Open Runtime Platform Internals

(Beta Draft)

Department of Computer Science

University of Science and Technology of China

2002/10

第一部分 Introduction

第1章 Java 概述

1.1 Java

1.1.1 Java 的结构

1995 年 5 月，Sun 公司正式提出了 Java 语言，它是一种适合于分布式计算的新型的面向对象的程序设计语言，是为网络而设计的，它对网络环境的适应性潜在于它的结构中，这种结构使得 Java 编写的程序具有安全性、健壮性、平台无关性，Java 程序可以在网络上传输并在各种各样的平台上运行。

Java 的结构是由以下四个不同但是相互关联的技术引出的：

- Java 程序设计语言
- Java 类文件格式
- Java 应用程序编程接口 API
- Java 虚拟机

当你编写并运行 Java 程序的时候，就会用到这四种技术。首先要把你的程序表示成一个用 Java 编程语言编写的源文件，然后把这个 Java 源文件编译成 Java 类文件，最后在 Java 虚拟机上运行这个类文件。编写程序的时候，要通过调用实现 Java 应用程序编程接口（API）的类中的方法访问系统资源，程序运行的时候，就通过调用实现 Java API 的类文件中的方法来实现程序中的 Java API 调用。这四个部分之间的联系可以表示如下：

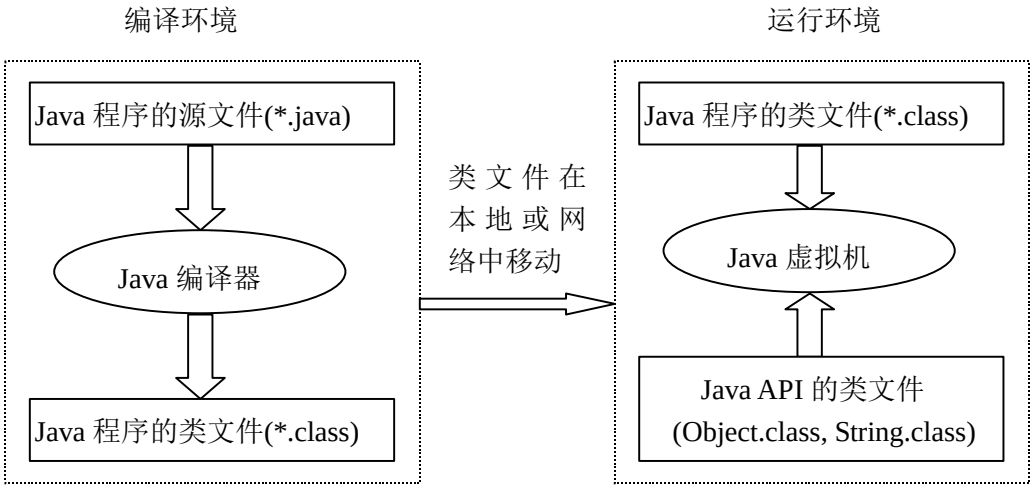


图 1.1 Java 编程环境

Java 虚拟机与 Java API 一起构成了 Java 运行系统，除此之外，它们也被叫做 Java 平台。由于 Java 平台本身是用软件实现的，因此，Java 程序可以运行在各种计算机上。

1.1.2 Java 语言的特性

Java 语言有很多有用的特性，这里只介绍以下几种：

- 平台无关性。Java 技术之所以在网络环境中有用，其关键原因之一就是 Java 可以创建不需修改就可以在多种平台上运行的二进制可执行代码。这在网络环境中是非常重要的，因为网络中互联的通常都是各种各样的计算机和设备。比如说，在一个典型的企业环境中，网络连接的可能有艺术部的 Macintosh，工程部的 UNIX 工作站，和其余的运行 Windows 的 PC 机。虽然这种

安排可以使得公司内不同种类的计算机和设备共享数据，但是这需要大量的管理工作，这样的网络会要求系统管理员在各种计算机上及时的更新程序的不同平台的版本。如果程序不需要修改就可以在任何计算机上运行，而不管计算机的类型，那么这将大大的减少管理员的工作量，尤其是当程序可以真正的在网络上传输的时候。Java 对平台无关性的支持贯彻在 Java 的结构中。Java 的结构以几种方式支持平台无关性，最主要的还是通过 Java 平台本身。Java 平台在运行的 Java 程序和底层的硬件与操作系统之间起一个缓冲的作用。在假设 Java API 的类文件可以在运行时获得的前提下，Java 程序被编译，在 Java 虚拟机上运行。Java 虚拟机运行程序，API 让程序访问计算机的底层资源。不管 Java 程序怎样运行，它只需要与 Java 平台交互，不需要关心底层的硬件和操作系统。这样，Java 程序就可以在任何有 Java 平台的计算机上运行。

- 安全性。Java 语言是为网络环境开发的，而安全性在网络环境中是一个非常重要的问题，为此，Java 在语言和运行环境中引入了多级安全措施。在语言中，Java 编译器为安全性提供的一个主要措施是它的内存分配和引用模型，首先，Java 的内存分配不像 C 和 C++一样是由 Java 编译器决定的，而是延迟到运行时由 Java 运行系统决定，内存的布局依赖于 Java 运行系统所在的软硬件平台的特性，其次，Java 没有 C 和 C++意义上的内存单元指针，Java 编译器通过符号指针引用内存，符号指针在运行的时候解释为实际的内存地址，程序员不能强制引用内存指针，也就是说，内存的分配和引用对于程序员是透明的，这样，程序员就不能够直接进行内存分配，从而使得 Java 的应用更安全可靠；在 Java 运行系统中，Java 提供的安全措施是进行字节码检查，在 Java 应用中，经常需要从别的地方引入代码，但是 Java 运行系统不能保证这些代码是安全的，因为网络病毒或者别的形式的入侵者可以绕过 Java 编译器生成危险的字节码，由于这个原因，在字节码执行之前，Java 运行系统就要对这些代码进行检查，以确保代码遵循以下安全规则：
    - 不存在伪造的指针。
    - 没有违反访问权限。
    - 严格遵循对象规范来访问对象。
    - 用合适的参数调用方法。
    - 没有栈溢出。

  通过 Java 语言的内在安全机制，再加上对字节码的验证，Java 就建立了一个严密的安全体系。
- 垃圾收集。用 C 和 C++写软件时，程序员必须非常仔细的处理内存的使用，当一个内存块不再使用时，就必须释放它，但是，在开发大的软件项目时，要程序员自己管理内存是非常困难的，而且内存管理不当通常还是造成系统故障和存储空间浪费的原因之一。为了解决这个问题，在 Java 中，Java 系统内嵌了一个垃圾收集程序，它扫描内存，自动释放不再使用的内存块，这样一来，程序员就不再需要关心内存管理问题，编写 Java 程序就会变得简单，而且减少了程序中因内存管理而出错的可能性。
- 运行时检查。有些操作，比如方法调用，instanceof 操作，类型转换，数组操作，在编译的时候无法确定其正确性，在运行的时候才能够确定它的值。

1.2 Java 虚拟机

1.2.1 Java 虚拟机的结构

从概念上看，Java 虚拟机（JVM）是一个想象中的、能运行 Java 字节码的操作平台，是由 Java 规范定义的抽象计算机，Java 技术的核心就是 JVM，所有的 Java 程序在 JVM 上运行。Java 虚拟机一般分为以下几个部分：类装载器（字节码验证器）、解释器和编译器，其组成情况如下图所示：

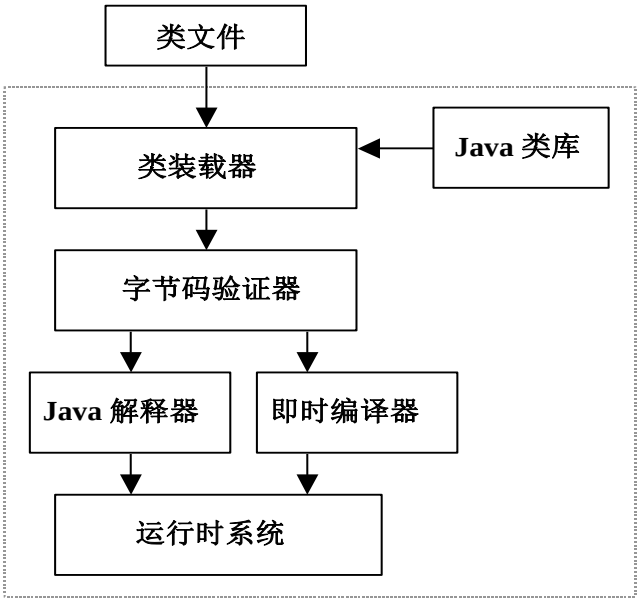


图 1.2　Java 虚拟机结构图

Java 虚拟机执行字节码的过程可以分为三步：代码的装入、代码的验证和代码的执行。代码的装入由类装载器完成，类装载器负责装入程序运行时需要的所有代码，其中包括程序代码中用到的所有类。随后，被装入的代码由字节码验证器进行安全性检查，以确保代码不违反 Java 的安全性规则，字节码验证器还可以发现操作数栈溢出、非法数据类型转化等多种错误，通过验证之后，代码就可以提交运行了。Java 字节码的运行有两种方式：即时编译方式和解释执行方式。即时编译方式是由即时编译器先将字节码转化为本地机器代码，然后再全速执行机器代码；解释执行方式是由解释器通过每次翻译并执行一小段代码来完成 Java 字节码程序的所有操作。其中，即时编译方法具有较高的性能。

Java 虚拟机定义了一组抽象的逻辑组件，包括指令集、寄存器组、栈结构、垃圾收集器和存储区五个部分，其中，寄存器组包括程序计数器、栈顶指针、用于指向当前执行方法的执行环境的指针和用于指向当前执行方法的局部变量的指针；栈用于提供操作参数、返回运行结果和为方法传递参数等；垃圾收集器用来收集不用的数据堆；存储区用于存放字节码的方法代码、符号表等。Java 虚拟机运行程序的时候，需要内存去存放很多东西，包括字节码，从装载的类文件中获取的信息，程序实例化的对象，方法的参数，返回值，局部变量和计算的中间结果。Java 虚拟机把执行程序需要的内存组织成几个运行时数据区。运行时数据区的规范相当抽象，不同的 Java 虚拟机实现的运行时数据区的结构也不相同。有些运行时数据区是程序的所有线程共享的，有些则是某个线程独有的。其中，方法区和堆是所有线程共享的。当 Java 虚拟机装载类文件的时候，它分析该类文件包含的二进制代码中的类型信息，并把这些类型信息放进方法区中。程序运行的时候，Java 虚拟机把程序实例化的所有对象放到堆上。每个线程建立之后，都会有一个自己的程序计数器和栈。如果线程正在执行一个 Java 方法（不是本地方法），程序计数器中的值就表示下一条要执行的指令，线程的栈中存放该线程 Java 方法调用的状态，包括它的局部变量，方法调用时用到的参数，返回值（如果有的话）和中间计算的结果。本地方法调用的状态以实现相关的方式存放在本地方法栈中。栈是由栈帧组成的，一个栈帧包含一个 Java 方法调用的状态，当线程调用方法时，Java 虚拟机就把一个新的栈帧压入该线程的栈上，方法结束后，Java 虚拟机弹出并抛弃该方法的栈帧。

1.2.2 类装载器

类装载器是用于从程序和 Java API 中装载类文件的。JVM 有两种类装载器：系统类装载器和用户定义的类装载器。自引导的类装载器是 JVM 实现的一部分，而用户定义的类装载器是 Java 运行程序的一部分。不同的类装载器装载的类被放到 JVM 的不同名字空间中。

类装载器不仅仅用于找到并导入类的二进制数据，它也被用来检查导入类的正确性，分配并初始化类变量，辅助符号引用的解析。而这些都是按照严格顺序执行的：

1. 装载(loading)：找到并导入一个类型（类或接口）的二进制数据；
2. 链接(linking)：执行检查(verification)，准备(preparation)和（可选）解析(resolution)。
   a) 检查(verification)：保证导入类型的正确性。
   b) 准备(preparation)：为类变量分配内存并把这些内存初始化为缺省值。
   c) 解析(resolution)：把类型中的符号引用转换为直接引用。
3. 初始化(initialization)：调用 Java 代码把类变量初始化为合适的初始值。

1.2.3 垃圾收集

“垃圾收集”意味着不再被程序需要的对象就是垃圾，应该被扔掉。垃圾收集也叫做“内存循环”，如果一个对象不再被程序引用，它所占用的堆空间就可以被回收，以便分配给新的对象。垃圾收集器必须确定哪些对象不再被引用，然后释放这些对象占用的空间，在释放这些对象的过程中，垃圾收机器必须运行这些对象的 finalizer。

Java 虚拟机规范并没有要求垃圾收集，但是在没有发明无限的内存之前，大多数 JVM 实现都是有垃圾收集的。垃圾收集可以改善堆碎片。堆碎片发生在正常的程序执行过程中，新的对象被分配，不再被引用的对象被释放，这样，被释放掉的堆空间就位于活对象占用的堆空间之间，这时，分配新对象的请求就可能导致必须扩展堆的大小，因为现存的堆中没有连续的自由空间存放新对象，最后，堆中只剩下一些间隔的小空间，造成资源浪费，并且，扩展堆的大小会严重影响程序的性能。垃圾收集还可以保证程序的完整性，Java 程序员不会因为错误地释放内存造成 JVM 崩溃。

任何垃圾收集算法都必须作两件最基本的事情。首先，它必须检测垃圾对象，然后，它必须回收这些对象占用的空间并在程序中重新利用这些空间。

垃圾检测是由定义一个根集(root set)然后从这个根集出发确定可达性来实现的。如果从根集出发，存在一条引用路径，程序可以利用这条路径访问某个对象，那么这个对象就是可达的。根集总是可被程序访问的，任何从根集可达的对象都被认为是“活”对象，不可达的对象就被认为是垃圾，因为它们不会影响程序将来的执行过程。

1.2.4 指令集概述

Java 虚拟机指令由一个表示要执行的操作的一字节的操作码，后跟零或多个操作数组成。很多指令没有操作数只有一个操作码。

忽略掉“异常”，Java 虚拟机解释器的内部循环其实就是

```
do {
        取下一个操作码；
        如果有操作数则取操作数；
        执行操作码指定的动作；
} while (还有指令);
```

操作数的数目和大小是由操作码决定的。如果操作码多于一个字节，就用 big-endian 高位字节在前的顺序存储。

字节码指令流都是单字节对齐的，只有 tableswitch 和 lookupswitch 异常，它们要求对它们的一些操作数强制进行内部 4 字节对齐。

Java 虚拟机指令集中的绝大多数指令都可以表示它们的操作数的类型。用 i 表示 int，l 表示 long，s 表示 short，b 表示 byte，c 表示 char，f 表示 float，d 表示 double，a 表示引用（reference）。那么，iload 指令就表示把一个 int 型的操作数压入操作数栈，而 fload 指令的动作与此相同，但是它的操作数是 float 型的。对于那些操作数的类型确定的指令，它们的指令助记符中就不需要带有类型信息，比如，arraylength，它肯定是在数组上操作，再比如 goto 指令，它是一个无条件转移指令，不能够操作在任何有类型的操作数上。

Java 虚拟机指令大致可以分为十种：装载和存储指令，运算指令，类型转换指令，对象创建和操作指令，操作数栈管理指令，控制转移指令，方法调用和返回指令，抛出和处理异常的指令，实现 finally 的指令，和同步指令。关于这些指令的具体含义，请参考 Java 虚拟机规范。

1.2.5 线程

JVM 可以支持多线程（同时运行多个线程）。这些线程独立地执行代码，对驻留在共享的主存储器中的值和对象进行操作。多线程可以通过多个硬件处理器、时间分片的单个硬件处理器或者时间分片的多个硬件处理器来支持多线程的运行。

任何线程都可以标记为守护线程(daemon thread)。线程运行的时候如果线程中的代码创建了一个新的线程，当且仅当创建线程是守护线程，新的线程才可以在开始的时候标记为守护线程。程序可以调用类 Thread 中的 setDaemon 方法来设置一个特定的线程是不是守护线程。JVM 启动的时候，只有一个非守护线程，这个线程通常会调用类中的 main 方法。JVM 也可以为了内部用途创建别的守护线程。当所有的非守护线程死亡的时候，JVM 退出。

为同步线程，Java 使用了监视器(monitor)，这是一种在同一时间只允许一个线程执行该监视器保护的代码区的高层机制。Java 的监视器支持两种形式的线程同步：互斥(mutual exclusion)与合作(cooperation)。互斥是由 JVM 通过对象锁支持的，使得多个线程可以独立地对共享数据进行操作而不会彼此干扰。合作是由 JVM 通过类 Object 的 wait-and-notify 方法来支持的，可以使得多个线程为了共同的目的协同工作。

1.2.6 本地方法接口

Java 虚拟机规范并没有要求 JVM 实现一定要支持本地方法接口，有些 JVM 实现根本不支持本地方法接口，有些则支持好几个本地方法接口，每一个都有其特定的目的。Sun 公司的 Java 本地接口(JNI)是为了可移植性设计的。JNI 设计的目的是它可以被任何 JVM 实现所支持，而不用管这个 JVM 实现用的是什么垃圾收集技术或对象表示方法。这反过来可以使得开发者在一个特定的主机平台上把同一个（JNI 兼容）本地方法的二进制代码链接到任何支持 JNI 的虚拟机实现上。

除了 JNI 之外，设计者可以选择创建专有的本地方法接口。为了提高性能，JVM 实现设计者可以决定提供它们自己的低层本地方法接口，这是与它们特定的实现结构紧密相关的。设计者也可以提供比 JNI 高层的本地方法接口，比如说，把 Java 对象变成一个组件软件模型的本地方法接口。

为了有效地工作，本地方法必须与 JVM 的内部状态进行某种程度的交互。例如，一个本地方法接口可能允许本地方法执行下面这些操作：

- 传递并返回数据。
- 访问实例变量或者调用被垃圾收集的堆上对象中的方法。
- 访问类变量或者调用类方法。
- 访问数组。
- 锁定一个对象以便当前线程单独使用。
- 在被垃圾收集的堆上创建新对象。
- 装载新的类。
- 抛出新的异常。
- 捕获由本地方法调用的 Java 方法抛出的异常。
- 捕获由 JVM 抛出的异步异常。

- 通知垃圾收集器不再使用某个对象。

设计一个提供这些服务的本地方法接口很复杂。设计需要保证垃圾收集器不会释放正在被本地方法使用的任何对象，如果 JVM 实现的垃圾收集器移动对象以减少堆分段，那么本地方法接口设计必须保证下面两个条件之一：

- 对象的引用被传递到一个本地方法之后，该对象才可以被移动；
- 对象的引用被传给一个本地方法之后，该对象被“钉住”，直到该本地方法返回或者通知它已经不需要该对象为止。

因此，本地方法接口与 Java 虚拟机的内部工作是紧密联系在一起的。

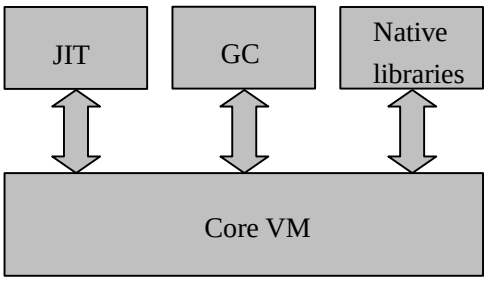
第2章 开放式运行平台

2.1 开放式运行平台的主要组成

开放式运行平台（ORP）是一个研究动态编译和垃圾收集技术的开放资源研究平台，它实现了对执行类型安全字节码的支持，可以运行很多为 Java 虚拟机编写的程序。

ORP 由虚拟机 VM(Virtual Machine)，即时编译器 JIT(Just-In-Time)和垃圾收集器 GC(Garbage Collector)三个模块组成，各个模块之间独立性很好。因此，这三个模块中的任何一个都可以被替换掉而不会影响其它模块的工作。如果用一个性能更好的模块代替了 ORP 中对应的模块，那么整个 ORP 的性能就会提高，ORP 的目的也正在于此。

如果用模块图表示 ORP，图示如下：



其中本地库（native libraries）也是必不可少的，它包括 Java API 中的类。

对于一个 Java 程序，它首先被 Java 编译器编译成字节码，然后再被 JIT 编译器编译成为机器码，在 Java 虚拟机（Core VM）中执行，对于程序执行过程中不再被引用的对象，也就是“垃圾”，则由 GC 来收集。ORP 的大致工作过程就是如此。

2.2 Core VM

Java 程序执行时，Java 虚拟机中的类装载器把类装载进来之后，类中的信息就存放在方法区中。在 ORP 中，对于每个被装载进来的类，ORP 都会为它创建一个 Class 对象，Class 对象就在程序和方法区之间充当接口的作用，如果程序要使用类中的数据，比如类中的域或者方法，就得通过该类的 Class 对象才能得到。

在 ORP 中，主要的数据结构有：Class，Class_Table，Vtable，Intfc_Table，Const_Pool，Class_Loader，等等。Class 的大概框架如下图中所示。Class_Table 的结构比较简单，就是一个 Class 数组。

```
class Class_Table {
public:
      ┆
private:
#define CLASS_TABLE_SIZE 1024
Class *_table[CLASS_TABLE_SIZE];
};
```

Intfc_Table 顾名思义就是接口表，表中列出了类实现的所有接口，其结构如下：

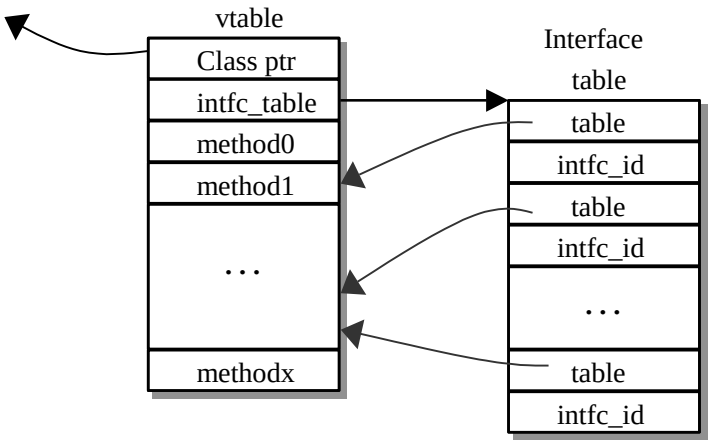
```
typedef struct {
unsigned char  **table;    //指向 Vtable 中的方法数组的指针
unsigned intfc_id;          //接口的 id
} Intfc_Table_Entry;

typedef struct Intfc_Table {
uint32 n_entries;          //接口的数目
Intfc_Table_Entry entry[1];     //接口表项数组
} Intfc_Table;
```

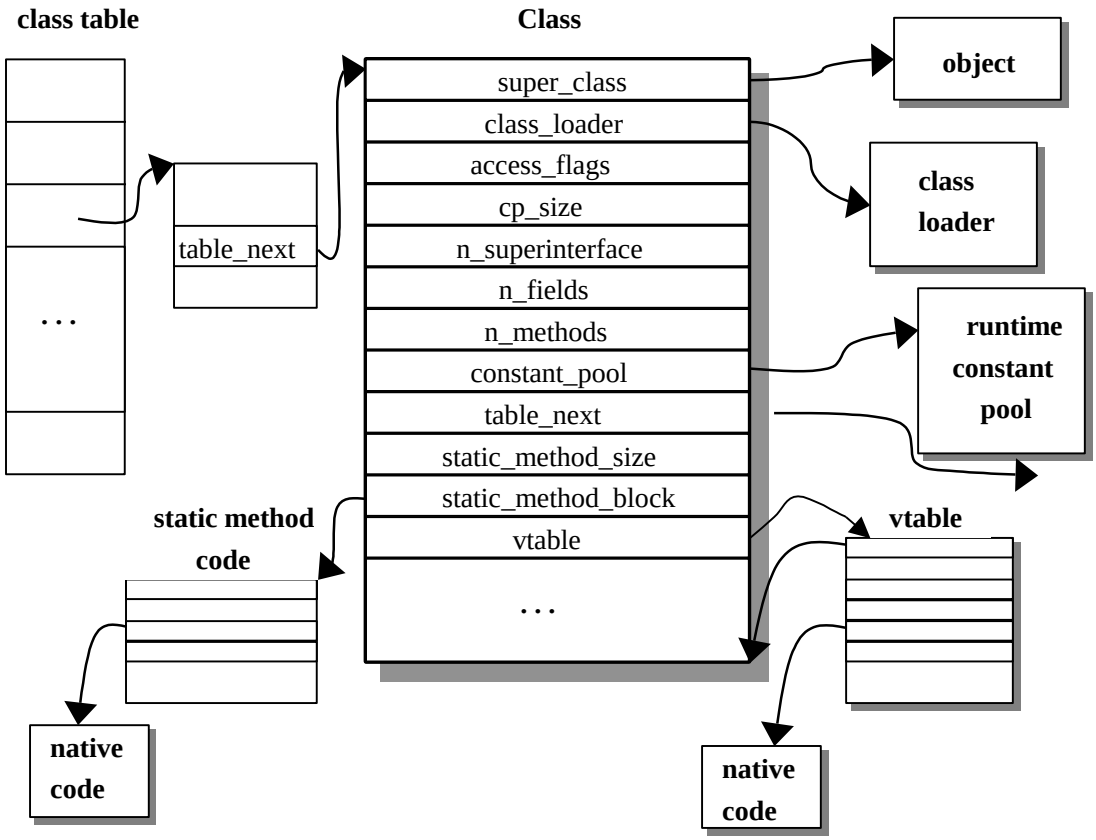
Vtable 中列出了类中除了静态方法之外的所有方法的代码。

```
typedef struct VTable {
Class                *clss;              //指向所属的类的指针
Intfc_Table          *intfc_table;       //接口表，如果没有接口表，则该项为空
unsigned char  *methods[1];              //方法的代码
} VTable;
```

对于类中的静态方法，ORP 为每个类分配一个静态数据区，用于放置指向静态方法代码的指针。

这几个数据结构之间的关系可以用下图表示：





其中，Class 对象中的 super_class 指向表示该类的超类的 Class 对象，class_loader 指向装载该类的类装载器，access_flags 的值是类和接口声明中所使用的修饰符，有 ACC_PUBLIC，ACC_FINAL，ACC_SUPER，ACC_INTERFACE，ACC_ABSTRACT 五种，cp_size 表示该类的常数池表中表项的数目，n_superinterfaces 是类或者接口所实现的直接超接口的数目，n_fields 是类或者接口中声明的变量（包括静态变量和实例变量）的数目，n_methods 是类中的方法的数目，constant_pool 指向该类的常数池，table_next 指向 Class_Table 中的下一个 Class 对象，static_method_size 是 static_method_block 的字节数，static_method_block 则存放指向类中声明的静态方法的代码的指针。

2.3 ORP 中的即时编译器（JIT）

JIT 编译器出现之前，在传统的 Java 虚拟机中，要执行一个 Java 程序，首先要通过 javac 这样的编译器把它编译成字节码，在执行的时候，调用解释器对字节码逐条进行解释执行。但是，解释执行的速度很慢，在 JIT 编译器出现之后，很多 Java 虚拟机都采用了 JIT 编译编译器，同样，ORP 也采用了 JIT 编译器。JIT 编译器与传统的编译器不同，它并不是对整个程序进行编译，而是只在需要的时候才进行编译。JIT 编译器的编译单位是方法，因此，当某个对象的方法被调用的时候，JIT 编译器才对这个方法的字节码进行编译，把字节码编译成机器码，以后如果这个方法被调用的时候，就可以使用这段编译好的机器码，如果这个方法被频繁的调用，那么机器码的执行效率就会比纯粹的解释执行的效率要高很多。

ORP 中采用了两种即时编译器 O1 和 O3。O1 叫做快速代码产生编译器，O3 叫做优化编译器。在 ORP 中，执行 Java 程序的时候，可以在命令行指定 O1 和 O3 的优先顺序，但是，缺省情况下，Java 虚拟机会首先用 O3 来编译方法，只有在 O3 失败的时候才会用到 O1，但是一般情况下，这两个编译器都是正确的，这样就不会再用到 O1。如果在命令行指定 O1 优先于 O3，并且打开重编译开关，那么，对象方法被首次调用的时候，会首先用 O1 编译，如果这段代码被频繁地调用，从而证明自己是一段“热”

代码的时候，就需要进行优化，这时会需要再用 O3 对这段代码进行编译，生成优化代码，以提高执行效率。ORP 采用的就是这样的动态编译机制，对不经常使用的代码选择快速的编译器，对频繁使用的代码选择优化编译器，而选择编译器的关键在于确定一段代码是否是“热”代码，这个工作是由 O1 编译器完成的。

O1 编译器可以快速地产生机器代码，并且能够进行一些轻量级的优化，比如，lazy 代码选择，公共子表达式消除，数组越界检查的消除，等等。O1 的编译时间是线性的，因此 O1 产生的机器代码执行速度比解释执行要快得多。如果在命令行还选择了重编译，那么，O1 在编译字节码的时候，会在生成的机器代码中插入一些统计代码，对方法被调用的次数进行计数，当计数值达到设定的阈值时，就会触发重编译，这段代码就会被 O3 编译从而进行优化。

O1 编译器只对代码进行轻量级的优化，而 O3 编译器则是对代码进行更为细致的优化，产生的代码的质量比 O1 产生的要好。为了产生高质量的代码，O3 编译器需要花很多时间在优化上，它采用传统的编译方法，为字节码建立一个中间表示，并基于这个中间表示进行优化。

然而，JIT 编译器与传统的编译器不同，传统的编译器只需要对程序作一次编译，生成可执行文件，之后该文件就可以被频繁的执行，不再需要编译，也就是说，编译时间不是运行时间的一部分。但是对于 JIT 编译器来说，因为是在程序执行的时候进行编译，编译时间就是运行时间的一部分，因此，如何在编译时间和执行效率之间找到一个平衡点就成为了 ORP 设计的关键之一。

2.4 ORP 中的垃圾收集器

诸如 Lisp，Smalltalk，Modula-3 和 ML 这样的语言的垃圾收机器已经存在很多年了，但是直到最近垃圾收集才成为主流运行环境的一部分，现在简单的介绍以下 ORP 的垃圾收集器 GC。ORP 提供了两个垃圾收集器，一个移动的代收集器和一个不移动的代收集器。

首先介绍 ORP 中的内存组织。在 ORP 中，内存分为三个区：被收集的（collected）区，被跟踪的（traced）的区和不被跟踪的（untraced）区。被收集的区中是垃圾收集器分配和回收的所有对象。被跟踪的区中是指向被收集的对象的指针，这些指针必须由收集器检查和回收，但是收集器并不管理被跟踪的区。被跟踪的区中包括的是静态分配的数据，运行栈与硬件寄存器，和 ORP 管理（显式的分配和回收）的区域。不被跟踪的区中是既不被收集器收集也不被检查的数据，这包括使用 JNI 的应用程序代码。我们感兴趣的是被收集的区。被收集的区（有时候指的是堆）分为两个区域：年轻对象空间（YOS）和成熟对象空间（MOS）。YOS 中是最近才分配的对象，这些对象经过几次垃圾收集后，就被提升到了成熟对象空间 MOS 去了，MOS 用 train 算法清理（scavenge）。

YOS 分为几个代（generation），重编译垃圾收集器时，YOS 中代的数目可以很容易的重置。每个 YOS 代又可以分为一个或几个 step。Step 1 是一个代中最年轻的 step，当对一代进行垃圾收集时，幸存下来的对象就被从当前的 step 移到下一个老的 step，一代中最老的 step 中的对象则被提升到下一代中最年轻的 step 中去。一个 step 中的所有幸存对象是一起移动的，这就避免了给每个对象添加年龄信息，这种方法节省了在对象中设置年龄计数器的空间和操作年龄计数器的时间。另外，当前的 ORP 实现允许在创建垃圾收集器时规定每一代中 step 的数目。每个 step 被存储成许多固定大小的块，这些块不必相邻。每个块由 2^i 个字节组成，并且以 2^i 字节边界对齐。从对象的地址可以有效的计算出一个对象的代，不比在对象中设置代或者 step 标记，从而可以使得垃圾收集器独立于语言的实现（比如对象的格式），减少维护对象中像这样的信息所耗费的时空代价。ORP 中 GC 的具体实现见第 15 章。

第二部分 核心虚拟机

第3章 核心虚拟机总览

我们在这一章将主要讲述 ORP 中 VM 的主要构成，我们将分成三个部分，第一部分，我们将讲述 VM 的主要组成部分；JIT 和 GC 是 ORP 中比较特殊的两个部分，在设计上，由于采用了良好的接口，可以使得 JIT/GC 的实现相互分离。第二部分，我们将介绍 ORP 中 VM 和整个 JIT 之间的交互；第三部分，我们将介绍 VM 和 GC 模块之间的交互。

总览部分，我们将简单介绍各个部分的基本结构、功能，主要的，我们将在这一章讲述在后续的章节中需要的数据结构描述，class 的布局。

3.1 各个部分的结构、功能

ORP 的各个部分如图 3.1 所示，整体上我们可以分成核心虚拟机，垃圾搜集器，即时编译器和相关的本地方法库。在组织上，我们可以把这些组成部分称为 Java 虚拟机的核心模式；通常的一些基本的类库、应用程序都可以称为应用模式。

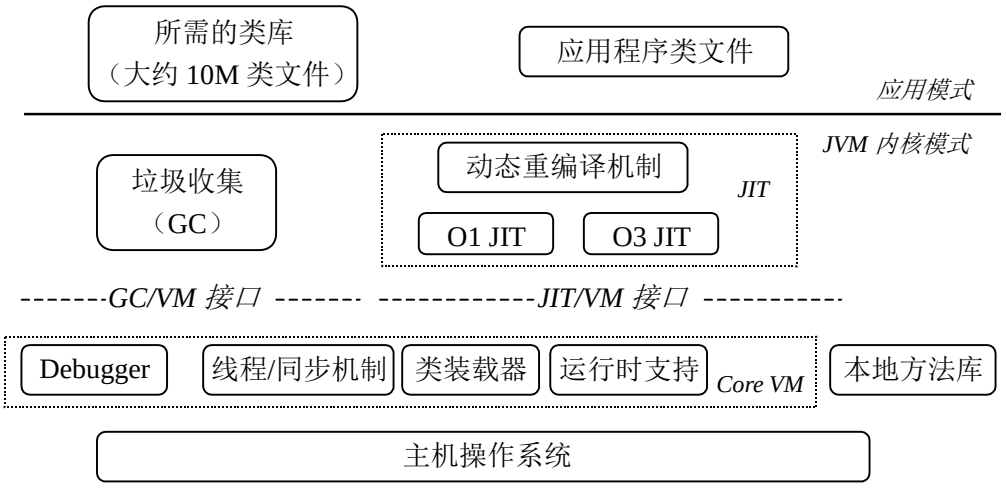


图 3.1: ORP 的主要组成

我们在这一章主要讨论核心虚拟机内部的组成，从图 3.1 种我们也可以看到，主要的组成分为类装载器，线程和同步支持机制、运行时支持部分。ORP 的设计中相当注重良好的接口设计来实现它的设计目标—研究动态编译和自动内存管理的基础设施。因此，我们会在后面详细的讨论核心虚拟机中的即时编译支持部分、垃圾收集支持部分。

下面，我们简单的来看一下各个部分的基本功能：

3.1.1 类装载器

类装载器在 Java 虚拟机规范中是装载应用程序的入口。ORP 中的类装载器的任务主要是通过特定的名字来找到相应的类(Class)或者接口(Interface)，然后创建这些类或接口在 ORP 内部的表示。维护运行期的常数池是类装载器的重要任务。常数池中保存了所有和符号引用相关的信息，是 Java 语言平台无关特性实现的重要部分。类装载器也是虚拟机的入口，它负责装载基本的类并初始化，并调用 main 方法。

在类装载器的处理上，可以把它的上述过程称为链接(Linking)。在虚拟机规范中，规定了做的三个

工作：验证(Verification)，准备(Preparation)，解析(Resolution)。验证主要是为了保证要装载的 Class 文件在结构上是合法的，没有被恶意篡改。在 ORP 中，这部分的工作并没有实现。准备主要是创建类的静态域并把它们初始化称它们的缺省的值。解析部分的工作是动态的在运行期常数池中通过符号引用来决定具体的值，这中间也包括对域、方法和类的访问权限以及域或方法是否存在，对于解释器，会在这个时期执行 bytecode；而即时编译器会首先即时编译该方法然后调用。

我们在第 4 章会详细讨论类装载器。

3.1.2 线程和同步支持机制

Java 语言中内建了多线程支持，因此在实现上线程和同步支持是很重要的组成部分。对于多线程的应用而言，这部分的性能至关重要。在 ORP 内部实现上，为了实现线程的高效性、公平性，线程采用了本地绑定(native-binding)的方法，将每一个 Java 线程都映射为实际运行的操作系统上的线程，使用操作系统的调度器来实现对它们的高效支持。同步是保证程序顺序一致性的必要手段，同步在保证 Java 应用语义正确中发挥着重要的作用，同时也是多线程协同工作正确性的重要保证。在典型的商务应用中，同步操作占了相当大的部分，因此高效的实现同步也是高性能虚拟机的重要因素。

我们将在第 5 章详细讨论 ORP 中的多线程支持，同步支持。

3.1.3 即时编译器(JIT)支持

即时编译器是高性能 Java 虚拟机中不可缺少的组成部分。ORP 作为提供运行期编译研究的基础设施，与核心虚拟机中的即时编译器支持例程是分不开的。

即是编译器内部的实现可以千差万别，但是和具体的核心虚拟机之间的交互可以通过某种预先定义的接口作规定。ORP 中核心虚拟机对即时编译器的支持除了提供必要的装载、解析类、方法、域等操作外，还要负责调用真正的即时编译器来编译方法，提供适当的运行时支持例程，提供反射例程以及提供异常处理、堆栈回退等处理上。

我们将在第 6 章详细讨论 ORP 中核心虚拟机对即时编译器的支持。

3.1.4 垃圾收集器(GC)支持

垃圾收集器是 Java 自动内存管理功能的提供者。在运行期，垃圾收集器必须能够从当前的运行状态中准确地给出当前的活动的根集，并由这些根集出发开始扫描所有活动的引用。在 ORP 中，这部分也是由预先定义好的接口函数来完成的。核心虚拟机必须为垃圾收集器实现取得当前的活动根集的方法，并且在完成垃圾收集后恢复各个线程运行的基本接口方法。

我们将在第 7 章讨论 ORP 中核心虚拟机对垃圾收集器的支持。

3.1.5 其他

核心虚拟机的组成并不是以上几个部分简单的组合，各个部分是有机的结合在一起的。除了上述的几个部分外，在 ORP 中还有调试器支持，以及一些性能相关的本地方法支持等，这些都是 ORP 的组成部分。但限于篇幅，我们并不涉及很多这方面的细节。

3.2 内部的数据结构

这里介绍的数据结构都是在核心虚拟机（Core VM）的 C++实现中描述 Java 类和 Java 类对象的最基本的 C++数据结构，包括 Class 结构，Field 类结构，Method 类结构，Vtable 结构和 Intfc_Table 结构。下面对这些数据结构的定义作一个大致介绍。

3.2.1 Class 数据结构

下面列出了 Class 结构中的一些主要的域：

```
typedef struct Class {
    Class    *super_class;          //Java 类的超类 Class 结构指针
    Class_Loader *class_loader;     //装载类的类装载器，如果是系统类装载器，NULL
    uint16 access_flags;            //访问标识，有 ACC_PUBLIC，ACC_FINAL，ACC_SUPER，
                                    // ACC_INTERFACE，ACC_ABSTRACT 五种
    uint16 cp_size;                 //类的常数池中表项的数目
    uint16 n_superinterfaces;       //类实现的直接超接口的数目
    uint16 n_fields;                //类中域的数目
    uint16 n_methods;               //类中方法的数目
    Const_Pool *const_pool;         //为该类创建的常数池，表项数目是 cp_size
    Field    *fields;               // 类中的域成员数组；数组大小是 n_fields
    Method       *methods;          // 类中的方法成员数组；大小是 n_methods
      Class_Superinterface *superinterfaces;  //类实现的直接超接口的数组，大小是
                                              //   n_superinterfaces
    const String *name;             //类的字符串名字
    Class    *table_next;           // Class_Table 中下一个 Class
    //接下来的域都是用字节计数的
    unsigned n_instance_refs;       // 是引用（对象或者数组）的实例变量的数目
    unsigned n_static_fields;       //类中静态域的数目
    Method       *static_initializer;       //类中的静态初始化函数，如果不存在，NULL
    unsigned static_data_size;          // 类中静态数据块的大小
    void *static_data_block;            //包含静态域数组的块
    unsigned static_method_size;        //类的静态方法块的字节数
```

unsigned char **static_method_block;　　//包含静态方法的块

VTable *vtable;　　//virtual method table，对接口，NULL

...

}

在 Java 程序执行之前，首先要将 Java 类装载进来，对于每个被装载进来的类，ORP 都会为它创建一个 Class 结构实例，Java 类的信息都存放在这个 Class 实例中，包括超类信息、域成员和方法成员的各种信息。如果程序要使用类中的数据，比如类中的域或者方法，就得通过该类的 Class 结构实例才能得到。

图中 3.1 形象的描述了 Class 结构与其它信息结构之间的关系。其中的 Class_Table 是一个全局对象，对于每个被装载成功的类，该类的 Class 结构实例都会被插入到这个 Class_Table 中。

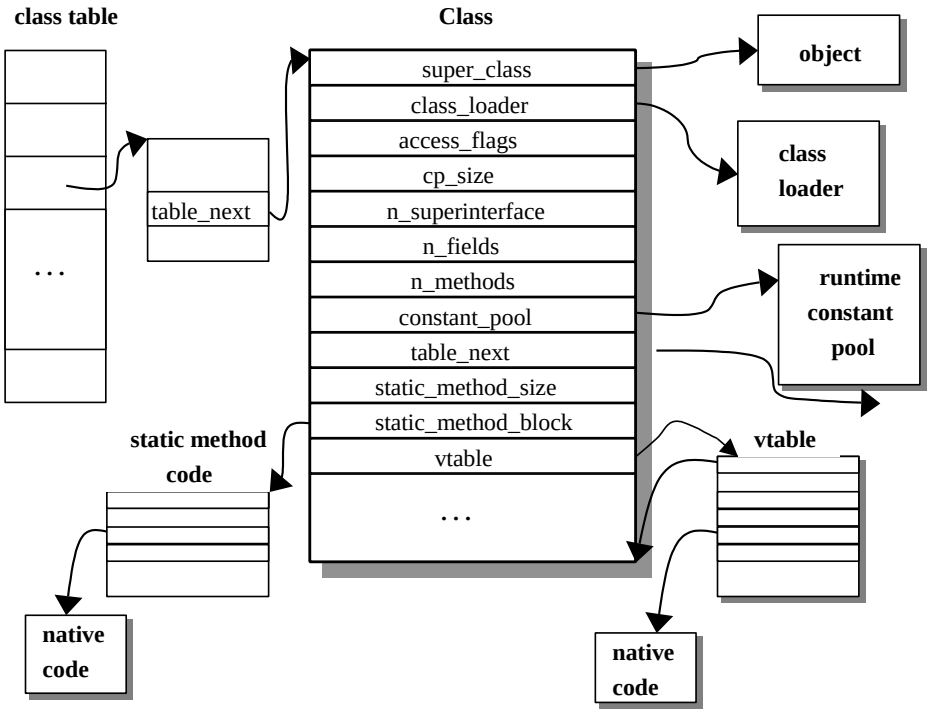


图 3.1 ORP 内部类结构

对于类中的成员，ORP 用类 Class_Member 来作为内部表示，以此为基类，衍生出 Field 类和 Method 类：

class Class_Member { … … }；

class Field ： public Class_Member{ … … }；

class Method ： public Class_Member { … … }；

ORP 分别为 Java 类的域成员和方法成员分别分配了一个 Field 数组和 Method 数组，用于存放关于域和方法的所有信息。在 Class 结构的定义中还有一个非常重要的项就是 Vtable，它是 Java 类的方法表，表中列出了类中除了静态方法之外的所有方法。

3.2.2 Field 数据结构

Field 类结构用于存放从类文件中得到的关于类中域成员的信息。下面列出了 Field 类中一些主要的信息域：

unsigned _offset;　//类成员的偏移量

//对于实例变量，是该变量在实例的数据块中的偏移量

//对于静态变量，是该变量在类的静态数据块中的偏移量

uint16 _access_flags; //访问权限标志，有 ACC_PUBLIC, ACC_PRIVATE,

//ACC_PROTECTED, ACC_STATIC, ACC_FINAL,

// ACC_VOLATILE, ACC_TRANSIENTl 六种

Signature *_signature; //域的名字和类型描述符

Class *_class; //域所属的类

//以下这两个域仅对静态域有效，它们描述了静态域的初始值

uint16 _const_value_index; //初始值的常数池索引

Const_Java_Value const_value; //静态变量的初始值，是由类文件中的

//ConstantValue 属性定义的

3.2.3 Method 数据结构

Method 类结构用于存放从类文件中得到的关于类中方法的信息。下面列出了 Method 结构中的一些主要的域：

State _state; //方法的状态，有 ST_NotCompiled（初始状态），

// ST_BeingCompiled（正在进行即时编译），

// ST_Compiled（已经被编译过）三种状态

void *_code; //方法的入口地址

void *_compile_me_stub; //编译方法的代码段指针

void *_jit_info_block; //JIT 编译器需要的附加信息

unsigned _jit_info_block_size;

JIT_Specific_Info *_jits; //JIT 编译器在编译方法时得到的详细信息

JIT *_jit; //编译方法的 JIT 编译器相关信息

unsigned _index; //该方法在方法表 Vtable 中的索引

uint16 _max_stack; //该方法执行中任何点操作数栈上字的最大个数

```
uint16 _max_locals;          //该方法使用的局部变量的个数

uint16 _n_exceptions;        // 方法可以抛出的“异常“的数目

uint16 _n_handlers;          //字节码中异常处理者的数目

String  **_exceptions;       //方法可以抛出的异常数组

uint32 _byte_code_length;  //该方法的字节码的字节数

Byte *_byte_codes;           //方法的字节码

Handler      *_handlers;     //字节码中异常处理者（exception handler）的数目

struct {
      unsigned is_init              : 1;
      unsigned is_clinit       : 1;

      unsigned is_finalize     : 1;   //是 finalize()方法

      unsigned is_overridden       : 1;   //  是否被子类中的方法覆盖

      unsigned is_nop              : 1;
        } _flags;
```

3.2.4 Intf_Table，Vtable,

Intfc_Table 顾名思义就是接口表，表中列出了类实现的所有直接超接口，表中的每一项记录了接口的 id 和实现这个接口的方法在 Vtable 中的位置。其定义如下：

```
typedef struct {

   unsigned char  **table;   //指向 Vtable 的指针

                             //Vtable 从该指针开始的若干方法实现了相应的接口

   unsigned intfc_id;        //接口的 id

} Intfc_Table_Entry;

typedef struct Intfc_Table {

uint32 n_entries;            //接口的数目

   Intfc_Table_Entry entry[1];    //接口表项数组指针

} Intfc_Table;
```

Vtable 中列出了类中除了静态方法之外的所有方法的代码，当然包括类实现的接口中的方法。

```
typedef struct VTable {
    Class               *clss;          //指向所属的类的指针
    … …
    Intfc_Table         *intfc_table;   //接口表，如果没有接口表，则该项为空
    unsigned char  *methods[1];         //方法的代码
} VTable;
```

Vtable 与 Intfc_Table 之间的关系可以用图 3.2 表示。如果一个变量 intfc 被声明为接口类型 C，可以用该变量指向任何实现了接口 C 的类的实例对象。用 intfc 调用该实例实现接口 C 中的方法 method1 的具体方法时，只需要通过 intfc 查找实例的接口表 intfc_table，找到与接口 C 的 id 相同的接口项，从相应的 table 项中能够迅速的访问方法表中实现接口 C 的方法列表，从而迅速地查找到方法 method1 在实例中的实现方法。

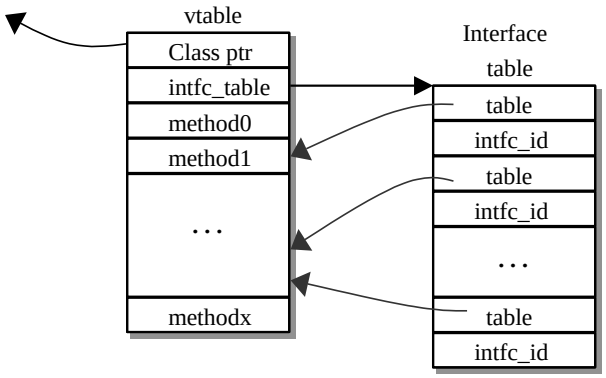


图 3.2 类的虚方法表和接口表

3.3 对象的内部分布

如果 Java 程序中声明了某个类的一个对象（实例），那么，在程序执行时，Java 虚拟机将为这个对象在堆中分配一个空间。但是，这个对象的结构在内存中是如何分布的呢？首先，这个对象必须有空间存放类中的实例数据成员，因为实例数据必须与实例对象关联；其次，它必须能够访问得到它所属的类的 Class 结构，而且为了能够在调用方法时迅速地找到对象动态绑定的方法，ORP 在对象的内部表示中加了一个 Vtable 域，使它指向所属类的方法表；最后，加上一个 32 位的对象的同步信息，ORP 的对象锁机制将利用这个信息实现同步方法对对象的同步访问。这个同步信息具体格式和意义参见 5.3.1 节。

图 3.3 是对象内部表示的示意图。

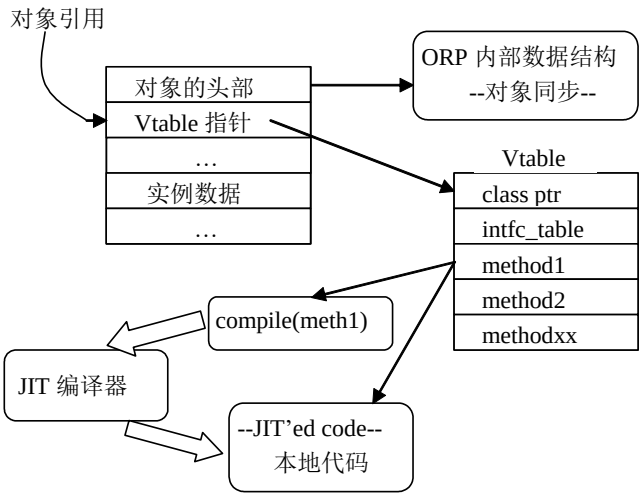


图 3.3 对象的内部分布

Vtable 中的某个方法 method1 的入口地址指向的是一个编译方法本身的例程代码。第一次调用就会执行这段例程编译方法：compile(meth1)，由 JIT 编译器就把这段字节码编译成为机器代码，同时还要修改 method1 方法入口指针，让它指向编译后的机器代码。这样，以后调用 method1 时，就将直接调用方法的机器代码。

第4章 类装载器（Class Loader）

按照 Java 虚拟机规范的要求，类装载器的工作主要分成三个部分：装载（loading），链接（linking）和初始化（initialization），通过这三个步骤，java 程序才可以访问到类中的信息。装载是把类的二进制形式装到 Java 虚拟机中的过程。链接把二进制形式的类数据结合到虚拟机的运行状态中去，可以分为三步：检验（Verification），准备（preparation），和解析（resolution）。检验保证类文件结构上是合法的，但是在 ORP 中检验部分没有实现；准备则为类分配所需的内存；解析把常数池中的符号引用转换成直接引用，在 ORP 中，解析被推迟到符号引用真正使用的时候才进行。检验、准备、解析之后，就可以进行初始化，初始化期间，类变量被赋以合适的初始值。

整个过程的图示如下：

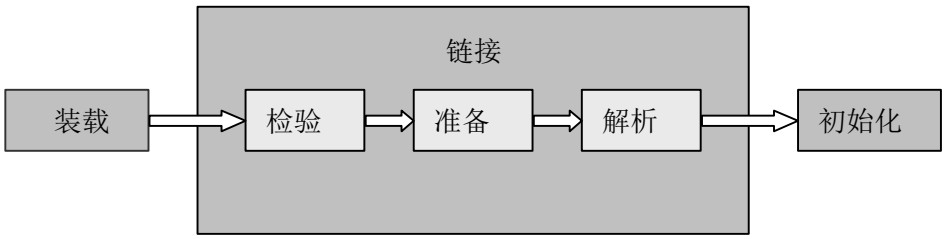


图 4.1　类装载器工作流程图

一般情况下，装载、链接和初始化是以图中所示的顺序来执行的，但是有时候，链接中的解析也可以在初始化之后执行。

4.1 装载（Loading）

按照 Java 程序的执行流程，Java 源程序（*.java）首先被转换成类文件（*.class），因此，要执行 Java 程序，必须把类文件中读进内存中。但是，在装载 Java 程序中定义的类之前，首先要把一些基本的类装进来，比如，java/lang/Class，java/lang/Object，java/io/Serializable，等等，这些类都是 Java API 中的类，关于这些类的信息存放在一个叫做 Global_Env 的类中，Global_Env 的定义如下：

```
class Global_Env {
public:

Mem_Manager&    mem_manager;      //内存管理器

String_Pool          string_pool;    //用于存放用到的一切名字

Signature_Table      sig_table;

Class_Table&    class_table;   //存放装载进的类

Package_Table&     package_table;

char     *classpath;   //类路径

Properties&     properties;
Package *default_package;
┇
```

```
//预装入的类

Class *Boolean_Class;
┆
Class *Long_Class;
Class *Void_Class;
Class *ArrayOfBoolean_Class;
┆
Class *ArrayOfLong_Class;

//Java API 中的类

Class *JavaLangObject_Class;
Class *JavaLangString_Class;
Class *JavaLangClass_Class;
Class *java_lang_Throwable_Class;
Class *java_lang_Error_Class;
Class *java_lang_ExceptionInInitializerError_Class;
Class *java_lang_NullPointerException_Class;
Class *java_lang_ArrayIndexOutOfBoundsException_Class;
Class *java_lang_ArrayStoreException_Class;
Class *java_lang_ArithmeticException_Class;
Class *java_lang_ClassCastException_Class;
Class *java_io_Serializable_Class;
Class *java_lang_Cloneable_Class;
Class *java_lang_Thread_Class;
Class *java_util_Date_Class;
Class *java_lang_Runtime_Class;

//构造器

Global_Env(Mem_Manager& mm, Class_Table& ct, Package_Table& pt, Properties& prop, char *cp);
};
```

另外 ORP 还定义了一个类 ORP_Global_State，定义如下：

```
class ORP_Global_State {
public:
    ORPExport static Global_Env *loader_env;
};
```

程序开始执行的时候，实例化一个 Global_Env 对象，然后就用 ORP_Global_State::loader_env 就指向这个对象。

对于每个装载进来的类，Java 虚拟机都会给它创建一个 Class 对象，把它的信息放进这个 Class 对象中，然后将这个 Class 对象插入 ORP_Global_State::loader_env->class_table 中，类的名字则放在 ORP_Global_State::loader_env ->string_pool 中。

在 ORP 中，为了装载一个类，Java 虚拟机必须做的工作，也就是大致的流程如下：

- 给定类的名字，找到相应的类文件。
- 创建一个表示这个类的 Class 对象。
- 对类文件进行分析，把分析得到的信息放进 Class 对象中。

- 把 Class 对象插入到相应的包中。

在 Java 虚拟机规范中，并没有说明是怎样得到类文件的。ORP 中用到的方式有：

- 从本地文件系统中装载 Java 类文件。
- 从 ZIP，JAR 文件中提取 Java 类文件。
- 从内存中装载 Java 类文件。

这些工作都是由类装载器来完成的。类装载器分为两种：系统类装载器（bootstrap classloader）和用户自定义的类装载器（user-defined classloader）。系统类装载器是 Java 虚拟机实现的一部分，也就是说，假如 Java 虚拟机是用 C 程序实现的，那么系统类装载器就是这个 C 程序的一部分，系统类装载器以缺省的方式装载包括 java API 在内的类，通常是从本地磁盘上装载的，因此，前面所说的那些预装载的类也都是由系统类装载器装载的。而用户定义的类装载器是 java.lang.Classloader 的子类的实例，假如用户定义了自己的类装载器，那么，程序运行时，Java 虚拟机可以把这个类装载器装载进来，然后利用这个类装载器来装载类，比如说，可以从网络上下载类文件。由此可以看出，系统类装载器是 Java 虚拟机实现的固有部分，而用户自定义的类装载器却不是，其实，用户自定义的类装载器是用 Java 语言编写的，编译成类文件，装进虚拟机中，然后像其它对象一样初始化，其实就是正在运行的 Java 程序的可执行代码的另一部分。由于有了用户自定义的类装载器，Java 程序就可以在运行的时候动态地扩展，确定需要哪些类，然后通过一个或多个用户自定义的类装载器装载这些类。因为用户自定义的类装载器是用 Java 语言写的，因此，只要是 Java 语言可以表示的类装载的方式，就可以编写相应的类装载器来装载类，比如说，可以从网络上下载，从某种数据库中提取，等等。

但是，ORP 并不支持多个类装载器，也就是说，在 ORP 1.0.9 版本中，虽然有关于用户自定义的类装载器的一些处理，但是并不会用到它。因此，在这里，我们只是简单的介绍一下用户自定义的类装载器。（如果有一个用户自定义的类装载器，怎样用它来装载类？Class 对象的 class_loader 域怎样设置使它指向一个用户自定义的类装载器？？？）

先介绍系统类装载器。如果用系统类装载器来装载类或者接口 N，首先，在 ORP_Global_State::loader_env->class_table 中查找 N，如果找到了，接着检查该 N 的状态，在 ORP 中，类或者接口的状态定义为：

```
enum Class_State {
  ST_Start,              //初始状态，类或者接口刚被装载进来时，设置为该状态
  ST_LoadingAncestors,   //装载超类或者超接口
  ST_Loaded,             //成功地被装载
  ST_Prepared,           //成功地准备
  ST_Initializing,       //正在被某个线程初始化
  ST_Initialized,        //已经初始化过
  ST_Error,              //由于检查或者准备过程失败或者初始化失败，导致状态错误
  };
```

如果 N 的状态是 ST_LoadingAncestors，说明这个 N 是它自身的超类或者超接口，因此不能够装载，抛出一个 ClassCircularityError。否则，检验 N 是否是一个数组类，如果是，那么就由 Java 虚拟机直接创建这个数组类，而不是由类装载器；如果 N 不是数组类，那么就从文件中装载 N，根据 ORP_Global_State::loader_env->classpath 中提供的目录信息，在相应的目录中查找该类文件，如果找不到，

就会抛出一个 NoClassDefFoundError，如果找到了表示 N 的类文件，对这个类文件进行分析，得到的信息填入为 N 创建的 Class 对象中，并且把该 Class 对象插入到 ORP_Global_State::loader_env->class_table 中，将 N 的状态设置为 ST_LoadingAncestors，之后，如果发生错误，就将该对象从 ORP_Global_State::loader_env->class_table 中删除。为 N 创建的 Class 对象在程序和内部数据结构（方法表，常数池等等）之间充当接口的作用。要访问存储在内部数据结构中的信息，程序就必须调用 N 的 Class 对象的方法。

一个类的装载会引起它的超类也被装载。N 装载成功后，就要查看它是否有超类。如果 N 没有超类，那么它应该是 java/lang/Object，否则，说明 N 有错误，把为 N 创建的 Class 对象从 ORP_Global_State::loader_env->class_table 中删除，并且抛出一个 ClassFormatError。如果有超类，那么现在就要装载这个超类，如果超类的装载不成功，仍然要将这个 Class 对象从 ORP_Global_State::loader_env->class_table 中删除，并且抛出在装载超类时产生的例外。超类的装载虽然成功了，但是也并不能保证它就是正确的，检查这个超类是不是一个接口或者是一个 final 类，如果是，抛出一个 InCompatibleClassChangeError，另外，还要检查 N 是不是一个接口，如果是，那么刚刚装载的超类就应该是 java/lang/Object，否则也会抛出 InCompatibleClassChangeError。通过检查之后，对超类进行准备，如果准备不成功，抛出 InCompatibleClassChangeError。

接下来装载 N 的超接口，过程与装载 N 的超类时一样。

所有的装载成功之后，把 N 的状态设置为 ST_Loaded。至此，N 的装载就完全成功了。

而对于数组类 N，Java 虚拟机首先要检查是否已经装载过一个与 N 的元素类型相同的数组类，如果是，就不必再创建这个数组类了，直接返回已经创建好的数组类即可，否则，为 N 创建一个 Class 对象，根据对 N 的名字（描述符）的分析，把信息填进这个 Class 对象中。如果 N 是一个多维数组，还要将 N 的元素类（去掉 N 的最外层一维得到的类）也装载进来。如果数组 N 的基类（N 的最内层的非数组类）如果不是基本类型（int，byte 等等），那么就是一个引用类型，把这个类装载进来并进行准备。

以上这些操作都没有用到用户自定义的类装载器。如果加上了用户自定义的类装载器，类装载成功后，会把该类插入到装载该类的用户自定义的类装载器中。用户自定义类装载器装载类的基本步骤与系统类装载器相同。

对于每个被装载的类 C，Java 虚拟机都会记录是哪个类装载器（自引导的或者用户自定义的）装载了它，当 C 首次引用另一个类 D 的时候，比如说调用了 D 中的方法，虚拟机要求被引用的类 D 也是用同一个类装载器装载的，这样类装载器返回的类 D 就是动态地与类 C 链接在了一起。因为 Java 虚拟机采用这种方法装载类，缺省地，类只能看到用同一个类装载器装载的其它类，这样，一个类装载器就构成了一个由它所装载的类组成的名字空间。由不同的类装载器装载的类在不同的名字空间，彼此不能访问，除非程序显示地允许。

另外，类的装载必须满足 Java 虚拟机规范中规定的装载限制（loading constraint）。为了描述装载限制，假设用 C 表示类的完整限定名，Ld 表示类的定义类装载器（defining class loader），而 Li（i=1, 2,…）表示类的启动类装载器（initiating class loader），两个类型之间的“＝”表示这两个类型指的是同一个放在方法区中的类。首先介绍一下什么是定义类装载器和启动类装载器。可以这样说，如果一个类装载器 A 被请求去装载一个类，A 又把该任务交给了类装载器 B，B 又委托给了 C，最后 C 真正的装载了这个类，并把这个类返回给 B，B 又返回给 A，那么类装载器 A、B、C 都可以叫做这个类的启动类装载器，而只有 C 叫做这个类的定义类装载器，C 同时是该类的启动类装载器和定义类装载器。下面是装载限制：

- 在解析对类中声明的类型为 T 的域的符号引用时，Java 虚拟机必须生成装载限制：
  $T^{L1}=T^{L2}$ （不管是了 L1 还是 L2 启动了类 T 的装载，结果都应该是同一个对象）
- 解析对类中声明的返回类型为 T_0，参数类型为（T_1, …, T_n）的方法的符号引用时，Java 虚拟机必须生成装载限制：
  $T_0^{L1} = T_0^{L2}, \ldots, T_n^{L1} = T_n^{L2}$
- 重载类中声明的返回类型为 T_0，参数类型为（T_1, …, T_n）的方法，Java 虚拟机必须生成装载限

制：

$T_0^{L1} = T_0^{L2}, \ldots, T_n^{L1} = T_n^{L2}$

4.2 链接（Linking）

4.2.1 检查（Verification）

检查保证类或者接口的二进制表示是正确并且完整的。例如，它检查每一个指令是否有一个合法的操作码；检查每一个分支指令转移到一个某个其它指令的开始处，而不是指令的中间；检查每一个方法都有一个结构上正确的签名（signature）；检查每一个指令遵从 java 编程语言的类型规则。

如果检查期间发生了错误，那么类 LinkageError 的子类 VerifyError 的一个实例将会在程序中引起这个类被检查的点被抛出，表示类或者接口的二进制定义没有能够通过要求的检查以保证它没有违反 JVM 的完整性。

但是在 ORP 中，并没有实现检查。形式上，如果被检查的类有超类，则检查该超类，一层一层向上递归，如果某个超类的装载不正确，整个检查就不会通过。

4.2.2 准备（Preparation）

在 Java 虚拟机装载了类并执行了它预先选择的检查之后，就可以进行类的准备了。在准备阶段，java 虚拟机为类或接口创建静态域，并把这些域初始化为它们的标准缺省值。准备和静态初始化器(static initializer)不同，它不需要执行任何 java 虚拟机代码。

在类或者接口 C 的准备期间，Java 虚拟机会强加装载限制，假设 L_1 是 C 的定义类装载器，对于 C 中声明的方法 m，如果它覆盖了其超类或者超接口中声明的方法，Java 虚拟机会施加如下的装载限制：假设 T_0 是 m 的返回类型名，T_1，…，T_n 是 m 的参数类型名，那么 $T_i^{L1}=T_i^{L2}$ ，（i=0, … , n）。类或者接口的准备可以在其创建之后的任意时间发生，但是必须在初始化之前结束。

另外，准备阶段期间，java 虚拟机也会为可以提高程序执行效率的数据结构分配内存，比如说方法表，其中每一项都是指向类中每个方法的代码的指针，包括从父类继承下来的方法。方法表使得在调用从父类继承下来的方法时不需要在父类中查找，从而可以提高效率。

在 ORP 中，对类或者接口 C 的准备流程大致如下：

1. 如果 C 的状态是 ST_Prepared，ST_Initializing，或者是 ST_Initialized，就说明这个类已经准备过了，返回 LD_OK。
2. 对类的直接超接口逐一进行准备。如果某个直接超接口的准备失败，就返回准备这个超接口时返回的结果。如果某个直接超接口其实不是一个接口，返回 LD_ParseError。
3. 如果 C 是一个接口，则直接初始化 C 的 Class 对象中的某些域。如果 C 不是一个接口，并且 C 有超类，则准备 C 的超类，如果 C 的超类准备的结果不是 LD_OK，返回这个结果，否则，初始化 C 的 Class 对象中的某些域。如果 C 没有超类，那么 C 一定是 java/lang/Object，同样要对 java/lang/Object 的 Class 对象中的某些域初始化。
4. 对 C 中的域赋偏移量。
5. 把类中的实例变量在实例数据块中的偏移地址记录在为 C 创建的 Class 结构中的 gc_information 中，以供 GC 参考。
6. 对 C 中的方法赋偏移量。
7. 把 C 所实现的所有超接口都放在 C 的接口表中。包括 C 的超类实现的所有超接口，和 C 的超接口实现的所有超接口，如果 C 本身也是一个接口，那么把它自身也加到它的接口表中。最后保证 C 的接口表中没有重复的接口。
8. 创建 C 的静态域和静态方法块。
9. 如果 C 是一个接口，那么初始化 C 的静态域，返回 LD_OK。对于接口来说，准备就完成了。
10. 如果 C 不是接口，计算 C 实现的接口中方法的总数目（不包括静态初始化器）；

11. 分配虚表描述符数组的空间。其中每个数组元素都是一个指向虚表中的方法的指针。

15. 最后，把 C 的状态设置成 ST_Prepared，表示已经准备过了。

至此，C 的准备就完成了。

4.2.3 解析（Resolution）

经过链接的前两个阶段（检查和准备）之后，在编译的时候就可以进行链接的第三个也是最后一个阶段：解析。解析就是定位从一个类型的常数池中符号引用的类，接口，域和方法，并用直接引用代替这些符号引用。但是，除非程序使用的符号引用都是第一次使用，这个阶段是可选的。

在类和接口的二进制表示中，对别的类和接口以及它们的域、方法和构造器的符号引用是通过使用这些类和接口的完整限定名来完成的，在使用这些符号引用之前，必须首先对这些符号引用进行解析，验证符号引用，也就是检查域，方法和类的访问权限，检查域或方法是否存在，并且通常会用一个直接引用来代替符号引用，这样，当这个符号引用被频繁使用的时候，可以提高处理效率。

可以对一个已经解析过的符号引用再次进行解析。如果该符号引用被成功地解析过，那么再次对它进行解析时，什么也不用做就可以成功地返回。

如果一个符号引用没有被 JVM 成功地解析，那么以后的解析这个符号引用的尝试都会抛出与最初的解析尝试时抛出的错误同样的错误。

Java 虚拟机指令 anewarray, checkcast, getfield, getstatic, instanceof, invokeinterface, invokespecial, invokestatic, invokevirtual, multianewarray, new, putfield 和 putstatic 都作了对运行时常数池的符号引用。这些指令中的任一个的执行都需要解析它的符号引用。

要进行解析，必须首先介绍常数池。每个类文件结构中都有一个常数池，常数池是一个长度不固定的表，除了表的第一项（constant_pool[0]）保留给 Java 虚拟机内部使用之外，其它的表项表示的都是该类文件结构及其子结构中引用的各种字符串常数，类名，域名，方法名，和其它常数。

在 ORP 中，为每个装载进来的类创建的 Class 结构中都有一个 const_pool 项，用于存放从该类文件的常数池中得到的信息，它是一个数组，数组元素的结构如下：

```
union Const_Pool {
    unsigned char    *tags;              //只有常数池的第 0 项才有
    uint16 name_index;                   //该常数池项的标记是 CONSTANT_Class
    uint16 string_index;                  // CONSTANT_String
    struct {                              // CONSTANT_{Field,Method,InterfaceMethod}ref
        uint16 class_index;
        uint16 name_and_type_index;
    } cp_tag;
    uint32          int_value;          // CONSTANT_Integer
    float           float_value;        // CONSTANT_Float
    uint32      low_bytes;
    uint32      high_bytes;          // CONSTANT_{Long,Double}
    struct {                          // CONSTANT_NameAndType
        uint16 name_index;
        uint16 descriptor_index;
    } ni_di;
    String *string;                 // CONSTANT_Utf8
    //  解析过之后，常数池中的项就是下面几种形式
    Signature *signature;       //  对于 CONSTANT_NameAndType
    Class           *clss;              // CONSTANT_Class
    Field       *field;             // CONSTANT_Fieldref
```

```
    Method    *method;        // CONSTANT_{Interface}Methodref
};
```

其中 tags 只出现在常数池的第 0 项中，是一个无符号字符的数组，每一项对应于常数池中相应项的标记，就是说，tags[i]就对应于 constant_pool[i]项的标记。

常数池中的项有以下几种标记：

```
enum Const_Pool_Tags {
CONSTANT_Class              =7,    //类
CONSTANT_Fieldref           =9,    //域
CONSTANT_Methodref          =10,   //方法
CONSTANT_InterfaceMethodref =11,   //接口方法
CONSTANT_String             =8,    //String
CONSTANT_Integer            =3,    //4 字节整数
CONSTANT_Float              =4,    //4 字节浮点数
CONSTANT_Long               =5,    //8 字节整数
CONSTANT_Double             =6,    //8 字节浮点数
CONSTANT_NameAndType        =12,   //域或者方法
CONSTANT_Utf8               =1,    //常数字符串
CONSTANT_Tags               =0,    //标记 tags 数组
};
```

下面的几个部分描述了在类或接口 D 的运行时常数池中解析符号引用的过程。解析的符号引用的种类不同，解析的细节也不同。

- 类或接口的解析

要解析 D 对类或者接口 C 的未解析的符号引用，就要执行下面的步骤：

1. 如果 D 有自己的类装载器（class loader），则用该类装载器装载类或者接口 C。否则用系统的装载器装载 C。如果装载 C 的过程失败，就把类装载失败时抛出的例外作为类解析失败的结果抛出。
2. 对 C 进行准备（prepare）。
3. 检查 D 对于 C 的访问权限。如果 C 被声明为 public，或者 C 就是 D 本身，那么 D 可以访问 C，设置 C 为已经解析过，并成功返回。
4. 检查 D 是否与 C 在同一个包中，如果是，设置 C 为已经解析过，并成功返回，否则，返回 IllegalAccessError。

- 对数组类的解析

如果数组类 C 的成员是引用类型，那么，解析对 C 的成员类型 D 的符号引用。

- 域的解析

如果 D 引用了 C 中声明的一个还没有被解析的域，那么首先要对 C 的符号引用进行解析，如果 C 解析失败，就抛出解析 C 时产生的例外，也就是说类解析失败的结果也可以作为域解析失败的结果被抛出。如果可以成功的解析对 C 的引用，之后抛出的例外就是与域解析失败相关的，而不是与类解析失败相关。

成功地解析了 C 之后，首先要在 C 和它的超类中查找被引用的域：

1. 如果 C 声明了一个域，该域的名字与描述符与这个域引用指定的名字和描述符相同，那么查找成功。被声明的域就是域查找的结果。
2. 否则，在 C 的超类中递归地查找该域。
3. 否则，域查找失败。

如果域查找失败，域解析就会抛出一个 NoSuchFieldError。如果域查找成功了，还要检查该域的访

问权限，如果 D 不能访问该域，域解析就会抛出一个 IllegalAccessError 例外。

最后把该域设置为已经解析过了。

- 方法解析

如果 D 引用了类或者接口 C 中的一个方法，而且该引用还没有被解析，就会采取下列步骤：

1. 首先解析声明了该方法的类或者接口 C。如果 C 解析失败，那么 C 解析过程中抛出的例外就作为方法解析的结果抛出。
2. 由于 CONSTANT_MethodRef 必须指向类而不是接口，如果声明了该方法的 C 是接口，就抛出一个 NoSuchMethodError，而 CONSTANT_InterfaceMethodRef 必须指向接口，如果声明了该方法的 C 不是接口，也抛出一个 NoSuchMethodError。
3. 在 C 和它的超类中查找被引用的方法：
   - 如果 C 声明了一个方法，该方法的名字和描述符与这个方法引用规定的一样，那么方法查找成功。
   - 否则，在 C 的超类或者超接口中递归地查找该方法，如果 C 的任何一个超类或者超接口声明了这个方法（名字与描述符与被引用的方法的名字与描述符相同），方法查找成功。
   - 否则，方法查找失败，抛出一个 NoSuchMethodError 例外。

   检查 D 对于该方法的访问权限。如果 D 不能访问该方法，抛出一个 IllegalAccessError
4. 如果解析成功，在常数池中把该方法标记为已解析。

- 接口方法解析

对接口方法的解析，基本上与上面的方法解析类似，首先解析该方法（与上面的步骤相同），如果解析成功，再检查该方法是否是静态的，如果是，则抛出一个 IncompatibleClassChangeError 例外，因为接口中的方法隐式地是抽象的，可以被实现，而静态则意味着不能够被覆盖，也就是不能够被实现或使用。

- 访问控制

在上面的解析中都会用到访问权限的检查，也就是说类或者接口 D 是否可以访问类或接口 C。D 可以访问 C，当且仅当满足下列条件之一：

- C 是公共的（public）。
- D 就是 C 本身。
- C 和 D 是同一个运行时包（package）的成员。

类或接口 C 中的域或者方法 R 对类或接口 D 是可访问的，当且仅当下列条件中的任一个成立：

- R 是公共的（public）。
- D 就是 C 本身。
- R 是被保护的（protected）并且在类 C 中声明，而 D 是 C 的子类或者就是 C 本身。

4.3 初始化（Initialization）

在类和接口的装载与链接时间的选择上，Java 虚拟机规范虽然给了虚拟机实现一定的灵活性，但是却严格的规定了初始化的时间。所有的 Java 虚拟机实现都必须在类或者接口的第一次活跃使用（active use）时初始化。下面的六种情况称之为活跃使用：

1. 创建一个类的新对象（在字节码中，就是 new 指令的执行）。
2. 调用类中声明的静态方法（在字节码中就是 invokestatic 指令的执行）。
3. 类中声明的一个非常量的静态域被赋值或者被使用（对应于 getstatic 和 putstatic 指令）。常量域都是（显示地或隐式地）final 和 static，并且是用一个编译时的常量表达式初始化的。对这样的域的引用必须在编译的时候解析成这个编译时的常量值的拷贝，因此使用这样的域绝对不会引起初始化。
4. 对 Java API 中某些反射(reflective)方法的调用，比如类 java/lang/Class 或者 java.lang.reflect 包中的类中的方法。

5. 对类的子类进行初始化。一个类被初始化之前，要求它的超类先被初始化。
6. Java 虚拟机启动时，把一个类指定为初始类（有 main()方法）。

除了上面这六种情况，其它的对类的使用都叫做被动使用（paasive use），不会触发类的初始化。

类被初始化之前，要求它的超类先被初始化，初始化它的超类时，又要求它的超类的超类先被初始化，如此递归下去，那么在初始化这个类之前，就会要求它的所有的超类先被初始化。但是这对于接口并不成立。接口被初始化只是因为接口中声明的非常量域被使用了，而决不是因为实现了这个接口的子接口或者类需要被初始化。因此，类被初始化之前，它的超类必须被初始化，但是类所实现的接口则不必被初始化。类似的，一个接口的超接口在接口被初始化之前也不必被初始化。

类或者接口的第一次活跃使用时，它必须被初始化，而在它初始化之前，它必须已经链接过，而在链接之前，它必须被装载过。因此，Java 虚拟机实现可以提前装载并且链接类或者接口，不需要等到第一次活跃使用时才做这些工作。如果第一次活跃使用时，类或者接口还没有被装载或者链接，那么就应该马上装载并且链接它，然后进行初始化。

类的初始化由在类中声明的静态初始化函数和静态域的初始化器的执行组成。接口的初始化由在接口中声明的域的初始化器的执行组成。

因为 java 语言是一个多线程的语言，因此，对类或者接口的初始化就要求严格的同步，因为在同一个时刻可能有另一个进程也要初始化同一个类或者接口。也有可能，对类或者接口的初始化作为该类或者接口的初始化的一部分被递归地请求，例如，类 A 中的变量初始化器可能会调用类 B 中的一个方法，而 B 也调用了 A 的一个方法。

在初始化之前，要初始化的类或者接口（Class 对象）已经进行检验和准备过。

初始化类或者接口的步骤如下：

1. 在表示要被初始化的类或者接口的 Class 对象上进行同步，也就是等到当前线程可以获得该对象的锁。
2. 如果另外一个线程正在对这个类或者接口进行初始化，那么就在这个 Class 对象上等待（wait），当当前线程从等待中唤醒时，重复执行该步骤。
3. 如果当前线程正在对这个类或者接口进行初始化，那么这一定是对初始化的递归请求，释放该 Class 对象上的锁，然后正常结束。
4. 如果这个类或者接口已经被初始化过了，就不需要进一步的操作，释放该 Class 对象上的锁，正常结束。
5. 如果这个 Class 对象的状态错误，就不会进行初始化。释放 Class 对象上的锁，然后抛出 NoClassDefFoundError 例外。
6. 否则，记录当前线程正在对这个 Class 对象进行初始化这个事实，释放这个 Class 对象上的锁。
7. 接下来，如果这个 Class 对象表示的是类而不是接口，并且这个类的超类还没有被初始化，那么对其超类递归地执行整个步骤。如果必要的话，先对超类进行检查和准备。如果对超类的初始化因为抛出一个例外而非正常结束，那么锁定这个 Class 对象，并把它标记为错误，通知所有正在等待的线程，释放这个锁，然后抛出初始化它的父类时抛出的例外，终止。
8. 初始化类的 final 变量和接口的域。因为它们的值是编译时常量，所以先被初始化。接着，按照文本顺序，执行类变量初始化器和类的静态初始化器，或者执行接口的域初始化器。但是静态初始化器和类变量初始化器不能够引用文本上在其之后声明的变量。
9. 如果初始化器的执行正常结束，那么锁定这个 Class 对象，把它标记为已经初始化过，通知所有正在等待的线程，释放这个锁，然后正常结束这个过程。
10. 否则，初始化器一定是抛出了某个例外 E 而非正常结束。如果 E 的类不是 Error 或者 Error 的一个子类，那么创建类 ExceptionInInitializerError 的一个实例，用 E 作为参数，在接下来的步骤中，用这个对象代替 E。但是，如果出现了 OutOfMemoryError 从而不能创建 ExceptionInInitializerError 的一个实例，那么就在接下来的步骤中用一个 OutOfMemoryError 对象代替 E。

11. 锁定这个 Class 对象，标记为错误，通知所有等待线程，释放这个锁，并用 E 或者在前面的步骤中确定的 E 的代替来终止这个过程。

第5章 ORP 的线程与同步(Synchronization)

Java 是一种支持并发的程序设计语言，在语言级就提供了支持多线程的特性。Java 中所有的对象都是由锁保护的，因此同步在整个系统中占有十分重要的地位。现有 Java 系统的基准测试中，锁性能直接影响了虚拟机的性能。这一章，我们首先在来看一下 Java 中的线程在 ORP 中的实现，然后再讨论 ORP 中的同步。我们将结合 ORP1.0.9 中所包含的锁的两个版本的实现来介绍。

5.1 ORP 中的线程

所谓线程是指进程中独立的程序执行路径，它本身具有进程的动态性，在同一进程的各个线程之间又具有共享资源特性。因此在支持并发特性上，这种机制能够更好的体现多 CPU 带来的优势，共享资源的各个线程能够同时在多个 CPU 上得到执行。

ORP 实现的线程在虚拟机中表现为 Java.lang.Thread ，而本身实现是建立在操作系统提供的线程机制基础上抽象的。在实现上，Java.lang.Thread:ORP 线程:系统线程之间是按照 1:1:1 的模型来组织的。下面，我们来说明 Java.lang.Thread 以及 ORP 线程的实现。

5.1.1 Java.lang.Thread

JVM 规范中说明了创建 Java.lang.Thread 对象以及它的派生对象是生成 Java 线程的唯一方法。创建后，这个线程并不是活动的，只有当执行了它的 start 方法后才是活动线程。

在 Java 中，Java 线程属于和系统关联比较紧密地部分，因此在 classpath 中把这个部分的实现交给核心虚拟机来完成。（其他的还有 Java.lang.Class,Java.lang.Runtime,Java.lang.Throwable, Java.lang.reflect.Constructor, Java.lang.Reflect.Field,Java.lang.Reflect.Method）。

在开始介绍 Java 线程前，我们先来看一下 ORP 中对它的定义：（参考 orp/base_natives/gnu_classpath/include/java_lang_Thread.h,）：

```
typedef struct Classpath_Java_java_lang_Thread {
VTable *vt;     //和所有的 Java 对象一样，第一项永远是 vtable;
POINTER_SIZE_INT obj_info;     //指向对象信息的指针
Classpath_Java_java_lang_ThreadGroup *group;  ??//当前 Java 线程属于的线程组(ThreadGroup)
Classpath_Java_java_lang_Runnable *toRun;??//运行的方法
Java_java_lang_String *name;                    ??//当前 Java 线程对象的名字
long daemon;                                    //是否属于守护线程
long priority;                                  //线程优先级
// contextClassLoader field is not a formal field temporarily
Java_java_lang_Object *contextClassLoader;      //类装载器，似乎并没有使用
long data;                                      ??//未使用
};
```

从上面的定义中看到了 Java 线程的优先级，和通常操作系统提供的线程一样，是为了实现不同的调度策略而提出的。在 ORP 中，把 Java 线程的优先级定义了 10 级，最低为 1，通常为 5，最高为 10。一般的高优先级的线程将优先得到执行权。

ORP 中定义了 Java 线程的一些重要的功能函数(完整的实现可以参考 gnu_classpath 目录下的 java_lang_Thread.cpp)：

```
void java_lang_Thread_sleep(Java_java_lang_Thread *p_this, int64 msec, int32 nsec)
void java_lang_Thread_start(Java_java_lang_Thread *p_this)
```

void java_lang_Thread_interrupt(Java_java_lang_Thread *p_java_thr)

void java_lang_Thread_yield (Java_java_lang_Thread *not_used)

void java_lang_Thread_join(Java_java_lang_Thread *p_java_thr, int64 timeout, int32 nanos)

这些函数分别实现了 Java 线程的 sleep，start,interrupt,yield,join 方法。我们下面针对各个方法，从功能和内部实现上作介绍：

- sleep 方法

使当前线程停止执行，进入睡眠。有重载的两种方法，分别为

public static void **sleep**(long millis)　　//毫秒数

public static void **sleep**(long millis,int nanos) //毫秒数，纳秒数

在 bytecode 内对 Java 线程的 sleep 方法的调用都将通过 java_lang_Thread_sleep 来进行。

内部实现上，对该方法的调用将直接传送给 ORP 线程的处理方法 java_lang_Thread_sleep_generic，我们将在 ORP 线程部分对这个部分作介绍。

- start 方法

使当前线程开始执行而成为活动线程，虚拟机会调用这个线程的 run 方法开始执行。在内部实现上，该方法将直接调用 ORP 线程的 java_lang_Thread_start_generic 方法。

- interrupt 方法

中断当前线程，在没有发生什么特殊情况下，将设置当前 Java 线程的中断标志。在内部实现上，对它的调用将直接调用 ORP 线程的 java_lang_Thread_interrupt_generic 方法。

- yield 方法

yield 方法的调用将暂停当前 Java 线程的执行而允许其他的 Java 线程来执行。功能与 unix 环境下的 yield、linux 下的 sched_yield 相同，主动放弃属于该线程对 CPU 的使用权。

在内部实现上，linux 系统下最终将转换成对 sched_yield 的调用；Windows 系统下将直接调用系统提供的 API 函数 SleepEx 的调用。

- join 方法

在本线程的执行路径上等待程序指定的对象线程结束，也有两种重载方法：

public static void **join**(long millis) //如果参数为 0，将一直等待

public static void **join**(long millis,int nanos) //毫秒数，纳秒数

在内部实现上，如下代码所示：

```
void java_lang_Thread_join(Java_java_lang_Thread *p_java_thr, int64 timeout, int32 nanos)
{
    volatile Java_java_lang_Thread *p_obj_volatile = p_java_thr;
…
    orp_monitor_enter( (Java_java_lang_Object *)p_obj_volatile);

    if (java_lang_Thread_isAlive( (Java_java_lang_Thread *)p_obj_volatile) )
        java_lang_Object_wait( (Java_java_lang_Object *)p_obj_volatile, timeout);

    orp_monitor_exit( (Java_java_lang_Object *)p_obj_volatile);
…
}
```

这儿调用了该 Java 线程对象的 wait 方法，这是一个对象的方法，它可以使执行该方法的当前线程进入睡眠，直到被通知或是被中断。在调用该方法时，一定要首先获得该对象的锁，在 wait 方法实现内部会释放对锁的使用。

