C 语言和 Java 语言的存储管理的特点和区别

C 语言

在这部分我来分析 Java 与 C 语言存储管理的特点，下面是 C 语言存储管理的特点。C 语言在编译过程中有些存储管理的特点他分为：内存组织方式，堆和栈，动态管理函数，内存丢失。下面我来分析 C 语言的存储管理特点。

1. 内存组织方式

开发人员将程序编写完成之后，程序要先装载到 i 计算机的内核或者半导体内存中，再运行程序。程序被组织成 4 个逻辑段：

a. 可执行代码，
b. 静态数据（可执行代码和静态数据存储在固定的内存位置），
c. 动态数据（堆），
d. 栈（局部数据对象、函数的参数以及调用函数的联系放在称为栈的内存池中）。

在 C 语言中，对象可以使用静态或动态的方式分配内存空间。
静态分配：编译器在处理程序源代码时分配。
动态分配：程序在执行时调用 malloc 库函数申请分配。
静态内存分配是在程序执行之前进行的因而效率比较高，而动态内存分配则可以灵活的处理未知数目的。
静态与动态内存分配的主要区别如下：
静态对象是有名字的变量，可以直接对其进行操作；动态对象是没有名字的一段地址，需要通过指针间接地对它进行操作。
静态对象的分配与释放由编译器自动处理；动态对象的分配与释放必须由程序员显式地管理，它通过 malloc()和 free 两个函数来完成。以下是采用静态分配方式的例子。

```
1 int a = 100;
```

此行代码指示编译器分配足够的存储区以存放一个整型值，该存储区与名字 a 相关联，并用数值 100 初始化该存储区。以下是采用动态分配方式的例子：

```
1 p1 = (char *)malloc(10*sizeof(int));
```

此行代码分配了 10 个 int 类型的对象，然后返回对象在内存中的地址，接着这个地址被用来初始化指针对象 p1，对于动态分配的内存唯一的访问方式是通过指针间接地访问，其释放方法为：`1 free(p1);`

2.堆与栈

通过内存组织方式，堆用来存放动态分配内存空间，而栈用来存放局部数据对象、函数的参数以及调用函数和被调用函数的联系。

#堆区（heap）：用于动态内存分配。堆在内存中位于 bss 区和栈区之间。一般由程序员分配和释放，若程序员不释放，程序结束时有可能由 OS 回收。**堆(heap)：** 堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用 malloc 等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用 free 等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）。在将应用程序加载到内存空间执行时，操作系统负责代码段、数据段和 BSS 段的加载，并将在内存中为这些段分配空间。栈段亦由操作系统分配和管理，而不需要程序员显示地管理；堆段由程序员自己管理，即显式地申请和释放空间。

#栈区（stack）：由编译器自动分配释放，存放函数的参数值、局部变量的值等。存放函数的参数值、局部变量的值，以及在进行任务切换时存放当前任务的上下文内容。其操作方式类似于数据结构中的栈。每当一个函数被调用，该函数返回地址和一些关于调用的信息，比如某些寄存器的内容，被存储到栈区。然后这个被调用的函数再为它的自动变量和临时变量在栈区上分配空间，这就是 C 实现函数递归调用的方法。每执行一次递归函数调用，一个新的栈框架就会被使用，这样这个新实例栈里的变量就不会和该函数的另一个实例栈里面的变量混淆。**栈(stack)** ：栈又称堆栈， 是用户存放程序临时创建的局部变量，也就是说我们函数括弧"{}"中定义的变量（但不包括 static 声明的变量，static 意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进先出特点，所以栈特别方便用来保存/ 恢复调用现场。从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区。

3.动态管理

a. malloc 函数

在 stdilb.h 头文件中包含该函数，作用是在内存中动态地分配一块 size 大小的内存空间。malloc 函数会返回一个指针，该指针指向分配的内存空间，如果出现错误则返回 NULL。

函数原型：

```
void *malloc(unsigned int size);
```

注意：使用 malloc 函数分配的内存空间是在堆中，而不是在栈中。因此在使用完这块 i 内存之后一定要将其释放掉，释放内存空间使用的是 free 函数。

b. colloc 函数

在 stdlib.h 头文件中包含该函数，函数的功能书在内存中动态分配 n 个长度为 size 的连续内存空间数组。calloc 函数会返回一个指针，该指针指向动态分配的连续内存空间地址。当分配空间错误时，返回 NULL。

