目录

[1.虚拟机内存结构 1](#_Toc496045461)

[2.对象在内存中的布局 3](#_Toc496045462)

[3.判断对象是否死亡 3](#_Toc496045463)

[4.引用的4中情况 4](#_Toc496045464)

[5.垃圾收集算法 4](#_Toc496045465)

[6.HotSpot虚拟机算法实现 5](#_Toc496045466)

[7.如何在GC发生时让所有线程都要附近的安全点停下 6](#_Toc496045467)

[8.垃圾收集器 7](#_Toc496045468)

[9.GC日志 9](#_Toc496045469)

[10.内存分配 9](#_Toc496045470)

[11.Class类文件的结构 10](#_Toc496045471)

[12.类的生命周期 12](#_Toc496045472)

[13.类加载器 14](#_Toc496045473)

[14.运行时栈帧的结构 16](#_Toc496045474)

[15. 方法调用 18](#_Toc496045475)

[16. 分派 18](#_Toc496045476)

[17.虚方法表 18](#_Toc496045477)

[18.Java内存模型（JMM） 19](#_Toc496045478)

[19.内存间的交互 19](#_Toc496045479)

[20.volatile变量 20](#_Toc496045480)

[21.原子性 21](#_Toc496045481)

[22.可见性 21](#_Toc496045482)

[23.有序性 21](#_Toc496045483)

[24.先行发生原则 21](#_Toc496045484)

[25.Java线程调度 22](#_Toc496045485)

[26.线程的状态 23](#_Toc496045486)

[27.线程安全 24](#_Toc496045487)

[28.线程安全的实现方法 25](#_Toc496045488)

[29.锁优化 26](#_Toc496045489)

[30.编译优化技术 28](#_Toc496045490)