在开始的时候说到了每个对象都有锁，与锁相对应的每个对象还有一个等待集(Wait Set)，用等待集来表示希望取得这个对象资源的线程集合。很显然，当一个对象创建的时候，等待集一定是空的。等待

集的提出是为了提高系统调度器的效率。对应的，Java 的每个对象都有如下方法：wait,notify,notifyall。分别对应着将线程放入等待集、唤醒一个、所有的在该对象上的线程。wait 的调用将把当前线程将把自己暂停执行，因为它需要的资源不可用，它将在这个资源被释放的时候得到通知。这通常可以使以下几种情况：其它的线程调用了该线程等待对象的 notify 方法，而刚好是该线程被选中；其他的某个线程调用了该对象的 notifyAll 方法。操作系统中的 PV 操作在 Java 中的反应，这可以用来解决类似于生产者-消费者问题,这类问题的解决更依赖于应用的设计者，与此相区别的互斥式问题将在 3.2 中详细讨论。

Java 线程作为 Java 语言支持并发特性的代表，它的调度是由 Java 虚拟机基于当前 Java 线程的优先级信息以及运行状态来进行的。在 JLS 中声明的方法和 Java 线程状态之间的转换关系如下图所示：

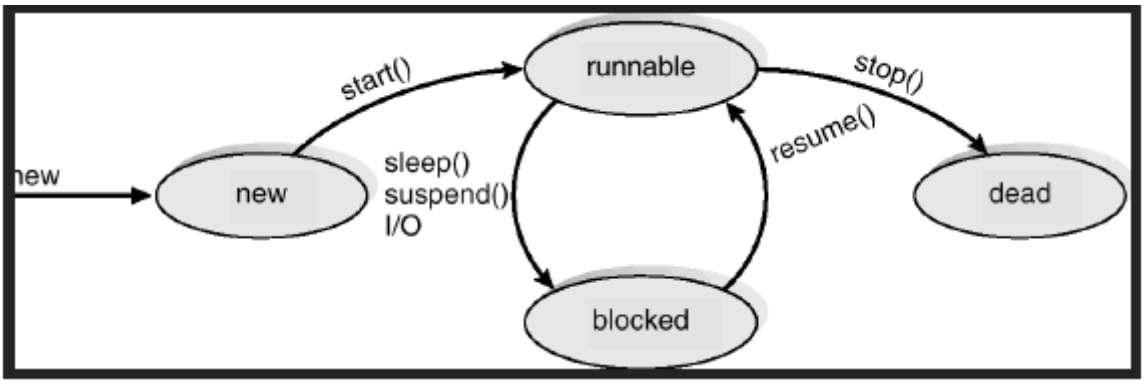


图 x-x: Java 线程状态迁移

在 ORP 中定义了如下的一些状态：

```
thread_is_sleeping          //当前 Java 线程正在睡眠，对应于图 x-x 中的阻塞(blocked)状态
thread_is_waiting,          //当前 Java 线程在等待调度，对应于图 x-x 中的 runnable 状态
thread_is_birthing,         //当前 Java 线程刚创建，对应于图 x-x 中的 new 状态
thread_is_running,          //当前 Java 线程正在运行
thread_is_dying,            //当前 Java 线程准备销毁，对应于图 x-x 中的 dead 状态
```

后面我们会看到，实际上 ORP 的实现中并没有对 Java 线程进行线程调度，而是直接交给本地的操作系统，由操作系统来对 Java 线程(执行 Java 代码的本地线程，这中间经过了一次 ORP 线程的封装)进行调度的，但 ORP 跟踪当前 Java 线程的执行状态。

5.1.2 ORP 的线程

ORP 的线程抽象是为了简化 Java 线程在 Linux/Windows 系统上的实现而设计的。在上面关于 Java 线程方法的实现中，我们已经简单的看到了一下它的部分使用，实际上 Java 线程的各种方法实际上都是通过 ORP 对应的线程操作来完成的。

我们首先来看一下基本的 ORP 线程的数据结构：(参考 orp\common\include\orp_threads.h)

```
class ORP_thread {
……
volatile    Java_java_lang_Thread    *p_java_lang_thread; //指向当前 ORP 线程代表的 Java 线程
volatile    Java_java_lang_Object    *p_current_object;    //线程当前需要同步的对象
volatile    Java_java_lang_Object    *p_exception_object; //当前 Java 线程的异常对象
    ……
java_state                app_status;                //当前 Java 线程的状态
    gc_state                    gc_status;                    //当前垃圾收集状态
……
```

```
    bool                        interrupt_thread_api_support; //纪录 Java 线程的 interrupt 方法执行结果
        bool                        interrupt_a_waiting_thread;   //纪录 Java 线程是否是中断了一个等待
线程
        bool                        thread_is_java_suspended;     //线程是由 Java 线程的方法暂停的
    Registers                   regs;                         //当前线程的执行上下文
    J2N_Saved_State *           last_java_frame;              //最近 Java 帧，在第 X 章第 X 节我们会讨论
    ……
    #if defined(ORP_NT)
    HANDLE                      event_handle_monitor; //与监视器相关的事件句柄
    HANDLE                      event_handle_sleep;   //与睡眠相关的事件句柄
    HANDLE                      event_handle_interrupt; //与中断相关的事件句柄
    HANDLE                      event_handle_suspend0;//与暂停相关的事件句柄
    HANDLE                      thread_handle;          //线程句柄
    HANDLE                      gc_resume_event_handle;//垃圾收集恢复线程相关的事件句柄
    DWORD                       thread_id;              //线程号

    #elif defined (ORP_POSIX)
    //下面是 posix 下对应的域
    int                         event_handle_monitor;
    int                         event_handle_sleep;
    int                         event_handle_interrupt;
    int                         event_handle_suspend0;
    int                         thread_handle;
    int                         gc_resume_event_handle;
    int                         thread_id;
    ……
    #endif
    ……
    #ifdef OBJECT_LOCK_V2
    int thread_index;
    int notify_recursion_count;
    HANDLE    event_handle_notify_or_interrupt;
    unsigned short stack_key;
    #endif
    ……
    };
```

在这里面，与监视器(monitor)、睡眠(sleep)、中断(interrupt)、暂停(suspend)、恢复(resume)相关的事件句柄是在 Windows 下使用的。它们都是用了相同的实现技巧，在分别的场合下实现从就绪状态到等待状态的转换：

如对 event_handle_sleep 的使用，在进入等待状态前，首先先把 event_handle_sleep 复位成为非触发态:

```
    ……
    ResetEvent(p_TLS_orpthread->event_handle_sleep);
    ……
```

然后调用 WaitForSingleObject 来进行等待：

```
stat = WaitForSingleObject(p_TLS_orpthread->event_handle_sleep,
                                                    (unsigned long)msec);
….
```

如果这段时间内 event_handle_sleep 没有被设置成为触发态，WaitForSingleObject 将在这个执行点上进入等待状态，直到等待时间超时返回，在这段时间内这个线程会因为处于等待状态而不参与操作系统的调度，而对应的 Java 线程处于睡眠状态。我们会在下一节中介绍这些操作系统相关的调用的实现。

5.1.3 操作系统线程

ORP 线程最终都是映射到系统线程上的，我们在这一节介绍统一的线程部分接口的实现。Windows 和 Linux 本身都提供了 1:1 的线程实现，即每个用户层的线程都有一个对应的内核调度实体。这区别于类似于 MIT 线程库的 M:1 实现，也区别于类似 Solaris 下 M:N 线程的实现。关于线程设计实现的更多细节可以参考书目[1]中第 5 章关于线程的介绍。

Windows 下的线程有了良好的封装，使用比较简单，已经为很多 Windows 的开发者接受。Linux 下的线程是由 GLIBC 中关于 linuxthreads 的 pthread 软件包实现的。ORP 为了共用更多的代码，特别实现了类似于 Windows 下线程的开发例程。下面我们首先介绍 ORP 中使用的 Windows 下关于线程控制部分的例程，然后我们来看一下 Linux 下线程的相关例程的，最后介绍 Linux 下的线程相关的 API 封装。

Windows 的线程相关例程

线程总是和调度相关的，因此在介绍相关的函数前，我们先来看一下 Windows 下线程调度中的状态转换：

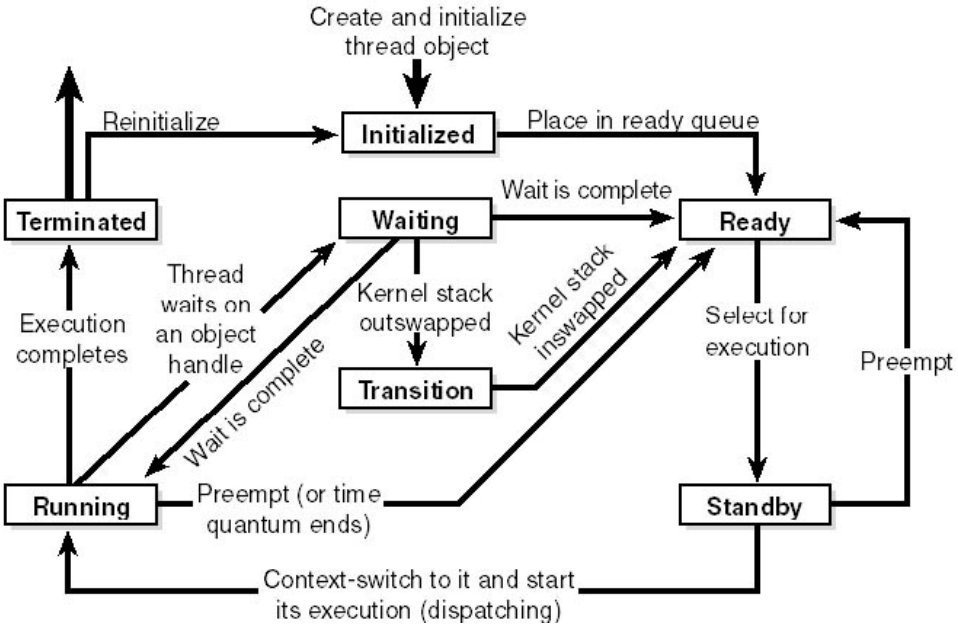


图 xx-xx: Windows 下的线程状态转换

从图 xx-xx 中，我们可以看到，Windows 下线程的状态一共有 7 种，分别是初始化、终止、运行、等待、就绪、传输和备用。一个运行的线程表示正在使用处理器；一个备用的线程表示正要使用一个。一个就绪的线程表示想使用处理器，但是由于没有空闲处理器，该线程必须等待。一个在传输中的线程表示正在等待一个资源以便执行，比如等待自己的执行堆栈以便在磁盘上分页。由于正在等待完成一个附属操作或资源之后才能空闲，一个处于等待的线程不使用处理器。

创建线程的函数原型：

该函数接收如下六个参数，执行完这个函数，如果参数正常的话，系统中就会创建一个从 start_address 开始的程序执行路径的线程。该线程的当前状态由 initflag 指定：如果为 0，线程就会运行；如果是

CREATE_SUSPENDED，则会是暂停状态。

unsigned long _beginthreadex(void *security, //线程对应的安全描述符，通常是对应线程的权限控制
unsigned stacksize, //线程的堆栈大小
unsigned(__stdcall *start_address)(void *), //线程的起始执行例程
void *arglist, //传递给线程起始例程的参数指针
unsigned initflag, //线程创建的标志
unsigned *thrdaddr //保存线程号的地址
);

销毁线程的函数原型：

与此相对应的，retcode 是线程的返回值。调用这个例程的线程会把自己销毁，并且返回 retcode 的值作为线程的退出代码。

void _ _cdecl _endthreadex (unsigned retcode) ;

(注：实际上，在_beginthreadex/_exitthreadex 内部，是 C/C++运行库对 WindowsAPI 函数 CreateThread/ExitThread 的封装，这是因为 C/C++运行库中有些例程并不是线程安全的。更详细的材料可以参考 Jeffrey Richter's Programming Applications for Microsoft Windows，以及 Visual C++所携带的部分的源代码。)

挂起/恢复线程的函数原型：

DWORD SuspendThread(HANDLE hThread);

DWORD ResumeThread(HANDLE hThread);

这两个例程接受一个线程的句柄，执行后将挂起/恢复这个句柄所对应的线程。

休眠函数原型：

VOID Sleep(DWORD dwMilliseconds);

VOID SleepEx(DWORD dwMilliseconds,BOOL bAlertable);

参数指定的是休眠的毫秒数。对它的调用将引起本线程放弃当前的时间片，线程也会进入等待状态，直到等待时间结束。Linux 下提供的 sleep 是以秒计的。

取得/设置线程上下文函数：

BOOL GetThreadContext(HANDLE hThread,
PCONTEXT pContext);

BOOL SetThreadContext(HANDLE hThread,
CONST CONTEXT *pContext);

线程的上下文是指线程的当前执行状态，为了执行这两个例程，首先需要把该线程挂起。否则，取得的内容是不确定的。在 IA32 上，线程上下文可以包括很多内容，可以包括各个通用寄存器、段描述符、浮点寄存器、各个调试寄存器等。这主要是由 CONTEXT 结构来描述的，关于这个结构的详细内容可以参考 MSDN 的相关文档。

提高线程，一般也离不开同步。我们下面也介绍一下 ORP 中使用到的 Windows 下的同步例程。

临界区保护

临界区是需要对一些共享资源独占使用的一段代码，它保证在任何时候只有一个线程能够进入该区域。Windows 下提供了 CRITICAL_SECTION 类型及其相关例程来支持临界区。这方面的例程有：

VOID InitializeCriticalSection(PCRITICAL_SECTION pcs); //初始化一个临界区
VOID DeleteCriticalSection(PCRITICAL_SECTION pcs); //复位一个临界区
VOID EnterCriticalSection(PCRITICAL_SECTION pcs); //进入临界区
BOOL TryEnterCriticalSection(PCRITICAL_SECTION pcs); //探测临界区状态
VOID LeaveCriticalSection(PCRITICAL_SECTION pcs); //离开临界区

在使用临界区前，首先要调用初始化例程对它进行初始化；初始化后，临界区没有绑定到任何线程，当某个线程调用进入临界区例程后，临界区内的数据结构就会更新，以不允许其他的线程再进入临界区。

任何其他线程调用这个例程都将引起线程进入等待状态（绑定的缺省超时的是 2592000 秒，大约 30 天，几乎不会超时；如果超时，将抛出异常），自身再次调用该例程会增加本身对临界区使用的计数，进入与离开例程必须要执行相同多数目次，要不然其他的线程将永远不能进入该临界区。探测临界区例程会检查临界区内的当前状态，如果当前没有其他线程在临界区就立即返回 TRUE 表示可以进入，否则立即返回 FALSE 表示不能进入，并不会使调用线程进入等待状态。在临界区内的线程调用离开临界区例程将对临界区内的计数减一，如果计数大于 0，就简单的返回；如果计数等于 0，该例程会检查是否有其他的线程正在等待这个临界区，如果有就更改临界区的成员数据并使某个线程再次调度；如没有就更改成员数据表示没有线程在占用这个资源。临界区使用完后，可以使用复位例程来清除 CRITICAL_SECTION 数据。注意：临界区内的计数增加、减少都是原子操作，因此是多处理器安全的。

事件保护

事件是 Windows 内核提供的一种常用的进程间通信的机制。每个事件都有两个状态，触发和非触发状态；对于事件的状态调用等待例程就可以达到同步的效果。

```
HANDLE CreateEvent(                          //创建事件对象
  PSECURITY_ATTRIBUTES psa,//事件对应的安全属性
  BOOL fManualReset,              //是否手动复位
  BOOL fInitialState,             //初始化状态
  PCTSTR pszName);                //事件名称
BOOL SetEvent(HANDLE hEvent);     //设置事件为触发态
BOOL ResetEvent(HANDLE hEvent);   //设置事件为非触发态
DWORD WaitForSingleObject(        //等待单个对象
  HANDLE hObject,
  DWORD dwMilliseconds);
```

对一个在非触发态的事件的调用等待例程将引起调用线程进入等待状态，线程会等待直到超时或是等待的事件对象状态为触发态。一般的同步使用方式如下：

```
hEvent=CreateEvent(…);      //创建一个自动复位事件对象

… TheadFunc(…)
{
WaitForSingleObject(hEvent,…);
//等待进入临界区,由于是自动事件对象，一个线程进入后将自动复位
…
}
…
```

在 ORP 中，对应于 ORP 线程的监视器、休眠、中断等事件对象都是自动复位对象。等待例程也常用来等待线程或者进程终止，在 Windows 下，每个线程/进程句柄也有对应的状态，线程/进程活动的时候，该句柄对应的状态是非触发态；如果线程/进程结束，则该句柄对应的状态就是触发态。Windows 本身提供了丰富的同步对象，关于它的更多更详细的论述，可以参考书目[x]。

Linux 的线程相关例程

Linux 下实现了 POSIX 兼容的 pthread[9]，我们在这儿简单介绍一下在 ORP 中使用到的一些相关例程：

```
int     pthread_create(pthread_t *, const pthread_attr_t *,void *(*)(void *), void *);     //创建一个线程
pthread_t     pthread_self(void);                  //返回当前线程号
int     pthread_mutex_init(pthread_mutex_t *, const pthread_mutexattr_t *);//初始化互斥量
int     pthread_mutex_lock(pthread_mutex_t *);    //锁定互斥量
```

int pthread_mutex_trylock(pthread_mutex_t *);//探测互斥量状态
int pthread_mutex_unlock(pthread_mutex_t *);//释放互斥量

int pthread_attr_init(pthread_attr_t *);//初始化线程属性对象
int pthread_attr_setdetachstate(pthread_attr_t *, int);// 设置线程属性对象内的线程的分离状态，可以选择 PTHREAD_CREATE_DETACHED 或者 PTHREAD_CREATE_JOINABLE

int pthread_key_create(pthread_key_t *, void (*)(void *));
//该函数在进程内创建一个对所有线程都可见的特定于线程的数据关键字
void *pthread_getspecific(pthread_key_t);//取得某个数据关键字关联的局部存储
int pthread_setspecific(pthread_key_t, const void *);//设置某个数据关键字关联的局部存储
int pthread_key_delete(pthread_key_t);//删除数据关键字

int pthread_cond_init(pthread_cond_t *, const pthread_condattr_t *);//初始化条件变量
int pthread_cond_timedwait(pthread_cond_t *, pthread_mutex_t *, const struct timespec *);
//条件变量的时限等待
int pthread_cond_broadcast(pthread_cond_t *);//唤醒所有的由条件变量阻塞的的线程
int pthread_cond_destroy(pthread_cond_t *); //删除条件变量

int pthread_kill(pthread_t *thread*, int *sig*); //向某个线程发送信号

pthread 实际上实现了非 POSIX 兼容的 Linux 内核线程的封装。在 Linux 系统下，进程/线程的调度如图 xx-xx 所示：

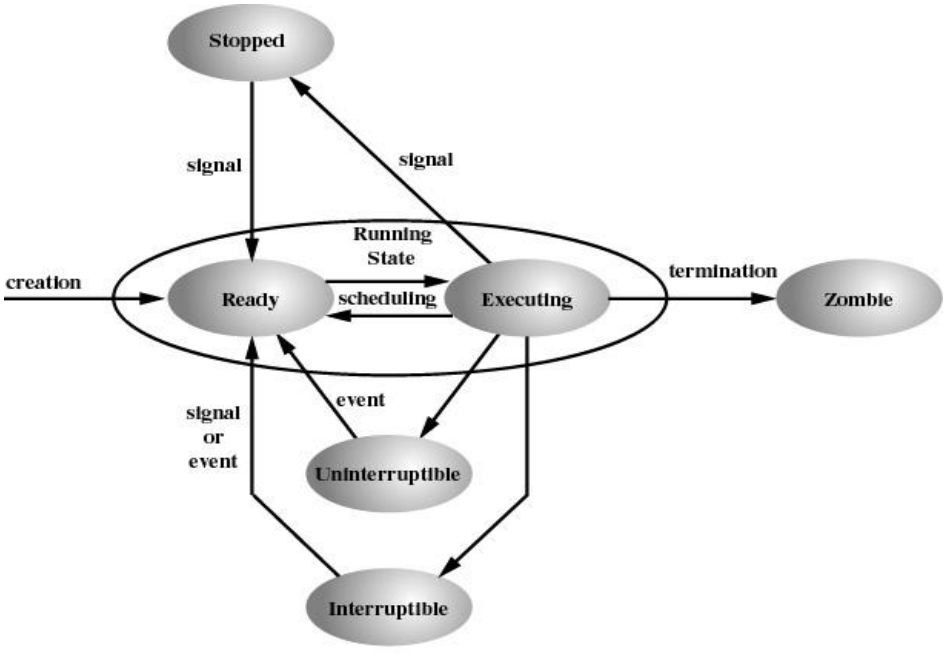


Figure 4.18 Linux Process/Thread Model

关于 Linux 下线程的内部实现，更多的可以参考[10]。我们下面来看一下基于 posix 的 pthread 的 Windows 化封装。

Linux 线程相关的 API 封装

从基本的功能上，我们可以看一下 Windows 和 POSIX 接口之间的对比：

平台 Windows Linux

功能		
创建线程	_beginthreadex	ptread_create
终止线程	_endthreadex	pthread_exit
等待线程终止	WaitForSigleObject	pthread_join
创建临界区	InitializeCriticalSection	pthread_mutex_init
进入/退出临界区	Enter/LeaveCriticalSection	pthread_mutex_lock/unlock
探测临界区状态	TryEnterCriticalSection	pthread_mutex_trylock
创建/删除同步对象	CreateEvent/CloseHandle	pthread_cond_init/destroy pthread_mutex_lock/unlock
设置同步对象	SetEvent	pthread_cond_broadcast
等待例程	WaitForSignelObject	pthread_cond_timedwait

在有了上表的比较后，我们基本上可以构造 Windows 化的 API 了，这主要包括线程例程、临界区例程以及同步例程(参考 orp\os\linux\os_wrapper.cpp)：

线程相关例程

创建线程，它们间的差别在于线程程序流的声明，通过强制类型转换，很容易的实现了该例程：

```
unsigned long _beginthreadex( void *security,   unsigned stacksize,
unsigned(__stdcall *start_address)(void *), void *arglist, unsigned initflag, unsigned *thrdaddr)
{
   pthread_t tid = 0;
   void *(*sa)(void *) = ( void *(*)(void *) )start_address;
   int stat = pthread_create(&tid, &pthread_attr, sa, arglist);
   *thrdaddr = 0;
   return tid;
}
```

暂停/恢复线程

POSIX 对线程的控制中并没有对线程有暂停/恢复的操作，因此需要自己实现。在 ORP 中，使用了线程的信号处理的方法，具体的是通过 pthread_kill 例程来对要暂停的线程发送 SIGUSR1 信号，而对要恢复的线程向该线程发送 SIGUSR2 信号。

```
void suspend_handler(int xx){
    sigcontext *sc;
    uint32 *ebp;
    int i;
    ORP_thread *thread = p_TLS_orpthread;

    asm("movl    %%ebp,%0" : "=g" (ebp)); //取得当前的栈帧

    for (i = 0; i < sc_nest; i++) {
        ebp = (uint32 *)ebp[0];
    }

    if (!use_ucontext) {
        sc = (sigcontext *)(ebp + 3);//取得当前的线程的上下文的指针
    } else {
        sc = (sigcontext *) &((struct ucontext *)ebp[4])->uc_mcontext;
    }
//把各个寄存器的当前状态保存到 ORP 线程中的寄存器结构中
```

```
        thread->regs.eax = sc->eax;
        thread->regs.ebx = sc->ebx;
        thread->regs.ecx = sc->ecx;
        thread->regs.edx = sc->edx;
        thread->regs.edi = sc->edi
        thread->regs.esi = sc->esi;
        thread->regs.ebp = sc->ebp;
        thread->regs.esp = sc->esp;
        thread->regs.eip = sc->eip;
        //设置暂停信号量，以表示本线程已经被暂停
        sem_post( &suspend_sem );
    //暂停处理，直到等待到 SIGUSR2 信号
    //这儿可能要疑惑，看一下 orp\os\linux\Signals_ia32.cpp 中的 initialize_signals 中对
    //sigset 的初始化：
    //sigaddset(&sigset, SIGUSR2);
                    //sigprocmask( SIG_BLOCK, &sigset, NULL);
    //这儿的调用首先把 SIGUSR2 添加到信号集，然后调用 sigprocmask 把 SIGUSR2 的信号阻塞
        sigwait( &sigset, &xx );
    //设置恢复信号量，表明本线程已经被恢复
        sem_post( &resume_sem );
    }
```

取得/设置当前线程的上下文

Linux 下并没有提供类似于 Windows 的封装，但是可以通过 POSIX 的信号处理来取得/设置当前的线程上下文。从上面的线程暂停的处理中，我们也看到了每次线程暂停下来后，在 ORP 线程的 regs 结构中，便保存了线程当前的上下文。设置上下文可以在线程暂停的过程中修改 ORP 线程的 regs 结构，待回复运行时，就已经是修改的线程上下文了。

临界区相关例程

临界区的处理上，除了临界区例程的返回值的定义不同外，其他的都只要转换成对应的调用就可以了。

同步对象例程

事件对象的封装相对于上面的两类要复杂一些。ORP 中，Linux 下对于事件对象封装如下：

```
typedef struct event_wrapper {
   pthread_mutex_t          mutex; //互斥量
   pthread_cond_t           cond;   //条件变量
   __uint32                 man_reset_flag; //是否自动复位
   __uint32                 state;          //状态
} event_wrapper;
```

为了在 Linux 下实现 Windows 下事件对象的语义，特别定义了上述结构。Windows 下事件对象的属性在上面讲到事件的时候已经提到过，对于自动复位事件对象的状态变换应该是原子的，因此在结构中特别定义了用户同步的互斥量。Linux 下除了提供了 Mutex、Semaphore 外，还提供了条件变量。条件变量允许实现在不同的条件下产生不同的结果：在某种条件下运行，在其他的条件下阻塞而使该进/线程睡眠。因此，配合上述结构中的 state 域，Window 下事件对象的可以通过互斥量和条件变量来实现。具体的 state 记录当前事件是否处于触发状态，互斥量用以多个线程之间的对更改状态保护，而条件变量将根据当前的事件的状态来选择合适的动作，如果未触发，则等待；如果触发，则返回。下面是各个相关例程的具体代码：

```
HANDLE CreateEvent(int *security, unsigned int man_reset_flag, unsigned int initial_state_flag,
 char * p_name)
{
    event_wrapper *p_event = (event_wrapper *)malloc( sizeof(struct event_wrapper) );
 //生成 event 结构
      int stat = pthread_mutex_init(&p_event->mutex, &mutex_attr_for_cond_wait); //初始化互斥量

      stat = pthread_cond_init(&p_event->cond, &cond_attr); //初始化条件变量
      p_event->man_reset_flag = man_reset_flag;  //是否自动复位
      p_event->state = initial_state_flag;              //初始化状态
 return (HANDLE)p_event;                          //返回句柄
}

BOOL ResetEvent(HANDLE hEvent)                                  //复位
{
   event_wrapper *p_event = (event_wrapper *)hEvent;                  //取得事件对象指针
   int xx = pthread_mutex_lock(&p_event->mutex);        //对互斥量上锁，进入临界区
   p_event->state = 0;                                              //修改状态为非触发态
   xx = pthread_mutex_unlock(&p_event->mutex);          //释放对临界区的控制
   return 1;
}


BOOL SetEvent(HANDLE hEvent)
{
   event_wrapper *p_event = (event_wrapper *)hEvent;

   int stat = pthread_mutex_lock(&p_event->mutex);                        //进入临界区
   p_event->state = 1;                                              //设置状态为触发态
   stat = pthread_cond_broadcast(&p_event->cond);                         //唤醒所有阻塞在
p_event->cond 上的对象
   stat = pthread_mutex_unlock(&p_event->mutex);                          //离开临界区
   return 1;
}


DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMillisec)
{
   if (dwMillisec < 10)
      return WAIT_TIMEOUT;

   struct timespec ts;
   ts.tv_sec    = dwMillisec/1000;
   ts.tv_nsec = (dwMillisec%1000)*1000000;
```

```
        struct timeval tv;
        int stat = gettimeofday(&tv, 0);

   ts.tv_sec    += tv.tv_sec;
        ts.tv_nsec += tv.tv_usec*1000;                                  //初始化超时设置

        event_wrapper *p_event = (event_wrapper *)hHandle;

        stat = pthread_mutex_lock(&p_event->mutex);             //进入临界区

        int wait_status = WAIT_OBJECT_0;

        while (1)
           {
              if (p_event->state != 0)                                  //是否触发，如果为触发态，退出循环；否则继续
                    break;

              wait_status = pthread_cond_timedwait(&p_event->cond,
                                                   &p_event->mutex,
                                                   &ts );
       //如果条件变量不是触发态，进入时限等待状态，在进入等待状态前，会释放对互斥量的锁；否则，返回 0
              if (wait_status == ETIMEDOUT)
        break;
              if (wait_status == 0)
        break;

           }
      if (p_event->man_reset_flag == 0)                         //如果是自动设置事件对象
              p_event->state = 0;                       //更新事件的状态

        stat = pthread_mutex_unlock(&p_event->mutex);

        if (wait_status == ETIMEDOUT)
           return WAIT_TIMEOUT;

        if (wait_status == 0)
              return WAIT_OBJECT_0;

        return WAIT_TIMEOUT;
      }
```

在这一节中，我们主要讨论了 Java 支持并发特性的基础—线程的内部封装以及在 Windows/Linux 下的实现，这对于理解语言并发的内部实现是非常有帮助的。在讨论了线程之后，我们来看一下和并发密切相关的同步在 ORP 中的实现。

5.2 锁在虚拟机中的作用

Java 支持并发特性很大程度上依赖于同步。同步是在多个并发的线程之间协调资源访问的一种机制[1]。在 Java 中这种机制是通过监视器（Monitor）[2,3]来获得的。Java 虚拟机中的监视器是一种允许同一时刻只有一个线程能够在代码的一个区域中执行的机制。Java 中的每个对象和类都是由锁保护的[3]。对于对象而言，它所保护的是对象的实例数据；对于类而言，它所保护的是类变量。

监视器提供了两种类型的同步：互斥式和协作式。互斥式同步是通过对象的锁来实现的，它允许多个线程能够独立的对共享的对象进行操作而不会相互干扰。而协作式同步是通过对象执行等待(wait)和通知(notify)来实现的。在 Java 的字节码（bytecode）中，有对监视器操作的专门指令，即获得互斥访问权的 monitorenter 以及释放互斥访问权的 monitorexit。

在 Java 程序中，有两种方法来获得同步。首先，我们可以声明需要同步操作的方法为同步的，这样的方法 Java 语言规范中称为“同步方法”，由关键字 synchronized 进行限定。例如：

```
synchronized void synchronizedmethod() {
…
}
```

这个方法所包括的所有的部分都是受保护的，即任何时候只能由一个线程能够在这个方法内部。

另外一种方法，我们可以在某个方法内部使用同步块来进行保护，这是通过对本身使用 synchronized 限定块来达到的。例如：

```
void somepartneedsynchronizedmethod() {
synchronized(this)  {
        …
    }
}
```

这两种方法在执行结果上是相同的，但在具体的内部实现上有着细微的差别。同步都是通过 monitorenter/monitoexit 指令来实现的，这在两种方法实现中都是一样的。但是，在第一方法中，在虚拟机获得对象锁之前，这个方法是不能够被调用的，而第二种方法中，方法先被调用，然后再获得锁；同样的，第一种方法中，方法执行完后，虚拟机释放锁，而第二种方法中，方法先释放锁，然后返回。还有一个细微的差别，那就是在第一种方法中，我们不需要为这个方法创建专门的*异常处理表*（参考 6.4），因为不管这个方法如何结束，虚拟机都会释放；但在第二种方法中，为了要释放锁，必须产生专门的异常处理表以在发生异常的情况下释放锁。

在讨论了锁的基本作用以后，我们来看一下锁在 ORP 中的具体实现。我们首先介绍一下 ORP 中同步相关例程的组织，然后我们分别的来看一下 ORP 中对象锁第一版和第二版的具体实现。

5.3 同步相关例程的组织

ORP 是要实现所有由 JVM 规范规定的指令，同步例程也不例外。我们在这儿先来看一下同步相关的 monitorenter/monitorexit 在 ORP 中的具体组织。

ORP 在设计上把即时编译器和垃圾收集器独立出来了，它们和核心虚拟机之间只能够通过接口来进行相互的工作。在前面我们也说到过，Java 的同步例程是由虚拟机指令 monitorenter/monitorexit 来完成

的。核心虚拟机自然是功能实体的提供者，但真正为 Java 程序即时编译生成这些指令的即时编译器如何使用这些实现的，下面，我们就来看一下在 ORP 中的具体组织。

我们在第三部分即时编译器中会看到更多的关于即时编译器和虚拟机之间的交互。这儿我们首先简单的说一下。在 ORP 内部，核心虚拟机为即时编译器提供了运行期支持例程，包括所有的需要使用复杂处理的 bytecode 的实现，如 newarray,f2l,l2f,monitorenter,monitorexit 等。这些例程在即时编译器中可以通过接口函数 orp_get_rt_support_addr 来取得。对应于 monitorenter/monitorexit，从代码中我们可以看到对应的是返回了 getaddress__orp_monitor_enter_naked()/ getaddress__orp_monitor_exit_naked()。

下面我们来看一下这两个例程的具体实现，在 ORP 中这两部分都是动态发射的代码，为了简单起见，我们把它们转换成通常的 iA32 的 MASM 格式。首先我们来看一下 ORP 的对象锁第一版本的 getaddress__orp_monitor_enter_naked()的实现：

```
push [esp+4]
call orp_monitor_cmp_value
add esp,4
mov ecx,[esp+4]
sub ecx,4
mov edx,eax
move ax,0
lock: cmpxchg [ecx],edx
sub eax,0
jne L0
ret 4
L0:
call getaddress__setup_java_to_native_frame
push [esp+sizeof(J2N_Saved_State)]
call orp_monitor_enter
call getaddress__pop_java_to_native_frame
ret 4
```

图 xx-xx：getaddress__orp_monitor_enter_naked 的实现

从图 xx-xx 可以看出，在进入这个函数前，[esp+4]的内容为对象指针。在处理上，首先取得当前线程的索引，从对象指针计算得到对象头锁的地址到 ecx 中，然后把得到的线程索引放入 edx，接下来是一个带 lock 前缀的 cmpxchg 指令，这可以保证在多 CPU 的系统上更改的原子性。cmpxchg [ecx],edx 指令的语义是这样的，它会比较[ecx]的内容和 eax 是否相等，如果相等，则把 edx 中的内容写到[ecx]中，并设置零标志位；如果不等，将会[ecx]中的内容写到 eax 中并清除零标志位。后续的 sub eax,0 指令就判断上述的 cmpxchg 指令是否成功。如果成功，eax 中应该为 0 否则肯定非 0。如果上述指令成功，则清栈后返回；如果上述指令不成功，表示不能用快速的处理方法，必须通过对 orp_monitor_enter 方法的调用来完成。由于这是一个本地方法，因此在调用前，必须为它创建 J2N_Saved_State，关于 J2N_Saved_State 的解说，可以参考第 6 章中的相关部分。我们在后面更多地论述都是针对 orp_monitor_enter 的，因此这儿就不多加评述了。

下面看一下 getaddress__orp_monitor_exit_naked()的实现：

```
push [esp+4]
call orp_monitor_cmp_value
mov ecx,[esp+8]
sub ecx,4
mov edx,0
lock:cmpxchg [ecx],edx
```

```
pop ecx
sub eax,ecx
jne L0
ret 4
L0:
call getaddress__setup_java_to_native_frame
push [esp+sizeof(J2N_Saved_State)]
call orp_monitor_exit
call getaddress__pop_java_to_native_frame
ret 4
```

图 xx-xx：getaddress__orp_monitor_exit_naked 的实现

基本的实现和 getaddress__orp_monitor_enter_naked()的类似，在这儿就不多说了。

在了解了在运行期 Java 代码中调用 monitorenter/monitorexit 时作的基本的工作后，我们来看一下比较复杂的 orp_monitor_enter/orp_monitor_exit 的具体处理。

5.4 ORP 中对象锁实现的第一版

ORP Object Lock V1 可以在 orp/base_natives/common/mon_enter_exit.cpp 下找到，相关联的可以参考一下 common_base_natives_classes 的工程文件。

由于 ORP 中锁的实现在 Windows、Linux 由于各自提供的同步对象、例程不一样，因此，我们在讨论了对象锁的基本构成后将针对 Windows 和 Linux 分别作描述。

5.4.1 对象锁的基本结构

对象锁在对象中是通过对象头中的一个双字来实现的，在源代码中的构造(参考 common\include\sync_bits.h) ：

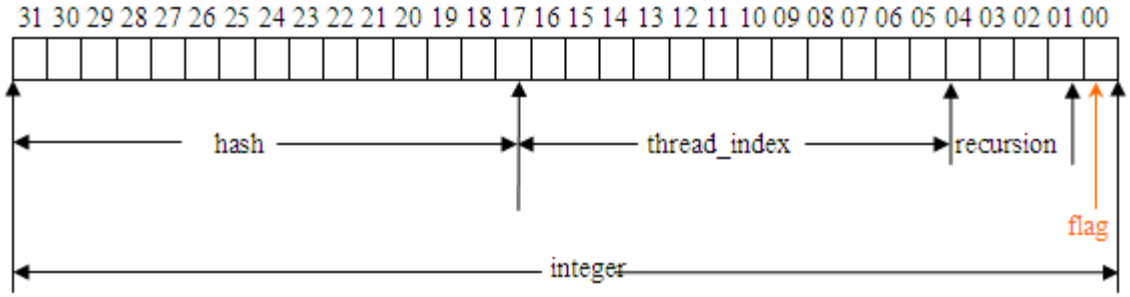


图 5.1 OLV1 中对象头的布局

我们首先对图 5.1 中的各个段进行解释：

第 0 位是一个标志，如果为 0 则表示整个内容是一个向前的指针(forwarding pointer)；如果为 1，则表示内容是一个真实的对象锁。这儿用到了一个技巧，如果是向前指针，那么肯定能够保证标志为 0。为什么？[在现代处理器结构上，为了处理高效，向前指针指向的地址肯定是 4 字节或者 8 字节对齐的，因此最低位肯定是 0。]

第 1 位到第 4 位共 4 位是递归锁的深度，所谓递归锁是指能够在获得锁后再次获得该锁的锁，递归锁的深度就记录了本身重复多少次获取了该锁。待到这个计数为 0，这个锁才是空闲的。为了提高处理的效率，并没有将递归深度进行右移，然后增加 1 后左移。而是直接的每次增加 2，直接对递归深度部分进行操作，原因 ((a>>1)+1)<<1 等价于 a+2。

第 5 位到第 16 位共 12 位是线程索引，如果某个线程占有了该锁，那么这个域里面的内容就是这个

线程的索引。

第 17 位到第 31 位共 15 位是 hash 的值，它记录了这个对象在创建时候的哈希值。

在介绍了对象锁的基本布局后，我们开始我们具体实现的介绍。我们将分别以 Windows、Linux 下的实现作介绍。

5.4.2 Windows 下对象锁的实现

我们首先来看一下 monitorenter 指令在 ORP 内部的实现：

当某个线程要取得对象的锁的时候，首先会从对象的指针找到锁的头，位置就在对象指针的前一个双字的地方；然后通过调用 acquire_header_lock 来尝试获取对象的锁。在 acquire_header_lock 的内部是通过对对象原来的锁信息和空闲锁信息相比较来完成空闲测试的。具体的代码如下：

```
acquire_header_lock(p_header)
{
  while (true) {
        volatile PVOID free_header =(PVOID)(*p_header & BUSY_FORWARDING_BIT_MASK);
        volatile PVOID locked_header =(PVOID)((POINTER_SIZE_INT)free_header |
BUSY_FORWARDING_BIT);
     if (InterlockedCompareExchange ((PVOID *)p_header,
                                              locked_header,
                                              free_header) == free_header) {
        return;
            }
while ((*p_header & BUSY_FORWARDING_BIT) == BUSY_FORWARDING_BIT) {
           Sleep (0);
        }
     }
}
```

图 5.2: acquire_header_lock()的 Windows 代码

类似于典型的 test-and-set 操作。InterlockedCompareExchange 是由 Windows 提供的 API 函数，它可以完成原子的比较置换操作。如果 p_header 所指的内容和比较的 free_header 内容相等的话就把 p_header 所指的内容置换成 locked_header，同时返回原来的值；如果不等，则什么都不做，只是返回 p_header 所指的内容[4]。如果空闲测试成功，程序就返回了；如果失败，程序将让权等待，即放弃本次的时间片，并等待直到锁可用。从图 5.2 可以看出，程序只有在标志位为 0、获得锁对象后才返回；如果发现标志位是上锁，将等待直到拥有该锁的线程释放该锁，并与其他的等待该锁的线程(如果有的话)竞争。最终仍然只有一个能够获得，其他的将仍然让权等待。

在获得头锁后，就需要对头锁里面的相应的信息进行更改了。如果发现拥有该锁的线程索引域和递归深度域都为 0 时，说明当前锁时空闲的，我们只要把它设置成当前线程索引以及当前的递归深度就可以了，然后释放头锁返回。这时我们就已经获得了这个对象的锁。否则，进入较为复杂的锁的处理情况。在这中间，如果递归深度为最大值时，则进入*慢处理*。如果线程索引等于当前线程，则分为两种情况，如果要递归深度即将溢出，转入额外的处理，我们称作*溢出处理*；否则直接对递归深度进行增量操作后释放头锁返回。如果线程索引不等于当前线程，则表示当前的锁正在被其他的线程使用需要等待，我们称之为*等待处理*。

接下来，我们分别介绍慢处理、溢出处理以及等待处理。

慢处理

监视器的递归深度为最大值时，就标志着该对象锁对应的是一个满处理的锁。同时，处理将进入慢

处理，从字面上就可以看出，这个过程会很慢。因为不再是简单的对对象的头 32 位做操作了。

在进入慢处理的锁中，对象锁中的信息也不再是前面介绍的线程索引和递归深度的合成了，而是指向一个锁块(Lock Block)的指针。前面也提到过，这主要是利用了在 IA32 结构上地址的对齐，对于 32 位地址总线的结构上，保证地址可以被 4 整除就可以确保对该地址的 32 位访问在一个访存周期内完成，在实际中使用了能够被 32 整除的地址，因此地址的低 5 位总是为 0，这就可以使用这个 5 位信息，这也就是递归深度和向前指针标记的使用由来。具体的代码可以参考 get_a_block 的实现。

进入慢处理后，首先要把当前的线程索引和锁块中保存的线程索引进行比较。如果相等，则为递归锁；如果不等，则检查是否是空闲锁，只要在锁的信息内填入合适的信息后就能够使用它，否则表明这个锁被其他线程占用着，本线程应该做等待或者其他的处理。

递归锁的处理中，要检查递归深度是否要或者已经溢出，如果都没有，简单的增加锁块内保存的对象锁内的递归深度后释放头锁返回；否则，如果将要溢出，则取得一块新的锁块，将新的锁块和老的锁块之间链接起来。（具体的：新块的 p_forward_link 指向老的锁块，老块的 p_back_link 指向新的锁块。）锁的递归深度将要溢出时，在锁信息内的递归深度已经不能够记录这部分信息了，因此把锁的递归深度记录到锁块的 lock_recursion_count_shifted_left 中。锁块中保存的老的对象头信息中相应的信息会被慢锁的标记取代。然后使用新的锁块的地址合成新的头信息，写入到相应的对象锁头中，接着就能够释放头锁而返回了。如果检查中发现锁块中保存的对象锁的递归深度已经标志是慢锁了，就简单的对锁块的 lock_recursion_count_shifted_left 作增量操作，然后释放头锁后返回。

如果本线程需要的锁已经被占用，就会新分配一个锁块。把新的锁块链入老的链中，标记自己的状态 lock_or_wait_state 为等待锁。在把对象锁中的信息修改为由信息锁块地址和慢锁标记及忙位合成的内容。然后就进入等待处理。

等待处理

等待处理的核心是建立在线程自挂起的实现机制上的。进入等待前，首先线程会重置事件状态为非触发态，释放了头锁之后就进入等待状态。如果持有该监视器的线程没有释放监视器的使用权，这个线程就会一直在休眠状态，直到释放使用权后唤醒该休眠线程。

线程从休眠状态中恢复后，该线程会首先关闭 GC。然后对头锁进行修改，包括重新取得头锁，然后把锁块放到线程的空闲锁块中，接着释放头锁。在这儿，头信息中的内容仍然是指向新近放到空闲锁块列表中的锁块。最后，过程返回表示该线程已经取得了该监视器的使用权。

溢出处理

溢出发生在递归锁深度等于 14 的时候（实际上在内部是 28(0x1c)，因为第 0 位是作其他用途的，每一次递归，锁深度都增加 2），对递归锁的深度的再次递增，会引起递归深度值与慢锁的标记冲突，因此有必要专门针对递归深度的变化作额外的处理。

溢出处理首先分配了一个锁块，用于记录实际的递归深度，然后在锁块的老对象头中记录拷贝锁信息中的线程索引及慢锁标志，并使用锁块中的递归锁深度域对递归深度进行记录。接着根据锁块的地址形成新的对象头信息，放入到对象头中，最后释放头锁后返回。

在看了 monitorenter 的实现之后，我们也基本上能够了解与之相对应的释放监视器的 monitorexit 的实现了。我们简单的来看一下它的实现，从上面的处理中也可以了解到，对于释放监视器动作的复杂性应该集中在对慢锁的处理上。

释放例程处理中，最简单的是对非竞争锁的释放了，它是通过 InterlockedCompareExchangePointer 将锁信息更改为 0，如果成功便返回。否则，即如果是竞争锁，则首先需要调用 acquire_header_lock 来取得头锁。获得头锁后，就可以对锁进行处理了。

首先，如果锁的线程索引以及递归深度域中的信息和本线程对应的线程索引及深度为 1 时相同，则将这部分信息置成空，然后释放头锁后返回。否则，检查是否是慢处理，如果是就转入到慢处理中。否则，检查线程索引是否是本线程的索引，如果不是则一定是出错误了，抛出异常。如果是本线程索引，则这是一个递归锁的释放例程，只需要简单的对递归深度减掉一个单位就可以了。然后释放头锁后返回。

下面我们来看一下释放例程中的慢处理：

首先，要从对象锁的信息中取出锁块链的指针，前面已经介绍过方法，在此我们不再重复。然后根据锁块中保存的老的对象头内容作处理，这包括：

1） 如果老的对象头中的标志表示这个锁是一个慢锁，则这个锁一定是一个递归深度溢出的锁，对于锁块中的lock_recursion_count_shifted_left减去一个单位。检查lock_recursion_count_shifted_left是否是即将溢出的临界值：

i. 如果是的话，也表示可以使用头锁中的递归深度来标记深度了。锁块链会把当前的锁块中链中取出，并根据锁块链的组织作不同的处理。即如果锁块链的向前指针中非空的话，把该锁块的向后指针清空，重新设置该锁块内的老的对象头信息，并根据新的指针值形成新的对象头内容赋给对象头，然后释放头锁后返回。否则，则老的对象头中保存的就是原来的对象头信息，根据新的递归深度、线程索引以及忙标志形成当前的对象头内容，赋值给当前的对象头，释放头锁后返回。

ii. 如果递归深度不是临界值，简单的释放头锁后返回（递归深度大于临界值，还不可以转换成轻量锁）；

2） 如果老的对象头中递归深度大于 0 并且小于等于临界值，对老对象头减去一个递归深度单位后释放头锁返回；

3） 如果锁块中的对应的*上锁等待*标志是*等待锁*，如果锁块中向前指针为空，则根据各个部分的信息形成新的对象头，并设置上锁等待标志为*占有线程需要调用释放例程*。然后根据等待该锁的线程的当前状态作不同的动作：

i. 线程在等待状态：置状态为运行状态，把要触发的事件对象设置为中断事件；

ii. 线程在休眠状态：不会出现这种情况，

iii. 线程在运行或者死亡：设置要触发事件为该线程的监视器事件；

接着释放头锁，触发事件后返回；如果锁块中的向前指针非空，则先为前向指针所指的锁块中的老的对象头设置新的值，合成新的对象头值；接着，从链中删除当前的锁块；然后，根据当前锁块所属线程的应用状态来选择要触发的事件。如果是等待状态(thread_is_waiting)，要触发的事件就是线程的中断事件；否则是线程的监视器事件，最后释放头锁，触发事件后返回；

4） 如果上锁等待标志是等待通知，新的头信息就是当前锁块中老对象头的对象哈希值和要通知线程索引的或，然后把锁块中老的对象头设置为新的头信息，释放头锁后返回。

为了便于理解，我们把运行期间时期内部数据结构的组织在图 5-xx 中给出。

到此，我们就介绍完了 ORP 中关于 orp_monitor_enter 和 orp_monitor_exit 的具体实现。

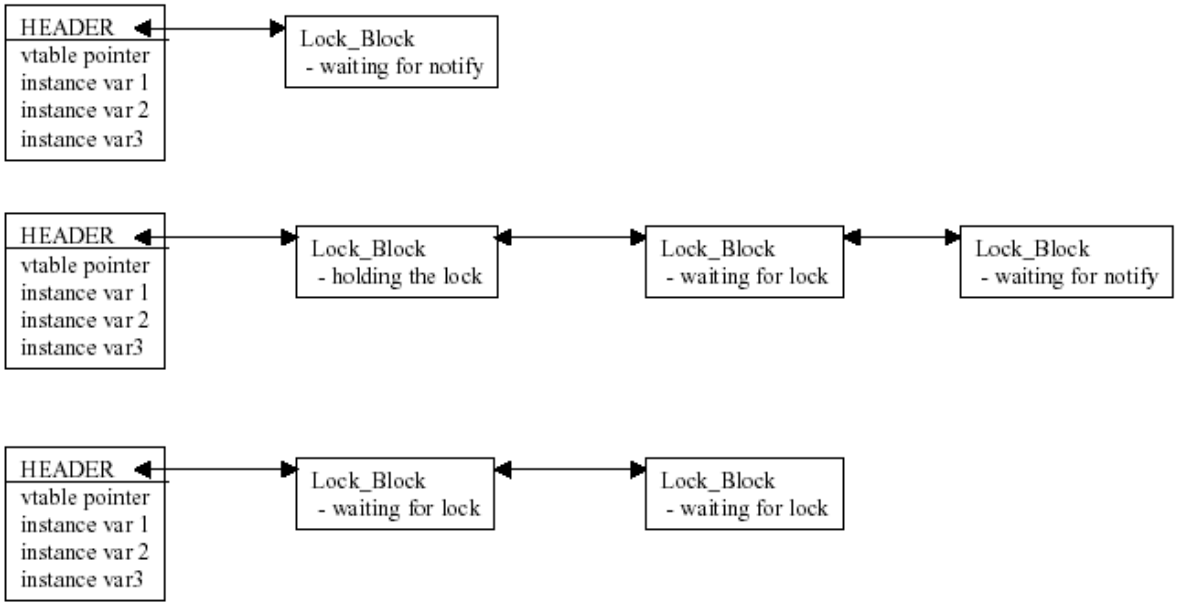


图 5-xx: 监视器内部数据结构在运行期的组织

5.4.3 Linux 下对象锁的实现

Linux 下绝大多数实现和 Windows 下的类似，只是在个别的内部实现上不同。具体的：

Linux 下没有互锁比较交换例程(InterlockedCompareExchange)，因此这部分是用汇编实现的，实现的核心与 5.3 种介绍的类似。熟悉 AT&T 格式汇编的读者可以看一下具体的实现。

Linux 下释放当前调度时间片使用系统调用 sched_yield()而 Windows 下使用 Sleep(0)，但效果是一样的。

5.5 ORP 中对象锁实现的第二版

ORP Object Lock V2 可以在 orp/base_natives/common_olv2/mon_enter_exit.cpp 下找到，相关联的一些可以参考一下 common_base_natives_olv2_classes 的工程文件。

在 mon_enter_exit_olv2.h 中对 header_bits 的定义如下：

```
union header_bits {
    int             h_int;
    unsigned char   h_byte[4];
    unsigned short  h_short[2];
};
```

具体的布局如图 5-xx 所示。

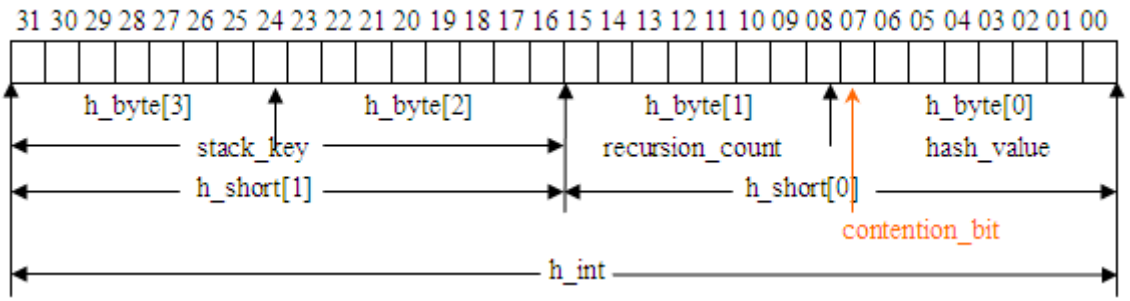


图 5.X OLV2 中对象头的布局

从图 5-xx 可以看到，对象锁的第二版把整个锁分成了 4 个部分，我们先来看一下这个部分的含义：

第 0-6 位 hash 值，用来记录对象的 hash 值；

第 7 位 竞争位，如果置位，表示这是一个竞争锁

第 8-第 15 位 递归深度计数，它记录了锁在递归使用过程中调用获得的次数

第 16-第 31 位 堆栈关键字，它记录了当前线程的堆栈的高 16 位(Windows)或高 11 位(Linux 下)；这是由于不同操作系统对线程堆栈的处理不同而引起的。Windows 下调用 beginthreadex 时传递的堆栈大小为 64K，因此，可以保证各个线程的高 16 位是各不相同的；而 Linux 下，每个线程的堆栈都有 2M，因此，可以保证各个线程的高 11 位是各不相同的。

由于锁的具体组织不同，在对即时编译器的运行期支持上，代码也不相同。下面我们就来看一下 OLV2 的 getaddress__orp_monitor_enter_naked()/getaddress__orp_monitor_exit_naked()的实现。

```
mov edx,esp
xor eax,eax
mov ecx,[esp+4]
shr edx,STACK_KEY_SHIFT
lock: cmpxchg [ecx+6],dx
jne L0
ret 4
L0:
call getaddress__setup_java_to_native_frame()
push [esp+sizeof(J2N_Saved_State)]
call orp_monitor_enter_slow
call getaddress__pop_java_to_native_frame()
ret 4
```

图 5-xx：OLV2 的 getaddress__orp_monitor_enter_naked()

可以看到，这儿已经把堆栈关键字代替了 OLV1 中的线程索引。为什么是[ecx+6]？老的对象布局中，头信息在对象指针前 4 个字节，到了新的对象布局中，把这部分信息移到了对象指针内容的后面，即[ecx+4]就是新的头信息。由于在堆栈关键字处在 hbyte[2],hbyte[3]上，因此刚好是[ecx+6]。

再来看一下 OLV2 的 getaddress__orp_monitor_exit_naked()的处理：

```
mov edx,esp
mov ecx,[esp+4]
shr edx,STACK_KEY_SHIFT
mov ax,[ecx+6]
cmp ax,dx
jne L0
mov ax.[ecx+6]
cmp ax,0
jne L1
mov edx,0
mov [ecx+6],dx
mov edx,eax
and edx,0x80
cmp edx,0x80
jne L2
```

```
ret 4
L2:
push [esp+4]
call find_an_interested_thread
add esp.4
ret 4
L1:
sub eax,1
mov [ecx+5],al
ret 4
L0:
push &(string_of_IllegalMonitorStateException
call throw_java_exception_wrapper
ret 4
```

图 5-xx：OLV2 的 getaddress__orp_monitor_exit_naked()

从图 5-xx 的实现可以看出，对于释放监视器的处理中，比 OLV1 简化了一个 lock 指令，因此可以降低执行监视器相关例程的运行期开销。

下面我们来看一下 orp_monitor_enter_slow 的处理。最简单的情况，头信息中的内容为 0，即该对象空闲，则使用带 lock 的 cmpxchg 指令来修改高两个字节。如果当前的堆栈关键字和本线程的堆栈关键字相同，那么这是一个递归锁，增加深度计数，如果计数溢出（在实现上直接使用了字节的递增，到 255 后加 1 将引起绕回到 0，因此根据这个字节的内容是否为 0 来判断溢出），调用相关的处理例程，和 OKV1 的深度计数 14 相比，现在的 255 要大得多。否则，说明当前对象正在被其他线程使用，进行有限次数的自旋，等待其他线程放弃监视器的控制权：如果等到了，得到后返回；否则对象将阻塞在该监视器上。阻塞在监视器上的动作将包括设置竞争位，在进入休眠前，检查是否已经可以获得这个对象的监视器，如果可以则不必阻塞直接获得后返回；否则，就调用相关的等待例程进入休眠状态。

从图 5-xx 中的代码可以看到，并没有对 orp_monitor_exit 的调用。实际上,OLV2 中 orp_monitor_exit 的处理和 getaddress__orp_monitor_exit_naked 一样。其中 find_an_interested_thread 就是从监视器等待列表中扫描找到第一个等于当前对象的索引，根据这个索引找到休眠的线程以及它所等待的事件，并唤醒该线程。

从上面的分析可以看出，相对于 OLV1 的复杂处理，OLV2 的处理要简洁高效得多。对象锁第二版大大的降低了由于同步所带来的运行期开销。

5.6 比较

我们专门用这一小节来对 OLV1 和 OLV2 做一下比较。

OLV1 中，如果相应的运行期 wrapper 不能够为当前线程获得监视器的话，所有的等待、溢出都是建立在对头信息锁的修改上的，对它的修改都要调用互锁例程，这对于多 CPU 系统上提高性能非常不利。并且，在慢处理、溢出处理中，都涉及到对头锁的获取和释放，因此 OLV1 的性能并不很好。同时，当调用 OLV1 的释放监视器例程时，同样需要进行一次 lock 前缀的指令。由于重复使用了头信息，每一个竞争锁都会导致第一次 lock 前缀的 cmpxchg 失败而进入 orp_monitor_exit 的处理，复杂的组织和处理会消耗掉很多额外的周期。

OLV2 中，把头信息的功能单一化，并且利用良好的分界使得可以通过访问字节/字的方式快速的访问/操作各个部分，对于简化代码处理减少处理时间是非常有帮助的。另外，OLV2 中并没有使用动态的结构来保存等待在某个对象上的线程，而使用了一个静态的 mon_enter_fields 类型的 monitor_enter_array 数组，用于记录该线程的指针以及等待的对象指针，在某个占有锁的线程释放时，可以通过扫描整个数组来唤醒该线程。这比起 OLV1 中的锁块结构要简单高效得多。另外，前面也提到了，OLV2 的锁的释放并不需要 lock 前缀的指令，最大程度的为高性能服务。OLV2 在实际使用中的性能要远好于 OLV1 的性能。

[Reference]

[1] Siberschatz Abraham, Galvin B. Peter , Operating System Concepts(6th Edition), John & Wiley, 2002

[2] Venners Bill, Inside the Java 2 Virtul Machine, Osborne McGraw-Hill, Jan, 2000

[3] Lindholm Tim,Yellin Frank, Java Virtual Machine Specification(2nd Edition), Addison-Wisley, Mar,1999

[4] Microsoft, MSDN 2001 July version, 2001

[5] GNU Classpath VM Integration Guide, http://www.gnu.org/software/classpath/doc/vmintegration.html

[6] Java Class Thread, http://java.sun.com/j2se/1.4/docs/api/java/lang/Thread.html

[7] Jeffrey Richter, Programming Applications for Microsoft Windows, Microsoft Press, 1999

[8] David A. Solomon, Mark E. Russinovich, Inside Windows 2000, Microsoft Press, 2000

[9]The Single UNIX ® Specification, Version 2,
http://www.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

[10] Understanding the Linux Kernel, O'Reilly,2001

第6章 即时编译器(JIT)的支持

这一章，我们主要讲述 VM 对 JIT 的支持，即主要讨论 byte code 在交给 JIT 做真正的即时编译前后，VM 所做的工作。

我们首先结合流程来看一下在交给具体的 JIT 前，ORP 所做的工作。我们在这儿主要关注的是 class 中的方法，在内部表示上，使用的是 Method 类。有关 Method 类的详细内容，可以参考本书同一部分的第一章，内部数据表示中的 Method 部分。每个 Method 中都有一个_code 域，这个域指向了该方法最近的入口点。用户指定的 class 文件在装载过程中，类装载器将根据不同的属性来使用不同的策略对待二进制表示。如果在 class 文件中被解析为方法体，就会调用 class_parse_methods 方法(参考 common/class_loader/Class_File_Loader.cpp)来进行初始化方法体的工作。code 真正的 bytecode 是装入到方法体的_byte_codes 域指向的位置，域_byte_code_length 记录了方法的 bytecode 数。_code 的最初状态将指向方法构造时创建的 compile_me_stub。当第一次执行这个方法时，JIT 会创建它的代码，如果是本地方法的话则由虚拟机创建。如果使用了动态优化并且这个方法是一个经常调用的方法，这个方法就会被重新编译，_code 会指向优化过的代码。这将在 JIT 部分中描述。但不管什么时候，_code 都是指向正确的方法入口点的。JIT 产生的代码依赖于上述的结果，也依赖于使用通过_code 的地址（这可以通过 JIT 接口中的 method_get_indirect_address 来获得）的间接调用来实现 invokestatic 和 invokespecial。在更新了_code 的值后，核心虚拟机会通过调用 Method::apply_vtables_patches()负责更新对应于当前方法的虚拟方法表入口。

在大概的描述了基本的工作之后，我们主要将分成以下的几个部分进行描述。首先我们会介绍一些在描述过程中会使用到的一些准备知识；接着是我们介绍一下核心虚拟机在编译前做的工作，他们是为方法的第一次编译做准备工作的；然后，我们会讨论 JIT 编译通常的 Java 方法，是主要的 JIT 和核心虚拟机之间的组织和交互；再次，我们讨论核心虚拟机编译本地方法时做的工作；最后，我们看一下核心虚拟机中提供的对异常处理和堆栈展开部分的支持。

6.1 准备知识

在 ORP 的内部，提供了几种接口调用方式，我们首先在这儿先简单的介绍一下它们的基本情况。然后，我们再来看一下 Java 代码和本地代码之间转换时用到的数据结构。这些对于理解后面的部分是有益的。

6.1.1 Java 本地接口(Java Native Interface, JNI)

由 Sun 制定的接口标准，它允许在 Java 虚拟机内运行的 Java 代码能够和其他语言编写的应用程序或者库之间互操作。Java 本地接口在 ORP 中是所有接口中首选的，也是唯一的对应用程序开发者可用的接口。

JNI 的方法都是通过接口指针来使用的，接口指针本身是一个指向指针的指针。它指向了一个指针数组，里面的每一项都指向了一个方法，如图 xx：

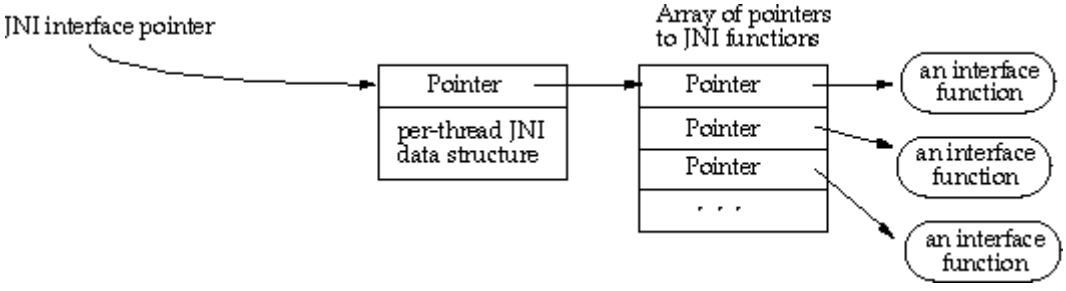


图 xx-xx: JNI 的接口指针

从上述的结构可以看出，每一个方法调用至少要经过两次访问内存的操作，才能找到要调用方法的地址，这在某种程度上可以提高适用性，但很大程度上限制了性能。因此，对于频繁使用的虚拟机内提供的方法使用 JNI 是不可取的，因此 ORP 中也使用了 RNI 接口。

6.1.2 原生本地接口(Raw Native Interface, RNI)

由微软制定的在微软的 JVM 中提供的一种允许 Java 和本地代码之间交互的接口。这种接口比 JNI 更底层，但是它需要函数名字遵循严格的命名规则，需要谨慎的处理和 Java 的垃圾收集器之间的工作：比如在耗时的工作间、施放时间片的地方、有可能阻塞在其他线程的地方以及等待用户输入的地方调用 GCEnable 和 GCDisable。RNI 方法必须是专门的为在 Java 环境下设计的。同时，RNI 方法可以快速的访问 Java 对象的内部结构和 Java 的类装载器，对 RNI 的方法调用的开销也比较小。

出于性能的考虑，ORP为经常调用的方法选择了RNI和直接调用本地代码的接口方式，但ORP中的RNI和微软的RNI不兼容。直接使用本地代码接口方式就是虚拟机直接调用本地方法，而不提供任何其他的外围方法来纪录必要的从Java代码到本地代码的转换的信息。缺少这部分的信息就意味着在这些方法中不能进行退栈操作。因此这种接口方式的方法是非常有限的，目前在ORP中只有为GNU的classpath直接调用`java.lang.System.currentTimeMillis`方法。

对于RNI和JNI方法，ORP会生成相应的外围方法来为堆栈回退操作保存足够的信息。

6.1.3 Java 到本地代码之间的转换

核心虚拟机中有部分和系统密切相关的代码需要用 C 语言来实现，但 Java 语言和 C 语言之间本身并没有这种接口机制。JNI 和 RNI，就是为了提供这种交互而提出来的。在运行时，为了保存当前 Java 代码运行的状态，ORP 中特别构造了 Java 到本地代码保存状态的数据结构。在调用 JNI 和 RNI 方法时，外围方法都将会为这些本地方法提供这些信息。

Java到本地代码保存状态(J2N_Saved_State)结构，参考 /orp/common/include/stack_manipulation.h中的定义：

```
struct J2N_Saved_State {
uint32 prev_ljf;
uint32 *p_ljf;
Object_Handle local_object_handles;
uint32 edi;
uint32 esi;
uint32 ebx;
uint32 ebp;
uint32 eip;
}; //J2N_Saved_State
```

我们来说明一下这里面比较重要的数据结构。

最近 Java 帧(Last Java Frame, LJF)，是 ORP 中提供的连接 Java 调用本地方法链的结构。由这个结构来维护系统中本地代码部分的调用链。很显然，在系统运行的时候，属于编译过的 Java 方法的本地代码的运行和 ORP 的代码是不加区分的混合分布的，为了在实现垃圾收集、异常处理、安全相关的过程中作堆栈回退处理，保存这个域是有必要的。在 ORP 中，LJF 的作用类似于其他语言中的活动帧指针。例如在 iA32 上 C 语言生成的代码中，通常的 C 方法的调用都是以保存老的活动帧指针，然后创建新的调用帧。（在有些书中，也把活动帧称作活动纪录。）C 中典型的代码：

```
push ebp
mov ebp,esp
……
```

mov esp,ebp
pop ebp

注意，关于 ebp 在系统中的使用并没有限定，这通常都是由语言的应用二进制接口(Application Binary Interface, ABI)定义的。完全可以把 ebp 当作通用寄存器来使用，在 O3 JIT 中，就把 ebp 当作通用寄存器来使用了，这在某种程度上可以减轻寄存器压力，提高处理的效率。

在进行 Java 的方法调用时，LJF 的典型伪代码也就是：

```
push p_ljf
push [p_ljf]                //保存前一个 ljf，也就是 prev_ljf
mov [p_ljf],current_ljf     //更改指针指向当前的 ljf
……
pop prev_ljf
pop p_ljf
mov [p_ljf],prev_ljf
……
```

我们会在后面的 4.6 堆栈回退实现中看到关于 LJF 的使用。

对象句柄结构,这对应于 `local_object_handles` 的类型：

```
struct Object_Handle_Struct {
    Java_java_lang_Object *java_reference;
    Object_Handle_Struct *prev;
    Object_Handle_Struct *next;
    Boolean allocated_on_the_stack;
};
```

从定义的结构中可以看到，所有的 Java 对象在被本地方法使用时传递了两个值，一个是指向这个对象的指针，另外一个是标记是否在栈上分配的状态。所有的传递到本地方法的对象句柄都链成一个双向链。这对于发生 gc 时寻找根集是有帮助的。这在 JNI 方法的时候有用，对于 RNI 和直接调用方式，虚拟机都是知道当前的引用情况的，因此并不需要传递这个引用表。

剩下的几个域除了 eip 之外，都是由调用者保存的寄存器(callee-saved registers)。如果在本地方法中发生了异常，所有的这些信息都是用来进行异常处理的。

6.1.4 调用规则

调用规则是语言中约定的函数调用过程中参数的压栈顺序，以及由谁来清理堆栈的规则。在 Visual C++中，有四种调用规则：即__cdecl,__stdcall,__fastcall,thicall，对应的清栈操作和参数传递顺序如表 xx 所示：

关键字	清栈操作	参数传递
__cdecl	调用者	参数都放在栈上，顺序是从右到左压栈，即我们通常说的 C 语言调用规则
__stdcall	被调用者	参数都放在栈上，顺序是从右到左压栈，即我们通常说的 C 语言调用规则
__fastcall	被调用者	保存在寄存器中(ECX,EDX)，然后压到栈上
thiscall (非关键字)	被调用者	压到栈上, **this** 指针保存在 ECX 中

在 ORP 中，即时编译器使用的调用规则和上面的几种都不一样：参数传递顺序由左到右，并且是由被调用者清栈的。

6.2 编译前工作

我们把在 Methodl 类装入原始的 byte code 到编译为真正的本地代码前的工作统称为编译前工作。这部分每个方法在创建时候生成的 compiler_me_stub，compile_me_stub 中将转去调用的 compile_method_trampoline。Jit_a_method 是在 compile_method_trampoline 中调用的，我们把它放置在这个部分进行描述。

6.2.1 Compile-Me Stub

在第 x 章的基本数据结构描述中，我们可以看到每个方法的入口点都是保存在 Method::_code 中的。而这个部分在 Method 被创建的时候，就被初始化成了：

```
mov eax,method
jmp compile_method_trampoline
```

其中 method 就是当前 Method 的 this 指针(参考：common/class_loader/Class.cpp)。真正的 compile_me_stub 的工作是由 create_compiler_me_stub(参考：arch\ia32\base\compile_IA32.cpp)来完成的。我们看一下它的代码：

```
char *create_compile_me_stub(Method *method)
{
    char *stub = (char *)gc_malloc_fixed_code_for_class_loading(10);
    char *p = stub;
    p = mov(p, &eax_opnd, &Imm_Opnd((uint32)method));
    p = jump(p, (char *)getaddress__compile_method_trampoline() );
    return stub;
} //create_compile_me_stub
```

图 4.1 create_compile_me_stub 函数的实现

这儿首先出现了 mov(…)、jump(…)的方法，在这儿我们做一下解释。在前面也已经提到过，ORP 本身的实现支持了 Windows 和 Linux 两个操作系统，为了很好的组织实现代码，都尽量使用了可移植代码。这样就减少了因为操作系统差异而导致的复杂性。在 ORP 的内部实现中大量使用了这种方法，首先为要发射代码分配一部分空间，然后使用专门指令的生成函数往这个空间中输出指令。在这个例子中的 mov 方法实现了在两种操作系统下对立即数操作的 mov 指令的编码。看一下 mov 方法的内部实现：

```
char *mov(char *inst,const R_Opnd *r,const Imm_Opnd *imm) {
//
// r = imm
//
*inst = (unsigned char)0xb8 + r->reg_no();
return imm->emit32(inst+1);
}
```

图 4-2: 可移植的一种 mov 方法实现

图 4-2 中实现了图 4-1 中调用的 mov 方法，即往某个寄存器中传送一个立即数的指令。这个指令使用 5 个字节，mov 操作前缀是 0xb8，寄存器号和前缀占用一个字节，而立即数的内容要四个字节，在上述代码中 imm->emit32 的方法就是完成了把立即数内容复制到发射的指令中。

在看了 compile_me stub 的实现后，我们来看一下 compiler_method_trampoline。

6.2.2 编译方法的蹦床代码（Compile Method Trampoline）

compiler_method_trampoline 方法接管由 compile_me_stub 传递过来方法调用，调用相应的即时编译方法（缺省的是优化编译器 O3，具体的流程可以参考第三部分的描述），并根据即时编译返回的结果执行代码。

下面看一下具体的代码，我们已经把必要的注释写在了代码中间：

```
// 为垃圾收集或异常处理时的堆栈展开保存状态，
// 这些状态对应的是J2N_Saved_State（Java到本地代码的保存状态结构）
// 由于堆栈布局增长序和内存的布局增长序相反，因此各个字段的压入
// 与J2N_Saved_State中的定义是反过来的。
// EIP的内容已尽被call指令压到栈上。
// 现在压入被调用者保存寄存器(callee-saved registers)。
push ebp
push ebx
push esi
push edi

// 局部对象句柄，这只会在JNI中用到，我们在前面对这个进行了说明
// 压入NULL以表示局部对象句柄列表是空的。
push 0
// 将方法的句柄保存到一个被调用者保存的寄存器中[第三条指令]
mov esi, eax
// 为当前栈帧增加一个到LJF列表的入口
// LJF: Last Java Frame,最近Java帧，我们会在堆栈展开部分描述
call get_addr_of_orp_last_java_frame // 取得最近Java帧
push eax                              //压入p_ljf
push [eax]                            //压入prev_ljf
mov [eax], esp                        //将当前栈指针保存到prev_ljf中
// 编译方法，这一步后将进入O1或者O3的compile_method方法对方法进行编译
push esi                             //方法句柄
call jit_a_method                    //调用jit_a_method
add esp, 4                           //C调用规则的清除参数
// jit_a_method的返回值在eax中，即本地代码的入口点。
// 从LJF列表中删除当前Java帧的入口
pop ecx                              //取出prev_ljf
pop edx                              //取出p_ljf
mov [edx], ecx                       //[p_ljf]=prev_ljf
// 恢复J2N状态中的其余部分
add esp, 4
pop edi
pop esi
pop ebx
pop ebp

// 继续跳转去执行刚刚编译好的方法
```

```
jump eax
```

这部分的蹦床代码是每个Java方法在第一次调用时必然执行的代码。在保存了当前Java的执行状态后，调用JIT部分的方法来对这个Java方法进行编译；编译完毕后，清除当前Java执行状态，然后跳转到新编译好的方法中开始运行。在jit_a_method调用返回后，实际上在相应方法的_code域中，已经更行为指向已经编译过的代码了。因此，第一次以后的调用将直接使用编译过的本地代码。

6.2.3 Jit-a-method

orp\arch\ia32\base\compile_IA32.cpp 中的 jit_a_method 按照方法的属性分别调用合适的“编译”函数。如果要编译的方法是本地代码，我们就需要定位实现这个方法的 C 函数，并且为这个方法产生一个调用的辅助基础代码，如果方法是字节码(Bytecode)，即在 class 文件中的 bytecode，就应该调用合适的即时编译器来生成本地代码。在核心虚拟机初始化的时候，会在整个虚拟机内注册两个编译方法类，即快速代码编译器 O1 和优化代码编译器 O3。针对 Java 代码的编译选用的即时编译器，缺省的将是优化编译器 O3，但可以通过运行时参数或者程序修改来调整缺省的运行参数。在编译为本地代码后，核心虚拟机会调用相应的方法把当前方法的代码位置更新为已编译的方法。参考 compile_IA32.cpp 中的 prepare_bytecode_method 方法。

下面我们分两种情况来说明 jit_a_method 的执行情况，首先我们看一下针对通常的 Java 的编译情况。

6.3 编译通常的 Java

Java 方法的编译都是通过核心虚拟机中的即时编译器接口来完成的，真正的编译是通过即时编译器来完成的。下面首先看一下，ORP 中的即时编译器接口：

6.3.1 即时编译器的接口

ORP 开发的目的之一就是为研究者提供即时编译、自动内存管理的研究平台。所以，即时编译器接口是 ORP 中的一个重要接口。ORP 同时支持多个即时编译器，在实际使用中，可以静态的链接到可知性文件中，或者也可以通过适当的宏选项编译成动态连接库。

核心虚拟机对即时编译器调用的 C++接口在文件 common\include]\jit_intf_cpp.h 中，该接口规定了即时编译器至少应该实现的方法。不管即时编译器内部是如何实现的，核心虚拟机都将通过这个接口来调用即时编译器。

在设计上，即时编译器本身使用 C++设计，而核心虚拟机对即时编译器的调用也是基于 C++的接口进行的，但当编译成动态连接库时，不同的系统在 C++的名字处理上有着不同的处理方式：Linux 系统下的 gcc 和 Windows 系统下的 Visual C++的对名字的处理就不一样，这样会就不能够通过统一的名字来找到该方法。为了避免这种问题发生，ORP 设计上使用了 C 语言的接口来屏蔽这种名字差异。这个部分在逻辑上不属于核心虚拟机部分，应该是由编译成动态连接库的即时编译器提供的，接口描述在文件 orp\interface\jit_export.h 中。当 ORP 从动态连接库中装载即时编译器时，它会为该即时编译器生成相应的代理对象。在 Dll_jit.cpp 中我们可以看到，首先它会从动态连接库中找到各个方法的地址，然后为 Dll_jit 类的各个域赋相应的值，配合 Dll_Jit_intf.h 中的声明的 JIT 接口的覆盖方法，就完成了”代理”的功能。关于这部分的接口描述，可以参考第三部分，即时编译器部分的 JIT 和虚拟机接口描述部分。

6.3.2 ORP 中的即时编译器

在整个 ORP 中，common\base\Compile.cpp 中定义的 jit_compilers 是系统中所有的即时编译器的集合，它被定义为：

```
JIT *jit_compilers[] = {0, 0, 0, 0, 0};
```

在系统启动前，整个虚拟机中并没有即时编译器；在真正启动属于应用的线程后，系统初始化的时，会执行如下的语句（mains\Main.cpp 的 main3 方法）：

```
orp_add_jit(o1_jit = new Level_1a_JIT());
orp_add_jit(o3_jit = new Level_3_JIT());
```

这两个语句会再 jit_compilers 中添加两个即时编译器，即通常的快速代码编译器 O1 和优化代码编译器 O3，在内部实现上 orp_add_jit 保证新加入的即时编译器在 jit_compilers 的首元素中(详细的可以参考 common\base\Compile.cpp)。如果命令行参数指定了-swapjit i j，ORP 会将 jit_compilers 中的第 i 个元素和第 j 个元素互换。

6.3.3 调用真正的即时编译器

前面说过，当 jit_a_method 发现需要即时编译的代码是 java 的时候，就会调用真正的即时编译器来完成编译。具体的，在 arch\ia32\base\Compile_IA32.cpp 的 prepare_bytecode_method 中：

```
for(jit = jit_compilers; *jit; jit++) {
        comp_handle.jit = *jit;
        res = (*jit)->compile_method((*jit)->global_compile_handle, &method, flags);
        if(res == JIT_SUCCESS) {
            break;
        }
```

从这儿也可以看到，为什么缺省的编译器是优化代码编译器 O3 了。该代码用第一个注册的即时编译器编译方法，如果返回成功，就推出循环。在现有的 ORP 中，O1 和 O3 都是返回 JIT_SUCCESS，因此在最前面的即时编译器将成功的编译该方法。缺省的，就是优化代码编译器 O3 了。

6.4 编译本地方法

编译本地代码的处理的主要是为了解决本地代码和 Java 代码之间交互的问题，我们下面首先从基本的处理方法、具体的处理来看一下针对本地代码的处理情况。最后，我们特别的针对 JNI 和 RNI 举两个例子，以提高对这方面的认识。

6.4.1 外围辅助代码(Wrapper)

ORP 会为每一个在运行过程中调用的本地代码生成专门的外围辅助代码。每个辅助代码都会根据给定方法的签名为它作完全相同的工作。这种方法也反映了 ORP 的设计原则，尽量在编译期作更多的事情，以减少在运行期的开销。

外围辅助代码生成函数会找到本地代码的入口，并决定为它选择哪一种接口。内建方法在表中会有内建本地方法的信息。在DLL内的方法如果使用了JNI的命名方法的话就会假定是JNI的接口，如果使用了RNI的命名方法就会假定是RNI的接口。ORP是通过扫描内建方法表来定位本地方法的。如果表中没有需要的方法，ORP就扫描使用RegisterNatives注册的本地方法。如果还是失败，ORP就用某个命名规则的名字来扫描所有装载入的DLL。如果还是没有找到，就会抛出异常。

外围代码生成后，本地方法就会像JIT编译过的方法一样。这个方法的入口点会被纪录，并且如果这个方法是虚方法，在vtable中的相应的入口也会被更新。

6.4.2 本地代码的编译

讨论了编译 Java 方法的处理后，我们来看一下对于要编译的方法是本地代码情况的详细处理。compile_IA32.cpp 文件中的方法 prepare_native_method 实现了处理。

1） 在处理过程中，首先要找到相应的本地方法。寻找本地方法的过程是通过 base_natives\common_olv2\Find_natives.cpp 的 find_native_method 方法来完成的。该方法从对应的 method 中构造出和方法的关联的名字：未重载的 Java 本地接口方法名字、重载的 Java 本地接口方法名字、未重载的内部本地接口(Internal Native Interface)方法

名字以及在动态连接库中的未重载的内部本地接口方法名字。关于 Java 本地接口(Java Native Interface,JNI)，原生本地接口(Raw Native Interface,RNI)的信息，在第 XX 节有论述。然后根据这些名字信息，找到该方法的实体。查找本地代码的过程首先会在内建的本地方法的**基本数组**中按照未重载的内部本地接口方法名字、重载的 Java 本地接口方法名字的顺序查找，如果找到有该名字的方法，就返回该函数的地址；如果未找到，并且没有为标准库使用动态连接库，就会在本地方法的**额外的数组**中按照未重载的内部本地接口方法名字、重载的 Java 本地接口方法名字的顺序寻找；否则会在 **rt.dll** 中查找未重载的 Java 本地接口方法名、重载的 Java 本地接口方法名以及未重载的内部本地接口方法名进行查找；如果找到，返回该函数地址；如果还没有找到，则继续到提供的**用户定义的动态连接库**中查找未重载的 Java 本地接口名、重载的 Java 本地接口名；如果还没有找到，进入该方法的最后一步，在**注册过的方法**中查找具有该方法名字的方法，找到就返回该函数所在的地址；如果没有找到，程序将报告错误。

2） 找到相应的本地方法后，就应该为它生成相应的调用辅助代码。过程会根据本地接口的类型作相应的处理，接着根据方法中参数中的双字的数目来计算辅助代码的大小，并在分配缓冲区中生成相应的辅助代码：

```
push ebp       ;同 compile_method_trampoline,保存 J2N_Saved_State
push ebx
push esi
push edi

push 0          ; JNI 的局部引用列表

if fastcall and ecx used ;如果使用了 fastcall 或者 ecx 使用的话
    push ecx
if fastcall and edx used;如果使用了 fastcall 或者 edx 使用的话
    push edx

call get_addr_of_orp_last_java_frame   ;取得该 Java 线程内的最近 Java 帧(LJF)

if fastcall and edx used
    pop edx
if fastcall and ecx used
    pop ecx

push eax              ;压入 orp_last_java_frame 的地址.
push [eax]            ; 压入 orp_last_java_frame
move [eax], esp     ;把 esp 作为新的 LJF.

;再次压入全部参数
[如果是静态方法，压入类指针]
[如果使用 JNI,如果方法是同步的,在这儿调用 monitorenter;调用 orp_enable_gc]
call func             ; 调用本地方法 func
[如果使用 JNI，在这儿调用 orp_disable_gc；如果方法是同步的调用 monitorexit]
pop ecx               ; 弹出 orp_last_java_frame.
pop ebx               ; 弹出 orp_last_java_frame 的地址.
```

```
mov [ebx], ecx      ; 恢复 orp_last_java_frame 的值

add esp, 4          ;跳过 JNI 的局部引用表
;本地代码会做以下的工作，但由于有可能在调用方法的时候发生拷贝 GC，因此
;重复处理
pop edi
pop esi
pop ebx
pop ebp
ret n               ; n 是根据参数的个数、快速调用的额外偏移量计算得到的
```

上面说明了 jit_a_method 针对本地代码的处理情况，接下来，我们来看一下针对 JNI 和 RNI 两种不同接口方式的详细处理。

6.4.3 JNI 例子

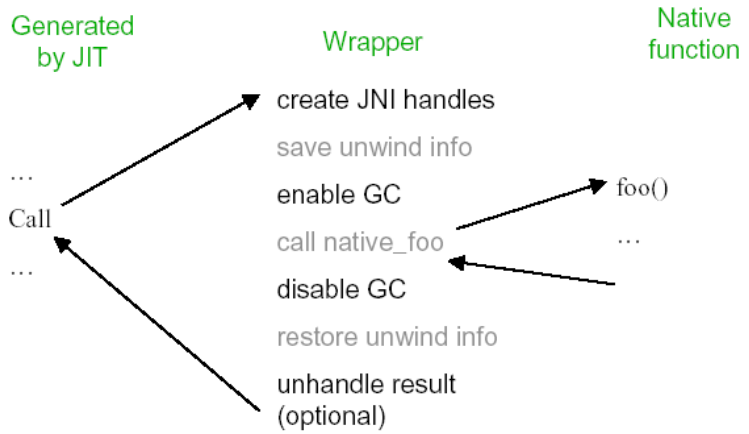


图 xx-xx: 本地代码方法的 JNI 外围辅助代码

图 xx-xx 中显示了针对 JNI 方式的本地代码方法的外围代码的处理框架。下面我们结合具体的例子来看一下外围辅助代码的处理。

考虑关于反射部分实现的本地代码，在 ORP 中使用了 JNI 方式来实现 Java 的方法：

Constructor getConstructor(Class[])

