遇到了重载方法（编译时多态），则按照以下步骤进行处理（找到最合适的方法）：

- ① 如果能在[声明类型](#)中匹配到方法签名完全一样(参数类型一致)的方法，那么这个方法是最合适的。
- ② 在第①条不能满足的情况下，寻找可以“凑合”的方法。标准就是通过将[参数类型进行自动转型之后再行匹配](#)。

如果匹配到多个自动转型后的方法签名f(A)和f(B)，则用下面的标准来确定合适的方法：传递给f(A)方法的参数都可以传递给f(B)，则f(A)最合适。反之f(B)最合适。

③ 如果仍然在声明类型中找不到“合适”的方法，则编译阶段就无法通过。

然后，根据在堆中创建对象的实际类型找到对应的方法表，从中确定具体的方法在内存中的位置。

```
Father f = new Son();  
  
f.sayHello();
```

运行时多态的查找路径（虚方法）：

方法有栈帧，栈帧有局部变量表，变量表第一个变量就是调用该方法的对象引用，也就是this引用。

找到this引用，是否就可以找到堆中的对象。

找到对象，是否就可以从对象的对象头中找到类型指针。

找到类型指针，是否就可以去方法区中找到Class类。

找到Class类，是否就可以找到每个类都拥有的方法表。

根据方法的名称和描述符，是否就可以从方法表中定位到指定的方法，获取该方法的内存地址。