### 1.虚拟机内存结构

线程私有：虚拟机栈，本地方法栈，程序计数器

线程共享：堆，方法区（包括运行时常量池）

##### 1.1程序计数器

当前程序锁执行的字节码行号指示器，记录下一条需要执行的指令。

##### 1.2虚拟机栈

生命周期与线程相同，每个方法在执行时都会创建一个栈帧。

方法执行的过程，就是栈帧入栈到出栈的过程。

栈帧用于存放局部变量表，操作数栈，动态链接，方法出口等信息。

局部变量表存放了编译期可知的基本数据类型和对象引用。

##### 1.3 本地方法栈

为虚拟机使用到的Native方法服务。

目前HotSpot虚拟机将本地方法栈和虚拟机栈合二为一。

##### 1.4 堆

存放对象实例，所有线程共享。

##### 1.5 方法区（永久代）

存放被虚拟机加载的类信息，常量，静态变量，即时编译器编译后的代码等。

##### 1.6 运行时常量池

方法区的一部分，用于存放编译期生成的各种字面量和符号引用。

##### 1.7 字面量，符号引用，直接引用

字面量：

通俗解释就是一个变量的值，但是这个值不能超过范围。

int a = 1; 1是a的字面量

213738648则不能是int的字面量，因为超出了int的范围。

符号引用：

以一组符号来描述引用的目标，符号可以是任何形式的字面量，只要使用时能够无歧义的定位到目标即可。

直接引用：

直接指向目标的指针（类变量，类方法）。

相对偏移量（实例变量，实例方法）。

一个能间接定位到目标的句柄。

### 2.对象在内存中的布局

对象头（哈希码，GC分代年龄，数据指针，如果是数组还会有数组长度），实例数据，对齐填充

### 3.判断对象是否死亡

##### 3.1引用计数算法

每个对象添加一个计数器，引用它加1，引用失效减1，为0则死亡，很难解决循环引用问题。

##### 3.2可达性分析算法

从gc root节点开始向下搜索，不可达则已死。

可作为gc root节点的情况：

3.2.1：虚拟机栈本地变量表引用的对象

3.2.2：方法区中类静态属性引用的对象

3.2.3：方法区中常量引用的对象

3.2.4：本地方法栈native方法引用的对象

### 4.引用的4中情况

强引用，软引用，弱引用，虚引用（幽灵引用）

4.1强引用：new 关键字，强引用还在，则不会被回收

4.2软引用：发生内存溢出钱，会把这些软引用对象列入回收范围

进行第二次回收时会将他们回收

4.3：弱引用：只能存活到下一次垃圾回收之前

4.4：虚引用：与对象的生存时间不发生关系，作用是在这个对象被回收的时候，收到一个系统通知

### 5.垃圾收集算法

##### 5.1：标记—清除 Mark-Sweep

过程：标记可回收对象，进行清除

缺点：标记和清除效率低，清除后会产生内存碎片

##### 5.2：复制算法

过程：将内存划分为相等的两块，将存活的对象复制到另一块内存，把已经使用的内存清理掉

缺点：使用的内存变为了原来的一半

进化：将一块内存按8:1的比例分为一块Eden区（80%）和两块Survivor区（10%）

每次使用Eden和一块Survivor，回收时，将存活的对象一次性复制到另一块Survivor上，如果另一块Survivor空间不足，则使用分配担保机制存入老年代

##### 5.3：标记—整理 Mark—Compact

过程：所有存活的对象向一端移动，然后清除掉边界以外的内存

##### 5.4：分代收集算法

过程：将堆分为新生代和老年代，根据区域特点选用不同的收集算法，如果新生代朝生夕死，则采用复制算法，老年代采用标记清除，或标记整理

### 6.HotSpot虚拟机算法实现

##### 6.1枚举根节点

回收时如果逐个检查引用（可达性分析）效率低下，通过OopMap数据结构来得知哪些地方存放着引用

##### 6.2安全点

不会为所有指令都生成OopMap，只会在特定位置生成，这些位置成为安全点。

方法调用，循环跳转，异常跳转等会产生安全点

### 7.如何在GC发生时让所有线程都要附近的安全点停下

##### 7.1抢先式中断

中断全部线程，如果发现有线程不在安全点上， 那么恢复线程，让它跑到安全点上（几乎不使用）

##### 7.2主动式中断

设置一个标志，线程执行时去轮询这个标志，标志为true则线程挂起。标志和安全点是重合的

##### 7.3安全区域

一段代码片段，在这个区域中的任意地方开始GC都是安全的。为了解决处于Sleep或Blocked线程达到安全点的问题。

过程：如果进入到了安全区域，那么标识自己已经进入，GC时不用管已经标识过的。如果离开，则检查是否完成了节点枚举或者整个GC，如果未完成，则必须等待离开信号。

### 8.垃圾收集器

##### 8.1 Serial

单线程新生代收集器，只会用一条线程完成收集工作

在Client模式下的虚拟机可以选择

新生代：复制算法

老年代：标记—整理

##### 8.2 ParNew

Serial的多线程版本，收集过程以及算法与Serival一致，可与CMS老年代收集器配合

##### 8.3 Parallel Scavenge

新生代收集器，多线程，主要关注吞吐量

适合在后台运算不需要太多交互的任务

采用复制算法

##### 8.4 Serial Old

Serial 的老年代版本，采用 标记—整理

##### 8.5 Parallel Old

Parallel Scavenge的老年代版本，采用多线程标记—整理

##### 8.6 CMS

老年代收集器，关注停顿时间，采用标记—清除

4个过程，初始标记，并发标记，重新标记，并发清除

收集过程与用户线程并发执行

缺点：并发收集占用CPU资源，降低吞吐率。

浮动垃圾，程序不断运行会产生新的垃圾。

JDK1.5老年代占用达到68%会触发，JDK1.6老年代占用达到92%会触发，触发的阈值可以设置，设置不当会导致FuLL GC 降低性能。

会出现内存碎片，可设置参数，多少次不压缩的Full GC之后，进行一次压缩的，默认0，每次都会压缩。

##### 8.7 G1

特点：并行与并发，分代手机，空间整合，可预测的停顿

将Java堆分为多个大小相等的独立区域，跟踪每个区域里垃圾的价值，维护一个优先列表，优先回收价值最大的。

区域空间对象的引用使用Remembered Set记录，如果引用的对象处于不同的区域，通过Card Table把引用信息记录到被引用对象的Remembered Set中。

过程：初始标记，并发标记，最终标记，筛选回收

### 9.GC日志

[DefNew:3324k -> 1527k (3712k) ,0.025secs] 3324k🡪152k(11904k) 0.03 secs

DefNew表示GC发生的区域，区域名字与使用的收集器有关

中括号内部 3324k🡪1527k(3712k)