在 JNI 方式中的声明（参考 orp\base_natives\gnu_classpath\java_lang_Class.cpp）：

JNIEXPORT jobject JNICALL

Java_java_lang_Class_getConstructor(JNIEnv *jenv, jobject clazz, jobjectArray parameterTypes)

4.4.2 中简要的说明了处理过程，但并没有针对具体的方法描述详细的处理，在这儿，我们针对这个方法给出详细的处理过程。

从上面可以看出，getConstructor 方法有一个参数，Class[]，加上本身的 class 的 this 指针，在调用 prepare_native_method 之前栈上一共有两个参数：this,以及另外一个指向某个 class 的指针。

发生 call 后，返回地址将被压到栈上。因此，在进入外围辅助代码的处理前，栈上还有返回地址。

然后程序就进入了外围辅助代码的处理：

```
//连同栈上的返回地址，开始构造 J2N 状态的保护
    push ebp
    push ebx
    push esi
    push edi
    push 0
```

```
call get_addr_of_orp_last_java_frame //取得最近Java帧
push eax                              //保存p_ljf
push dword ptr [eax]                  //保存prev_ljf
mov dword ptr [eax],esp               //放入新的ljf
//把当前的栈指针保存在被调用者保存的寄存器中
mov esi,esp
//在栈上分配2个16字节大小的JNI句柄
sub esp,20h
//用反序重新压入2个Java的参数
push dword ptr [esp+40h]            //Class指针
push dword ptr [esp+48h]            //this指针
//为init_object_handles_for_args压入三个参数
push esp                          //重新压入参数
push esi                          //压入ljf
push 924EC8h                      //压入方法句柄
call init_object_handles_for_args
add esp,0Ch
//压入JNI环境指针
push offset jni_env
// 在JNI方法内部，GC应该是开启的
call orp_enable_gc
//调用本地方法
call Java_java_lang_Class_getConstructor
mov ebx,eax                       //保存返回地址
call orp_disable_gc
//如果需要的话，处理结果
or ebx,ebx
je result_is_null
mov ebx,dword ptr [ebx]
result_is_null:
// 为free_local_object_handles压入两个参数
// 注意：free_local_object_handles使用了快速调用(fastcall)的调用规则
// 即，两个参数是由ecx,edx来传递的
mov edx,esp //尾
mov ecx,dword ptr [esi+8] //首
call free_local_object_handles
//如果有需要处理的异常，抛出它（后面会讲到，这属于异步异常的处理）
call get_current_thread_exception
or eax,eax
je no_pending_exception
call rethrow_current_thread_exception
no_pending_exception:
//从栈上除去对象句柄
mov esp,esi
//把结果存回eax
```

```
mov eax,ebx
// 恢复J2N状态
pop ecx
pop ebx
mov dword ptr [ebx],ecx
add esp,4
pop edi
pop esi
pop ebx
pop ebp
// 返回并除去两个参数
ret 8
```

因此相对应的栈上数据的分布情况如图 xx-xx 所示：

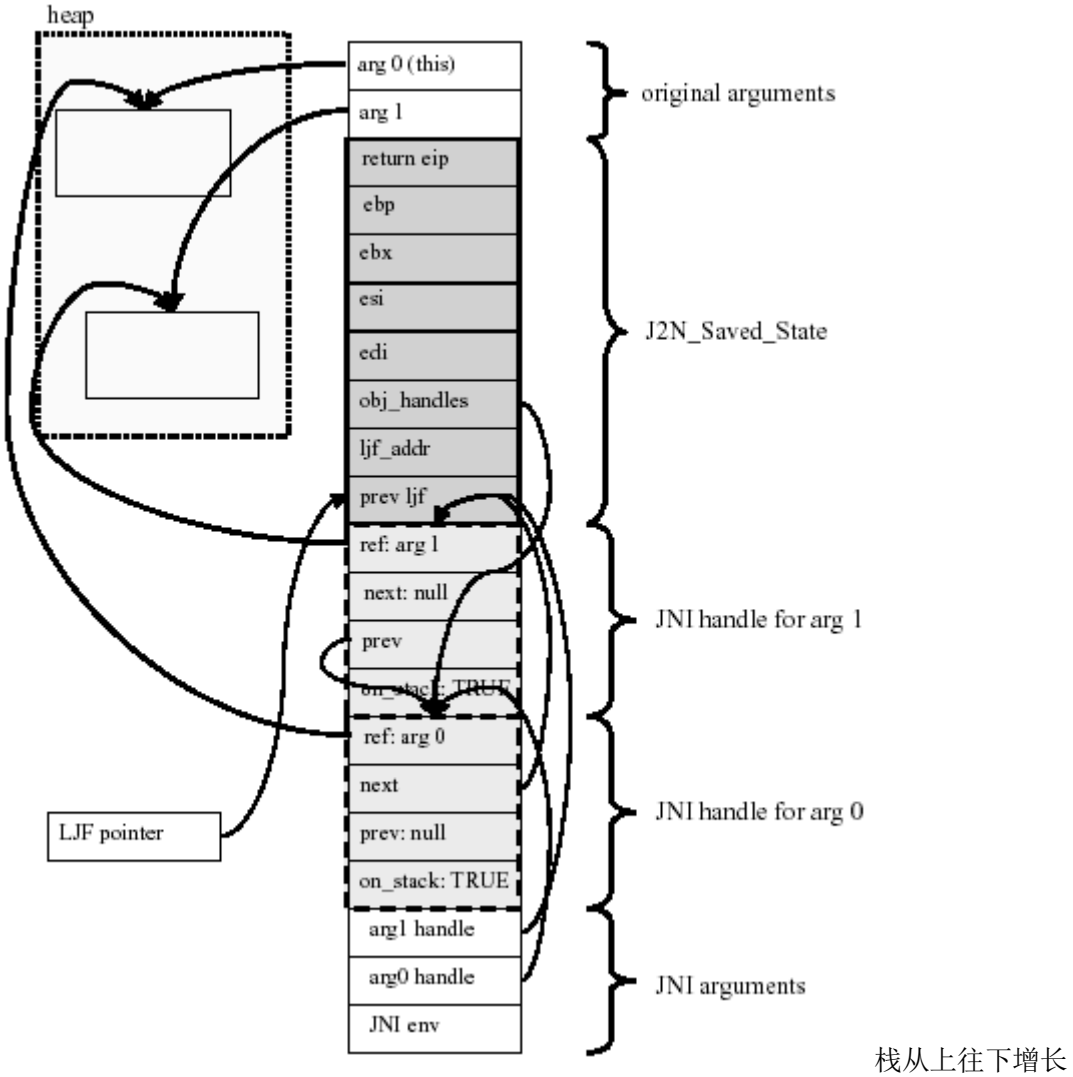


图 xx-xx：JNI 调用时栈的状态

6.4.4 RNI 例子

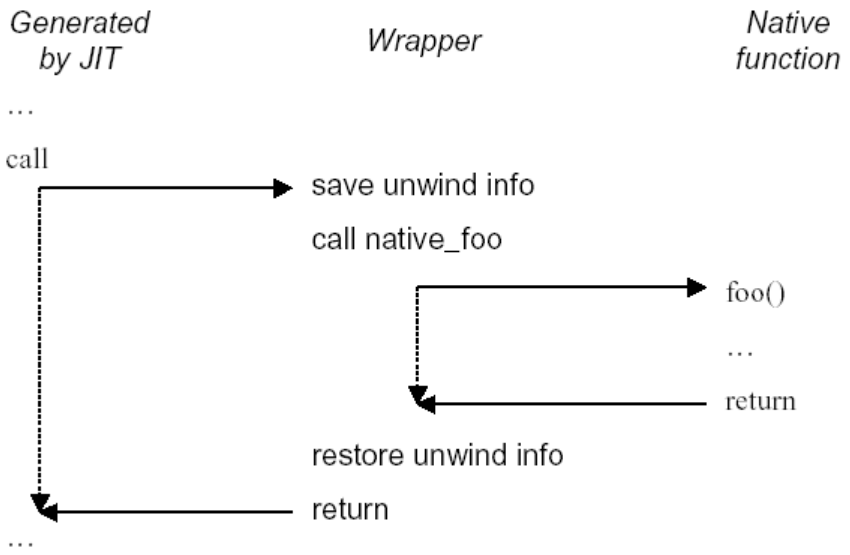


图 xx-xx: 本地代码方法的 RNI 外围辅助代码

图 xx-xx 显示了 RNI 调用时的外围代码的处理情况，和 JNI 相比 RNI 的处理要简单一些。外围的辅助代码的核心是保存堆栈回退信息。简单的来说，主要的工作为：创建最近 Java 帧，重新压入参数，调用本地方法，弹出最近 Java 帧。下面我们结合具体的例子详细的看一下针对 RNI 方式的处理。

考虑如下的 Java 方法声明：

```
public native void java.lang.System.arraycopy
(   java.lang.Object src,
    int srcOffset,
    Java.lang.Object dst,
    int dstOffset,
    int length);
```

以及对应的C代码的实现：

```
java_lang_System_arraycopy(  Java_java_lang_Class *,
                             Java_java_lang_Object *src,
                             int32 srcOffset,
                             Java_java_lang_Object *dst,
                             int32 dstOffset,
                             int32 length)
```

这儿要注意的是ORP中使用的调用规则和C的调用规则不同。所有的由ORP的即时编译器编译生成的代码的调用规则都是从左到右压栈，由被调用者清栈的。第二个不同的是C函数还有一个额外的包含了声明者个方法的类的句柄参数。

与JNI方式的类似，在进入prepare_native_method前，栈上已经有了方法的五个参数以及返回地址；接着程序进入了RNI外围辅助代码的处理：

```
// 保存J2N状态
push ebp
push ebx
push esi
push edi
//对于RNI方法，并不需要保存局部引用句柄，因此一直是空
```

```
push 0
call get_addr_of_orp_last_java_frame  //取得最近Java帧
push eax                              //保存p_ljf
push [eax]                            //保存prev_ljf
//把ljf指向当前创建的最近Java帧
mov  [eax],esp
//重新以反序压入五个Java参数
push [esp+32]    //length，注意：相邻元素间偏移是8,因为push指令会引起sp+4
push [esp+40]    //dstOffset
push [esp+48]    //dst
push [esp+56]    //srcOffset
push [esp+64]    //src
// 对于静态的RNI方法都有一个额外的参数：用有该方法的类的类句柄
push 0xd73674  // java_lang_System
//调用本地代码的arraycopy
call java_lang_System_arraycopy
// C调用规则要求调用者清栈，由于该方法是6个参数，因此是加上24
add esp,24
//从最近Java链上除去当前的LJF
pop ecx
pop ebx
mov [ebx],ecx
//退去0
add esp,4
//恢复被调用者保存的寄存器的内容
pop edi
pop esi
pop ebx
pop ebp
//ORP的调用规则是被调用者清栈的操作，因此清除5*4=20个字节
ret 20
```

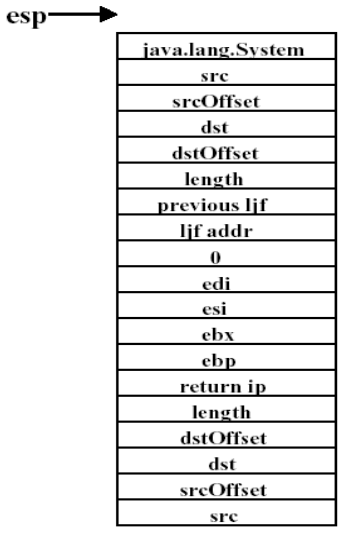


图 xx-xx：arraycopy 方法的 RNI 调用时栈的状态

6.5 异常处理

异常是指程序在执行了语义违背的操作后由语言或者操作系统提供的对于这种操作的处理。我们下面从基本语义，ORP 中对同步异常的实现以及在不同操作系统下异步异常的实现说明 ORP 对于 Java 语义下的异常的实现。

6.5.1 基本语义

Java 的优异特性，还表现在支持异常处理上。这个优异的特性也是 Java 语言能够得到广泛应用的一个重要原因，这种特性保证了提供的代码可以在出现与 Java 语义相违背情况下向由虚拟机向程序发送信号，并可以从异常发生点转到程序员指定点对它进行处理。如果没有合适的异常处理函数，应用程序一般会显示错误信息后退出。例如，向一个接受整型数据的方法传递一个非整型数据，我们可以看到类似这样的信息：

```
java.lang.NumberFormatException: badnumber
        at java.lang.Integer.parseInt
    at sum.sumOfIntsAsStrings
    at sum.printSumOfInts
    at sum.main
```

这对于应用的开发者和使用者，都是非常有帮助的。

Java 语言中为异常处理提供了几个基本的块 try/catch/finally，按照 Java 的语义说明，针对如下的代码：

```
try {
…
}
catch(ExceptionType1 e1) {
…
}
catch(ExceptionType2 e2) {
…
}
catch(ExceptionType3 e3) {
…
}
…
finally{
…
}
```

代码首先会执行 try 块中的代码，如果 try 块中的代码执行过程中发生异常，就会从 try 块中抛出一个异常类型，catch 块将根据这个异常类型来判断是否需要进入处理，如果异常类型匹配，控制就会进入到 catch 块的内部作处理。finally 块会在 try 和 catch 块后得到控制,如果在 try 块内没有异常发生，控制会在 try 块后转入 finally 块；如果有异常发生，控制会从相应的 catch 块转入 finally。

Java 的语义同样也要求，如果在当前方法内部没有对当前的异常类型的处理的话，应该进行回退，直到调用者中有 catch 块捕捉这个异常或者回退到了 main。前面图中的错误信息就是缺省的回退过程中打印出来的信息。

Java 语言规范中规定了这些语义，具体的可以参考《Java Language Specification》的 14.19。一个 try

语句的结果是由 try/catch/finally 三块的执行结果决定的，如果 try 块的执行结果正常，finally 的执行结果正常，那么这个 try 语句的结果就是正常的。如果 try 块的执行中抛出了异常，catch 块执行中捕捉到了这个异常，并且执行结果正常的，finally 块的结果也是正常的，那么执行结果是正常的。只要 catch 块/finally 块的结果中有一个是不正常的，那么整个结果都是不正常的。如果 try 语句执行结果不正常，将进行上述的回退，以对这个异常进行处理。

接下来，我们来看一下在核心虚拟机内异常处理的实现情况。在 ORP 的内部，实现上分成三种异常。一种异常是同步异常，即异常是发生某个不符合语义操作前的结果，如在进行除操作前，发现除数为 0；或者在进行 arrayCopy 前，发现 src 或者 dest 为空，都将引起虚拟机抛出异常。另外一种是 throw 语句抛出的异常，从处理上来说和第一种一样属于同步异常。除此之外的是异步异常，即异常是在不符合语义的动作发生后的结果，但 ORP 并不能立刻发现发生了异常，只能在异常已经发生一段时间后由操作系统提供的设施才能对出现的异常进行处理。在 ORP 中，同步异常的处理是系统无关的，这得益于 ORP 对机器代码的平台无关封装；对于异步异常，由于操作系统的不同，内部的实现有一些不同。我们将分别对同步异常、以及异步异常在 Windows、Linux 下的实现进行描述。

6.5.2 同步异常在 ORP 中的实现

在开始讲同步异常的处理之前，我们先来看一下 ORP 中对异常处理作的优化。在即时编译器的部分我们会看到关于懒惰异常处理得更多的细节。在这儿我们简单的说一下它的基本思想。ORP 的设计人员在对大量的 Java 应用程序运行的结果进行分析中发现，抛出的异常对象在大多数的异常处理中，相当大的一部分只是使用到了它的类型信息而并不需要执行异常对象复杂的初始化、构造函数。因此，在 ORP 的优化即时编译器中，在编译异常处理时会进行简单的数据流分析来检查是不是使用了异常对象，如果没有用到就可以简单的用异常对象的类型代替异常对象，来减少运行期的开销。当然，关于这部分的处理还是比较复杂的，更详细的讨论，参考本书第 9 章的优化即时编译器部分的异常处理部分的论述。

在 java.lang.system 中 arrayCopy 中，在进行所有的拷贝操作前，虚拟机会检查源对象和目标对象的指针，如果有一个为空，就执行 throw_java_exception。部分代码如下：

(详细的请参考 orp\base_natives\gnu_classpath\java_lang_system.cpp)

```
java_lang_System_arraycopy(Java_java_lang_System *,
                           Java_java_lang_Object *src,
                           int32 srcOffset,
                           Java_java_lang_Object *dst,
                           int32 dstOffset,
                           int32 length)
{
    if(!(src && dst)) {
        throw_java_exception("java/lang/NullPointerException");
    }
……
}
```

图 xx-xx: arrayCopy 的部分实现

throw_java_exception 的实现可以参考 orp\common\ base\Exceptions.cpp，在这儿我们简单的说一下它的处理过程：

首先，根据异常名找到对应的异常类；

然后，创建异常对象（懒惰异常处理中，并没有这一步）；

最后，调用 throw_java_exception_from_native 抛出异常。

throw_java_exception_from_native 函数会根据当前的最近 Java 帧回退到上一个 Java 帧。然后在相应

的 Java 方法中调用 orp_athrow 方法。orp_athrow 是 ORP 中对于同步异常处理重要功能函数，它的主要功能是找到合适的异常处理例程，然后转移到处理例程中。因此这个过程永远不会返回。orp_athrow 的详细处理中，首先从参数中构造 Frame_Context，然后传递这个结构给 orp_throw，根据 orp_throw 返回的结构通过 transfer_control 进行控制流的转移。具体的 orp_athrow 的代码如下：

```
void __stdcall orp_athrow(      Boolean is_first,
                                volatile uint32 edi_arg,
                                volatile uint32 esi_arg,
                                volatile uint32 ebx_arg,
                                volatile uint32 ebp_arg,
                                volatile uint32 esp_arg,
                                volatile uint32 eip_arg,
                                volatile Java_java_lang_Object *obj)
{
    Frame_Context fc;
    fc.p_edi = (uint32 *)&edi_arg;
    fc.p_esi = (uint32 *)&esi_arg;
    fc.p_ebx = (uint32 *)&ebx_arg;
    fc.p_ebp = (uint32 *)&ebp_arg;
    fc.p_eip = (uint32 *)&eip_arg;
    fc.esp     = esp_arg + 4;   //清除栈顶的元素？？？？？
    fc.ljf     = (uint32)get_orp_last_java_frame();
    orp_throw(&fc, (volatile Java_java_lang_Object **)&obj, is_first);
    transfer_control(&fc, (uint32)obj);
} //orp_athrow
```

下面分别的来看一下 orp_throw 和 transfer_control 的处理。

orp_throw 的处理是 ORP 中实现异常处理的主要逻辑。这部分也是懒惰异常处理优化的一个重要部分。在进入 orp_throw 的时候有可能有两种情况，一种情况传递进来的就是异常对象，另外一种情况下传递进来的是异常对象类，分别对应了通常的异常处理和懒惰异常处理。然后根据传递入参数，找到对应的异常对象类。在处理上，对于第一种情况，只要通过 obj 对象的 vtable 找到对应的 class 就可以了。即 clss=obj->vt->clss;对于第二种情况的，实际上 obj 就是该异常类了，clss 取得的是 java.lang.Class，因此在代码中通过 clss 和 java.lang.Class 的比较来判断是否是异常对象。接着，处理例程将复制当前的执行上下文，以在必需的时候根据这个信息来构造堆栈跟踪信息(stack trace)。然后，就进入了通常的异常处理的执行顺序。

根据发生异常的 eip 信息，找到相应的 Java 方法的特定即时编译信息(JIT_Specific_Info)；

根据发生异常的 eip 信息，判断发生异常时所处代码的类型；

如果发生异常所在的 eip 的代码类型为 ORP_TYPE_JAVA，即当前执行的代码是由 Java 代码即时编译出来的，就进入循环开始寻找相应的异常处理例程：

从 JIT_Specific_Info 中取得当前方法的异常处理函数；对于每个处理函数，检查当前 eip 是不是在它的捕捉的范围内，并且判断当前的异常对象类型是否相容，如果条件符合，即找到了合适的异常处理例程，调用 JIT 部分的函数 fix_handler_context，????，取得异常处理例程的 eip 并保存到自身的上下文结构中。然后将本线程内的异常对象清空。如果是懒惰异常处理， 将调用 create_object_lazily 来懒惰的创建这个异常对象。如果该对象已经在即时编译阶段证明是死对象，那么这个调用并不会创建这个对象；如果不是死对象，由于精确异常的要求，要求在异常发生的地方创建这个对象，因此需要将当前线程的 throw_context 恢复成异常发生时的上下文。到这儿，就已经找到了对应的处理函数了，函数能够返回了。如果 eip 不在当前异常处理不作的范围内或者异常对象类型不相容，那么就需要堆栈回退来找到合适的

处理例程。在回退前，首先要根据当前方法的属性作适当的清除工作：这里面主要涉及的是当前方法如果是同步方法，在进行堆栈回退前，必须释放监视器的使用。然后调用即时编译器的接口函数: unwind_stack_frame 来进行堆栈回退。如果回退后栈帧 eip 所处的类型仍然为 ORP_TYPE_JAVA，在找到相应的 JIT_Specific_Info 后继续循环；如果回退后栈帧 eip 所处的类型不是 Java 代码，所作的处理就是以下三步：首先，如果是懒惰异常处理，创建这个异常对象；其次，向当前的 orp_thread 结构中放置相应的异常对象；最后，在本地代码中恢复执行，就像 Java 方法已经返回一样。流程如图 xx-xx 所示。

图 xx-xx: orp_throw 的处理过程

transfer_control 的工作比较直接，它接受函数参数中传递的上下文，然后装入到相应的各个寄存器中，直接进入到相应的上下文中作异常处理。

从解说我们可以看到，orp_athrow 的执行很好的对应了 JVM 中对应于 athrow 的语义。在具体实现上，它分成了寻找当前应用的异常处理例程的 orp_throw 和转去真正执行异常处理的 transfer_control 两个部分。这可以使得在异步异常的处理过程中重用旬早异常处理例程的部分，在一定程度上提高的代码的重用性。

从上述对异常处理过程的解说，我们也可以看到同步异常的同步也表现在控制流的直接转移上。而异步异常只能在发生异常后由操作系统的其他设施来通知当前进程才能够进行处理。下面我们来看一下异步异常在 Windows 和 Linux 下的实现。

6.5.3 Windows 下 ORP 的异常处理

Windows 系统本身提供了结构化异常处理（Structured Exception Handling），对于实现 ORP 的异常分发不无裨益。在展开讲述 ORP 在 Windows 下的异常处理前，我们先来看一下 Windows 下的结构化异常处理。

Windows 下提供了丰富的结构化异常处理方法，在这儿我们只说明 ORP 中使用的部分。关于结构化异常处理得更多的内容，请参考本章末的参考文献。ORP 中使用了 Visual C++提供的__try/__except 语句。这两个语句的基本语义，和 Java 中提供的 try/catch 模式有点类似。

基本语法

```
__try{
……
}
__except(expression)
{
……
}
```

这些语句的基本语义：__try 块是监视块，在这个块里面的任何一个代码引起执行异常都将引起控制流转移到__except 块。然后进行计算 expression，由它的值来决定如何处理。expression 可以取三个值：

EXCEPTION_CONTINUE_EXECUTION (–1)，表示忽略当前的异常，从发生异常的地方继续执行；

EXCEPTION_CONTINUE_SEARCH (0)，异常没有被识别，继续到栈上去寻找处理例程；

EXCEPTION_EXECUTE_HANDLER (1)，异常被捕捉，执行__except 块的处理例程。然后转到处理例程后的语句执行。

同时,也封装有取得当前异常信息的例程：

```
PEXCEPTION_POINTERS GetExceptionInformation();
```

其中返回值类型的定义：

```
typedef struct _EXCEPTION_POINTERS {
```

```
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

ExceptionRecord 中提供了机器无关的异常描述；而 ContextRecord 中是特定于处理器的发生异常时处理器状态，对这个结构的修改将引起程序执行上下文的改变。我们在后面将会看到，就是使用这个特性来转移到异常处理例程中去的。

在实际的实现中，Windows 下的异常处理非常简洁。ORP 的线程抽象中，对 Windows 部分的处理是从 orp\os\nt\ia32\nt_exception_filter.cpp 的 call_the_run_method2 方法开始的，具体的代码可以如下：（参考 orp\base_natives\common_olv2\thread_generic.cpp）

```
p_orp_thread->thread_handle = (HANDLE)_beginthreadex(
                                (void *)0, (unsigned)(64*1024),
                                (unsigned(__stdcall *)(void *))call_the_run_method2,
                                        (void *)p_orp_thread,
                                        (unsigned)0, // Just start the thread.
                                        (unsigned *)&p_orp_thread->thread_id);
```

在 call_the_run_method2 中的处理，除了在线程的局部分配中划出一部分作局部存储(Local Storage)外，就是调用 call_the_run_method3 了，该函数实体如下：

```
int call_the_run_method3( void * p_xx ){
    LPEXCEPTION_POINTERS p_NT_exception;
    int NT_exception_filter(LPEXCEPTION_POINTERS p_NT_exception);
    __try {
        call_the_run_method(p_xx);
        return 0;
    }
    __except ( p_NT_exception = GetExceptionInformation(),
        NT_exception_filter(p_NT_exception) ) {
          return 0;
    }
```

从上述的代码可以看出，利用 Windows 提供的结构化异常处理机制，可以统一的进行异常的捕捉，计算处理的过程是由 orp\os\nt\ia32\nt_exception_filter.cpp 的 NT_exception_filter 过程来完成的：

```
int NT_exception_filter(LPEXCEPTION_POINTERS p_NT_exception)
{
    Global_Env *env = ORP_Global_State::loader_env;

    ORP_Code_Type orpct =
        orp_identify_eip((void *)p_NT_exception->ContextRecord->Eip);
    if(orpct != ORP_TYPE_JAVA) {
        // 对于代码类型不是 Java 代码的情况，返回继续寻找
        return EXCEPTION_CONTINUE_SEARCH;
    }

    volatile Java_java_lang_Object *exc = 0;
    switch(p_NT_exception->ExceptionRecord->ExceptionCode) { //异常代码
    case STATUS_ACCESS_VIOLATION: //访问异常
        {
```

```
            //懒惰异常对象创建
            exc = (Java_java_lang_Object *)env->java_lang_NullPointerException_Class;
        }
        break;

    case STATUS_INTEGER_DIVIDE_BY_ZERO://被零除
        {
            exc = (Java_java_lang_Object *)env->java_lang_ArithmeticException_Class;
        }
        break;

    case STATUS_PRIVILEGED_INSTRUCTION://特权指令
        {
                //某些指令只能在特权态(privilge Level,对于 IA32 的 ring 0 执行，
                //在通常的用户态(Ring3)调用会引起/特权指令异常[1]
            // ORP 的即时编译器使用"outs" (操作码为 0x6e)当作"int 3"
            //  在即时编译过的代码中设置断点(for stuff like enumerating
            // live references.)
            //  把断点的 eip 信息保存到 Java 的线程块中这样可以让
            // orp_at_a_jit_breakpoint()恢复它

            uint8 opcode = *( (uint8 *)(p_NT_exception->ContextRecord->Eip) );
            switch (opcode) {
            case 0x6e:
            case 0x6f:
                {
                    void *getaddress__orp_at_a_jit_breakpoint();
                    void *address_of_at_a_jit_breakpoint = getaddress__orp_at_a_jit_breakpoint();

                    p_TLS_orpthread->regs.eip = (uint32)(p_NT_exception->ContextRecord->Eip);

                    p_NT_exception->ContextRecord->Eip=
                            (unsigned long)(address_of_at_a_jit_breakpoint);
                    break;
                }
            default:
                {
                    //其他的出错情况
                    orp_exit(99553);
                }
            }
            return EXCEPTION_CONTINUE_EXECUTION; //忽略异常，继续执行
```

[1] 注：在 IA32 结构上，提供了 ring0-ring3 四个级别的运行态保护。但是在成熟的实际的操作系统设计中，如 Windows NT/2000/XP,Linux 等，只使用了两个状态：ring0，即所谓的特权态；ring3，即用户态。用户态的程序只能运行操作系统和处理器规定的指令集，任何执行非允许的特权指令都会引起特权指令异常。

```
        }
        break;

    default:
        return EXCEPTION_CONTINUE_SEARCH;      //继续回退，寻找合适的处理
    }

    Frame_Context fc;
//取得发生异常时的上下文
    fc.p_edi = (uint32 *)&(p_NT_exception->ContextRecord->Edi);
    fc.p_esi = (uint32 *)&(p_NT_exception->ContextRecord->Esi);
    fc.p_ebx = (uint32 *)&(p_NT_exception->ContextRecord->Ebx);
    fc.p_ebp = (uint32 *)&(p_NT_exception->ContextRecord->Ebp);
    fc.p_eip = (uint32 *)&(p_NT_exception->ContextRecord->Eip);
    fc.esp     = p_NT_exception->ContextRecord->Esp;
    fc.ljf     = (uint32)get_orp_last_java_frame();

    orp_null_ptr_throw(&fc, &exc); //这儿实际上是调用 orp_throw，取得相应的处理上下文，
                                   //如果必要的话生成相应的异常对象

//修改异常结构中的上下文，以转去执行异常处理例程
    p_NT_exception->ContextRecord->Esp = fc.esp;
    p_NT_exception->ContextRecord->Eip = *fc.p_eip;
    p_NT_exception->ContextRecord->Ebp = *fc.p_ebp;
    p_NT_exception->ContextRecord->Ebx = *fc.p_ebx;
    p_NT_exception->ContextRecord->Esi = *fc.p_esi;
    p_NT_exception->ContextRecord->Edi = *fc.p_edi;
    p_NT_exception->ContextRecord->Eax = (uint32)exc;

    return EXCEPTION_CONTINUE_EXECUTION;//转去执行异常处理例程
} //NT_exception_filter
```

从异常过滤处理例程的处理过程来看，基本的处理方法和同步异常的处理类似：都是通过 orp_throw 来获得相应的处理例程然后转到真正的处理中去。不同的是同步异常抛出并不等待真正的异常发生就转去执行异常处理例程了；而异步异常处理中，是在异常发生后由程序捕捉到后才转去作处理的。

从上面的实现中也可以看出，实际上在 ORP 内部，只有空指针引用异常是异步异常。

6.5.4 Linux 下 ORP 的异常处理

在 linux 下的异常处理相对而言，并不只是语言上的提供的特性可以解决的。这主要取决于操作系统的设计。在 Windows 系统下，提供了用户态的异常处理的接口，因此可以使用这部分接口——结构化异常处理做我们希望做的工作。在 Linux 系统下，并没有提供类似的用户态接口，但却提供了信号处理这种机制来完成类似的工作。下面，我们就来看一下 linux 下异常的处理过程。

在 Java 的语法下，使用

```
try {
……
}
```

```
catch(ThrowableExcp1 e) {
……
}
finally{
……
}
```

根据前面对同步异常和 Windows 下的异步异常的处理认识，如果 try 块内出现了空指针引用的问题，而同步异常处理没有捕捉到这种异常，那么应用程序会出现异常。按照 linux 缺省的处理，产生异常的代码将产生信号而导致核心虚拟机的运行退出，这是虚拟机所不允许的。我们必须捕捉这些信号并且控制这些信号来为虚拟机服务。很简单，在 try 块内的语句出现异常，我们必须捕捉到这个异常信号，然后控制程序流转去执行 catch 或 finally 的处理。

接下来，看一下一般信号的处理在 linux 下的处理。如果出现异常，系统将陷入内核，内核根据目前引起异常的原因，查看当前进程中的信号处理设置。如果该信号是缺省处理方式，系统将在创建 core 文件后退出该进程。如果设置了处理函数，内核将在进程控制块的信号字段中加入相应的信号的级或标识，在堆栈中构造相应的中断处理环境，并在堆栈中送入 signum,在当前进程的堆栈中放入信号处理的例程的地址，然后内核将调用 ret_from_intr 引起重新调度。Linux 的调度模块如果在对调度中选择该进程，发现该进程有信号要处理，就会转去执行该信号的处理。Linux 下的 signal 调用的声明：

```
void (*signal(int signum, void (*sighandler)(int)))(int);
```

但 BSD 的声明：

```
void (*signal(int signum, void (*sighandler)(int,context)))(int);
```

context 中提供了当前进程的上下文执行环境，以在完成相应的处理后恢复当前进程的上下文。缺省的，如果出现异常，context 中的 eip 指向了出现这个异常的指令。为了跳过这条指令，我们只需要控制 context 中寄存器结构中的 eip 字段。

实际上在 linux 的内部也有相同的参数，只是没有提供给我们使用。在 C 的处理下，通常的函数在执行前为了维护自己的调用帧，都有这样的操作：(MASM 汇编格式)

```
push ebp
mov ebp,esp
```

而所有的参数的寻址都是通过 EBP+偏移量进行的。我们看一下在一个函数调用发生时的栈的情况：

栈顶显然是刚压入的 EBP

然后是函数执行完成后的返回地址 nextIP

然后是第一个参数….

因此，对于函数参数的寻址上，我们实际上可以通过 EBP+8 来选择第一个参数，而 EBP+12 选择第二个参数。（做了参数是 32 位假设）因此，对 context 的处理，我们可以使用 ebp 来寻址。

由于 linux 系统下的 glibc 的关系，signal 的处理更加复杂。为了得到 POSIX 兼容的处理，glibc 对 signal 进行了封装。在真正的调用系统的 signal 的处理前，还做了一些其他的操作，因此为了达到控制返回地址的目的，我们还是需要找到系统中真正的 context 的位置。

在这部分的处理中，使用了一点技巧：

context 的块在 glibc 的处理中存在多块，为了找到最后起作用的块，我们应该在这部分找到它。

在 ORP 的实现中使用的方法：使用当前的保存的 eip 作为标签，然后在当前的调用帧中扫描，以取得最后传递给系统的 singal 处理时使用的 context。

在 linux 系统下，在 ORP 进行初始化的过程中，执行了：

initialize_signals() ,它完成的主要工作就是挂接一些相关的信号处理函数。

在它的内部实现上，使用了 SIGTRAP 来制作标签：(AT&T 汇编格式)

```
//设置 SIGTRAP 的处理方法为 locate_sigcontext
signal(SIGTRAP, (void (*)(int))locate_sigcontext);
```

```
//下面的汇编代码将导致由操作系统传递给本身一个 SIGTRAP 信号
asm(
//Call 0 ，只是在堆栈上保存了下一指令的地址
    ".byte 0xe8\n\t.long 0\n\t"       // call 0

    //取出当前指令的地址
    "popl    %%eax\n\t" // 1 byte
//加上偏移量 9（每条汇编后面的字节数累加），确定单步后的 eip
    "addb    $9,%%al\n\t"     // 2 byte

    //将发生单步时的 eip 存入静态全局变量 exam_point
//exam_point 用于定位 sigcontext。这是因为当"int 0x3"发生时，
//操作系统会把返回地址保存到信号的上下文中，以用于信号处理完后
//恢复原有程序的运行。这儿使用的技巧是保存地址到 exam_point 中，
//因此我们可以通过和 exam_point 比较保存在信号上下文中的返回的 EIP
//来识别正确的栈帧
    "movl    %%eax,%0\n\t"  // 5 byte

    //触发 SIGTRAP：触发 int3 到系统内核将发送 SIGTRAP 信号到当前的进程，
//在用户栈上设置好信号栈帧，然后返回用户态运行，即调用 locate_sigcontext()
    "int     $0x3" : "=m" (exam_point)
);
```

locate_sigcontext 的处理，使用了两种信号上下文的查找方法来作处理。一种是使用 sigcontext，即 linux 的内核实现使用方法;另外一种是使用 ucontext,在某些其它平台上有使用。具体的,locate_sigcontext 使用 exam_point 作为标签，比较当前活动调用帧中的 sigcontext 中 eip 的值是否与 exam_point 相等。如果相等，使用该纪录中的 ebp 为新的调用帧继续寻找。直到重复三次为止。针对不同的 glibc 版本,实现上有差异，因此处理上使用了不同的方法。glibc 2.1.2 中的 Linuxthreads 为我们的处理函数提供了 sigcontext。但实际上内核并不使用这个部分来恢复进程的上下文，因此需要继续寻找内核创建的 sigcontext。但在 glibc2.2.3 中，修改内核创建的 sigcontext 是没有用的。在真正的信号处理处理完后，LinuxThreads 还会做一次拷贝而覆盖内核创建的 sigcontext。因此，为了避免出现复杂的版本相关的处理，ORP 中的处理既修改了内核创建的 sigcontext 又修改了 linuxThreads 创建的 sigcontext。如果 sigcontext 结构中对应的 eip 地址和 exam_point 不相等，就尝试使用 ucontext 结构来检查，如果 ucontext 结构的相等，则设置相应的标志，即 use_ucontext 为真。

在做了上述的 hack 之后，可以初始化真正的信号处理函数：

SIGSEGV　　段违例信号，一般访问非法异常

SIGFPE　　除 0 错误

SIGINT　　中断处理信号处理，强制退出

在初始化的时候，把它们相应的处理函数设置成了 null_java_reference_handler, null_java_divided_by_zero_handler 和 interrupt_handler.

代码如下：

```
signal(SIGSEGV, null_java_reference_handler);
    signal(SIGFPE, null_java_divide_by_zero_handler);
    signal(SIGINT, (void (*)(int)) interrupt_handler);
```

我们在这儿看一下 null_java_reference_handler 的处理：

基本的处理方法和 Windows 下的类似，不同的是由于信号传递和 glibc/linuxthreads 的封装而引起的稍稍的复杂处理。进入处理，首先要从寄存器中取出 ebp 的内容，然后根据在 locate_sigcontext 的过程中检查到的系统的信号上下文嵌套次数取到系统信号上下文对应的 ebp。然后根据从 locate_sigcontext 过程中设置的标志 use_ucontext 来取得当前的信号上下文，如果是使用了 sigcontext 的话，还要取得最顶部的信号上下文。从信号上下文结构中取出 eip 来进行代码识别，同样的如果这儿检查到当前的 eip 的内容是特权指令"outs"的话，修改信号上下文的中的 eip 为取得 JIT 断点的代码后返回。否则，根据当前的上下文和当前的异常类调用 orp_null_ptr_throw，以取得异常处理的上下文。最后，设置系统的信号上下文和顶部的信号上下文信息为异常处理上下文。信号处理完毕返回时，将转移到异常处理例程中去处理。

对于 null_java_divide_by_zero_handler 的处理情况和上述过程一样，因此不再重复。至于 SIGINT 完成工作和异常处理没有关系，只是作退出的辅助处理。

至此，我们已经介绍完了 Java 中异常处理在 ORP 中的实现。

6.6 堆栈回退（Stack UnWinding）

堆栈回退是指调用帧的回退，即由当前的调用帧回退到调用者的调用帧，通常这个过程至少会执行一遍。堆栈回退是 Java 进行异常处理、垃圾收集和安全相关操作中必须的操作。因此，我们在这儿对它的实现作一个介绍。

在 ORP 中，可以把堆栈回退概括为两类，一类是破坏性的回退，即如果回退，执行流将永久更改。这主要用在异常处理中，如果当前的栈帧内没有合适的处理例程，ORP 就会进行破坏性的回退，直到找到相应的异常处理例程或者回退到了顶部—main 方法。另外一类是非破坏性回退，即回退并不会引起执行流的更改，而只是拷贝了上一个栈帧的内容，这主要用在发生垃圾收集时，需要在各个调用帧中扫描活动引用，垃圾收集完毕，执行流从原来暂停的地方继续执行；在执行安全相关的检查时，也需要非破坏性的回退，以检查调用者的访问控制是否也符合相关的访问控制，如果成功，执行流继续执行。我们在这儿主要讨论垃圾搜集时进行的非破坏性回退。

我们在 4.1.3 中提到过，最近 Java 帧是 ORP 本地代码中的支持回退的结构。前面说过，在 Java 的应用真正运行的时候，并不能够保证所有的方法都是 Java 方法，因此在栈上有可能存在 Java 方法和本地方法混合的情况。为了能够在堆栈回退过程中跳过这些本地方法，需要用专门的外围辅助代码来维护这些结构，在 4.4 的编译本地方法中我们已经看到了详细的处理了。在 ORP 线程的结构中的 last_java_frame 总是指向了最近 Java 帧。由于整个这部分的信息是链起来的，通过这个信息可以找到所有的在栈上活动的本地方法调用。

我们先来看一下发生 GC 时使用的只读式堆栈回退在 ORP 中的具体实现：

```
Boolean unwind_to_next_java_frame(Frame_Context        *context,
                                  ORP_thread           *p_thr,
                                  Boolean               is_first
                                  )
{
    context->ljf = (uint32)p_thr->last_java_frame; //取得该 ORP 线程的最近 Java 帧
    uint32 eip_var = *(context->p_eip);            //取得 J2N_Saved_State 中的返回地址
    JIT_Specific_Info *jit_info = methods.find_deadlock_free((void *)eip_var);//取得 IP 相关的 JIT 信息
    ORP_Code_Type orpct = orp_identify_eip((void *)eip_var);  //取得 IP 相关的代码类型

    while(1) {
        if(orpct == ORP_TYPE_JAVA) {   //如果是 Java 代码
            jit_info->get_jit()->unwind_stack_frame(jit_info->get_method(),
```

```
                                                    context,
                                                    is_first);//调用相应的 JIT 进行回退
        } else {
                //否则是本地代码，取得当前 J2N 状态中的返回地址，取得前一个 LJF 及其他信
            息
            Boolean ok = ro_unwind_native_stack_frame(context);
            if(!ok) {
                return FALSE;
            }
        }
        eip_var = *(context->p_eip);//取得返回地址
        jit_info = methods.find((void *)eip_var);//这一步根本不需要!!
        orpct = orp_identify_eip((void *)eip_var);
        if(orpct == ORP_TYPE_JAVA)
            return TRUE;
    }

    return FALSE;
} //unwind_to_next_java_frame
```

图 xx-xx：unwind_to_next_java_frame 的处理

从图 xx-xx 中的代码可以看到，代码的基本处理方法：首先识别当前的代码类型，如果是 Java 代码，则直接 JIT 的回退处理例程进行回退；否则，取得前一个 LJF。根据获取的新的上下文中的 EIP 进行判断，如果现在的 EIP 的代码类型是 Java 代码，则返回；否则继续上述过程。

从上面的代码和分析中可以看到，对于 Java 代码的堆栈回退的处理时通过 JIT 接口方法：

virtual void unwind_stack_frame(Method_Handle　method, Frame_Context *context);

来完成的。由于快速代码生成器和优化即时编译器生成代码的策略不同，产生代码的运行期布局不同，因此这部分只能由即时编译器来提供回退支持。而通用的程序代码框架：

```
JIT_Specific_Info * jit_info;
jit_info = methods.find(ip);
...
JIT *jit = jit_info->get_jit();
Method *method = jit_info->get_method();
jit->unwind_stack_frame(method, context);
```

下面我们结合具体的例子来看一下堆栈回退的具体处理：

如图 xx-xx 所示，如果此时在本地调用帧中需要进行回退，我们来看一下处理过程：首先将根据当前的 EIP 判断代码类型，很显然代码类型不是 Java 代码，因此，将 context 的 eip 中装入了当前的 LJF 中的返回地址，并将 context 中的 ljf 指向了 prev_ljf（位于图中部），其他部分的信息复制到 context 的相应域中。此时，取得代码类型的时候便得到了是 Java 类型的代码（因为返回地址一定属于图中离栈顶最近的 Java 帧），函数成功返回。

如果图中的本地代码后又有了 Java 帧，而此时需要回退，我们看一下处理过程。由于当前处于 Java 类型代码中，因此调用相应的 JIT 部分的回退处理函数，得到相应的 EIP 地址，很显然，这属于本地代码帧，因此重复上一段描述的处理。

对上述两种情况的描述可以清楚地看到 LJF 的作用。

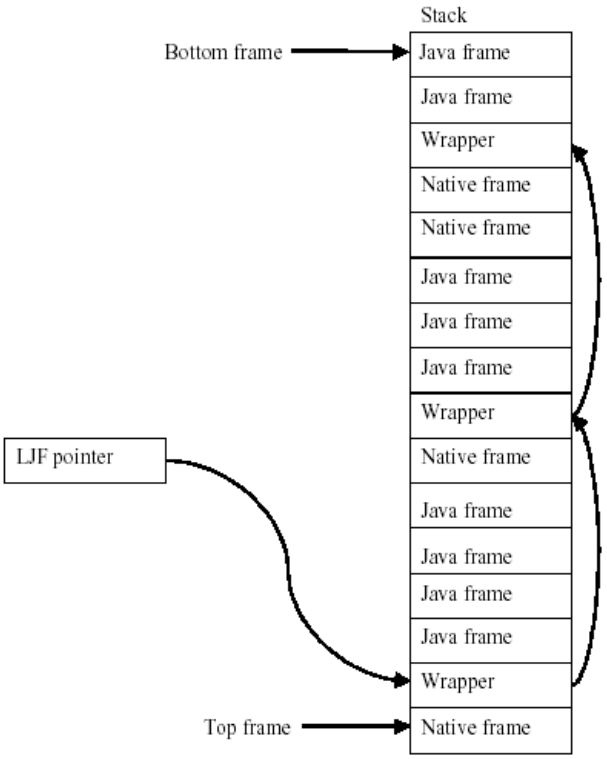


图 xx-xx：运行期堆栈

从图 xx-xx 的代码中我们可以看到，实际上对于堆栈回退，我们可以归纳一下两大类处理：Java 代码的回退和本地代码的回退。在 Java 代码的回退中，又可以分类，破坏性回退和非破坏性回退。破坏性回退和非破坏性回退直接的差别在于对同步方法的处理上，破坏性回退必须在堆栈回退前释放对监视器的控制（更详细的信息，请参考代码：arch/ia32/base/stack_manipulation_ia32.cpp 中 unwind_to_next_java_frame 相关实现）。本地代码回退中，我们也可以分为两类，一类是回退过程中修改 ORP 线程的 LJF 指针的破坏性回退，另外一类是不修改 ORP 线程 LJF 指针的只读回退。他们直接的差别只在于是否修改 LJF 上。

参考文献

Java Virtual Machine Specification, Second Edition

Java Language Specification

Open Runtime Platform Source code 1.0.9 available at http://sourceforge.net/projects/orp/

Java Native Interface Specification.http://java.sun.com/products/jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html

Richard. W. Stevens, Advanced Programming in Unix Enviroment, Addison-Wiley

Understanding the Linux Kernel, O'Reilly, 2001

第7章 垃圾收集器支持

垃圾收集器在整个虚拟机中充当了内存管理者的角色，它的性能的好坏也是 Java 虚拟机性能好坏的重要来源。垃圾收集的主要功能在于在管理的内存区域内，保存活动对象，回收死对象空间。所谓活动对象，是指在运行期活动代码中有对该对象的引用。直观的，我们可以把它分成两类：一类是全局引用；另外一类，是在活动调用栈上的局部引用。垃圾收集器从这些基本的活动引用集出发，递归的扫描引用对象内各个引用域，就能够找到所有的活动对象。在做收集时，基于这些信息作适当的动作。

ORP 中垃圾收集器通过和核心虚拟机之间的接口而独立成为了一个组成部分。在本章，我们主要的来看一下 ORP 中核心虚拟机对垃圾收集器部分的支持。这主要体现在找到活动引用集，并根据活动对象的各个域的类型来继续扫描活动对象。

7.1 根集的枚举（Root-Set Enumeration）

一般的，把所有的活动对象称作根集。我们来看一下 ORP 中是如何枚举出这些根集的。

ORP 中使用的一般工作流程：停止所有的其他的 Java 线程，然后枚举出完整的根集；然后执行垃圾收集；然后恢复线程的运行。

具体的工作流如下：

1） 如果为某个新对象分配空间的调用发现内存不够时，就会触发 GC 发生；

2） GC 调用 orp_enumerate_root_set_all_threads()

3） orp_enumerate_root_set_all_threads()在处理上需要暂停所有的 Java 线程，我们在稍后会讲它的全部的详细处理，在得到根集后，方法返回；

4） 进行垃圾收集，回收死对象占用的空间；

5） GC 调用 orp_resume_threads_after

6） orp_resume_threads_after 恢复 Java 线程的运行。

上述的过程中，第1，2，4，5步由GC部分完成，3，6是由核心虚拟机来完成的。

在开始详细的讲述处理前，我们先来看一下一些基本的GC相关的信息：

ORP线程的gc_status，它记录了当前线程的GC状态，对应的类型：

```
enum gc_state {
    zero = 0,
    gc_moving_to_safepoint,
    gc_at_safepoint,
    gc_enumeration_done,
};
```

状态之间的转换如下：

原始状态	新状态
zero	gc_moving_to_safepoint
gc_moving_to_safepoint	gc_at_safepoint
gc_at_safepoint	gc_enumeration_done
gc_enumeration_done	zero

ORP线程的app_status，它记录了当前Java线程的状态，对应的类型：

```
enum java_state {
    zip = 0,
```

thread_is_sleeping,
thread_is_waiting,
thread_is_birthing,
thread_is_running,
thread_is_dying,
};

状态之间的转换如下：

zip->thread_is_birthing->thread_is_running->thread_is_dying

↙↗ ↘↖

thread_is_sleeping thread_is_waiting

ORP线程的gc_enabled_status，它记录了当前GC的开启关闭状况，对应的类型：

```
enum gc_enable_disable_state {
    disabled,
    enabled,
    enabled_will_block,
    enabled_hijack,
    bogus,
};
```

状态之间的转换如下：

enabled_will_block -(block)-> enabled --> disabled

而gc只能够发生在enabled_will_block/enabled之间。

下面我们来看一下orp_enumerate_root_set_all_threads详细的处理过程：

首先，处理过程会关闭GC，避免重入；

```
orp_disable_gc();
```

然后，进行GC工作的线程会为本线程取得最近Java帧以及其他的一些相关上下文；并设置GC控制线程为当前的线程，当前的全局安全点状态是正在枚举所有的线程。

```
get_control_of_threads_for_gc();
p_the_safepoint_control_thread = p_TLS_orpthread;
global_safepoint_status = enumerate_the_universe;
```

接着，会根据是否有调试接口选择适当的枚举例程：

```
if (jvmdi_debugger) {
    enum_when_a_debugger();
}
else {
    enum_when_no_debugger();
}
```

这是整个根集枚举的主要处理部分。我们首先来看一下enum_when_no_debugger的处理。

enum_when_no_debugger的处理会对所有活动的线程作如下的工作：

判断该线程是否是在Java中被暂停了(thread_is_java_suspended)，这通常发生在Java中调用了sleep/wait方法或者由于没有得到监视器的控制权而进入休眠，此时，一定在GC安全点，调用orp_enum对会根据引起线程在gc安全点的trap_state来选择合适的方法来进行枚举，如选择orp_thread_enumerate_from_native还是选择orp_thread_enumerate_from_java。枚举结束后，选择下一个线程继续循环。具体的处理可以参考orp\arch\ia32\base\root_set_enum_ia32.cpp中的orp_enum，trap_state的状态可以参考orp\common\include\orp_threads.h。

判断该Java线程是否是在休眠(app_status == thread_is_sleeping)，如果在休眠，取得该线程当前的gc_enabled_status，如果是enabled即允许状态，原子性的修改为enable_would_block，并调用orp_enum进行枚举。枚举结束，选择下一个线程继续循环。

判断该Java线程是否在等待监视器(app_status == thread_is_waiting)，如果是在等待监视器，开始和休眠一样的处理过程。

代码到这儿的话，那么在做完GC后，一定要恢复线程的运行。因此，设置当前线程的gc_resume_event_handle事件对象为触发态。

现在线程的GC状态一定是zero状态，标记GC状态为gc_moving_to_safepoint，即向GC安全点靠近；

接下来的处理比较复杂，表面上是一个永久循环，但里面实际上有两个循环出口，下面我们分别的来说明循环出口。其中一个循环出口是系统相关的，我们首先来说明这个出口：

这个部分的系统差异本身是由于暂停/恢复线程操作的语义不同而引起的。尽管在代码上表现出一些不同，但具体的处理还是类似的。首先需要调用暂停线程的操作，然后根据暂停线程当前的EIP地址来检查是否处于Java代码中，如果是的话，就把现在线程的上下文记录到线程的gc帧上下文中(gc_frame_context)，然后根据这部分代码对应的即时编译器检查当前执行点是否是GC安全的，如果是则把该线程的gc_status置为gc_at_safe_point,trap_state置为x_suspend_in_java_frame。如果该线程的状态为gc_at_safe_point，则调用orp_enum执行枚举过程，然后到达退出循环的出口。否则恢复线程的运行，并且当前线程休眠2ms，给刚刚恢复的线程以运行的机会。

另外一个出口的处理：该线程的gc_enabled_status必须是enbled、enabled_will_block或者enabled_hijack。这说明目前该线程在某个RNI的方法中，并且是在gc安全的代码中。如果该线程不在gc安全点，设置gc_status为gc_at_safe_point，并置trap_status为x_orp_enable_gc，然后设置该线程的gc_frame_context的最近Java帧为该线程最近Java帧，并记录该上下文中的p_eip为&gc_frame_context_eip_var。接着调用orp_enum执行枚举过程，到达退出循环的出口。

到这儿每个线程的处理结束。

由于在对每个线程的处理中，实际上是跳过了对当前正在枚举线程的处理，因此，接下来的处理中需要处理当前线程，即process_the_current_thread()。

最后要做的，调用orp_enumerate_root_set_global_refs()找到所有的全局引用。

至此，enum_when_no_debugger的处理结束。

另外一种可能的情况是enum_when_a_debugger来处理，与上面的处理不同。enum_when_a_debugger的处理是调用all_threads_to_safepoint来把所有的线程暂停在安全点后，逐个的orp_enum的。在函数all_threads_to_safepoint()内部是对每个线程调用move_a_thread_to_safepoint()方法来完成上述的功能的，在内部的处理和上述enum_when_no_debugger的对各个线程的循环类似，只是少了调用orp_enum。到每个线程都在GC安全点停下来后，就可以是对各个线程调用orp_enum。对各个线程枚举完毕后，调用orp_enumerate_root_set_global_refs()来找到所有的全局引用。

最后，开启当前线程的GC开关。

根集得到后，垃圾收集器就可以进行拷贝活动对象、修改引用值或者只是简单的回收死对象的空间，并根据需要进行重整。

垃圾收集完毕，将调用orp_resume_threads_after恢复各个Java线程的执行。

在本章开始我们讲过，根集包含两个部分：全局引用以及线程的局部引用。下面我们具体的来看一下针对它们的分类处理：

7.1.1 全局引用（Global reference）

全局引用是通过 orp_enumerate_root_set_global_refs() 来完成的，具体的代码可以参考：

orp\common\base\root_set_enum.cpp。主要的可以将全局引用分为以下几类：

所有类的引用类型的静态域；

需要执行终结方法（finalizable）的实例对象；

JNI 方法的全局句柄；

Interned 的字符串；

如果是对象锁 2.0 的，还需要对 mon_enter_array/mon_wait_array 的 p_obj 域进行标记，关于 mon_enter_array/mon_wait_array 的描述，请参考第 5 章关于对象锁第二版的描述；

全局的类装载器；

7.1.2 线程的局部引用（Thread Local Reference）

对于每个线程，活动的饮用可以分成以下几类：

由 JIT 生成的栈帧。核心虚拟机在识别出这样的栈帧后调用相应的即时编译器来在栈上找到引用；

对于每个 JNI 方法调用生成的局部 JNI 句柄列表；前面介绍过 J2N_Saved_Status 结构，这可以通过对保存的链遍历，找到每个 JNI 方法的 local_object_handles 来完成；由 orp_enumerate_object_handles 函数实现；

由活动的 RNI 方法压入的 GC 帧，这些帧是通过 orp_push_gc_frame(…)/orp_pop_gc_frame(…)来压入/删除的；由 orp_enumerate_references_in_gc_frames 函数实现枚举；

ORP 线程的四个域：p_java_lang_thread,p_current_object,p_stop_object 以及 p_exceptiion_object。由 orp_enumerate_root_set_single_thread_not_on_stack()函数实现。

orp_enum 的处理中 orp_thread_enumerate_from_native 和 orp_thread_enumerate_from_java 实际上都是针对线程的局部引用的枚举过程。这两个方法都会转去调用 orp_enumerate_root_set_from_stack 函数。orp_enumerate_root_set_from_stack 函数首先会取得线程的 p_eip 的内容，然后根据它得到相应的代码类型。如果是 Java 代码，调用即时编译器的接口函数 get_root_set_from_stack_frame 来完成枚举。关于这个函数的实现，请参考即时编译器部分的相关描述。

第三部分 Just-In-Time 即时编译器

第8章 ORP 虚拟机的编译模型

即时编译器，也就是 JIT（Just-In-Time）编译器提出了一种新的动态编译的模式。JIT 编译器在对象的方法被调用时将它从字节码翻译成机器语言。JIT 的执行效率直接关系到虚拟机的性能，所以在 ORP 中，JIT 编译器被作为一个重要的组成部分来实现。

8.1 JIT 的出现

1996 年 10 月 Sun 发布了第一个 Java JIT 编译器，并把它加入 Solaris 和 JavaWorkShop 操作系统中。JIT 编译器与传统的编译器不同，它并不遵循对一个完整程序编译－链接－执行的过程。当一个对象的某个方法第一次被调用时，虚拟机才激活 JIT 将它编译成机器代码，编译器被称为“即时”也原由于此。

为了更好地了解 JIT 编译器，我们需要简单回顾一下 Java 虚拟机的工作过程。当一个 Java 应用程序被编写后，首先运行如 Javac 这样的 Java 编译器把它编译成字节码程序，并被放置到后缀名为“.class”文件中。这个 class 文件能被任何带有 Java 虚拟机的机器所处理。在 JIT 编译器尚未出现之前，Java 虚拟机对字节码程序解释执行。所谓的解释执行，就是解释器按照特定的例程解释每一条字节码指令的语义。它的缺点是太过简单而且速度很慢，早期的 SUN JDK 解释器要比类似的 C++代码慢 5-30 倍。

JIT 编译器出现后，Java 虚拟机的处理有了很大的不同。对于 Java 源代码中的一个表达式：A*B，可能被编译成这样的字节码序列【fload_1，fload_3，fmul】。图 8.1 中展示了解释执行和 JIT 编译并执行的不同之处。

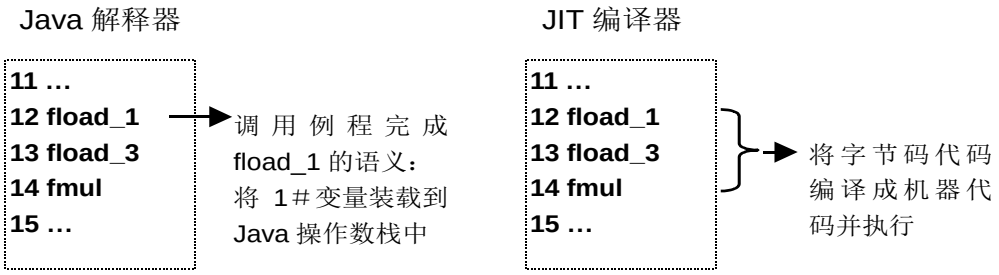


图 8.1 解释执行和 JIT 编译不同的处理方式

当一个 class 文件被读入后，将被交给 JIT 编译器。JIT 编译器把将要运行的字节码编译成机器代码。Java 是一种平台独立的语言，在各个主机平台上移动的是字节码中间表示，所以把所有的 class 文件静态的编译成机器代码是不经济的，因为当 class 文件被传送到另一台主机上时，还需要将它重新编译成机器代码。JIT 编译器采用即时的模式，直到一个方法被调用时才将字节码翻译成机器代码，当一个方法被两次以上的调用时，机器代码的执行效率足以补偿花在编译上的时间。而且 JIT 编译器以一个对象方法为单位进行编译，能够进行一些全局的优化，生成的代码质量比较高， JIT 编译器生成代码的执行速度可以达到解释执行的 10 倍。

JIT 的出现使得 Java 程序的执行效率得到了很大提高，但是执行过程不得不等待编译的结束，编译更加复杂，编译时间加长。为了缓解执行效率和编译开销的矛盾，很多虚拟机都会使用快速解释器和优化编译器的组合或者是简单编译器和复杂编译器的组合。ORP 使用的就是后一种组合，并且采用动态重编译机制来突出 JIT 编译器的种种优势。

8.2 ORP 的编译框架

在传统的静态编译中，编译时间可以忽略不计，因为经过一次编译之后，生成机器代码的可执行文件，可以被多次执行。而对于 JIT 编译器来说却并非如此， JIT 在运行时编译字节码，编译时间是运行时间的一部分。因此，JIT 编译器对编译时间的要求非常敏感。花许多时间在对所有的代码进行细致地优化是很不明智的，因为并非所有的代码都会被频繁的调用。而优化的目标代码的质量对于那些被频繁执行的代码却非常重要，因为程序的运行时间很大程度决定于这部分代码的执行时间。ORP 编译器部分的设计关键就是在代码质量和编译时间中找到一个最佳的平衡点。

ORP 的编译器实现了一个动态的重编译机制。这个重编译机制的关键在于对不同的代码自适应、有选择地使用不同的编译方案：对于那些执行频率低的“冷”代码，采用快速而较粗糙的编译器；而对执行频率高的“热”代码，则花更多的时间对其做细致地编译。这个重编译机制的自适应性表现在它能够收集运行时的信息，判断代码是否是热点，而及时地调整编译策略。

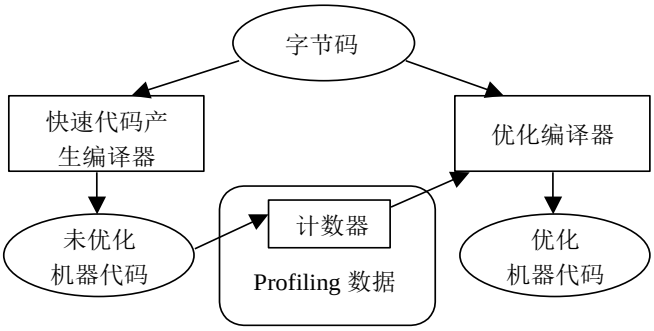


图 8.2 ORP 动态重编译机制结构

图 8.2 展示了 ORP 的编译结构。这个编译结构有三个重要的组成部分：快速代码产生编译器、优化编译器、profiling 信息。

快速代码生成编译器，在 ORP 中通常也被称为 O1 编译器。所有的对象方法在第一次被调用时都用这个编译器编译成机器代码。ORP 快速编译器不但能达到比较理想的编译速度，而且能够对机器代码进行轻量级地优化，产生的机器代码的执行速度要比解释执行快得多。O1 编译器的目标是快速地产生机器代码，并且维持一个合理的代码质量。它对字节码进行两次扫描：第一次收集例如基本程序块和 Java 操作数栈栈深这样的信息；第二次用 lazy 代码选择的方法生成机器代码，并进行轻量级地优化。编译的时间复杂度是线性的。我们将在第 9 章中详细地讨论快速代码生成编译器。

O1 编译器在机器代码中插入一些统计代码，用来收集 profiling 信息，这些信息记录了方法或是一段循环代码被调用的次数。Profiling 信息包含进行重编译的触发条件，有了 profiling 信息，就能够判断某段代码是否是“热”代码，当代码的调用次数达到某个阈值时，它将被优化编译器重新编译。在 O3 编译器对方法重新编译时还将利用 O1 收集的这些 profiling 信息。

优化编译器，也叫 O3 编译器，对代码进行更为细致地优化，产生质量较好的目标代码。之后，当方法被再次调用时，就会执行优化版本的机器代码。O3 编译器为了产生高质量的目标代码，需要花很多时间在优化上。它采用传统的编译方法，为字节码建立一个中间表示，基于中间表示进行全局优化。我们将在第 10 章中详细讨论 ORP 优化编译器。

ORP 允许用户指定编译模式。用户可以通过命令行参数来要求虚拟机只用 O1 模式或 O3 模式来编译字节码，或者采用上面所说的重编译机制来编译字节码。用户甚至可以指定 O1 和 O3 以外的，自己编写的 JIT 来作为虚拟机的编译器。

8.3 基本的编译过程

在正式地介绍 ORP 的编译系统之前，我们有必要先看一下作为一个编译器应该完成那些工作。

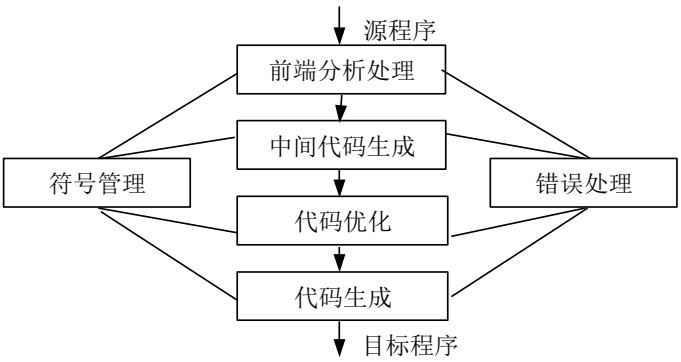


图 8.3 编译的流程

图 8.2 是一个编译过程的一个典型流程。每一个编译器大多都有这几个阶段，在具体的编译器实现中，可能把若干个阶段结合在一起，而各个阶段之间也可能会有一些中间表示。

编译器的前端处理通常包括词法分析、语法分析、语义分析。词法分析阶段读源程序的字符，形成各个词法成分的记号流，标识符、关键字等字符序列被识别出来。语法分析阶段把记号流按语言的语法结构层次分组，形成语法短语的分析树。语义分析阶段检查程序的语意正确性，以保证程序各部分能有意义地结合在一起，并且为以后的代码生成阶段收集类型信息。

JIT 编译器的源语言是 JVM 字节码，它事实上已经是从 Java 编译过来的一种中间语言。字节码就是一系列 Java 虚拟机指令，每一条指令由一个字节的操作码后面跟上零个或多个操作数组成。JIT 编译器没有前端的分析处理。

8.3.1 中间代码生成和代码优化

在语法分析和语义分析后，编译器通常产生源程序的显式中间表示。在源语言和机器语言之间的翻译过程很可能有若干个中间语言。ORP 的优化编译器为了优化的需要，在产生机器代码之前把字节码程序翻译为一个中间表示，然后再从中间表示中产生机器代码。

代码优化阶段视图改进代码，产生执行较快的机器代码。理想的优化目标是使编译器产生的目标代码和手写的一样好。编译算法能够改进一部分代码，使它运行得更快一些，或占用更少得空间。这些改进通常只适用于符合某些特殊模式的情况。接下来我们介绍一些在 JIT 编译器中有用的代码优化情况。

1. 消除公共子表达式

如果表达式 E 先前一计算，并且从先前的计算至现在，E 中变量的值没有改变，那么 E 的这次出现就称为公共子表达式。如果我们能够利用先前的计算结构，就可以避免表达式的重复计算。这样减少重复计算的行为就叫着消除公共子表达式。传统的消除公共子表达式的算法，通常是基于数据流分析和值计数，这种算法通常需要较大的时间和空间复杂度。ORP 的 JIT 编译器设计了适应于它的算法来进行消除公共子表达式的优化。

2. 循环优化

循环，尤其不包含其他循环的内循环，是一个重要的可优化的地方，因为程序消耗大部分的时间在这部分代码块上。如果减少了内循环的指令数，程序的运行时间就可以大大的减少。因此减少循环中指令总数的是进行循环优化的基本思想。减少循环中代码总数的一个重要办法式代码外提，这种变换把循环不变量计算，即所产生的结果独立于循环执行次数的表达式，放到循环的前面。除了代码外提，还有诸如删除归纳变量、消弱操作强度等优化手段。

3. 减少数组边界检查

Java 语言规范中要求：所有的数组访问都要进行运行时检查，如果数组元素的下标超过了数组的边界将会引起一个异常的抛出（ArrayIndexOutOfBoundsException）。因此，当字节码程序中出现数组元素的访问时，编译器需要插入一段检查和跳转的代码来保证符合 Java 语言规范。如果能够证明数组元素的下标在一个安全的范围内，那么就可以省去这部分边界检查代码的执行时间。

除了这些常见的情况外，为了提高代码的质量，JIT 编译器还进行另外一些静态和动态的代码优化。在后续的两章中，我们将详细地介绍 O1 和 O3 编译器的代码优化算法。

8.3.2 代码生成

编译的最后一个阶段就是目标代码的生成，生成可重定位的机器代码或者汇编码。这个阶段为源程序所用的每个变量选择存储单元，并把中间代码翻译成等价的机器指令序列。

这个阶段中的一个关键问题就是寄存器分配。运算对象在寄存器中的指令通常比运算对象在内存中的指令短些，执行也快些。因此，充分利用寄存器对生成好的代码尤其重要。IA32 提供的寄存器组非常丰富，以下是 IA32 机器指令中常见的寄存器：

- 32 位通用寄存器：EAX，EBX，ECX，EDX，ESI，EDI，ESP，EBP。
- 16 位和 8 位通用寄存器：取以上寄存器的不同字段构成。
- 段寄存器：CS，DS，SS，ES，FS，GS。
- 标志寄存器和控制寄存器：EFLAGS，CR0 等。

如果源程序中用户定义的变量和编译时各阶段产生的暂时变量能够最大限度地映射到寄存器中去，将极大地提高程序运行的速度和减少生成机器代码的长度，所以合理地运用 IA32 提供的寄存器也是 JIT 编译器能够产生高质量代码的一个关键。在下面几章中，我们会详细地介绍各个 JIT 编译器的寄存器分配算法。

ORP 建立在 Intel 的 IA32 处理器体系之上，它的 JIT 编译器在为方法选择机器代码时充分利用了 IA32 的处理器指令的特点。Intel 的 IA32 体系处在计算机革新的最前沿，是目前最流行的处理器体系。它的历史可以追溯到最早的 Intel8086 处理器，而最新的版本是 Pentium 4 和 Xeon 处理器。Intel 386 处理器是 IA32 家族中最早的 32 位处理器，具有 32 位寄存器，32 位地址总线，支持 4GB 的物理地址空间。

IA32 处理器提供了丰富的寻址方式，它的指令操作数包括：立即数、寄存器数、直接寻址、寄存器间接寻址、寄存器相对寻址、机制编制寻址，六种寻址模式。表 8.1 中列出了这六种寻址模式。ORP 的 JIT 编译器将利用这些寻址方式，产生精练的机器代码。

操作数		地址计算方式
立即数（Immediate Operand）		操作数直接放在指令中，紧跟在操作码之后。可以是 8 位、16 位、或 32 位。
寄存器数（Register Operand）		操作数在寄存器中，指令指定寄存器号。可以是 IA32 提供的寄存器组中间的一个。
内存操作（Memory Operand）	直接寻址（Direct addressing）	物理地址＝段地址+Displacement Displacement：一个 8、16、32 位值，直接给出操作数偏移
	寄存器间接寻址（Register indirect）	物理地址＝段地址+Base Base：一个通用寄存器作为基址寄存器，存放操作数偏移
	寄存器相对寻址（Register relative）	物理地址＝段地址+Base+Displacement 由 Displacement 和 Base 共同给出操作数偏移
	基址变址寻址（Based indexed）	物理地址＝段地址+Base+Index×Scale+Displacement Index：一个通用寄存器作为变址寄存器，通常用作数组索引 Scale：变址步长，可以看作一个数组元素所占的字长

表 8.1 IA32 的寻址模式

在 ORP1.0.9 版本的实现中，用类 Opnd 来为 x86 指令的操作数建模，它的作用是存储操作数信息，在产生代码时在存放代码的内存区内生成 x86 操作数作为机器指令的一部分。按照其寻址方式，类 Opnd 衍生了几个子类：

- Imm_Opnd：立即数操作数，该类对象存储了一个立即数的值。
- R_Opnd：寄存器操作数，该类对象存放一个寄存器操作数的寄存器名称。
- M_Opnd：这是所有内存操作数类的基类，本身采用直接寻址方式。它包含一个数据成员存放各种内存寻址方式都需要的偏移量。它衍生了以下这些子类。
- M_Base_Opnd：采用相对寻址方式的内存操作数，它包含了操作数的基址寄存器 base_reg。
- M_Var_Opnd、M_Spill_Opnd：直接继承了 M_Base_Opnd 类，也采用相对寻址方式，但是它们的基址寄存器 base_reg 是特定的寄存器（ESP 或 EBP），它们用来表示两类特殊的相对寻址操作数，我们在后续的章节中会介绍到。
- M_Index_Opnd：采用变址寻址方式的内存操作数，它包含了操作数的基址寄存器 base_reg 和变址寄存器 index_reg，以及变址步长 shift_amount。

在介绍后面几章 JIT 编译器代码选择时，我们会看到这些操作数如何表示 Java 操作数和一些中间结果，并应用在机器指令中。

8.4 JIT 和 ORP 的接口

ORP 为核心 VM 和 JIT 定义了一系列的接口，JIT 和 VM 之间通过这些接口交互，而不需要知道实现细节。因此 ORP 中的 JIT 部分可以被独立更新，用户可以编写自己的 JIT 代码，只需要提供与定义一致的接口。

所有的接口可以分成 4 类，分别定义在 orp\interface 目录下的 4 个文件中。

- *orp_types.h：在 VM 中用到的基本类型*
- *jit_intf.h：由 VM 提供的函数，JIT 在编译时需要调用*
  JIT 在编译一个方法时可能需要向类装载器查询有关某个类、或者某个对象的域和方法的信息。例如，JIT 往往需要某个类是否初始化了，某个对象的域是否已经被解析。如果它需要某个未初始化类的静态域成员，或者需要对象的一个未被解析域，那么 JIT 就要调用 ORP 的接口函数对类初始化，或是解析某个域。这个文件中还定义了一些接口函数是用来传递 ORP 提供给 JIT 的一些额外的信息，例如一个方法的异常信息。
- *jit_runtime_support.h：由 VM 提供的运行时支持函数*
  这部分包含了 VM 提供的两方面的运行时支持。其一是给 JIT 调用的，例如 JIT 在运行时为垃圾收集计算根集时，需要调用 VM 接口函数把根集返回给 GC。其二是给 JIT 产生的代码在运行时调用的。例如在编译字节码 new 创建一个新的类对象或是翻译 monitorenter（同步方法进入对象的监视器）时，JIT 产生一组机器指令调用 VM 提供的函数例程，在运行时将执行 VM 提供的例程代码来完成工作。
- *jit_export.h：由 JIT 提供的函数，VM 在编译方法和运行时需要调用*
  VM 需要一个方法的机器代码时要调用 JIT 提供的接口函数编译这个方法。另外 VM 在运行时有一些工作，例如垃圾收集、异常处理，必须依靠 JIT 才能够完成。JIT 提供给 VM 接口函数使它能够得到某个方法栈帧上的根集，使它能够 unwind 活动记录栈。

第9章 快速代码产生编译器

对于一个 JIT 编译器中，编译的速度有时要比编译的质量重要得多。在 ORP 的平台上提供的 O1 模式编译器（快速代码产生编译器），采用 Lazy 代码选择的算法，并且在编译中只进行简单而快捷的优化手段，以换取较快的编译速度。在这一章中，我们将对这种模式的 JIT 编译进行详尽地介绍。9.1 节是对快速代码产生地编译的概述，描述快速代码产生的基本思想和主要流程。在 9.2 节中将对编译过程中的主要阶段分别做细致地介绍。对于一个 Java 语言的 JIT 编译器与虚拟机的交互也是很重要的，它将提供对垃圾收集、异常处理的运行时支持，因此我们在 9.3 节中对这部分内容进行阐述。

9.1 快速代码产生的编译

9.1.1 基本流程描述

在 ORP 的 O1 编译模式和重编译机制下，Java 类的方法总是由快速代码产生编译器生成，它意图在较短的时间内高效地产生 Intel IA32 上的机器语言。其中的关键在于它的 lazy 代码选择算法可以在对字节码进行一遍扫描之后，直接产生机器代码。而且，快速代码产生编译器在为寄存器分配构建控制流图时，并不产生显式的中间表示。不同与其它一些 JIT 编译器把字节码转换成显式的中间表示，它用字节码本身来表示表达式和维护一些额外的数据结构，例如程序的基本代码块、控制流的跳转边。快速代码生成只进行一些轻量级地优化，所谓轻量级地优化在于优化代码所用的时间和附加的数据空间都是非常少的。在图 9.1 中，给出了快速代码产生的基本流程。

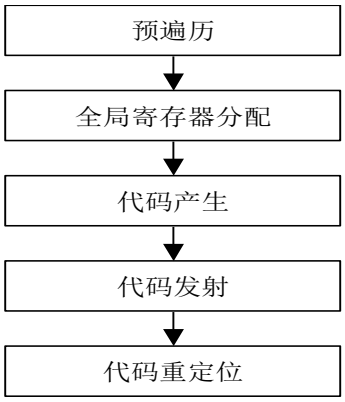


图 9.1 快速代码产生编译的基本流程

快速代码生成编译分成 5 个阶段：

预遍历：为了保证全局寄存器分配和代码产生阶段能够在对方法的字节码进行一次遍历之后完成，需要为后面的工作收集一些所需的数据。在这个阶段将会为程序的基本代码块段定出边界、记录控制流回边数目，在字节码上记录控制流信息。而且，这个阶段还要为全局寄存器分配和垃圾收集记录有用的信息，例如，在垃圾收集可能发生的程序点操作数栈上引用信息；每个方法的局部变量的静态引用数，这个信息是为局部变量进行寄存器分配的参考信息。

全局寄存器分配：在这个阶段，编译器将把物理寄存器分配给局部变量。这里的物理寄存器是指 4 个被调用者保存的寄存器：**ebx**，**ebp**，**esi**，**edi**。

代码产生：这个阶段是编译器使用 Lazy 代码选择算法生成机器代码的重要步骤，在这个阶段还将进行轻量级的代码优化：删除公共子表达式、减少数组边界检查等。这个阶段产生的机器代码放在一个临时空间中，机器代码中的偏移量都是相对这个临时空间的首地址而言的。

代码发射：在这个阶段，编译器才将产生的代码拷贝到它们在内存中的最终位置，当方法被调用时，程序将会跳转到这个内存位置的第一条指令。

代码重定位：代码在从临时存储空间拷贝到最终的内存位置后，其中的数据和代码的偏移量都应该相对这个最终位置而言，就像跳转语句的目标地址偏移。这个过程也叫代码的补丁过程（patching）。

快速代码产生的五个阶段中，只有寄存器分配阶段的时间复杂度和空间复杂度不是线性的，因此能够称得上是快速的 JIT 编译器。

```
// 提供给 ORP 的编译接口
JITExport JIT_Result JIT_compile_method（…）{ return l1a_compile_method（…）；}
JIT_Result l1a_compile_method（…）{…}
```

```
JIT_Result O1a_gen_code（…）{
  …；
  CG_Prepass prepass（..）；// CG_Prepass 对象初始化，对方法预遍历
  Code_Emitter code_emitter（mem_manager，byteCodeSize）；// 临时代码缓冲区管理对象
  Register_Allocator *regalloc = new（mem_manager）Register_Allocator；
  regalloc->register_allocate（&prepass，methodHandle，… ）// 全局寄存器分配
  …；
  Jit_Method_Info *method_info =（Jit_Method_Info *）info_manager.alloc（mi_size_estimate）；
  init_method_info（method_info，regalloc，…）// 初始化方法编译信息结构
  …；
  ESP_Frame esp_frame（…）；
  EBP_Frame ebp_frame（…）；
  Frame *frame；// 生成与方法栈帧结构有关的 Frame 类对象
  if（method_info->is_esp_based）frame = &esp_frame；else frame = &ebp_frame；
  …；
  Stack  stack（…，code_emitter，*frame，prepass，…）；// mimic 栈类对象初始化
  JIT_Result res；  // 运用 Lazy 代码选择策略为方法产生机器代码
  return res = select_code（mem_manager，code_emitter，prepass，*frame，stack，…，
          methodHandle，byteCodeAddr，byteCodeSize，method_info, regalloc，…）；
}
```

图 9.2 O1 编译器主要函数流程

9.1.2 快速代码产生编译器与解释执行

在动态重编译机制中，也可以用解释执行来代替快速代码产生编译，也就是在一个方法被确定为一个热点之前，只用解释执行来处理。为什么在 ORP 的重编译机制中不用解释执行而用快速代码产生呢？ORP 选择快速代码产生有两个理由：

第一，解释执行虽然在编译时间上占有一定的优势，但是它的执行性能效率极低。在一个不是很大的程序中，解释执行字节码和执行机器代码的性能就可能相差一个数量级。这个性能差距使得 Java 虚拟机判断一个方法是否是热点更加困难，因为如此大的性能差距会使虚拟机难以确定进行重编译的尺度，会把许多“冷”方法也重新优化编译，而把字节码优化编译到机器代码需要的花费非常大。如果我们把

触发重编译的时机看作一个窗口，解释执行到优化编译的窗口就非常小，很难把握这个时机，错过了这个窗口，不是花费过多的时间在编译一些“冷”代码上，就是要为解释执行的性能付出很大的时间代价。而快速代码产生编译器的性能与优化编译器的性能相差只有 30%左右，这就使得触发重编译的窗口要比解释执行方法大得多。另外，由于快速代码生成编译器的速度也非常快，花费这些时间来换取代码较好的执行性能是值得的。

第二，当一个 Java 应用程序以测试（debugging）的模式运行时，也需要 JIT 编译器来提供程序状态的监视和对执行的控制，比方说 JIT 编译器要能够在某个程序点给出变量当时的值，或者要支持断点设置。这就要求编译器不仅能够提供源程序变量在内存中的位置，还要能够给出一条机器代码对应的字节码的偏移。在优化编译器中提供这些支持是不可能的，因为由于全局优化和代码规划使得目标代码和字节码之间的对应变得困难了。缓慢的执行速度使得在解释执行下进行测试也不可能被用户所接受。还有一种解决的办法就是：一个方法被优化编译，当用户要求测试时，退回到解释执行的方式下。这种方法又叫反优化（deoptimization），它的缺点在于：由于要支持机器代码到解释执行的反编译，使得编译器不得不记录庞大的额外数据，这会使 Java 虚拟机复杂得多。而快速代码生成编译没有以上这些缺点，而且它的优化能够保持运行的值和源程序具有一致性，所以 ORP 选择了在快速代码生成编译上实现测试模式。

9.2 编译细节

9.2.1 预遍历过程

9.2.1.1 信息与存储结构

预遍历为快速代码产生的后续部分收集数据，在这个阶段主要收集和计算这三类数据：

- 在每个基本代码块的入口处，Java 操作数栈的栈深。基本代码块是在程序的控制流图上的一个基本计算结点。代码选择阶段在每个基本代码块的入口处需要在模拟的 Java 操作栈上确定当时每一个操作数的位置，这需要知道操作数栈当时的深度。
- 方法中每一个局部变量的静态引用数。局部变量的静态引用数是该变量在方法代码中被显式地读或写的次数，它反映了一个变量被使用的频率。这个参数对全局寄存器分配算法有重要的作用。
- 在垃圾收集可能发生的程序点，Java 操作数栈上的引用信息。为了支持垃圾收集时能够计算出活对象的根集（在操作数栈上和寄存器中的仍然在使用的对象），预遍历阶段记录了在程序点上操作数栈中每一个栈单元是否是引用的信息。

ORP 的编译器实现通过产生一个 CG_Prepass 类的实例来完成预遍历过程。在 CG_Prepass 类中有相应的数据成员用来保存这些信息：

```
class CG_Prepass{
  public:
      CG_Prepass(unsigned maxLocals,const unsigned char *first_bc,unsigned code_length,
          Class_Handle class_handle, Method_Handle method_handle,
          Compile_Handle comp, Mem_Manager&, unsigned maxStk,
          class Register_Allocator *regalloc);
      Bytecode_Info *bytecode_info(unsigned i) {return &_bytecode_info[i];}

          // 公共方法，用来访问记录每条字节码指令信息的列表

      unsigned ro_data_size () {return _ro_data_size;}          // 只读常量数据数目
```

```
        unsigned rw_data_size () {return _rw_data_size;}  // 可读可写数据数目
        … …
        unsigned num_get_put_static;            // 字节码中 get/put statics 指令的数目
        unsigned num_call_sites;            // 中断插入调用点的数目
        unsigned num_returns;                   // 字节码中{a,i,f,l,d}return 指令的数目
        unsigned num_blocks;                    // 基本代码块的数目
        unsigned num_edges;                     // 控制流图中边的数目
        unsigned num_entries_back_edge;         // 控制流图中回边的数目
        unsigned *ref_count;                    // 整型数组，记录局部变量静态引用数
        Bit_Vector_List *gc_site_vectors;       // 记录垃圾收集点操作数栈上指针情况的位矢量数组
        Recomp_Entry *recomp_entries;           // 需要插入计算重编译代码的程序点列表
        Finally_Bytecode_List *finally_bc_list; // 字节码中 finally 处理块的列表
    private:
    … …
}
```

上面的代码列出了 CG_Prepass 类中与需要收集的信息相关的方法和属性。

类的构造函数 CG_Prepass 事实上就是进行预遍历的主要函数，快速代码产生 JIT 编译器在产生一个 CG_Prepass 对象时，把进行预遍历需要的信息作为参数传入，其中包括要编译的方法的一些静态信息：局部变量的数目、第一条字节码的偏移量、所有字节码的数目、常数池中方法所属的类的句柄、方法的句柄、最大栈深，以及一些相关模块的接口：编译器对象指针、内存管理对象的指针、寄存器分配对象指针。预遍历过程将遍历每一条字节码指令，用 Bytecode_Info 结构的列表来存储每一条字节码的属性信息，并提供了对此列表的访问函数。而控制流图的信息也就包含在这些字节码信息当中。

```
struct Bytecode_Info {
    union {
        struct {
            unsigned depth:16;              // 该条指令处的栈深
            unsigned is_block_entry:1;      // 是否是基本代码块的第一条指令
            unsigned is_label:1;        // 是否是一个标签，即跳转指令的目标
            unsigned is_try_start:1;        // 是否是异常捕捉 try 代码块的第一条指令
```

