Runtime Environments

运行时刻环境（run-time environment）处理很多事务

- 为数据分配安排存储位置
- 确定访问变量的机制
- 过程的连接
- 参数的传递
- 与操作系统，IO相关的接口

存储组织

典型的存储分配方式是将存储空间划分为代码区，静态区，堆区，栈区，其中代码区存放目标程序的代码，静态区，堆区，栈区放置不同类型的数据

编译器的存储分配可以分为

- 静态分配，即编译时刻就能做出存储分配决定，如全局变量的分配
- 动态分配，即运行时分配存储
    - 栈式存储：与过程的调用/返回同步进行存储的分配/回收，变量的生命期和过程的生命期相同
    - 堆存储：数据的生命周期长于创建其的某次过程调用，由程序员手工回收或是使用垃圾回收机制

栈式分配

活动树

过程调用在时间上总是嵌套的，后调用的先返回

程序运行的所有过程活动可以用树来表示

- 节点对应于一个过程活动
- 根对应 main
- 过程 p 的节点的子节点对应这次活动调用的各个过程

树的前序遍历对应过程调用的顺序，而后序遍历对应过程返回的顺序

活动记录

活动调用和返回通常一个控制栈（control stack）进行管理，每个活动都有一个栈中的活动记录（activation record）

活动树的根位于栈底，栈的内容是从活动树到达当前活动的路径。

一个活动记录中可能有的数据包括

- 临时值，如表达式的中间结果不能存放在寄存器中时
- 该过程的局部变量
- 保存的机器状态，包括调用者的机器状态，如返回值和一些寄存器信息
- 访问链，当被调用的过程需要访问另一个活动记录的内容时
- 控制链，指向调用者的活动记录
- 返回值与实参

调用代码序列

调用代码序列（calling sequence）用于实现过程调用，为一个活动记录分配空间，填写信息

返回代码序列（return sequence）用于恢复机器状态，使得调用过程能在调用结束后继续进行

一般实践中调用代码序列会被分割到调用者和被调用者中，考虑到多次调用的情况，应当尽可能把调用代码序列放到被调用者中

一般为活动记录布局时的原则为

- 调用者和被调用者之间传递的值放在活动记录的开始位置，如实参和返回值，这样调用者可以将实参放在自己活动记录的顶部而不用创建被调用者的整个活动记录
- 固定长度的项放在中间位置，如机器状态，控制链和访问链
- 不知道大小的项放在活动记录尾部
- 栈指针在固定长度项的末端

对于变长的数据对象，如果其生命周期局限于过程的生命周期，则同样可以分配在栈中

栈中非局部数据的访问

没有嵌套过程

对于不允许声明嵌套过程的语言（C 语言）而言，变量的存储分配和访问是较为简单的

- 全局变量分配在静态区，其位置不变，并在编译时可知，访问时可以使用确定的地址

- 其他变量一定是栈顶活动的局部变量，可通过运行时栈的栈指针访问

因此很容易将 C 语言的函数作为参数传递

- 参数只需要包括函数代码的开始地址
- 函数中访问非局部变量的模式与过程是如何激活的无关

嵌套过程

当语言允许嵌套地声明过程，并且遵循通常的静态作用域规则时，访问变量变得复杂

如一个过程 p 的声明嵌套在过程 q 内，p 想访问 q 中的变量需要找到 q 的活动记录，而 q 与 p 可能是递归的，在栈中有多个活动记录。如何动态找到相关的活动记录需要使用访问链

引入**嵌套深度**（nesting depth）的概念，对于不内嵌在任何过程中的过程，设定其嵌套深度为 1，如果一个过程 p 在嵌套深度为 i 的过程中定义，则设定 p 的嵌套深度为 $i+1$

访问链

访问链是在活动记录中的指针，如果 p 直接嵌套在过程 q 中，则 p 的任何活动的访问链指向当前最近的 q 的活动

假设嵌套深度为 n_p 的过程 p 访问变量 x，而 x 在深度为 n_q 的过程中声明，则由于 $n_p - n_q$ 已知，故在栈顶沿访问链前进 $n_p - n_q$ 次即可找到 x 所在的活动记录

当过程 q 显式地调用过程 p 的时候

- 如果 p 的深度大于 q，根据作用域规则，p 在 q 中直接定义，p 的访问链指向当前活动记录
- 如果是递归调用，即 p 的深度等于 q 的深度，则新活动的访问链与当前活动的访问链相同
- 如果 p 的深度小于 q，则说明存在过程 r，p 在 r 中直接定义，而 q 嵌套在 r 中，此时 p 的访问链指向栈中最高的 r 的活动记录

在传递过程性参数时，还要传递正确的访问链，这个访问链由调用者直接提供，因为一个过程 r 将过程 p 作为参数传递，说明 p 可以被 r 直接访问到，按照上文的内容为 p 确定访问链即可

显示表

使用显示表可以使访问链的访问代价为常数，具体而言，用数组为每个嵌套深度保留一个指针，指针 i 指向当前栈中最高的，嵌套深度为 i 的活动记录。如果程序访问嵌套深度为 i 的过程 q 中的变量，则 q 的活动记录就是指针 i 指向的活动记录