函数原型：

```
void *calloc (unsigned n,unsigned size);
```

第一个参数表示分配数组中元素的个数，而第二个参数表示元素的类型。
eg:pArray=(int*)calloc(3,sizeof(int));

c. realloc 函数

包含在头文件 stdlib.h，其功能书改变 ptr 指针指向的空间大小为 size 大小。设定的 size 大小可以是任意的。返回值是一个指向新地址的指针，如果出现错误则返回 NULL。

函数原型：

```
void *realloc (void *ptr,size_tsize);
```

eg:fDouble=(double*)malloc(sizeof(double));

ilnt=realloc(fDouble,sizeof(int));

d. free 函数

该函数的功能是使用由指针 ptr 指向的内存区，使部分内存区内被其他变量使用。ptr 是最近一次调用 calloc 或 malloc 函数时返回的值，free 函数无返回值。

函数原型：

```
void free(void *ptr);
```

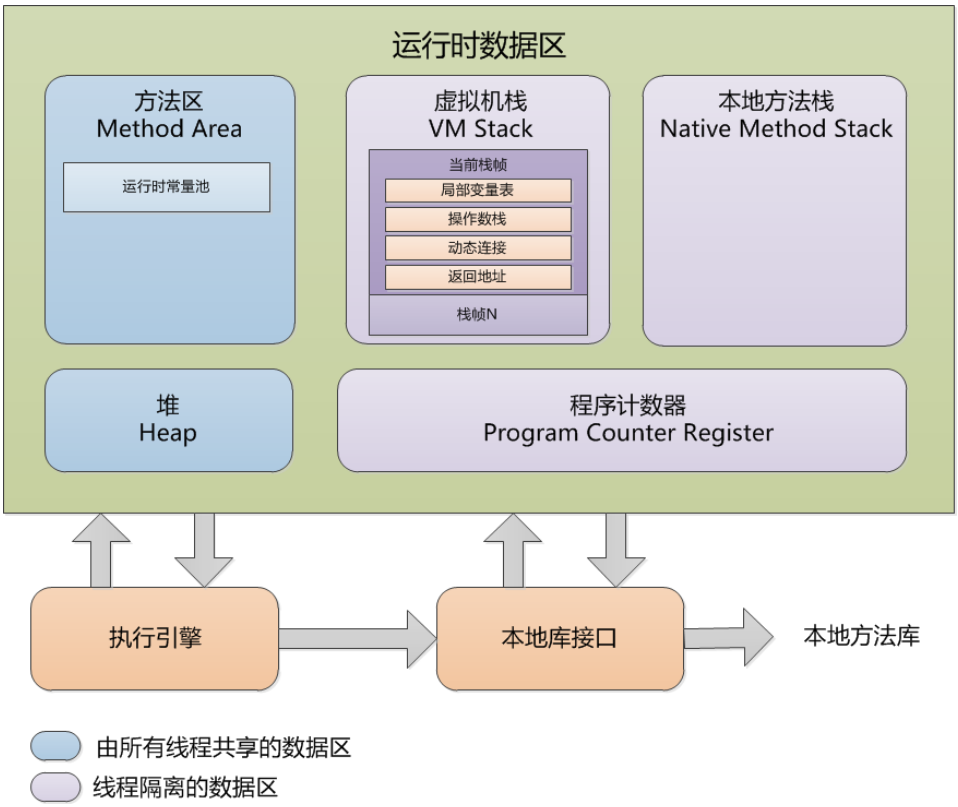
4.内存丢失

在使用 malloc 等函数分配内存后，要对其使用 free 函数进行释放。因为内存不进行释放会造成内存遗漏，从而可能会导致系统崩溃。free 函数的用处在于实时地执行回收内存的操作。一般当程序结束后，操作系统会将完成释放的功能，但建议实时回收内存，防止过度占用系统内存，影响系统的性能，导致系统因此可能崩溃。若开辟的虚拟空间因为指针的所指的地址丢失，从而无法正常 free 该虚拟空间，则称为内存丢失。

原文：https://blog.csdn.net/allen_tony/article/details/62435812

Java 语言

Java 是一种面向对象的语言，在 Java 编译过程中与 C 语言不一样，是因为 Java 还需要通过解释器，Java 在存储管理比 C 语言好很多，像在 Java 我们不需要使用 free 函数，但 Java 已经有了自动垃圾收回，在下部分我来分析 Java 的存储管理特点：



Java 当中它的数据分为进程与线程，进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，进程是系统进行资源分配和调度的一个独立单位。

线程是进程的一个实体，是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源(如程序计数器，一组寄存器和栈)，但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