GC前该区域已使用—>GC后已使用（该区域总容量）

中括号外表示堆信息

secs表示GC花费时间

### 10.内存分配

10.1优先在Eden分配，如果Eden空间不足，会发生一次MinorGC（新生代GC），GC时如果所剩空间不足以存放新对象，Survivor空间又无法存入原有存活对象，那么会将原有对象移入老年代，新对象分配在Eden区

10.2 Serival和ParNew收集器可以通过设置来保证打对象直接进入老年代

10.3 长期存活的对象进入老年代，MinorGC时如果有存活对象能被Survivor容纳，那么年龄为1，每熬过一次MinorGC年龄加1，默认15岁会晋升到老年代

-XX：MaxTenuringThreshold 设置晋升年龄

10.4 如果Survivor空间中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或者等于该年龄的对象就可以直接进入老年代。

10.5 空间分配担保，老年代的连续空间大于新生代的对象总大小，或者两次晋升的平均大小，就会进行MinorGC，否则将进行FULLGC。

### 11.Class类文件的结构

8位字节为基础单位的二进制流文件

8位字节以上的数据项，按照高位在前的方式存储

主要组成部分：无符号数和表

##### 11.1 无符号数

u1,u2,u4,u8 代表1字节，2字节，4字节和8字节

可以描述数字，索引引用，数量值，按照UTF-8构成的字符串值。

##### 11.2 表

由多个无符号数或者其他表构成的复合数据类型，

以\_info结尾

##### 11.3 Class文件内容顺序

1-4字节，魔数，0xCAFEBABE

5-6字节，次版本号

7-8字节，主版本号

常量池

访问标志

类索引

父类索引

接口索引

字段表集合

方法表集合

属性表集合

常量池：

Class文件中第一个表类型的数据，0x0016 =22 代表21项常量，索引为1-21，第0项表示“不引用任何一个常量池项目”

池中两大类常量：字面量和符号引用

字面量：文本字符串，声明为final的常量。

符号引用：

1.类和接口的全限定名

2.字段名称和描述符

3.方法名称和描述符

访问标志：

两个字节，类或者接口的访问信息，包括是类还是接口，是否是public,是否是abstract，是否是final（类）

类索引：

类的全限定名，u2

父类索引：

父类的全限定名，单根继承，所以只有一个，除了Object之外，其他父类索引都不为0，u2

接口索引：

u2类型集合，实现了哪些接口，顺序是implements关键字后从左至右

字段表集合：

类级变量（static）以及实例变量，不包括局部变量

方法表集合：

类中的方法

属性表集合：

描述某些专有信息，如方法的代码

### 12.类的生命周期

加载，验证，准备，解析，初始化，使用，卸载

验证，准备，解析统称为连接

##### 12.1 五种初始化情况

12.1.1：使用new，读取或者设置一个类的静态字段，调用一个类的静态方法。被final修饰，编译器把结果放入常量池的静态字段除外。

12.1.2：对类进行反射调用的时候。

12.1.3：初始化一个类的时候，如果父类还没有初始化，先初始化父类。

12.1.4：初始化main方法的类。

12.1.5：java.lang.invoke.MethodHandle实例，解析结果为REF\_getStatic,REF\_putStatic,REF\_invokeStatic的方法句柄