```
            unsigned is_try_end:1;          // 是否是 try 代码块的最后一条指令
            unsigned is_exception_handler_entry:1; // 是否是异常处理代码块的入口指令
            unsigned is_back_edge_entry:1;          // 是否是回边的入口指令
        } attr;
    unsigned init;                  // 结构初始化时置成 0，为 1 时属性域有效
    };
};
```

在做完对方法字节码的预遍历之后，将得到要编译的方法的一些静态数据。其中包括控制流图的信息：基本代码块的数目、边的数目、回边（向后跳转的边，表现为跳转指令的目标偏移量为负值）的数目；还有局部变量的静态引用情况和一些特殊指令的出现次数，例如类的静态数据读写、调用返回指令。ORP 选择函数调用点作为 GC 安全点，具体的原因我们将在 9.3.2 节中介绍，在预遍历过程中需要对函数调用点进行粗略统计，该信息记录在 num_call_sites 中。而 gc_site_vectors 是一个位矢量链表，它记录了在这些函数调用程序点操作数的引用情况。链表中的每一个结点含有与当时操作数栈深度相同的 bit 位，bit 位为 1 时表示栈上相应的操作数是一个引用。这些信息的用法我们将会在后续章节中介绍。

9.2.1.2 信息收集过程

CG_Prepass 的构造函数通过对要编译方法的字节码一次遍历收集上面提到的所有信息。图 9.3 给出遍历算法的流程图。

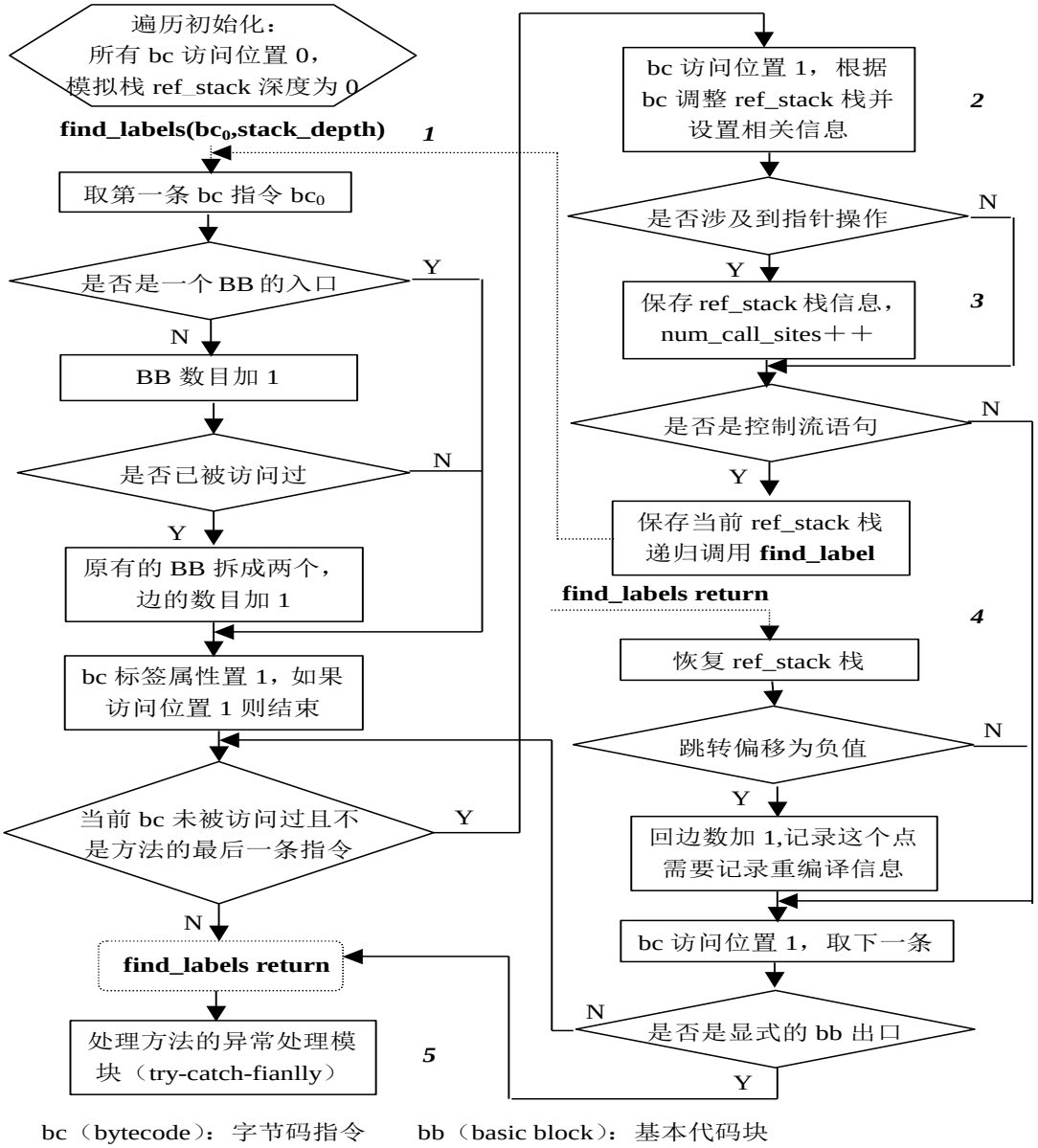


图 9.3 预遍历流程图

1. find_labels 函数

预遍历过程的重点是 CG_Prepass 类的成员函数 find_labels，它的主要功能是在一个基本代码块中遍历字节码。它的参数 bc 是该基本代码块入口指令的指针，stack_depth 是进入基本代码块时 Java 操作数的栈深。CG_Prepass 构造函数调用 find_labels 函数时，bc 为方法的第一条字节码指令，栈深为 0，为所有的字节码指令设置一个访问位，初始时所有访问位置 0。

函数将把第一条指令的 Bytecode_Info 属性置为一个基本块的入口，且基本块数目加 1。如果这条指令已经访问过，说明这种情况必然是某个跳转语句跳入了一个已访问过基本块，分割出一个新的基本块，边的数目要加 1。在字节码指令没有被访问过的情况下，逐条分析每条指令，确定代码的控制流图和其他一些静态信息。如果遇到 goto 指令、各种 return 指令、异常抛出语句 athrow，这些明显是一个基本块的结束指令时，find_labels 结束。

2. 分析字节码指令

find_labels 函数逐条分析每条指令是为了收集上一节提到的各种信息，其中一个重要部分就是 Java 操作栈上的信息。CG_Prepass 对象用一个长度为最大栈深的位矢量 ref_stack 来模拟 Java 操作数栈，每

一位表示相应操作数的类型：引用或非引用。这里的引用包括对象引用、数组引用。函数根据每条指令的操作语义，对 ref_stack 进行压栈、弹栈操作，并根据指令的操作数类型，设置 ref_stack 的相应 bit 位。如果是 iload、istore 这样的对局部变量进行读写的指令，则要增加相应局部变量的静态引用数。对于上一节提到的特殊指令，函数也将逐条计数。

在预遍历字节码时，还将进行一定的静态检查，例如像 iaload 这条指令，将从一个数组中取出一个整型值，这时 Java 操作数栈的状态应为：

…，arrayref，index

栈顶是一个非引用操作数，而次栈顶是一个数组引用，函数将检查 ref_stack 模拟栈是否处在这样一个状态，如果不是，则预遍历失败，并且导致方法编译失败。

3. 收集操作数栈引用信息

在预编译时并不知道程序点是否是一个函数调用点，为保险起见当一条字节码指令对 ref_stack 模拟栈的引用类型进行调整时，它都把当时的 ref_stack 模拟栈复制一份保存起来。例如：iaload、faload 这些从数组装载操作数的指令，根据这些指令的操作语义，操作数栈有这样的变化：

…，arrayref，index =>

…，value

又例如调用实例方法、接口方法的指令：invokespecial、invokeinterface，它们的栈的变化也包含了对象引用的弹栈：

…，objectref，[arg1，[arg2…]] =>

…

在这些情况下，要把该程序点的 ref_stack 栈复制下来，连同当前字节码指令的指针和栈深保存到 CG_Prepass 对象的 gc_site_vectors 链表中去。而且这些程序点都当作是函数调用点计算，所以 num_call_sites 计数加 1。在预遍历过程中计算得到的函数调用点要比实际函数调用点多得多。

4. 控制流语句处理

当字节码指令是 ifeq、goto、jsr 这些控制流跳转指令时，标志着目标字节码是一个基本块的入口地址，因此基本块数目和控制流边的计数都要加 1。而且，应该递归调用 find_labels 函数来遍历新的基本块，目标字节码作为一个基本块入口地址以及当前栈深作为参数传入，在此之前必须把当前的 ref_stack 栈保存。对于 ifeq 这样的选择跳转语句，从它们的下一条指令也是一个新的基本块，同样基本块数目和控制流边的计数也要加 1。在 find_labels 函数返回之后，要把递归调用之前的 ref_stack 栈恢复，取出下一条指令作为新基本块的入口地址，继续向前遍历。

如果跳转指令的目标是一个负的偏移，说明这是一条向前跳转的回边, 这样的回边则可能是循环的标志。根据经验，重编译机制的热点通常是一个函数的调用或是一个循环的反复执行，因此 ORP 的 JIT 编译器在这些点要加入代码对这条回边的 profiling 数据（例如回边经过频率）进行记录。而且，CG_Prepass 对象的 recomp_entries 链表将增加一个结点来记录这条回边的信息。

jsr 这条指令的处理又更加特殊一些，因为这条语句将跳转到一个 subroutine 子程序中去，它标志着一个异常处理 finally 块的开始。在垃圾收集时，需要对 finally 块的第一条 astore 指令特殊处理（参见 9.3.2），所以 jsr 指令的目标指令将被记录到 CG_Prepass 对象的 finally_bc_list 链表中去。

5. 异常处理模块处理

当 CG_Prepass 构造函数从 find_labels 函数退出后，还有一类基本块被忽略，这就是异常处理的 try-catch-finally 模块。try-catch-finally 模块在字节码中并没有显式地被标出来，但是方法相应的 Method 对象中记录了要编译方法捕捉、处理的异常。所以 CG_Prepass 构造函数的最后工作就是通过方法句柄（对象指针）获取异常处理信息，把所有的 try-catch-finally 模块作为基本块标识出来。

9.2.2 寄存器分配

寄存器分配是代码生成阶段中的一个重要工作，它决定了在程序的不同执行点上，哪些值应该放在寄存器中。那些运算对象在寄存器中的指令通常都比运算对象在内存中的指令的执行时间要短，而一个处理器中寄存器资源是非常有限的,因此合理地利用寄存器是编译器产生高质量代码的机器代码的关键。

在一个处理器的寄存器组中，控制寄存器和标志寄存器都有专门的用途，用在特定的指令中，通常能够用来作为变量和中间结果暂存的是通用寄存器。IA32 提供了 7 个 32 位通用寄存器：EAX、EBX、ECX、EDX、EBP、ESI、EDI，另外加上栈顶指针寄存器 ESP。按照函数调用的惯例，这些寄存器又分为两类：调用者保存寄存器（caller-saved register）和被调用者保存寄存器（callee-saved register）。EAX、ECX、EDX 属于调用者保存寄存器，而 EBX、EBP、ESI、EDI 属于被调用者保存寄存器，而 ESP 通常作为栈指针，不属于这两个类中的任何一个。在 ORP 实现中定义了一个枚举类型 X86_Reg_No，来表示这 8 个寄存器：

```
enum X86_Reg_No
    {eax_reg=0, ecx_reg=1, edx_reg=2, ebx_reg=3, esp_reg=4, ebp_reg=5, esi_reg=6, edi_reg=7, n_reg };
```

ORP 的 JIT 编译器的寄存器分配分为两部分工作：4 个被调用者保存寄存器用作全局寄存器分配；3 个调用者保存寄存器用作局部寄存器分配。全局寄存器分配把寄存器分配给单个方法中的局部变量，局部寄存器把寄存器用作表达式计值中中间结果的暂存器。

9.2.2.1 全局寄存器分配

全局寄存器分配的任务是在单个函数方法中，为局部变量（这里的局部变量包括是函数的参数和函数内部定义的变量）分配寄存器。这是一个活跃的研究领域，目前已有了一些有效的算法。研究证明，选择最优的全局寄存器分配方案是一个 NP 完全问题，现有的全局寄存器分配算法都是寻找一种次优的方案，其中占主导地位的是图着色算法。图着色算法对时间和空间的要求都很高，这对 JIT 编译器特别是 O1 编译器的快速、实时要求是不适合的。所以 ORP 的快速代码产生编译器就需要解决一个问题：如何平衡高消耗的分配算法和期望的代码性能之间的冲突。

快速代码产生编译器提供了两种全局寄存器分配的算法：简单分配算法和基于优先级算法。O1 编译器中用类 Register_Allocator 的对象来完成全局寄存器分配的任务，类 Register_Allocator 具有三个子类：Register_Allocator_None、Register_Allocator_Simple、Register_Allocator_Priority。后两个子类分别实现了简单分配算法和基于优先级算法。而第一个子类 Register_Allocator_None 不做全局寄存器分配，不为函数的局部变量分配被调用者保存寄存器，我们将忽略这个类。

作为基类的 Regster_Allocator 定义了全局寄存器分配对象的对外接口。一个寄存器分配对象将完成以下三部分工作：

- 寄存器分配：应用特定算法，确定在每个程序点存放在被调用者保存寄存器中的一组局部变量。
- 将分配情况保存在一定的数据结构中。
- 提供编译时寄存器分配情况的查询：在代码产生的阶段，寄存器分配对象要能够回答关于分配情况的查询。例如：在某个程序点特定的寄存器放的是哪一个局部变量；在某个程序点特定局部变量放在哪个寄存器当中；在函数的入口处是否要将某个参数装载到寄存器当中。

下面将介绍 O1 编译器中两种不同的全局寄存器分配算法。

1. 简单分配算法

简单分配算法的特点是极其简单而且比较有效。Register_Allocator_Simple 对象中用 4 个短整型私有成员记载分配结果，即寄存器与局部变量的映射：esi_local、edi_local、ebp_local、ebx_local。图 9.4 的伪代码说明了这个分配方案的算法。

```
// Register_Allocator_Simple 类的全局寄存器分配过程，返回值为分配出去的寄存器数
if（方法内含有异常处理）return 3 ；
设置一个寄存器可得位矢量 avail_regs =11101000B ；      // 4 个 1 分别对应 4 个被调用者保存寄存器
n_assigned = 0；
取预遍历阶段获得的每个局部变量的静态引用计数链表 ref_count；
for （每个被调用者保存寄存器 callee_saved_reg）
{
    if （avail_regs 相应寄存器位上标志位为 0）continue;
    遍历 ref_count 链表，取大于 1 的最大静态引用数的局部变量索引 var_idx；
    if（var_idx = = - 1）return n_assigned；                // 再没有局部变量的静态引用数大于 1
    n_assigned + + ；
    switch（callee_saved_reg）
    {
        case esi_reg：esi_local = var_idx；break；
        case edi_reg：edi_local = var_idx；break；
        case ebp_reg：ebp_local = var_idx；break；
        case ebx_reg：ebx_local = var_idx；break；
        default：break；
    }
    ref_count[callee_saved_reg]=0；
}
return n_assigned;
```

图 9.4 简单全局寄存器分配算法

在第一行中有这样的语句：如果方法中含有异常处理，就预留给异常处理 3 个全局寄存器。

从上面的伪代码中，我们可以得到简单全局寄存器分配算法的基本思想，就是在所有的局部变量中挑出静态引用数最大且都大于 1 的 4 个（或少于 4 个），被调用者保存寄存器被分配给它们。静态引用数最大，即进行操作最多的局部变量放在寄存器中，就能够节省大部分读写内存的时间，这也是非常自然

的想法。而且，算法的时间复杂度非常小，为 O（B），其中 B 是字节码指令的长度。这事实上包括预遍历过程中收集各个局部变量的时间和挑出最大静态引用数的时间。

简单全局寄存器分配算法得到的结果将在整个方法的编译过程中保持，这就出现了一个问题：即使一个分配了寄存器的局部变量生命期已经结束，这个寄存器仍然不能分配给别的局部变量。也就是说两个生命期不重叠的局部变量不能共享一个寄存器，这就造成了寄存器资源的浪费。

2. 基于优先级算法

O1 编译器的基于优先级算法类似于 Chow 算法，这是一种基于优先级的图着色全局寄存器分配算法。图着色算法通常需要构造一个以基本代码块为顶点的干涉图，这个过程需要耗费较多的时间和空间。因此，O1 编译器对它加以改造，使之适用于 JIT 编译器的要求。

算法的思想是这样的：把所有的变量按静态引用数排序，静态引用数越大的变量优先级越大；按优先级顺序对每个局部变量 u，在控制流图上做深度优先搜索（DFS），从每一个用到变量 u 的基本块开始（u 的生命期结束的可能点）沿基本块的入边向后回溯，到定义 u 的基本块结束（u 的生命期开始点）；在深度优先搜索 u 的生命期覆盖的基本块时，跟踪记录被调用者保存寄存器的分配情况，如果寄存器 R 在所有这些基本块中都是空闲的，那么就把寄存器 R 分配给局部变量 u。在实际操作中，字节码程序并没有定义变量，在一个方法中对局部变量的操作只有装载（load）和存储（store），如何判断局部变量的生命期是一个需要解决的问题。这个算法的复杂度是 O（B+NV），其中 B 是方法中字节码指令的条数，N 是基本块的数目，V 是局部变量的数目。

在 O1 编译器中，类 Register_Allocator_Priority 实现了这个算法。类 Register_Allocator_Priority 的核心是一个类 Jcfg 的对象，它完成了生成控制流图、寄存器分配等主要任务。

类 Jcfg 生成控制流图的过程和预遍历过程很相似：逐条分析方法中的字节码指令，分辨出基本代码块的边界，记录每条转移边。不同的地方在于需要记录的信息，由于寄存器分配算法的需要，Jcfg 记录了这些信息：每个代码块进出的边，每个装载和存储的变量。Jcfg 把这些信息存储在结构 Jcfg_node 中。每一个 Jcfg_node 表示控制流图上的一个基本块信息结点，cfg_node_array 是所有基本块信息结点的列表。图 9.5 为 Jcfg_node 结构的定义。类 Jcfg 用两个整型数组来存储边的信息：forward_edge_array，backward_edge_array。数组中的每一个元素记录了边的向前或向后到达的基本块索引。也就是说对于一条从基本块 i 到基本块 j 的边，它被作为基本块 i 的出边存储在 forward_edge_array 数组中，相应元素的值为 j，同时它又作为基本块 j 的入边存储在 backward_edge_array 数组中，相应元素的值为 i。

```
struct Jcfg_node
{
    const unsigned char *bc_start;    // 基本块的第一条字节码指针

    const unsigned char *bc_end;      // 基本块的最后一条字节码指针

    unsigned bc_length;               // 基本块的所有字节码指令条数

    int *out_edges;                   // 基本块出边数组，内容为 forward_edge_array 某个元素地址

    int num_out_edges;                // 基本块出边的数目

    int *in_edges;                    // 基本块入边数组，内容为 backward_edge_array 某个元素地址

    int num_in_edges;                 // 基本块入边的数目

    Bit_Vector *loaded_vars, *stored_vars;    // 位矢量记录基本块是否有对每个局部变量做操作
```

```
    short allocation[4];                // 基本块内 4 个被调用者存储寄存器的分配情况
    unsigned char registers_available;  // 位矢量记录当前可获得（未分配）的寄存器
    unsigned char visited;              // 在做深度优先搜索的一次遍历中，该基本块是否已经遍历过
    int cur_out_edge_idx;               // 两个临时变量供深度优先搜索使用
    int cur_in_edge_idx;
};
```

图 9.5 基本块信息结点 Jcfg_node 结构

在控制流图生成后，分配算法中需要的信息也被记录下来。接下来，我们在图 9.6 中给出寄存器分配的伪代码。

```
void Jcfg :: register_allocate ( )
{
    建立 tmparray 整型数组存放各个变量的索引及静态引用数；
    tmparray 数组按静态引用数从高到低排序；
    // tmparray[i]放的是静态引用数排第 i 位的变量索引，tmparray[i+num_locals]放它的静态引用数

    for (int i=0; i<basic_block_count; i++)        // 开始时每一个基本块中所有全局寄存器可得
        cfg_node_array[i].registers_available =11101000B；
    for (i=0; i<num_locals; i++)
    {   // 为每个局部变量寻找可分配寄存器
        varnum=tmparray[i];
        if (tmparray[i+num_ocal]<2) break;      // 只考虑静态引用数大于等于 2 的局部变量

        for (i=0; i<basic_block_count; i++)
            cfg_node_array[i].visited = 0; // 一个变量 DFS 开始之前，所有基本块都未到达过
        dfs_result = ra_dfs(varnum,basic_block_count,cfg_node_array);
        // 深度优先搜索函数 ra_dfs 返回一个字节位矢量，表示可以分配给该变量的寄存器

        // 如果 dfs_result!=0，分配寄存器，修改基本块中寄存器分配情况
```

```
            regno = lowest_bit_set((unsigned char)dfs_result)；// 选取序号最低的一个寄存器
            for (i=0; i<basic_block_count; i++)
            {    // 在 DFS 中到达的所有基本块中记录 regno 寄存器被分配的情况
                 if (cfg_node_array[i].visited)
                 {    cfg_node_array[i].registers_available 的 regno 位上清零；
                      cfg_node_array[i].allocation[regno]=varnum；}
            } // 修改基本块中寄存器分配情况循环结束
      } //为每个局部变量寻找可分配寄存器循环结束
} //全局寄存器分配完成

static unsigned int ra_dfs (…)
{    // 返回值为寄存器的分配情况的位矢量
      unsigned char registers = 11111111B;     int result = 1;
      for(node=0; result && node<basic_block_count; node++)
      {    // 选择对变量有操作的基本块，调用 ra_dfs_1 开始做深度优先搜索
           if (cfg_node_array[node].visited) continue;
           if (cfg_node_array[node].loaded_vars->is_set(varnum) ||      // 以那些对变量有操作的
              cfg_node_array[node].stored_vars->is_set(varnum))         // 基本块作为 DFS 的开始点
                result = ra_dfs_1(varnum, basic_block_count, cfg_node_array, node, registers );
      }
}
static int ra_dfs_1(… register …)
{    // 返回值为 0 时结束该变量的 DFS，否则 ra_dfs 将继续寻找 DFS 的另一个开始点
      if (cfg_node_array[node].visited) return 1;
      cfg_node_array[node].visited = 1;
      registers &= cfg_node_array[node].registers_available;    // 叠代得到 DFS 此时可获得的寄存器
      if (registers == 0x0) return 0;          // 没有寄存器在 DFS 遍历到的基本块中都可得，失败返回
      if (cfg_node_array[node].stored_vars->is_set(varnum)) return 1;
      // 基本块中变量的写操作发生在读操作之前
      for (i=0; i<cfg_node_array[node].num_in_edges; i++)
```

```
    { // 递归调用 ra_dfs_1，沿着基本块入边做深度优先搜索
      if (!ra_dfs_1(varnum, basic_block_count, cfg_node_array, cfg_node_array[node].in_edges[i],
                    registers))    return 0;
    }
    return 1;
}
```

图 9.6 基于优先级全局寄存器分配

函数 ra_dfs 的作用是寻找做深度优先搜索的根结点，然后调用 ra_dfs1 从这个根结点开始沿基本块入边开始作深度优先搜索。如果 ra_dfs1 返回 0，则表示没有可以分配给该变量的寄存器，于是 ra_dfs 返回；否则，ra_dfs 寻找下一个根结点，继续做深度优先搜索。从 ra_dfs1 函数中，我们可以看到深度优先搜索的一条路径在这四种情况下结束：基本块已经访问过；在没有可分配的寄存器；基本块没有入边；在基本块中变量的写操作发生在读操作之前。最后一种情况需要解释一下：当对一个变量做写操作（store）时，这个变量以前的值将被注销（kill），将被视为另一个值的生命期的开始，所以认为是深度优先搜索的回溯过程的终结处；但是当写操作发生在读操作（load）之后，我们并不希望深度优先搜索在此结束，因为在一个基本块内变量之前的值并没有被注销，写之前的读操作得到的值仍然是前一个基本块中该变量值的延续。所以实际操作中，变量的生命期可以看成是变量所有定值的生命期。

在基于优先级的全局寄存器分配结束之后，可能会有一些基本块，不是所有的被调用者保存寄存器都被分配出去。这些空闲的被调用者保存寄存器将会被局部寄存器分配器利用起来。

9.2.2.2 局部寄存器分配

调用者保存寄存器通常用作局部寄存器、涂写寄存器（scratch registers），包括了 EAX、ECX、EDX 三个寄存器。当 O1 编译器为表达式计值时，往往需要局部寄存器来存放临时表达式的值，这时候就由局部寄存器分配器来选择一个局部寄存器来满足这个表达式计值的需要。在 O1 编译器中，又把局部寄存器分配的过程叫做寄存器管理。寄存器管理不但要协调调用者保存寄存器的使用，还要充分利用起空闲的被调用者保存寄存器。

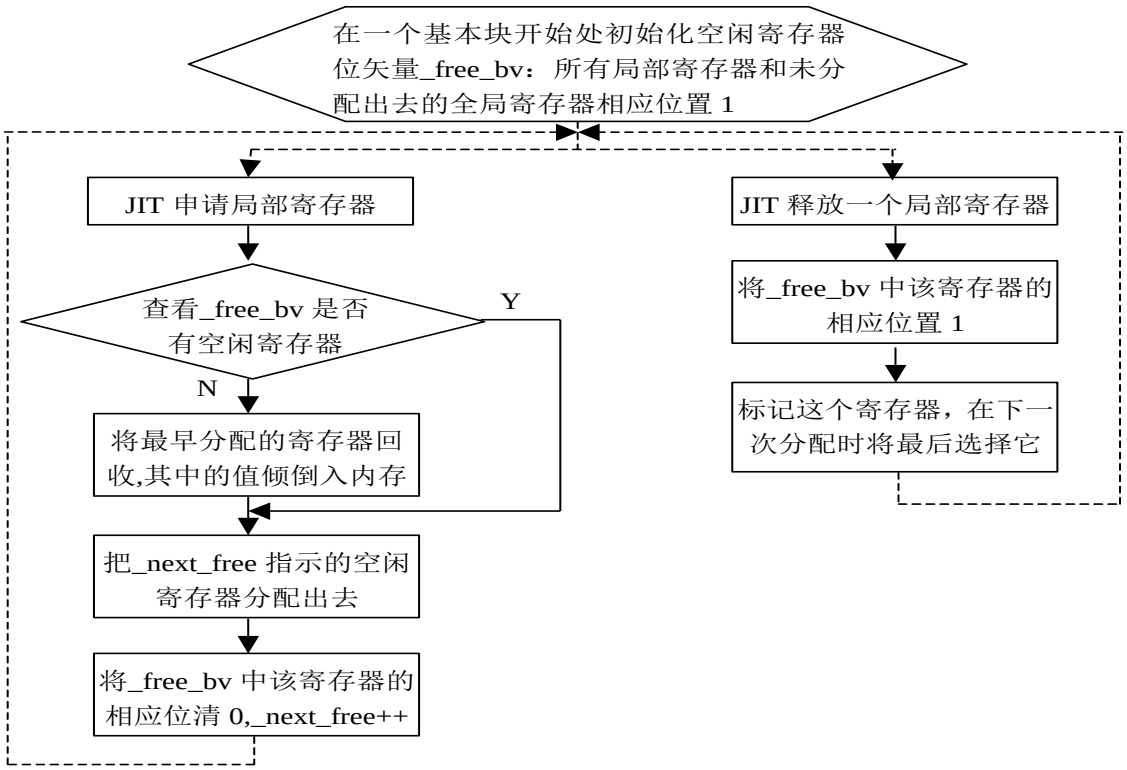


图 9.7 局部寄存器管理

O1 的代码选择器通常在这两种情况下需要一个局部寄存器：

- 目标代码中就会出现大量在内存、内存和寄存器间移动值的指令，当需要把一个内存中的值移到另一个内存中去的时候，由于 x86 处理器中没有在两个内存单元间移动的 mov 指令以这时候需要一个寄存器来作为中转的空间。

```
mov [ esp+offset ]，[ eax－4 ]          ；错误！在两个内存单元中移动值
mov   ecx，[ eax+4 ]                    ；用涂写寄存器 ecx 作为中转单元
mov [ esp+offset ]，ecx
```

- x86 处理器中双操作数运算指令的两个操作数中除源操作数为立即数的情况外，必须有一个操作数在寄存器中。而且大多数的运算指令是破坏目标操作数的：dst = src op dst，所以在翻译机器指令的时候总是使目标操作数是一个涂写寄存器。

在这些情况下，O1 的代码选择器就会向寄存器管理申请一个局部寄存器。O1 编译器的寄存器管理通过类 Register_Manager 实现，它的管理算法也比较简单。当编译器需要局部寄存器时，将会调用 Register_Manager 的方法 get_reg，这个方法将会返回一个空闲局部寄存器的寄存器号。图 9.7 给出了 Register_Manager 类的管理流程。

Register_Manager 类用一个字节的位矢量_free_bv 来记录 8 个通用寄存器的使用情况，每一个寄存器相应于一位，该位为 1 时该寄存器空闲，否则已被分配出。

7	6	5	4	3	2	1	0
EDI	ESI	EBP	ESP	EBX	EDX	ECX	EAX

在开始编译每一个基本块的时候，所有的局部寄存器中存放的值都会存入内存中，这个过程也叫倾倒（spill）。空闲寄存器位矢量_free_bv 中，局部寄存器相应的位都置为 1，如果在这个基本块中还有被调用者保存寄存器没有分配出去，则相应的 bit 位上也置为 1。

Register_Manager 还有一个重要成员：_next_free。它提示了下一个应该分配的空闲寄存器，在一个基本块的开始处_next_free 被初始化为 1，也就是说 ECX 寄存器总是做早被分配出去。当_next_free 所指示的寄存器被分配出去后，_next_free 循环加 1。例如一个基本块_free_bv 被初始化为 00000111，_next_free 初始值为 1，ECX 被分配出去后_next_free 变成 2；EDX 又被分配出去后，_next_free 不断加 1，但是 3、4、5、6、7 号寄存器都不空闲，那么_next_free 等于 0；之后_next_free 按此过程循环。当一个局部寄存器被释放时，_next_free 将会指向这个寄存器的下一个空闲寄存器，这样刚被释放的寄存器就不会马上又被分配出去，它会等到下一轮分配时最后被选择到。这样用_next_free 作为指示，事实上一种循环分配策略，在 O1 进行删除公共子表达式优化时将受益于这一策略，我们将在 9.3.1 节提到。

当编译器调用 get_reg 函数，向 Register_Manager 提出局部寄存器请求时，如果_free_bv 指示有空闲寄存器就把_next_free 指示的寄存器分配出去，_next_free 循环加 1。如果已经没有空闲寄存器，将会选择一个寄存器释放，按照循环分配策略，这个寄存器应该是最早被分配出去的，这个寄存器可以通过查找 Java 操作数栈中处在最深位置的寄存器操作数得到（O1 的 Lazy 代码选择算法用了一个模拟栈来模拟 Java 操作数栈的行为，所以可以查找这个模拟栈来得到最早被分配出去的寄存器）。编译器将会产生把这个寄存器中的内容倾倒到内存中的指令，以后需要这个值时就应该到内存中读取，而寄存器就被释放出来，相应_free_bv 位就会被置为 1，释放出来的寄存器就能够被分配出去。

当编译器产生完为一个表达式计值的指令序列之后，用在源操作数中的局部寄存器会被释放，Register_Manager 类将把这些被释放的局部寄存器对应的_free_bv 中的位矢量置为 1，并相应地调整_next_free。

9.2.3 Lazy 代码选择

O1 编译器高效快速的关键在于用 Lazy 代码选择算法。算法通过对字节码的一次遍历，直接在一个临时代码缓冲区产生机器代码，机器代码最后将在代码发射阶段拷贝到内存中的最终位置上。在这一节中，我们就对 O1 编译器产生机器代码的过程，特别是 Lazy 代码选择算法进行详尽的介绍。

9.2.3.1 mimic 模拟栈和操作数

Lazy 代码选择算法为了达到较高的编译速度，在代码选择中始终注意这两个问题：第一，在局部寄存器中保存中间值，例如 Java 操作数栈中的值；第二，为了减少寄存器管理的压力，充分利用 IA32 丰富的寻址模式，尽量把立即操作数装载和内存操作数访问合并到运算指令中去。在解决这两个问题时，Lazy 代码选择算法把源操作数信息通过一个辅助数据结构 mimic 栈保存、传递。mimic 栈顾名思义是一个模拟栈，它将模拟 Java 操作数栈运行时的状态。

我们知道，对每一个 Java 方法调用，Java 虚拟机分配一个 Java 栈帧（Frame），它包含一个操作数栈的片断。绝大多数的 Java 虚拟机字节码指令从当前操作数栈中取值，对它们进行操作，并把结果返回到同一个操作数栈。操作数栈也用于向方法传递参数和接收方法的结果。而 mimic 栈就是要模拟字节码指令对操作作数栈的影响，反映出操作数栈上操作数的变化。在为每一条字节码指令选择指令序列时，指令序列的源操作数从 mimic 栈中弹出，而指令序列执行的结果仍然被压入 mimic 栈中，作为后续指令序列的源操作数。

O1 编译器中类 Stack 实现了 mimic 栈，类 Stack 主要完成下面这一些工作：

- 模拟 Java 操作数栈：

  Stack 类中实现了一个栈数据结构，包括栈空间、栈顶指针。数据成员

  ```
  Operand     ** _elems;
  unsign      _top;
  ```

  分别是栈空间的指针和指示栈顶整型数。Stack 类初始化时，分配了一个等同于 Java 操作数栈最大深度的操作数指针数组作为栈空间，这个最大深度是在预遍历过程中收集到的信息，elems 指向这个指针数组的首地址。栈空间中的每一个元素都是一个类 Operand 的对象，它代表了一个操作数。

在 Stack 类初始化时，栈顶指针_top 的值为 0，_elems[_top-1]总是此时栈中的栈顶元素。Stack 类也有相应的方法来实现栈的压入、弹出操作。

```
Operand *pop ( );                                    // 弹出栈顶元素(32 位)

void pop (unsigned n);                               // 弹出栈顶 n 个元素

void pop64 (Operand *&opnd_lo, Operand *&opnd_hi);   // 弹出栈顶元素(64 位)

void push (Operand *opnd);                           // 将 32 位操作数 opnd 压入栈中

void push64 (Operand *opnd_lo,Operand *opnd_hi);     // 将 64 位操作数 opnd 压入栈中
```

- 在适当的时候把寄存器操作数中的内容倾倒入内存：

  在 9.2.2.2 节局部寄存器分配的介绍中，我们已经知道：在回收一个局部寄存器时，需要找到最早分配出去的那个寄存器，将其中的内容倾倒入内存。这个工作将由 Stack 类完成，当寄存器管理类 Register_Manager 需要回收一个局部寄存器时，它将调用 Stack 类的方法 spill_one 来完成上面的工作。mimic 栈模拟了 Java 操作数栈的动态，所以在需要找到最早分配出去的局部寄存器的时候，mimic 栈会从栈底元素开始查找，第一个被找到的局部寄存器操作数必然是最早分配出去的。Stack 类用了一个小技巧来使查找的范围更小，它有一个数据成员_spill_mark。初始化时_spill_mark 为 0，当第一次在栈中查找局部寄存器操作数时，从栈底_elems[0]开始，如果找到第 i 个元素是局部寄存器操作数，那么更新_spill_mark 为 i+1。这样在下一次查找时，只需从_elems[_spill_mark]开始查找，因为_spill_mark 之前的操作数必然不是局部寄存器操作数。找到要回收的寄存器操作数后，将产生把其中内容倾倒入内存中的代码，这时 mimic 栈中原寄存器操作数所在位置的元素将会被现在存放内容的内存操作数所取代。

mimic 栈中的操作数元素用 Operand 类建立模型，在 O1 编译器的 C++实现中，Operand 类的层次结构如图 9.8 所示。

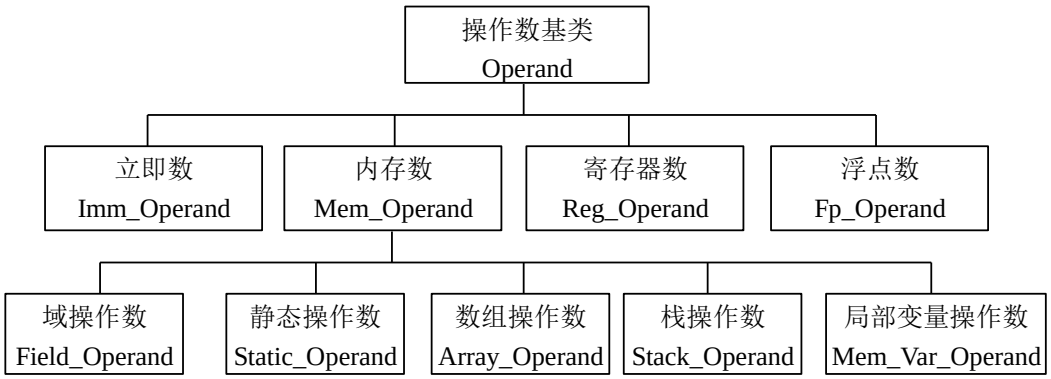


图 9.8 Operand 类层次结构图

mimic 栈上的每一个 Operand 类对象对应于一个我们在 8.3.2 节提到的 x86 指令操作数对象，从 Operand 类衍生的各个子类中几乎都有一个成员 opnd 是相应 x86 指令操作数类的对象。Operand 类是所有操作数类的基类。它衍生出 4 个主要的操作数子类，其中内存操作数子类又衍生出 5 个子类。表 9.1 给出了各个子类所表示的具体操作数。

Operand 子类	相应的 Opnd 子类	操作数说明

Imm_Operand	Imm_Opnd	立即操作数，可以作为计算指令中的一部分
Reg_Operand	R_Opnd	寄存器操作数，它的值存放在物理寄存器当中；能够被大多数寄存器计算指令直接访问
Fp_Operand	\	放在 IA32 浮点寄存器操作数栈中的浮点操作数。Fp_Operand 类没有相应的 x86 指令操作数对象，但是它具有一个域 fpstack_cnt 记录了操作数在浮点寄存器栈中的位置。
Field_Operand	M_Base_Opnd	操作数是一个类的域成员引用，采用相对寻址方式
Static_Operand	M_Patch_Opnd (M_Opnd 的变体)	操作数是一个类的静态域成员，还可能是一些浮点常数。采用直接寻址或是相对方式。
Array_Operand	M_Index_Opnd	操作数是数组元素，采用变址寻址方式
Mem_Var_Operand	M_Var_Opnd	操作数是方法的参数或者局部变量，采用相对寻址方式，其基址寄存器是栈顶寄存器
Stack_Operand	M_Spill_Opnd	操作数是 x86 运行栈帧中的某个位置地址，采用相对寻址方式，其基址寄存器是栈顶寄存器

表 9.1 Operand 各个操作数说明

当代码产生器从 mimic 栈中弹出一个元素作为指令操作数时，这个 Operand 类对象会通过相应的 x86 操作数对象产生指令的操作数部分的机器码，使指令能够准确地访问到这个操作数。

9.2.3.2 Lazy 代码选择的基本思想

在这一节中，我们将介绍 Lazy 代码选择的基本思想，O1 编译器中对算法的实现以及各种支持代码生成的数据结构将在后续的几个小节中详细介绍。

实现 Lazy 代码选择算法的代码选择器力求在对字节码进行一次遍历后就能产生质量较好的机器代码，但是它并不是在遍历的过程中逐条字节码翻译。它在遍历字节码的过程用 mimic 栈模拟字节码执行时 Java 操作数栈中操作数的进出，只有当其中的操作数作为运算指令的操作数时，才为这条指令选择一组合适的机器代码，访问指令的操作数进行运算，运算得到的目标操作数依旧放到 mimic 栈中，作为后续指令的操作数。

当代码选择器为一个字节码 B 选择运算指令时，B 的操作数 O 从 mimic 栈中弹出，代码选择器将尝试把操作数直接合成到运算指令中。如果尝试成功，那么字节码 B 的指令选择成功。否则，如果操作数 O 不能被直接合成到指令中，那么代码选择器将向寄存器管理器申请一个局部寄存器，代码选择器将会产生一组代码把操作数 O 先放入局部寄存器中，然后产生一条用这个局部寄存器作为操作数的运算指令。例如，字节码 B 是加法运算 iadd，它把 Java 操作数栈中的栈顶两个整型数相加，相加的和压入操作数栈栈顶。代码选择器为 B 选择指令时，它从 mimic 栈中弹出栈顶两个操作数 O1、O2，试图产生一条加法机器指令：

ADD　　O1，O2

但是，我们已经知道 x86 双操作数指令的两个操作数，如果没有立即数，那么至少有一个必须是寄存器数。如果 mimic 栈弹出的 O1、O2 都是 Mem_Operand 对象，即都是内存操作数，那么代码选择器是不能直接产生这条加法指令的。这时代码选择器就会向寄存器管理器申请一个局部寄存器，如果寄存器管理器返回一个空闲局部寄存器 ECX，那么代码选择器将先产生一条把 O1 移动到寄存器中的指令，然后才生成加法指令：

MOV　　ECX，O1

ADD　　ECX，O2

代码选择器在产生这两条指令后，把表示 ECX 的寄存器操作数，一个 Reg_Operand 对象压入 mimic

栈中。为什么是操作数 O1 移动到寄存器当中，而不是操作数 O2 移动呢？我们在 9.2.2.2 局部寄存器分配中已经提到：大多数的运算指令是破坏目标操作数的，所以我们总是为目标操作数选择一个可涂写的局部寄存器。

一般来说根据 Lazy 代码选择的策略，操作数的进栈、出栈动作只在 mimic 栈上进行模拟，机器代码的选择延迟到操作数作为某条指令的操作数时才进行。但是为了保证能够产生正确的机器代码，在下面这两种情况中 Lazy 代码选择策略是不适用的：

- 对局部变量存储的字节码，如 istore，它们把 Java 操作数栈栈顶元素存储到某个局部变量中。
- iinc 字节码，它将某个局部变量增加常数值。

这两种情况都涉及到局部变量的更新，代码选择器必须马上为这种更新选择机器代码。因为在这两种指令之前和之后，mimic 栈中被更新局部变量操作数对应的值已经不相同了。

图 9.9 给出了一个 O1 代码选择器产生代码和 mimic 栈变化的例子。

在这个例子中，javac 编译器为 Java 代码 z = x+1 产生了一组字节码。代码选择器在遍历到第一条 iload 字节码指令时，根据指令语义在 mimic 栈中压入局部变量 x。需要说明的是为什么压入栈的是寄存器操作数 ESI。这是因为我们假定在全局寄存器分配中，全局寄存器 ESI 被分配给了局部变量 x。当一条字节码对一个局部变量做操作时，代码选择器总会先查看全局寄存器分配情况，如果为这个局部变量分配了一个全局寄存器，那么所有对这个局部变量的操作将实施于这个全局寄存器上。

接着，立即数 1 被压入了 mimic 栈，直到这时候都没有机器代码产生。

然后就是加法运算 iadd，这时 mimic 栈顶的两个操作数 1、ESI 被弹出栈。我们总是希望运算指令的目标操作数放在局部寄存器中，所以向寄存器管理器申请并得到一个空闲的局部寄存器 EAX，产生了图中所示的 MOV 和 ADD 机器代码，完成了对三条字节码的代码选择，并且把运算指令的目标操作数 EAX 压入 mimic 栈中。

最后一条字节码指令把运算结果从 mimic 栈中弹出并存储到局部变量 z 中，由于是局部变量更新指令，这条字节码的代码选择马上进行并不做任何延迟。代码选择器产生了一条从寄存器到内存中局部变量的 MOV 指令，局部变量存储在 x86 运行栈帧特定的位置，总是相对栈顶寄存器 ESP 寻址，z 是该局部变量相对栈顶的偏移。

如果在为最后一条 istore 字节码选择指令之前，mimic 栈中有一个 Mem_Var_Operand 操作数对象 O1 表示局部变量 z，O1 总是对应 z 中的值，而我们需要的操作数应该是 z 更新之前的值。如果 mimic 栈不采取任何动作，那么在 O1 被代码选择器作为操作数被合成到后续指令中时，将会把 z 更新之后的值作为操作数。为了保证使用正确的操作数，产生正确机器代码，在为 istore 选择指令之前，代码选择器将会产生一条 MOV 指令把局部变量 z 的值存储到某个内存单元中去，mimic 栈中 O1 将会被表示这个内存单元的操作数对象 O2 所替换。这样在为 istore 选择代码之后，O2 对应的操作数仍然是 z 未更新时的值。

Java 源代码： z = x + 1 // 其中 x，z 为局部变量
字节码： iload x ；将局部变量 x 装载到 Java 操作数栈中
iconst_1 ；将常数 1 装载到 Java 操作数栈中
iadd ；弹出栈顶两元素相加，结果压入操作数栈
istore z ；弹出栈顶元素（相加结果），放到局部变量 x 中

Lazy 代码选择过程：

字节码	机器代码	mimic 栈
iload x		ESI
iconst_1		1 ESI
iadd	**MOV** EAX，ESI **ADD** EAX，1	EAX
istore z	**MOV** z[ESP]，EAX	

图 9.9 Lazy 代码选择器为 z=x+1 产生机器指令序列

从上面的描述和例子中可以看出，Lazy 代码选择的优点就在于：用 mimic 栈模拟操作数栈，把选择访问操作数的指令推迟到操作数被使用之时；这样更能利用 IA32 丰富的寻址方式把操作数的访问合并到运算指令中去，且能够减少一些没有必要的移动值的动作。

9.2.3.3 临时代码及数据缓冲区

代码选择器产生的代码将放在一个临时的缓冲区，在代码回填结束后，再从这个临时区域拷贝到代码的最终位置。在 ORP1.0.9 版本的 O1 编译器中，这个临时代码缓冲区由 Code_Emitter 类实现，Code_Emiiter 类向 ORP 的内存管理器申请了一块内存空间作为缓冲区，并且向代码选择器提供了访问这个缓冲区的接口。

```
class Code_Emitter {
private：
    Arena    *_arena；          // 构成代码缓冲区的实存块链表，表头的实存块是最近分配的一个
    unsigned    _arena_begin_offset；        // 表头实存块的第一条指令相对于第一条指令的偏移
    char    *_next；            // 将要产生的下一条指令的位置
    char    *_end；             // 作为一个哨兵，检查产生的指令是否越过了代码缓冲区的边界
    Mem_Manager&    _mem_manager；    // ORP 的内存管理器对象指针
    void    _check_inst_space ()；          // 保证缓冲区中有足够的空间存放下一条指令
    void    _alloc_arena(unsigned nbytes)；  // 向内存管理器申请分配一个实存块
public：
```

```
    /* 下面一组域成员记录了-statistics 开关打开时用于重编译机制的信息：统计调用次数的计数
    器的地址和在方法中的偏移以及 Profiling 数据记录 */
    unsigned* offset_buf ；
    unsigned   offset_buf_offset；
    struct Profile_Rec* prof_rec；
    unsigned   inner_bb_cnt_offset；

    unsigned get_size ( )；      // 得到已经产生的机器代码的字节数

    void copy (char *buffer)；// 这个函数把缓冲区中的代码拷贝到 buffer 引导的内存中去

    /* 一系列的代码产生函数，包括运算指令、控制流指令、栈操作指令、调用函数之前和函数
    返回前的例程指令序列。这些函数将会在缓冲区中生成机器代码 */
    void emit_inc (const M_Opnd *m)；
    void emit_mov (const R_Opnd *r，const RM_Opnd *rm )；
    … …
}
```

图 9.10 临时代码缓冲区类 Code_Emitter

临时代码缓冲区有若干个实存块（Arena）构成，每一个实存块有 1024 字节。在产生每条指令之前，_check_inst_space 函数都会被调用，检查缓冲区中是否还有足够的空间存放一条指令。最长的 x86 机器代码是 14 个字节，所以_check_inst_space 函数将检查缓冲区中剩下的空间是不是有 14 个字节，如果没有，它将会调用_alloc_arena 函数向 ORP 的内存管理器申请一个新的实存块。新的实存块总是被链接在_arena 链表的头部，所以新生成的代码总是放在表头的实存块中。_next 指针指向了表头实存块的第一个空闲的内存单元，将要产生的下一条机器就从这个单元开始存放。_end 指针指向的是表头实存块的最后一个内存单元，它是缓冲区的边界，产生的代码不能越过这条边界。图 9.11 较形象地描绘了临时代码缓冲区的数据结构。

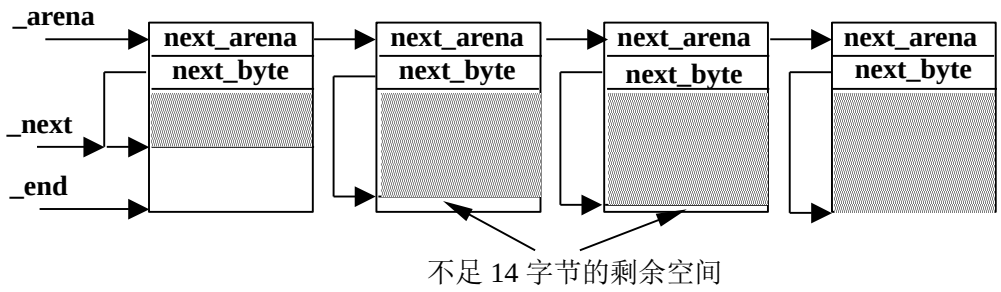


图 9.11 代码缓冲区实存块的示意图

Code_Emitter 类有一组相对应于 x86 机器指令的函数，它们的作用是在_next 指针开始的若干个内存单元中生成相应的 x86 机器指令。在机器指令生成之后，_next 指针向后移动到缓冲区的空白单元。在代码生成之后的代码发射阶段，Code_Emitter 类的 copy 函数将会被调用，从链表的最后一个实存块开始到表头实存块中的内容拷贝到指定的内存位置。

在程序编译的过程中，有一部分数据是静态可确定的，例如常数。编译器通常都把这部分数据放在一个静态数据区，以便于目标机器代码能够直接访问这些数据。在 JIT 的编译过程中也为这部分数据分配了一块缓冲空间，静态数据的值会暂时放到这个空间内，在代码发射阶段，缓冲空间中的内容将会被复制到最终的静态数据区内。Data_Emitter 类是对这个临时数据缓冲区的实现。

```
class Data_Emitter {
public：
    Data_Emitter (Mem_Manager& m,unsigned sz)；// 构造函数，分配大小为 sz 空间作为缓冲区
    void emit_float (float f)；              // 在_offset 偏移处写入一个 32 位浮点数 f
    void emit_double (double d)；            // 在_offset 偏移处写入一个 64 位浮点数 d
    void emit_int (int i)；                  // 在_offset 偏移处写入一个 32 位整数 i
    unsigned get_offset ()；                 // 获得缓冲区_offse 的值
    unsigned get_size()；                    // 获得缓冲区大小
    Data_Label *make_label ()；              // 为_offset 偏移处的内存单元制作一个数据标签
    void copy (char *data_block)；           // 将缓冲区中数据复制到 data_block 指定内存中
private：
    Mem_Manager& mem_manager；       // ORP 内存管理器对象指针
    unsigned size, _offset；         // 缓冲区大小；_offset 指示了下一个数据写入的位置
    Data_Label *_labels；            // 缓冲区中所有数据标签的链表
    char *_buffer；                  // 缓冲区的内存空间
};

class Data_Label {                   // 数据标签类
public：
    Data_Label (unsigned off，Data_Label *n)：offset(off), next(n),code(NULL) {} // 构造函数
    unsigned offset；                // 该标签在缓冲区的位移
    Data_Label *const next；         // 下一个标签对象
    char *code；                     // 该标签指示的数据块单元的值
```

```
    void apply(char *data_block) ；          // 在 data_block 相应的位置填上数据标签的值
};
```

图 9.12 临时数据缓冲区类 Data_Emitter

当编译器需要产生一个新的静态数据时，将会调用 Data_Emitter 的 emit 函数在缓冲区的_offset 指示的内存单元中写入新的数据，然后_offset 向后移动相应的字节数。如果编译器需要产生一个新的静态数据，但是以后还需要改写这个数据单元的内容，例如编译器要为跳转语句的目标处代码产生一个标签，但是很可能还不知道这个跳转目标的偏移，需要在后续的编译过程中将它补上，在这种情况下，Data_Emitter 为这个数据所在的一个内存单元制作了一个数据标签。数据标签由 Data_Label 类实现，它指示了数据在数据块中的偏移，编译器能够通过数据标签访问数据缓冲区相应的内存单元。

9.2.3.4 运行活动记录栈帧

在程序运行时，方法的一次执行（也叫一个活动）所需要的信息用一块连续的存储区来管理，这块存储区叫做活动记录或帧（Frame）。活动记录通常用控制栈来实现，当调用出现时，一个活动的执行被中断，有关机器状态的信息，如程序计数器（PC）和全局寄存器的值，就保存在这个栈中。当控制从调用返回时，在恢复了有关寄存器的值和程序计数器之后，原先的活动能够继续。生存期包含在这个活动中的数据对象也可以分配在这个栈中，与这个活动的其他有关信息放在一起。不同的编译器活动记录栈帧的组成域不尽相同，O1 也有自己的栈帧结构。

图 9.13 显示了一个方法被调用时活动记录帧的组成。在 O1 编译器中，有两种不同的栈帧结构，一种是基于栈顶指针 ESP 的栈帧结构，另一种是基于活动帧指针 EBP。通常的栈帧结构是基于 ESP 的，当方法中包含异常处理代码时，将采用基于 EBP 的栈帧结构。基于 EBP 的栈帧结构紧接着返回 IP，在栈上压入 EBP 的值，而 EBP 寄存器中则存放当时栈顶指针 ESP 的值。在异常处理中需要回退这个基于 EBP 的栈帧时，只需要使将 EBP 中的值赋给 ESP，这个栈帧就被迅速地从栈中弹出。两种栈帧结构的区别还体现在对栈中元素寻址方式的不同上。基于 ESP 栈中元素的寻址模式总是：[ESP+offset]，基址寄存器是 ESP；而基于 EBP 栈中元素的寻址模式总是：[EBP+offset]，基址寄存器是 EBP。

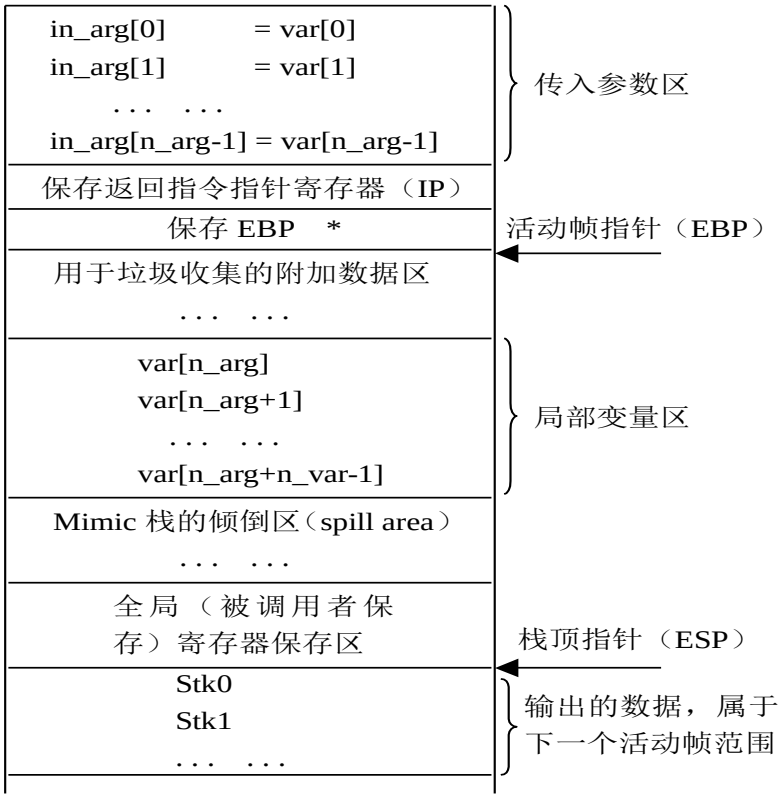


图 9.13 一个活动记录栈帧示意图

在活动记录帧的最底部总是传入这个方法的参数，由调用者压入栈中。调用指令会把调用方法之后的下一条代码地址压入紧接着的一个单元，当调用返回时这个单元的值将会覆盖到 IP 寄存器中。基于 EBP 的栈帧在接下来的一个单元中将保存调用方法的 EBP 寄存器的值，而 EBP 寄存器的值会被这个栈单元的地址，也就是当时的栈顶指针所代替。

在接下来的一个附加数据区中，将存放关于方法变量的一个位矢量。这个位矢量的位长等于函数变量（包括参数和局部变量）的个数，当某一位为 1 时表示对应的变量是一个类或者是一个数组的引用。这个信息将会被垃圾收集所使用。

方法的局部变量存放在紧接着的一个区域，当方法调用返回，活动记录栈帧被弹出，这些局部变量的生命期也随之结束。方法的参数和局部变量通称变量，在 mimic 栈中的表示都是以 Mem_Var_Operand 对象出现。它们采用相同的寻址方式，即 M_Var_Opnd 对象的相对寻址方式，基址寄存器是 ESP 或者 EBP，究竟是哪一个决定于方法采用的是基于 ESP 的栈帧还是基于 EBP 的栈帧。

接下来的一个区域叫 spill 区，它具有的字长等同于方法的 Java 操作数栈最大深度字长（这个信息在预遍历阶段已经得到）。我们从 9.2.3.1 及 9.2.3.2 两节中已经知道，在代码选择的过程中可能要把 mimic 栈中的某个寄存器操作数倾倒到内存中，spill 区就是这个 mimic 栈用来倾倒操作数的区域。被倾倒的操作数在 spill 区中的位置与它对应的 Operand 对象原先在 mimic 栈中的位置一致。这个区中的操作数在 mimic 栈中的表示都是以 Stack_Operand 出现，相应的 x86 指令操作数是 M_Spill_Opnd，采用相对寻址方式，基址寄存器也是 ESP 或者 EBP。

从寄存器分配一节（9.2.2）中，我们知道全局寄存器：EBX、EBP、ESI、EDI，是被调用者保存寄存器，方法被调用时需要保存这些寄存器的值。这些全局寄存器保存在紧接着 spill 区的空间。由于基于 EBP 栈帧之前已经保存过 EBP 寄存器的值，所以在基于 EBP 栈帧中这个区域只保存 EBX、ESI、EDI 三个寄存器的值。

最后一个部分是方法将要调用另外一个方法时提供的参数，它也是被调用方法的输入参数。调用者和被调用者的栈帧在这里出现重叠，或者可以说这部分已经属于被调用方法的栈帧的组成部分。

ORP1.0.9 版本中用类 Frame 记录了一个方法栈帧的信息，并提供了寻找栈帧中特定区域特定栈空间偏移的方法。

```
class Frame {
public:
    const unsigned n_args;              // 传入参数的个数
    const unsigned n_vars;              // 局部变量的个数
    const unsigned n_extra;             // 附加数据区的字长
    const unsigned n_spill;             // spill 区的字长
    const unsigned n_callee;            // 要保存的全局寄存器的数目
    const X86_Reg_No base_reg;          // 栈帧是基于哪个寄存器的结构，EBP 或是 ESP

    void push(unsigned n=1) {n_stack += n; }   // 在栈帧中压入一个输出参数
    void pop(unsigned n=1)  {n_stack -= n; }   // 从栈帧中弹出一个输出参数
    void clear()            {n_stack = 0; }    // 清空输出参数
    // 下面一组虚函数的实现取决于是用那一种结构的栈帧，是基于哪个基址寄存器。
    virtual int var_offset(unsigned n) = 0;          // 提供第 n 个变量相对基址寄存器的偏移
    virtual int spill_offset(unsigned s) = 0;        // 提供 spill 区第 s 个元素相对基址寄存器的偏移
    virtual int extra_offset(unsigned x) = 0;        // 提供附加区第 x 个字相对基址寄存器的偏移
    virtual int callee_offset(X86_Reg_No reg) = 0;   // 提供某个全局寄存器相对基址寄存器的偏移
    virtual int outarg_offset(unsigned n) = 0;       // 提供第 n 个输出参数相对基址寄存器的偏移
    virtual bool contiguous_loc(unsigned i, unsigned j) = 0;
    // 判断两个变量（局部变量或传入参数）在栈帧中的位置是否相连。
protected:
    Frame ( … ) {}      // 构造函数
    unsigned n_stack;   // 输出参数的个数
};
```

图 9.14 Frame 类源码

9.2.3.5 方法编译信息结构

O1 实现中用 Jit_Method_Info 结构存储在编译过程中方法的相关信息。

```
typedef struct Jit_Method_Info {
    const char *name;                   // 方法名
    unsigned code_len;                  // 字节码长度
    const unsigned char *code;          // 字节码数组
    unsigned cnt;                       // 用来跟踪当前调用点的计数器
    unsigned num_spills;                // 活动栈帧中 spill 区的大小
    unsigned num_gc_tag_words;          // 用来存储方法变量 GC 信息的位矢量字长
    unsigned num_in_args;               // 方法参数的数目
    unsigned num_vars;                  // 方法局部变量的数目
    unsigned num_callee_saved_regs;             // 可用的全局寄存器数目
    short esi_local;                    // 这 4 个域给出了全局寄存器分配的结果：
    short edi_local;                    // 全局寄存器和局部变量的映射
    short ebx_local;
    short ebp_local;
    char *vars_register_allocated;      // 局部变量是否有分配全局寄存器的标志链表
    unsigned char registers_saved;      // 标识那些分配给局部变量的全局寄存器的位矢量
    struct Runtime_Register_Allocator rra;
    // 该结构提供运行时寄存器分配信息支持，由相应的 Register_Allocator 类对象提供相关函数
    Runtime_Throw_Info *runtime_throws;  // 记录方法运行时可能抛出异常的 try 语句偏移的链表
    unsigned num_call_sites;            // 方法中出现调用点的数目
    char *var_type_vector;              // 方法局部变量类型的数组
    unsigned frame_size;                // 方法的活动栈帧的大小
```

```
    unsigned is_esp_based;          // 方法的活动栈帧的结构是否是基于 ESP 的

    Call_Site_Info cs_info[1];      // 调用点信息数组的头指针

} Jit_Method_Info;
```

图 9.15 方法信息结构 Jit_Method_Info

从上图中的方法信息结构源码中，我们可以看到 Jit_Method_Info 记录了这几类的信息：

- 基本信息：包括方法的名字、字节码长度、传入参数的数目等
- 活动栈帧信息：包括栈帧大小、结构类型等，有了这些消息能够重新构造出方法的 Frame 对象
- 全局寄存器分配信息。
- 调用点信息。

其中需要详细说明的是调用点（call site）信息。在一个方法中通常都有调用点，这其中可能是调用了另外一个方法，也可能是调用一些运行时支持例程，比如分配一个新对象或者数组边界检查例程。在这些调用点，程序的控制将会发生转移，需要记录一些信息，以便于调用返回时能够恢复先前的活动。由于 ORP 把调用点作为 GC 安全点，调用点信息还应该包括当时 mimic 栈中所有单元的引用情况。调用点信息由 Call_Site_Info 结构存储，图 9.16 给出了结构的各个域。我们将在后面的几节中看到这些域的用处。预先分配给 cs_info 链表的 Call_Site_Info 结构的数目等于预遍历过程中计算得到的 num_call_site。

```
typedef struct Call_Site_Info {
    unsigned stack_depth;               // 在调用点 mimic 栈的深度
    unsigned short num_out_args;        // 在调用点方法输出的参数数目
    unsigned short num_records;         // esp_record 数组的大小
    unsigned call_IP;                   // 调用序列的开始地址（第一条参数压栈指令地址）
    unsigned precall_IP;                // 实际调用指令地址（call 指令的地址）
    unsigned ret_IP;                    // 紧接在 call 指令后的代码地址（返回地址）
    struct Esp_Record *esp_record;      // 栈顶指针记录数组
    unsigned char is_caller_pop;        // 是否由调用者弹出参数
    unsigned char outarg_bv;            // 输出的参数中哪一个是引用
    unsigned char returns_ref;          // 被调用的函数是否返回一个引用
    unsigned short type_vector_length;  // type_vector 数组的长度
    void *m_handle;                     // 用来找到是一个引用的输出参数
    char *type_vector;                  // 该数组中包含 mimic 栈中各项的类型（是否是引用）
    const unsigned char *bc;            // 这个调用点相应的字节码的指针
    short register_mapping[4];          // 每个全局寄存器寄存器放的局部变量
} Call_Site_Info;
```

图 9.16 调用点信息结构 Call_Site_Info

9.2.3.6 生成机器代码

O1 编译器在经过了预遍历、寄存器分配两个阶段，并且产生了临时代码缓冲区、mimic 栈、方法信息块这些辅助数据结构的类对象之后，就可以开始进入代码选择阶段，运用 Lazy 代码选择策略为方法产生机器代码。在 ORP1.0.9 中，Lazy 代码选择是由 select_code 函数（orp-1.0.9/arch/ia32/ia32_o1_jit/lazy_code_selector.cpp ）完成。select_code 函数的工作不单包括代码生成还包括了快速代码产生的最后两个阶段的工作：代码发射和代码重定位。图 9.17 展示了这个函数的工作流程。

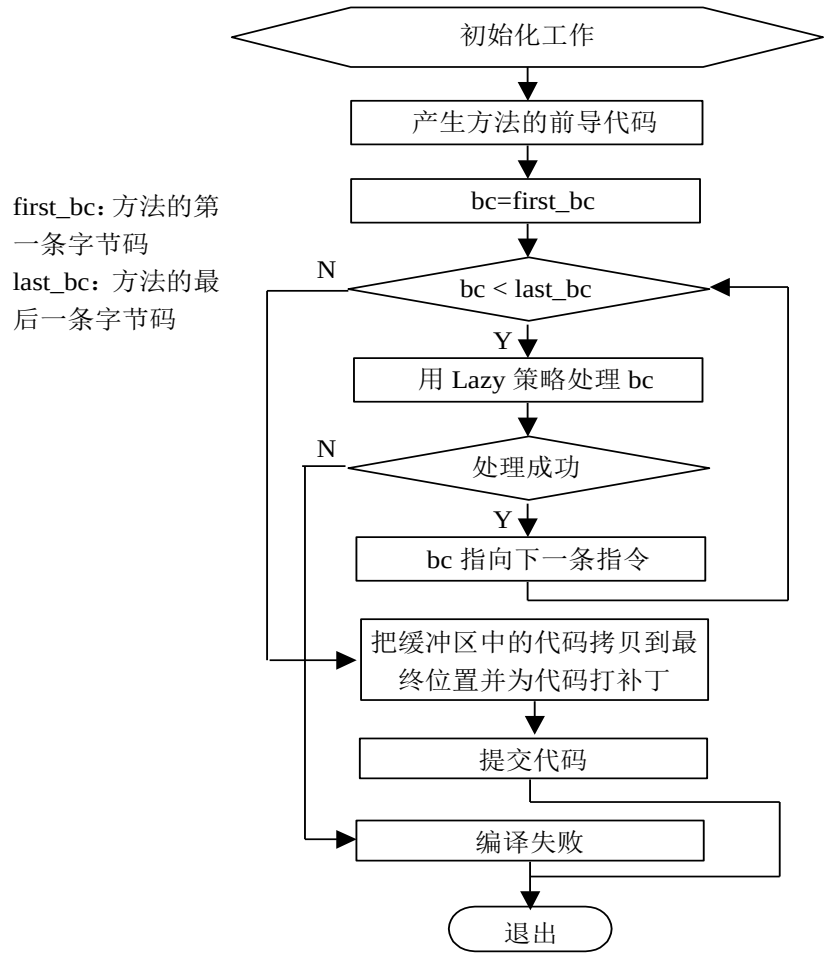


图 9.17　select_code 函数流程图

Code_Emitter 类对象指针、CG_Prepass 类对象指针、Frame 类对象指针、Stack 类对象指针、Jit_Method_Info 类对象指针、Register_Allocator 类对象指针以及类句柄、方法句柄等必要的信息作为参数被传入 select_code 函数。

在 select_code 函数的初始化工作中生成一个 Data_Emitter 对象：ro_data_emitter，分配了一块临时数据空间存放静态变量。ro_data_emitter 是只读数据块，用来存放浮点常数和分支语句的标签。

在开始进行代码选择之前，select_code 函数还生成了三个补丁链表：prof_patch、code_patch_list、table_entry_patch_list。补丁链表是为了记录一些在编译时暂时不能确定的数据或偏移，在 Lazy 代码选择对字节码进行完一遍扫描之后，再将这些补丁填充上去。补丁链表中的每一项记录了这样一些补丁在数据或代码块中的偏移，以及在补丁处应填上的值。三个补丁链表记录了三种不同用处的补丁，它们分别是：prof_patch 记录了 profiling 数据触发重编译时跳转指令的偏移的补丁；code_patch_list 记录了暂时不能确定的调用返回地址和分支跳转的目标地址补丁；table_entry_patch_list 则记录了 switch 开关语句的入口地址补丁。每一个补丁记录是 Patch 类（或其子类）的对象。

做好这些初始化工作之后，函数就开始对一个方法选择机器代码了。

1.　产生方法的前导代码（prolog）

在进入一个方法之后，有一些调用例程是必须执行的，所以首先产生一个调用的前导代码。这部分代码的主要功能是保存栈顶指针、在活动栈帧中开辟方法的局部存储空间、保存被调用者保存寄存器等。图 9.18 给出了前导代码的汇编代码。

```
push    ebp                 ；基于 EBP 的栈帧需要将 EBP 寄存器压栈
mov     ebp，esp            ；将栈顶指针保存到 EBP 寄存器中（同样仅对基于 EBP 的栈帧）
sub     esp，n_locals+n_spill_words ；栈顶指针下移，为局部变量区和 spill 区留出空间
push    ebx                 ；保存被调用者保存寄存器（全局寄存器）
push    ebp                 ；基于 ESP 的栈帧需要这条指令
push    esi
push    edi
mov     gc_tags，init       ；这组指令初始化了栈帧中的附加数据区
< ... monitorenter ... >    ；对于同步方法，将产生调用 VM 运行时支持的代码
```

图 9.18 方法的前导部分汇编代码

比照 9.2.3.4 中的栈帧结构，我们可以很容易明白这段汇编码完成的任务。其中需要说明的是最后两行。倒数第二行的 mov 语句事实上是一组 mov 指令。函数变量是否为引用的信息被组装为若干个字的位矢量， mov 指令的源操作数就是这几个构成位矢量的字，而目的操作数则是栈帧中的附加数据区中的内存单元。最后一行也是一组指令，只有同步方法需要产生这组代码。如果方法是同步方法，需要进入监视器得到对象的锁，以防止其他进程调用这个对象的同步方法。这个任务只有通过调用 VM 运行时支持函数才能达到，VM 的接口函数提供了这各支持函数的地址：

void * getaddress__orp_monitor_enter_naked ( )；

2. 为字节码选择机器代码

前导代码产生之后，select_code 函数就开始对方法的字节码进行一次遍历并选择机器指令。

在对字节码的遍历过程中，需要在基本块的入口对 mimic 栈初始化。在进入每一个基本块时，寄存器分配情况发生了变化，而 mimic 栈中很可能不是空的，为了防止寄存器中的值被覆盖，需要把 mimic 栈中所有操作数都倾倒到栈帧的 spill 区中。

在基本块的入口处，代码选择器产生一组 MOV 指令，把上一个基本块结束时 mimic 栈中的所有操作数存放到 spill 区的相应位置；然后把 mimic 栈的栈顶指针初始化为 0；接下来要做的就是把 mimic 栈初始化到这个基本块开始时应该有的情况。由于预遍历过程已经收集了每一个基本块开始时 Java 操作数栈的深度，所以我们可以知道这一个基本块开始时 mimic 栈中有哪些操作数，将把 spill 区中这个栈深范围内的所有内存单元都压入 mimic 栈中。这样初始化 mimic 栈的一个好处是：保证了从不同路径进入同一个基本块时，mimic 栈的情况是一致的。

当代码选择器遍历到字节码 bc 时，将按照 Lazy 代码选择策略进行处理。

对于 x_load、x_push、pop_x 等字节码，代码选择器并不为它们产生机器代码，只在 mimic 栈上进行进栈出栈的操作，在必要时产生机器代码将 mimic 栈中的操作数倾倒到栈帧中的 spill 区中。

当遍历到运算和 x_store 字节码时，代码选择器为它们选择运算指令，从 mimic 栈中弹出操作数，尝试把它们合并到指令当中去。运算指令的目标操作数依旧压入 mimic 栈中。

对于条件转移或者非条件转移字节码的翻译中需要注意的是如果跳转的目标偏移是正的情况，即是向前跳转的指令。在这种情况下，跳转的目标还没有编译，目标机器代码的偏移暂时不能知道，这时候

将生成一个新的补丁对象加入 code_patch_list 当中，记录下这个需要回填的指令地址。在 select_code 函数中还有一个 map 数组，数组的大小和方法的字节码长度相同。map 数组的一项对应于一条字节码，记录了跳转到与它这条字节码的所有补丁。当为某条字节码生成完机器代码之后，将会查找 map 数组与它对应的项，得到所有以它为目标代码的跳转指令补丁，把机器代码的偏移回填到各个补丁当中去。

3. 访问常数池

有一些字节码需要访问堆中的存储单元，例如字节码 ldc index，index 是当前类的常数池的有效索引，它把常数池中的项压入 Java 操作数栈中。

代码选择器在处理字节码 ldc 时，如果常数池中的项的类型是 int、long、float、double，那么首先通过方法的类句柄获得常数池中的相应常数的值。对于整型常数 int、long，把他们作为立即数压入 mimic 栈中；而对于 float、double 类型的常数，则把它们先存放在 ro_data_emitter 分配的数据块中，然后才把数据块中相应的内存单元压入 mimic 栈中。如果常数池中的项是一个字符串，则需要调用 VM 运行时支持函数进行处理，这个函数的地址由下面的接口函数获得：

void getaddress__orp_instantiate_cp_string_naked ( )；

另外还有一类字节码要访问常数池，那就是 getfield、putfield 等访问方法的某个域成员的字节码。我们以 getfield 为例说明代码选择器是如何访问一个方法的非静态域成员的。

首先，将调用 VM 的编译时接口函数试图解析常数池中的这个域成员，得到域的句柄。

如果解析失败，则产生一段代码调用 VM 运行时支持函数，在运行中执行到域访问时抛出一各链接错误异常。

否则，解析成功得到域成员在类结构中的偏移 offset。从 mimic 栈中弹出类对象的引用，将它放在寄存器 reg 中，产生相对寻址操作数[reg]offset 的 Field_Operand 对象压入 mimic 栈中。

如果域成员是类的静态域，那么在解析成功之后还要判断该类是否已经初始化了。如果它所在的类还没有初始化，那么需要调用 ORP 的接口函数，在运行时对类进行初始化。在得到静态域成员的地址 addr 之后，产生直接寻址操作数[addr]的 Static_Operand 对象压入 mimic 栈中。

4. 方法和接口的调用

方法的字节码中出现 invokevirtual、invokespecial、invokestatic、invokeinterface 时，它将调用另一个方法或者接口。代码选择器为这些调用产生的代码遵循以下的例程：

- 方法的 Jit_Method_Info 结构的域 cnt 加 1，用 cs_info 链表上一个空的 Call_Site_Info 结构来记录这个调用点的信息。
- 在方法调用时有一些寄存器会在调用方法中被覆盖，所以必须把 mimic 栈中有可能被覆盖的操作数倾倒到 spill 区中，这些操作数是：Field、Array、Static、FP、调用者保存寄存器。
- 调用字节码通常跟上方法和接口在常数池中的索引作为操作数，首先要根据索引将常数池中相应的项解析，获得方法句柄。如果解析失败，则 JIT 编译失败。
- 产生将被调用方法所需的参数压栈的代码，并且将每个参数进栈之前的栈顶指针用 Esp_Record 结构数组保存，数组的头指针保存在 Call_Site_Info 结构的 esp_record 域中。第一个参数压栈的指令地址放在 Call_Site_Info 结构的 Call_IP 域中。
- 获得被调用方法的地址，并产生调用代码。对于调用对象方法的 invokevirtual，通过方法句柄得到类对象（实例）方法指针在对象方法表中偏移 offset，然后通过 mimic 栈中类对象的指针获得对象方法表地址放到 eax 寄存器中，产生调用代码：call [eax+offset]；对于调用初始化方法的 invokespecial 和调用静态方法的 invokestatic，通过方法句柄能够直接得到被调用方法地址 addr，产生调用代码：call [addr]；对于接口方法调用 invokeinterface 得到被调用方法地址的步骤比较复杂。首先，通过字节码中的常数池索引得到常数池中接口方法的句柄，从而可以得到接口方法在接口虚方法表中的偏移 offset 和定义接口方法的接口句柄。接着，利用接口句柄和从 mimic 栈中得到实现接口对象的指针，调用 VM 编译时支持函数获得接口的虚方法表指针。支持函数的代码地址由下面这个函数获得：

void * getaddress__orp_get_interface_vtable_old_naked ( )；

返回值放在 eax 寄存器中；最后产生调用语句 call [eax+offset]。

- 将 mimic 栈中的被调用方法的参数弹出栈。
- 将存放被调用方法返回值的操作数压入 mimic 栈中。被调用方法的返回回值如果是整型则在 EAX 和 EDX 中，如果是浮点数则放浮点寄存器栈中。

还有一类特殊的函数调用，那就是上文中多次出现的 VM 运行时支持函数的调用。在产生调用某个 VM 运行时支持函数的代码时，代码选择器遵循一定的例程。在这样一个函数调用点，select_code 函数需要完成以下这些工作：

- 方法的 Jit_Method_Info 结构的域 cnt 加 1，用 cs_info 链表上一个空的 Call_Site_Info 结构来记录这个调用点的信息。
- 产生将被调用函数所需的参数压栈的代码，并且将每个参数进栈之前的栈顶指针（ESP 寄存器的值）用 Esp_Record 结构数组保存，数组的头指针放在 Call_Site_Info 结构的 esp_record 域中。第一个参数压栈的指令地址放在 Call_Site_Info 结构的 Call_IP 域中。
- 产生调用指令 call addr，其中 addr 是被调用函数的机器代码的入口地址。把调用指令地址放在 Call_Site_Info 结构的 precall_IP 域中。由于 call 指令的操作数是被调用函数的入口地址与 call 指令之间的偏移，call 指令的操作数在代码重定位阶段需要重写，所以在 code_patch_list 链表中增加一个 Patch 对象，记录了这条调用指令本身地址以及被调用函数的入口地址。
- 根据被调用函数的其他信息补充 Call_Site_Info 中其它域的值。

方法中实际的函数调用点的数目是以上这两类程序点数目之和，要比预遍历过程估计的 num_call_site 少得多。在 select_code 函数结束对方法字节码的代码选择之后，将会检查预遍历过程收集的调用点引用位矢量数组 gc_site_vectors，将其中确实是实际函数调用点的引用位矢量拷贝到相应的 Call_Site_Info 结构中去。

5. 方法的收尾代码（epilog）

当方法中出现 ireturn、areturn 等返回指令时，就是方法的出口，表示调用的结束。在方法调用的收尾代码同样也要遵循调用例程。图 9.19 给出了方法收尾代码的汇编代码：

```
< ... call MonitorExit ...>      ；同步方法调用 ORP 运行时支持函数退出监视器
mov   ret val to ret reg
；将放回值放入寄存器当中。整型数放在 EAX、EDX 中，浮点数放在浮点寄存器栈中
pop      edi                     ；将被调用者保存寄存器弹出栈
pop      esi
pop      ebp                     ；基于 EBP 栈帧结构不需要这条指令
pop      ebx
mov      esp, ebp                ；基于 EBP 栈帧恢复栈顶指针，首先将 EBP 中的值恢复到 ESP
pop      ebp                     ；寄存器中，然后从栈中弹出 EBP 寄存器
add      esp, frame_size         ；基于 ESP 栈帧恢复栈顶指针，直接将 ESP 寄存器减去栈帧大小
ret      num_arg                 ；返回指令，操作数为方法的参数个数
```

图 9.19 方法调用的收尾代码

在收尾代码的主要工作就是退出活动记录栈帧，将活动返回到调用函数一层，并且将结果返回。

6. 建立方法编译信息

在 JIT 对方法的代码选择完成之后，就知道了关于方法的各种信息：局部变量在栈帧中的偏移、每个函数调用点的参数和返回地址的信息、在某个调用点活动栈中各个空间是否为引用的信息等。这些信息在编译的过程中为 JIT 所知，而在方法运行时为 ORP 的某些支持所用。在 O1 的代码选择开始时，其中一部分信息放在 Jit_Method_Info 结构的参数 method_info 中传入，在编译的过程中各个调用点的信息和其他一些编译信息被补充到 method_info 的 cs_info 链表中。这个结构中的信息对支持异常处理时 Unwind 活动栈帧、垃圾收集时计算根集都有非常重要的作用。但是这些信息的存储量较大而且对于许多 Java 应用小程序来说异常处理和垃圾收集都未必会遇上，所以在编译结束后并不保留 method_info 的信息，而是生成一个简单编译信息记录 Small_Method_Info smi：

```
typedef struct Small_Method_Info {

    Jit_Method_Info *mi;          // 方法第一次编译时，mi = = NULL

    Profile_Rec *prof_rec;        // 记录 Profiling 信息

    unsigned class_initializer[1]; // 记录处理 putstatic/getstatic 指令所需要的类初始化调用

} Small_Method_Info;
```

smi 存放在方法句柄（Method 类对象）O1 相对应的 JIT_Specific_Info 结构的_jit_info_block 中。Jit_Method_Info 结构的产生采用 Lazy 策略，第一次编译时 smi.mi 为空，当需要垃圾收集和 unwind 栈帧的支持时再次调用 l1a_compile_method 函数生成 Jit_Method_Info 结构，将结构的指针放在 smi.mi 中。这里要说明的是 l1a_compile_method 函数：

```
l1a_compile_method（Compile_Handle compilation，Method_Handle meth，
                    JIT_Flags flags，CODE_MI code_mi）；
```

它的最后一个参数是 CODE_MI 枚举类型，有两个值 cm_gen_code 和 cm_gen_method 分别代表两种编译模式。在方法第一次使用 O1 编译器编译时采用 cm_gen_code，只产生 Small_Method_Info 结构的信息，不生成 Jit_Method_Info 结构；当需要更详细的编译信息时，采用 cm_gen_method 模式再次编译生成 Jit_Method_Info 结构的编译信息。

在 cm_gen_method 模式下，select_code 函数的最后将生成一个新的 Jit_Method_Info 结构 new_mi，method_info 的信息被整理后拷贝到 new_mi 中。new_mi 的指针将会被赋给相应的 Small_Method_Info 的 mi 域。

在代码选择完成之后，select_code 还完成了代码发射和重定位的工作，将生成的目标代码拷贝到内存中的最终位置，并且根据各个补丁链表的信息将代码中的暂缺的各种偏移填补上。这些工作完成之后，就知道了每条字节码对应的机器代码所在的内存空间地址。因此，像异常处理（exception handler）这样的信息也要做相应的更新。在编译之前，只知道每个异常处理开始和结束的字节码，因此在方法的异常处理信息中只有字节码信息，在编译之后就得到了异常处理的机器代码范围，这一信息将被更新到方法对应的 Method 类实例相关的域中去。

9.2.4 轻量级优化

在 O1 编译器中由于效率的要求，只进行某些简单而有效的优化。在接下来的一节中，我们将介绍 O1 编译器在进行代码选择时进行的轻量级优化。

9.2.4.1 消除公共子表达式

传统的基于数据流分析的消除公共子表达式的方法在时间和空间上的花费都太大了，不适合 JIT 编译器。ORP 的 O1 编译器开发了一种轻量级的 CSE（消除公共子表达式 Common Subexpression Elimination）算法，它关注于一个基本块范围内的公共子表达式。

O1 的 CSE 算法用 Java 字节码本身来表示几个表达式合并。考虑这样的表达式：x + y，假设 x 和 y 分别是局部变量 1 和 2。那么这个表达式将会被翻译成下面的字节码序列：【iload_1，iload_2，iadd】。因为这三条字节码指令的 16 进制表示分别是 0x1b、0x1c、0x60，所以 16 进制字节码表示 0x1b1c60 表示了表达式“x + y”。如果字节码指令流中再次出现 0x1b1c60 子序列，我们可以认为 x＋y 表达式再次出现。

这样，表达式可以用一对数来表示：<offset，length>，这对数被称作表达式标签，其中 offset 表示表达式字节码序列开始时的偏移（相对方法的第一条字节码指令），length 表示表达式字节码序列的长度。为了检查两个长度相同的表达式 exp1<offset1，n>和 exp2<offset2，n>是否匹配，即表示同样的语法表达式，只需要比较在 offset1 处开始的和 offset2 处开始的 n 个字节的字节码序列是否相同。在实现中，一个字节码子序列的长度最多是 2^{16}，所以一个表达式标签只需一个字就能表示，offset 和 length 各占 16 位。