在调用过程 p 时，在 p 的记录中保存当前深度为 n_p 的指针，并且将指针 n_p 设为当前活动记录，从 p 返回时再恢复指针

堆管理

堆用于存储生命周期不确定，或是生存到被显示删除为止的数据

存储管理器

存储管理器用于分配和回收堆空间，其基本功能为

- 分配：当程序请求内存时，产生一段连续的堆空间，优先从空闲的堆空间分配，如果空闲不足则向 OS 请求更多堆空间
- 回收：将不再使用的空间返回空闲空间缓冲，用于满足其他分配请求

一般期望存储管理器能够满足

- 空间效率：使程序需要的堆空间最小
- 程序效率：提高程序的效率
- 低开销：使内存分配/回收的操作高效

碎片整理

随程序的运行，堆空间被划分为若干占用存储块和空闲存储块的交错，过小的空闲存储块就会成为碎片。在回收时，要将连续的空闲块合并，以尽可能减少堆空间的碎片

分配堆空间时有两种策略

- best-fit：将请求的内存分配在满足请求的最小的空闲块
- first-fit：将请求的内存分配在第一个满足请求的空闲块，总体性能较差但是通常有较好的局部性

也可以使用容器来管理堆，即设定不同大小的空闲块规格，相同规格的块放在同一容器中，如 GNU 的 C 编译器将存储块对齐到 8 字节，存储块大小为 16，24，32……

对于接合空闲块，可以选择在内存块边界设置 free/used 位，指示相邻的内存块是空闲还是占用。也可以用双向链表实现

手工存储管理

手工存储管理常见的问题有两类

- 内存泄露：未释放不能再被引用的数据
- 悬空指针引用：引用已经被释放的数据

解决方法通常有自动存储管理和采用正确的编程模式

- 对象所有者（object ownership）：每个对象有且仅有一个 owner，只有通过 owner 才能删除这个对象。当 owner 消亡时，对象或是已被删除，或是传递给了另一个 owner。这种编程模式可以防止内存泄漏，但是不能解决悬空指针的问题，因为有可能通过一个非 owner 的指针访问一个已被删除的对象
- 引用计数（reference counting）：为每个动态分配的对象附上一个计数，记录当前对其的引用数量，当一个对象引用计数为 0 时则将其回收，但是不能解决访问不到的循环引用对象
- 基于区域的分配（region-based allocation）：当被创建的对象仅在一个计算过程中使用时，可以将其统一分配在一个区域，当该步骤完成时，释放整个区域

垃圾回收

垃圾（garbage）通常指不能再被引用的数据。

垃圾回收是自动回收不可达数据的机制，可以解除手动存储管理的负担

垃圾回收有一个基本要求：语言是**类型安全**的，即回收器能够确定一个数据元素是否为一个指向某内存块的指针

垃圾回收的性能指标为

- 总体运行时间：垃圾回收不应当显著增加程序的总运行时间
- 空间使用：应当最大限度应用可用内存
- 停顿时间：垃圾回收启动时，可能引起程序停顿，这个停顿时间应当尽可能短
- 程序局部性：应当能够改善程序的空间/时间局部性

可达性

一个数据可达即指其能被程序访问到

将不需要对任何指针解引用就能被直接访问的数据称为**根集**（如对于 Java 而言，就是静态成员和栈中的变量）

则可达性的定义为

- 根集中的成员都是可达的
- 对于一个对象，如果指向其的一个指针在可达对象的某字段中，或数组元素中，则该对象可达

对于可达性有性质：如果一个对象变得不可达，它就不会再变成可达的

有四种基本操作可以改变可达对象集合的大小

- 对象分配：返回一个指向新存储块的引用，向可达对象集合增加成员
- 参数传递/返回值

- 引用赋值，如 u = v，会导致 u 中原来的引用不可达，可能使 u 原本指向的对象变得不可达，甚至递归地导致更多对象不可达
- 过程返回：局部变量会消亡，根集会减小，可能导致对象不可达

引用计数垃圾回收器

为每个对象维护引用计数，当引用计数为 0 时回收该对象，引用计数按照如下方式分配

- 对象分配：引用计数设为 1
- 参数传递：引用计数 +1
- 引用赋值：对于 u = v，u 指向的对象引用计数 -1，v 指向的对象引用计数 +1
- 过程返回：局部变量指向的对象的引用计数 -1

在回收一个引用计数为 0 的对象时，此对象中各个指针所指对象引用计数 -1

这种垃圾回收机制的缺点是不能处理不可达的循环引用对象，且开销较大

基于跟踪的垃圾回收

基于跟踪的回收机制不是在垃圾产生时就回收，而是定期运行检查不可达对象并将其回收。通常是在空闲空间耗尽或低于某个阈值时激活回收器

标记-清扫式垃圾回收

标记-清扫式的垃圾回收是一种全面停顿的垃圾回收算法。回收器找到所有不可达对象，然后将其放入空闲列表

- 标记：从根集开始跟踪并标记出所有可达对象
- 清扫：遍历整个堆区，释放不可达对象

即计算可到达对象的集合，然后释放其补集

有一种对标记-清扫算法的优化，即用列表记录所有已分配对象，求已分配对象集和可达对象集的差集

标记并压缩的垃圾回收器

进行压缩的垃圾回收器会在计算出可达对象后将可达对象移动到堆区的一端，以消除存储碎片。整个过程可以分为三步：标记，计算新位置，移动并设置新的引用

拷贝回收器

拷贝回收器预先保留了将对象移入的空间，整个空间划分为两个半空间（semispace），在其中一个半空间分配内存，当其满时，开始垃圾回收，将可达的对象拷贝到另一个半空间，然后两个半空间角色互换