##### 12.2 加载

加载过程要完成3件事

12.2.1：获取此类的二进制字节流

12.2.2：将字节流转化为方法区的运行时数据结构

12.2.3：在内存中生成一个对应的Class对象，作为方法区各种数据的访问入口

##### 12.3 验证

确保字节流符合要求，并且不会危害虚拟机安全，4个阶段

文件格式验证，元数据验证，字节码验证，符号引用验证

##### 12.4 准备

为类变量（static）分配内存并设置初始值

##### 12.5 解析

将常量池内的符号引用替换为直接引用

##### 12.6初始化

执行类构造器的<clinit>方法的过程

clinit方法：由类中的所有类变量（static）的赋值动作和静态语句块合并产生。

静态语句块可为语句块之后的变量赋值，但不能访问。

虚拟机保证在子类的clinit方法执行之前，父类的clinit方法已经执行完毕，所以第一个被执行的clinit方法肯定是java.lang.Object。

父类的静态语句块优先于子类的变量赋值操作。

没有静态语句块也没有变量赋值操作，则不会生成clinit方法。

执行接口的clinit方法不需要执行父接口的clinit方法。

只有一个线程会执行到类的clinit方法。

### 13.类加载器

即使两个类来源于同一个class文件，被同一个虚拟机加载，只要他们的类加载器不同，那这两个类必定不相等。

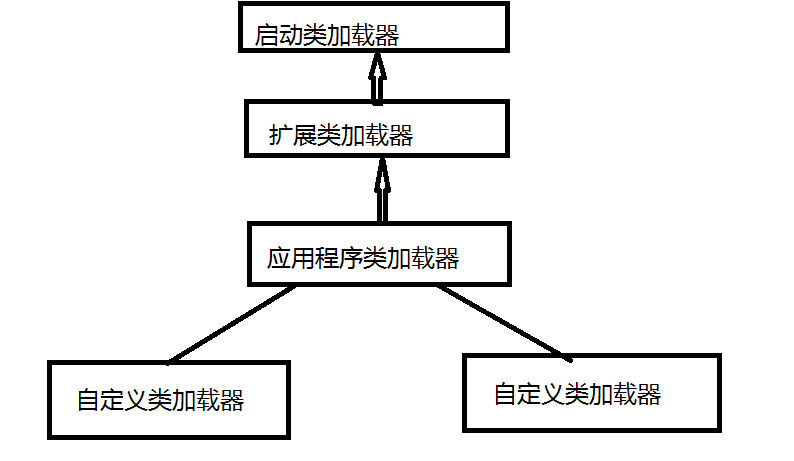
##### 13.1 三种系统提供的类加载器

启动类加载器：加载存放在JAVA\_HOME\lib目录中的，仅按照文件名识别，如rt.jar。

扩展类加载器：加载JAVA\_HOME\lib\ext目录中的。

应用程序类加载器：加载用户类路径（classpath）上指定的类库，如果没有自定义的加载器，则这个为默认。

##### 13.2 双亲委派模型



图为双亲委派模型，除了顶层的启动类加载器外，其余的加载器都应当有自己的父类加载器，父子关系使用组合关系，复用父加载器的代码。

工作过程：如果要加载一个类，把这个请求委派给父类加载器，每一个都如此，所以最终都是由顶层启动类加载器加载，如果无法加载，则由自己去加载。

如Object类，所有加载器都要加载，会委派给顶层去加载，所以Object类在环境中都是同一个。

### 14.运行时栈帧的结构

每一个方法从调用开始至执行完成都一应着一个栈帧在虚拟机栈中入栈到出站的过程。

栈帧包括：局部变量表，操作数栈，动态连接，方法返回地址和附加信息。

位于栈顶的栈帧才是有效的，与这个栈帧相关联的方法称为当前方法。

##### 14.1局部变量表

用于存放方法参数，方法内部的局部变量。以Slot为单位，一个Slot可以存放32位以内的数据类型，long和double用两个Slot来存储。

变量表索引从0开始，如果执行的是非static方法，第0位索引表示方法所属对象的实例引用，即this关键字，从1开始按参数顺序占用Slot，在根据方法体内部定义的变量顺序分配其余的Slot。

##### 14.2操作数栈

后入先出的栈

方法刚开始执行的时候，这个操作数栈是空的，执行过程中会发生出栈，入栈的操作。

##### 14.3 动态连接

每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，一部分符号引用会在运行期间转换为直接引用，成为动态连接。

##### 14.4 方法的返回地址

两种方式退出方法

14.4.1遇到方法返回的字节码指令，为正常出口

14.4.2 遇到了异常，并且在异常表中没有匹配的异常处理器（异常未被处理），称为异常出口，不会产生任何返回值。

方法正常退出时，调用者的PC计数器的值作为返回地址

异常退出时，通过异常处理器表来确定返回地址

退出方法执行的操作：恢复上层方法的局部变量表和操作数栈，把返回值（如果有）压入调用者的操作数栈，调整PC计数器的值，指向后一条指令（字节码）

### 15. 方法调用

唯一任务，确定要调用哪一个方法。

解析：在类加载的解析阶段，会将一部分符号引用转化为直接引用。条件为：编译器可知，运行期不可变。

例如：静态方法，私有方法，实例构造器，父类方法，final方法。

### 16. 分派

##### 16.1 静态分派

Human man = new Man();

Human 称为变量的静态类型，Man为变量的实际类型。

静态分派的典型应用是方法重载，重载是根据参数的静态类型作为判定依据。

##### 16.2 动态分派

主要体现在重写上，根据变量的实际类型来确定。

静态分派属于多分派类型，动态分派属于单分派类型。

### 17.虚方法表

在方法区中建立一个虚方法表，在连接阶段进行初始化，用于存放方法的实际入口。如果方法的在子类没有被重写，那么地址入口与父类一致，如果重写了，则指向子类实现的入口地址。