```
class CSE_Exp {
    union {
        struct {
            unsigned short _bc_index;          // 16 位 offset
            unsigned short _len;               // 16 位 length
        } cse_tag;                             // 表达式标签
        unsigned _value;                       // 表达式的值
    };
    public:
    … …
    // 包含了为表达式设置、修改、查询、比较标签和值的函数。
}
```

上面这段源代码是 O1 的代码选择器对每个可能的表达式的类表示。为了检测方法的字节码指令流中的公共子表达式，代码选择器跟踪每个局部寄存器 R 中存放的表达式。每一个局部寄存器都用它所含的表达式的标签来注释。在为字节码 B 选择代码之前，代码选择器往前看若干个字节，看看从 B 开始的若干个字节码是否与局部寄存器中已有的表达式是否匹配。如果恰好语局部寄存器 R 中存放的表达式匹配，那么代码选择器就将寄存器 R 压入 mimic 栈中，作为后续程序的操作数，并且跳过匹配到的公共子表达式处理后面的字节码。代码选择器在匹配当前字节码流中可能有的公共子表达式时，按照所含表达式长度从大到小的顺序来检查每个局部寄存器，这样就能够保证匹配到最大长度的公共子表达式。如果匹配不成功，代码选择器才按一般的例程来为字节码 B 选择代码。如果在选择过程中改动了某个寄存器 R 的值，那么这个寄存器的表达式标签就必须更新。在每一个基本块的开始处，所有的局部寄存器内的表达式标签都会被初始化，这是因为在不同基本块内局部寄存器的分配的分配情况不尽相同。

接下来我们看一下代码选择器是怎样利用 mimic 栈来跟踪局部寄存器中的表达式。mimic 栈中的每个栈空间都有的 CSE_Exp 对象与之对应，这个 CSE_Exp 对象表示了此时栈空间中的操作数是由哪一组字节码序列计算出来的。在代码选择器为字节码 B 选择代码时，很可能先从 mimic 栈中弹出若干个操作数，假设弹出两个操作数，它们对应的 CSE_Exp 分别是 Exp1 和 Exp2。每个操作数被弹出后，它们对应

的 CSE_Exp 对象被暂时保存起来。如果这条字节码将产生一个结果，这个结果将被压入 mimic 栈中，代码选择器将试图把字节码 B 和 Exp1 和 Exp2 所表示的字节码序列合并起来。如果 Exp1 和 Exp2 所表示的字节码序列和 B 的连续的，那么合并成功，代码选择器将为这段连续的字节码序列建立 CSE_Exp 对象，用它来对应 mimic 栈中的结果操作数；如果合并失败，那么代码选择器将为字节码 B 建立 CSE_Exp 对象，用它来对应结果操作数。这样 mimic 栈中的每个操作数是由哪些字节码序列得到的，都被这些 CSE_Exp 对象记录下来。如果 mimic 栈上的操作数被弹出到某个局部寄存器中，那么这个用字节码序列表示的表达式信息也被拷贝到局部变量的表达式标签中，寄存器原先保存的值的表达式信息将被覆盖。为了保证编译时间是线性的，O1 的代码选择器把局部寄存器保存的公共子表达式长度限定在 16 个字节以内。

在下面这两种情况下，一个局部寄存器 R 中的表达式 E 将会被清除：

- 寄存器 R 的值被某些指令所修改。如果寄存器 R 是一个调用者保存的寄存器，那么在一个调用点它所包含的表达式信息将被清除。在这种情况下 R 的表达式标签将会指示其中不含任何值。当寄存器管理把 R 的内容倾倒入内存、回收后重新分配时，R 的表达式信息也被清除。这也是我们在局部寄存器分配中为什么总是把最早分配出去的寄存器 R 回收的原因，最近被计算的表达式重用的机会比较大，而这样的回收策略使得公共子表达式被匹配到的机会变大。在这种情况下，R 的表达式标签将会被换成它包含的新表达式的标签（或者不含任何值，如果 R 只被用来存放临时值）。
- E 中所包含的局部变量被一些赋值语句和方法调用所改变。表达式 E 中可能装载了某个局部变量、数组元素、对象的域成员，这些元素在某些方法调用和赋值语句中很可能被修改，这时局部寄存器中相应的值就已经过时，不能被重用，因此表达式 E 的信息将会被清除。在一个方法调用点，代码选择器将清除包含载入数组元素或者对象域成员的表达式信息；在一个赋值语句改变某个局部变量、数组元素、对象域成员时，代码选择器将清除所有装载了这个元素的表达式的信息。

代码选择器为每一个局部寄存器分配了一个位矢量，用来存放寄存器中表达式的清除集合（kill set），即表达式中所有可能导致它被清除的元素。每一个局部变量都有唯一的索引，而每一个对象域成员在常数池中也有唯一的索引，因此这允许我们用位矢量来标志第几个局部变量或是常数池第几个元素属于表达式的清除集合。在 O1 的实现中，为了节省存储空间，每一个局部寄存器的清除集合都用 256 的位矢量来标识。如果索引为 index 的局部变量或者对象域成员在表达式的清除集合中，那么位矢量的 index 位上就被置为 1。如果索引为 index 的局部变量或者常数池索引的对象被重写，那么位矢量的 index 位上为 1 的那些局部寄存器中的表达式都将被清除。如果一个表达式某个元素的索引超过了 256，那么代码选择器就清除这个公共表达式，放弃对这个公共子表达式使用的机会。

代码选择器用一种更保守的方法来处理数组元素引起表达式被清除的情况，因为数组元素的假名分析非常复杂，这种花费是 JIT 编译器无法负担的。O1 编译器利用了 Java 的一个特性：在不同类型的数组元素之间不可能互为假名。每一个寄存器 R 也有一个固定长度的位矢量来记录它存放的表达式包含了哪些类型的数组元素。当代码选择器面对一条字节码要对类型 T 的数组元素赋值的时候，就清除所有包含了这种类型数组元素的所有表达式。

在对一个寄存器更新时，要计算将要放入寄存器的表达式的清除集合。表达式的字节码序列将被逐条遍历，发现其中装载局部变量、数组元素、对象域成员的字节码指令（如 iload_0、iaload、getfield），将其中的索引和数组元素类型记录下来。在遇到赋值语句（如 istore_0、iastore、putfield 等）和函数调用语句时，检查每个寄存器中表达式的清除集合，将过时的表达式清除。

传统的 CSE 方法产生一个临时的空间来存放公共子表达式的值，可能会增加寄存器分配的压力，从而引起倾倒代码的增加。而 O1 编译器的 CSE 方法不会引起这样的问题，因为寄存器 R 如果被代码选择器重用，那么其中表达式的值将会清除，而不再保存。但是这种 CSE 方法也有很大的局限，首先，像“x+y”和“y+x”这两个表达式，它会认为是不同的表达式序列；其次，它不能检测出那些语法形式不同但值相同的表达式，例如：“x = w；x + y”和“w + y”；第三，它只能表示那些在字节码指令流中连续的表

达式。

9.2.4.2 减少边界检查

Java 语言规范要求每个数组元素访问之前都要进行运行时检查，看元素的下标是否越界。JIT 编译器如果能够证明元素的下标是在正确的范围之内，那么就能够减少这部分检查的代码，或者证明元素的下标越界，从而不需检查就抛出数组越界的异常。

O1 编译器用了一个简单的机制来减少那些下标是立即数的数组元素访问的越界检查。对于那些包含了数组 A 引用的 mimic 栈空间和变量，代码选择器跟踪数组 A 不需做越界检查的最大常数下标。当一个常数下标的越界检查代码产生时，或者一个大小为常数数组产生时，这个信息会被更新。例如，如果一个代码选择器已经产生了数组访问 A[7]的边界检查代码，那么当后续代码中出现了数组访问 A[5]，就无需对它产生越界检查的代码了。而且当一个数组由 newarray 字节码创建时，编译器将会记录数组创建时的大小，用它减少边界检查代码。

在 O1 编译器的实现中，用类 Bounds_Checking 来记录局部变量和局部寄存器中包含的数组边界信息。

```
class Bounds_Checking {

    int _var_base[MAX_BOUNDS_ENTRY];   // 包含数组引用（基址）的局部变量索引数组

    int _bound[MAX_BOUNDS_ENTRY];      // 相应数组不用作边界检查的最大常数下标

    int _scratch_bound[n_reg];         // 放在局部寄存器中的数组引用的边界信息

  public：
    … …

    //对边界信息进行初始化、更新、查询的公共方法

}
```

在为一个方法选择代码时，代码选择器生成 Bounds_Checking 类的一个对象 bounds。为了节省空间，MAX_BOUNDS_ENTRY 定义为 16，如果包含数组引用的局部变量的索引是 var_no，那么取 var_no 的低 16 位得到数组索引，把 var_no 和相应的边界信息放在数组中：

```
idx = var_no & ( MAX_BOUNDS_ENTRY-1 )；
bounds._var_base[idx] = var_no；
bounds._bound[idx] = bnd；  // bnd 为该局部变量所放的数组不用作边界检查的最大常数下标
```

所以，索引低 16 位相同的局部变量可能要竞争 bounds 的同一个数组空间。

当代码选择器遇到 xaload、xastore 这样的数组访问字节码，且操作数中的数组下标是一个立即数时，遵循图 9.20 中的流程。

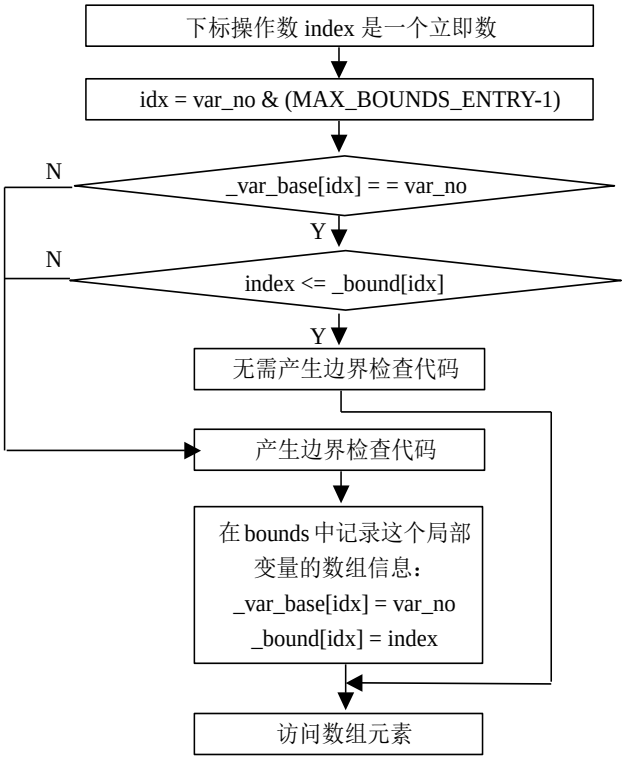


图 9.20 减少边界检查策略下访问数组元素的流程

当字节码流中出现了 newarray，代码选择器需要产生代码来调用 VM 的运行时支持函数来生成这个新的数组。VM 运行时支持函数成功生成数组返回时，新数组的引用放在寄存器 EAX 中。如果 newarray 指令中新数组的大小 size 是一个立即数，代码选择器会将新数组的大小记录在 bounds._scratch_bound[eax_reg]中，并且在 EAX 的值被存储到局部变量中去的时候，仍然能够跟踪这个边界。这样，如果对这个数组元素的访问下标小于等于这个边界，那么就不要产生边界检查代码；否则就产生代码抛出数组越界异常的代码。

这个减少数组边界检查的算法有两个局限：首先，它只局限在一个基本块内，当进入一个基本块时，bounds 的所有元素都初始化为-1；其次，只对那些下标为常数（立即数）的数组访问起作用。

9.2.4.3 其他简单的优化技巧

O1 编译器的代码选择器在选择代码时还采取了一些简单优化措施。这些优化措施的思想和实现都非常简单，以至于我们称之为技巧而不叫算法。下面介绍其中的两个：

- 我们知道不同机器指令所占用的时钟周期是不相同的。乘法、触发、求余指令运算较复杂，占用时钟周期多，而逻辑运算指令占用的时钟周期就要短得多。在代码选择器选择运算指令时，如果运算的操作数是常数或者立即数，它会尽量地降低运算的强度。例如，对于一个乘法指令，如果它的一个乘数是 2 的 n 次幂，那么代码选择器将会把乘法指令转换称逻辑左移 n 位的指令。
- O1 的代码选择器还通过跟踪装载到寄存器里的值来减少 load-after-store 冗余。所谓的 load-after-store 冗余指的是在把一个局部变量存储到内存中去之后又马上装载到栈中来，事实上如果这个局部变量的值仍然存在某个寄存器当中，而在装载和存储之间局部变量的值没有被改变，那么可以直接使用寄存器中的值，而无需从内存中装载。当一个局部变量被存放到局部寄存器中时，代码选择器记录了这个局部变量的索引号。当遇到将这个局部变量 load 的字节码时，它会去检查每个局部寄存器是否标记这个局部变量的索引号。如果存在着这个寄存器，那么代码选择器就把这个寄存器压入 mimic 栈中，而不是把 Mem_Var_Operand 的内存操作数压入

mimic 栈。

O1 编译器中的这些轻量级优化策略虽然都比较简单存在着比较多的局限，但是正因为其算法简单，在时间和空间上的花费较少，而且通常能取得不错的效果，是适合 JIT 快速代码生成的优化策略。

9.3 运行时支持

JIT 编译器需要为 Java 虚拟机提供运行时支持，使 Java 虚拟机能够完成一些依赖于 JIT 编译器的工作，例如：unwind 活动栈帧、为垃圾收集计算根集、查找测试断点的机器代码偏移。这些支持通过 JIT 编译器和 ORP 之间的运行时接口被 ORP 调用。在这一节我们将介绍 O1 编译器是如何提供这些运行时支持的。

9.3.1 回退活动栈帧

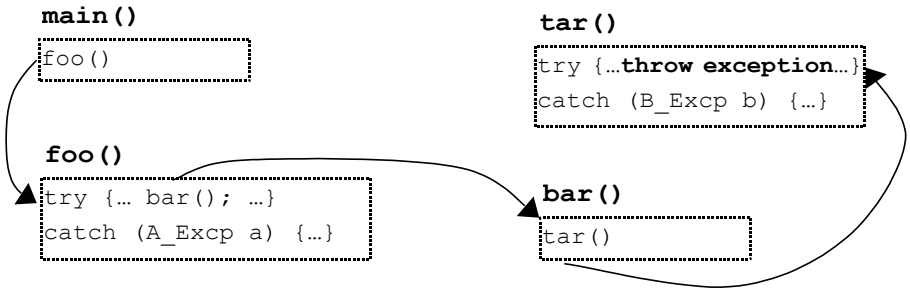


图 9.21 调用链例子

方法的调用序列和环境总是记录在这个线程的活动记录栈中，每一个方法在活动记录栈中占有一个帧。回退活动栈帧就是在活动记录栈中，从正在执行的方法，也就是活动记录栈中处在栈顶的一个帧开始，沿活动记录栈顶向底回溯，恢复一个栈帧的环境。一个栈帧的环境包括栈信息，如栈顶指针、帧结构等，全局寄存器的分配情况等。回退活动栈帧通常发生在一个异常抛出时，如果抛出异常的方法没有对这个异常的处理，那么虚拟机需要通过回退活动栈帧寻找在调用链上哪个方法具有对这个异常的处理。而虚拟机是不知道一个线程活动记录栈的结构的，这只有 JIT 编译器才能够提供。我们通过一个例子来说明 O1JIT 编译器是如何提供回退活动栈帧的支持的。图 9.21 给出了一条方法的调用链。

在这个例子中，main 方法调用 foo 方法，foo 方法调用 bar 方法，而 tar 方法又为 bar 方法所调用。线程执行到 tar 方法时，抛出一个异常。我们可以看到在 tar 方法中能够捕捉类型为 B_Excp 的异常并进行处理，如果 tar 中抛出的异常是 B_Excp 的对象，那么执行 catch 中的代码。如果这个异常的类型不是 B_Excp，假设是异常类 A_Excp 的对象，那么 ORP 要沿着调用链寻找能够捕捉 A_Excp 异常的调用者。

在异常发生时活动栈帧时，栈顶是 tar 方法的活动栈帧（tar 方法的栈帧结构是基于 EBP 的，而 bar 方法的是基于 ESP 的）。当 ORP 在 tar 方法的信息中找不到异常类 A_Excp 的处理时，它就要沿调用链向前寻找 tar 的调用者 bar 是否有 A_Excp 的处理。于是它调用 JIT 提供的运行时接口函数 JIT_unwind_stack_frame 向前回溯一个栈帧，调用这个函数，ORP 需要提供三个参数：

Method_Handle　meth，
Frame_Context　*context，
Boolean　is_first

第一个参数是方法句柄，通过这个参数 JIT 能够得到方法 Jit_Method_Info 结构的编译信息，有了这个参数 JIT 能够得到方法 tar 的栈帧类 Frame 对象（类声明见图 9.14），此时活动栈中情况如图 9.22（a）中所示；第二个参数是栈帧的上下文，它既是 ORP 向 JIT 提供的上下文参数，又是 JIT 向 ORP 返回回退栈帧结果的途径；最后一个参数指示了将要回退的这个栈帧是不是栈顶的第一个帧。

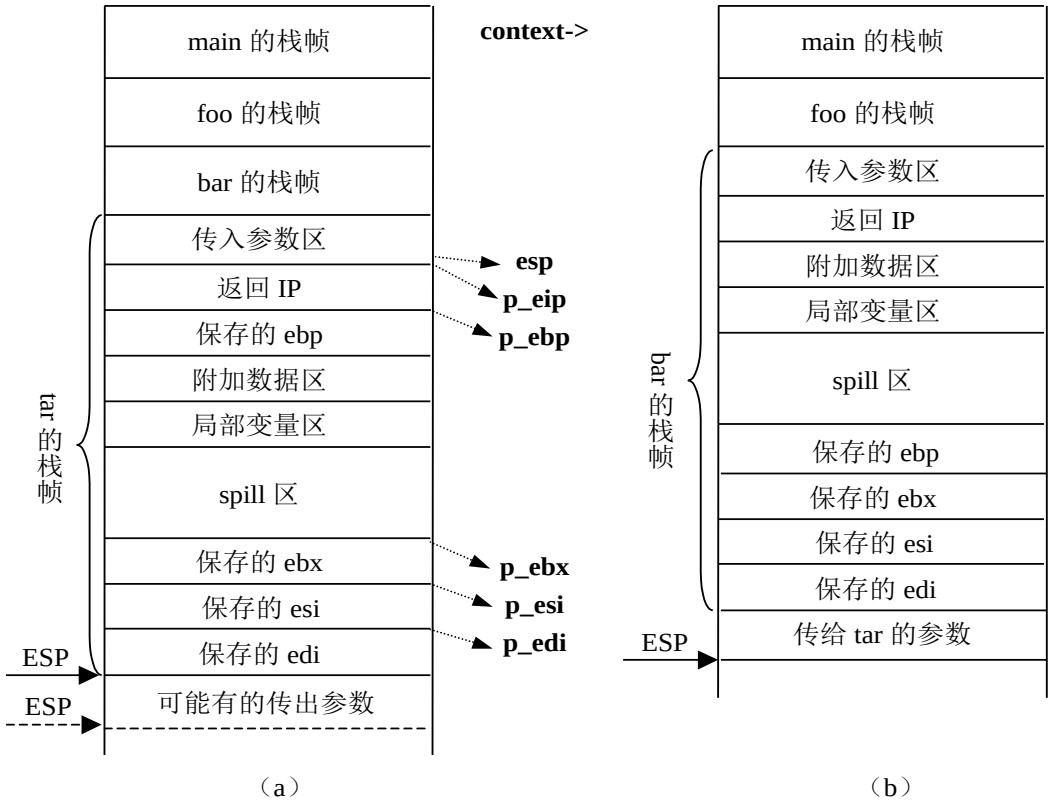


图 9.22 Unwind 方法 tar 的栈帧

上下文参数 context 存储在这样一个结构中：

```
struct Frame_Context {
    uint32 esp;          // 栈顶寄存器 esp 的值
    uint32 *p_ebp;       // 存放寄存器 ebp 的值的栈空间地址
    uint32 *p_eip;       // 存放寄存器 eip 的值（返回 ip）的地址
    uint32 *p_edi;       // 存放寄存器 edi 的值的栈空间地址
    uint32 *p_esi;       // 存放寄存器 esi 的值的栈空间地址
    uint32 *p_ebx;       // 存放寄存器 ebx 的值的栈空间地址
    uint32 ljf;
    uint32 *p_eax;       // 存放寄存器 eax 的值的栈空间地址，在支持垃圾收集时用到
    uint32 *p_ecx;       // 存放寄存器 ecx 的值的栈空间地址
    uint32 *p_edx;       // 存放寄存器 edx 的值的栈空间地址
```

Boolean is_first;　　// 栈帧是不是活动栈上的第一个帧

};

由于 tar 方法可能在一个调用点发生了异常，所以需要判断这时候它调用的函数参数是否已经压栈。*(context->p_eip)为发生异常时指令寄存器中的值，我们可以查找方法编译信息确定这条指令是否在 tar 方法的某个调用点上，是否已经压入传出参数，以判断出正确的栈顶位置。Tar 的栈帧结构是基于 ebp 的，*(context->p_ebp)给出了基准寄存器 ebp 所指的位置在内存实际地址。

tar 在被 bar 调用时，把全局寄存器的值压入栈中，我们要恢复 bar 的运行时环境，需要把存放全局寄存器的栈空间地址放入 context 的相应域中，返回给 ORP。这时候就可以回退方法 tar 的栈帧，esp 寄存器的值将恢复到调用 tar 之前的情况，恢复之后活动栈的情况如图 9.22（b）所示（传入参数暂时不弹出）。存放 ebp、返回地址的栈空间的地址，以及栈顶寄存器 esp 的值被更新到 context 的相应域中，具体寄存器的更新将由 ORP 根据 context 中的返回信息完成。

我们知道在方法 bar 中仍然没有对异常类 A_excp 的处理，所以还需要继续回退栈帧。对 bar 栈帧的回退将重复上面的动作。由于 bar 方法不是回退的第一个方法，所以它必然是发生在调用点上，栈上有传给被调用函数的参数，而*(context->p_eip)中放的是调用的返回地址。

当回退到 foo 的栈帧时，ORP 发现 foo 方法能够处理 A_excp 的异常，就不再继续做回退。运行栈上的情况就会恢复到它调用 bar 时的情况，将执行 foo 的异常处理的代码。

9.3.2 对垃圾收集的支持

9.3.2.1 O1JIT 的 GC 安全点

ORP 在进行垃圾收集时，需要收集各个线程的活动记录栈上和寄存器中的根集，完成这一工作一样要依赖 JIT 编译器。因为只有 JIT 编译器才能精确的为局部变量、栈空间和寄存器中的引用定位。

在 ORP 的 GC 机制中，限制垃圾收集只发生某些编译器定义的程序点上，这些程序点被称为是 **GC 安全点**。GC 安全点简化了编译器的设计，因为编译器只需要记录这些程序点上的根集情况。ORP 的 O1 编译器把调用点作为 GC 安全点。我们知道 O1 编译的 Lazy 代码选择器在遇到方法的调用点会把 mimic 栈中可能被被调用函数覆盖的操作数倾倒入栈帧的 spill 区中，局部寄存器中的操作数不会再使用，而全局变量将会被被调用函数保存并重新分配。寄存器中的值在调用点上即将失效，所以在这些程序点上我们无需考虑寄存器中的活引用情况。在调用点上进行垃圾收集，JIT 要收集的活引用根集只在栈空间中，把调用点作为 GC 安全点将使 JIT 收集根集的函数更加简单。从 9.2.3.5 节中，我们可以看到在 O1 编译器的处理过程中，只有调用点的 mimic 栈上各个元素的引用类型被放在 Call_Site_Info 结构中保存下来。

在编译方法时得到的各个 GC 安全点上的根集情况叫做 **GC 映射**（GC map），它包括了全局指针和活动栈帧上的指针所指的所有对象。在 O1 编译器中活动栈帧上的这部分 GC 映射就表现为各个调用点的 Call_Site_Info 信息中的 type_vector 位矢量。这部分信息是相当庞大的，而当 GC 发生时只有很少数目的方法正在激活状态，而 GC 映射对于其它方法来说完全是多余的。ORP 对 GC 映射采用了 Lazy 的策略，只有当 GC 发生时才为涉及到的方法产生 GC 映射。在方法第一次编译时采用 cm_gen_code 模式，只产生 Small_Method_Info 结构的信息，不生成包含 GC 映射信息的 Jit_Method_Info 结构；当需要 GC 映射时，用 cm_gen_method 模式重新编译方法生成 GC 映射。由于 GC 映射的 Lazy 策略需要对方法重新编译，所以这个策略只适用于快速而花费较小的 O1 编译器。

9.3.2.2 收集栈帧中的根集

JIT 对 ORP 提供一个运行时函数 JIT_get_root_set_from_stack_frame 支持垃圾收集工作，ORP 调用这个函数需要提供以下这些参数：

```
JIT_get_root_set_from_stack_frame (
    Method_Handle     method,        // 方法的句柄，通过它能够得到 Jit_Method_Info 编译信息
    GC_Enumeration_Handle     enum_handle，// GC 句柄，通过它将收集到的根集返回给垃圾收集器
    Frame_Context     *context,      // 栈帧上下文
    Boolean           is_first       // 栈帧是不是该线程活动记录栈上栈顶的第一个帧
)
```

这个函数的功能是收集在 GC 安全点线程的活动记录栈其中一个栈帧所有为引用的栈空间。我们一样以图 9.21 中的这个线程为例，说明 O1 编译器是如何收集这个线程中的根集的。假设垃圾收集点是在 tar 方法的一个调用点上，ORP 首先收集 tar 方法栈帧中的活引用，传入的参数 context 是 tar 栈帧的上下文：

- context->esp 为这个调用点的栈顶指针的位置
- *（context->p_eip）为这个调用点的指令寄存器的值，也就是 call 指令代码的位置

有了 method 参数可以得到 Jit_Method_Info 结构的编译信息 mi（结构声明见图 9.15），从而生成 tar 的栈帧类 Frame 对象，它提供了计算栈帧中每一个栈元素（输入输出参数、局部变量、spill 区元素等）实际地址的方法。在代码选择函数 select_code 的最后把方法的所有调用点信息链表 cs_info 按 call 指令的偏移大小排序，所以我们能够利用*（context->p_eip）迅速的查找到这个调用点的信息 csi（Call_Site_Info 结构见图 9.16）。

- csi->type_vector 位矢量记录了这个调用点所有 Java 操作数的类型，根据这个信息，我们能够知道与之对应的 spill 区上每一个栈空间是否为引用。
- csi->outarg_bv 位矢量记录了这个调用点 tar 输出给被调用函数的参数是否为引用的信息。
- 栈帧中的附加数据区存放了方法的局部变量是否为操作数的位矢量。通常我们能够通过字节码指令来判断把什么样类型的操作数存放到局部变量中，例如 astore_1 就把一个引用放到索引号为 1 的局部变量中，而 istore_1 则把整型数放到这个局部变量中。但是有一个例外就是是异常处理 finally 模块的第一条字节码 astore 指令，它把 finally 模块的返回地址存放到局部变量中，所以这个 astoe 并没有使局部变量的类型变为引用，在垃圾收集时不需要考虑这个局部变量。在代码选择的过程中，遇到对局部变量进行写操作的指令，代码选择器会产生代码及时更新附加数据区中相应的一位。由于在预遍历的过程中收集了所有 finally 块的字节码范围（9.2.1.2），所以代码选择器碰到 astore 语句时，会先判断它是否属于一个 finally 处理块，再决定是否需要产生代码更新附加数据区的位矢量。
- 如果代码选择器为某一个局部变量分配了一个全局寄存器，那么在它的调用函数中将会在活动栈中保存这个全局寄存器。如果局部变量是引用类型，那么保存与它对应的全局变量的栈空间也应该是引用类型，一样要计算到根集中。例如，tar 方法保存了 bar 中全局寄存器的值，在这种情况下，tar 的栈帧中保存全局变量的几个栈空间的地址被保存到 context->p_ebp、context->p_ebx、context->p_edi、context->p_esi 这几个域中返回给 ORP。当 ORP 计算 bar 中的根集时将把 context 作为参数传入，如果检查到 bar 中某个局部变量是引用且某个全局寄存器被分配给这个局部变量，那么 context 相应域中的栈地址将被作为活引用传给 GC。

这样栈帧中各个区的活引用情况都能被检查出来。如果某一个栈空间被确定为引用类型，那么将调用 ORP 的接口函数 orp_enumerate_root_reference，把这个栈空间的地址传给 GC，由 GC 记录下来。

在 tar 栈帧的根集计算完毕之后，将把 bar 调用 tar 时的栈顶寄存器 ESP 的值和存放返回地址的栈空间位置分别保存在 context->esp 和 context->p_eip 中返回给 ORP，以便 ORP 沿调用链继续调用 JIT_get_root_set_from_stack_frame 函数计算 bar 的栈帧的根集。调用链上每个栈帧都计算完毕之后，这个线程的根集计算工作就完成了。

9.3.2.3 提供写栅栏支持

在垃圾收集时由于某些算法的需要，需要为堆中某些对象的引用域提供写保护，对这些域进行修改时需要作一些额外的操作，也就是所谓的写栅栏。JIT 在翻译字节码的时候能够知道在对象中是否含有其他对象的指针。当 JIT 编译器遇到字节码指令 putfield 时，也就是对对象的某个域写的时候，将会判断这个域是不是引用类型（某个类的对象或是数组），如果是则需要产生代码调用 ORP 核心 VM 的运行时支持函数。这个支持函数的地址通过下面这个函数得到

void * getaddress__gc_write_barrier_fastcall（）；

这个 VM 运行时支持函数调用了 GC 模块提供的接口函数 gc_write_barrier 来为完成相应算法在修改引用域时的额外操作。

9.3.3 对 Debug 调试的支持

为了支持对 Java 程序的调试，虚拟机需要提供 JVMDI 接口（Java Virtual Machine Debugger Interface），它为监测应用程序在 Java 虚拟机中执行状态提供了途径。虚拟机提供的 JVMDI 接口也需要 JIT 编译器的支持，因为只有 JIT 编译器才知道字节码的断点相对应于哪条机器代码，才知道变量在某个程序点的值。ORP 是在 ORP 自带的两个 JIT 编译器中，只有 O1 支持 Debug 的支持，O3 优化编译器对代码的优化程度较大，目标代码指令序列的安排已经和字节码有了很大的不同，所以很难做到字节码和目标代码程序点的一一对应。

O1 编译器为 ORP 提供了 5 个接口函数支持 Debug 调试：

- unsigned l1a_num_breakpoints（Method_Handle method，uint32 eip）；
- void l1a_get_breakpoints（Method_Handle method，uint32 *bp，Frame_Context *context）；

这两个函数的功能是获得方法中调用点的数目和地址。在知道了方法的句柄之后，很容易得到结构为 Jit_Method_Info 的编译信息，其中的 cs_info 域包含了所有调用点的信息。第二个函数把方法每个调用点 call 指令的地址放入参数 bp 数组中返回给 ORP。

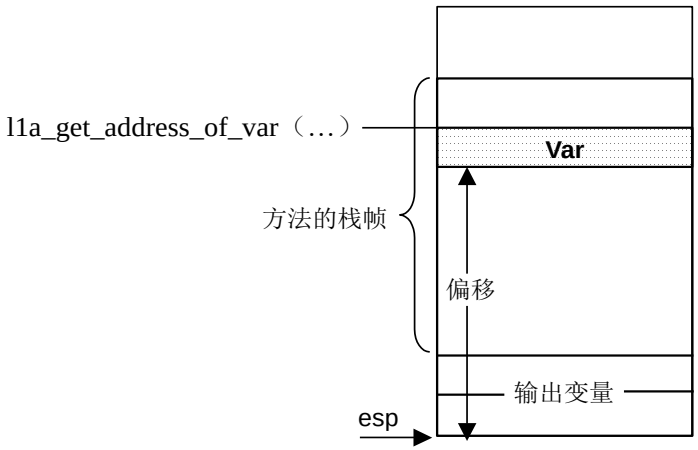


图 9.23 获得变量 Var 的实际地址

- void *l1a_get_address_of_var（Frame_Context *context，Boolean is_first，unsigned var_no）；
- void * l1a_get_address_of_this（Method_Handle meth，Frame_Context *context，Boolean is_first）；

O1JIT 通过这两个函数告诉 ORP 方法的每个局部变量运行时的地址。有了方法句柄之后，能够得到 Jit_Method_Info 的编译信息，从而能够重建相应与方法的栈帧类 Frame 对象。ORP 通过参数 context 传入了栈顶指针的实际位置，通过 Frame 对象计算偏移的函数得到索引为 var_no 的局部变量的与栈顶指针的偏移，就能够得到该局部变量在运行时的实际地址。对象指针 this 是一个特殊的局部变量，在把 Java

应用程序翻译成字节码时通常把它作为方法的索引为 0 的局部变量。

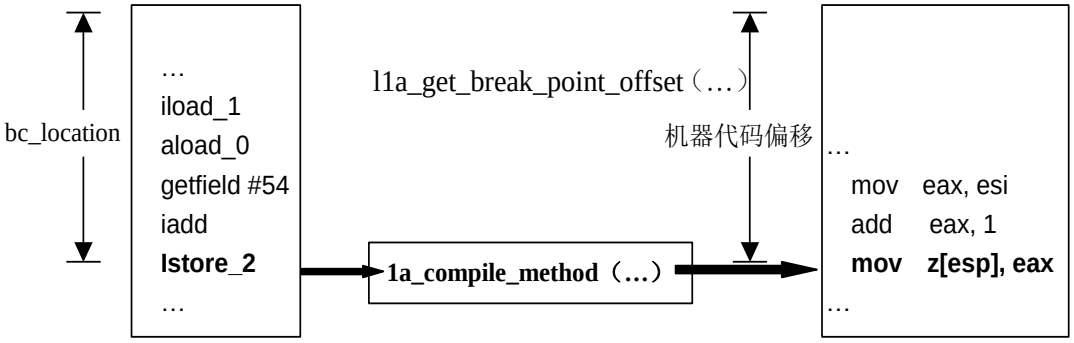


图 9.24 获得断点的偏移

- int32 l1a_get_break_point_offset（Compile_Handle compilation，Method_Handle meth，JIT_Flags flags，unsigned bc_location）；

在进行 Debug 调试时往往需要在 Java 源代码中设置断点。源码中的断点的位置对应字节码中的哪个程序点有 Javac 编译器提供，而字节码到机器代码的位置的对应只有 JIT 编译器能够知道。这个函数由 ORP 调用，参数 bc_location 为断点在字节码指令流中的偏移。函数将采用 cm_gen_method 的模式调用 O1 编译函数 1a_compile_method 对方法字节码进行重新编译，不同的地方在于：在对字节码逐条选择代码的时候，一旦发现当前字节码的偏移与 bc_location 相同，就记下将要产生的这条指令在临时代码缓冲区中的偏移，停止编译，将这个偏移返回给 ORP。ORP 根据这个偏移能够在方法实际的机器代码块中找到这个断点对应的机器指令地址。

虚拟机通过事件消息（Events）来通知 JVMDI 的用户（程序员调试员或者其他的编程工具）他们感兴趣的事件发生了。而 JVMDI 用户则通过对这些事件进行反应的函数来观察和控制应用的执行。有一些事件是虚拟机能够监测到的，而有一些事件需要应用程序自己来发送消息，例如这两个事件，它们分别通知应用进入和跳出一个方法。

JVMDI_EVENT_METHOD_ENTRY

JVMDI_EVENT_METHOD_EXIT

当应用程序在用户 Debug 调试的模式下进行编译时，O1 编译器需要在代码中插入一些支持 JVMDI 事件消息的发送。当 O1 编译器在产生完一个方法机器代码的前导部分后，如果发现允许进入方法事件被传送，那么需要插入一段代码调用 ORP 的接口函数，发送 JVMDI_EVENT_METHOD_ENTRY 事件消息。同样 O1 编译器在产生一个方法机器代码的收尾代码时，如果发现允许跳出方法事件消息被发送，那么在产生 ret 语句之前，也必须产生代码调用 ORP 的接口函数，发送 JVMDI_EVENT_METHOD_EXIT 事件消息。这样在代码执行到进入方法或者跳出方法时，将调用 ORP 的接口函数把相应的事件消息发送出去，ORP 根据相应的消息处理函数为 JVMDI 用户提供服务。

第10章 优化编译器

O1 JIT 将字节码直接翻译成机器码，并对程序进行一些轻量级的优化。为了提高一些执行频率较高的程序的执行效率，我们使用 O3 JIT 对其编译。这也是 1.0.9 版本的 ORP 中缺省使用的即时编译器。

O3 JIT 将字节码转换成中间表示，并应用了更多的优化策略。我们将首先简要介绍优化编译的流程以及它使用的主要优化方法。从 10.2 节开始我们会详细介绍整个优化编译的流程，以及一些特定于 Java 虚拟机的静态优化和动态优化策略。

10.1 优化编译器的结构和优化策略

10.1.1 优化编译器的结构

优化编译器采用传统的编译方法，它的基本结构如图10.1所示。图的左边可以看作优化编译器概念上的前端，右边则可以看作编译器的后端。前端建立中间表示（IR）并对IR做一些全局优化；后端对IR作进一步的降低（lower）并最终映射为机器指令。

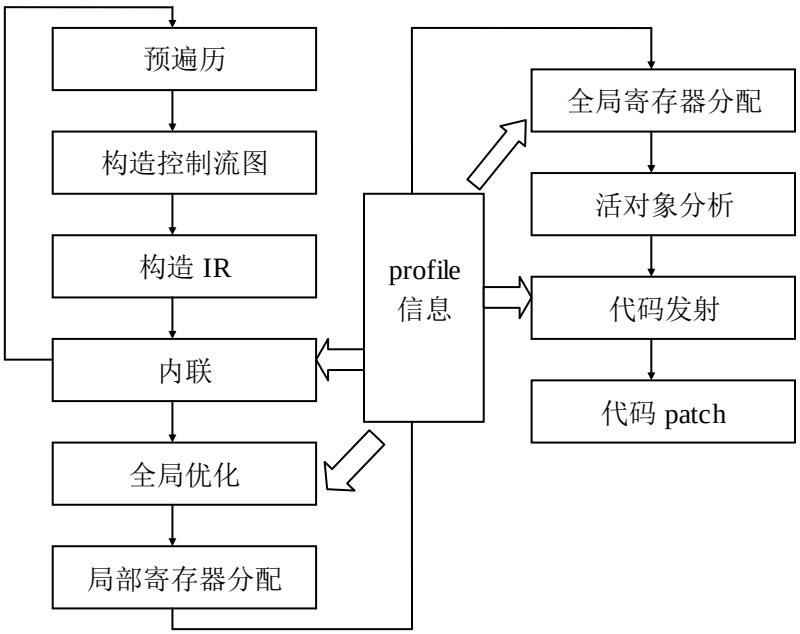


图 10.1 优化编译器结构

图中的Profile信息用来指导优化决定，比如：内联策略，对哪些方法应用重量级的优化，代码发射中的代码布局。我们先简单介绍一下各个阶段完成的工作。

- 预遍历阶段：O3 JIT中的预遍历和O1 JIT中类似，它遍历字节码划分基本块边界。
- 控制流图（CFG）构造阶段：创建流图节点，连接节点构造流图。
- IR构造阶段：为每个基本块构造IR的元组指令序列，并在构造IR时完成跨越扩展基本块（extended bb）的局部公共子表达式消除（Local CSE）。
- 内联阶段：对每条指令做迭代，根据profile信息确定哪些调用是内联的候选。为内联调用位置建立控制流和IR，然后合并到调用者的控制流图和异常处理表中。
- 全局优化阶段：这个阶段完成了复写传播，常数折叠（const folding），复写传播（Copy propagation），死代码消除，数组越界检查消除， 循环不变量的代码移动（code motion）。 由于IA-32只有7个通用寄存器，它们是很宝贵的资源。所以我们会尽量避免那些增加寄存器压力的优化，例如代码移动、部分冗余消除（partial-redundancy elimination）。
- 局部寄存器分配和折叠：为方法的参数和局部变量分配寄存器；折叠立即数并为一些指令扩展底层表示。
- 全局寄存器分配： 分配物理寄存器并产生spill 代码（load/store指令）。
- GC 支持：在每条指令处计算GC map，使得每条指令都是GC-safe的。这里使用了简单的压缩技术减少GC map的大小。这里活对象分析包括：在垃圾收集中报告活引用（live references）并

记录寄存器和spill位置；节点入口的liveness以及活动范围的结束。

- 代码发射：对每条指令迭代，发射本地码和GC map的压缩字节流。

10.1.2 优化策略

在 O3 JIT 中除了使用一些局部的优化策略外，主要使用的优化策略可以归纳为：

- 局部公共子表达式消除
- 复写传播和类型传播
- 常数折叠
- 死代码消除
- 越界检查消除
- 循环不变量提升
- 内联
- 浮点栈（FP stack）的优化
- 寄存器分配
- 尾递归的消除

考虑到编译时间，O3 JIT 并没有像一般的编译器如 gcc,VC 那样做一些特别昂贵的优化，比如 ：

- 全局的公共子表达式消除，这样会带来过多的 load,store 指令，增加寄存器的压力。
- 部分冗余消除：同样也是会增加寄存器的压力
- 向量化（Vectorization）

Java 语言和虚拟机的新特性，如：运行期检查，垃圾收集，动态类加载、精确的异常处理等，为编译器的优化提出了新的要求。所以 O3 JIT 还应用了一些特定于 Java 虚拟机的优化。按照发生的阶段它们又可以分成静态优化和动态优化。静态优化通过静态分析减少运行时的开销。动态优化在运行期间发生，它利用编译期间不能获得的信息来修补本地码，减少 GC map 大小，建立栈帧缓存以及尽量避免产生异常对象等。

10.2 编译流程

O3 JIT 的主要流程包含在 arch\ia32\ia32_o3_jit\code_gen.cpp 中的 O3_compile_method 方法中。这个方法基本遵循了图 10.1 所描述的优化流程。在这一节，我们将对流程的每个模块作详细介绍。

在预遍历和建立 IR 之前还做了一些准备工作。

(1) 创建寄存器映射：为了区分建立 IR 时用到的各种寄存器，方便优化，这里我们对所有寄存器编号，划定它们的范围，当我们申请所需要的寄存器，就会得到一个合适的编号。

0, … ,n_reg (寄存器 id 从 0 开始，0-7 为物理寄存器 eax,ecx,edx,ebx,ebp,esp,esi,edi)
start_reg_id >= n_reg (start_reg_id 从大于等于 8 开始编号)
start_reg_id … start_reg_id + max_locals – 1(虚拟寄存器 Virtual_Reg 的范围，用于保存局部变量)
start_reg_id + max_locals ---->start_reg_id + max_locals + max_stack – 1（Java 运行栈的范围）
start_reg_id + max_locals + max_stack --->(临时寄存器 Temp_Reg 的范围，用于保存临时变量)

(2) 创建 kill id 的映射 Kill_ID_Map：这是为建立 CSE 做准备。它记录了表达式中变量和对象域集合的信息。这是一个 256 位的 ID 的 bit vector 表，其中每个变量在表中有唯一的索引，每个对象域有唯一的常数池索引。它们会随值的更新而更新。

(3) 创建局部公共子表达式池，这个结构将用来存放过程中已经出现的子表达式：

```
Mem_Manager lcse_mm(bytecode_size*sizeof(LCSE));
LCSE_Pool lcse_pool(lcse_mm);
```

(4) 利用上面生成的 reg_map，kill_id_map 和 lcse_pool 创建表达式池。

```
Expressions expressions(mem_manager,reg_map,kill_id_map,lcse_pool);
```

(5) 创建方法流图 Flow_Graph *fg;

O3_compile_method 中的全局函数 build_IR_of_method 完成了从预遍历到内联整个的循环过程。它是编译器前端形成 IR 的重要阶段。从 10.2.1 节到 10.2.4 节都是对这一过程的详细讨论。

10.2.1 预遍历

O3 JIT 的预遍历阶段划分方法的基本块（bb）并得到基本块的信息。这个过程是通过创建一个 CG_Prepass 对象完成。图 10.2 就是预遍历构造函数的流程。

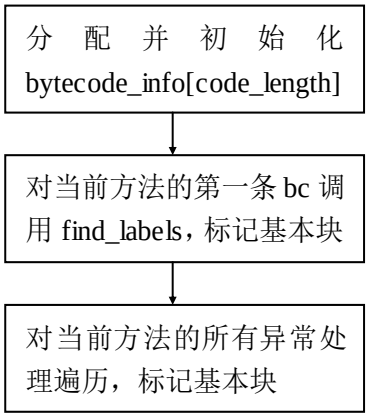


图 10.2 预遍历流程

1. 分配并初始化当前方法代码长度个 Bytecode_info 结构。

```
struct Bytecode_Info {
    unsigned is_block_entry:1;      //是否是 bb 的开始
    unsigned is_visited:1;          //在预遍历中是否已经访问
    unsigned fgnode_inited:1;       //Flow_Graph 是否已经初始化了 fgnode
    Cfg_Node *fgnode;               //为空，若被 Flow_Graph 处理了，则创建流图节点
    Cfg_Node *subr_ret;
    Return_Address_Tracking *stk;
};
```

通过遍历，结构的前两项属性得到赋值，而其他属性（fgnode）将被用于构造控制流图。

2. 接下来预遍历开始，我们对第一条 bc，即 bytecode_info[0]，调用 find_labels 函数。这是一个递归函数，它对当前方法分析，得到方法所有的基本块。

首先：因为是从方法的第一条语句，也是基本块的第一条语句进入，所以设置该条语句为一个新的基本块入口，如果当前的 bb 访问过了，则返回。

```
byte_info->is_block_entry = 1;      //设置新块入口标记
if (byte_info->is_visited)  return;     //若当前 bb 已访问过，则返回
```

其次：在当前基本块中遍历，这是一个循环过程，循环体分析每条指令得到可能的新 bb：

```
while (!byte_info->is_visited && idx < code_length){
    byte_info->is_visited = 1;          //设置当前块的访问标记
```

```
switch (bc[idx]){                    //对各种语句作处理，决定如何进一步划分 bb

case 0x99: case 0x9a:                //如果是 if{eq,ne,lt,ge,gt,le}这些条件分支指令

case 0x9b: case 0x9c:
case 0x9d: case 0x9e:

    offset = OFFSET2(bc,idx+1);              //转移目标块的偏移地址，依然位于当前方法中

    find_labels(comp_handle,bc, idx+offset);     //在转移目标块中继续标记 bb 边界

    bytecode_info[idx+3].is_block_entry = 1;     //另一个顺序分支也是新的 bb 开始

    break;

case 0xb4: {                 //如果是 getfield 语句，对域做解析

      index = OFFSET2(bc,idx+1);
      Field_Handle fh = resolve_nonstatic_field(comp_handle,class_handle,index,&exc);
      if (fh == NULL){

                result = JIT_RESOLVE_ERROR;  //解析错误

                break;
               }
      }
      break;

case 0xbb:                   //对于 new 系列指令，为代码生成预先解析 class

      index = OFFSET2(bc,idx+1);     //常数池索引

      resolve_class(comp_handle,class_handle, index,&exc);
      break;
case …   //其他语句的处理
}// switch 语句结束

idx += instruction_length(bc, idx);      //找到当前 bc 对应的操作长度

byte_info = &bytecode_info[idx];         //更新 idx 的值，仍然在当前方法中遍历代码

}//while 循环结束
```

这里我们介绍了一些典型指令的处理，具体可参见 O3 JIT 的 CG_Prepass::find_labels 函数。

3．对当前方法所有的异常处理划分 bb。由于 find_labels 函数中的 while 循环不能找到 try-catch-finally 子程序的边界，所以对异常要单独划分基本块。我们可以获得方法所有异常处理的描述信息。在 Java 规范中，每个方法异常表的每项描述一个异常处理，它包含四种信息：try 模块开始位置；try 模块的结束位置；异常处理开始位置；捕获异常的类型。我们将 try 模块的开始和结束位置标记为基本块入口，并对异常处理模块调用 find_labels 函数。

10.2.2 创建控制流图及流图转换

10.2.2.1 创建 CFG

在传统的编译器中，控制流图从抽象语法树转化得到。O1JIT 为了做全局寄存器的分配，构造了控制流图。O3 JIT 为了建立 IR，优化代码，也要对字节码遍历构造控制流图。

控制流图的构造过程基本在 Flow_Gragh 的构造函数中完成。在调用这个构造函数时，我们把刚刚得到的预遍历对象作为参数传入以利用预遍历得到的基本块信息。这个构造函数的流程和预遍历相似，其中也有个关键的、类似 find_labels 的递归函数 create_flow_graph。图 10.3 是构造 CFG 的简单流程：

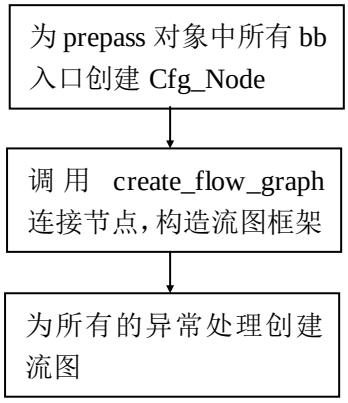


图 10.3 构造 CFG

1．对当前 prepass 对象的所有 Bytecode_Info 结构迭代，如果是 bb 入口，则构造 CFG 节点。CFG 节点是构成流图的基本单位，下面列出了它的一些主要属性，在后面的介绍中我们可以看到这些属性的使用。

```
class Cfg_Node
  {    …
       Cfg_Node *idom;                    //立即必经节点
       short   label;                     //节点的唯一表识
       Flow_Graph *flowgraph;             //节点所在的流图
private: …
       Cfg_Node **_out_edges;             //后继节点
       Cfg_Node **_in_edges;              //前驱节点
       Cfg_Int _out_edge_size;            //后继节点数目
       Cfg_Int _in_edge_size;             //前驱节点数目
       Eh_Node *_eh_out_edge;             //异常处理节点，它只能是出边联接的后即节点
       Cfg_Node_List *_linearization_node;  //控制流节点的线性序列（DFS 的反序？）
```

```
    Dlink _IR_instruction_list;          //节点 IR 指令的序列
    Live_LCSE *_live_lcse;               //活动的局部公共子表达式
    unsigned _first_bc_idx, _bc_length;  //第一条 bc 开始位置，节点所有 bc 长度
    unsigned _traversing:1;              //该节点和后继是否已被遍历
    unsigned _cold_code:1;               //这个 bb 是否不经常执行
    unsigned _code_length;               //为这个 bb 发射的机器码字节数
};
```

2. 连接所有的 CFG 节点，构造流图框架，这部分的代码片断如下：

```
Cfg_Node *tmp_nodes = create_flow_graph(0, retstack);     //从第一条指令进入
Cfg_Node *prolog = new(mem_manager) Cfg_Node(mem_manager, 0, this, &linear_node_ordering);
                                                          //创建 prolog 节点
prolog->add_edge(mem_manager,tmp_nodes);                  //添加从 this 节点到目标节点的边
nodes = prolog;          //flow_graph 的属性 nodes 指向了流图开始位置 ,它是从流图对象到
                         //节点对象之间的连接
```

create_flow_graph()是个递归过程，也是构造 CFG 的核心。它通过设置 CFG_Node 的节点属性将所有节点连接起来。

```
Cfg_Node *Flow_Graph::create_flow_graph(unsigned bc_start, Return_Address_Tracking *stk){
    …
    //如果已经处理过该节点，则返回
    if (_prepass->bytecode_info[bc_start].fgnode_inited)
        return _prepass->bytecode_info[bc_start].fgnode;
    //否则，找到节点的字节码范围，创建该节点的出边和后继节点
    unsigned cur_bc = bc_start;
    unsigned last_stmt;
    do{
        process_bytecode(stk, _bytecodes, cur_bc, _c_handle, _cmpl_handle);
        last_stmt = cur_bc;
        cur_bc += instruction_length(_bytecodes,cur_bc);
       }
    while (cur_bc < _bc_length && !_prepass->bytecode_info[cur_bc].is_block_entry);
    //循环结束后 bc_start 是当前 bb 的第一条指令 ;cur_bc 是下一个 bb 的第一条指令 ;last_stmt
```

```
//是当前 bb 的最后一条指令
Cfg_Node *result = _prepass->bytecode_info[bc_start].fgnode;
_prepass->bytecode_info[bc_start].fgnode_inited = 1;//设置 CFG_Node 访问属性
result->set_bytecodes(bc_start, cur_bc-bc_start);  // 设置 CFG_Node 的字节码开始位置
                                                    //_first_bc_idx 和字节码长度_bc_length
switch (_bytecodes[last_stmt])
{
case 0x99: case 0x9a: case 0x9b: case 0x9c: case 0x9d: case 0x9e:
//如果是 if{eq,ne,lt,ge,gt,le}指令，添加从当前节点到两个目标节点的出边，并继续处理后
继节点
    result->add_edge(mem_manager, create_flow_graph(cur_bc, stk->clone()));
    result->add_edge(mem_manager,
        create_flow_graph(last_stmt+OFFSET2(_bytecodes, last_stmt+1), stk));
    break;
case 0xa7: //如果是 goto 指令
    result->add_edge(mem_manager,
        create_flow_graph(last_stmt+OFFSET2(_bytecodes, last_stmt+1), stk));
    break;
…       //其他指令的处理
}//switch 语句结束
return result;
} //create_flow_graph 函数结束
```

和 find_labels 函数一样，这里也只列出部分指令的处理，详细情况可参见 flow_graph.cpp 中的 create_flow_graph 定义。

3. 创建所有异常处理的流图，添加异常处理节点 Eh_Node（类似于 CFG_Node）。

最后形成的控制流图结构如图 10.4 所示，图中的 epilog 节点以及每个节点的指令序列（即节点的 _IR_instruction_list 属性）都是在建立 IR 时得到的。

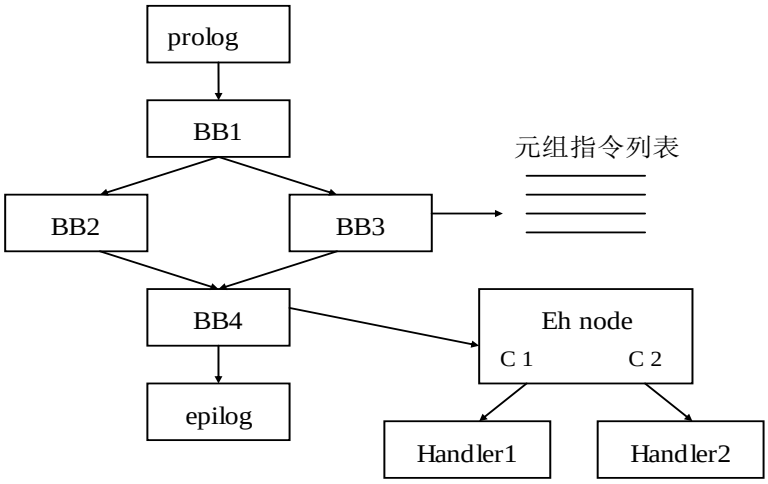


图 10.4 CFG 结构

10.2.2.2 循环转换

在生成IR之前，我们调用循环转换函数loop_transformation对CFG中的循环结构作转换。这种对CFG结构的早期优化减少了循环执行开销并提高了后续优化的效率。这里应用的转换包括：loop normalization，loop linearization，loop peeling和loop unrolling。为了确定循环结构，我们先调用Flow_Graph::build_dom_tree。这个函数使用Tarjan算法，通过对CFG深度优先遍历建立反映节点之间dominator关系的必经节点树。

- loop normalization：循环规格化是指将循环尽量转换成从1开始到n结束，步长为1的标准形式。

  例如：for (i=a;i<c;i=i+b)
  ```
  …;
  ```
  转换之后就成为：
  ```
  n= (c-a+b)/b;
  for (ii=1;ii<n;ii++)
  {
    i=a+(ii-1)*b
    …
  }
  ```
- loop linearization：循环的线性转换提高了内存访问的局部性，也带了更多的优化机会，比如更多的循环不变量。
  ```
  for (i=0;i<100;i++)
  {
    for (j=0;j<100;j++)
        c[j]=c[j]+a[i,j]+b[j]
  }
  ```
  转换为
  ```
  for (j=0;j<100;j++)
  {
    for(i=0;i<100;i++)
        c[j]=c[j]+a[i,j]+b[j]
  }
  ```
- loop peeling：这种转换通过在循环开始之前插入循环体的拷贝，从而移出循环的头一次或头几

次迭代。

- loop unrolling：这种转换用多个循环体的拷贝取代原有循环体，并调整相应的循环次数。循环体被复制的次数叫做unrolling factor，这里我们取3。所以如果是下面的循环

```
for (i=0;i<60;i++)
        s+=a[i];
```

转换之后就成为：

```
for(i=0 ;i<60;i+=3)
{
    s+=a[i];
    s+=a[i+1];
    s+=a[i+2];
}
```

10.2.3 建立中间表示（IR）

在编译器中，建立 IR 的过程总是和建立 CFG 的过程联系在一起。控制流图表达了不同块之间的关系，而块中的操作则是通过 IR 来表示。这里的 IR 是源操作数最多为三个的元组表达式。我们首先介绍 IR 的基本结构，然后介绍建立 IR 的过程，最后介绍在建立 IR 过程中应用到的一些优化策略。

10.2.3.1 一些数据结构

1. 操作数类（Class Operand）和子操作数类

   在 O1 JIT 中，我们介绍了模拟栈上的操作数层次结构。同样，构造 IR 时也需要各种类别的操作数。图 10.5 显示了 IR 中所有操作数的种类(kind)，操作数的种类给出了操作数在虚拟机内的实际分布。

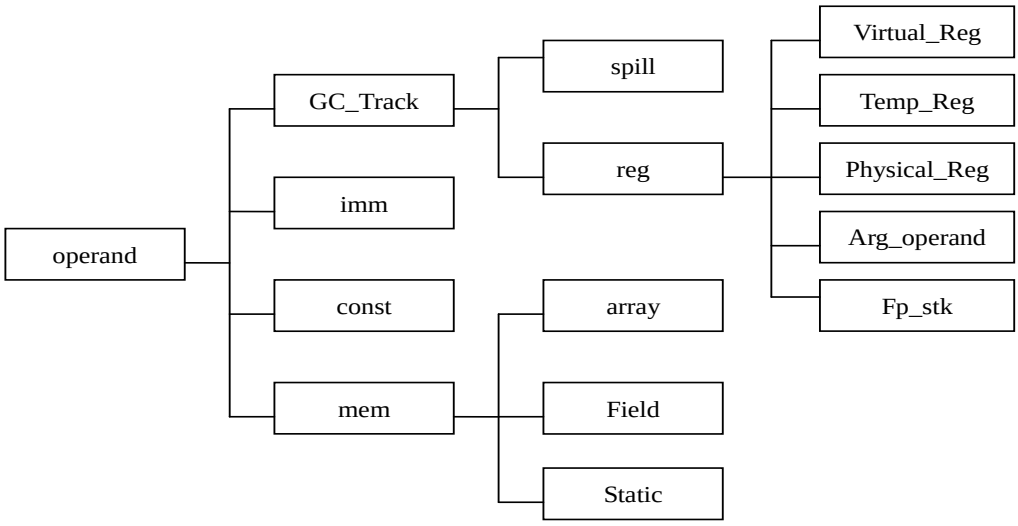


图 10.5 IR 操作数的种类（kind）

操作数的另一个属性是它的类型（type），类型保证了 Java 语义所限定的操作，常说的 Java 的类型安全特性就是基于这种基于类型的操作而言的。显然，操作数的类型给出了虚拟机对 Java 语义要求类型的表示。O3 JIT 中操作数的类型定义在 O3_JIT_TYPE 中，直觉上，至少应该包括所有的 Java 类型，当然还有一些辅助的其他类型：

```
const char JIT_TYPE_RETADDR = 'R';   //用于表示 jsr 指令的返回地址
```

```
const char JIT_TYPE_ADDR       = 'A';   //用于表示方法调用地址

const char JIT_TYPE_UNKNOWN = 'U'; //用于表示 finally 块中的未知类型

//以下都是 Java 类型

const char JIT_TYPE_BYTE       = JAVA_TYPE_BYTE;
const char JIT_TYPE_CHAR       = JAVA_TYPE_CHAR;
const char JIT_TYPE_DOUBLE     = JAVA_TYPE_DOUBLE;
const char JIT_TYPE_FLOAT      = JAVA_TYPE_FLOAT;
const char JIT_TYPE_INT        = JAVA_TYPE_INT;
const char JIT_TYPE_LONG       = JAVA_TYPE_LONG;
const char JIT_TYPE_SHORT      = JAVA_TYPE_SHORT;
const char JIT_TYPE_BOOLEAN = JAVA_TYPE_BOOLEAN;
const char JIT_TYPE_CLASS      = JAVA_TYPE_CLASS;
const char JIT_TYPE_ARRAY      = JAVA_TYPE_ARRAY;
const char JIT_TYPE_VOID       = JAVA_TYPE_VOID;
const char JIT_TYPE_STRING     = JAVA_TYPE_STRING;
const char JIT_TYPE_INVALID = JAVA_TYPE_INVALID;
const char JIT_TYPE_END        = JAVA_TYPE_END;
```

2. 表达式 Exp 和表达式组 Expressions

- 每个方法对应一个 Expressions，它是若干 Exp 组成的集合，Expressions 的一个重要作用就是纪录基本块中活动的公共子表达式信息。

```
class Expressions {
   Expressions(Mem_Manager& mem_manager,RegID_Map& rm, Kill_ID_Map& km, LCSE_Pool&
  pool) : mem(mem_manager),   kill_id_map(km), _live_lcse(NULL), lcse_pool(pool), reg_map(rm) {
   _next_exp_id = 0;
   _data_space = 0;
   unsigned i;

   for (i = 0; i < MAX_HASH_ENTRY; i++) _table[i] = NULL; //初始化 Expressions 的哈希表入口

  }
   void set_live_lcse(Live_LCSE *l) {_live_lcse = l;}
   Live_LCSE *live_lcse() {return _live_lcse;}

   Temp_Reg *create_new_temp_reg(O3_Jit_Type type) { …}//分配一个类型 type 的临时寄存器

   void gen_local_cse(Exp *e,Inst *i, bool for_assgn = false);

   Mem_Manager& mem;       //用于内存分配

   RegID_Map& reg_map;     //寄存器 id 映射

   Kill_ID_Map& kill_id_map;

   LCSE_Pool& lcse_pool;     //保持 LCSE 列表

private:
```

```
    Link<Exp> *_table[MAX_HASH_ENTRY];    //由 Exp 构成的 expressions 哈希表

    unsigned _next_exp_id;    //当前块中元组表示的开始 id

    Live_LCSE *_live_lcse;   //活动的 lcse 列表

    …
}
```

- 每个 Exp 对象分很多种类（kind），如：域访问 Field；赋值 Assign；整数加 Add；carry 整数加 Adc；比较 Compare；小于比较 Cmp_LT 等。每个 Exp 可以包含在一条或多条元组指令 Inst 中，反之每条 Inst 都包含了某种特定的 Exp。下面列出了 O3 JIT 所有用到的 Exp 类型：

```
ExpKind =enum {
        Opnd, Array, Field, Length, Widen, Assign, Add, Adc, Sub, Sbb, Fadd, Fsub, Mul,
        Fmul, Smul, Div, Fdiv, Rem, Frem, Neg, Fneg, Convt, And, Or, Xor, Shl, Sar, Shr,
        Compare, Cmp_LT, Cmp_GT, Test, Beq, Bne, Blt, Bge, Bgt, Ble, Call, New,
        Newarray, Anewarray, Multinew, Vtable, Return, Next_PC, Jump, Tableswitch,
        Lookupswitch, String, Instanceof, Checkcast, Monitorenter, Monitorexit, Monitorenter_static,
        Monitorexit_static, Athrow, InterfaceVtable, Push, Pop, ClassInit, WriteBarrier,
        ReadBarrier, Intr_Sin, Intr_Cos, Intr_Sqrt, Intr_Rndint, Intr_Abs, n_exp
    };
```

3．元组指令 Inst

有了表达式的操作种类和操作数，就可以构成 bb 中的 IR 指令了。由于在编译过程中 IR 指令经常被插入或删除，所以它们通常都被组织成双向链表(Dlink)。在代码发射时，这些元组指令基本上和机器指令一一对应。

```
class Inst : public Dlink {          //Inst 还有一些 Dlink 的继承属性，例如_next,_prev
public:
    Exp *const exp;                  //表达式种类

    const unsigned char n_srcs;      //本条指令中源操作数的个数

    unsigned char live_ranges_ended;
    unsigned char region_live_ranges_ended;

    void emit_inst(O3_Emitter& xe, X86_Opnd_Pool& opnds) {emit(xe, opnds); }//inst 对应的发射指令

protected:
    virtual void emit(O3_Emitter& xe, X86_Opnd_Pool& opnds) {assert(0);}
    unsigned char _dead:1;
    unsigned char _been_expanded:1;
    unsigned char _marker:1;
    unsigned char _gc_unsafe:1;

    Operand *_srcs[MAX_SRCS];            //源操作数个数，最多为 MAX_SRCS=3 个

    Operand *_dst;                       //目标操作数
```

```
    //一个二元操作数 IR 指令的构造函数
    Inst(Operand *src0,Operand *src1, Exp *e, Inst *inst_head) : n_srcs(2), exp(e) {
      insert_before(inst_head);              //附加到 inst 列表的尾部
      _srcs[0] = src0; _srcs[1] = src1;      //设定操作数
      INIT_INST();                           //指令初始化
    }
    …
  };
```

O3 JIT 中 IR 的 Inst 类型有：

```
InstKind=enum {
  Widen_Inst, Assign_Inst, Obj_Info_Inst, Branch_Inst, Compare_Inst, Add_Inst, Sub_Inst, Mul_Inst,
  Div_Inst, Neg_Inst, Bitwise_Inst, Call_Inst, Return_Inst, Convt_Inst, NextPC_Inst, Jump_Inst,
  Switch_Inst, String_Inst, Classinit_Inst, Writebarrier_Inst, Type_Inst, Push_Inst, Pop_Inst, Math_Inst,
  Deref_Inst, Native_Inst
}
```

10.2.3.2 建立 IR

flow_graph 的成员方法 Build_IR()负责建立 IR。这个方法产生显式的参数传递代码，完成从形参到实参的转换，（即 v0 = arg0 ,v1 = arg1 , ...）并调用 flow_graph 的另一个成员方法 Build_IR_node 遍历流图所有节点，建立节点 IR。Build_IR_node 是个递归过程，它的流程如图 10.5 所示。

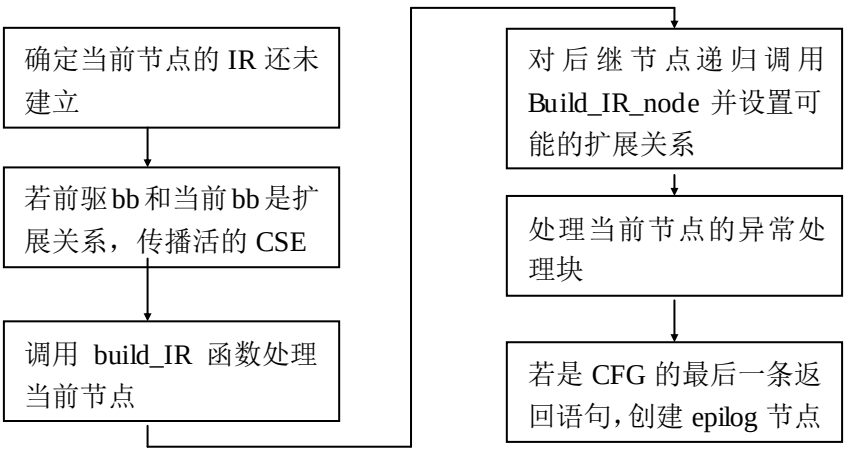


图 10.6　flow_graph::Build_IR_node 函数流程

左边第二个模块提到了前驱 bb 和后继 bb 的扩展关系。只有当后继节点只有前驱一个入边时，那么我们才能设置 is_extended_bb 为真，并将前驱 bb 中活的 CSE 传播到后继 bb 中。build_IR()函数（注意这个函数和 flow_graph 成员方法 Build_IR 的区别）处理当前节点的每条 bc，建立元组指令列表，并将这些指令依次添加到 CFG 节点的指令列表（_IR_instruction_list）中。下面是 build_IR 的部分代码片断：

```
void build_IR( Inst *inst_head, //传入当前节点的_IR_instruction_list，新建的 Inst 指令添加到这个列表中
               char    *stack_sig_in,      //前一个 bb 的栈类型签名，即栈上元素类型数组
```

```
                char     *stack_sig_out,       //当前 bb 的栈类型签名
                …)
{
  if (is_exception_entry) {                    //在异常处理的入口，栈上应该只有一个异常对象
      assert(stack_sig_in_size == 1);
      stack_sig_in[0] = JIT_TYPE_CLASS;
  }
  stack.reset();   //在块的入口初始化模拟栈
  int i;
  for (i=0; i < stack_sig_in_size; i++) //在模拟栈上压入前一个 bb 栈元素
  {
      Inst *ii;
      if (is_exception_entry)
          ii = exprs.lookup_preg(eax_reg, JIT_TYPE_CLASS, inst_head);
      else
          ii = exprs.lookup_stack(stack.depth(),(O3_Jit_Type)stack_sig_in[i],inst_head);
      stack.push(ii);
  }
  const unsigned char *bc = first_bc + node->first_bc_idx();   //当前指令
  const unsigned char *last_bc = bc + node->bc_length();       //最后一条指令
  const unsigned char *prev_bc;
  const unsigned char *curr_bc = bc;       //指向当前指令的指针
  unsigned bc_index = bc - first_bc;
  //下面的 while 循环为当前节点的所有指令建立 IR
  while (bc < last_bc) {
      prev_bc = curr_bc;
      curr_bc = bc;
      bc_index = curr_bc-first_bc;         //当前 bc 的偏移
      unsigned char bytecode = *bc;        //当前 bc 的值
      switch (bytecode) {                  //0x00…0xc7 的字节码处理
      case 0x02: case 0x03: case 0x04: case 0x05:
      case 0x06: case 0x07: case 0x08: // iconst -1,0,...,5
          src = exprs.lookup_imm(bytecode-0x03,JIT_TYPE_INT,inst_head);//立即数表达式
```

```
            stack.push(src);                //压入操作数栈
            break;
        case 0x09: case 0x0a:               // lconst 0,1
            val.l.hi = 0; val.l.lo = bytecode-0x09;
            stack.push(exprs.lookup_const(&val,JIT_TYPE_LONG,inst_head));//常数表达式压入栈
            break;
        case 0x36:case 0x37: case 0x38: case 0x39: case 0x3a:  // istore, lstore fstoredstore,astore 指令
            type = (O3_Jit_Type)("IJFDL"[bytecode-0x36]);   //判断 store 操作数类型
            //产生赋值指令"vreg = src"
            gen_assign(mem_manager,exprs,stack,inst_head,bc[1],stack.pop(),type);
            break;
        …
        } // switch 语句结束
        bc += instruction_length(first_bc, bc_index);
    } // while 语句结束
    if (stack_sig_out_size == -1){
        spill_left_on_stack(mem_manager,exprs,stack,inst_head,stack_sig_out,stack_sig_out_size);
    }//循环结束，将操作数栈上的值移出，这些值的类型列表形成了下一个 bb 的 stack_sig_in
    exprs.live_lcse()->snap_shot_of_lcse(); //在 bb 结束处更新 Expressions 的 live cse 集合
}
```

在上面程序片断的 while 循环中，我们利用模拟栈，压入合适的操作数（operand）并在构造 IR 指令时弹出这些操作数，形成合适的 Inst 元组指令，然后将这些 Inst 指令加入到节点指令列表中。而对于其他一些对象操作（如域的解析），则按照 Java 规范操作。

10.2.3.3 局部公共子表达式删除（local CSE）

在生成各种指令对应的IR时，为避免产生冗余指令，一个重要的优化策略就是公共子表达式的删除。在O1 JIT中，我们已经接触了简单的CSE算法。为了判断两个表达式是否相同，O1 JIT简单的比较了涂写寄存器中保存的连续字节码子串。在O3 JIT中，我们通过Expressions、Exp、Inst和LCSE结构实现了更加完整的跨扩展块的局部CSE算法。

在 10.2.3.2 中提到，如果前驱节点和后继节点是扩展关系，我们会将前驱节点的活动 CSE 传播到后继节点，接下来在单个 bb 中作局部公共子表达式的删除。每个 Exp 中变量和对象域构成了它的 kill_set。对于新产生的表达式，对于赋值表达式的右边，我们在 exprs.local_cse 中寻找，如果找到则不用再生成左边表达式；如果赋值改变了表达式的某项，则 kill 这个 lcse。

10.2.4 内联（inlining）

面向对象的语言中存在很多经常被调用的小方法。这些函数很简单，调用和返回的代价要比真正执

行还要大。所以我们采用内联优化，用函数方法体取代函数调用。决定内联什么样的函数是很重要的。在O3 JIT中，内联策略基于代码大小和profiling 信息。如果方法入口的执行频率低于特定值，那么方法就被认为是冷方法而不会内联。为了避免代码量增加太多，我们也不会内联一个字节码总数大于25字节的方法。内联是个递归过程。编译器遍历IR，决定哪个调用位置需要被内联，然后编译器再为这些内联方法建立CFG和IR。在把新创建的CFG嫁接进调用者的流图前，编译器为内联方法重复相同的内联过程。为了避免沿一个深调用链内联，使得嵌套过深，当沿着调用链的总内联字节码超过40字节，内联就会停止。

10.2.4.1 宏定义和内联策略

为了说明问题的方便，我们列出内联策略相关的一些宏定义：

```
#define SMALL_METHOD_SIZE       25    //每个内联方法不得超过 25 字节

#define TINY_METHOD_SIZE        5     //对于小于 5 个字节的方法如 java/lang/Object.<init>或是

                                      //"aload_0; getfield #10; ireturn;"，我们无条件的内联

#define MAX_INLINE_SIZE         40    //沿着一条调用链，内联方法总的字节码数不超过 40

#define COLD_PATH_RATIO         10    //一个方法不经常执行，那么该节点是冷节点
```

10.2.4.2 内联过程

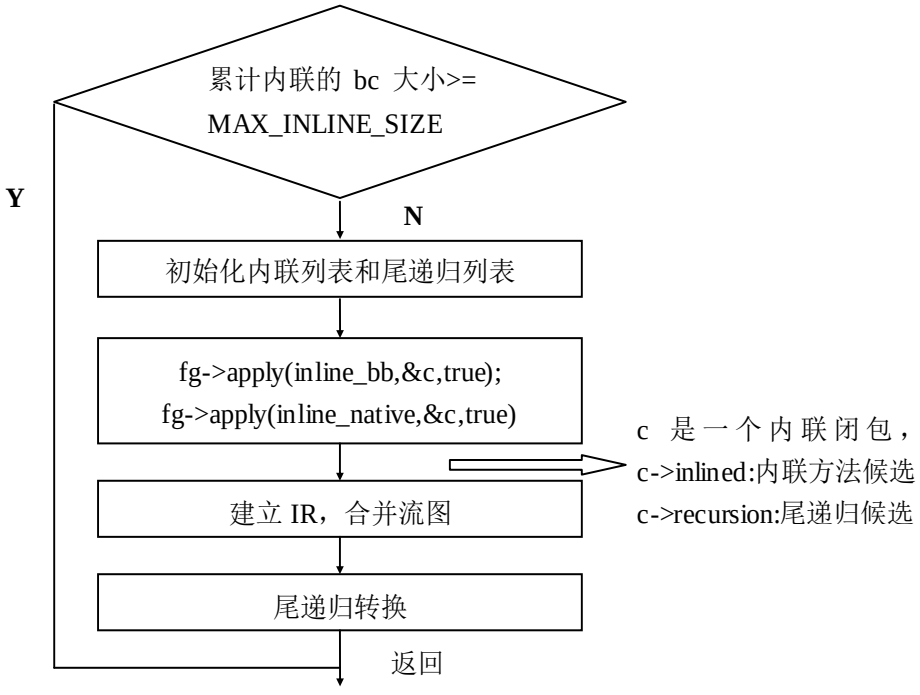


图 10.7 内联流程

图 10.6 是内联的流程图。CFG 的 apply 函数是一个特殊的函数，在程序的很多地方都有应用。它的第一个参数是个函数指针*f，它的程序体遍历整个流图，取得当前节点 node，执行 f(node)。这个函数使得很多需要遍历 CFG 的方法可以共用部分代码，大大减少了重复代码。这里我们应用了 inline_bb 和 inline_native 两个函数。前者遍历节点的 IR 指令列表并内联方法调用，后者内联 native 代码。

1．inline_bb 函数：决定内联候选方法列表和尾递归候选列表。

```
void inline_bb(Cfg_Node *node, Closure *c) {
    Inline_Closure *ic = (Inline_Closure*)c;    //设置 ic 为指向内联闭包的指针
    Inst *i;…
    for (i = head->next(); i != head; i = i->next()) {//head 是节点的第一条 IR 指令
    {
        //找到 checkcast 调用这样我们可以插入代码做快速检查，checkcast 的内联检查将在静态
        //优化中提到
        if (i->is_type_inst() &&((Type_Inst*)i)->kind == Type_Inst::cast && i->is_native_inline())
        {… }
        if (!i->is_call()) continue;
        if (!i->is_inlinable_call()) continue;
        Method_Handle mh = ((Call_Inst*)i)->get_mhandle();  //此时的 i 是个没有被内联的调用指令
        //寻找尾递归，设置 ic->recursion
        if (mh == node->flowgraph->m_handle() && node->flowgraph->epilog() != NULL) {//检查是
        否调用和当前方法有相同的 method_handle；检查是否节点是返回块，即是否当前方法的
        最后一条语句。
            Cfg_Node *epilog = node->flowgraph->epilog();
            Eh_Node *eh = epilog->eh_out_edge();
            Cfg_Node *b = node;
            while (b->out_edge_size() == 1 && b->out_edges(0) != epilog && b->eh_out_edge() == eh)
                    b = b->out_edges(0);     //单条出边&到达 epilog 节点&相同的异常处理
            if (b->out_edge_size() == 1 && b->out_edges(0) == epilog &&b->eh_out_edge() == eh)
              {
                ic->recursion = new (ic->inlined_mem) Recursion(node, (Call_Inst*)i, ic->recursion);
                continue;
              }
        }
        bool safe_inline = safe_to_inline(mh, (Call_Inst*)i); //判断内联是否安全
        //最终决定是否希望内联这个方法
        unsigned inlined_bc_size = ic->inlined_bc_size;
        if (safe_inline ||(((Call_Inst*)i)->kind == Call_Inst::virtual_call)) {
            Profile_Rec *inlined_prof = NULL;
            if (inline_policy(ic->fg, node, mh, inlined_bc_size,ic->global_inlined_bc_size, inlined_prof,
```

```
                                (Call_Inst*)i)) {
                    bool needs_deref = should_deref_before_inline(mh, (Call_Inst*)i);

                    //加入一个 Inlined_Method 对象到内联候选方法列表中

                    ic->inlined = new (ic->inlined_mem) Inlined_Method (node, (Call_Inst*)i, ic->inlined,
                                    safe_inline, needs_deref, inlined_bc_size, inlined_prof);
                    }
                }//if 语句结束
            }//for 节点指令遍历结束
        }inline_bb 结束
```

关于这个函数的几点说明：

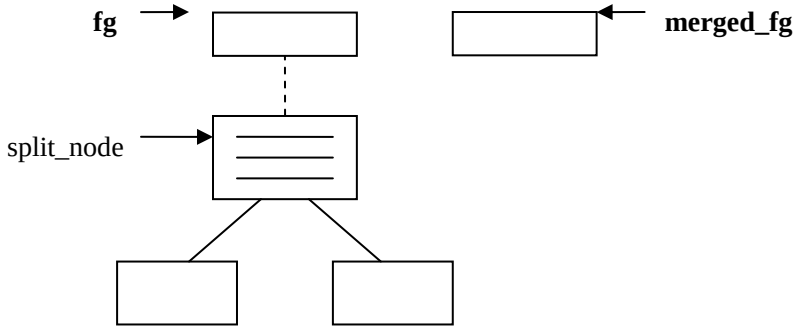
1) 尾递归：如果一个函数的最后一条语句是一个对自身的调用，那么我们就把这个调用叫做尾递归。这种特殊的调用可以被优化成返回函数顶部的一个分支。这样避免了通常调用的开销，所以也可以看作内联的一种。也就是说我们用一个绑定参数的分支来取代函数调用。这里设置闭包的 recursion 属性做为尾递归候选列表。
2) 内联的安全：方法内联是很重要的优化策略，但是静态编译器必须要决定内联是否安全。这是因为，首先：静态编译器必须确定该方法没有在子类中被覆盖。所以 static, final, 和 private 方法的内联总是安全的，因为这些方法是不会被覆盖的。然而， public 和 protected 方法因为可以在子类中被覆盖，所以编译器不能内联这些方法。其次，即使可以通过静态分析决定这些方法并没有被覆盖，编译器也无法内联 public 和 protected 方法。这是因为 Java 允许类在运行期加载，而动态加载的类就可以改变程序的结构。这种动态加载可以使任何基于运行前静态分析做的内联变得无效。有关虚方法调用（Invokevirtual）的内联我们将在 10.3.4 节介绍。
3) 内联策略：内联方法不但要安全，还要符合特定的内联策略。
   - 如果内联方法的 handle 为空（native 方法）或是不经常执行的路径，不进行内联。
   - 按照 Java 语言规范的定义。如果类 C 中的方法调用了类 S 中的方法 m ，那么类 C 和 S 要有相同的 class loader 并且类 S 不是类 SecurityMananger ，也不是它的子类，这样我们才可以考虑内联这个方法 m。
   - 不内联一个异常对象的构造函数
   - 方法字节码大小少于 SMALL_METHOD_SIZE 字节；调用链累计 bc 大小少于 MAX_INLINE_SIZE
   - 如果定义了重编译选项，利用 O1 产生的 profile 信息决定当前调用是否 hot，否则默认可以内联

2. 得到了内联方法列表后，我们为内联方法建立 IR，合并流图。

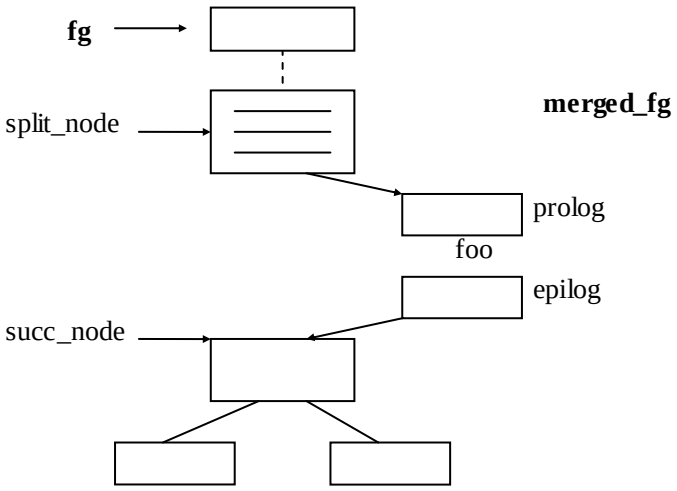
```
Inlined *im;
for (im = inlined; im != NULL; im = im->next) {

    im->build_IR(&c);                          //建立内联方法的流图 im->merged_fg

    im->merge_flow_graph(fg,exprs,recursion);  //合并流图

}
```

内联方法的 build_IR 方法递归调用 build_IR_of_method 建立 IR 和 CFG(图 10.1 中的回边)，此时内联节点的 caller_fg 就不为空了。通过这个函数我们得到 im->merged_fg，即被调用的内联函数的流图。内联方法和原流图的合并是个很有趣的过程。这里我们介绍一般内联过程（虚方法的内联还需要插入代码，call_node 检查所跳转的内联方法是否正确 checkcast）：

1) 初始流图 fg，包含 call foo 语句的节点 split_node，内联方法 foo 的流图 merged_fg

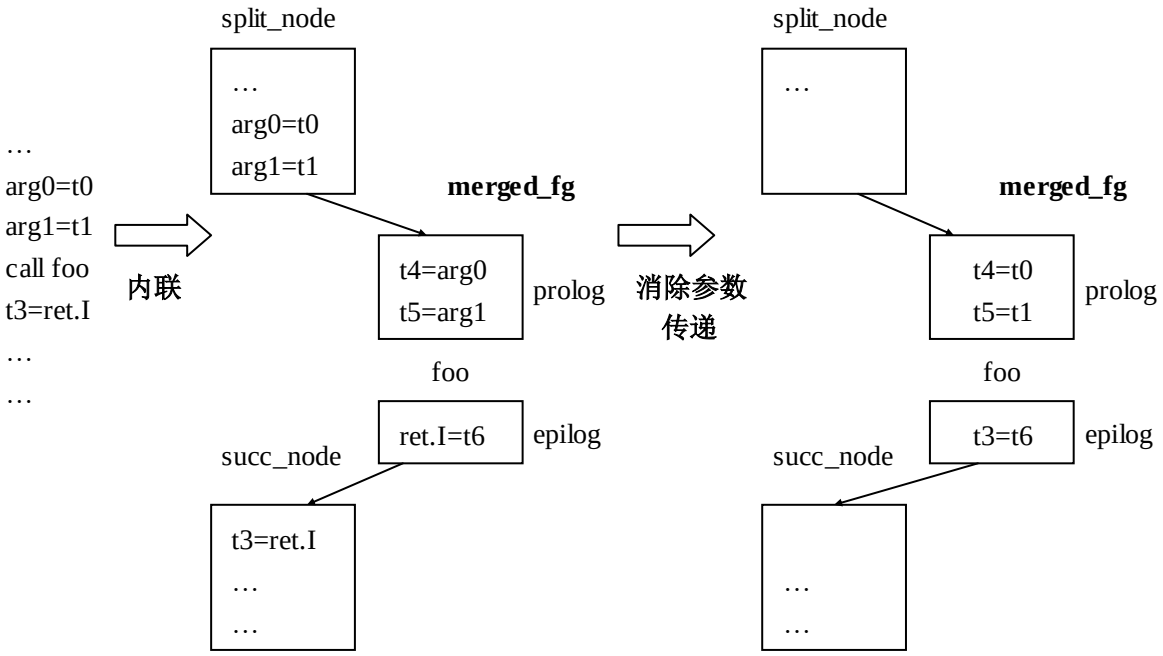


2) 分裂 split_node 成两个节点，一个节点为 split_node，另一个为 succ_node。后继节点的 inst 序列为空，但是继承了 split 节点的出边节点。将 merged_fg 的所有节点按线性复制到 succ_node 之前，添加从 split_node 到 merged_fg 的 prolog 的边，以及从 merged_fg 的 epilog 到 succ_node 的边



3) 第 2)步完成后，节点结构基本确定，接下来处理 split 节点中的指令

首先将 foo 调用之后的 inst 指令拷贝到 succ_node 中，删除内联方法 epilog 中的"return" 指令，将 merger_fg 的异常处理合并到 fg 中，（将调用，参数转换和返回指令转移到 call_node 中）。然后消除参数传递和返回语句。这个过程在 eliminate_arg_ret(merged_fg->prolog(), merged_fg->epilog())中完成。下图反映了特定指令序列 split 节点经过内联、参数和返回语句删除而发生的变化。



3. 尾递归转化：通过 inline_bb，我们收集了尾递归的信息并保存在流图信息中，这样就可以对尾递归做转化了。

内联完成后，整个 build_IR_of_method 结束，返回到 O3_compile_method 中。方法所有的基本信息都包含在了流图对象中。接下来就可以分析流图，进行全局优化和代码生成。在这之前，如果发现这是个同步方法，那么我们还要调用为进入和退出方法生成的 monitor 的 IR 指令（Build_ir_routines.cpp 的 gen_synch_method_enter，gen_synch_method_exit）

10.2.5 全局优化

全局优化函数主要包含了复写传播 copy_prop(fg, exprs)和 fold_branch(fg)。

- 复写传播 又叫做值传播。它指的是如果X是Z的一个拷贝，那么在X的拷贝处和使用处中间，只要X和Z都没有改变，那么使用X的地方就可以被Z取代。这种传播本身不会带来优化，但是它为我们带来了消除死代码的机会。如x:=a; print x;或是x:=false;if x then…。前者的赋值语句可以消除，后者的条件语句可以消除。编译器的前端通常会产生很多的临时变量，比如循环计数。所以这种优化是很有必要的。
- 分支折叠fold_branch(fg) 我们希望折叠形如eax = 0;test eax, eax; je target的分支。这种情况可能发生在内联或是常数/复写传播之后。

10.2.6 寄存器分配

编译器不仅要产生正确的执行代码，还要合理的分配资源，这其中就包括寄存器资源。和内存操作相比，寄存器操作更短更快，所以寄存器的分配对于程序的执行效率尤为重要。在寄存器分配中，编译器要决定在每个程序点有哪些值是要保存在寄存器中（allocation），以及应该保存在哪个寄存器中（assignment）。其中，局部寄存器为每个方法中或是基本块中的每条指令决定哪些变量要放在寄存器中。在每个方法结束处，我们还需要保存那些依然活动的变量。所以为了避免过多的 store/load 指令，还可以为哪些经常使用（如全局）的变量分配寄存器，这就是全局寄存器分配。在这里我们并不存在跨过程的寄存器分配，而且为简单起见，还指定了局部/全局的分配候选。在 O3JIT 中，我们用 4 个 callee-saved 寄存器（ebx,ebp,esi,edi）作为全局寄存器保存每个方法帧中的局部变量（包括函数参数），三个 caller-saved 寄存器（eax,ecx,edx）则用来保存指令中的临时变量。虽然指定个数未必带来最有效的分配策略，但这

确实是很简单的一种分配方法。

10.2.6.1 局部寄存器分配

O3 JIT 局部寄存器为保存临时计算结果的变量分配三个 caller-saved 寄存器，它由三部分组成（local_reg_alloc.cpp 中的 local_reg_allocation）：

1. 决定局部寄存器的分配候选

我们首先要确定哪些变量要被分配到局部寄存器中。对于确定的局部寄存器候选，设置 Reg_Operand 的 yyy. _assign_local_reg 位：