似乎现在更好理解了一些：

方法区和堆是分配给进程的，也就是所有线程共享的。

而栈和程序计数器，则是分配给每个独立线程的，是运行过程中必不可少的资源。

下面我们逐个看下栈、堆、方法区和程序计数器。

1、方法区（Method Area）

方法区（Method Area）与 Java 堆一样，是各个线程共享的内存区域，它用于存储已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数据。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。

2、程序计数器（Program Counter Register）

程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，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。

下面重点解下 Java 内存管理中的栈和堆。

3、栈（Stacks）

在 Java 中，JVM 中的栈记录了线程的方法调用。每个线程拥有一个栈。在某个线程的运行过程中，如果有新的方法调用，那么该线程对应的栈就会增加一个存储单元，即帧(frame)。在 frame 中，保存有该方法调用的参数、局部变量和返回地址。

Java 的参数和局部变量只能是基本类型的变量(比如 int)，或者对象的引用(reference)。因此，在栈中，只保存有基本类型的变量和对象引用。引用所指向的对象保存在堆中。(引用可能为 Null 值，即不指向任何对象)。

当被调用方法运行结束时，该方法对应的帧将被删除，参数和局部变量所占据的空间也随之释放。线程回到原方法，继续执行。当所有的栈都清空时，程序也随之运行结束。

本地方法栈与虚拟机栈的区别：

本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的 Native 方法服务。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如 Sun HotSpot 虚拟机）直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出 StackOverflowError 和 OutOfMemoryError 异常。

4、堆（Heap）

堆是 JVM 中一块可自由分配给对象的区域。当我们谈论垃圾回收(garbage collection)时，我们主要回收堆(heap)的空间。

Java 的普通对象存活在堆中。与栈不同，堆的空间不会随着方法调用结束而清空。因此，在某个方法中创建的对象，可以在方法调用结束之后，继续存在于堆中。这带来的一个问题是，如果我们不断的创建新的对象，内存空间将最终消耗殆尽。

垃圾回收（Garbage Collection，GC）

垃圾回收(garbage collection，简称 GC)可以自动清空堆中不再使用的对象。垃圾回收机制最早出现于 1959 年，被用于解决 Lisp 语言中的问题。垃圾回收是 Java 的一大特征。并不是所有的语言都有垃圾回收功能。比如在 C/C++中，并没有垃圾回收的机制。程序员需要手动释放堆中的内存。

由于不需要手动释放内存，程序员在编程中也可以减少犯错的机会。利用垃圾回收，程序员可以避免一些指针和内存泄露相关的 bug(这一类 bug 通常很隐蔽)。但另一方面，垃圾回收需要耗费更多的计算时间。垃圾回收实际上是将原本属于程序员的责任转移给计算机。使用垃圾回收的程序需要更长的运行时间。

在 Java 中，对象的是通过引用使用的(把对象相像成致命的毒物，引用就像是用于提取毒物的镊子)。如果不再有引用指向对象，那么我们就再也无从调用或者处理该对象。这样的对象将不可到达(unreachable)。垃圾回收用于释放不可到达对象所占据的内存。这是垃圾回收的基本原则。

早期的垃圾回收采用引用计数(reference counting)的机制。每个对象包含一个计数器。当有新的指向该对象的引用时，计数器加 1。当引用移除时，计数器减 1。当计数器为 0 时，认为该对象可以进行垃圾回收。

然而，一个可能的问题是，如果有两个对象循环引用(cyclic reference)，比如两个对象互相引用，而且此时没有其它(指向 A 或者指向 B)的引用，我们实际上根本无法通过引用到达这两个对象。

因此，我们以栈和 static 数据为根(root)，从根出发，跟随所有的引用，就可以找到所有的可到达对象。也就是说，一个可到达对象，一定被根引用，或者被其他可到达对象引用。

5、再整理下

通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用内存中的栈空间；

而通过 new 关键字和构造器创建的对象放在堆空间；

程序中的字面量（literal）如直接书写的 100、”hello”和常量都是放在静态区中；

栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，理论上整个内存没有被其他进程使用的空间甚至硬盘上的虚拟内存都可以被当成堆空间来使用。

Java 与 C 语言存储的区别：

1. 很显然，在 Java 它支持自动垃圾收回但在 C 语言我们需要用 free 函数执行垃圾收回，这避免发生内存泄漏。
2. 在 Java 中，JVM 中的栈记录了线程的方法调用。而在 C 语言存放函数的参数值、局部变量的值等。
3. 在 C 语言动态管理比较麻烦，但在 Java 动态管理比较简单只要用 new 就可以了，Java 在动态管理比 C 语言简单。

参考文件：https://www.cnblogs.com/andy-zhou/p/5316084.html