### 18.Java内存模型（JMM）

定义程序中各个变量的访问规则，实例字段，静态字段，构成数组对象的元素。

不包括局部变量和方法参数，因为是线程私有。

所有变量存储在主内存，对应Java堆中的实例数据部分，每条线程有自己的工作内存，对应着虚拟机栈中的部分区域。

工作内存中保存了该线程使用到的变量（主内存的副本拷贝），对变量的所有操作都在工作内存中进行，工作内存中变量只对该线程可见，线程间变量值的传递通过主内存完成。

### 19.内存间的交互

8种操作

lock：将主内存变量标识为一条线程占用，即上锁。

unlock：将主内存lock状态的变量释放，其他线程可锁定，即

解锁。

read：将主内存变量的值传送到工作内存中，为load准备。

load：将read获取到的值放入工作内存的变量副本。

use：将工作内存中的变量值传递给执行引擎。

assign：将执行引擎接收到的值赋值给工作内存的变量。

store：将工作内存的变量值传递到主内存，为write准备。

write：将store获取到的值放入主内存。

### 20.volatile变量

作用：

保证此变量对所有线程的可见性。

禁止指令的重排序优化。

效率：

volatile变量读操作性能与普通变量差别不大，写操作会慢

一些。因为需要在本地代码中插入许多内存屏障来保证不会发生乱序执行，开销一般来讲比锁要低。

规则：

每次使用变量前都必须从主内存中刷新值，保证能看见其 线程对变量的修改。

每次修改变量后都必须同步回主内存中，保证其他线程可以看到自己的修改。

变量不会被指令重排序优化，保证代码的执行顺序与程序顺序相同。

非原子性协定：

允许虚拟机将没有被volatile修改的64位数据分为两次32位操作来进行。long和double

目前虚拟机几乎都将64位数据读写操作做为原子操作，所以long和double一般不需要用volatile修饰。

### 21.原子性

基本数据类型的读写具备原子性，同步块中的也具备原子性。

### 22.可见性

一个线程修改了共享变量的值，其他线程立即得知。

普通变量和volatile变量都是如此，volatile变量是立即刷新主内存，普通变量则不会。

### 23.有序性

如果在本线程内观察，所有操作都是有序的，一个线程中观察另一个线程，所有操作都是无序的。

前半句：线程内表现都是串行。

后半句：指令重排序，工作内存与主内存同步延迟的现象。

### 24.先行发生原则

操作A先行发生与操作B，A发生在B操作之前，A产生的影响（赋值，方法调用等）可以被B观察到。

Java中天然的先行发生关系：

24.1程序次序规则：一个线程内，按照控制流程顺序，前面的操作，先行发生与后面的操作。

24.2 管程锁定规则：一个unlock操作先行发生于后面对同一个锁的lock操作，时间先后。

24.3 volatile变量规则：对变量的写操作先行发生于后面的读操作，时间先后。

24.4 线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作。

24.5 线程终止规则：线程中所有操作都先行发生于此线程的终止检测。

24.6 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生。

24.7 对象终结规则：一个对象的初始化完成先行发生于它的finalsize()方法开始。

24.8 传递性规则：操作A先行发生于操作B，B先C，可以得出，A先行发生于C的结论。

一个操作时间上的先发生，不代表这个操作会是先行发生。

时间先后与先行发生原则没有太大关系。

### 25.Java线程调度

协同式和抢占式

##### 25.1 协同式

线程自己工作执行完成以后，主动通知系统切换到另外一个线程，不存在同步问题。

缺点：一个线程阻塞会导致整个系统崩溃。

##### 25.2 抢占式

每个线程由系统分配执行时间，切换不由线程本身决定，不会有一个线程导致系统崩溃的问题。

### 26.线程的状态

新建，运行，无限期等待，限期等待，阻塞

##### 26.1 新建

创建后未启动的线程。

##### 26.2 运行

Running和Ready 可能在执行，也可能在等待CPU为他分配执行时间。

##### 26.3 无限期等待

不会被分配执行时间，需要等待显示的唤醒。

##### 26.4 限期等待

不会被分配执行时间，一定时间后由系统自动唤醒。