```
class Reg_Operand{
protected:
    union {
        struct {
            unsigned _assigned_physical_reg:8;        //实际被分配的物理寄存器
            unsigned _global_reg_alloc_cand:1;        //如果是全局寄存器候选，设为 1
            unsigned _assign_local_reg:1;             //如果是局部寄存器候选，设为 1
            unsigned _assigned_entry:5;               //用于局部寄存器分配
            unsigned _temp_reg_has_multiple_defs:1; //如果设为 1，则是虚拟寄存器
        } yyy;
        unsigned xxx;
    };
}
```

对于流图中每条指令，我们要判断它的源操作数是否是局部寄存器候选。实际上在生成 Inst 中间指令时用到的临时寄存器操作数 Temp_Reg 缺省的都是局部寄存器的候选；而表示局部变量的虚拟寄存器操作数 Virtual_Reg 缺省的是全局寄存器分配的候选。这里我们要保证：数组操作数 Array_Operand 的 base 和 index 必须在寄存器中；域操作数 Field_Operand 的 base 必须在寄存器中。由于 move 指令只能有一个 memory 操作数，所以如果赋值语句将 src 的值存贮回内存，那么 src 必须是在 register 中。

2. 折叠立即操作数（imm）和内存操作数（mem）

操作数的折叠和 code motion 类似，它减轻了寄存器的压力。例如：对于指令序列"t1 = [t0 + 4], t2 = t1, add t2, t3"，我们在 add 指令中用内存操作数[t0 + 4]，而不是 t1 取代 t2。

操作数的折叠过程遍历所有指令，对于指令中所有的源操作数，寻找可以折叠的 tmp_reg 操作数，用新的操作数（如[t0+4]）取代指令中原有的操作数（t2），并把原来定义 tmp_reg 的指令标记为死代码。要注意的一点是：由于 iA32 寻址模式要求某些特定操作必须在寄存器中进行，所以对于域/数组操作，我们只对寄存器赋值指令做折叠，如 r_idx=r1,r_base=r2,t=[r_idx+r_base]中前两条。

3. 为局部临时寄存器分配寄存器 ALL_X86_CALLER_REGS，即寄存器的 assignment

我们通过 Local_Reg_Manager 对象跟踪哪个局部寄存器表示哪个局部变量的活动范围。分配结束后返回这个对象的_ever_used_pregs 属性，它记录了分配的结果。

```
class Local_Reg_Manager {
private:
    unsigned const _avail_local_regs; //如果对应 reg 位为 1，则可用来赋值
```

```
    unsigned _max_local_regs;          //局部寄存器最多为 3 个
    unsigned _live_preg_lr;            //记录物理寄存器的活动范围。如果某位置 1，对应物理寄存器
是活的
    unsigned _next_free;               //下一个空闲的寄存器
    unsigned _free_entry;              //位向量指明在_ready_be_assigned 是否有空闲分配入口
    unsigned _ever_used_pregs;         //已经使用的物理寄存器
    Physical_Reg *_pregs[MAX_LOCAL_REGS];                  // X86 物理寄存器组
    Reg_Operand  *_ready_be_assigned[MAX_LOCAL_REGS];//放入其中的操作数等待分配寄存器
    unsigned       _free_bv[MAX_LOCAL_REGS];               //可以使用的寄存器
    X86_Reg_No     _preference[MAX_LOCAL_REGS];        //寄存器的分配优先选择
};
```

局部寄存器的分配算法是个单遍分配方法，它的算法描述如下：

```
{
生成 Local_Reg_Manager 对象 lrm 记录变量和寄存器信息;

for 从 CFG 的尾部进入，按照执行反序遍历得到的每条指令 i
{
    移出操作数折叠阶段产生的死赋值语句；
    假设 i 形如 t0=t1+off，
    对于目标操作数 t0，定义标记着活动范围的结束{
    如果 t0 已经被赋予物理寄存器，从这一点开始就可以释放这个寄存器，更新 lrm 的_live_preg_lr，
    将该寄存器对应活动位置 0。

    否则如果 t0 是局部寄存器候选
        if (reg->assigned_entry() == NOT_YET_ASSIGNED) 表明这个变量从未作为源操作数被
        分配寄存器，是个死变量，返回，

        找到合适的寄存器集合_free_bv[entry]。如果集合中的寄存器全部都被分配出去了，将这
        个 reg_opnd 设为全局寄存器的候选；否则挑选一个合适的作为物理寄存器，设置 t0 操
        作数的 yyy._assign_local_reg 。

        从_free_bv 中移去刚分配的寄存器并更新 lrm 的_ready_be_assigned[entry]，_free_entry 和
        _next_free 的值
    }
```

对于源操作数 t1，使用标记着活动范围的开始{
如果 t1 已经分配了物理寄存器，将该寄存器设置为活动寄存器

否则如果 t1 是局部寄存器候选
if (reg->assigned_entry() != NOT_YET_ASSIGNED) 返回
如果(!_free_entry)没有空闲寄存器，标记为全局寄存器的候选，否则如果存在 free entry 找到合适的寄存器 using the next_free_entry hint

将该寄存器从 free_bv 中删去，更新 lrm 的_ready_be_assigned[entry]，_free_entry 和 _next_free 的值
}
}//指令 i 处理完毕
}//算法结束

图 10.8 局部寄存器分配算法

10.2.6.2 全局寄存器分配

现在要为局部变量和局部分配新产生的全局分配候选分配寄存器了。O3 JIT 缺省采用基于优先级的全局寄存器分配算法；如果在命令行定义"nopriority"参数，则采用简单寄存器分配算法。由于先做了局部寄存器分配，所以在做完全局寄存器分配后，必须要为寄存器压力过大的程序点产生 spill 代码。为了提高执行效率，还可以适当的调整代码顺序（code schedule）。

1. 基于优先级的分配算法（priority_reg_alloc）

全局寄存器分配的算法由 arch\ia32\ia32_o3_jit\priority_reg_alloc.cpp 中的 priority_reg_alloc()函数实现，它返回一个使用到的 callee-saved 寄存器的 bitmask。对引用到的每个变量，我们保持引用这个变量或是变量活动的 bb 的位向量。这包括物理寄存器以及保存由于调用或异常而被破坏的涂写寄存器。我们还为所有变量保持一个静态引用计数。对每个 bb，我们保持一个 8-位的 mask 表明在 bb 中哪个寄存器是空闲的。对于每个带有静态引用计数的变量，我们找到一个在所有对变量引用的 bb 中都是空闲的寄存器。将这个寄存器分配给变量，并在所有这些块中标记这个寄存器不可用。下面是基于优先级的分配算法的具体过程：

首先，程序会从输入参数的流图中对所有的引用进行计数，这部分的主要功能是由 count_all_refs()函数来完成的。统计引用计数的过程获得当前 CFG 的_IR_instruction_list，然后对这个—_IR_instruction_list 中的每条指令从前往后对每个操作数：目标操作数、源操作数，调用 count_ref_opnd()进行计数。这个函数在具体的处理上首先会判断是否是 GC_Track 类型的操作数，如果是，在相应的优先级闭包中找到对应的索引，增加其引用次数，并且如果是需要记录使用情况的话，将会记录相应的使用块；否则，表明是一个复合类型，需要对其分解，分成基址和索引两个部分并递归调用 count_ref_opnd()过程。

然后，对引用计数进行排序，按照使用的频繁程度分配合适的寄存器。如果要编译的方法使用了浮点运算的话，需要在这儿对浮点部分的引用单独的作排序。在实现中，从第一步得到的引用数组中复制作为浮点引用的基础，然后调用 sort_fp_refcounts()：即对这个引用数组进行扫描，找出所有的浮点引用，使用快速排序对整个浮点引用数组进行排序。对应于通常的引用数组，直接调用 sort_refcounts。在实现上基本的处理和浮点部分的处理类似。

接着，对引用计数数组中的每个元素进行物理寄存器分配。这个过程中进行了物理寄存器的分配，在具体的实现上：

- 根据引用计数选择相应的分配集，对于引用计数大于 2 的选择 ALL_X86_CALLER_REGS 和 ALL_X86_CALLEE_REGS，而对于引用计数小于 2 的选择结果集和 ALL_X86_CALLER_REGS；

- 根据在基本块内的使用情况来获得活动区间，然后由分配集和每个基本块中可用的物理寄存器集进行与操作，得到真正的分配集；
- 如果分配集不等于 0，即表示分配成功，从低往高的扫描到底使用了那个物理寄存器，一旦成功，就把该寄存器分配给该元素，同时标记该物理寄存器在这些基本块中不再可用。
- 记录分配的操作数，并调用 Operand 的 set_assigned_preg()记录分配结果；把该寄存器加入结果集；

如果使用了浮点运算，这一步同样也要对每个浮点操作数选择浮点寄存器。在处理上，类似于上述的通用寄存器的分配，不同的在于选择浮点寄存器上。浮点寄存器是否可用是通过对每个基本块中的浮点寄存器集中是否有空闲的寄存器来进行的，这并不需要跟踪每个，只需要跟踪可用总数就可以了。选到了可用的浮点寄存器后，调用 set_globally_allocated_fp()进行分配。

最后，根据分配的结果作相应的更新活动操作数，由 update_liveness()函数来实现；作分配结果集的修正(全局寄存器分配只分配被调用者保存的寄存器)；为了防止溢出处理跨过 monitoexit 操作，必须对同步方法作将返回值保存在被调用者保存的寄存器中，这是由 keep_return_values_in_regs()函数完成的。

2. 简单分配算法（simple_reg_alloc）

和 O1 JIT 一样，O3 JIT 的简单寄存器分配算法的基本思想是将 4 个 callee-saved 寄存器分配给有最高静态引用计数的 4 个全局分配候选。

首先：调用 fg->apply(count_refs, &closure)对流图所有的临时变量统计引用计数。

然后：对所有寄存器做循环，如果有空闲寄存器，将它分配给最高引用计数的变量；如果没有，则退出并等待产生 spill 代码。

简单分配算法的复杂度为 O(B)，其中 B 为方法的字节码数目。由于分配的是 callee-saved 寄存器，我们要标记使用了哪些寄存器，这样在调用函数的 prolog/epilog 中才能插入 save/restore 指令。另外，如果函数体中没有调用指令，caller-saved 寄存器不会被破坏，那么我们还可以使用空闲的 caller-saved 寄存器做全局寄存器分配。在这个算法中，由于分配出去的寄存器使得所有 bb 中对应该寄存器的位都会被标记为 unavailable，而不是只让引用这个变量的 bb 中该位置 0，所以这个算法无法重用已分配过的寄存器。

3. 简单的溢出处理 simple_spill_code

这个过程中，首先通过深度遍历方法对控制流图中的各个节点建立数据流序，并将遍历过程中访问顺序记录到节点数组中，这是由 create_dataflow_ordering()完成的。然后基于遍历得到节点数组，逆序对各个节点调用 simple_spill_bb()来处理溢出代码。

simple_spill_bb()的处理上，会遍历整个基本块的指令，针对每条指令作寄存器分配。由于中间表示的种类比较多，针对类似的中间表示使用了相同的处理方法，如把 a op b 的操作统一作了处理，这里面包含了所有的二元操作如 a+b,a-b,a*b,a and b,a or b 等等。在实现上，把所有的中间表示分了 13 种情况，其中实际用到的有 9 种。下面我们来看一下 13 种中间表示：

1) a = b op c
2) a = b op [c,d]
3) a = [b,c] op d
4) [a,b] = c op d
5) a = b op [c]
6) a = [b] op c
7) [a] = b op c
8) a = b
9) a = [b,c]
10) [a,b] = c
11) a = [b]

12) [a] = b

13) deref b

这些中间表示是所有的参与寄存器分配的，实际使用中 3,4,5,6,7 的情况并不会发生。我们下面以第一种情况来看一下它的处理：

这种情况下都没有内存操作数，最好的情况是 a 和 b 相等，并且 a 或者 c 是物理寄存器，这种指令不需要为它分配其他的寄存器了。如果不是这种情况，就需要做分配。如果 b 已经分配了物理寄存器并且 b 的活动区域已经结束，那么就可以选择 b 所在的寄存器作为 a 的候选 r1；否则，如果 a 是在物理寄存器中并且 a 当前是可用的，就选择 a 所在的寄存器为 r1；否则，取得一个可用的寄存器 r1 来分配给这个指令；根据上面执行得到的寄存器候选执行 r1=b;r1=r1 op c;a=r1;这样就会这个中间表示完成了寄存器分配。在处理中，对于溢出的处理是由 get_available_register()来完成的。

get_available_register()内部，将基于当前的可用的寄存器情况作分配，如果有空闲的寄存器，就可以从中选择一个来使用；如果没有空闲的寄存器，就要选择某个操作数作为候选的溢出操作数以让出寄存器给其他指令使用。

在 ORP 内部的处理上，这部分是由 spill_register(unsigned &spilled, unsigned dont_spill, Inst *inst_head, Spill_Closure *sc, O3_Jit_Type ty,int bytereg)来完成的。它通过检查溢出候选来选择某个要溢出的寄存器。这可以溢出调用者保存的寄存器以及在程序的前奏(Prolog)中保存的任意被调用者保存寄存器，但不能溢出任一 dont_spill 中的内容以及已经溢出过的。同样的，如果需要使用一个字节寄存器的话，必须限定溢出某些特定的寄存器。选择溢出寄存器后，就可以根据上面的条件选择合适的溢出寄存器了。对于要溢出的寄存器，首先会为它分配一个溢出位置，这是一个虚拟寄存器(Virtual_Reg)，然后将分配的物理寄存器的溢出指针指向这个操作数。为了完成溢出，必须要把当前物理寄存器中的内容复制到溢出区，在使用这个溢出寄存器的指令前，需要为复制产生一个赋值语句，即目标为溢出区，源操作数为物理寄存器。产生了这些指令后，就可以把该寄存器分配给新的操作数使用了。

4. 代码调度（code schedule）

为了避免处理器执行时浪费机器周期等待操作数，我们可以调整指令执行顺序，使得指令本身固有的延迟被其他指令的执行隐藏。和寄存器分配一样，代码调度也是个NP完全问题。而且指令调度和寄存器分配也总是互相关联的。如果编译器调整指令减少执行时间，它往往会增加需要的寄存器数目。如果先分配寄存器，那么可以调整的指令数目又会受到限制。O3 JIT中的代码调整仅仅针对比较指令。我们从流图的底部往前找，找到操作数小于两个的简单比较语句，然后再往前寻找能够安放这个比较语句的最远的位置。测试能否提前则根据前面的指令执行会不会影响标志位，以及会不会改变比较指令中的源操作数来判断。这样对于compare-branch语句就减少了前后指令之间的依赖，在计算branch时compare的结果已经计算出来了。

10.2.7 代码发射

10.2.7.1 代码发射前的优化

为了让目标码尽量不包含冗余指令和一些可以被优化的结构，我们在代码发射之前作了一些优化。这些优化包括：移出流图空块，删除不可达的节点，死代码删除和 peephole 优化。

- 死代码删除

  复写传播、操作数折叠以及代码移动使得程序中产生了各种各样的死代码，我们要找到那些死代码并删除它们。而死代码的删除又会形成一些流图空块和不可达的节点，所以它们也需要从流图中删除。？？？？？？[elimgc_map backward analysis]

- Peephole 优化

  peephole 优化既可以针对中间表示，也可以针对目标码。它是用一组更快更短的指令序列

(peephole)取代原有序列的优化，有时甚至是用单条指令来取代程序中的多条指令。Peephole 就像是在程序上移动的一个小窗口。它的一次移动往往可以带来更多的优化机会。一般，为了得到最好的优化效果会对程序进行多遍扫描。O3 JIT 中的 peephole 优化包括：将"eax=[]; eax=eax+imm; []=eax"指令序列替换为"[]=[]+imm"；将"and eax, 0xff; movsx eax, al"序列中多余的 and 指令除去；将寄存器层叠赋值序列"mov reg1 reg2; mov reg2 reg3"替换为 mov reg1 reg3 等。

10.2.7.2 代码发射

O3 中所有的方法都基于 ESP 帧，栈帧指针可以不保存，所以 EBP 可以被释放做通用寄存器，从而缓解 IA32 寄存器少的压力，但这种使用会影响通常的栈上分配存储的寻址。O3 需要跟踪每一个对栈指针操作的指令，来调整偏址。如：在栈顶分配了一个 32 位的整型数据，如果没有对栈的操作，对它的引用将是[ESP]，但如果后续指令中有对栈的操作，如 push 指令、sub esp,IMM 指令，那么对于原来值的引用将会变成[ESP+4]、[ESP+IMM]。关于 ESP 帧的详细布局，可以参考第 9 章中的相关论述。

arch\ia32\ia32_o3_jit\code_emitter.cpp 中的 code_emission()方法实现实现了代码发射，它的基本过程为：每个 Java 方法生成相应的 prolog,，方法体，以及 epilog。下面是具体的处理过程：

1．为这个方法体创建 ESP 帧，然后基于创建的帧，发射对应方法的 prolog 和 epilog。这分别由 insert_prolog()和 insert_epilog()函数来完成的。

- insert_prolog 返回了 push 指令序列的第一条指令，这些指令是为了生成 GC 映像时使用的，它首先根据栈的额外区以及溢出区的大小来计算要预留的堆栈的大小(sub esp,counts)，生成代码对应的 IR 指令。然后根据使用情况把用到的调用者保存的寄存器的值(EBX,EBP,ESI,EDI)压到栈上，同样也是生成对应的 IR 指令。

```
sub   esp,n_extra+n_spill
push ebx
push ebp
push esi
push edi
```

- insert_epilog 返回了 pop 指令的第一条指令，同样也是为了在 GC 映像数据结构中使用。它的处理首先会弹出由被调用者保存的寄存器(EDI,ESI,EBP,EBX)的内容。然后清除 prolog 中预留的额外区域(add esp,counts)。

```
< ... call MonExit ...>              //  同步方法
pop edi
pop esi
pop ebp
pop ebx
add   esp, n_locals+n_spill_words
```

2．调用流图的 emit_code()函数为方法体发射代码。这个过程为方法生成 GC_Map 以纪录 gc 相关的类型信息，并对方法体内的每个基本块调用 emit_block()函数。emit_block()函数从_IR_instruction_list 中取得 IR 双向链表的头，然后对于每条 IR 指令 i 循环执行如下的操作：

- 检查是否是冗余指令，例如：eax=eax,test eax eax, eax=eax+0,eax=eax*1 等，如果是冗余指令，从列表中删除；
- 检查是否是第一个 push 指令，前面的 prolog 的生成中会记录这个信息，如果是的话在 GC 映像信息中记录第一个 push 的偏移；否则检查是否是第一个 pop 指令，同样前面的 epilog 发射中会记录这个信息，如果是的话，在 GC 映像信息中记录第一个 pop 的偏移；否则检查是否是返回指令，如果是的话，在 GC 映像信息中记录返回的偏移；
- 调用 Inst 的 emit_inst 方法（10.2.4.1）发射这条指令。emit_inst 调用自身的虚拟方法 emit，而每个从 Inst 继承的指令都会用自己的 emit 函数来发射对应本条指令的代码。关于代码发射部分更

多的细节，可以参考 arch\ia32\ia32_o3_jit\ir.cpp 中对应的各个指令的 emit 方法；

- 在 GC 映像信息中加入本条指令 gcmap->add_inst(i, pre_offset, xe.get_offset(), frame)；我们来看一下 gcmap->add_inst 的处理过程：

  根据需要调整 gc_unsafe 和 gc_esprecords 的大小计算新的 esp 的调整值，这取决于两方面的因素：

  - 一是本指令对 esp 的影响，这方面的指令有 push/pop/add esp,xx/sub esp,xx/call.详细的可以参考这些指令的成员方法：esp_effect.call 的处理上有点特别，对于 multianewarry 方法的调用是由调用者清栈的，因此对于这种调用 call 指令实际上对 esp 没有影响。
  - 另外是上一次的 esp 的调整值。如果当前指令是非调用者清栈的 call，那么新的 esp 调整值就是 0。因为当前指令对后续指令对堆栈的作用上没有影响。否则，新的调整值就应该是上一次的调整值减去当前指令对栈的影响。如果新的调整值和老的调整值不一样，就需要生成新的 gc_map 记录来记录信息。

  接着进入 add_inst_gc 的处理，这里面主要把引用的来源分成这三类：push 指令的引用，call 指令的引用以及目标操作数的引用，这也是引用所有可能出现的地方。具体的，首先会检查是否是引用类的 push 指令，如果是的话，栈顶元素就是活动的；否则，检查是否是 call 指令。call 指令内的处理检查了返回值的类型是否是引用类型，这就意味着 eax 在 call 调用发生后里面的内容是引用。同时检查调用方法的参数中有没有引用类型；否则，检查目标操作数，如果目标操作数是 JIT_TYPE_CLASS 类型，那么活动的操作数就是目标操作数。

  接下来，根据上面收集的信息开始修正，包括当前指令对活动引用的影响，记录了当前指令后活动引用在寄存器中的分布，栈的变化；根据这些信息，如果和前一条指令没有变化，则可以不用生成新的 gc 映像信息，否则需要创建新的 gc 映像信息，来反应这条指令执行完后引用在寄存器内及栈上的分布。这也是 ORP 中的 GC 几乎可以发生在任何一条指令的内在条件。
- 取下一条指令；

emit_block 接着会根据需要发射转移指令，记录代码长度信息，然后返回 emit_code()；此时，由于代码发射器已经将代码发射到某个内存区域，只需要将它复制到方法对应的代码块中；并且，发射与该方法相关联的 GC 映像信息以及方法信息；

3．调用 code_emission()函数，用这个方法的代码注册异常处理，这是通过调用 register_exceptions()方法来完成的。这个处理与 Java 中异常处理的方法类似，通过记录开始指令地址，结束指令地址，处理函数地址以及接受对象类型来注册异常处理的。

4．在上面我们对发射的代码进行了复制，因此需要为使用绝对跳转指令的地方来修改指令的偏移。先会修改对应于转移指令的偏移；然后修改跳转表。这是通过它们各自的 apply 方法来完成的。

10.3 静态优化

为了提高 class 文件的执行效率，O3 JIT 采取了各种常用的编译期优化方法。在生成中间表示到代码发射的过程中，我们已经遇到了很多。从这一节开始，我们将介绍 O3 JIT 应用的一些特定于 Java 虚拟机的优化。这一节介绍静态优化，即利用编译期间的分析减少运行期的开销。下一节介绍运行期进行的动态优化。

10.3.1 类的初始化

JVM 规范要求任何类在第一次使用之前都要被初始化。所以 JIT 需要确定：在使用类的一个非常数域或是对这个域赋值之前，类已经被初始化了。我们采用一种简单的方法消除对类初始化的检查。在为指令 getstatic，putstatic 建立 IR 时，我们调用 VM 提供的帮助函数 class_is_initialized，它可以检查这个域所在的类是否已被初始化，即类的 Class_State 是否为 ST_Initialized（见 4.3 类的初始化）。如果这个类在编译生成 IR 指令时未初始化，那么就只有在运行期间，调用 getstatic 或 putstatic 之前做检查。如果运行期的检查仍未通过，那么对这个域的访问就是第一次活动使用，我们必须先初始化这个类。

10.3.2 Checkcast 指令

JVM规范规定：类型转换如果无法在编译期间证明合法，将只能在运行时检查。O3 JIT为checkcast生成了一个运行期的帮助函数。我们可以使用两种方法来减少checkcast指令的运行期开销。

首先，我们通过局部公共子表达式消除来决定是否需要调用帮助函数。如果一个checkcast指令将一个类型为A的对象x转换为类型B，那么在为它建立IR时，我们检查在这一点是否可以得到"x instance of B"属性。如果它是表达式组中的一个局部公共子表达式，那么就不用产生checkcast的IR指令。下面的代码片断是O3 JIT为checkcast生成IR的片断（arch\ia32\ia32_o3_jit\Build_ir_routines.cpp）：

```
Exp *instance = exprs.lookup_inst_exp(Exp::Instanceof,to,src->exp,JIT_TYPE_INT);
if (exprs.is_local_cse(instance))
    stack.push(src);
else {
    Inst *cast_to = exprs.lookup_imm((unsigned)ch,JIT_TYPE_ADDR,inst_head);
    stack.push(exprs.lookup_inst(Exp::Checkcast,cast_to,src,JIT_TYPE_CLASS,inst_head));
}
```

为了实现 CSE 传播（见 10.2.3.2），我们只在 instanceof 用作控制流条件，即满足表达式组 br-test-instanceof 才传播 instanceof 信息。这样那些满足 instanceof 的后继节点中就得到了 instanceof 的局部公共子表达式。

其次，我们还可以部分内联帮助函数的公共执行路径，以消除这条路径上的运行期调用开销。

10.3.3 越界检查消除

在建立数组元素访问的IR时，我们要生成越界检查指令。Java语言规定所有的数组访问是在运行期作边界检查的，越界访问会产生ArrayIndexOutOfBounds的异常。如果编译器能证明访问索引总是在正确的范围内，那么就可以消除这些检查代码；如果不能证明，那么对数组的引用必须包含检查代码（一条无符号比较指令和一条分支跳转指令）。对于数组计算密集型的应用程序，如循环体中的数组运算，越界检查的代价是很高的。所以O3 JIT作了一些分析计算出循环中可能的数组访问范围。如果范围已知，那么编译器就生成克隆的循环并且消除所有越界检查代码，而在循环外部插入代码检查是否访问的上下界都在正确范围内 （如图10.8 (b)所示i, j和99）。如果任何一个检查失败了，那么就执行原来的带有越界检查代码的循环，以确定ArrayIndexOutOfBounds异常抛出的位置。

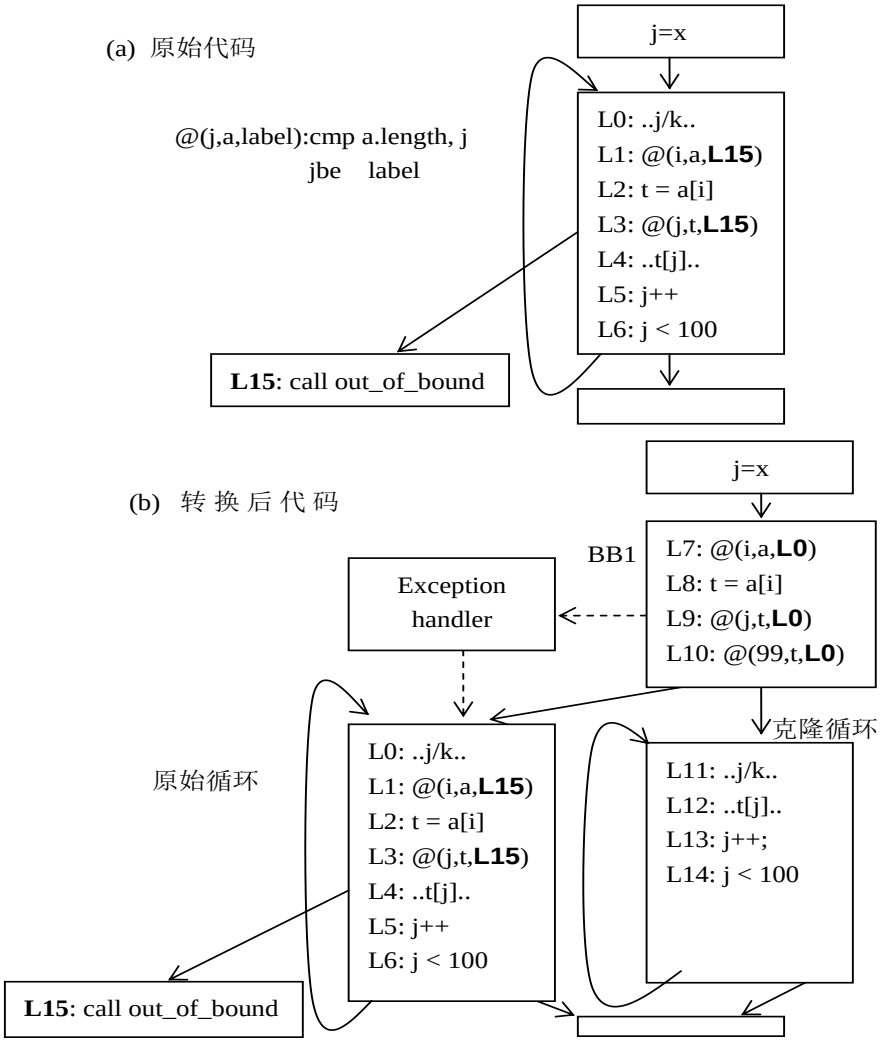


图10.9 越界检查的消除

我们用@(j,arry,label)表示包含两条指令的越界检查代码：“cmp arry.length, j” 比较索引值j和数组arry的长度， “jbe label” 如果大于等于arry.length，那么跳到label处。比较j和数组下界0没有必要，因为当j为负值，代码序列也会跳转到label处。编译器为新克隆的循环做代码提升（code hoist），将不变代码移出循环体（例如这里的t=a[i]）。

图中BB1中加入的代码可能会抛出异常，从而导致异常不能按正确顺序抛出。Java语言要求精确的异常处，即：异常抛出之前的所有执行语句和求值表达式都必须正确执行。为了达到这个要求，编译器不会在BB1中提升那些有副作用的代码，例如：数组store指令或putfield指令。这样即使BB1中发生异常，编译器也无需去恢复数组、域和变量原来的值。此外，图中创建了一个异常处理（其处理流程是原始循环）捕获可能在BB1中发生的异常。一旦捕获了任何异常，控制就发生转移（b 图所示）并执行原来的循环。执行原始循环后，异常就按照正确顺序抛出了。例如：假设循环第一次执行j/k 会抛出除0的算术异常，即原始循环的执行抛出了ArithmeticException异常；又假设BB1中的“@(i,a,L0)”也抛出NullPointerException异常。那么一旦异常处理捕获到这个异常，就去执行原有代码。这样保证了正确的异常实现。

克隆循环可以轻易的导致代码的膨胀，所以决定应用越界检查消除的策略很重要。O3 JIT中应用的策略基于三个参数：代码大小，可以消除的越界检查代码数目，profiling信息。O3 JIT只考虑最内层循环中的越界检查消除。对一个最内层循环来说，如果全部指令数目和候选数目的比率在特定阈值之下（这里为30），那么就不做这种优化。因为即使消除所有的越界检查代码也只能提高很少的效率。如果应用

了动态重编译，优化编译器还会使用O1 JIT产生的profile信息计算循环平均的trip count。如果trip count 小于10，那么由于效率提高不大也不做优化。下面的bound_checking_elimination函数片断体现了这些策略。

```
{   …
    Back_Edge be(mm);
    fg->apply(find_back_edges, &be);

    if (be.back_edge_count == 0) return;   //遍历 CFG 寻找回边并确定有循环存在

    //收集 O1 的 profiling 信息

    if (instrumenting) {
        Profile_Rec *prof = o1_method_get_profile_info(comp_env.m_handle);
        uint64 entry_count = (prof->m_policy == prof->m_entry) ? __UINT64_C(1) :
                                        prof->m_policy -   prof->m_entry;
        unsigned i;
        for (i = 0; i < prof->n_back_edge; i++) {

            //计算平均 trip count

            uint64 trip_cnt = (recompilation_policy_loop - prof->back_edge[i])/entry_count;
            if (trip_cnt > __UINT64_C(10)) break;
        }
        if (i == prof->n_back_edge) return;
    }
    Cfg_Node **tails = (Cfg_Node**)mm.alloc(be.back_edge_count*sizeof(Cfg_Node*));
    int i;
    for (i = 0; i < be.back_edge_count; i++)
    tails[i] = be.back_edge_tails[i];

    qsort(tails,be.back_edge_count,sizeof(Cfg_Node*),compare_depth); //按循环深度排列 tails

    int max_label = fg->reassign_label();

    //决定我们希望消除的 bound checking 和 inducion assignment 的最大数目，设 max_set_size = 64

    int max_set_size = 64;

    Bound_Set bound_set(mm, max_set_size); //为流分析创建 bound set

    //为数据流分析创建 bit vectors

    Loop_Bound_Closure lbc(mm, bound_set, be.back_edge_heads, max_set_size,
                                    max_label, be.back_edge_count);
    Bound_Elim bound(mm, exprs, fg, comp_env, lbc, gc_requires_write_barriers);

    bound.start(be, tails);   //找到最内层循环，如果值得做消除优化则移出越界检查表达式并创建克
```

隆循环

```
}
```

10.3.4 内联检测

内联是编译器经常采用的一种用于减少方法调用开销的技巧。内联扩大了编译范围，发掘更多的优化机会并减少了创建栈帧，传递参数和返回值的运行期开销。Java中的方法调用除非被声明为static, final或是special，缺省都是动态分派的virtual方法。虚方法调用无法轻易消除，因为编译器通常不知道究竟调用的是那个函数。图10.10是ORP中对象的布局，它对方法如何内联有直接影响。 X是指向一个对象的引用，对象的第一个域指向虚拟方法表。对于对象的每个虚方法，在vtable中都有一个专门的入口，指向方法的本地码。为了得到虚方法foo的地址，我们需要对指针做两次内存访问（dereference）。首先得到vtable (t = [x])，然后得到foo的地址(`[t+64]`)。

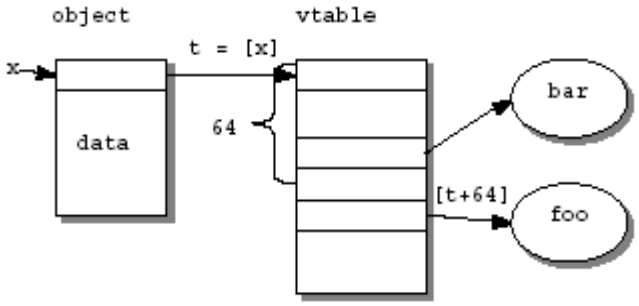


图10.10　对象布局

虚方法内联时，JIT生成运行期测试来验证内联的被调用函数是否是正确的调用实例。如果验证失败，则执行一般方法调用的代码序列。运行期测试通常有两种：一种是检查vtable；另一种是检查方法调用的实际目标地址。在10.4.1中，我们还会描述一些提高内联精确性的加强机制以及条件测试的代码序列。

10.3.4.1 检查 vtable

这种方法比较对象的vtable和内联方法的类的vtable。如果测试成功，就可以安全的执行内联代码，因为内联的方法就是将在运行期被动态分派的方法；如果测试失败，就执行常规的分派代码序列来调用虚方法。这种方法的代码序列如下。

```
mov eax, DWORD PTR [eax] //得到对象的 vtable

cmp eax, 0bc3508h
jnz _default_invocation

//内联的被调用函数

. . .
_default_invocation:

//一般调用代码

. .
```

检查vtable的方法只需要一次内存访问([x])就可以决定是执行内联的A类的foo方法还是执行常规的调用序列。原始调用的开销减少为一次内存访问，一次比较指令和一个分支。但是这种方法的缺点是检查vtable有局限性。考虑代码 x .foo( )。假定x可以是类A也可以是B，A是B的超类，A的foo方法也没有被B覆盖。如果x的动态类型总是类B，但是它的静态类型设为A（A x=new B()），那么检查就总会失败。因为对象类型为B，内联方法类为A，它们有着不同的VTables。上面的代码就总是会执行非内联的一般调用路径，即使每次实际上都是调用A中定义的foo方法。

10.3.4.2 检查目标地址

这种方法插入代码比较x.foo() 调用的实际方法地址和A的foo方法地址：

```
mov eax, DWORD PTR [eax] //得到对象的 vtable
mov ecx, DWORD PTR [eax+64] //方法的目标地址
cmp ecx, [BE762Ch] //和 A 的 foo 方法地址比较
jnz _default_invocation
//内联的被调用函数
. . .
_default_invocation:
//一般调用代码
…
```

这种方法更精确，只是至少需要两次内存访问([x]和[t+64])。而且在即时编译中，一个方法只在第一次执行前才被编译。所以如果A的foo还没有被编译，我们就不知道A的foo的实际地址。那么编译器必须分配内存空间，一旦A的foo被编译就向其中放入A的foo的地址。这时测试就需要做3次内存访问。

O3 JIT使用第一种方法，因为它只需要一次方法访问。只是我们需要解决这种方法的局限性。我们实现了两种机制来减轻这个问题：类型传递和10.4.1中的动态内联补丁。类型返回信息用于提高内联估测的正确性。我们在复写传播中将类型信息传播，以记录调用方法的对象的实际类类型。例如x.foo()，如果类型传播证明x的实际类型是类B，而不是类A，而类的继承分析又能证明A和B之间没有类覆盖foo方法，那么我们就可以产生比较x的vtable和B的vtable的测试。

10.4 动态优化

O3 JIT有一些在运行期间应用的优化方法。包括：修补本地码，压缩GC信息大小，为栈帧建立缓存和避免创建异常对象（lazy exception）等。

10.4.1 动态内联补丁(dynamic patching)

通常，一个方法从被JIT内联到程序执行完是不会被覆盖的。只要类继承关系不改变，内联的方法就始终是正确的调用实例。所以我们希望假设内联方法在程序执行过程中从不被覆盖。但是Java是允许类在运行期间动态加载的，也就是说类的继承关系可能改变。当假设不成立时，我们应用动态修补（dynamic patching）技术修正本地码，以保证程序的正确性。

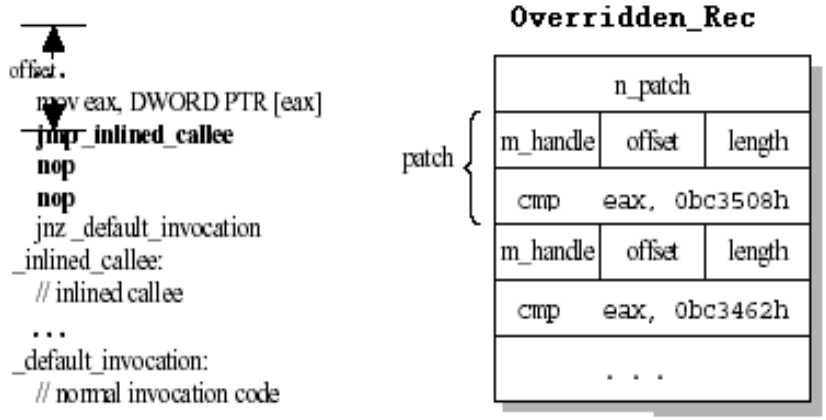


图10.11 动态内联补丁

我们已经知道O3 JIT产生的内联代码序列。在代码发射之前，编译器用一个jmp指令取代cmp，直接跳到内联代码（如上图所示），如果内联方法此时还没有被覆盖。条件测试的开销（一个cmp和一个jnz）就被减少为一个直接跳转指令。我们为刚刚取代的cmp指令创建一个补丁入口。这个补丁包含被调用的内联方法的method_handle， 代码偏移(offset) ，IA32 的cmp指令长度， 以及存储cmp指令的字节数组。因为cmp指令的长度比jmp长，所以替换之后填充了一些nop指令。然后编译器通过一个API调用method_set_inline_assumption (caller,callee)来通知VM：调用函数（caller）已经内联了被调用函数（callee）。我们为调用方法创建Overridden_rec作为它的部分method_info，这个结构包含了调用者所有的补丁以及补丁总数。优化编译器还提供了一个回调接口函数method_was_ overridden(caller,callee)，允许VM通知编译器被调用者已经被覆盖了，因此需要修改调用者代码。method_was_ overridden方法找到调用者的overridden_rec并为所有method_handle与被调用者一致的patch做修改。

代码修补必须是基于线程安全的（thread-safe），因为其他线程可能正在执行我们将要patch的指令。例如：直接跳转指令。编译器通过三步实现代码补丁的线程安全（如图10.11所示）：

```
Byte *code = method_get_code_block_addr(caller);
Byte *first_byte_addr = code + patch->code_offset;
// 第一步：自旋跳转指令
__asm {
mov eax, first_byte_addr
mov cx, 0xFEEB // spinning
xchg word ptr [eax], cx
}
// 第二步：恢复 cmp 指令（除了头两个字节）
for (int j = 2; j < patch->length; j++)
code[patch->offset + j] = patch->orig_code[j];
// 第三步：恢复头两个字节
Byte *first_orig_addr = (Byte *)patch->orig_code;
__asm {
mov eax, first_byte_addr
mov edx, first_orig_addr
mov cx, word ptr [edx]
xchg word ptr [eax], cx
}
```

图 10.12 代码补丁

1．图10.10中的直接跳转到的内联方法由于现在已经被覆盖，因此需要修改。但要保证在代码修改的

过程中不会有其他代码执行到了修改的代码。因此，一开始修改方法就会把一个直接转移指令改写成跳转到自身的指令L1:jump L1。这样就保证了其他处理器/线程如果执行这个方法的话，它会在L1处自旋跳转（spinning jump）。接下来就是该怎样修改了。因为mov指令不是原子操作，它只能够保证局部的修改局部可见。即如果P0修改了内存位置R, 发生在P0修改之后的P1读R有可能读到旧的值。为了避免这种情况发生，可以通过使用LOCK前缀的指令来保证修改的全局可见性。即指令LOCK:MOV R11,[XX]。但是IA32中XCHG本身的实现就有这种原子性。也就是说XCHG R,[M]或XCHG [M],R指令，对内存M的修改和有LOCK前缀的指令一样,这就是所谓的LOCKING PROTOCOL。这里0xfeeb是 L1:jmp L1的机器码在内存中的分布(0xeb 0xfe)：0xeb是IA32中无条件短跳转，后面紧跟的一个字节是目标地址和下一条指令之间的偏移量。这里是0xfe就是-2。因此会跳转到下一个指令的前两个字节的地方，即仍旧是jmp指令。

2. 第二步修改了0xfeeb的指令后的部分。原始cmp指令除了最开始的两个字节都被恢复。在修改后面的字节时，如果有其他的线程执行这段内联代码，都会在那儿自旋。注意这儿的cmp register, [address]的指令长度是5个字节的，所以需要修改3个字节。
3. 修改完了后面的三个字节，就可以用xchg修改前面的0xfeeb了。所以cmp最开始的两个字节又被xchg自动重写。

10.4.2 其他

除了内联代码动态修补，O3 JIT 还实现了其他一些动态优化方法，包括：压缩 gc 信息、异常的 lazy 创建以及建立回退过程的缓存等。这些优化方法同时也是 O3 JIT 对垃圾收集和异常处理的特定支持，所以我们将在 10.5 和 10.6 节中分别介绍。

10.5 GC 的运行期支持

我们已经知道VM 和JIT之间的接口分为编译期和运行期。编译期接口包括由VM使用的，如为编译方法调用JIT；由JIT使用的，如得到编译方法所需要的结构信息并通知VM编译方法的异常表信息。运行期接口包括由VM使用的，要求JIT完成的一些JIT相关任务：如GC或是异常发生时的栈帧回退，根集枚举等；由JIT产生的代码使用的，如一些帮助例程。

JIT和GC之间的关联是JIT和普通编译器的区别，也是实现高性能VM的关键。在核心虚拟机要求JIT实现的接口函数中，虚方法can_enumerate()判断当前执行的代码是否处于GC安全点；虚方法get_root_set_from_stack_frame()为GC计算当前栈上所有引用的根集，无破坏的回退栈帧并修改调用上下文。

- can_enumerate
  从 5.1 节我们知道：当 GC 状态为 gc_moving_to_safepoint（即需要向 GC 安全点靠近）时，核心 VM 调用暂停线程的操作，然后根据暂停线程当前的 EIP 地址来检查是否处于 Java 代码中，如果是的话，就要根据这部分代码对应的 JIT 检查当前执行点是否是 GC 安全的。如果检查结果是 GC 安全的（在 O3 JIT 中实现了每条指令的 GC 安全），那么就可以调用 get_root_set_from_stack_frame 进行枚举。
- get_root_set_from_stack_frame
  当 Java 堆被耗尽，GC 发生时，编译器必须报告垃圾收集器此时所有物理寄存器中和栈上包含的活对象引用，它们是由全局指针（类的静态域）或是活动栈帧中的指针指向的对象集合，即根集。一般，只有编译器能够决定活动栈帧中的活引用。在 10.2.7 代码发射一节，我们已经知道 O3 JIT 在编译方法时为每个方法产生了一个辅助信息表 GC_Map，它将编译代码的每个点对应到包含活引用的寄存器和栈位置。然后垃圾收集器就可以回退所有栈帧并使用它们的 GC_Map 建立完整的根集。

在单线程应用中，GC 通常发生在对 new 函数的调用时。因而在所有栈帧中，IP 都是指在一个调用位置。因此 GC_Map 只要为调用位置计算根集，只要调用指令是 GC-safe 就可以了。但是对于多线程应

用来说，某个线程耗尽堆时，其他线程可以在任何可能不是通常的 GC-safe 的地方，它们也必须停下来产生它们的根集。通常的解决是让线程继续执行到达 GC-safe 位置。为了提高多线程应用中 GC 的性能，O3 JIT 的实验设计可以使得除了在一些特殊的指令处，每条指令都是 GC-safe 点。而且通过压缩技术，所有的 GC_Map 只是所产生机器指令的 20%左右。

10.5.1 收集 GC 信息

O3 JIT 通过对 IR 做两遍数据流分析来收集最初的 GC 信息。第一遍为正向，从前往后为每个寄存器和栈位置赋值类型信息 （实际上这遍类型信息的赋值是隐式的，因为类信息在建立 IR 时被赋值）。第二遍为反向，从后向前计算 liveness 信息（只有活对象引用需要被包含在根集中，死对象不用报告）。由于我们的 IR 和本地指令一一对应，所以为每条 GC-safe 的本地码产生 GC_Map 也被简化了。

为了为每条指令提供 GC 支持，GC_Map 必须提供为每条指令提供两种信息。首先，该指令处包含活引用的寄存器和栈位置的集合；其次，该指令对栈的调整（adjustment）。所谓栈的调整值是当前栈指针回到原来值所必须被调整的值。例如：一个线程恰好在它为某个函数调用压入两个参数后停止了，那么栈指针需要被回调两个 word，回到原来的值。另外，如果线程在 prolog 或 epilog 中被停下来，此时新的栈帧正在创建或是正在被破坏，那么栈指针可能也需要被回复成原来的值。如果编译器保留了栈帧指针，那么这种调整就可能不需要，但是由于 IA32 寄存器资源有限，在 O3 JIT 中会释放栈帧指针用作他用。总之，对于每条指令，GC map 必须记录三项：在这条指令中哪个寄存器会改变 liveness，哪个栈位置会改变 liveness，以及指令对栈指针的影响。

在 arch\ia32\ia32_o3_jit 目录下的 Gc_eh_support.h 中定义了每个方法的 GC_Map 和对应方法基本块的 GC_Map_BB 结构。

```
class GC_Map
    {
    public:
        GC_Map(Mem_Manager &m, Flow_Graph *fg, unsigned numvars):
            mem(m), fg(fg), bbcount(0),_gc_unsafe_size(0), _gc_unsafe_capacity(0),
            largest_inst_length(0),_tmp_lr_array((unsigned *)m.alloc(numvars * sizeof(unsigned))),
           _first_push_offset((unsigned)-1), _first_pop_offset((unsigned)-1), _return_offset((unsigned)-1) {}
        static void fix_handler_context(Method_Handle meth, Frame_Context *context, Boolean isFirst);

        // GC 发生时，O3 JIT 栈帧回退实际的实现函数

        static void unwind_stack_frame(Method_Handle meth, Frame_Context *context,    Boolean isFirst );

        //O3 JIT 的 get_root_set_from_stack_frame 函数实际的实现函数

        static void get_root_set_from_stack_frame()
        static uint32 get_address_of_this()
        static Boolean call_returns_a_reference();

        static Boolean can_enumerate(Method_Handle method, uint32 eip); //对这个函数的调用 O3 JIT 总
是返回为真

    private:
        Mem_Manager &mem;

        Flow_Graph *fg;          //当前方法流图

        GC_Map_BB bbhead;    //构成 GC_Map 的基本单位——每个 bb 的 gc 信息构成的双向链表
```

```
    unsigned bbcount;          //bb 的个数

    unsigned _first_push_offset, _first_pop_offset, _return_offset; //第一个 push 的偏移，第一个 pop 的
偏移，以及返回指令的偏移

    int _gc_unsafe_size, _gc_unsafe_capacity;
    unsigned *_gc_unsafe_offsets;
    static void unwind();
    static unsigned enumerate();
    unsigned *_tmp_lr_array;

    void add_inst(Inst *inst, unsigned pre_offset, unsigned emitter_offset, Frame &frame);//向 GC_Map
中添加指令
};

class GC_Map_BB : public Dlink
{
    friend class GC_Map;
public:
    GC_Map_BB() {}
    void *operator new(size_t sz, Mem_Manager& m) { return m.alloc(sz); }

    GC_Map_BB *next() {return (GC_Map_BB *)_next;} //下一个 GC_Map_BB

    GC_Map_BB *prev() {return (GC_Map_BB *)_prev;} //前一个 GC_Map_BB

    enum GC_ref_change_op
    {
        lr_ends,
        lr_starts,
        call,
        none
    };

    struct unified_gc_esp_rec          //统一的 esp 栈帧对应的 gc 信息

    {

        unsigned inst_length;          //指令的字节长度

        unsigned char live_ref_regs;   //包含活引用的寄存器的 snapshot

        char live_ref_change;          //转换成 GC_ref_change_op

        char top_of_stack_live;
        unsigned stack_change_index;
```

```
        int esp_adjustment;                    //指令结束时对 esp 的调整

        void init(unsigned inst_len) {
            inst_length = inst_len;
            live_ref_regs = 0;
            top_of_stack_live = 0;
            live_ref_change = none;
        }
    };
private:
    unsigned initial_emitter_offset;

    unsigned char gc_initial_live_ref_regs;   //在 bb 入口的活动引用寄存器的位图

    unsigned char gc_current_live_ref_regs;

    unsigned gc_num_initial_live_ref_stack; //在 bb 入口的活动引用栈位置的数目

    int gc_live_ref_esp_offsets_size, gc_live_ref_esp_offsets_capacity;
    int *gc_live_ref_esp_offsets;
    int gc_esp_rec_size, gc_esp_rec_capacity;
    struct unified_gc_esp_rec *gc_esp_records;

    unsigned current_emitter_offset;          //纪录 jsr 的返回地址

    int esp_adjustment_bytes;
    int esp_last_adjustment;
    void add_inst_gc(Inst *inst, unsigned emitter_offset, Mem_Manager &mem, Frame &frame,
                     GC_Map *map, bool &created_new_record);
};
```

从 GC_Map_BB 结构中我们看到了记录的 GC 基本信息包括：

- **指令长度：**IA32中的指令由于指令和寻址方式的不同长度也不同。我们必须要记录每条指令的长度。由于大部分指令对应单条指令，所以我们可以先用一个bit指明，并在运行时使用反汇编获得这条指令的长度。但是即使使用了这种压缩技巧，指令长度仍然占到整个GC map的10%左右。
- **指令作用结果：**指令对寄存器，栈位置以及栈指针的状态都可能有影响。当寄存器指向的引用的活动范围开始或结束时，寄存器的状态就会改变；类似的，包含引用的栈位置的活动范围开始或结束，栈位置的状态就会改变；参数压栈或出栈，方法被调用或是在prolog/epilog中constructed/destroyed栈帧时，栈指针的状态就发生了改变。
- **寄存器的变化：**如果一条指令开始或结束某个包含引用的物理寄存器的活动范围，那么记录被这条指令改变的寄存器。大部分寄存器的变化都涉及eax，这是因为按照调用约定，当方法返回一个引用时，通常由eax保存。
- **栈变化的类别：**指令改变栈顶指针，如push, pop, call指令，或是在prolog/epilog中create/ destroy栈帧。记录栈指针的改变有两个原因。首先：当栈位置包含活引用，我们必须标记这个位置相对栈顶指针的偏移，所以我们必须确定在报告活动的栈位置时，栈指针的值是正确的。其次，当我们回退到调用栈帧时，我们必须保持栈指针和JVM的栈回退约定一致。

下面的几个例子说明了GC map的组成：

- mov eax [esp+8]　假设[esp+8]的栈位置处包含了一个引用，并且这条指令是这个栈位置的最后

一次使用。这条指令影响了寄存器的状态，现在eax包含了一个引用，开始了eax的活动范围。此外，该栈位置的活动范围结束了。该指令中没有栈指针的改变。

- push ecx　　假设ecx包含一个非引用。它的活动范围可能在这条指令结束，但是我们不会在GC_Map中记录它，因为它不是一个引用。此时只有栈指针发生改变，我们记录为压入一个非引用。
- push [esp+8]　假设[esp+8]包含一个引用，并且是它的最后一次使用。这条指令不影响寄存器状态。它不影响栈的指针（push of a reference），但影响了栈位置，因为 [esp+8]活动范围结束了。
- mov eax, [ebx+8]　这是getfield字节码通常产生的指令代码。假设这是一个得到引用的getfield，并且是ebx的最后一次使用，它包含getfield的基指针。这时，对栈指针或栈位置都没有影响，但是寄存器状态发生了变化。Eax的活动范围开始，ebx活动范围结束。这个单条指令带来多个寄存器状态的变化。

10.5.2 GC 信息压缩和缓存

为每个方法生成 GC_Map 是很大的开销，并且它的大小也最好控制在几十字节以内。在 O1 JIT 中为了减小 GC_Map 的开销采取了一种 lazy 的方法，它只在需要时才产生活动方法的 GC 信息。在 O3 JIT 中，我们使用了简单的压缩技术，主要是 Huffman 编码，这样即使为每条指令提供 gc 支持，产生的 GC map 也只是代码大小的 20%。

压缩技术分为两层。在高层我们利用一个事实：每条指令对栈指针和活引用的影响是很小的。因此我们在每个 bb 开始处编码(emit_block 函数中)，建立活引用和栈调整的快照（snapshot），并且只去记录每条指令对原先状态的改变。这种改变可以是：一个栈引用活动范围的开始或结束；一个寄存器引用活动范围的开始或结束，影响栈指针的一个操作 （例如 push 或是调用指令）。对于 push 操作，我们还需要记录是否压入了引用，因为接下来栈顶就构成了该指令部分的根集。

在底层，我们使用 Huffman 编码压缩每条指令带来的变化。我们使用“顺序字节流”结构将这些数据写入内存。有时指令对原来的根集或是栈调整没有影响。例如：增加一个整型变量的值，或者加载一个非引用域到寄存器或是某个栈位置。我们可以将所有这些指令看作一大块宏指令。类似的，如果两个相邻块，第一块结束状态和第二块开始状态如果相同，那么就不需要包含第二块的 snapshot，而是将两块看作单个块。

基本块索引和GC缓存 由于编码的原因，解码也需要从最开始处逐条处理GC map直到到达目标指令。如果GC map经常访问，处理也会很昂贵。为了加快访问，一种解决方法是在map中嵌入一个索引，允许对任何给定基本块的开始直接访问。另一种方法是实现一个小缓存，将指令地址映射到根集。这是根据一个事实，即较深的栈帧通常都会在收集过程中保持不变。

10.5.3 Write barrier 和 GC-unsafe 指令

我们知道如果向对象的某个域中存储了一个JIT_TYPE_CLASS的引用后（例如：putfield指令，astore系列数组存贮指令和putstatic指令），O3 JIT产生的IR指令必须调用write barrier方法。某些和write barrier相关的指令，即使我们保存了完整的根集信息也不能允许GC发生，这就是所谓的gc_unsafe指令。

在分代的或渐增式的垃圾收集器中，GC需要知道什么时候年老对象指向了年轻对象。在我们的实现中，首先存贮这个指针，然后将这两个对象传递给write barrier函数， 如果GC发生在存贮和write barrier调用之间，那么GC的状态很可能就会发生错误，从而使得它错误地收集了活对象。一种解决方法是在store之前和之后都产生一个write barrier，这样使得运行期开销就是双倍的了。也有一些方法可以避免这种开销，但是有一点是共同的：GC map都必须能够识别这些GC-unsafe指令。

10.5.4 JSR 问题

JSR 问题可以说是 Java 类型安全唯一的例外。JSR/RET 定义了一个内部于方法的子程序（subroutine）调用，但不用创建新的 Java 栈帧。JVM 规范规定：如果在子程序中不对变量做引用，那么从不同路径

进入 finally 块中的同一变量就可以是不同的类型。在一般的情况下，这种规定并不会违反类型安全，但是如果在子程序中发生了垃圾收集，变量的一种类型是引用类型，另一种是非引用类型，而编译器无法静态决定确切的类型，那么隐含的安全问题就被暴露了。

Agens,Deltlefs 等首先提出了这个问题并通过重写字节码解决了这个问题。在早期的快速代码生成器中采用了为不确定类型保持一个类型标签以指明当前类型的方法。但是这就需要在方法入口初始化位图，如果向变量赋值，还要更新变量。所以运行开销非常大。

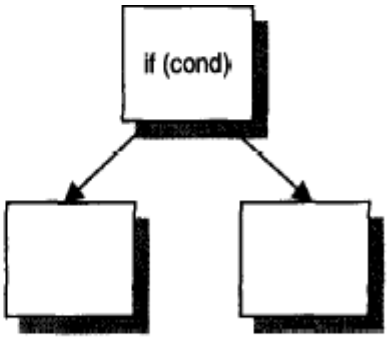
O3 JIT 采用了一种新的方法。这种方法分为两部分，首先是编译期的分析，在编译时我们决定在每个基本块中有哪些变量类型是未知的，并记录 JSR 的返回地址。接下来是运行期分析，我们在运行时恢复未知类型变量的确切类型。

10.5.4.1 编译期分析

在10.5.1节我们已经提到了收集liveness信息的两遍数据流分析。现在我们对第二遍反向分析作扩充以解决JSR问题。首先：将活动变量分为四种——活动引用，活动非引用，活动未知类型，死对象。在Bit_Vector_group.h中分别定义为Bit_Vector *_refs, *_nonrefs, *_dead; Bit_Vector *_unknown。其次：根据不同流图结构合并后继节点的liveness集合。合并的种类为enum MergeType { merge_jsr, merge_ret, merge_default }。下面我们将一一介绍三种情况下计算liveness的算法。为了说明问题方便，我们分别定义Rn,Nn,Un和Dn为基本块n中的活引用，非引用，未知类型和死对象集合。它们的并集为函数中所有用到的变量。我们还定义Succ(n)为CFG中n的后继节点集合。

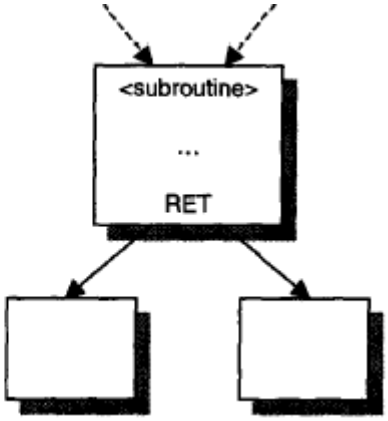
- merge_default　　缺省的情况下，死变量是在所有后继中都是死的变量。活引用变量在所有后继节点中可能为活动的，也可能为死变量（但是至少会在一个后继中为活动的）。活动的非引用变量也是同样。活动的未知变量是所有其他的变量集合。对于这种合并情况，JVM规范规定：一个变量如果在一个后继开始处是活动引用，但在另一个后继开始处是活动非引用，那么这就是不合法的。类似的，一个变量也不可以在一个后继中是活动未知类型，但在另一个后继中是活动引用或是非引用。

$$D_n = \cap_{s \in succ(n)} D_s$$
$$R_n = (\cap_{s \in succ(n)} R_s + D_s) - D_n$$
$$N_n = (\cap_{s \in succ(n)} N_s + D_s) - D_n$$
$$U_n = \text{all} - (R_n \cup N_n \cup D_n)$$



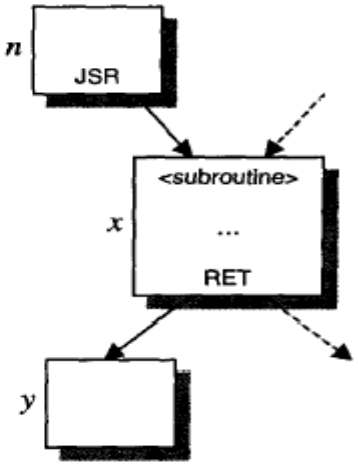
- merge_ret　　如果一个变量是死对象，活动引用，活动非引用，那么它必须在所有的后继节点中类型一致，否则它就是活动未知类型。注意：由于 JVM 规范规定，这种间接分支所有可能的目标都是在编译时静态可知的。

$$D_n = \cap_{s \in succ(n)} D_s$$
$$R_n = \cap_{s \in succ(n)} R_s$$
$$N_n = \cap_{s \in succ(n)} N_s$$
$$U_n = \text{all} - (R_n \cup N_n \cup D_n)$$



- merge_jsr　　当一个基本块是以一条JSR指令结尾的，它在CFG中就有两个后继：子程序的开始节点（图中节点x），以及紧跟JSR指令的代码（图中节点y）。节点x是显式的CFG后继，而节点y是隐式的后继。如果一个变量在子程序的开始处是一个活动未知类型，那么它是不可能在子程序中被访问的（JVM规范）。因此，在JSR指令处的变量类型必须和在紧随JSR之后的指令中的类型相同。另外，如果变量在子程序的开始处类型确定，那么这个类型就被无改变的传播到JSR指令处。

$$D_n = D_x \cup (U_x \cap D_y)$$
$$R_n = R_x \cup (U_x \cap R_y)$$
$$N_n = N_x \cup (U_x \cap N_y)$$
$$U_n = U_x \cap U_y$$



10.5.4.2 运行期恢复

现在再对方法编译时，GC_Map不仅会记录每个基本块开始处的liveness信息（哪个变量包含了活引用，哪个变量包含了活动未知类型）。而且，当基本块包含在一个子程序中时，编译器还纪录了在哪里可以找到子程序的返回地址。当运行期间GC发生时，编译器对GC_Map解码并决定此时哪个变量包含活引用，哪个变量包含活动未知类型。如果是活动未知类型，那么编译器还必须判断它是否包含活引用。对于活动未知类型的变量，编译器首先定位对应子程序的返回地址的基本块，然后查看在那个基本块开始处的活动引用和活动未知类型的变量集合。如果变量在子程序中是未知类型，并且在子程序外部是活动引用的，那么就被记作一个活动引用；如果变量在子程序中是未知类型，并且在子程序外部既不是未知类型，也不是活动引用，那么变量就不被记作活动引用；如果变量在子程序内部和外部都是未知类型，那么子程序必定是被嵌套的，编译器继续这种递归分析，直到不再有这种包含在任何子程序中的基本块。

10.6 O3 JIT 中的异常处理

10.6.1 异常处理模型

在核心虚拟机的介绍中，我们已经知道了基本的异常处理。为了说明问题方便，我们仍然给出一个

抛出异常的 Java 应用程序：

```
class sum {
   public static void main(String args[]) {
   sum i = new sum();
   i.printSumOfInts(args);
   }
   public void printSumOfInts(String strArr[]) {
      try {
         int result = sumOfIntsAsStrings(strArr);
         System.out.println("The sum is " + result);
      } catch(NumberFormatException e) {
         System.out.println("Error!");
      }
   }
   public synchronized int sumOfIntsAsStrings(String strArr[]) {
      int sum = 0;
      for(int i = 0; i < strArr.length; i++)
      sum += Integer.parseInt(strArr[i]);
      return sum;
   }
}
```

图 10.13 一个 Java 应用

这个程序把命令行参数看作整数值并打印它们的和。异常机制被用来处理无效的参数。如果发现无效参数，就打印一个错误信息。为了说明更多问题，我们声明一个方法是同步的，但实际上在这个单线程程序中没有必要作同步。如果参数无效，库方法Integer.parseInt就创建并抛出NumberFormatException。然后异常被声明在printSumOfInts中的处理捕获并处理。这里要注意两点：

1. sumOfIntsAsStrings方法是同步的。这意味着进入方法中，也就进入了和this对象相关的monitor，退出方法时，也要释放这个monitor。如果抛出异常，VM必须确定monitor作为异常抛出过程的一部分被释放。

2.一个异常对象包含了栈路径（stack trace），它可以在任意点被打印，即使在路径中引用的一些栈帧已经不存在了。这里我们可以在异常处理中加入调用语句e.printStackTrace()，则产生下面的输出：

```
java.lang.NumberFormatException: badnumber
at java.lang.Integer.parseInt
at sum.sumOfIntsAsStrings
at sum.printSumOfInts
at sum.main
```

JVM规范中按照异常的起因把它们分成三组，分别是VM同步测试到的异常执行条件；执行throw语句产生的异常；异步异常。我们把第二种异常叫做用户异常，因为它们可以从用户代码中创建并抛出，这种异常的lazy抛出我们将详细讨论。与之相对的其他两种异常叫做VM异常，它们是VM 测试并抛出的。

10.6.2 unwind 过程的数据压缩和缓存

在 10.6 例中，抛出异常时栈上有 4 个帧。最底部的是 sum.main，然后是 printSumOfInts，sumOfIntsAsStrings 和 Integer.parseInt。当前活动帧对应方法 Integer.parseInt 并在这个执行点抛出了一个 NumberFormatException 异常。调用栈上的任何方法都可以注册一个和 NumberFormatException 类型兼容的处理。这里只有一个声明在 printSumOfInts 中的异常处理。

一个简单VM实现一旦测试到合适条件就会创建一个类型为NumberFormatException的异常对象。作

为异常对象创建的一部分，将会调用对象的构造函数。构造函数会创建栈路径并把它存贮在异常对象中。栈路径的构造是对栈的无破坏遍历，它包含了从栈顶到栈底的所有帧。

对象创建后，栈从顶部的活动帧开始被破坏性的遍历，直到遇到一个合适的异常处理或是到达栈的底部。无破坏性（又称只读）回退和破坏性的栈回退的区别在于：后一种情况下，当我们回退到前一个帧时也同时释放了各种VM资源。当一个帧被破坏性的回退之后，就再也无法在那种上下文中执行了。这些被释放的资源包括同步方法的Java monitor以及VM内部数据结构。在上例中，破坏性的回退会退出和同步方法sumOfIntsAsStrings 相关的monitor。

回退过程从一个线程的上下文开始并决定调用者的上下文。如果活动栈帧是JIT编译的一个Java方法，那么VM调用JIT运行期支持接口做回退工作。这样就允许JIT编译器使用的帧布局有极大的灵活性，因为对于VM 来说帧结构是不可见的，所以新的JIT编译器插入到核心VM 中无需对VM做任何调整。如果活动栈帧不是JIT编译器编译的Java方法，那么VM就使用它的内部数据结构找到调用者的上下文。这种情况通常发生在native方法中。在核心VM一节，我们知道VM会维持充分的状态信息，使得对native方法的回退成为可能。即使除了跟随一个如JNI本地接口外，再没有其他的本地码支持环境。

栈的遍历通过一个循环得到实现，循环从栈顶帧上下文开始，到栈底（无破坏的回退）或是到正确的帧（破坏性的回退）处结束。最重要的操作是方法查找和栈帧回退函数。

- Method_Lookup_Table:: find函数取IP作参数并返回一个指向JIT_Specific_Info的指针，这个结构表示了方法和生成方法代码的JIT编译器的信息。这个结构在核心VM中介绍过。每个Java方法编译成一段连续的内存区域。假设我们把查找表（lookup table）组织成分类的IP范围 (start_IP..end_IP) 的数组，那么对方法信息的查找就是对表做二分查找。
- 栈的回退过程使用JIT编译时产生的信息来回退栈帧，直到caller的栈帧。这个过程必须重新存贮IP值，栈指针以及callee-saved 寄存器。由于Java方法可能是被重编译的，可能同时存在同一个方法被不同编译器编译的帧。所以栈遍历循环必须使用合适的编译器来回退每个帧。这是通过方法查找返回的JIT_Specific_Info信息得到的（jit_info->get_jit()->unwind_stack_frame）。VM使用IP寄存器的值决定是否存在为这个IP登记的异常处理，如果有，则判断是否抛出异常类型是否符合异常表中记录的捕获类型。

我们通过使用缓存减少相同栈帧的复制工作来加速栈的遍历。这种优化依赖一个事实：通常栈上有足够长的栈帧，可以被多个异常分派的异常代码遍历。我们使用两个缓存，一个加快方法查找（method lookup），另一个用于加快回退过程自身。

方法查找缓存是一个直接的映射缓存，通过IP值计算的一个hash 值做索引。如果缓存中的IP值符合当前值，结果就会立即返回，否则做二分查找，在返回结果之前更新cache 入口。在我们的VM中使用了512个入口的cache，这个大小足够用于比较大的应用程序了。

回退缓存由JIT维持。在我们的实现中，核心VM 不知道对应编译器编译的方法的栈帧布局。编译器负责创建适当信息使得回退过程能够实现。一个指向合适的回退数据结构的指针保存在JIT_Specific_Info结构中。我们的设计要求JIT编译器提供一个运行期的函数，给定寄存器上下文和一个指向编译时创建的unwind data的指针，就能更新调用者的上下文。这包括恢复栈指针(SP)，IP 和callee-saved 寄存器。栈回退函数的设计要满足两个目标： 首先：回退 的数据结构必须尽可能的小，在10.5中介绍了如何减小GC信息的大小。相同的数据结构既可以包含GC信息也可以包含栈回退所需要的信息，因为根集枚举和栈回退函数有很多类似的地方。其次：回退过程应该越快越好。除了实现有效的压缩，我们通过在编译器中使用一个缓存来加速回退过程。这个回退缓存使得对应cached IP值的栈帧可以更快的回退。

10.6.3 Lazy 异常处理

Lazy 异常处理是 O3 JIT 动态优化的一种策略。一些 Java 应用程序使用异常只是为了改变控制流。在这些应用中，异常对象的类型被用来决定是哪个 handler 捕获了异常，但是异常对象的内容不会被访问。此时，创建栈路径是没有必要的。如果总是不用创建异常对象，那么会减少很多工作。例如图 10.6 中，异常对象在 printSumOfInts 的异常处理中并没有用到。一般编译器分析就可以测试到对象在处理入口就

是死的，如果这个异常对象根本就没有用到，就可以不用创建。问题是：异常是在一个不同的方法中抛出的，抛出时并不知道哪个 handler 会捕获异常，因此无法知道是否有必要创建这个对象。所以几乎所有 Java VM 观察到异常情况都会保守的创建、初始化全部的异常对象。而在 O3 JIT 中实现的 lazy 异常抛出减少了创建异常对象的需要。（copy_prop.cpp 中的 prop_type_info）

Lazy方法是这样的：先假设异常对象是不需要的，寻找handler，一旦找到，通过JIT提供的数据结构决定异常对象在handler入口是否是活的。如果编译器证明异常对象是死的，那么就不会创建这个对象；如果是活的，那么就必须创建。困难在于异常对象必须在抛出处的上下文中被初始化。而VM可能已经破坏性的回退了部分（也可能为空）栈顶帧。在10.6.3.3副作用一节，我们会详细讨论。VM异常实现Lazy抛出相对简单。这些预定义的运行期异常是JVM语义要求的。比如ArrayIndexOutOfBoundsException，NullPointer-Exception。这些异常对象的创建和抛出完全由VM完成，操作顺序对应用程序也不可见。但实现用户异常的lazy抛出就要做更多分析，这将在用户异常和副作用中讨论。

10.6.3.1 VM 异常

VM 异常的lazy实现相对简单。当VM监测到一个异常条件，它首先做破坏性的回退，找到合适的handler，然后查询数据结构看异常对象在handler入口是否是活的。如果对象是活的，那么对象按照标准VM中的对象那样创建；如果对象是死的，那么就不用创建这个对象。因为在我们的实现中，VM异常没有副作用，应用语义不会改变。同样，我们VM异常的构造函数自身也不会抛出异常，所以 “使用正确的上下文”中提到的一些问题不会发生在VM异常中。

10.6.3.2 用户异常

用户异常通常通过下面的字节码序列创建：

```
new <UserException>
dup
[压入构造函数参数]
invokespecial <UserException(...)>
athrow
```

这些指令不必相邻，它们可以分布在多个块，甚至多个方法中。然而实际上，全部的指令序列通常是在同一个bb中，它的分析可能在一个优化编译器中。如果我们知道不执行构造函数或是在不同的上下文中执行不会改变程序语义，那么我们对用户异常的处理和VM异常的处理是一样的。

10.6.3.3 一些副作用

副作用是指可能会导致一些违反程序语义的问题。我们认为在异常对象的构造函数中有下面的操作集合是有副作用的：对域、数组元素做store，调用没有被内联的操作。 在调用上下文中还提到，我们也不允许同步操作和可能导致异常的操作。

我们使用javac的指令序列作为实例，其中有异常对象的lazy创建，并且所有抛出异常都是用户异常。下面是该应用程序特定的指令序列。所有非标准类都声明在spec.benchmarks._213_javac包中。 这是Identifier.Resolve(Environment, Identifier) 方法中的代码片断。

```
new <ClassNotFound>
dup

aload_2 //压入一个构造函数

argument
invokespecial <ClassNotFound(Identifier)>
athrow
```

这个例子遵循用户异常的一般模式。优化的本质是推迟ClassNotFound对象的创建，直到VM找到正

确的handler并确定异常对象的确在handler中是活的。在很多应用中，异常用于非正常控制流，但是异常对象只是用于选择正确的handler。在这些应用中，编译器能检测到异常对象在handler中是死的。

当这种情况发生在javac中，它抛出异常只是将控制转移到handler中，而此时的异常对象是死的。如果异常构造函数有副作用，那么就不可以做lazy异常。我们只在编译器能证明构造函数没有副作用时才使用lazy异常。注意：尽管构造函数的参数总是被求值，VM也可能采用更加激进的方法来尽可能晚地计算传递给构造函数的参数。

为了决定一个构造函数是否是无副作用的，编译器必须分析构造函数和所有被递归调用的方法。 一个构造函数通常调用父类的构造函数，父类构造函数又会调用它的父类。因为ClassNotFound是Exception的子类，所以ClassNotFound, Exception, Throwable和Object的构造函数被依次调用。另两个方法Identifier.toString()，Throwable.fillInStackTrace()也被调用了。

1．域更新

下面的指令流序列中即使有对对象域的store也没有副作用，也可以被认为是创建lazy异常的候选。这是因为：对死对象的状态改变没有明显的副作用。ClassNotFound(Identifier) 的构造函数代码片断中可能有副作用s的是对域名的store。

aload_0 // this **指针**

aload_1 // **构造函数参数**

putfield <Identifier name>

这个例子中对异常对象的一个域做了store操作，而这个对象的创建正是我们想要消除的。所以如上面讨论的，我们不认为这种store有副作用。

2．方法调用

任何方法调用都可能有副作用，而编译器就要确定它是否存在。优化编译器会试图递归的内联构造函数中的所有方法调用。虚方法调用同样会被内联，只是会有一个运行期的检查。此时，只有内联方法的路径才被认为是lazy异常机制的候选。如果调用一个native方法，编译器必须假定方法可能是有副作用的。对于native方法还有一种机制可以取代这种保守的假设：每个方法有个相关的flag，它说明方法是否可以不用编译器分析就能假定为没有副作用的。为了做lazy异常分析，我们设置这个flag，标记native方法java.lang.Throwable.fillInStackTrace()是无副作用的。这个假设是有效的，因为VM内部实现了fillInStackTrace()，在我们的VM是可以保证在这个方法中没有副作用。

另一个可能带有副作用的代码片断ClassNotFound(Identifier)为：

```
aload_0
aload_1
invokevirtual <String toString()>
invokespecial <Exception(String)>
```

这个例子中的两个方法代表了不同的问题：

- Identifier.toString() 作为虚方法调用，并需要运行期检查来保证对象的类的确是Identifier。这种分析在内联时就会做，所以不需要再添加新的代码。Identifier.toString()如果参数不为空就不需要抛出异常。编译器的内联结构使它的参数等于Identifier.Resolve(Enviroment, Identifier)方法的变量2。这个变量可以通过它的早期使用来判断是否非空，这决定了异常抛出的代码。
- 第二个方法Exception(String)难于分析是由于另一个原因。调用并不是virtual的，所以内联并不需要额外开销。然而，这个方法的内联会导致另一个方法Throwable(String)被内联。
  Throwable(String)方法的副作用分析更加麻烦。
  首先，这个方法修改了创建对象的一个域，但是这只是一个虚假的副作用，因为它涉及到对我们希望避免创建的对象的修改，所以下面的代码假定是没有副作用的。

aload_0 // this 指针

aload_1 // 构造函数参数

putfield <String detailMessage>

其次，这个方法又有如下代码片断：

aload_0

invokevirtual <Throwable fillInStackTrace()>

优化编译器可以内联虚方法(+运行检查)，但这儿的问题是：Throwable.fillInStackTrace()是一个native方法，JIT 编译器保守的假定所有的native方法都可能有任意的副作用。在上面的讨论中提到，VM其实可以把它标记为无副作用的。

3．使用正确的上下文

优化编译器总是对异常对象构造函数所有的参数求值，即使构造函数从未调用。 所以和参数求值相关的任何代码执行总会在正确的时间和上下文中执行。然而编译器必须保证构造函数自身，以及它所调用的方法都不会有副作用s。但是当VM发现正确的handler，并发现异常对象必须被创建时，异常执行的上下文可能已进不存在了，因为一些栈帧可能已经被破坏性的回退， 一种解决方法可以先用无破坏的回退找到handler，如果异常对象是活的，就在正确的上下文中创建这个对象，然后再次遍历这个栈。这个方法虽然简单，但是要对栈多遍历一次，损失了性能，使用lazy异常就失去了意义。所以我们只在编译器分析能够证明：在handler的上下文中能够安全执行构造函数时才使用lazy异常。下面就是这样的两种情况。

- 构造函数中的异常

最严重的问题可能来自异常对象自身的构造函数抛出的异常。优化编译器要确保构造函数不会抛出任何新异常，这些异常可以被栈帧位于栈顶和对应于 lazy 异常捕获 handler 的之间的方法捕获handler。例如：A 调用 B，B 抛出一个类 foo 的异常 lazily。B 中没有 foo 的 handler，所以 VM 破坏性的回退 B 帧。现在在 A 中找到一个 foo 的 handler，并且异常对象在 handler 中是活的。VM 就会创建一个类 foo 的对象，假设这个对象是在 B 的上下文中构造，如果 foo 构造函数抛出了一个新的异常，如类 bar 的异常，就有可能这种异常的 handler 存在于 B 中，控制也要转移到那里。为解决这个问题，我们保证只有在编译器能证明构造函数不会抛出异常时才允许 lazy 异常。

- 同步

在Java中，如果有个方法声明为同步的，那么进入了这个方法，就进入了和this指针引用对象相关的monitor。破坏性的回退则会退出这个monitor。但是有可能程序抛出一个异常，而异常对象的构造函数依赖一个事实：当前的线程要进入栈上某个方法的this指针所引用对象的monitor。如果在退出monitor后调用构造函数lazily，那么就会发生死锁或是由于同步不充分产生不正确的结果。为避免死锁，如果构造函数的编译器分析发现任何monitorenter字节码或是任何同步方法的调用，我们就不允许lazy异常。注意：由于同步不充分产生的错误实际上不可能存在，因为我们不允许lazy异常构造函数有副作用。

定义在 opt_throws.cpp 中的 optimize_throws 方法对 CFG 中的每个节点遍历，找到最后一条指令为 Call_Inst::athrow_call 的节点，并对这个节点调用 eliminate_athrow(node) 消除 athrow 语句。下面就是这个函数的代码：

```
void Lazy_Throw::eliminate_athrow(Cfg_Node *node)
{
    //设置构造函数参数的空检查和 vtable 检查为空
    init_check();
```

```
Inst *head = node->IR_instruction_list();
Inst *last = head->prev();

//确定不是空的基本块并且它包含 athrow 语句

assert(last != head && last->is_call() && ((Call_Inst*)last)->kind == Call_Inst::athrow_call);

//接下来寻找创建异常对象的 new 调用指令

Inst *obj_new = find_obj_new(head,((Call_Inst*)last)->get_arg(0));

//如果对象的 new 指令没有找到，例如不在当前块中，那么我们就不做 lazy 异常抛出

if (obj_new == NULL)    return;

//确定对象是 throwable 的

assert(obj_new->prev()->is_call());
Class_Handle exc_ch = throwable_class((Call_Inst*)obj_new->prev());
if (exc_ch == NULL)    return;

//判断 newobj_new 语句与 athrow 语句之间的语句是否有副作用

Call_Inst *constructor = NULL;
Call_Inst *new_inst = (Call_Inst*)obj_new->prev();
Inst *inst;
for (inst = obj_new->next(); inst != last; inst = inst->next())
{
    //如果有将对象存贮在局部变量，域或是数组中的语句，那么我们认为对象创建是必需的

    if (inst->is_assignment())
    {
        Operand *dst = inst->dst();
         // putfield/putstatic/vars
        if ((dst->is_mem() || dst->is_vreg()) && is_excp_obj(inst->src(0), obj_new->dst()))
            return;
    }
    else if (inst->is_call())
   {
    Call_Inst *cinst = (Call_Inst *) inst;

    //  找到抛出对象的构造函数

    //  我们只处理异常抛出序列 obj = new(); obj.constructor(); athrow(obj);

    if (cinst->kind == Call_Inst::special_call &&method_get_class(cinst->get_mhandle()) == exc_ch)
        {

            //判断构造函数是否有副作用

           if(method_has_side_effects(cinst,obj_new->dst()))    return;
           assert(constructor == NULL);
```

```
                constructor = cinst;
            }
        }
    }//for 循环结束

    //现在我们知道用 lazy 异常取代 athrow 是安全的。接下来要判断是否需要为保持 athrow 生成一个节点
    bool keep_athrow = false;
    for (unsigned i = 0; i < MAX_LAZY_ARGS; i++)
        if ((i != 0 && null_check[i]) || vrtl_check[i] != NULL)
        {
            keep_athrow = true;
            break;
        }
    if (keep_athrow)      //需要为插入代码创建 blocks
        replace_athrow(node, new_inst, constructor, (Call_Inst*)last, exc_ch);
    else
        remove_athrow(node, new_inst, constructor, (Call_Inst*)last, exc_ch);
}
```

第11章 动态重编译

ORP 即时编译器的一个重要特征就是它的动态重编译机制。在动态重编译机制下，方法第一次被调用时总是用 O1JIT 编译。当方法被频繁地调用时，将被确认为一个“热”代码，会被 O3 编译器重新优化编译以提高执行效率。在图 8.2 中已经说明了 ORP 动态重编译机制的组成。这个结构包括了三个组成部分：快速代码产生的 O1 编译器、O3 优化编译器、profiling 信息。

在 ORP1.0.9 提供给用户的命令行参数中有三个是与重编译有关的：

- instrument：这个参数事实上是允许重编译的开关，命令行带上这个参数之后，O1 编译器将在目标代码中插入测量代码（instrumenting code），收集必需的 profiling 数据。
- recomp_thread：命令行带上这个参数后，ORP 将开设一条独立的线程对方法进行监测和重编译。
- statistics：命令行带上这个参数后，缺省地打开 instrument 开关。另外 O1 编译器收集的 profiling 信息不但包括触发重编译所必需的一些数据，而且包括了一些更详细的统计数据，例如：lookupswitch 某一个标签被比较的次数、一段数组边界检查代码被执行的次数等。

在这一章中我们将详细说明 ORP1.0.9 中对这种动态重编译机制的实现。

11.1 Profiling 数据

Profiling 数据描述了一个方法被调用的情况以及其中代码被重复执行的情况，它反应了代码“热”的程度。O1 快速代码生成编译器收集的 Profiling 数据主要是两个方面：其一是方法被调用的次数，它指示了方法是否是调用密集的；其二是方法的回边被经过的次数，一段循环代码通常被编译成带回边的代码块，所以这个数据指示了方法是否是循环密集的。当 Profiling 数据显示方法是调用密集或是循环密集之后，这个方法将被 O3 优化编译器重新编译，为方法产生质量更高的机器代码。为了收集这两方面的 Profiling 数据，O1 编译器在方法的入口处和回边处插入了测量代码。

在 O1 编译器的实现中，进入代码选择函数 select_code 之前为每个方法生成一个 Profiling 信息记录：