##### 26.5 阻塞

线程被阻塞，等待获取一个排他锁。

##### 26.6结束

已终止，结束执行。

### 27.线程安全

不可变，绝对线程安全，相对线程安全，线程兼容，线程对立

##### 27.1 不可变

final 关键字。

##### 27.2 绝对线程安全

不管运行时环境如何，调用者都不需要任何额外的同步措施。

##### 27.3 相对线程安全

常说的线程安全，保证对这个对象单独的操作是线程安全的。

Vector,HashTable,Collections的synchronizedConllection()方法

##### 27.4 线程兼容

常说的线程不安全，对象本身并不是线程安全，调用时使用同步手段来保证线程安全。

##### 27.5 线程对立

无论是否采取了同步措施，都无法在多线程环境中使用。

### 28.线程安全的实现方法

##### 28.1 互斥同步（阻塞）

共享数据在同一时刻只能被一个线程使用，如synchronized关键字和并发包中的ReentrantLock。

会在同步块的前后分别形成monitorenter和monitorexit

需要一个引用类型的参数来指明要锁定和解锁的对象。

如果没有明确指定，去取对象的对象实例（实例方法）或者Class对象（类方法static）来作为锁对象。

##### 28.2 非阻塞同步

先进行操作，如果没有其他线程争用，则操作成功，如果有争用，则采取补偿措施（不断尝试，直到成功为止）。

##### 28.3 无同步方案

28.3.1 可重入代码

如果一个方法，只要输入了相同的参数，就能返回相同的结果，则满足可重入行的要求，为线程安全。

28.3.2 线程本地存储

共享数据的代码能否在一个线程中执行，如果能则共享数据的可见范围限制在一个线程内。

如 生成者—消费者模式。

Web应用交互，一个请求对应一个线程。

### 29.锁优化

##### 29.1 自旋锁与自适应自旋

互斥同步性能最大影响是阻塞，挂起线程和恢复线程都需要转入内核态来完成。

自旋锁是让线程执行一个忙循环，而不是挂起。

JDK1.6默认开始，并增加了自适应的自旋锁。

根据每次自旋的效果，来判定获取该对象的锁是自旋还是挂起，如果自旋的话，循环的次数。

因为自旋锁属于占用执行时间，所以不适宜等待时间过长，因此根据对象以往获取锁的状态来判定，如果自旋时间过长还得不到锁，则放弃自旋，直接选择挂起线程。

##### 29.2 锁消除

一段代码中，堆上的所有数据都不会逃逸出去，被其他线程访问到，就可以把他们当做栈上的数据。无需同步加锁进行。

##### 29.3 锁粗化

如果虚拟机探测到有一串零碎的操作都对同一个对象加锁，

将会把锁范围扩展到操作序列外部。如：

StringBuffer sb = new StringBuffer();

sb.append(s1);

…..

sb.append(s5);

sb.toString();

则会将锁扩展到第一个append之前至最后一个append之后。

##### 29.4 轻量级锁

减少传统重量级锁使用操作系统互斥量产生的性能消耗。

##### 29.5 偏向锁

无竞争的情况下无需同步线程。

线程第一次获取对象锁，CAS操作把线程ID记录到对象头中，CAS成功则这个线程每次进入到这个锁相关的同步块时，无需任何同步操作。

如果有另外一个线程尝试获取该锁时，偏向模式结束。

### 30.编译优化技术

##### 30.1 公共子表达式消除

如果一个表达式E已经经过计算，并且E中所有变量的值都没有发生变化，那么E为公共表达式。

##### 30.2 数组边界检查消除

编译期能判断变量的取值范围永远在[0,arr.length]之间，则无线检查。

例如，在一个循环中进行数组的访问。

##### 30.3 方法内联

##### 30.4 逃逸分析

定义一个对象后，被外部方法调用，或当做参数传递，称为方法逃逸。被其他线程访问，称为线程逃逸。

别的方法和线程无法访问到这个对象，则不会逃逸，可以优化。

栈上分配：

不会逃逸的对象分配在栈上，方法结束，自动销毁。

同步消除：

不会逃逸则同步措施可以消除。

标量替换：

数据无法继续分解则成为标量，如基本类型。

可以分解成为聚合量，如Java对象。

如果不会逃逸，创建一个Java对象时，不直接创建该对象，而是创建若干个标量代替。