```
typedef struct Profile_Rec {
    bool been_recompiled;          // 方法是否已经被重编译过了
    unsigned short n_back_edge;    // 方法的回边数目，决定了 back_edge 数组的大小
    PROF_COUNTER m_policy;         // 方法被确认为调用密集的阈值，缺省时是 1000
    PROF_COUNTER m_entry;          // 记录方法被调用次数的计数器
    PROF_COUNTER back_edge[1];     // 计数器数组头指针，记录了每一个回边被经过的次数
} Profile_Rec;
```

这个 Profiling 信息记录将存放在简单编译信息结构 Small_Method_Info 中，通过方法句柄（相应的 Method 类对象）能够得到这个记录。其中计数器类型 PROF_COUNTER 是 32 位或 64 位整型的别名。每一个方法有一个调用计数器，每调用方法一次就更新计数器。方法每一条回边也有一个计数器，回边的目标基本块入口指令每被执行就更新相应的计数器。

当-statistics 开关被打开时，Profile_Rec 中的计数器数目大大地增加。这时将为每个基本块的入口都

设立一个计数器用来统计进入基本块的次数，而不单单是回边的目标基本块入口。而且还为每个基本块内部预留了 10 个计数器，这些计数器是用来统计基本块内一些可能被频繁调用的代码被执行的次数，例如：在编译 lookupswitch 字节码时，将为每一条标签产生比较跳转指令，这时将为每一条标签的比较跳转指令分配一个计数器，统计这条标签被比较的次数；另外，一个数组的边界检查也是可能会被频繁调用，也有一个计数器来统计某个数组边界检查代码被执行的次数。所有这些计数器初始值都被置为 0，被监测的代码每被执行一次相应的计数器就加 1。这些统计数据将会被一些分析例程所使用。

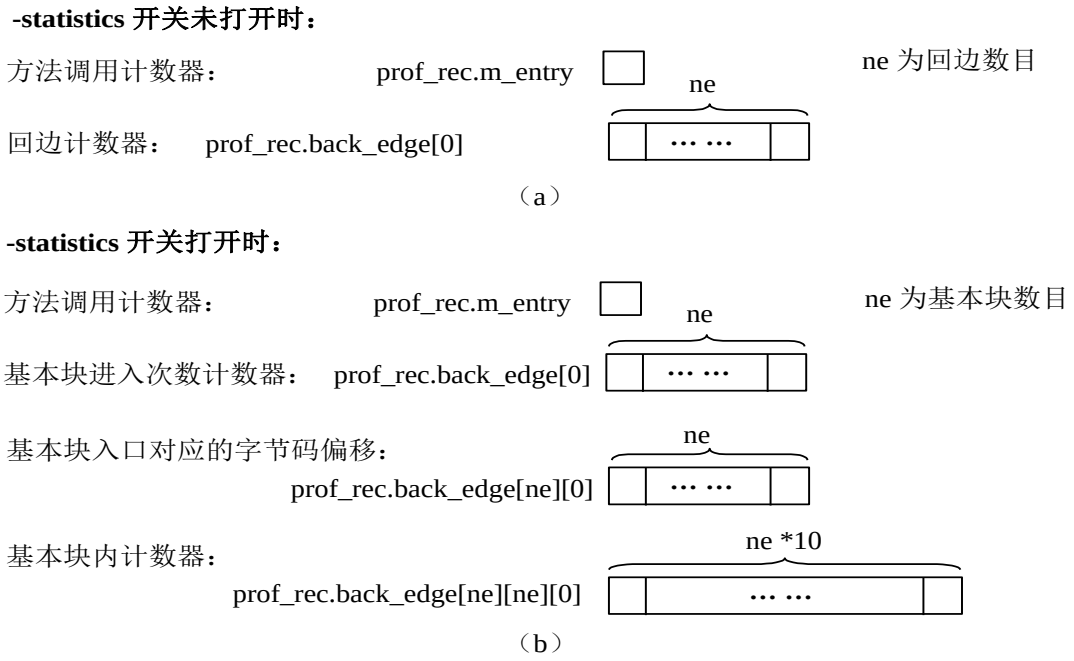


图 11.1 Profile_Rec 结构的计数器结构

11.2 触发重编译

在动态重编译机制中的一个关键问题是如何触发方法的重编译，也就是决定什么时候对哪些方法进行重编译。重编译的编译时间花费非常大，因此要尽量的避免把“冷”代码重编译。ORP 提供了两种触发重编译的机制：测量代码触发重编译和线程触发重编译。

11.2.1 测量代码触发重编译（Instrumenting）

第一种触发机制在测量代码中触发重编译。采用这种触发机制时，ORP 的命令行参数只打开-instrument 开关，而不打开-recomp_thread 和-statistics 开关。这时 Profile_Rec 中的计数器如图 11.1 所示所有的计数器初始化时都设为一个阈值，在 ORP1.0.9 实现中，方法调用计数器的阈值为 1000，回边计数器阈值围 10000。O1 编译器在代码中插入测量代码，当被监测的代码被执行一次，就把计数器减 1。方法调用计数器被减为 0 时，方法被认为时调用密集的；任何一个回边计数器被减为 0，则认为这个方法时循环密集的。这两种情况之一出现后，方法立即跳转到触发重编译的代码。

以函数入口为例，O1 编译器在产生方法的前导部分代码时，插入了这样一组语句：

```
DEC     [cnt_addr]                          ；方法调用计数器减 1

JZ      TRIGGER_RECOMP_METHOD_ENTRY  ；结果为 0，则跳转到触发重编译的代码

CONTINUE_METHOD_ENTRY：
…
```

在产生这一组代码时，O1 的代码选择器还没有产生触发重编译的代码，它并不知道标签

TRIGGER_RECOMP_METHOD_ENTRY 的偏移，但是它在 prof_patch 链表（9.2.3.6）中加入了一个补丁，记录了跳转指令在临时代码缓冲区中的地址，以及跳转指令下一条语句的地址。在为代码选择器遍历完所有的字节码之后，将产生调用一段函数代码，这个函数将触发重编译：

TRIGGER_RECOMP_METHOD_ENTRY：

```
PUSH    jit_handle                  ；触发重编译函数需要的两个参数：JIT 句柄，

PUSH    method_info                 ；以及方法的编译信息（Jit_Method_Info）

CALL    addr_of_trigger             ；调用触发重编译的函数

JMP     CONTINUE_METHOD_ENTRY ；调用返回后，回到跳转指令的下一条指令
```

O1 代码选择器在产生这一段代码时，通过 prof_patch 中的补丁记录能够准确地得到相对于 JMP 语句 CONTINUE_METHOD_ENTRY 标签的偏移，并且也能为之前的 JZ 语句打补丁。它调用的这个函数将方法的 Profile_Rec 记录的 been_recompiled 域设为 TURE，并且调用函数：

jit_recompile_method（method_handle，o3_jit）；

这个函数将调用 p_jit_a_method_lock ->_lock（），将方法上锁，不让其他线程编译这个方法；然后调用 O3 优化编译器对方法进行编译，生成高质量的机器代码；编译成功后将方法句柄（相应的 Method 对象）中的机器代码指针_code 指向产生的优化代码，方法下一次被调用时执行的就是优化代码了；最后调用 p_jit_a_method_lock->_unlock（）释放锁。

在重编译返回后，仍然跳转回原来的 O1 产生的机器代码，继续执行方法的这次调用。

测量代码触发重编译的好处在于，一旦门限阈值达到重编译立即执行，能够尽快地使用新产生的优化代码。但是其缺点在于为所有的 Java 应用程序选择统一的阈值。有时我们希望较早地对“热”代码重编译，则把阈值设低一些，结构很可能是一些“冷”代码也会花费很多时间进行重编译。如果把阈值设高一些，就可能导致“热”代码不能及早地重编译。

11.2.2 线程触发重编译（Threading）

ORP 命令行参数-recomp_thread 开关打开之后，将采用基于线程触发重编译的方法。采用这种方法时为重编译建立一个分离的线程，通过将程序的编译和执行重叠来减少编译时间，适用于多处理器的系统，通常也叫连续编译。

在线程触发重编译中，O1 编译器只需要产生测量代码收集 Profiling 数据。方法的机器代码产生时，所有的计数器初始化为 0。当每一个被监测的代码被执行时，测量代码只需将相应的计数器加 1。

当-recomp_thread 开关打开时，ORP 的 main 函数将开设一条单独的线程，这个线程的流程如图 11.2 所示。

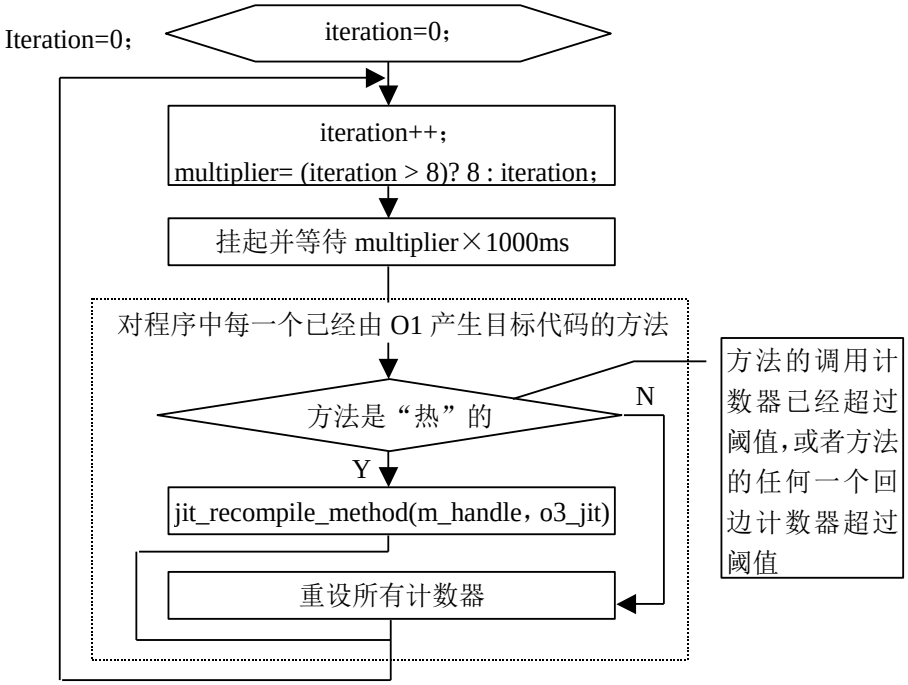


图 11.2 监测方法执行并触发重编译的线程

这个线程每隔一段时间扫描每一个用 O1 快速编译器产生机器代码的方法 Profile_Rec 记录，检查各个计数器是否已经超过了阈值。如果有一个计数器超过阈值，那么说明该方法在这段时间里频繁调用，被认为是“热”代码，将被 O3 编译器重新编译。基于观察有这样的结论：大多数的“热”方法都在程序的较早阶段被执行。所以在一开始时，线程挂起等待的时间比较短（1s），随着程序的运行等待时间线性增加（每段时间增加 1s，当增加到 8s 时不再增加）。

图中显示的是-statistics 开关没有打开时的计数器情况。ORP1.0.9 版本中的重编译线程只实现了这种最简单的分析。如果-statistics 开关被打开，将有更多的统计数据，重编译线程可以根据这些数据进行更复杂的“热”代码分析，使“热”代码的重编译触发时机更加合理。

采用分离的线程使重编译时程序无需停止执行，编译和程序的执行时间重叠，从而减少了编译的时间。但是也存在着两个缺点：第一，这个线程为了判断出“热”方法进行重编译，必须扫描所有方法的 profiling 数据；第二，当“热”方法的 profiling 数据达到阈值时，由于线程可能正处在挂起阶段，并不能马上被重编译，必须等待线程的下一个工作时段。

参考文献：

Practicing JUDO: Java Under Dynamic Optimizations Cierniak,Lueh and Stichnoth. PLDI 2000

Support for Garbage Collection at Every Instruction in a Java Compiler, Stichnoth, Lueh, and Cierniak. PLDI 1999

Fast, Effective Code Generation in a Just-In-Time Java Compiler, Adl-Tabatabai, Cierniak, Lueh, Parikh and Stichnoth. PLDI 1998

第三部分 垃圾收集

第12章 什么是垃圾收集

垃圾收集（garbage collection，简称 GC），也可以称为自动内存管理，它是一种对动态分配内存空间的自动回收重用机制。在概念上，与手工内存管理相对。垃圾收集通常是一个语言或者系统的运行系统的一部分，或者是一个另外加上去的库，并需要从编译器，操作系统，硬件，或者从这三者的任意组合获得一些帮助。垃圾收集自动地确定那些不会再被使用的数据对象，然后释放它们占据的空间，使之可以被运行时系统重新使用。一个对象如果不能让运行时系统通过指针遍历到，那么它就被看作是垃圾（garbage）。而活对象（live object），也就是可达对象（reachable object），乃是收集器要维护并保留的目标。

追溯它的历史 ，它第一次出现于 1958 年，由 John McCarthy 首先实现，使之作为 Lisp 实现的一部分。现在有很多语言都提供了 GC，其中重要的一些包括 Java，Perl，Modula-3，Prolog，ML，和 Smalltalk。

12.1 垃圾收集兴起的推动力

为什么我们要使用自动管理内存空间的机制呢？随着面向对象语言的出现，数据结构变得越来越复杂，传统的依靠程序员去显式地释放对象空间的做法显得越来越烦琐，易于发生错误。首先，应用 GC 避免了显式地回收内存空间，也就避免了一些可能因此而产生的问题，如，内存漏洞，过早释放内存，以及重复释放内存。

1. 内存漏洞就是指内存中某处被分配的空间虽然已经不再被使用，但是并没有被回收。内存漏洞的累积会导致应用程序使用的内存空间无限制地增长，直到耗尽内存空间，使得应用程序不得不提早终止。在手工内存管理中，如果程序员没有在合适的地方回收空间就会导致内存漏洞的发生。
2. 在手工内存管理中，过早释放内存经常发生在这样的情况下，那就是程序员发现程序的某一部分决定了某块内存已经结束使用了，就释放了这部分内存，却不知道程序的另一部分仍然在使用它。过早释放内存的结果是灾难性的，因为回收的空间可能会被用来存储跟以前完全不同的数据结构，而这时，也许还有其他地方对这个老空间的引用存在，这个引用就变成了悬空的了。结果一个内存空间同时被翻译成两个不同的对象，对其中一个对象的更新给另一个对象造成的影响是不能估计的。在自动内存管理中，这样的错误一般不会发生。
3. 重复释放内存意思就是试图去释放一块已经被释放过的内存。在手工内存管理中，这个错误通常发生在程序的两个部分都认为自己应该负责去管理这块内存。可以想象，如果被释放过的内存已经被重新使用了，那么重复释放它会导致严重的错误。

另外，上述这些问题所导致的错误并不容易被发现。因为通常来说只要当程序运行到一个原来没有预料到的轨迹上，这样的错误才会显示出来。然而就算是到了这个时候，错误仍然是零星地发生，使得调试起来异常的困难。只要程序运行在正常的轨迹中，缓慢的内存漏洞可能根本不会被发现。只要内存分配恰好不会重新使用这个特殊的空间，悬空的指针也可能不会产生任何问题。然而，在任何时候，只要对内存分配做一些小小的改动，这些隐含的错误都可能会导致程序崩溃。

其次，垃圾收集对于实现完全模块化程序设计也是很重要的。完全模块化设计的意思就是尽量避免不必要的模块间依赖关系。要做到这一点，当一个例程运行在某个对象上时，它除了因为某种特殊的原因而需要和其他例程做协调之外，不应该必需要知道其他的例程可能对同一个对象做什么。但是，如果

该对象需要被显式地释放空间，某个模块就必须负责对它的管理，它必须知道其他的模块什么时候不再使用这个对象了。也可以这么理解，既然一个对象的存活性（liveness）是一个全局性质，例程中为了管理它，必然要引入非局部性的代码。这就使得本来可以是毫不相关的例程之间产生了依赖关系。另外，全局性的代码也降低了程序的可扩展性，因为如果要实现新的功能，例程中所有的全局性代码都需要得到更新。而引入垃圾收集机制之后，模块间就不需要对对象进行内存管理，使得模块间的界面简洁明了，大大简化了程序的设计。

最后，应用垃圾收集技术还可以对内存的使用性能带来一定的改进。由于它在执行的时候需要移动对象，所以它可以做出以下的一些有益的行为。紧缩（compact），在移动对象的过程中将活的对象都移动到一个连续的空间中，这就减少碎片化的问题；重新聚集（recluster），将同类的对象，相关联的对象移动到一块儿去，这样做可以极大的提高高级缓存的利用效率。

12.2 一个两阶段抽象

垃圾收集自动的回收不再可达的对象的空间，这些不再可达的对象就叫做垃圾，抽象的说，一个垃圾收集器最基本的功能就是以下两个部分：

1．用某种方法区别活对象和垃圾对象。这一阶段也叫垃圾探测

2．回收垃圾对象的内存空间，使得运行的程序可以重新使用它们

实际使用中，这两个阶段可能会是相互交织起来运行的，而且回收技术很大程度上依赖于垃圾探测使用的技术。也就是说，从本质上是探测阶段所使用的技术决定了各个垃圾收集算法的不同性质和功能。各个算法的命名也基本上是由探测阶段的特点所决定的。

下面讲一下存活性这个概念，这个概念对于垃圾收集器来说是很关键的。上文中，我们将活对象和可达对象等同，实际上这是一种不精确的表达。虽然在垃圾收集这个领域内，你可以粗略地将两者看成一件事情，但是实际上，两者在概念上是有区别的：不可达的对象一定不是活的，但是一个可达对象并不一定是活的。

在一个优化的编译器中，当控制流和数据流的分析决定了一个值不会再被运行的程序使用了，这个值就被看作死亡。一般来说，一个垃圾收集器它不可能精确地决定哪个对象是活的，连编译器也并不总能判断出来这一点。因此所有的垃圾收集器都应用一种对存活性的近似的保守性标准，它比较简单的且不会具有动态性。为了讲述这个标准，要先定义几个概念：

根集（Root set）：从一个程序可以直接得到的数据的集合。通常根集包含有活动栈中的本地变量，寄存器中的值，还有全局变量，静态变量等。

可达数据（Reachable data）：从根集开始跟随指针（引用）而访问到的数据。

活数据（Live data）：程序将来会用到的数据。

这个标准就是通过根集和从这些根开始的可达性来定义的。当在某一个点垃圾收集发生了，根集中所有的变量是看成是活的。从这些变量直接可达的堆对象可以被运行的程序所访问，所以它们也必须被保存。另外，既然程序可能会从这些直接可达的堆对象遍历指针到达其他的对象，任何从一个活对象可达的对象都应该被看成是活的。因此活对象的集合也就是从根集开始的任何一条指针路径上的对象集合。任何不能从根集可达的对象都是垃圾，因为不可能会有合法的程序序列使得程序会访问这个对象。垃圾对象不会影响将来的计算过程，因此它们的空间可以被安全的回收。

在下面的介绍中，我们将垃圾收集器当作是一个语言实现中的一部分。通常的安排是该语言的分配例程在一个内存要求得不到满足的时候，就执行特殊的操作来回收空间，也就是说不需要调用“deallocator”例程，因为它在对分配器的调用中已经隐式地出现了。

另外，还有一点要说明，大多数垃圾收集器都需要来自编译器的支持，如，GC 必须能够认识对象的格式，根据 GC 算法的细节，可能需要对编译器中的代码生成器作一些改动，在编译期间发射某些额外的信息，还可能在运行时运行不同的代码序列。

Intel 的开放式运行时平台（open runtime platform）中也实现了自己的垃圾收集模块。它的实现运用

了很多的有关垃圾收集的技术，我们在下面将会逐一的讲到。一些它没有选择的算法和机制等等，我们也会稍微有所涉猎，以便读者可以对垃圾收集这一个领域有一个较连贯和完整的认识。对于 ORP 中 GC 模块所使用到的技术，我们当然会更加详细的说明，并在第 15 章中结合具体的实现来进一步解释。

第13章 垃圾收集的技术

在这一章中，我们从简单到复杂的来一一介绍一些重要的垃圾收集的算法。所讲述到的算法主要是在单处理器环境下使用的垃圾收集算法，而且重点是在近 10 年来发展起来的那些技术。这里我们主要参考了 Wilson 的一篇杰出的论文【Wil92】。如果读者想要了解 1980 前的 GC 技术，请自行参考 Cohen 的一份全面的调查和分类的报告【Coh81】。

13.1 基础的垃圾收集技术

这一节主要目的是介绍一些有关垃圾收集的术语，并且为下一节介绍先进的垃圾收集技术设定一个框架。因此介绍了一些传统的垃圾收集技术。事实上，所有其他的垃圾收集技术都是基于这一节所介绍的三种基本的 GC 技术而建立起来的。我们主要介绍它们的中心思想，然后给出使用这些基础技术所带来的一些问题。

垃圾收集的第一个抽象阶段－垃圾探测可以通过以下几种方法去完成：引用计数,标记,或者拷贝。因为以上每种方案都会对第二个阶段－回收产生影响，因此我们会在介绍上述探测方案的过程中，逐步地介绍回收阶段使用的方法。

注：在将要讲述的算法中，引用计数垃圾收集器是特殊的，它不属于跟踪型收集器（tracing garbage collector）的范畴，事实上，它并不沿着指针遍历整个活对象图，只是用计数值来模拟可达性。而跟踪型收集器当真要求遍历活对象图，将它们和可以回收的不能到达的垃圾对象区分开来。

13.1.1 引用计数（Reference counting）

使用了这项技术的系统中，每个对象都会有一个计数器，这个计数器记的是现在有多少引用指向它。每当一个指向对象的引用被创建，该对象的计数值就加一。当一个已经存在的指向某对象的引用被删除或者重写，这个对象的计数就减一。比如现在有这样一个指针赋值操作，P：＝Q，那么 Q 所指向的对象的计数值就加一，而 P 所指向的对象的计数值就减一。当一个对象的计数器的值被减到零的时候，说明已经没有指针指向这个对象了，那么它所占的空间就可以被回收。

当一个对象被回收的时候，它的每个指针域都要被检查。因为来自一个要被回收的垃圾对象的引用在决定存活性的时候是不被计数的，所以如果它包含指向别的对象的指针，这些对象的引用计数也要减一。由此可见，回收一个对象可能引起传递性地给别的对象的计数值减一，然后再回收那些因此而使得计数值变成零的对象。例如，如果指向一个大型数据结构的仅有的一个指针变成垃圾，在这个大型数据结构中的所有的对象的计数值最后都变为零，因此所有的对象都要被回收。

简单的引用计数垃圾收集器有很多问题和缺点，下面列举出最主要的三个：

循环的问题：有效性问题是引用计数的技术对于循环的数据结构会失效。如果在一组对象中的指针创建了一个循环，这些对象的引用计数永远不可能减到零，即使从根集已经不可能到达这些对象了。而且事实上，这种循环在一般的程序中会经常地产生。

效率问题：引用计数的效率问题是它的代价一般来说跟运行程序所作的工作成正比，而且比例常数相当大。

首先因为每次指针被修改，引用计数器都要被更新一次。例如每当一个指针被创建或是销毁的时候，都需要调整引用计数值。如果一个变量的值从一个指针改成另一个指针，那么这两个指针所指向的两个对象的引用计数值都要被修改，一个对象的引用计数加一，一个减一，然后检查是否为零。还有短期生存的栈变量也可能因此在简单的引用计数方案中导致很大的开销。例如当传递一个参数的时候，一个新

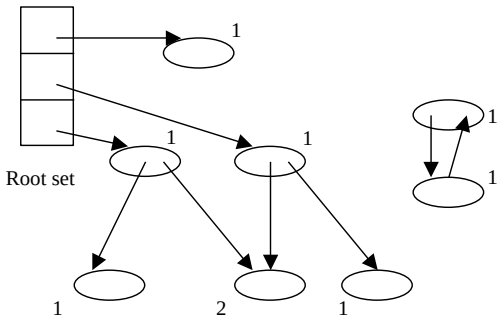


图 13.1 计数值永远不可能为零的循环结构

的指针出现在栈上，通常很快它就会消失，因为大多数的进程活动在被调用之后都很快地返回。在这些例子中，引用计数值增加，然后很快又减回原来的值。可以预料，如果我们消除这些加加减减，就可以极大地优化它们。有一种引用计数算法的改进算法叫做延迟引用计数算法（deferred reference counting），当引用是存储在栈上的，它就通过延迟技术尽量避免对它们指向的对象的计数值进行调整。但是通过延迟的技术优化算法会抵消引用计数技术实时的优点，因此，需要小心的计算扫描栈，更新引用计数值的间隔的时间参数以达到比较令人满意的效果。

其次，当一个对象的计数值被减为零以后要被回收，必需要执行一些 bookkeeping 代码使得回收的空间能被运行的程序访问到。典型地，这包括将被释放的对象链入自由列表中，这种列表中都是可以被重新使用的对象，程序分配内存的要求可以在这里得到满足。所以，每个对象都需要消耗若干条指令来进行回收操作，导致总的代价就跟运行程序所分配的对象的个数成正比了。在这点上需要付出的代价对于引用计数算法来说是无法避免的。

计数器溢出：一般来说，引用计数算法给每个对象都增加一个域，用来作为该对象的计数器，但是这个域的大小是有限制的。如果指向一个对象的引用数无限制增长，使得计数器溢出，该算法就会失效。不过在实际应用中，这个情况比较罕见，除非这个引用计数域非常的小。

引用计数算法也有其优点所在。首先，它具有快速回收的性质。因此它常用来在文件系统，数据库和操作系统内核中回收非循环的数据结构。如果应用它的地方经判断有出现循环结构的可能，那么通常会让它和跟踪型垃圾收集器协同工作，其中跟踪型收集器很少运行。这样的组合当然在效率上比单纯的跟踪型收集器要低，但是引用计数收集器的快速回收性也是很重要的。

其次，引用计数算法还有一个很重要的优点。那就是它的渐增性质－垃圾收集的工作（也就是更新引用计数值）是跟应用程序本身的运行紧密交织的。因收集器的工作而引起的程序运行的暂停时间一般来说是很短的。因此，它可以作为渐增式垃圾收集器的一种形式。（注：渐增式垃圾收集器参见 13.2.2 节）。但是移走一个引用有时候可能会传递性地导致很多对象都要被一齐回收，这就使得它不适用于实时系统。因为在实时系统中，应用程序运行的暂停时间上限必需要得到充分的保证。不过，有一些稍微复杂一点的引用计数算法的变体可以解决这个问题。例如，收集器可以将对于整个数据结构的传递性回收延迟，也可以每次只做一点。方法就是将那些引用计数值已经为零，但是现在不想处理的对象放到一个列表中。收集器维护这个列表，在适当的时候回收其中的对象，目的是保证实时系统对暂停时间间隔的严格要求。

再次，虽然引用计数收集器的效率不如跟踪型收集器，但是它不需要来自语言本身或者编译器的支持就可以工作。这也是它比跟踪型收集器优越的地方，当然，如果有了编译器的支持，一定会极大地提高它的性能。

最后，引用计数算法还有一个优点，那就是当大部分的堆空间都已经被占用的时候，使用引用计数的垃圾收集器的性能也不会受到影响。很多其他的收集器在这个时候会需要更多的空间用于交换以提高效率。

总结上述引用计数算法的优缺点，可以说该算法的效率不高，再加上循环所产生的有效性问题，导

致这个技术在近年对大部分的程序来说都失去了吸引力。但是这并不是说引用计数技术已经丝毫作用也没有了。对于大部分垃圾，它的快速回收性质以及渐增性仍然是它的优点所在。将来也许会发现这种技术的其他一些用途，也许在混合型的收集器中使用，如上文我们说到，可以让跟踪型收集器协助它回收循环型数据结构；也许通过特殊的硬件可以提高它的性能。然而，引用计数一般来说不会作为在传统的单处理器上主要的垃圾收集技术。对于大部分高性能的通用系统来说，引用计数已经被跟踪型垃圾收集器代替。

13.1.2 标记和清除（Mark & Sweep）

标记和清除垃圾收集器是跟踪型收集器的一种，它不会移动对象。这个技术的名字其实就根据本算法对在 12.2 节中讲述的 GC 算法的两个抽象阶段的具体实现的特点来决定的。

首先区别活对象和垃圾。这是通过跟踪做到的。从根集开始遍历整个指针关系图。可以到达的对象就被用某种方式标记。

第二个阶段，回收垃圾。一旦活对象跟垃圾对象区别开来以后，内存就会被清扫一遍，也就是说详细地检查一遍，以发现所有的没有被标记的对象，然后回收它们。

传统的标记清除收集器有三大问题。

首先是碎片（Fragmentation）的问题。所谓碎片问题就是当我们要分配一个 N 字节大小的对象的时候，发现有很多小于 N 字节的空闲块存在，但是就是没有合适的空闲块可以分配给这个对象。在标记和清除收集器中处理大小不同的对象会产生碎片，很难消除。结果被回收的垃圾对象和活对象交织在一起，使得对大对象的分配非常的困难。通过维护不同大小的被释放对象列表并合并相邻的自由空间可以缓解这个问题，但是困难依然存在。（系统必须决定是给小的数据对象分配比实际需要的更大的空间，还是将大的连续的空间划分开来，冒着存在永久的碎片的危险。这个碎片的问题不仅仅在标记清除垃圾收集中存在，它也发生在引用计数垃圾收集器以及大部分的显式堆管理机制中）

第二个问题是效率的问题。收集的代价大小和堆的大小成正比。每次垃圾循环都必须对所有的活对象进行标记，对所有的垃圾对象进行收集，这就给任何效率的提高设置了本质上的限制。然而理论上总代价和堆大小的线性关系并不象第一眼看上去那么棘手。如果活对象倾向于成群的聚集在内存中，那就可以大大地降低总代价中清除阶段所占比例的常数。结果是主要的代价集中在标记阶段，跟必须要遍历的活对象的多少成正比，而不是和整个内存空间的大小成正比。

第三个问题涉及引用的局部性（Locality of Reference）问题。所谓引用的局部性是应用程序的一个特性，它表现为在程序中对邻近的内存位置的访问操作也是相邻的。例如一个依次读一个数组中的所有的成员的程序，或者反复的使用同一个内存变量的程序，就拥有良引用局部性。一个应用程序如果拥有良引用局部性，它就能很好地适用于使用虚存机制和高级缓存这样的机制的系统。因为它能有效地减少工作集和提高命中率。在标记和清除收集器中，既然对象都不会被移动，活对象在一次收集以后仍然在原位置，和空闲空间相交织。然后，新的对象分配了这空闲空间，其结果就是不同年龄的对象交织在一起。这给引用的局部性带来了消极的影响，因为这种交织有可能使得活对象被分散到很多的虚拟内存页中，导致这些页需要频繁地在内存中换进换出。不过这个问题并不象有些人想的那么糟糕，因为对象经常是成群地被创建，这些群一般情况都是同时处于活动状态。然而碎片和局部性问题在一般情况下是不可避免的，也是程序中一个潜在的问题。

需要注意的是这些问题并不是完全不能克服的。通过运用足够灵巧的实现技术，可以解决其中一些问题。标记清除收集器的技术（或者相关的混合型）正在分速地发展，这使得它们到后来在某些方面和拷贝收集器相似。（注：拷贝收集器参见 13.1.4 节）。因此一般没有必要在高技术的标记清除收集器和拷贝收集器之间分出优劣。在下面的一小节中，讲述的就是一种标记清除算法的变体形式，它对原始的算法进行了一些有益的改进。

13.1.3 标记紧缩（mark-compact）

这是一种在标记清除算法的基础上发展起来的算法，和拷贝算法非常的相似。这种算法修正了标记清除收集器的碎片化和分配等问题。它跟标记清除一样，在标记阶段它遍历和标记所有可达的对象。然

后，在紧缩阶段，通常对内存空间进行数遍扫描，用来对这些对象进行紧凑并更新引用值。作为紧缩的结果，活对象都被移动到一个或者数个连续的空间中。这就使得剩下的空间也成为一个或者数个连续的空闲空间。

这个滑动紧凑方法具有数个有意思的性质。首先，连续的空闲空间消除了碎片化问题，给分配例程提供了大块的空闲空间，这样为可变大小的对象和大型对象分配空间就都显得容易了。其次，垃圾空间只是简单地被“挤出来”，不会打乱原来内存中对象的顺序。这就可以缓解局部性的问题，因为分配的顺序通常和随后的访问顺序更相似，而不是跟垃圾收集器随便定的顺序相似。

虽然从滑动紧缩而得到的局部性质是有利的，但是收集过程本身也具有标记清除算法的一个不好的属性，那就是对数据需要数个扫描处理过程。经过最初的标记阶段后，滑动紧缩器还需要对活对象做两到三个这样的扫描处理。一次扫描计算对象将要移往的新地点；随后的扫描处理必须更新指针，使得它们指向原指向对象的新地址；然后实际地移动对象。如果数据的大部分都存活下来需要被紧缩的话，这种算法可能因此而比标记清除更加慢得多。

13.1.4 拷贝收集算法（Copying）

拷贝垃圾收集算法又叫做 scavenging 垃圾收集算法。术语“scavenging”应用在拷贝遍历阶段，因为它包含了在垃圾中捡出活对象和把它们移走这两项工作。

拷贝技术和标记紧缩算法很相似（但不像标记清除），它也并不真的收集垃圾。它将所有的活对象从一个空间移到另一个空间，然后原来的这个空间自然就是可以再使用的，因为这个空间里面就只剩下垃圾了。所以在这些系统中的“Garbage Collection”就只是隐式的，一些研究者甚至避免对这一过程应用这个术语。

拷贝收集器，就像标记紧缩收集器一样，将可以被遍历到的对象移入一个连续的空间。但是紧缩收集器使用一个独立的标记阶段来遍历活对象，而拷贝收集器将数据的遍历和拷贝的过程整合到一块，这样大多数的对象就只需要被遍历一次。当对象被遍历到的时候，它就移入目的地－也就是一块连续的空间中。它需要做的工作量和活数据的多少成比例，和内存空间的大小没有关系，而标记清除收集器就必须要扫描整个内存空间。因此从理论上来说，它的效率很高。

下面来介绍一种简单形式的拷贝收集器：半区收集器（semispace collector），它是一种“stop-and-Copy”算法，使用 Cheney 算法来进行拷贝遍历过程【Che70】。

在这个算法中，空间被分成了两块连续的空间，它们被称作半区：Fromspace 和 Tospace。在通常的程序运行中，只有 Fromspace 是活动的。当运行程序要求内存的时候，就会在 Fromspace 中向上线性地分配内存。这种分配的策略在很大程度上象在栈上的或在一个滑动紧缩收集器中的分配，有快速的优点；同时在给可变大小的对象分配内存的时候也没有碎片问题。

当运行程序要求的内存分配超过了现有的空闲空间的大小，那么运行程序就暂时被停下来，调用拷贝收集器来回收空间（术语“stop-and-copy”就是这么得名的）。所有的活对象从 Fromspace 拷贝到 Tospace。一旦拷贝完成，Tospace 的身份就发生了转变，现在它是下一阶段程序运行的 Fromspace，原来的 Fromspace 的身份就变成了这一阶段的 Tospace.。然后程序继续运行。我们可以简单地把上述论述归纳成一句话，那就是两个空间的角色在每次 GC 被调用的时候对换一下，这样一个过程叫做一个 Flip。

也许，最简单的拷贝遍历的形式就是 Cheney 算法了。下面我们来讲述一下这个算法的思想和过程。

Cheney 算法是一个迭代算法。在这个算法中，拷贝和扫描是交替进行的。我们需要两个指针：扫描指针和空闲指针。空闲指针用来指示 Tospace 中空闲空间的首地址。而扫描指针是用来指示 Tospace 中下一个要扫描的对象，在这个指针以前的对象都已经被完全地扫描过了。初始时，两个指针都指向 Tospace 的开头。

在讲具体的算法过程之前，我们先要介绍一下三色抽象的概念。三色抽象概念是渐增式算法的基础，这里我们借用一下以助于理解 Cheney 算法的执行过程。在做垃圾收集的时候，所有的对象可能是三种颜色中的一种：黑色，灰色，白色。黑色的节点就表示这个对象已经做完了垃圾收集，以后都不会再考虑它了。灰色节点表示这个对象已经被访问过了，但是还没有完成，以后还要访问它。白色节点表示那

些没有被访问过的节点，在 GC 完成后被当作是垃圾的对象。当所有的可达的对象都成为黑色之后，垃圾收集才算是结束了。

我们将根据一个具体的例子来阐述算法的具体过程。图 13.2 给出了在这个例子中，做 GC 之前整个内存空间的情况，包括空闲指针和扫描指针的位置：

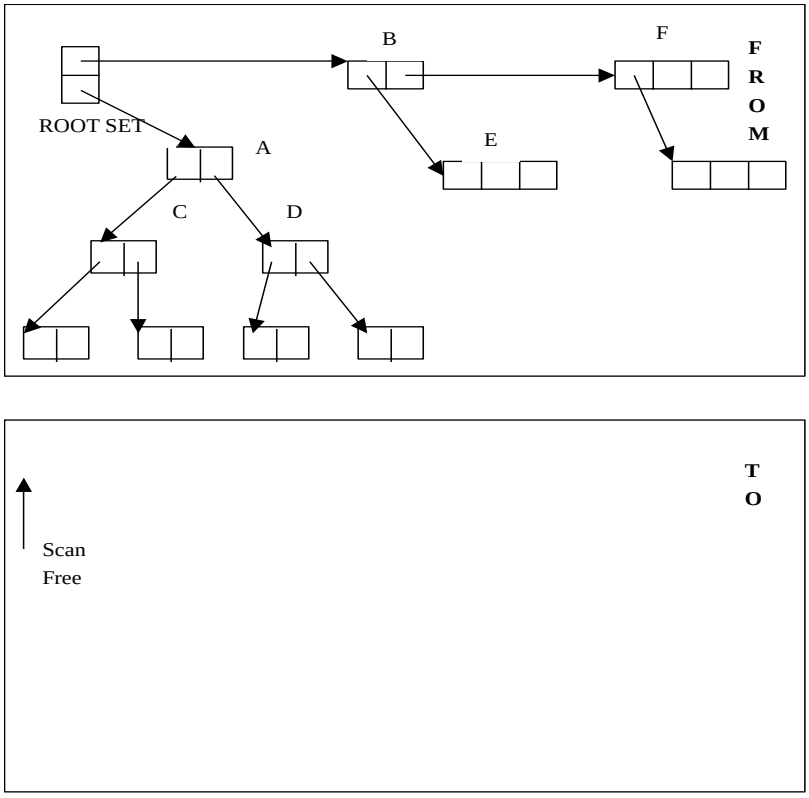


图 13.2 做 GC 前内存空间的状态

首先我们定义可以直接从根集到达的对象集合形成宽度优先遍历的初始对象队列。实际操作中，首先将初始对象队列都拷贝到 Tospace 中，空闲指针往下移动。被拷贝的对象原来指向 Fromspace 中的其他对象的指针仍然指向原来的对象。在实际操作中一个“扫描”指针从第一个对象开始前进，一个 slot 一个 slot 地移动。每次碰到指向 Fromspace 的指针，被它引用的对象就传送到队列的尾部，然后更新指针，使得它指向新的位置。然后扫描指针继续向前扫描。这就实现了宽度优先遍历的“节点扩张”的概念，到达并拷贝那个节点的所有的后代。扫描过程并不会停在第一个对象的尾部，它将对后续的对象继续下去，发现它们的后代也拷贝它们。最后扫描到达队列的尾部，标志着所有可达的对象都已经被扫描完后代了。这就意味着这里没有更多的对象需要被拷贝，scavenging 阶段结束。

例如，从图 13.2 我们可以看到，对于这个例子，初始对象队列就是 A 和 B，它们两个是直接从根集得到的对象。于是 A 和 B 首先被拷贝到 Tospace 中。扫描指针不动，空闲指针下移。接着 A 对象在 Tospace 中被扫描，先将它变成灰色，可以看到，它有指针指向 C，D 两个对象，先访问 C，也同样地拷贝它，将它放到队列的尾部，更新 A 的这个指针，使它指向新的正确的位置。然后访问 D，也有同样的措施。这以后 A 就扫描完毕了，所以它就被染成黑色，表示它不会再被考虑到。扫描指针下移，现在指向了 B 节点。如此类推，最后所有的可达节点都到了 Tospace 中，并变成了黑色，GC 就完成了。图 13.3 给出了收集过程中 Tospace 的初始的几个状态，Tospace 用线性的地址顺序显示出来，用来强调线性地扫描和拷贝。该图中，我们省略了 Fromspace，也就省略了被拷贝到 Tospace 中的对象指向原来所指向的 Fromspace 中的对象的指针。只给出 Tospace 中对象的状态。

上面所述的算法是非常简单的，按照它的方法，有可能使得通过多种途径到达的对象被重复地拷贝。因此还需要对它稍微改进一下。这就引入了 forwarding 指针的概念。所谓 forwarding 指针就是指当一个对象被拷贝到 Tospace 之后，写到该对象的旧版本中的一个特殊的引用，它表明 Fromspace 中这个对象

已经过时，并且指示到哪里去找到该对象的新版本。当扫描过程发现一个指针指向 Fromspace，就会检查它指向的 Fromspace 中的那个对象是否有 forwarding 指针。如果它有 forwarding 指针，就表示它已经被拷贝到 Tospace 中去了。因此这个指针就将被更新，使得它指向新的地址。这就保证了每个对象只被拷贝

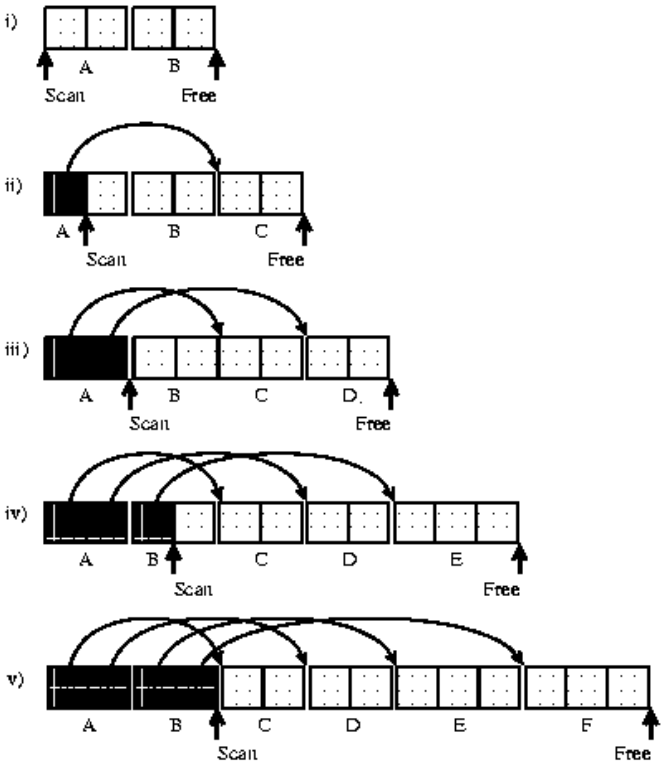


图 13.3 Cheney 宽度优先拷贝算法的一部分过程

一次，所有的指向这个对象的指针都会被更新，指向新的拷贝。

现在我们来考虑一个拷贝垃圾收集器的效率问题， 从理论上来说只要给足够的内存，这种垃圾收集器可以得到很高的性能。因为每次收集所作的工作是和收集期间活数据的多少成比例，假定在程序运行的任何时刻都有大致相同数目的数据存活，降低垃圾收集的频率就将降低整个垃圾收集所作的工作量。而能够降低垃圾收集频率的一种简单的途径就是增加堆中内存的大小，因为如果每个半区更大的话，程序会在填满它之前运行较长的一段时间。我们可以从另一个角度理解这样做得到的好处：通过降低做垃圾收集的频率，垃圾收集期间对象的平均年龄将会增加，在垃圾收集前已经变成垃圾的对象不再需要被拷贝，所以一个对象永远不需要被拷贝的几率就会增加。

例如，假定在一个程序运行时，总共有 20M 的空间被分配了，但是只有在任何给定的时刻，只有 1M 的数据存活下来。如果我们有两个 3M 的半区，垃圾将被收集十次。（既然当前的半区在一次收集之后是 1/3 满的，这就留下了 2M 的空间用来在下一次做 GC 之前供分配用）。这就意味着系统将拷贝将近一半总共分配的数据。如果半区的空间加倍，每次收集以后会有 5M 空闲的空间留下来。这就在运行程序的时候只需要进行 3 次或者 4 次垃圾收集。频率缩到原来的 1/3。这就很直截了当地降低了超过一半的垃圾收集的代价。

不幸的是，在实际应用中很难达到这么高的性能，因为大容量的内存非常的昂贵。如果使用虚拟内存，分配和回收循环的弱局部性通常会导致额外的换页（因为只有堆上每个地方都要被使用完后，才会对内存空间进行回收和重新使用。）对一个高速的处理器来说，单单是换出最近分配的数据的页面的代价就非常大，而且由拷贝收集自身引起的换页代价可能更加的巨大，因为所有的活对象必须在处理中被接触到。因此通常我们不会使得做垃圾收集的堆大于可以得到的主内存。

一般来说，我们不可能取得一个简单的垃圾收集算法最高的效率；目前只能在换页的时间抵消任何优点之前增加内存的大小来延迟和避免收集。

认识到这个问题不是仅仅对拷贝收集器是非常重要的。所有的垃圾收集的策略都包括相似的空间时间折中－垃圾收集被推迟，这样垃圾探测的工作就不会那么频繁地做，但这也意味着空间不会被那么快被收集到。简单的来说，增加内存的大小所带来的好处将会因为没有被回收的垃圾而被抵消掉。

除了对拷贝算法效率上面的这点实际的考虑之外，它还有一些其他的缺点。首先，在高层，当拷贝大对象的时候代价太大。另外还有一个问题就是半区算法需要的空间是实际需要的逻辑空间的两倍，这就更加剧了内存紧张所导致的效率问题。其次，在低层，宽度优先的算法减弱了局部性质，而且还存在分页的问题。

有缺点就有改进。拷贝算法有很多变体形式，对它进行改进的方法各种各样，我们可以列举一些，比如，可以在整个内存空间中划出一个特殊的空间，作为大对象空间，这个空间可以被不一样的收集器来处理；可以给长期存活的对象存储的空间，这些空间只扫描，不会拷贝；可以使用其他的浏览策略来维持局部性质等等

那么怎样决定在什么时候应该使用拷贝垃圾收集算法呢？下面所列的一些情况都有利于使用拷贝算法：当内存空间的管理是由便宜的分配动作所支配的；如果产生了很多小的，短期生存的对象；因垃圾收集而产生的延迟无关紧要。

总的来说，拷贝算法的前景是很好的，它被广泛用作其他垃圾收集算法的基础，如渐增式的算法，分代算法。在 ORP 中的 GC 模块不仅单独实现了拷贝算法，而且也是把它用作实现其他算法的基础。具体的实现可以参考第 15 章中 GC-v3 的内容。

13.2 高级的垃圾收集技术

在上一节中，讲述了三种简单的垃圾收集器。在实际的应用中，它们有很多的问题有待于解决。例如， 虽然拷贝收集器在开初设计的时候就是为了解决局部性的问题，但是在它们的简单的版本中，这个提高不是很大。对于一个简单的半区拷贝收集器来说，局部性问题很可能比标记清除收集器更加严重－因为只有一半的空间可以在两次收集之间应用到。有规律的重用两块空间而导致的弱局部性问题比标记清除收集器所导致的碎片化问题更加的致命。

为了获得好的局部性的唯一的途径就是保证空间足够的大，可以容纳有规律的重用空间。（另一个方法需要依赖于优化措施例如提前提取，但是对于虚拟内存的来说这个方法是不可行的－磁盘不能跟上分配的速度，因为 RAM 和磁盘之间的速度是不能比的）。分代收集器（Generation-based Collection）通过频繁地重用一个较小的空间来解决这个问题。

最后，一个简单的跟踪收集器的工作的时间分布状态在某一个交互的编程环境中也会引起麻烦。它可能引起一个用户的工作产生一个分裂，导致系统突然变得没有反应，在后台做几秒钟的垃圾收集工作，在这样的系统中这种现象是很普通的。对于大型的堆，这种暂停就更加的明显了，暂停的时间可以按秒的量级来计算了，或者如果大量的数据都被分散在虚拟内存中那么这种暂停的时间长度就有分钟的量级了。分代收集器也可以缓解这个问题，因为大部分的垃圾收集之发生在一个小范围的内存中。然而到最后它必须要收集更大的空间，这时候暂停就会变得较长了。对于实时应用程序，这可能也是不可以接受的。渐增式的收集器（incremental collection）可以将工作的粒度缩小，使得垃圾收集运行导致的时间间隔得到一定的保证。

13.2.1 分代垃圾收集算法（Generation-based Collection）

（由于历史的原因，通常来说人们广泛的相信只有拷贝收集器可以被做成分代的，但是事实上并不是这么回事。分代的标记清除收集器虽然较难实现，但是他们确实存在并且也很实用。在这一节中，我们的讲述还是集中于分代拷贝垃圾收集算法。）

给定了一定数目的内存以后，简单拷贝算法的性能被这样一个事实限制了，那就是它必须在每次垃圾收集过程中拷贝所有的活对象。但是在大多数的程序运行过程中，大多数的对象都只能存活一段非常短的时间，而其中一小部分将存活非常长的时间。即使两次启动垃圾收集的间隔时间很短，譬如两者之间只隔数个千字节的分配操作，多数的对象仍然会在收集之前死去，永远不需要被拷贝的。然而对于那

些已经被拷贝过一次的对象，它们中的绝大对数都将存活起码好几次收集。这些对象在每次打扫时都会被拷贝，垃圾收集器花了大量的时间来重复拷贝同一个旧对象，这也是简单收集器无效率性的一个主要原因。

分代算法基于这样一个观察结果，设法去避免大多数这种重复的拷贝。首先，介绍一下**代**（generation）这个概念。在分代算法中，每个对象都有自己的年龄，越早分配的对象年龄越大，也就是越老。而代是一个对象集合，其中包含的对象都具有相似的年龄。分代收集器将所有的对象根据年龄划分归纳到数个代中，每个代也就有了年龄上的区分。新创建的对象都被放到最年轻的代中，当空间不够的时候，收集器就使用根集（包括从老的代指向最新代的所有的指针－参见 13.2.1.2 节）来回收最年轻的代，其他的代并不被回收。经过几次垃圾收集的循环存活下来的对象，就会被提升到下一个老一点的代中。当这个代也满了之后，它和所有比它年轻的代一起参加垃圾回收。因此容纳年轻对象的代非常频繁的做收集工作。这样做的好处是这里大多数的对象通常来说都会迅速地死去，释放空间，拷贝少数存活下来的不会带来太大的开销。在分代算法中，越是老的对象，做收集的频率越低。总的来说分代拷贝垃圾收集算法所得到的好处主要有两点：较少的拷贝开销，以及平均来说更低的由于 GC 运行所引起的延迟时间。

13.2.1.1 GC 所管理的内存空间的逻辑划分

GC 所管理的内存空间包含所有被垃圾收集器分配和回收的对象，有时候就是指堆。它又被划分为两个空间：YOS 和 MOS。（注：我们这里所讲的是对分代算法来说很通用的一种堆逻辑划分。在具体实现算法的时候可能会跟这里有一些不一样的地方，但是基本的原理还是相通的）。

YOS，也就是年轻对象空间（young objects space）：

它包含最近分配的对象。当对象在重复的几次清扫过程以后还能存活，它们将被提升到 MOS 空间，MOS 空间就是成熟对象空间（mature object space）。

YOS 中可能包含数个代，代被编号成 0，1，2。。。，根据年龄的增长来增长编号。代的数目可以在 GC 被重新编译的时候预先设置。每个 YOS 的代又被划分为一个或者几个**梯级**（step）。梯级根据年龄和类型在代中聚集不同组的对象。对梯级也用 0，1，2。。。来标识。一般来说在一个代中最小序号的梯级拥有在这个代中最年轻的对象，这个梯级又叫做 **Nursery**。每个梯级在系统中都有一个独一的数字代号。例如，代 1 可能有梯级 0，1，代 1 可能有梯级 2，3，4，如此这般。一般来说，当一个代在做收集的时候，存活下来的对象将被从它们现在所在的梯级移入下一个梯级中去。已经在一个代中的最老的梯级中存活下来的对象将被移入下一个代中的最年轻的梯级中去。可以简单的说，这就是提升梯级 k 的幸存者到梯级 k＋1。在这种情况下，一个代中梯级的数目就决定了为了将对象提升到下一个代之前要对这个代做打扫的次数。

已经被移入最老的代中的对象（这个最老的代也可以叫做 train_generation）提升时将被 MOS 中，YOS 中的对象具体移到 MOS 中的哪里就只是一个策略问题。在 MOS 中，根据对它作收集所使用的算法，它的内存空间有自己的组织方式，比如在下面所要讲的 Train 算法中，它将被划分成数个 Trains。具体内容参看第 14.2 节对 Train 算法的描述。

在 ORP 的规划中，一个梯级中所有存活下来的对象都将一齐被移动。这就避免了对每个单独的对象都要加上年龄信息。实际上，这个信息是被嵌入到梯级中的。这种技术省略了为每个对象提供年龄计数器的空间，以及为操作这些计数器而耗费的时间。而且这么做也就避免了为每个单独的对象做提升的决定，只需要遵循编译时候已经决定的计划，它能规定每个代中的每个梯级中的存活的对象将被移到什么地方。

使用梯级的方法使得年龄信息可以随我们的希望任意的精细或者是粗糙。ORP 现在的实现就可以在 GC 被创建的时候指定代中的梯级的个数。

每个梯级由一定数目的固定大小的块（block）组成。每个块是 2^I 字节大小，比如，ORP 中一个块的大小是 64KB。一个梯级中的块数目可能随着时间而有变化。梯级中的块通常不需要相邻，而这种灵活性是有代价的：当对象不能充满一个块的时候就会产生碎片。解决方法就是在最年轻的代中，指定最低序号的那个梯级，也就是 Nursery，是连续的空间。如此设置的另一个好处在于因此我们可以更方便地

使用页陷阱（而不是一个显式地限制检查）来探测 Nursery 溢出并触发一个打扫过程。

使用“块”这样一个逻辑概念可以获得四个主要的好处：第一，它允许梯级和代的大小都能轻易地得到改变，因为梯级的内存空间不需要是连续的。第二，它们允许快速地判断一个对象的代，梯级和将要提升到的梯级：对象的地址可以简单地被右移 N 位，然后根据所得值来对包含所需信息的块表进行索引。第三，块和页面陷阱或者卡标记机制都是天然的配合的。第四，它们在某些情况下比使用半区拷贝收集器减少了需要的存储量。如果一个代中在做一次打扫之前有 b 字节数据，幸存者占用了 a 字节，这样一个半区拷贝算法将要使用 2b 字节的空间，但是这里的机制只需要使用 b+a 字节的空间。优越的程度取决于 a/b 的比例大小。

然而“块”引进了一个问题：怎样能够处理大于块大小的对象呢？为了处理这样的对象，就需要提供了一个大对象空间 **LOS**（large objects space）。事实上，通常在 LOS 中放置任何一个将要占用一个块中绝对大多空间的对象。可以设置一个启发式的参数作为门限值，如，一个块的 1/8 空间，凡是所占空间超过它的对象都放到 LOS 中去。也可以规定那些如果存活下来，拷贝的代价太大的对象存储进 LOS。给对象在 LOS 中分配空间使用的是基于 splay 树的空闲列表的方式，而且一旦某个对象被分配在 LOS 中，这个对象就不会再被移动。然而，LOS 对象仍然属于一个梯级，也就是说它仍然是有年龄信息的。对于这一点，可以通过将对象嵌入相应梯级的数据结构中一个双链接列表来指示。当一个 LOS 对象要被提升，只需简单地将它从一个列表上解下，放到另一个列表上去就行了。当清扫工作完成了，任何还停留在被清扫的梯级的 LOS 列表中的 LOS 对象也都被释放。

虽然一个非 LOS 的对象可以使用简单的移动和索引技术来确定它所处的代，梯级和块，但是 LOS 可能会在同一个块中包含从属于不同的梯级和不同的代的对象，因此 LOS 对象难以使用相同的技术来确定自己的从属关系。这个问题可以这样解决：存储一个向后的引用，这个引用从一个 LOS 对象的头部指向包含它的那个梯级。这样如果给定了一个指向 LOS 对象的基址的指针，决定这个对象处于那个梯级就会相对容易一些，但是如果给定的是一个指向这个对象中间的指针，那就需要定位这个对象的头部之后才可以决定它属于那个梯级。

MOS：也就是成熟对象空间（mature objects space）

分代算法的技术对于小堆来说工作良好。然而，对于大堆，随同年轻一点的代来清扫更老的代就可能会导致由 GC 引起的间隔时间过长。通常，这种性质叫做分裂性（disruptive）。为了避免这个情况，需要限制堆中代的数目。任何活过几次清扫的对象都被移到 MOS 中去。

对于 MOS 来说，它的对象都已经成熟，而且 MOS 本身也是很大的。因此不能再用分代算法对它进行收集。一般来说，要对 MOS 使用非分裂性的算法进行收集。一个典型的 MOS 收集算法就是下面我们所要讲的 Train 算法。这里暂且不提。

13.2.1.2 记忆集和卡标记

在分代的算法中，有一个重要的问题要解决，否则分代算法的机制就不能工作。我们知道分代算法的优点的产生源泉主要就在于我们可以只对新代进行打扫，而不对旧代进行垃圾收集。但是数据存活性是一个全局的性质，老内存空间的数据也必须要被考虑到。例如，如果有一个指针从老的空间指向新的空间，这个指针必须在打扫阶段被发现，用来作为遍历的根集之一。否则，它所指向的活对象有可能错被当成是死的，或者导致这个指针不会被正确地更新。任何一个这样的事情发生都会毁坏堆中的数据结构的整体性和一致性。

为了解决这个问题，初始的分代收集技术采用的方法是间接表的方法。这个机制不允许指针从老区直接指向新区；如果发现有从老区指向新区的指针，就让它通过一个表来间接指向。这个间接表将被用作根集的一部分。现在普遍使用的技术则是写栅栏（write barrier）和读栅栏(read barrier)。所谓写栅栏，就是每当对某个对象的写操作发生就运行一些额外的代码，以达到某个目的。这个对象就称之为被置于写栅栏之后，或者称之为被写保护了。同样的，读栅栏的意义就显而易见了。应用在分代算法中，读栅栏技术的效率和间接表的效率相似。一般来说，更经常地把它应用在渐增式垃圾收集器和并发垃圾收集器中。下面，重点讲一下写栅栏技术。写栅栏技术在垃圾收集领域中是很常见的一种机制，主要应用在

两个方面：其一就是我们现在所说的，应用在分代算法中。用来记录那些从老区指向新区的指针的 slots 以备在做打扫的时候可以找得到它们。而不再需要对整个老区进行扫描以得到这些指针。所谓“slot”就是可能包含指针的内存位置。有时候，也用来指包含对象成员的那些内存位置。其二就是应用在渐增式垃圾收集器和并发垃圾收集器中。简单的说，因为在这两种收集器中，应用程序的线程在 GC 还没有完成的时候就有可能运行，改变有向图。因此为了保持数据的一致性，需要收集器和应用程序的线程进行一些协调。在这里，就可能需要来自写栅栏的帮助。例如有些渐增式算法需要维护一个不变量，使得黑色的对象永远不会指向白色的对象，这样就不会破坏算法的正确性。因此算法不允许应用程序的线程将一个指向白色对象的指针写到一个黑色对象中去。为了做到这一点，人们往往将黑色对象放在写栅栏之后，进行写保护，以维护这一不变量。较为详细的关于写栅栏在渐增式算法中的应用可以参见 13.2.2 节，这里就不再冗述了。

记忆集（Remembered sets） 和脏数据位（dirty bits）是实现写栅栏的两种选择。两者各有优点。Dirty bits 可以使得每条存储指令只需要用最小的，有限的开销来维护。而记忆集的优点是它能够精确地，简洁地记录下必要的信息。有研究证明如果将两者结合使用可以得到更好的结果。并且可以更好地避免各自的缺点。Hosking and Hudson 在【HH93】中对这一点进行了阐述。

下面分别对两者进行解释介绍， 最后描述两者的结合形式。

包含从老区指向新区的指针的 slots 的集合被叫做记忆集（remembered set）。在每个存储操作的地点，如果这个存储操作创建了一个从一个旧一点的对象到新一点的对象的引用，系统就必须保证被更新的 slots 被加到记忆集中去。每代都有自己的记忆集，GC 将检查它正在做收集的代的记忆集中的所有的对象，然后产生活对象。

最早的卡标记算法是 Wilson 提出的，在这个方案里堆被划分为卡片，每个卡片大小是 2^k 个字。也就是说，要确定一个有指针修改的地方对应的卡片号码，只需要将这个地方的地址右移 k 位就可以了。在一个单独的位向量中，每个卡片都有一个相应的位。写栅栏算法根据被更新的 slots，将相应的卡片的位标记上。在做 GC 的时候，收集器浏览这个位向量，每当它发现一个被标记的位的时候，就检查在该位相应的卡片中所有的堆上指针。卡标记算法和操作系统使用的脏页（dirty page）方案很相似，区别就在于前者不是对所有的写都跟踪，而是只对指针的写做跟踪。

Chambers and Ungar 改进了 Wilson 的算法，为了减少读，修改这个二进制位而增加的卡标记代码，他们建议为每个卡片使用一个字节来记录它是否被标记了。虽然这个方案比 Wilson 的多使用了 8 倍的空间，但是这个空间开支仍然很小；例如，如果每个卡片的大小是 128 个字节，那么字节映射的大小总共会小于整个堆大小的 1%。这个方案的最大好处就是它的速度快。在【Cha92】中描述的 SELF 系统中，一个存储检查除了本身的存储操作指令外，还需要 3 个额外的 SPARC 指令：首先，计算被更新的字的地址。其次，将这个地址左移 k 位，得到卡索引。最后，将这个索引加上字节映射的基址得到对应的字节，将之标记。

在【Urs93】中，对这个标准的卡标记算法还有进一步的优化措施，在这个优化中，对于大部分的存储指令，我们只需要 2 条额外的指令就可以完成写栅栏算法：原因就在于我们不再需要计算被更新的字的精确的位置。说得详细一点，这里的主要的手段就是放松了原来的卡标记机制所要维护的不变量。原来的标准不变量是：

在卡标记数组中，字节 I 被标记了= = card I 可能包含一个从旧指向新的指针。

现在将它放松为：

在卡标记数组中，字节 I 被标记了= =cards I…I+L 可能包含一个从旧指向新的指针。

这里的 L 是一个小常数。从本质上说，L 就是多给了写栅栏一些在选择需要标记的字节时候的自由－最后被标记的字节可能距离“正确的”字节有 L 个字节远（往地址小的地方跑）。只要被更新的 slot 的偏移量（也就是距离对象开头的距离）小于 L*2^k 个字节，就可以省略计算具体的被更新的地址的操作了。

通常，L＝1 就可以满足所有的存储操作（除了写入数组元素的存储操作）。例如，在 SELF 系统中，卡片的大小是 128 字节，因此所有写入一个对象前 30 个域的存储操作都可以使用这个快速代码序列（一

个对象的前两个域被系统使用，用户使用的第一个域的偏移地址是 8 个字节）。

如果系统可以决定对象的界限，偏差限制（leeway restriction）可以被最大程度地提升：在这种情况下，被放松的不变量可以表述如下：

字节 I 标记了= = card I 或者起始地址在 card I 中的最后一个对象可能包含从旧指向新的指针。

当扫描一个卡片的时候，GC 只需要保证这个卡片中最后一个对象被完全地扫描了，即使它只有一部分是属于这个卡片的。换句话说，即使超过了卡片的最后一个字，GC 仍然继续扫描，直到遇到下一个对象的头。然而，如果遇到非常大的对象，这种做法可能会导致很高的扫描开销。因为扫描的量不再被固定的最大偏差限制，而要由最后一个对象的大小来限制。但是，如果我们对于存储进数组的操作标记精确的卡片，那么这个情况就有所缓解。也就是说如果最后一个对象是一个数组，GC 就不需要继续扫描下面的卡片。

最后做标记的整个序列是这样的：

将地址右移 k 位

加上卡表（card table）的虚拟基地址（保存在一个专用寄存器（dedicated register）中），在这个位置中存储入一个值为 0xff 的字节，0xff 当做一个特殊的值，可以作为标记用。

还有一个问题需要说明，上面我们使用的是卡表的虚拟基地址，为什么要使用它代替实际的基址呢？因为我们每次在对对象引用移位以获取卡索引之前，都必须从其中抽取堆的基址。看下面的代码：

原来的代码：

```
card_table_base[(object_ref->heap_base)>>bits_to_shift]:=MARK;
```

改进以后的代码：

```
virtual_card_table_base=card_table_base->(heap_base>>bits_to_shift)
virtual_card_table_base[object_ref >> bits_to_shift] = MARK;
```

可以看到每次卡标记的操作花费的开销就少点了。需要说明的是，即使采用了最快速的卡标记代码，标记的开销仍然会占整个维护写栅栏的整个开销的 40%到 100％，另一个部分开销是垃圾收集阶段扫描卡片所需要的。

在 ORP 的设计中，卡片又叫做页，因为实际上它把卡片的大小设置为虚拟页的大小。

决定我们感兴趣的指针所要花费的时间是根据记录信息粒度不同而不同的。早先的研究表明了这个就是区别各种不同的写栅栏算法实现的主要因素。使用记忆集的算法可以提供最好的性能，因为它们只记录那些可能包含我们感兴趣的指针的 slots。相反的，浏览被标记的卡片的时间跟卡片的大小成比例。虽然软件实现的卡标记机制可以随便选择卡片的大小，但是页陷阱（page trapping ）机制由虚存页面大小限制（一般都是上千个字节），因此做扫描的开销就会非常的大。然而，卡标记算法的优点是在运行的时候它的开支是可以预先预料到的。每个存储指令都加上固定的额外的工作就可以了。而记忆集算法通常更加的复杂，它要根据使用的数据机构不同而不同。通常它的开销要比卡标记算法要大。并且一个给定的存储操作可能会引起的开销也是不定的。比如说，一个被修改的 slot 可能已经在记忆集中了，否则它就要被加入记忆集中去。如果遇到溢出，集合必须增长以容纳下新的条目。这些条件代码在现代的管道体系结构上运行可能会导致致命的错误。

综上所述，两种方法各有优缺点。最好就是结合两种方法，记忆集方法的精确和卡标记方法的简单。

可以这样来做，在每次作垃圾收集的清除前，扫描被标记的卡片，这些卡片中的旧指向新（older-to-younger）的指针被总结进入合适的记忆集中，也就是它指向的代的记忆集中去。把这作为清除的基础。然后卡片被看作已经是干净的了，也就是说没有标记。下面的清除只需要通过重新扫描从上一个清除过程结束以后又被标记的卡片来更新记忆集。

这样，我们通过将容纳 older-to-younger 指针的 slots 总结进记忆集中（同时清除卡标记），就避免了在每次收集的时候重复的扫描卡片。用一句话来概括，这里描述的混合机制用卡标记机制来跟踪指针存储，用记忆集机制来总结感兴趣的指针。实现了两者的优势互补的结合。另外还有一个问题需要说明，因为对于记忆集机制来说，在最差的情况下，一个给定的代的记忆集会跟整个旧代一样大，所以这时候可能反而是用扫描卡片的方法所花的代价要小一些。针对这种情况，可以改进上述的混合机制，检测当

记忆集变得过大的时候，就动态地切换到卡片扫描机制。也就是说如果一个给定的卡片包含了很多 older-to-younger 指针，以至于不能有效地用记忆集来记录，那么这个卡片就被标记。这样这些 older-to-younger 指针就会被在下一次做清除中扫描的时候重新发现。GC 可以在每次清除的时候重新评价这个决定，可以切换回去，在记忆集中记录 older-to-younger 指针。阐述这个问题的目的在于说明，应用这两种机制实现写栅栏的时候完全可以灵活地根据需要来进行搭配，以提高效率。

13.2.2 渐增式跟踪型垃圾收集（incremental tracing garbage collection）

在第 13 章开头，我们讨论了对于有交互的应用程序和实时应用程序，为了满足对时间上的严格要求，有必要使用细粒度的渐增式垃圾收集。这种垃圾收集的算法可以在一个垃圾收集的循环中间暂停下来，让应用程序继续运行，然后再使得垃圾收集过程继续。也就是说收集器的操作可以渐增地进行，因此叫做渐增式收集器。

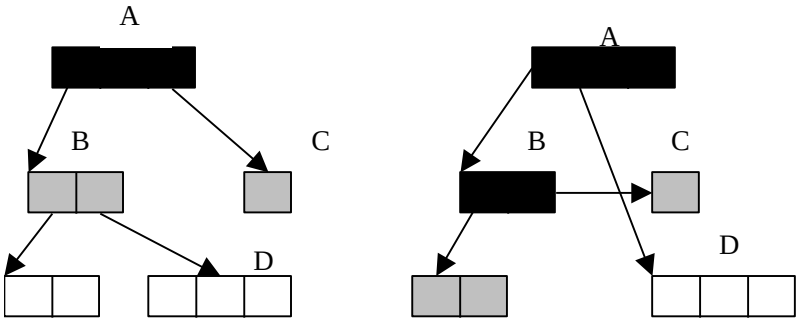
渐增式跟踪的困难就在于当一个收集器在做遍历以得到一个可达的数据结构的图的时候，这个图可能会被运行着的程序在收集器看不到的时候改变。因此，讨论渐增式收集器的时候，我们通常是将运行着的程序称为改变者（Mutator）。一个渐增式垃圾收集的方案必须有某种方法来跟踪可达对象图的变化，也许面对那些变化，需要重新计算遍历图中一些部分。

13.2.2.1 三色标记

在对拷贝式垃圾收集进行介绍的章节中，我们讲到了三色标记抽象的概念。这个概念对于理解渐增式垃圾收集器来说是非常重要的，是它的基础。简单回忆一下，三色标记的抽象的思想就是垃圾收集算法可以被描述为一个遍历可达对象图和将它们染色的过程。做垃圾收集的时候，所有的对象从概念上讲标为白色，在垃圾收集的末尾，那些将要保留下来的对象必须被染成黑色。当所有的可达的对象都被标成黑色的以后，遍历过程就结束了。

上文我们说到渐增式算法允许 Mutator 在收集过程中改扮可达对象图。可以发现，如果 Mutator 做了下面这两件事情，就会有一个原本可达的对象被收集器遗漏。因此也就无法保证算法的正确性：第一，将一个指向白对象的指针写入一个黑对象。并且，第二，所有的从任何灰对象到该白色对象的路径都被毁掉。图 13.3 给出了 Mutator 对可达对象图的一种改变，这种改变就会导致收集算法失败。假设对象 A 已经被彻底地扫描过了（因此它是黑的）；它的后代已经被遍历到了，因此是灰色的。假设改变者将 A 到 C 的指针和 B 到 D 的指针修改成从 B 指向 C 和从 A 指向 D 这两个新的指针。现在指向 D 的唯一的指针在 A 的域中，A 现在已经被收集器扫描过了。如果遍历继续进行而不进行协调，C 将会从 B 再一次遍历到，而 D 就永远不会被遍历到。

因此 Mutator 在运行的时候需要注意维护可达对象图的一个性质－三色不变量（tri-color invariant）。只要保证了有向图的这种性质，就可以保证收集器不会漏掉可达的对象，同时允许 Mutator 在收集过程中改变该有向图。三色不变量包括两种类型：强三色不变量和弱三色不变量，分别对应上面说的两点。也就是说，保持强三色不变量的可达对象图中不允许出现从黑色节点到白色节点的有向边。而保持弱三色不变量的可达对象图中所有的被黑色对象所指向的白色对象都可以从某灰色对象可达。如果 Mutator 创建了一个从黑色的对象指向白色的对象指针，它必须和收集器做某种程度的协调，保证收集器对系统的状态记录可以随着更新。



改变前状态　　改变后状态

图 13.3 对三色不变量的违反情况

13.2.2.2 渐增式方法

有两个基本的方法用来协调收集器和改变者，维护上述的不变量。其一是使用读栅栏，它探测什么时候 Mutator 试图访问一个指向白色对象的指针，就立刻将这个对象变成灰色；既然 Mutator 不能够读那些指向白色对象的指针，它就不能将它们写到黑色对象中。因此它维护的应该是强三色不变量。Baker 的渐增式拷贝收集器也许是最著名的应用读栅栏的渐增式拷贝收集器了。细节参见【Bak78】。其二是使用一个写栅栏，当程序试图将一个指针写入一个对象中的时候，这个写操作都会被捕捉到。并在必要的情况下执行一些额外的操作以维护不变量。

写栅栏方法也可以根据所能解决问题的不同方面而划分为两个不同的类别。应用强三色不变量，那么当发现有其他的从灰对象指向白对象的指针，这个白对象就会被保留，如果没有，那它肯定就是垃圾，不需要被保留的了。如果应用的是弱三色不变量，那么对象可以从初始的指针到达并被保留下来。两种写栅栏方法就集中在问题的这两个方面。

开始时快照法（Snapshot-at-beginning）收集器应用的是弱三色不变量。它使得收集器处理任何 Mutator 要修改的引用，因为这个引用可能是指向一个白色对象的路径中的一部分。它会存储这些指针，这样收集器可以发现它们。因此指向白色对象的路径不可能被破坏以后就没有其他的路径指向这个白色的对象了。它的名称的来源在于它跟踪在每个收集循环开始的时候就存在的引用。不过这并不意味着在收集器运行期间被创建的相关引用就不需要被跟踪了。注意，不要求跟踪所有的对引用的修改，只需要跟踪那些会使得收集器漏掉任何一个可达对象的更新操作就可以了。最简单的和最有名的快照收集算法就是 Yuasa 算法，参见【Yua90】。如果一个 slot 被写入，重写的值首先被存储下来并压入一个标记栈用作后来的检查用。这保证了没有对象会对于垃圾收集器的遍历来说变成不可达的－所有的活在垃圾收集开始的时刻的对象都将被可达，即使指向它们的指针被重写了。在图 7 的例子中，当从 B 指向 D 的指针被用指向 C 的指针重写的时候，这个指针就会被压栈。

渐增式的更新（incremental update）收集器应用的是强三色不变量。如果一个指向一个白对象的指针被拷贝到一个黑对象中，它就将该白色对象或者该黑色对象染成灰色。它的名称的来源就在于它渐增式的更新收集器对可达对象的视图，以跟踪 Mutator 对它造成的变化。渐增式更新算法一般来说都被应用到并行系统中，因此在很大程度上被目标是单处理器的实现者所忽视。也许这种算法中最有名的是 Dijkstra et al.【DLM+78】。

最后，顺便提一下并发垃圾收集算法（concurrent garbage collection），它和渐增式收集器的区别在于，渐增式垃圾收集器和改变者的工作交织进行，这种交织通常是顺序的。如果将两者看成独立的线程，它们被调度成为协同程序。而并发收集器和改变者的交织运行是无序的，也就是说两者可能同时想访问或者更新同一个数据，抢占可能发生在任何时间。在多处理器上，应用并发收集器允许 GC 线程和其他的应用程序的线程同时运行。因此虽然它面对的问题和渐增式收集器类似，但是要更加的困难和复杂。

第14章 一些重要的垃圾收集算法

14.1 Sapphire-Copying Garbage Collection without stopping the world

Hudson 和 Moss 在【HM01】这篇论文中，阐述了 Sapphire 算法的思想和过程。Sapphire 是一个并发垃圾收集算法，是为类型安全的堆分配（type-safe heap-allocating）语言设计的。Sapphire 扩展自复制拷贝收集算法（Replication copying collection），这是一种渐增式拷贝垃圾收集技术，在这种技术下，应用程序线程观察和更新主要的是对象的旧拷贝。

总的看来，Sapphire 算法有如下一些特点：

1. Sapphire 算法的应用目标是多处理器服务端程序。这种程序的特点就是有很多的线程，并且在主存中有相当量的共享数据。Sapphire 算法在这样的共享存储器多处理器的系统上还具有可扩展性。能够运行在小规模到中等规模的共享存储器多处理器平台上并且工作良好
2. Sapphire 算法的重点在于尽量减少为了支持收集器，任一个应用程序的线程必须被阻塞的时间长度。收集器并行地和所有的 Mutator 线程，也就是应用程序的线程一起运行。
3. Sapphire 算法渐增式地 Flip 所有的线程。也就是说它一次只 Flip 一个线程。所谓“Flip”也就是将一个线程的视野从旧拷贝变换到新的拷贝。而以前的算法都要包括这样一个步骤：先将所有的应用程序的线程停掉，遍历它们的栈，然后这些栈中的指针都将被重新定向，从对象的旧拷贝转到指向对象的新拷贝。
4. 从实际的实现来看，它还避免了读栅栏。

14.1.1 GC 所涉及到内存逻辑

Sapphire 算法从逻辑上把它所管理的堆分成了 U 区和 C 区。另外，这里对进程栈也进行了一些描述。

U：这是堆上的一个空间（堆由所有的线程潜在的共享），U 代表的意义是 uncollected。在收集器被激活的时候，它里面的对象不会被回收。也就是说 U 区对象不在等待垃圾收集器收集的对象集合之中。为了方便，我们也将所有的非对象内部的 non-thread-specific slots 划归这个空间。

C：也是堆上的一个空间，C 代表的意义是 collected。C 区的对象集合就是等待垃圾收集器收集的对象集合。C 区又被划做：

－**O**：代表的意义是老区（old place）。当收集器开始工作的一开始，C 区的所有对象存在此地，这里的数据可称作对象的旧拷贝（old copy）或者对象的 O 拷贝。

－**N**：代表的意义是新区（new space）。这里存放的是通过这次收集循环以后存活下来的对象的拷贝。这里的数据可被称为对象的新拷贝（new copy）或者对象的 N 拷贝。

C 区中只包含对象，也就是说，它没有单独的 slot 存在（bare slots），不像 U 区，它包含那些非对象内部的 slots。

S：栈，每个线程都有一个单独的栈，归这个线程私有。S 区包含 slots，但是没有对象，也就是说，将不会有指针从堆对象指向栈。为了简单起见，我们在 S 中还包含其他线程本地 slots，例如那些对应于包含引用的寄存器的 slot。图 14.1 给出了上述各个空间的图示。

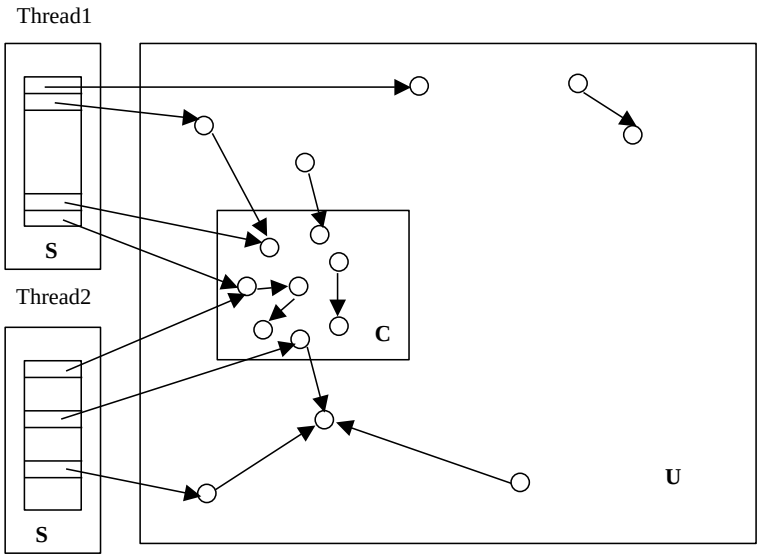


图 14.1 栈 S，U 区和 C 区

哪个对象被收集或者不被收集（U 和 C）对 Sapphire 算法来说是一个随意的决定。例如，你可以使用分代或者成熟对象空间（Train 算法）中的相关决策。另外，Sapphire 算法也需要写栅栏技术的支持，因为它只收集 C 区的对象，因此跟分代算法中所说的一样，它需要发现从 U 指向 C 的指针，我们知道一个例如记忆集这样的数据结构可以省去扫描 U 区的工作。不过 Sapphire 算法的写栅栏和分代算法中的也有不一样的地方。原因在于，Sapphire 假定新对象是分配在 U 区，而不是 C 区。注意到这个规定有助于保证标记和拷贝过程的终结，因为这样做 C 区不会有增长。同时，这个规定对于创建每个新对象都要加上额外的写栅栏的开销，因为 Sapphire 算法需要处理它们指向 C 区对象的 slot。

14.1.2 Sapphire 算法

Sapphire 算法的整个过程分为两大阶段。

第一个阶段，称之为标记和拷贝阶段（mark and copy），过程如下：

a) 决定 O 区中的哪个对象可以从 U 区中的和 S 区中的根集可达。

b) 在 N 区建立可达的 O 区对象的备份。在标记和拷贝阶段，Mutator 读和更新的只是对象的 O 拷贝。任何给定的可达对象的 O 和 N 拷贝保持松散的同步：在两个同步点之间，由 Mutator 线程对 O 拷贝的任何变化都将会在通过第二个同步点之前传播给 N 拷贝。这里利用了 Java 虚拟机规范的内存同步规则。这种松散同步的关键思想在于更新两份拷贝的动作不需要是原子的和同时的。如果所有的 Mutator 线程都在同步点，一旦我们处在 Mutator 可以同时看见 O 和 N 拷贝的收集阶段，O 和 N 拷贝将会是一致的。我们称这个性质是 O 区和 N 区之间的**动态一致性**（dynamic consistency）。

第二个阶段，称之为 Flip 阶段，这个阶段致力于 Flip 栈 S 和 U 区的指针，使得它们都指向 N 区而不是 O 区。在 Flip 栈 S 的时候，是渐增式的一个线程一个线程地 Flip 各自的线程栈。在 Sapphire 中，这个阶段使用一个写栅栏（也就是说，没有读栅栏）。Sapphire 允许还没有被 Flip 过的线程访问对象的 O 和 N 拷贝，即使它们是同一个对象的拷贝。而以前的并行拷贝收集器或者将访问 O 对象的动作转向到它的 N 拷贝（使用一个读栅栏），或者保证所有的访问都是访问 O 对象（然后，一次性的 Flip 完所有的线程）。也就是说，是渐增的 Flip 动作和不使用读栅栏使得出现了同时访问 O 和 N 拷贝的可能性。因此 Sapphire 算法需要一个稍微严格一点的对两份拷贝更新的同步机制。

另外，这种可能性也会影响指针相等性的比较，因为我们在这种情况下必须能够回答指向同一个对象的 O 和 N 拷贝的指针从 Java 编程员的角度来看是相等的。Brooks 在【Bro84】中实现的 eq 和这里需要的相等性比较操作是类似的，可以参考。

14.1.2.1 标记和拷贝阶段：获得动态一致性

在这个阶段中，规定的子阶段有：标记，分配和拷贝。

现在用传统的三色标记规则的术语来解释这一阶段所作的事情。跟前面讲的一样，这里仍然需要遵守一个颜色限制规则－no-black-points-to-white 规则：那就是一个黑色的 slot 不能指向一个白色的对象。Sapphire 将 S 中的 slot 默认地看成是灰色的，因此它们可能会包含指向任何颜色的对象的指针。这意味着将一个引用存储在一个栈的 slot 中就不需要对它进行任何工作以保持颜色限制规则了。然而对共享的内存空间（全局和堆对象）的更新就需要以写栅栏的形式进行一些措施以维护上述的颜色限制规则。

初始，我们将堆中所有已存的对象和 slot 都看作是白色的。随着收集的进行，对象从白色，到灰色，再到黑色。在 Sapphire 中，黑色的对象不可能再变成其他颜色，不可能再被重新扫描。标记阶段的目标是使得所有的可达的 C 区的对象变成黑色。更进一步，在标记阶段开始的时候任何不可达的对象将保持是白色的，收集器最后将会回收它们。新分配的对象被看作是黑色的。

标记子阶段：它又包括三个步骤：

1. 标记前（Pre-mark），这个步骤安装标记子阶段的写栅栏，
2. 根集标记（Root-mark），这个步骤处理不在栈里的那些根。
3. 堆和栈的标记（Heap/stack-mark），最后来完成标记。

在标记阶段所使用的写栅栏就是普通的分代算法所使用的写栅栏，不同的地方仅在于它只是简单地将该被标记的对象排队，等待以后让收集器进行标记，Mutator 并不直接进行任何标记动作。这个写栅栏阻止任何指向白色对象的指针被写入堆中。它将该白色对象入队列并默认地看成是灰色的。因此这个写栅栏保证了 no-black-points-to-white 规则。

为什么是排队而不是直接让 Mutator 来做标记的工作？因为最终，在实际的实现版本中，要把标记和拷贝结合起来，标记步骤将包含分配空间给一个对象的新拷贝。如果我们使得 Mutator 来做这个分配将使得同步进入瓶颈。所以这里通过使得收集器做分配和拷贝来避免这个瓶颈。更进一步说，每个 Mutator 有它自己的队列，所以排队没有同步开销。当收集器扫描一个 Mutator 的栈的时候，它同时会清空这个队列，将其队列中的内容插入收集器输入队列。

根集标记步骤巡逻 U 区的 slots，并将任何被这些 slots 指向的 C 区中的白色对象变灰，并对它使用标记阶段的写栅栏操作，然后该 slot 就被认为是被染黑了。注意，这个步骤中，新对象被看作是黑色的。既然收集器在调用写栅栏，因此相关的对象被排队，即刻就会出现在收集器的输入队列中。

虽然我们可以扫描整个 U 区，来发现相关的 slot，但是使用分代算法中所描述的记忆集数据结构来定位相关的 slot 显然会更加的有效率。

在堆和栈的标记步骤中，收集器从它的输入队列，一个显式的灰色（标记的）对象的集合，和线程栈开始工作。对每个被排队的对象，收集器检查是否这个对象已经被标记了。如果是这样，收集器就遗弃这个队列入口；否则，就标记这个对象，将它放到灰色对象的集合中以便以后扫描它。对于每一个灰色集合中的对象，它包含的所有的 slots 都被在指向的对象上使用写栅栏染黑以后，这个对象自己就被认为是黑色的。收集器将重复地进行这一过程，直到输入队列和显式的灰色对象集合都变成空的。

注意，一个对象可能会被一个或者不同的 Mutator 线程排队多次；然而，最终，收集器将标记这个对象，它将不会再被 Mutator 排队了。

堆和栈的标记也包括寻找指向 O 对象的 S 区指针。为了扫描一个 Mutator 线程的栈，收集器循环地一次阻塞一个线程，将被阻塞的线程简短地停在垃圾收集安全点上（GC safe point），然后扫描这个线程的栈和寄存器来寻找指向白色 O 对象的指针，同时对每个这样的指针激发写栅栏。当线程被停掉的时候，收集器将被线程排队的对象移到收集器的队列中去。收集器重新启动线程，然后处理自己的输入队列和灰色对象集合直到它们都再次变成空的。当所有的线程栈中都没有新的灰色对象出现时，这个阶段才结束。

这里列举两个对栈扫描的改进措施。第一，从它们最后一次被扫描之后就一直挂起的线程不需要再被扫描。第二，如果我们使用栈栅栏【CH98】，我们可以避免重新扫描那些自从上次扫描过后就再也没

有被线程重新放进栈里的旧帧。

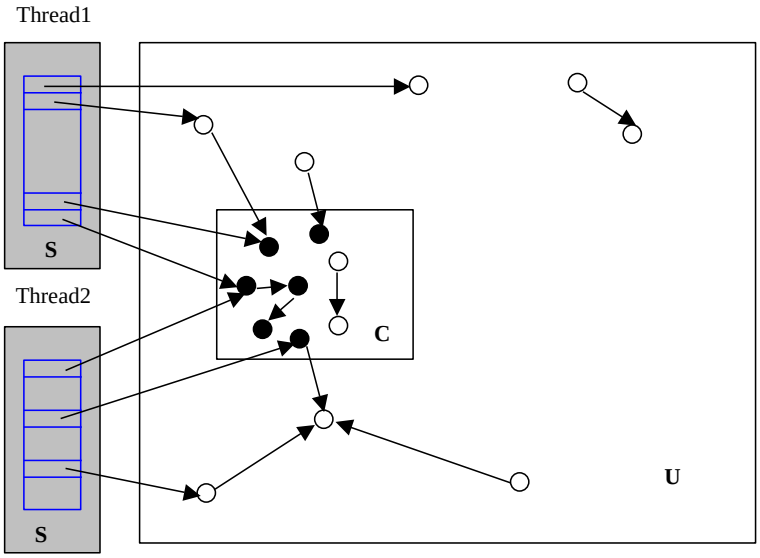


图 13.2 扫描栈和堆，对 C 区中的可达对象进行标记

分配和拷贝子阶段：

标记阶段建立了哪个 O 对象是可达的。一旦我们决定了可达的 O 对象，我们就开始给每个可达的 O 对象分配一个 N 拷贝（这是分配阶段），然后拷贝这个 O 拷贝的内容到已经分配好的 N 拷贝中（这是拷贝阶段）

分配子阶段：一旦所有的可达 O 对象都已经标记了，收集器就为每个这样的对象分配一个 N 拷贝的空间，并且设置 O 拷贝的 forwarding 指针指向这个为 N 拷贝保留的空间。

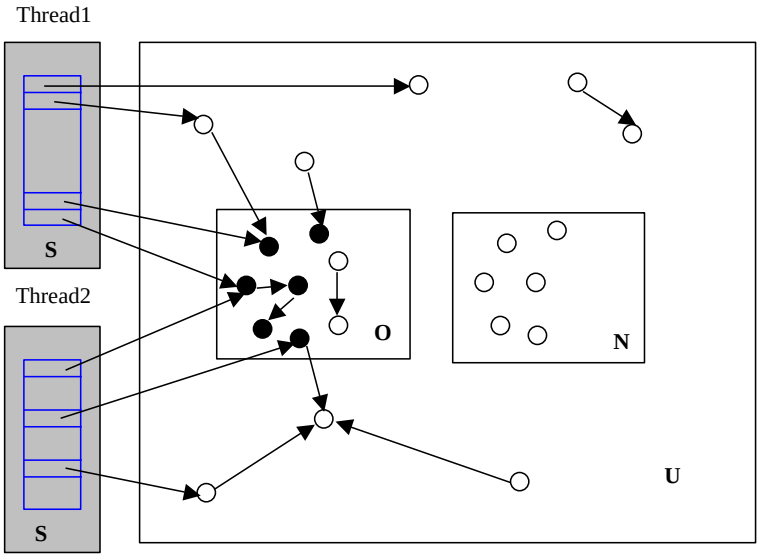


图 13.3 分配子阶段，保持动态一致性

拷贝子阶段：这个阶段包含两个步骤：

1． 拷贝前（pre-copy）步骤：建立拷贝子阶段的写栅栏。

2． 拷贝（copy）步骤：

为了保持 O 和 N 拷贝之间的动态一致性，拷贝子阶段需要一个新的，不同于标记子阶段所使用的写

栅栏。它不仅应用到在堆上写指针的值，也应用到在堆上写非指针的值。而且当执行一个存储指针的操作时，不管这两个相关的对象各自是在什么区中，都需要写栅栏的保护。这点就不像我们在分代算法中所描述的写栅栏，在那里它只有在发现被存储的指针是一个 older-to-younger 指针的时候才会需要。最后，注意，一个在 N 拷贝中的指针总是指向 U 或者 N 空间，而不是 O 空间；我们利用写栅栏保持这样一个不变量，那就是 N 拷贝永远不会指向 O 拷贝。下面用伪代码写出满足所有这些要求的写栅栏：

```
//copy phrase write barrier:
//假设有这样一个更新操作：p->f=q ， p,q 可能在 U 区或者 O 区
copy-write-barrier(p,f,q) {
    if (forwarded(p)) {//如果 p 对象已经被拷贝到 N 区了
       pp=forward(p);//寻找 p 对象在 N 区的拷贝
       qq=forward(q);//如果 q 不是指针，那么 qq=q。
       pp->f=qq;//更新 N 区拷贝
    }
}
```

拷贝步骤拷贝了每个黑色的 O 对象的内容到相应的被分配好的 N 拷贝中去。如果一个被拷贝的数据是一个指向 O 对象的指针，它首先调整这个指针，让它指向这个对象的 N 拷贝。当收集器拷贝一个对象的内容的时候，Mutators 可能会并发地更新对象。上述写栅栏可以使 Mutators 将这种更新从 O 拷贝传播到 N 拷贝，保持了两者的动态一致性。

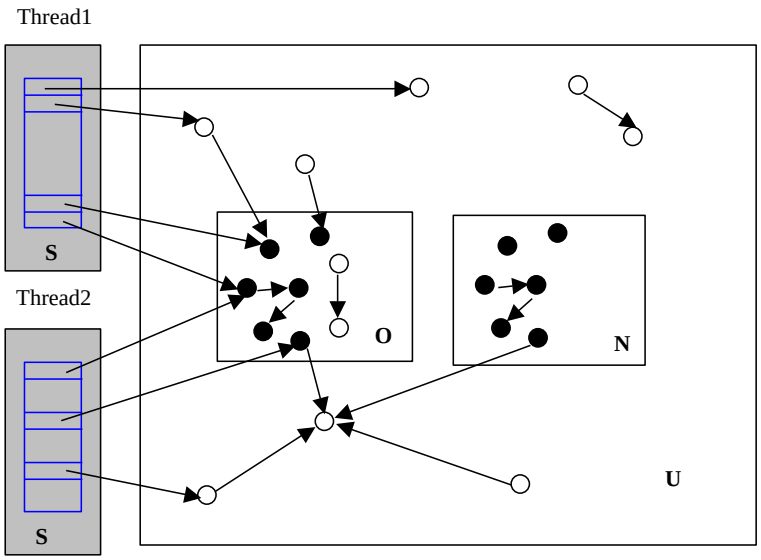


图 13.3 拷贝子阶段

14.1.2.2 Flip 阶段

这个阶段分为：pre-flip 子阶段，heap-flip 子阶段，thread-flip 子阶段和 post-flip 子阶段。这个阶段的目标是系统地除去可能被某个线程看见或者使用的 O 指针。

Pre-flip 子阶段安装一个在 Flip 阶段的写栅栏，帮助跟踪可能包含了指向 O 对象的指针的 slots，阻止它们被写入 U 区或者 N 区中的对象中。它必须在 heap-flip 子阶段之前安装，否则，没有被 Flip 过的线程可能把 O 指针写到 U 区的 slot 中去。

以 N 拷贝不会指向 O 拷贝这样一个不变量开始，建立并维护没有 U 或者 N 的 slot 会引用 O 拷贝这样的规则。heap-flip 子阶段和 thread-flip 子阶段开始系统地除去任何指向 O 拷贝的指针。并利用上述的写栅栏维护这一不变量并保持两份拷贝的同步。

heap-flip 子阶段扫描每个可能包含一个 O 指针的 U 区的 slot，将这个 O 指针指向它们的 N 拷贝。

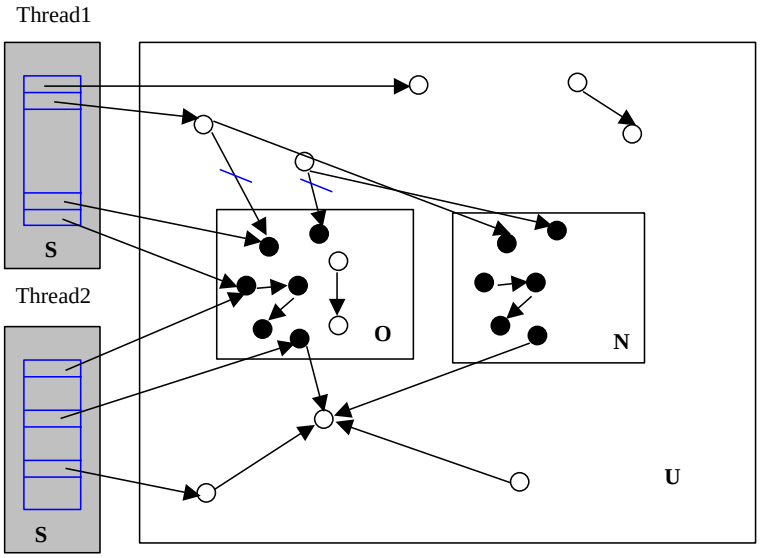
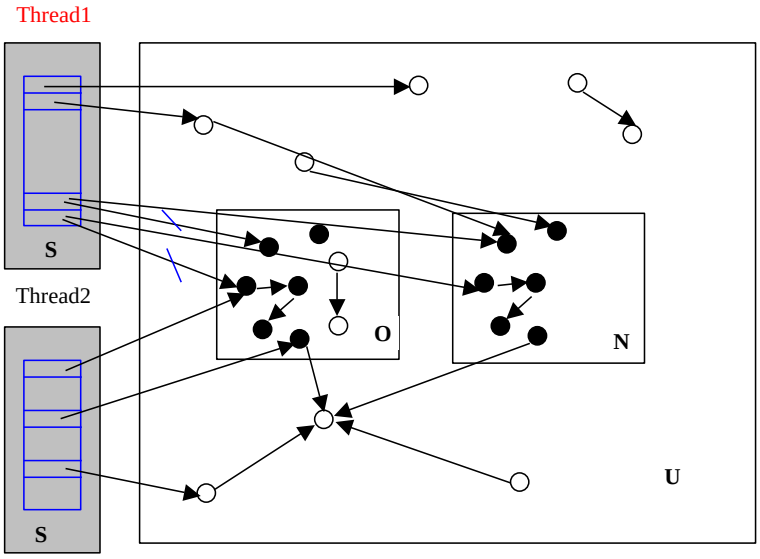


图 13.4heap-flip 子阶段

然后，thread-flip 子阶段开始 Flip 线程。还没有被 Flip 的线程可能拥有指向 O 和 N 拷贝的指针，甚至是指向同一个对象的两个拷贝的指针，但是已经被 Flip 过的线程永远不会指向 O 拷贝。只要存在任何没有被 Flip 过的线程，所有的线程都必须同样更新 O 和 N 拷贝。因为我们利用了 Java 互斥语义的方式，O 和 N 哪个先更新并没有关系。



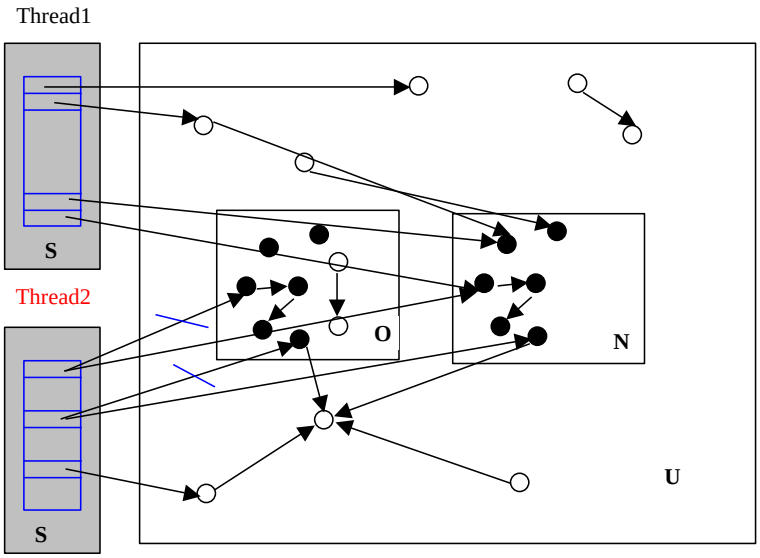


图 13.5 thread-flip 子阶段

现在，所有的线程都只能看见 N 区。post-flip 子阶段做的事情是清扫和释放。一旦所有的线程都已经被 Flip 过了，我们可以关掉特殊的写栅栏，返回到通常意义下的写栅栏，也就是不在垃圾收集运行时所使用的写栅栏。然后回收 O 区。

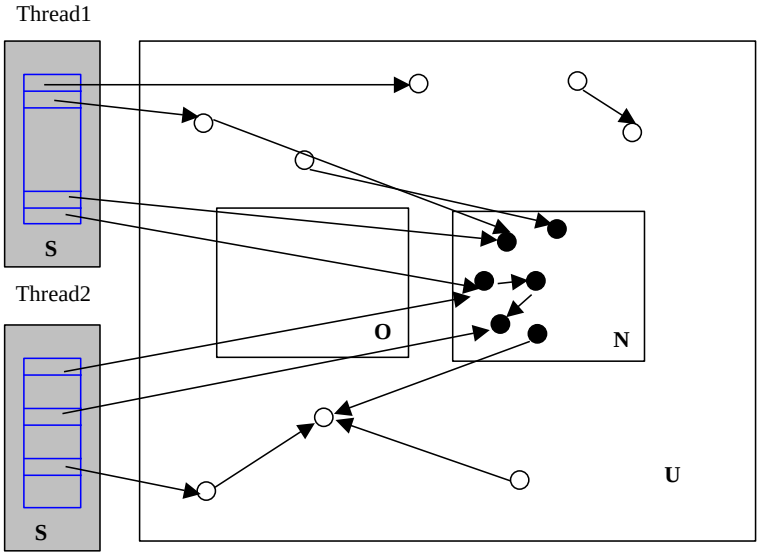


图 13.6 释放 O 区

14.2 Train 算法

Hudson 和 Moss 在【HM92】这篇论文中对 Train 算法进行了描述。这是一种渐增式的垃圾收集算法，具有非分裂的性质。它扩充自分代垃圾收集器算法，目的是用来非分裂性地收集大型的旧代，也就是成熟对象空间。Train 算法在每次收集的时候处理成熟对象空间的有限大小的片；它能保证最后将收集所有的垃圾。这个算法没有假设依靠任何特殊的硬件上或者操作系统的支持，例如，对 forwarding 指针或者受保护的陷阱的支持。这个算法拷贝对象，所以它很自然的支持紧缩和重新聚集。

前面所讲的分代垃圾收集器算法在减少整个垃圾收集的时间方面非常的有效。且其大部分的收集过程也都呈现非分裂性。然而，在更大的旧代需要被收集的时候它还是会表现出分裂的性质来。为了用一种非分裂的风格来回收旧对象，Train 算法被提出。它具有下面的性质：

渐增性（incremental）：每次渐增性的收集的时候它最大移动的字节数是很小的。

紧缩和聚集：这个算法通过拷贝支持对对象的紧缩和重新聚集。

安全性：不会回收活的对象。

完整性：对于大型的循环结构的对象也可以统统回收。

Train 算法的提出在很大程度上借鉴了 Bishop 在【Bis77】中陈述的工作。Train 设计者的贡献主要在于如何把 Bishop 的概念化工作应用来创建一个具有上述特性的收集器。它被 Intel，Sun，IBM，Novell 以及各种学术研究者所使用。

下面我们开始介绍这个算法的具体内容。在这里，我们将会描述成熟对象空间，它的结构和它的回收算法。用来实现这个算法的组件是和那些被用来实现为收集年轻对象而实现的分代算法是一样的。特别的如，块，记忆集，和扫描和拷贝机制对于收集 MOS 和 YOS 空间都是一样的。首先我们将描述成熟空间是怎么样被划分为各种逻辑空间的。第二，讨论 Train 算法中是怎样应用记忆集机制来跟踪 MOS 不同空间之间的指针的。第三，我们将陈述一些规则，它们决定成熟对象究竟应该被放在什么地方。第四，展示对于 MOS 中一个空间的收集过程。这样一个过程的结果就是移动相关的对象，最后使得任何不可达的对象被孤立并回收。

14.2.1 MOS 的结构

MOS 被划分为一个个的空间（areas），就像年轻对象空间被划分为代一样。一个空间的结构和一个代的结构是相似的，因为它们都一样，所有的指向这个空间的指针都将在被在清扫的时间段中被发现（也就是说，每个空间都有一个记忆集）。一个空间包含一个或多个块。这些块和年轻对象空间使用的块是一样的，两者还共享同样的薄记（bookkeeping）函数，包括快速决定一个块所在的空间，并且在一次收集过程中，决定一个对象应该被拷贝到哪个空间。另外，跟堆中的代不一样，年龄信息不再是令人感兴趣的了，所以这里的空间不会包含梯级。

空间是限定大小的。这样每个单独的空间都可以被快速地收集。收集器每次只对一个空间工作。但是这样一来会产生一个问题。在 Bishop 算法中为了回收一个多空间循环结构，因为本地的信息不足够来做出这个对象是不是在全局来说都是不可达的这样的决定，所以必须将一个跨多空间的循环垃圾移到一个单独的空间中，以便于回收它。既然 Bishop 没有限制空间的大小，这个方法就没有任何问题。然而，在 Train 算法中，如果我们限制了一个空间的大小而且要被回收的相连接的数据结构比一个单独的空间要大的话，那该在那么办呢？可以说，Train 算法最关键的贡献就在于既保证了回收大型数据结构垃圾的能力，同时又给空间加了大小限制，从而使回收变成非分裂性的。

为了更进一步描述 Train 算法，我们首先介绍一些在算法描述中使用的术语。从成熟对象空间外指向一个成熟对象的指针称为**根指针**。根指针存储在年轻对象空间，LOS，栈，寄存器，和静态空间中。从根直接可达的对象称为**领导者**。不是直接可达的，但是仍然可以从成熟对象空间中的对象可达的对象称为**追随者**。

Train 算法名称的来自于使用了一个 Train 的比喻描述这个算法。一个空间我们可以把它想象成一个 Train 的**车厢**（car）。这些车厢限定了每次垃圾收集被激起的时候所作的工作量。一个车厢组包含了一个互相链接的对象的数据结构，这样的车厢组被称为 Train。Train 用来给大型的相关的对象分组，这样它们可以被看作一个单元来管理。这就是对空间限制大小所产生的上述问题的解决方法的关键结构。

14.2.2 根和记忆集

每个空间都有一个相关联的记忆集，它包含所有的从这个空间外面指向这个空间内部的对象的指针。然而，因为只有当所有的年轻对象空间都被清扫了，成熟对象空间的空间才需要开始做垃圾收集；并且所有的清扫过程已经处理了所有的根（栈，寄存器，静态空间），所以每个空间的记忆集只需要跟踪从其他的 MOS 空间来的引用就可以了。一个 Train 的记忆集就是它所包含的每个车厢的记忆集，然后减去

Train 内部的指向关系。

Train 算法选择收集空间的顺序是一个 round-robin 顺序。设想我们给每个空间赋予了一个序列号，当一个空间被清扫了，它就被赋予了下一轮最高的序列号。这样记忆集只需要记录从具有高点序列号到低点序列号空间的引用就可以了。当一个空间正在被收集，它的序列号将会是最低的，因此我们能够发现所有的从其他空间指向这个被收集的空间的引用。

这样处理记忆集的方式带来了两个好处。第一，我们减少了记忆集信息的总量。如果从指针的指向关系来看，它们是均匀分布的，记忆集的大小会是任意选择一个空间做收集的方法所导致的记忆集的一半大。当然，对于指针的分布究竟是什么情况，并没有很清楚的研究结果。所以这个好处的程度并不清楚。第二，也许是更重要的一点，当一个空间被清扫的时候，我们不需要更新其他空间的记忆集。这是因为没有哪个被清扫的空间的信息还可能被记录在其他空间的记忆集中，它已经是最低序号的空间了。

14.2.3 收集 MOS 中的一个空间

就跟前面提过的一样，我们用 round-robin 的顺序处理空间，一次回收一个空间。

在回收一个车厢之前通常都有一个检查：如果没有根指针指向将要被回收的那个车厢所属的 Train，那么我们就检查这个 Train 的记忆集。如果这个记忆集是空的，那么整个 Train 都将被回收，不需要进一步的动作。

当一个车厢正要被收集，可以称它为 from car。在 from car 中每个可达的对象都有一个相关联的 to car，也就是可达对象将要拷贝到的空间。它究竟是哪个车厢是由这个对象是怎么被引用的而决定的。首先，我们拷贝 from car 中任何由根所指向的对象，也就是领导者，放到一个新创建的 Train 或者其他已存在的 Train 中。究竟选择哪个 Train，是一个策略问题，并不会影响我们算法的正确性。接着，我们扫描这个被拷贝的对象，然后用典型的拷贝收集器的风格，拷贝所有在 from car 中的追随者。

同时，我们将从年轻对象空间被提升到 MOS 来的对象移到拥有指向它们的引用的那些 Train 中去；如果这些被提升的对象是直接被根所指向的，那么它们可以移到任何 Train 中。因为 YOS 中被回收的代的大小都是被限制的，因此被提升的对象的数据量也是有限的，所以我们可以限制提升引起的分裂性。

到这里，from car 中仍然可能包含了可达的追随者对象，它们只可能是从其他的车厢可达的。使用 from car 的记忆集，我们可以定位所有的从当前 Train 外面指向 from car 内部对象的引用，并将所指向的对象移动到包含这个引用的 Train 中去。最后 from car 中剩下的可达对象仅仅是从同一个 Train 中其他的车厢可达的对象了。这些对象都将被移到这个 Train 的最后一个车厢中去。最后，from car 中就只剩下不可达的对象，然后回收该车厢。Train 算法

如果对象要移动到的 Train 已经满了，我们就给这个 Train 加一个车厢，然后将对象拷贝在这里。在任何一种情况下，一个被从当前 Train 外面引用的对象都要被移到其他的 Train 中去，将垃圾都集中放到尽量少的 Train 中，最后集中到一个单独的 Train 中去。

当然，我们需要扫描所有被移动的对象，将任何遗留下来的可达对象都从这个车厢中清除出去。既然 Train 算法的收集过程在检查任何 Train 内部的引用之前，先要检查一个对象是否是从其他的 Train 被引用了，因此从其他的 Train 可直接指向的对象都要被移到这个 Train 外面。这点很重要，因为一个 Train 可能包含一个跨多空间循环的对象，它属于其他的 Train（也就是说，不能从当前的领导者可达），需要被移出。

既然算法周期性的在 MOS 中复制所有的成熟对象，因此它不需要额外的开销就可以达到重新聚集活对象的目的。在拷贝过程中，我们可以应用复杂的紧缩和聚集技术。另外，这个算法避免了在标记和清除收集器中会产生的碎片化的问题。

有关于这个算法的正确性问题，也就是它最终是否可以收集所有的垃圾的论证，有兴趣的读者可以参看论文【HM92】。

14.2.4 一个简单的例子

下面用一个例子简单的说明上述算法是怎么工作的，并结合图进行解释。这样读者就会对 Train 算法有较为感性的认识。为了简单性，我们将假设一个车厢中最多容纳 3 个对象。这意味着收集器的任何

一次启动将最多只移动 3 个对象。

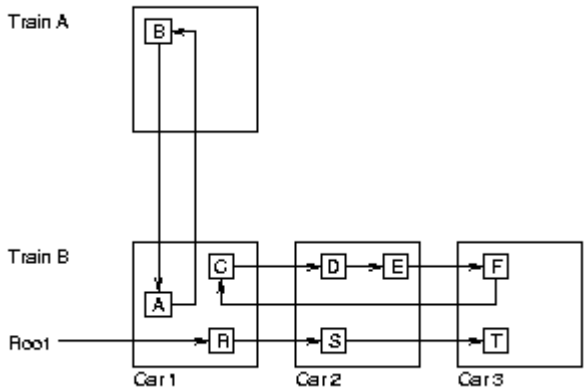


图 14.1 刚开始的分布情况

在图 14.1 中可以看出，这个例子中给出了 3 个数据结构。一个结构包含对象 R，S，和 T，它们从根集可达。另一个结构包含对象 A 和 B，是一个循环的垃圾结构，它跨过两个 Train。第三个结构，包括对象 C，D，E 和 F，形成了一个大型的循环垃圾结构，它太大了以至于不能存放在一个单独车厢中。我们将给出 C-D-E-F 是怎么样被孤立并且最后被释放的，以及 A-B 是怎么样被合并到一起来的和释放的。

首先从 Train B 开始。在这个 Train 中有没有任何对象是从外面的 Train 可达的？可以发现对象 A 和 R 都是可达的，所以我们集中精力于 car 1。Leader R 被撤出到另一个 Train 中去。到底是去哪个 Train 是一个策略的决定问题。这里我们选择 Train A 而不是创建一个新的 Train。下一步，follower B 从 Train A 可达，所以它被撤出到 Train A 中去。对象 C 是从 Train B 可达，因此 C 被移到 Train B 的最后一个车厢中去。Car 1 使用的空间现在就可以被回收了。图 14.2 给出了经过第一次收集之后的 Train 的状态。

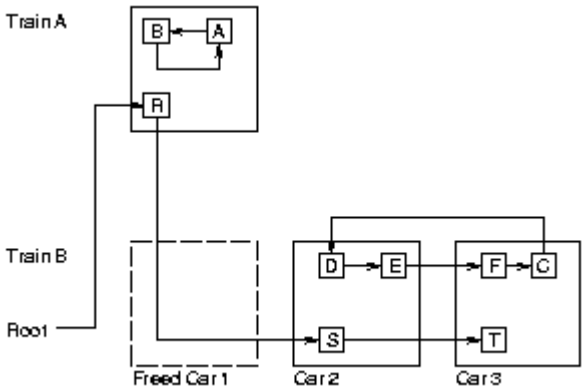


图 14.2 撤出 leader R，将结构 A-B 聚集成组，拷贝对象 C

收集器的第二次被调用将焦点集中在 car 2 上。这里没有对象是从根集被引用的，因此我们寻找 car 2 中被从 Train B 外面引用的对象。对象 S 被从 Train A 引用，所以我们将 S 移进 Train A。因为 Train A 中所有的车厢都是满的，所以我们需要加一个新的车厢来为对象 D 提供空间。经过对对象 D 的扫描，发现了对象 E 在 car 2 中。E 被撤出到 car 4。图 14.3 给出了这个时候的状态图。请注意活的 R-S-T 结构是怎么样与死去的 C-D-E-F 结构分隔开的。

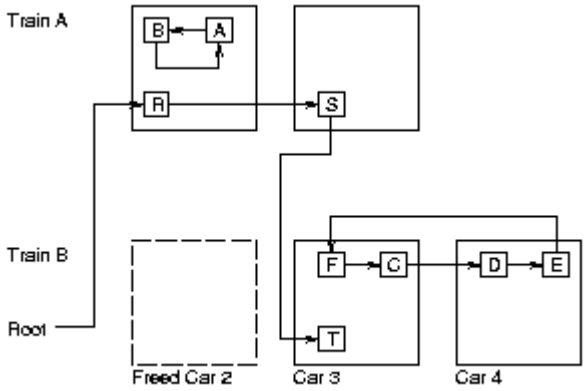


图 14.3 撤出 follower S，然后拷贝 D 和 E

在算法的下一次调用的时候，我们注意到 Train B 仍然被引用到，所以我们的焦点仍然在 Train B 上，这次对 car 3 进行收集。仍然没有对象是从根集直接可达的。Follower T 被从 Train A 引用，所以它被移到 Train A 中。对象 T 被扫描，但是没有包含指向 car 3 的引用。接着，我们考虑来自 Train B 内部的引用。对象 F 就这样被引用了，所以它被移动到 car 4。对对象 F 的扫描发现了一个对对象 C 的引用。因为 car 4 是满的，所以一个新的车厢被加入这个 Train 的尾巴，然后对象 C 被移到这个新增加的车厢中去。到了这个点上(见图 14.4)，结构 R-S-T 已经和结构 C-D-E-F 分开了。

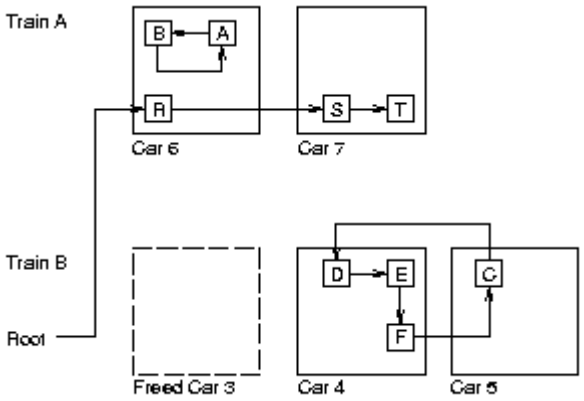


图 14.4 大型的循环垃圾结构被隔离到一个单独的 Train 中了

在收集器的下一次调用的时候，发现 Train B 已经没有引用指向它了，所以整个 Train 都可以被立刻回收。注意到我们将 C-D-E-F 结构移到同一个 Train 中，在这里它被回收，即使它比一个车厢要大。这样我们就只剩下 Train A 了。

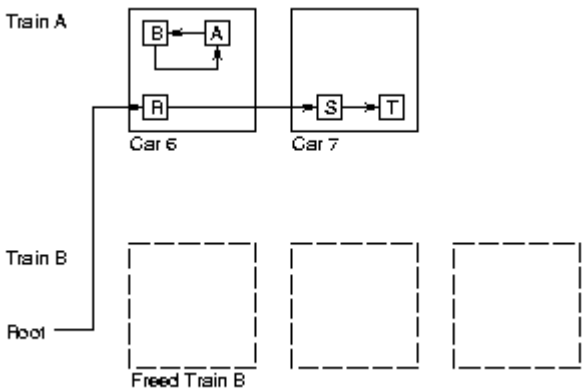


图 14.5 没有任何引用指向的 Train 将被释放

在图 14.5 中，注意到 Train A 有一个引用从 Train 外面指向它，所以我们将焦点集中到 car 6 上。因为对象 R 从根集直接可达，根据算法，它应该被移动到另一个 Train 中去。在这种情况下，需要创建一个新的 Train C，然后将对象 R 移进去。结构 A-B，它本来形成了一个循环的列表，跨越多个 Train，现在被聚集到一个车厢中，这个车厢中现在已经没有可达的对象了，所以它也被回收了。

图 14.6 没有给出被回收的 Train B，但是给出了新的 Train C，它包含对象 R。现在考虑 car 7。对象 S 被移进到 Train C 并且还被扫描，以得到指向 car 7 的引用。对象 T 被发现了，然后移进 Train C。car 7 现在可以被回收了。

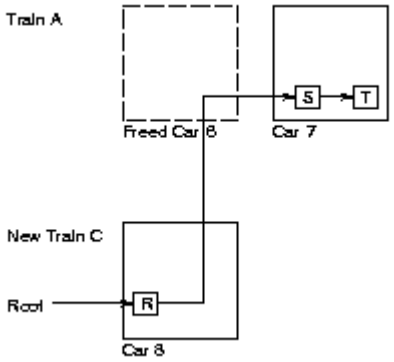


图 14.6 撤出对象 R，所以循环结构 A-B 现在就可以被回收了

在图 14.7 中，现在留下来的是一个 Train，它有三个被整理好的聚集的活对象。它们是在这个例子一开始就只有的三个可达对象。这个算法成功的将所有的不可达的对象归类到不可达的 Train 中，在那里，它们可以被非分裂性地释放。

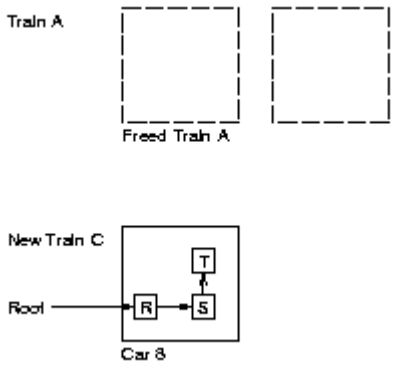


图 14.7 活的结构 R-S-T 被聚集到一个 Train 中来了

第15章 ORP 中 GC 的实现

ORP 中 GC 的实现采用了多种垃圾收集算法。我们可以通过改变预编译选项来获得不同的 GC 模块。现在我们就结合一种具体的 GC 模块，预编译选项是 GC_GEN_V3，ORP 版本号为 1.0.9 来讲述 GC 的实现，力图使大家对自动内存管理有个直观的认识。这一部分根据 ORP 的源代码和一些关于垃圾收集的论文编写而成，为了更深入理解 GC 的实现，请大家参考 ORP 中 GC 部分的源代码。

15.1 GC 的初始化过程

GC 的初始化过程是在整个虚拟机启动时由 ORP 调用的。GC 初始化过程包括堆空间的划分，用来管理这个堆空间所需的一些重要全局变量如空闲块表 free_blocks ，根集 root_set 等的初始化。在 ORP 中，我们将整个堆空间按代（gerneration）的观点分成几个逻辑区域，这些逻辑区域如 nursery，step，LOS 等都要被初始化，所要做的工作就是给这些逻辑区域分配一些空闲的内存块。另外为了访问别的模块的接口函数我们还要要创建其他模块的控制对象等。在这里我们主要讨论堆空间的初始化过程。初始化过程是 GC 实现其内存管理功能的基础。

15.1.1 GC 堆的组织

GC 在初始化时会向操作系统请求自己所要管理的堆空间，这个堆就是虚拟机自己的内存空间，在这个虚拟机上运行的应用程序所需的内存都是在这个堆上分配的。这个堆是一个连续的地址空间，起始地址必须满足操作系统的虚页边界条件。堆的大小可以通过命令行设定，缺省情况下堆的初始大小是 32M，并且可以扩展到 64M。基于减少内存碎片和内存分配时间的考虑，将堆分成 64K（2^{16}）大小的块（block），所以一开始有可用块 512 个，并最大可扩展到 1024 个块。堆的结构如图 15.1 所示。

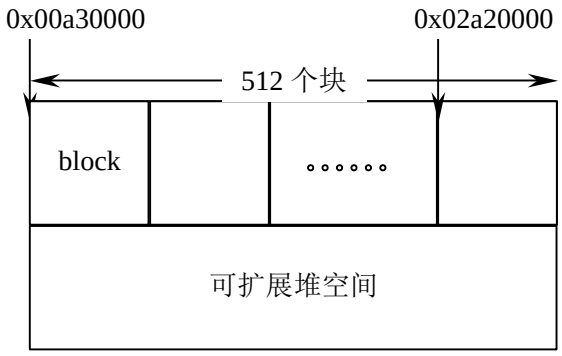


图 15.1 堆结构

我们假定堆的起始地址为 0x00a30000(为便于描述，后面都使用这个地址为堆的首地址),则最后一个块的首地址为 0x02a20000, 如果前半部分空间用完，我们可以使用后面的空间。

每个块分成页（page 或 card），页的大小 2^k 和操作系统虚拟内存页的大小是一样的，一般页的大小是 4096 字节，所以每个块有 16 个页。关于这个块的信息放在它的前 3 个页，所以一个块可供应用程序对象使用的页只有 13 个。对于一个对象占几个连续块的情况，除第一个块外其余块的可使用页都是 16 个。块的信息由 block_info 结构来描述，其中块通用信息占一个页，帮助实现 SAPPHIRE 垃圾收集算法的信息 tri_color_info（本章未涉及）占 2 个页。块的结构如图 15.2 所示。从图中我们可以看到，对象可以跨越两个页，所以页不是对象分配的单位。后面我们可以看到，页实际上是标记的单位。采用和操作系统一样大小的页，不会造成 ORP 中的一个页一部分在物理内存，一部分在磁盘上的情况。

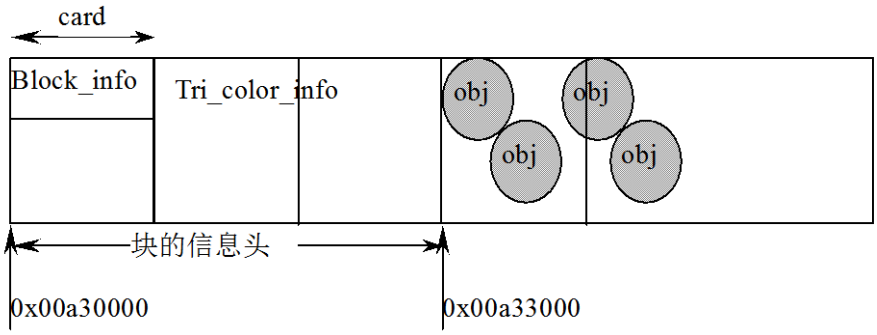


图 15.2 块结构

block_info 包含的域及其作用见图 15.3。 现在要完全理解各个域的作用还为时过早，在后面的章节我们在使用这些域时，会对它们的作用作详细的解释。

```
bool c_area_p;
// 为真表示这个块处于被收集区，处于被收集区的块在垃圾收集之后要么清除死对象，要么整
//个块又变成空闲块
bool in_nursery_p;//为真表示这个块属于 nursery
bool in_los_p;//为真表示这个块属于 LOS
bool in_step_p;//为真表示这个块属于 step
//一个 train 包含几个 car,而 car 是由几个块组成的链表

POINTER_SIZE_INT train_birthday;
//仅当这个块属于一个 train 时有效,表示所属 train 的生日
POINTER_SIZE_INT car_birthday;
//仅当这个块属于一个 train 时有效，表示所属 car 的生日；对于不属于 train 的块，如 nursery，
//step， LOS 块，这两项都是 0;这两项的作用主要就是在两块之间找我们感兴趣的引用关系，
//后有详细讨论

void *free; //指示从块中哪个地址开始是空闲的
void *scan;//指示当前 cheney 扫描已扫描到的位置
void *ceiling;//指示这个块的结束地址，没什么大的用处

block_info *next;//这个域指示和本块属性相同的下一个块

block_list_info *list_info; //如果这个块属于一个块链表，这个域指示此块所属块链表的信息

unsigned int block_status_table_index;
//每个块都在一个全局块状态表中有一个条目，这个域为对应条目的索引

unsigned int number_of_blocks;
//对于 LOS 块 和空闲块 ,这个块能代表其后和自己属于一个整体的连续块 ,这个连续块我们暂
//且称为块区;这个域就是所代表的块的数目，包括自己在内
```

```
nursery_state      nursery_status;
//如果这个块属于 nursery，这个域标记他的状态，状态包括 free_uncleared_nursery
// thread_clearing_nursery 等等。这个域只对 nursery 块有效（bogus_nursery 状态对 step 块有效）

bool in_free_p; //标记这个块是否为空闲块

block_info *all_blocks_next;
//下一个块区。这个块可能代表几个连续的块(也可能只有它自己)，用这个域可直接找到
//紧邻的（对物理地址而言）块区的代表

POINTER_SIZE_INT los_object_size;
//LOS 块被分成一定大小的段。块不同，段的大小也不一样，这个域就指示段的大小

byte card_table[16];//用来做页标记
Java_java_lang_Object *card_last_object_table[16];
//记录块中每个页的最后一个对象的首地址，此块中的每个页都有相应的条目
MARK mark_table[1024];
//用来标记 LOS 对象。因为 LOS 的对象必须满足 64 字节的边界，所以每 64 个字节对应一个条
//目。标记的具体含义在后面会讨论。
```

图 15.3 block_info 的成员含义

跟 block_status_table_index 相关的是一个叫 gc_block_status_table 的表。这个表记录了所有块的状态。有了 block_status_table_index，我们就能很快地访问块的状态。这些状态是 block_in_free，block_in_step，block_in_mos, block_in_nursery。首先所有的块都应该有状态 block_in_free，在划分一些块给 nursery 后，这些块的状态成了 block_in_nursery，随着 nursery 的状态成了 spent_nursery,这些处于 nursery 中的块被移入 step 中，所以这些块的状态变成 block_in_step。在回收阶段，step 中的活对象会被移到别的块中，这些块的状态就是 block_in_mos。step 或 train 中的块被回收后，状态回到了 block_in_free。块有四种大状态，每种状态还分几种子状态，比如 block_in_nusery 状态，它就有几种子状态，所以我们用专门的一个域 nursery_status 来记录。

再来说说块区。对象的大小是不同的，有些对象一个块就能容纳，有些对象要占几个块。为了对象分配的统一实现，我们采用块区的概念。块区是几个连续块，当然也可以是一个单块。块区的代表就是它们的第一个块。空闲块区可以分裂成多个小块区（注意：这多个小块区即使都是空闲的，它们也不能看作是一个块区），如果我们找不到要求大小的块区，我们可以对连续的空闲块区做合并。我们在请求一个块区时，比如大小为 2 的块区，总是要找大小和 2 最接近的空闲块区，以使剩下的最大空闲块区尽可能大。

15.1.2 空闲块表的初始化

在堆被分配好并被划分成块之后，我们需要一个表 free_blocks 来记录所有的空闲块区。表的第 i 个条目指向一个块区链，真正在这个链上的只是各个块区的代表块，代表的连续块数目在 number_of_blocks 域中，并且这个值必须大于等于 2^i,小于 2^{i+1}。可以看出，这实际上就是采用一个改动过的 buddy 算法来管理空闲块区，buddy 算法是用来解决外部碎片（external fragmentation）问题的，并在 linux 内核中用于空闲页帧块的管理。改动主要体现在块区合并，块区大小的要求不同上。请读者参考 buddy 算法来理解所作的改动。这里再提前说一下内部碎片（internal fragmentation）问题。ORP 的堆空间除了 LOS，对象都是顺序分配的，所以几乎不存在内部碎片问题，但 LOS 对象的分配不是顺序分配的，我们将块分成固定大小的段来处理内部碎片问题，在 15.1.5 节中我们会详细说明。

用这种 buddy 变种算法当我们请求一个块区时，我们能找到最满足要求的一个空闲块区。所谓最满

足条件，是指能使堆中最大空闲块区最大，不至于在堆中存在大量空闲小块区的情况下，却不能满足一个大块区的请求。空闲块表的一个状态如图 15.4 所示。当我们请求一个块区时，根据块区大小决定从哪个条目开始搜索。假设我们需要两个空闲连续块，则会从第 1 个条目开始搜索，最后分裂第 8 个条目的块区为两个块区，大小分别为 2，255。大小为 255 的块区被连接到第 7 个条目。

这里要注意的是当我们请求几个块区，然后立即释放，free_blocks 也不会恢复到请求之前的状态。比如在图 15.4 中，我们请求大小为 2 空闲块区，free_blocks 的第 8 个条目的 257 个块发生分裂。当前一个块区被回收时，它只不过被连到 free_blocks 的第 1 个条目上。同样图 15.4 中第 0 个条目的两个块区，即使它们是紧邻的，也不会被合并兵加入到第 1 个条目的链表中。只有当找不到足够大的空闲块区时，我们才会做合并。合并的过程简单示意如图 15.5 所示。合并的伪代码会在 15.3 节给出。

GC 初始化空闲块表时，所有的块都被连到表的第 9 个条目上，这 512 个块形成的块区的代表就是堆的第一个块，初始状态如图 15.5 的下半部分所示。根据这个表，我们可以很容易地管理空闲块区，包括将回收后的块区插入空闲块表，和将被分配的块区移出块表，并调整块表使之满足此表的约束条件。

连接空闲块区到空闲块表的伪代码如图 15.6 所示。

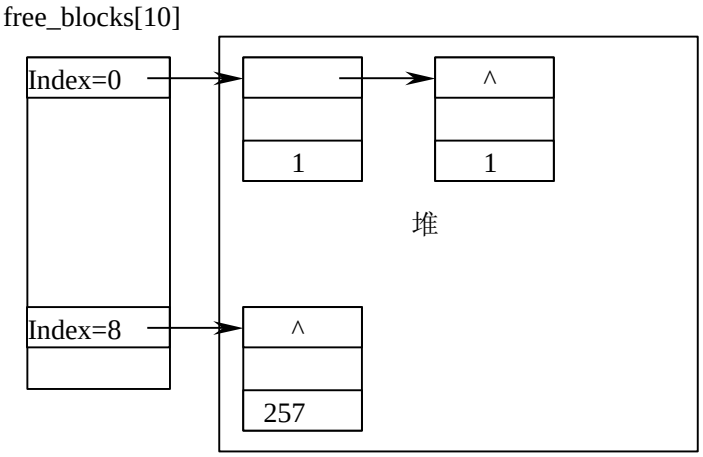


图 15.4 空闲块表示例

从空闲块表中取一个指定大小的块区的代码 15.1.3 中描述。这是一个通用过程，之所以在 15.1.3 中描述，是因为这个过程最先在这里用到。

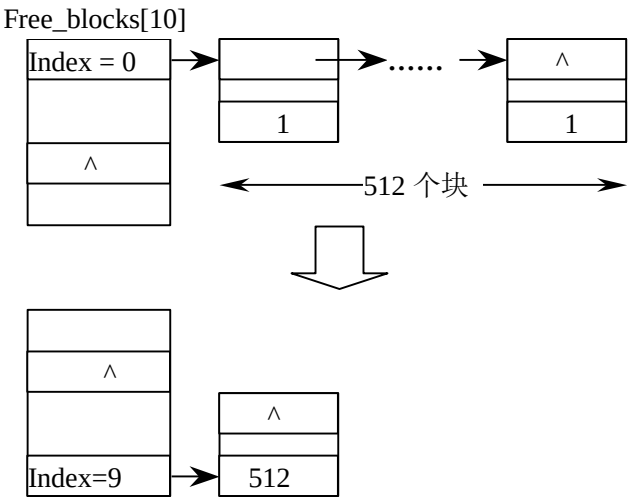


图 15.5 空闲块区的合并

```
void link_free_blocks(block_info *freed_block, int blocks) {
    //freed_block 为要插入的空闲块区的代表，blocks 为这个块所代表的块区中块的数目
    freed_block->number_of_blocks = blocks;
    for 每个 freed_block 代表的块
        gc_block_status_table[i] = block_in_free;
    根据 blocks 的大小和表的约束条件在 free_blocks 找到合适的插入条目 properindex
    link_blocks_ts(&free_blocks[properindex],freed_block);
    /*将此块线程安全地插入空闲表相应条目所指示的块链表中，注意是插在表头*/
    return;
    /*表面上这个函数只插入一个空闲块，实际上这个空闲块所代表的空闲块区都可被 GC 当作空
    闲块来管理了*/
}
```

图 15.6 将空闲块区插入到空闲块表的过程

15.1.3 nursery 的初始化

当应用程序使用 new 来请求内存空间或 ORP 向 GC 请求内存时，GC 一般会在 nursery 中分配空间给它们使用。所以一开始我们就要给 nursery 几个空闲块，预设块的数目是 16 个。这个数目和线程数有关，当线程增多时，相应要增加 nursery 中块的数目。nursery 初始化的过程是获得需要数目的空闲块，修改他们的 block_info，然后连接到 nurseries[]指针数组中。伪代码如图 15.7 所示。

```
block_info *nurseries[MAX_NURSERIES];// MAX_NURSERIES = 4096
void add_nurseries(int howmany){
    for i= allocatednursery to alloctednursery+howmany do
        nurseries[i]    = get_new_nursery();
    //allocatednursery 为已分配的 nursery 块个数，在 GC 初始化时值为 0
    //add_nurseries 不仅在初始化时调用，还在需要增加 nursery 块数时调用；应用程序的线程增多
    //时，16 个块已经不够，所以需要增多 nursery 块
    free_nursery_hint = allocatednursery;//暗示 nursery 中哪儿有空闲块，这是提高效率一个技巧
    allocatednursery+= howmany;
}
block_info *get_new_nursery(){
    block_info *this_nursery = get_new_block(size_in_block);//这里 size_in_block 等于 1
    //size_in_block=1 表明只需要一个空闲块，这一个块同样是一个块区
```

```
    gc_block_status_table[this_nursery->block_status_table_index] = block_in_nursery;
    this_nursery->in_nursery_p = true;
    this_nursery->in_free_p = false;
    this_nursery->nursery_status = free_uncleared_nursery;

    //初始化时 nursery 块的状态是 free_uncleared_nursery，因为块未清零

    //nursery 块的其他状态的意义留待后面描述

    return this_nursery;
}
```

图 15.7 增加 nursery 块的过程

获得空闲块区的过程包括两个部分，一是查 free_blocks 表(上节内容)获得空闲块区，二是维护块区分裂后剩下的的空闲块区，修改 free_blocks 表。获得空闲块区的代码如图 15.8 所示。如果我们需要一个大小大于 1 的块区，那它一定是在 LOS 分配。除了 LOS 和空闲空间之外，所有的块区大小都为 1，所以有时我们就直接称块区为块了。

```
block_info *get_new_block(size_in_block){

//关于 free_blocks，参见 15.1.2

    bucketindex = 0;

    while (bucket_index < MAX_BUCKET){//扫描整个 free_blocks 表

        if (size_in_blocks <= bucket_block_size){

            //要找连续块数目最小且比请求的大小要大的条目

            result = p_get_head_ts (&free_blocks[bucket_index]);

            //线程安全地取得相应条目的头指针，即块区代表块的地址，并将这个块从链中删除

            if (result){

                //如果找到最合适的块区，首先初始化这个空闲块区

                init_block (result, size_in_blocks);
                extra_blocks = result->number_of_blocks - size_in_blocks;

                //这个块区分裂后还剩下一个大小为 extra_blocks 个连续空闲块的块区

                //分裂后新的空闲块区不能丢掉，要连到 free_blocks 中

                if(extra_blocks){
                    freed_block = (block_info *)((POINTER_SIZE_INT)result
                                    + (size_in_blocks * GC_BLOCK_SIZE_BYTES));

                    //找到剩余空闲块区的代表

                    link_free_blocks (freed_block, extra_blocks);

                    //剩余空闲块区连到空闲块表上去，见上节

                    freed_block->all_blocks_next = result->all_blocks_next;
```

```
                    result->all_blocks_next = freed_block;
                    //原来的大块区变成了两个小块区，相关信息要维护
                    //这就是上两条语句的作用。从这里我们可以看出 all_blocks_next 域的作用
                     //它指向紧邻的块区，这个域在我们做块区合并时非常有用。
                }
                return result;
            }
        }
        bucket_index++;
        bucket_block_size = 2 * bucket_block_size; //找下一个条目
    }//end of while
    //如果允许通过扩展来获得新空闲块,我们就扩展堆，
    //否则对整个堆做回收
    extend heap or reclaim heap；//当然在 nursery 初始化时肯定不会需要扩展堆或触发垃圾收集
    //垃圾收集过程后面有详细介绍，也是本章的重点
    return get_new_block(size_in_block);//扩展或回收后再尝试获取连续空闲块
}
```

图 15.8 获取一个空闲块区的过程

将 16 个块分配给 nursery 之后，nurseries 和空闲块表的状态如图 15.9 所示。

15.1.4 step 的初始化

step 和 nursery 一样都是代。在前两章中，我们称 step 为梯级，它是代的细化，这里 step 就是一个代。nursery 属于最年轻的一代，step 属于比 nursery 年老的一代。但是从它们的块的 train_birthday 等于 0 来看，它们又是同等并最年老的一代。注意后面我们谈到年轻年老都是以 train_birthday 和 car_birthday 来判断的，这个并不影响我们先前关于年轻年老的讨论，因为我们同样可以认为年轻的更有生命力。nursery 中的对象死亡率最高，已用完的 nursery 块会被移入 step。GC 初始化时同样要对 step 初始化。我们用一个 step_info 的对象来描述 step。step_info 结构就是 block_list_info 结构。step 实际上就是一个块表。我们来讨论一下 block_list_info 的构成，这个结构不仅用来描述 step，也用来描述很多其他逻辑结构，如 car，所以很重要。图 15.10 描述了它的结构。

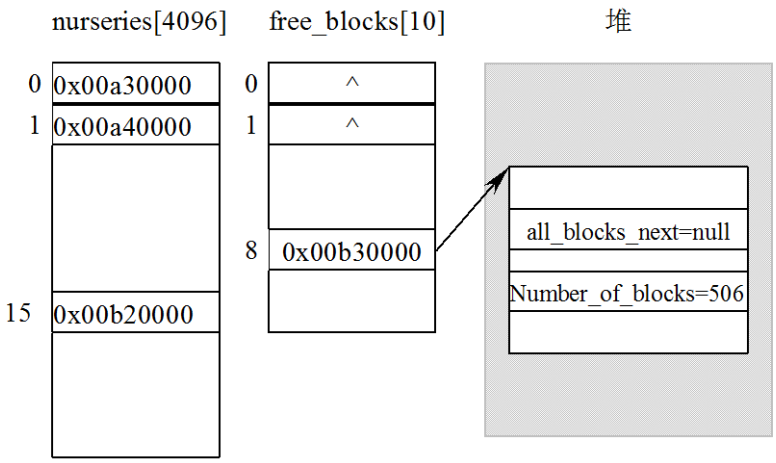


图 15.9 nurseries 和 free_blocks 数组的状态

```
typedef block_list_info step_info;
typedef block_list_info car_info;
struct block_list_info{
    block_info *alloc_block;
    //step 中的块分为一开始分配用来放存活对象的块和后来从 nursery 转移过来的块，这个指针用
    //来指向一开始分配的块。因为转移过来的块始终加在表头，所以这个域始终指向表尾的块
    block_info *blocks;
    //指向块链表的头
    block_info *scan_block;
    //指向表中扫描到的块，当 scan_block 等于 alloc_block，并且这个块 block_info 中 free 指针等于
    //scan 指针时，说明表中所有块都被扫描过了
    train_info *my_train;
    //car 也是一个块链表。所以当它是个 car 时，这个域用来指向它所属的 train
    //关于 train 和 car 后面还有描述
    POINTER_SIZE_INT birthday;//一个说明 car 生日的 32 位整数，第一个 car 的生日为 0
    block_list_info *next;//下一个 step 或 car，当前的实现只有一个 step 和一个 car
}
```

图 15.10 描述 step 的结构

说了 step 的数据结构之后，我们来说说如何来给 step 分配块。step 初始化只需要一个块，即 alloc_block。和 nursery 的初始化一样，获得一个空闲块，修改他的状态，最后修改 step_info 中的各个域。图 15.11 是初始化过程的伪代码。

```
step_info *step;
```

```
step = (block_list_info *)malloc (sizeof(block_list_info));
memset ((void *)step, 0, (sizeof (block_list_info)));
block_info *tmp = get_new_block(1);

//获得一个空闲块，由图 15.8 和找空闲块的算法来看这个空闲块应该是 0x00b30000


这个块在 gc_block_status_table 中对应条目改为状态 block_in_step；


tmp->in_step_p = true;
tmp->in_free_p = false;
tmp->list_info = step;

//上两条语句修改获得的块的 block_info，表明这个块已被 step 使用


step->alloc_block = tmp;
step->blocks = tmp;
step->scan_block = tmp;

//初始化时 step 中只有唯一一个块，既是表头也是表尾，扫描的指针当然也在表头
```

图 15.11 step 的初始化过程

15.1.5 LOS 的初始化

LOS(Large Object Space)从字面上看是存放大对象的地方，实际上不仅存放大对象，也存放那些固定的不移动的小对象，如一些标准类对象等等。GC 初始化时，我们要为 LOS 分配空闲块。分配空闲块的过程和前面 nursery，step 分配的过程一样，唯一的差别只在分配后，块的状态不同。所以这里就不描述具体过程了。这一节我们主要讲述 LOS 空间的管理。

我们用一个数组 los_buckets 来记录每个 LOS 块。这个数组的元素个数是 GC_LOS_BUCKETS(现在定义的大小是 10 个块)，类型为 block_info*。LOS 块被分成固定大小的内存段，段的大小对不同的块是不同的。对于 los_buckets 的第 i 个条目指向的块链（初始化时这个链表只有一个节点），所有块的段大小是 $64*2^i$。这样做的好处是解决内部碎片问题，并且容易实现对 LOS 对象的标记。空闲的段用链表连接起来。什么我们要用链表来访问下一个段，而不是地址直接加 $64*2^i$ 呢？因为 LOS 对象也可能会死亡，并且 LOS 活对象不能被移动，所以我们就无法知道对象死亡后留下的空闲段到底可不可用，使用链表来访问所有空闲段就可以解决这个问题。当段被释放时，它被加到链表中，当段被使用时，被移出链表。从这里可以看出 LOS 块的对象分配是通过空闲段链表完成的，而不是通过移动块的 free 指针。图 15.12 是空闲段链表的初始化。

```
struct los_free_link {
    los_free_link *next;
};
los_ini()
{

    为 los_bucket 的 10 个条目分配块，并更改相应的块状态;

    object_size = 64;
    for i = 0 to 9 do
```

```
{//将所有 10 个块的内存分成相应大小的段，并将段链接成链表
    object_size=object_size<<i;
    tmp = los_bucket[i];
    tmp->free = GC_BLOCK_ALLOC_START(tmp);
    //除去用作块信息的 3 个页(card)，剩下的才是真正可分配的内存

    //free 域实际成了空闲段链表的头
    tmp = (los_free_link *)los_block->free;
    //直接利用空闲段来存放链接信息
    next_object = (los_free_link *)
        ((POINTER_SIZE_INT)tmp + (POINTER_SIZE_INT)object_size);
    while 块中仍有容纳下一个对象的空间
    {
        tmp->next = (los_free_link *)next_object;
        tmp = tmp->next;
        next_object+=object_size;
    }
  }
}
```

图 15.12 所有 LOS 块空闲段链表的初始化

图 15.13 说明了 LOS 的组织。

下面我们来看看在 LOS 中对象分配的循环最先适合算法（Circular First Fit）。首先找到最接近需要大小的 los_bucket 条目，然后取空闲段链表的表头指针作为返回结果，并更新表头指针。伪代码如图 15.14 所示。

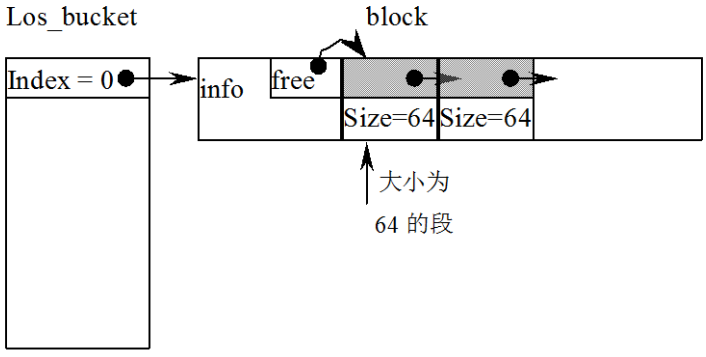


图 15.13 LOS 的组织

```
block_info single_object_blocks;
Java_java_lang_Object *gc_los_malloc（size）{
    for (i=64; i < GC_BLOCK_ALLOC_SIZE; i = i*2) {
```

```
            if (size < i) {
                break;
            }
            bucket_index++;
        }//找到正确的条目，结果在 bucket_index 中
        block_info *right_block = los_bucket [bucket_index];
        Java_java_lang_Object *result = null;
        while (result ==null){
            result = (Java_java_lang_Object *)right_block->free;
            if(result)
                线程安全地改变 right_block->free 到下一个段；
            else {
                如果 right_block->next 不为空，则 right_block = right_block->next，继续 while 循环
                否则添加一个新的 LOS 块，这个块的段大小和 right_block 一样;
                /*要获得一个新的块可能会触发垃圾收集；如果请求的字节大于现有的所有段大小，
                则分配在 LOS 中一个特殊区域*/
                newblk->next =right_block;
                right_block=newblk;
                los_bucket [bucket_index] = right_block;
              }
        }
        (los_free_link *)result->next = null;//清除 next 域，因为这个段已不是空闲段了
        return result;
    }
```

图 15.14 在 LOS 中分配内存的过程

los_bucket 指明的块中，段的大小没有超出半个块的范围。当我们遇到一个大小超出一个块的对象时，在 los_bucket 中也找不到一个合适的段。我们要请求一个块区来存放这样的对象，这一类的块区同样属于 LOS。我们用变量 single_object_blocks 来访问所有这样的块区。所以 LOS 中的所有块都可以通过 los_bucket 和 single_object_blocks 来访问。

15.1.6 根集 root_set 的初始化

根集 root_set 包含了当前运行栈，寄存器中的所有对象引用。这些对象将来可能还会被用到，所以在垃圾收集时必须保留下来。GC 维护一个 root_set，以便在做垃圾收集时判断哪些对象是必须被清除出堆的，哪些对象是必须保留下来的。注意，GC 不能知道运行栈，寄存器的状态，为了获得根集，GC 要使用 JIT 提供的服务。这一节主要讲述的是 root_set 的数据结构 Root_List，以及它的初始化。

类 Root_List 包括如下几个域：

```
unsigned _size_in_entries;
unsigned _current_pointer;
```

```
unsigned _resident_count;
Java_java_lang_Object ***_store;
```

_size_in_entries 指示表最大有多少个条目，缺省为 1024 个；_resident_count 为实际存在的条目个数；_current_pointer 指示当前扫描到哪个条目；_store 为一个以 NULL 结尾的数组，所以实际我们在访问_store 时，用不到_current_pointer。图 15.15 为 Root_List 初始化的伪代码。

```
Root_List::Root_List(){
    _size_in_entries = DEFAULT_OBJECT_SIZE_IN_ENTRIES;
    _store = (Java_java_lang_Object ***)malloc(_size_in_entries *
                                        sizeof(Java_java_lang_Object **));
    _current_pointer = 0;
    for (i = 0; i < _size_in_entries; i++) {
        _store[i] = NULL;
    }
    _resident_count = 0;
}
```

图 15.15 Root_List 的初始化

下面将说明_store 数组的具体含义。_store 数组的各个元素类型是 Java_java_lang_Object**，每个元素中存放了一个 slot 的地址，所以通过_store 我们可以访问到所有从根集可达的堆对象，这样的对象我们称之为活对象，反之，就是死对象。图 15.16 说明了_store 数组的含义。

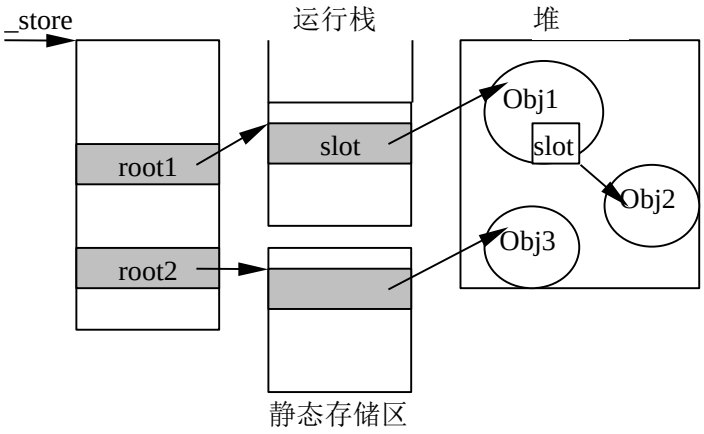


图 15.16 _store 数组示意图

15.2 GC 和其他模块的交互界面

为了和 ORP 的其他模块协同工作，GC 需要提供一些接口函数给别的模块使用，同时为了完成它自身的垃圾收集任务，它也需要其他模块给予的支持。需要改进模块时，只要保持接口不变，各个模块间仍能很好地协同工作。这一节讲述的是 GC 主要接口函数的规格说明（不是全部），不涉及内部实现细节。

15.2.1 GC 提供的接口函数

- **gc_next_command_line_argument**

这个接口的作用是根据 orp 的命令行参数设置垃圾收集模块的一些选项。

- **gc_init**

这个函数是 GC 提供给 ORP 用来在 ORP 启动时初始化 GC 自身的。这个函数功能主要有如下几点：

设置一些性能参数，检查编译标志，初始化内部指针表，创建一个 ORP_Control 的对象（通过这个对象 GC 可以使用 ORP 提供的服务），然后初始化为完成内存管理任务而定义的数据结构，见 15.1 节。这个接口函数必须在除 gc_next_command_line_argument 之外的所有接口函数调用之前被调用。

• gc_orp_initialized

ORP 使用这个函数来通知 GC，ORP 已完全初始化好了，这样 GC 的完全停止事件才能在任何时间发生，因为 ORP 已经准备好随时枚举活引用。在这个函数调用之前，因为 ORP 未完成初始化，GC 可能会遇到一些奇怪的情况，比如空的或不完全的 vtable。GC 必须能面对这种情况，和 ORP 一起保证虚拟机启动的进行。这个接口调用两次会造成未定义的后果。

• gc_wrapup

一旦 ORP 不再需要堆空间和管理堆空间所需的数据结构，这个接口就被调用。ORP 使用这个接口来让 GC 运行破坏器，释放从操作系统得到的内存空间。它被调用后，ORP 就不能依赖任何 GC 创建的数据结构了。如果之前调用了 gc_enumerate_finalizable_objects，并且调用这个接口又发现了一个其终结器未运行的对象，这种情况下会报错。注意：ORP 对终结器的实现有问题。

• gc_enumerate_finalizable_objects

这是一个 ORP 在退出前调用的函数。GC 假定此时所有的对象都是死的，并且枚举所有那些重载了 finalize()的对象。当前版本 ORP 的 GC 并没有实现这个接口。

• gc_class_ref_map

这个接口提供给 Class Loader 使用。

输入：一个指针 p_class，指向一个类数据结构（不是类的实例对象）的基址；一个整型数组，保存这个类中所有指针成员的偏移量。这个整型数组以 0 结束。

根据偏移量数组修改 p_class 的 gc_information 域，这样我们就能很容易直接从类数据结构知道哪些成员是指针。有了这个指针偏移量信息，我们就能从一个活对象找到另一个与它有引用关系的后代对象，这个对象一定也是活对象。

• gc_get_class_ref_map

输入：指向类的 vtable 的指针；存放成员指针偏移量的数组及其长度。

这个接口也提供给 Class Loader 使用。它的作用是把类成员指针偏移量数组复制到 vtable 的 gc_information 指示的数组。

• gc_requires_barriers

这个接口提供给 ORP（JIT）使用。返回设定的栅栏的类型。现有四种 GC 栅栏: GC_NO_BARRIER, GC_CARD_MARK_WRITE_BARRIER, GC_SAPPHIRE_WRITE_BARRIER, GC_TRACE_BARRIER 。ORP 使用它来获知 GC 是否需要读写栅栏。如果 GC 需要写栅栏，而 JIT 不提供写栅栏的支持，结果是未定义的。栅栏的类型存放在全局变量 which_barrier 中，这个全局变量在 ORP 运行过程中不会改变。

• gc_write_barrier

这是一个内联函数调用。提供给 JIT 使用。

输入：带有 slot 的对象的指针。

在 GC_GEN_V3 的 GC 模块中，他的作用是将此对象所在块 block_info 中的 card_table 相应的表项标记成 true，以表示这个页中有一个对象含有其他对象的引用。后面会说到只有页中有有趣 slot 才会做页标记，但实际上有点保守，因为有些有 slot 但却不是有趣 slot 的页也被标记。

• gc_heap_wrote_object

ORP（JIT）使用。除了 SAPPHIRE 算法，它和 gc_write_barrier 接口的功能是一样的。

• gc_heap_slot_write_ref

输入：带有 slot 的对象基地址；slot 在这个对象中的绝对地址（不是偏移量）；被引用对象的地址。

ORP（JIT）使用。作用就是将被引用对象的地址写入到 slot 中去。

• gc_heap_write_ref

除了第二个输入参数改为偏移量外，功能同 gc_heap_slot_write_ref。

• **gc_heap_write_int8**

输入：带有 slot 的对象基地址；slot 在对象中的偏移量；要写入这个 slot 的 8 比特整数。

ORP（JIT）使用。作用是将 8 比特整数写入对象中指定的 slot。

•gc_heap_write_int16；gc_heap_slot_write_int16；gc_heap_write_int32；gc_heap_slot_write_int32；gc_heap_write_float；gc_heap_slot_write_float；gc_heap_write_double；gc_heap_slot_write_double 等

所有这些接口都类似于 gc_heap_write_int8。我们为什么要提供这么多写堆中 slot 的接口？实际上，JIT 完全可以直接对 slot 进行写，提供这些接口的目的是 SAPPHIRE 算法不容许直接写 slot。为了使各个算法的接口统一，GC 专门提供了写 slot 的接口。

• **gc_heap_read_XXX; gc_heap_slot_read_XXX;**

输入：含有 slot 的对象指针；slot 的绝对地址或偏移量。

XXX 代表各种不同长度的数据类型，这一组接口当前仅仅将 slot 内容读出返回。

• **gc_add_root_set_entry**

输入：一个根引用，它指向运行栈或静态存储区中一个 slot，参见图 15.16。

此接口提供给 ORP 调用。GC 请求根集信息时，ORP 根据 JIT 的帮助获得一个个根引用，每获得一个根引用，它就会回调这个接口函数，将根引用填入 Root_List 对象的_store 数组。多次调用能得到完全的根集。关于 Root_List 参见 15.1.6。

• **gc_pinned_malloc_noclass**

输入：对象大小

输出：分配的内存空间的首地址。

提供给 ORP 使用。被分配的对象是固定的，不可终结的，并且不是数组。使用这个接口函数分配的空间属于 LOS。如何在 LOS 中分配对象，参见图 15.14。在 java.lang.Class 之前装载的类对象的内存分配都要使用这个接口函数，因为 java.lang.Class 的 vtable 还没有成功建立。当前需要用这个接口函数完成分配的仅有三个类对象 java.lang.Object，java.io.Serializable 和 java.lang.Class。

• **gc_malloc_or_null**

输入：对象大小 size，对象所属类的 vtable 地址。

输出：分配的内存空间的首地址。

这是一个用来快速分配内存的基本例程，GC 提供给 ORP 使用。它将对象所属类的 vtable 地址写入对象中，保证整个对象都被清零，然后返回这个对象的指针。这个例程特点是如果堆中没有可分配空间，允许返回 NULL，继而调用带垃圾收集功能的慢速分配例程。

• **gc_malloc**

输入：对象大小 size，对象所属类的 vtable 地址。

输出：对象在堆中分配的地址。

如果对象不能被快速分配，ORP 或 JIT 就要调用这个接口函数来为对象分配空间。

对象有可快速分配对象和受限对象之分。对象所受限制有，是否有终结性，是否满足边界对齐，是否为了性能考虑固定对象等等。为了区别不同的对象，我们利用 size 的高位。如果高位为 0，对象分配没有限制，如果大于 0，这个例程就会查询类数据结构决定这个对象的分配有何种限制。

类的具体限制可以通过 vtable 的 class_properties 域获得。我们可以很容易地通过 size 获得对象的真实大小。然后这个接口函数调用 gc_malloc_slow 慢速的获得一块内存，所谓慢速，就是可能不得不做垃圾收集等影响性能的工作。注意：因为 gc_malloc 不能返回 NULL，为了作到这一点，它可能不得不做垃圾收集，调用 gc_malloc 的位置必须在 gc 安全点，即 GC 假定 ORP 已经准备好支持一次垃圾收集。

内存分配序列一般是先调用 gc_malloc_or_null，然后检查返回是否为 NULL，如果为 NULL，则执行一些很少能执行到的代码来到达 gc 安全点，到达 gc 安全点就能调用 gc_malloc 了。

• **gc_thread_init**

现有版本的实现仅对 SAPPHIRE 算法有效。它为启动时的线程设置一个初始 nursery，实际未实现。

• **gc_release_nursery**

输入：一个 nursery 块的地址

如果一个线程有一个不再使用的 nursery 块，就调用这个接口。这个 nursery 对象状态必须是 active_nursery，这个接口函数将它的状态改为 spent_nursery。

• **gc_force_gc**

提供给 ORP 使用，强制执行垃圾收集。通常对应一个 java.lang.Runtime.gc 调用。

• **gc_free_memory**

获得空闲内存信息。对应于 java.lang.Runtime.freeMemory。

• **gc_get_new_nursery**

提供给 ORP 使用。获得一块状态为 active_nursery 的块，此函数返回块的首地址。这个接口通常在 gc_thread_init 被调用后的线程初始化过程中被调用，之后 orp_get_nursery()就有一个 nursery 块可以返回了。

• **gc_class_loaded**

输入：一个 vtable 的地址。

将这个 vtable 加入到已装载的 vtable 目录中。类装载器使用这个接口来通知 GC 另外一个类已经被装载了。

• **gc_is_object_pinned**

输入：一个 java 对象的地址。

如果这个对象在 LOS 中分配，则此接口返回 true，否则为 false。

15.2.2 GC 需要使用的其他模块的接口函数

• **orp_enumerate_root_set_all_threads**

当几乎没有可用空间分配给对象时，GC 只有准备做垃圾收集。在做垃圾收集之前，GC 必须知道哪些对象是活的，即以后还可能用到，哪些对象是死的，即以后不再使用，可以被清除掉了。这些信息 GC 自己是不可能知道的，GC 只有求助于 JIT，来获得这些信息。JIT 首先停止所有线程，对每一个线程，如果它的当前指令处于 GC 安全点，则为此线程枚举根集，如果不处于 GC 安全点，则此线程继续运行到 GC 安全点。JIT 如何为每个线程枚举根集参见 JIT 部分。枚举出来的根集被填入 Root_List 对象的_store 数组，这个数组以 NULL 元素结尾。

• **orp_enable_gc**

仅有当前线程（p_TLS_orpthread 指示了当前线程的句柄）能启用（enable）和禁止（disable）GC，GC 线程能将 GC 的启用状态改变成启用将阻塞（enabled_will_block）。不能在 JIT 产生的代码中启用 GC，仅在禁止状态时才能启用。通过 GC 绑架（hijack）代码可从禁止状态自动转移到启用将阻塞状态。还有其他情况能启用 GC。

• **orp_disable_gc**

如果当前线程的 GC 状态是启用的，则改为禁止的，并返回真。如果原来就是禁止的，则返回假。如果是启用将阻塞的，则等待当前线程的一个事件发生，然后继续判断当前线程的当前状态。它的返回值没有意义。GC 仅在机器码（native code，不是用 JIT 产生的机器码）中才能被禁止。在 JIT 产生的机器码中，GC 总是处于启用状态，这并不意味着 GC 能在 JIT 产生的代码的任一点发生，它只能在 GC 安全点发生。

• **orp_change_nursery**

输入：nursery 块的地址 p_nursery

改变 p_TLS_orpthread->p_nursery 到 p_nursery。如果 p_nursery 为空的话，意味着从线程结构中移去 nursery。

• **thread_gc_number_of_threads**

ORP 提供给 GC 使用。返回活动线程的数目，不包括当前线程。

• **orp_resume_threads_after**

在垃圾收集完成之后通知每一个 jit 做 gc_end，并且启动所有线程（除了那些调用了 suspend 的线程）；其他作用参见 ORP 的相关部分。

15.3 GC 的主要功能及其实现

15.3.1 类对象的内存分配

ORP 在启动和运行中需要装载一些类对象到堆中，它要向 GC 申请存放这个类对象的空间。首先我们来看看 ORP 是如何获得一块存放 java/lang/Object 类对象的空间的。ORP 根据类属性决定了它是一个固定对象。ORP 调用 GC 提供的固定对象分配接口函数在堆中获得一块内存，这个接口函数是 gc_pinned_malloc_noclass，当前只有 java.lang.Object， java.io.Serializable， java.lang.Class 三个类对象要通过这个调用来分配内存。GC 将固定对象分配在 LOS 中。关于 LOS 的组织参见 15.1.5。Object 这个类对象在堆中占用 220 个字节，根据图 15.14 的算法，在 LOS 中获得空间。

对于除上文所说的三个类对象以外其他类对象的内存分配通过调用 gc_malloc 接口函数完成，gc_malloc 和 gc_pinned_malloc_noclass 的区别在于，调用 gc_malloc 时，类对象的 vtable 已经构造完成，参数中对象大小并不是对象实际在内存中占用的字节数，而是结合类限制和对象实际大小的一个值。在做 gc_malloc 前，ORP 已经为可能发生的垃圾收集做好了准备。gc_malloc 调用一个助手函数 gc_malloc_slow。这个助手函数的参数是对象实际大小，和对象的 vtable 的地址。gc_malloc_slow 根据 vtable 中的 class_properties 来决定，是无约束的分配，还是在 LOS 中分配。现在要分配的是类对象，所以在 LOS 中分配。获得一块清除过的空间后，将新分配对象的 vt 域用 vtable 来初始化。

超出一个块可分配内存的一半大小的类对象分配在 LOS 中。首先我们根据对象大小得到一个空闲块区，并修改这些块的信息域，图 15.17 的伪代码说明了这个过程。

```
Java_java_lang_Object *gc_pinned_malloc (unsigned size, VTable *p_vtable,){
    /*size > 一个块可分配内存的一半*/
    block_info *block   = get_multi_block(size,….);
    //根据 size 可得连续的多个空闲块，block 是这些块的代表,这有可能触发垃圾收集

    将这些空闲块对应的全局块状态表条目改为 block_in_los

    /*所有的单对象块我们用一个链表连起来，头为 single_object_blocks*/

    线程安全的改变 block->next，和 single_object_blocks
    block->los_object_size = size;
    result = GC_BLOCK_ALLOC_START(block);//result 是块的可分配空间的首地址

    将分配的空间清零；
    result->vt = (VTable *)p_vtable;
    return result;
}
```

图 15.17 单对象块的分配过程

15.3.2 Java 对象的内存分配

当对象 vtable 指明的类属性为 0 时，我们把它看做一个 Java 对象。Java 对象有着不同于类对象的分

配方法。当 Java 对象大小超出大对象大小的阀值时，我们将 Java 对象分配在 LOS 中，否则分配在 Nursery 中。LOS 中对象的分配类似上一小节类对象的分配。而在 Nursery 中的分配分为快速分配和慢速分配两种。

·快速分配：

快速分配对 JIT 施加最少的限制，不需要要求每个分配点都是 gc 安全的，因为快速分配可以返回 NULL，而无须做垃圾收集。如果快速分配不成功，则尝试慢速分配。gc_malloc_or_null 是快速分配的基本例程。图 15.18 说明了这个例程的实现细节。

```
Java_java_lang_Object *gc_malloc_or_null(unsigned size, VTable *p_vtable)
{
    if (size > los_threshold_bytes) {
        return NULL;

        //如果对象大小不适合在 nursery 中分配，则返回 NULL，由调用者自行处理

        //同样如果 size 的次高位被置成 1，说明这个对象应该使用慢速分配

    }
    block_info *my_nursery = (block_info *)orp_get_nursery();

    //由 ORP 提示一个此线程属于 nursery 的块

    p_obj_start = my_nursery->free;//要分配的对象空间当然要从这个块的空闲处开始

    new_free = p_obj_start + size;//新的空闲空间的起始地址为老的空闲地址加上新分配对象的大小

    if (new_free 和 p_obj_start 都在同一个块内，即剩余空闲空间能容纳此对象)

    {
        p_return_object = p_obj_start;
        p_return_object->vt = (VTable *)p_vtable;

        my_nursery->free = new_free;//更新新的空闲指针

    }else
        p_return_object = NULL;
    return p_return_object;
}
```

图 15.18 快速分配过程

·慢速分配：

慢速分配结果只有两种可能，一是分配成功，二是抛出异常。LOS 上的慢速分配在 15.3.1 节讲过，这里不再重复。这一部分主要讲述慢速分配函数 gc_malloc_slow_no_constraints。gc_malloc_slow 根据对象类型限制的不同决定调用 gc_malloc_slow_no_constraints，还是 gc_pinned_malloc。一般来说，如果这个对象的 p_vtable->class_properties 为 0（表明对象无限制），调用前者。如果类属性的 CL_PROP_ALIGNMENT_MASK 或者 CL_PROP_PINNED_MASK 位为 1，则调用后者。除此之外的所有情况下都调用前者。

我们以 char[] i; i = new char[4000];语句序列为例说明无限制慢速分配是如何进行的。图 15.19 的伪码说明了这个过程。

```
Java_java_lang_Object *gc_malloc_slow_no_constraints (unsigned size, VTable *p_vtable)
```

```
{
restart:
    if (size >= los_threshold_bytes) {
        //太大的对象，我们只有在 LOS 为它分配空间
        return p_gc->gc_pinned_malloc(……) //见图 15.17
    }
    block_info *old_nursery = (block_info *)orp_get_nursery();
    //ORP 提供一个此线程属于 nursery 的块
    如果这个 nursery 块的剩余空间能容纳此对象，则修改空闲指针并返回分配好的地址；
    //同快速分配
    orp_change_nursery(NULL);//从当前线程结构中移去此 nursery 块
    block_info *p_new_nursery = p_cycle_nursery(p_old_nursery,false);
    //线程使用完了一个 nursery 块，用这个函数来获得一个新的 nursery 块，这个块状态必定是
    // acitive_nursery 。false 表示不允许返回 NULL，所以一定能获得一个 active_nursery 块
    orp_change_nursery(p_new_nursery);//为了以后的快速分配，将新的 active_nursery 块记录下来
    goto restart; //我们刚刚获得一个 nursery 块，所以还要在这个块中给对象分配一块空间
}
block_info *p_cycle_nursery(block_info *p_old_nursery, bool returnNullOnFail)
{
    if (p_old_nursery) p_old_nursery->nursery_status = spent_nursery;
    //此 nursery 块的原状态肯定是 active_nursery，由于它的剩余空间已不能容纳新对象，所以要将
    //它的状态置为 spent_nursery
    result = get_free_nursery();//获得一个空闲的 nursery 块，如果成功，块状态一定是 active_nursery
    if(result == NULL) {
        if (returnNullOnFail) return NULL;//如果允许返回 NULL，则返回 NULL
        while {result ==NULL}{
            通知 gc，nursery 块用完了，或者取一些空闲块作为 nursery 块，或者做垃圾收集
            result = get_free_nursery();
        }
    }
```

```
        修改获得的状态为 active_nursery 的块的 block_info 的相关域；
        return result;
    }
    block_info *get_free_nursery (){
        result = NULL;
        //nursery 中的块初始化时状态都是 free_uncleared_nursery，而分配只能在 active_nursery 进行
        for every i    if nurseries[i]-> nursery_status ==free_nursery
        {
            result = nurseries[i];
            线程安全地改变 result 的 nursery_status 为 active_nursery，如果成功则 break;
            否则 result = NULL；//本线程竞争失败，继续寻找
        }
        if(!result)
        //如果没有 free_nursery 的 nursery 块，我们只有求之于 free_uncleared_nursery 块，这是没有清
        //零的空闲块
        {
            找到一个 free_uncleared_nursery 的块 result;
            线程安全地改变 result 的状态到 thread_clearing_nursery；
            //这个状态是个过度状态，表明线程会对它清零
            将 result 所指的块的可分配空间清零；
            result->nursery_status = active_nursery;
        }
        return result;//如果所有 16 个 nursery 块都用完，只有返回空
    }
```

图 15.19 慢速分配过程

15.3.3 推迟的垃圾收集

让我们回到图 15.19 的 p_cycle_nursery 函数，当它想获得一个空闲的 nursery 块而不可得时，即 get_free_nursery()返回 NULL，它只有通知 gc 现在已无空闲的 nursery 块，然后 gc 开始回收堆。在 nursery 初始化时，我们仅仅给 nursery 分配了 16 个块，这 16 个块用完后，堆中还有很多空闲块可被分配。如果这时 gc 开始清除死对象，显然效率不高，所以在这种情况下，gc 可以选择推迟垃圾收集（除非 java 程序强制垃圾收集）。为了对垃圾收集分代概念有个直观的了解，我们来看看最早的 16 个 nursery 块被用完时堆的可能情况。当前堆总共 512 个块，属于 nursery 的块有 16 个，他们的状态都是 spent_nursery；属于 LOS 的块有 10+x 个（x 为单对象块的数目）；属于 step 的块还是初始化时的那一个块；属于 MOS（mature

object space）的块为 0，因为到目前为止，还没做过垃圾收集，所以不存在成熟对象。

推迟垃圾收集的执行上下文主要是：

1. 如果 nursery 块的数目相对线程的数目来说太少了，我们会增加等于线程个数的数目的 nursery 块。nursery 块太少的具体标准是 nursery_count < (int)(number_of_threads * 4。这样我们就有了新的空闲 nursery 可供使用了，再做 get_free_nursery()就不会返回 NULL。
2. 如果我们增加新的同原来同样多的 nursery 块后，已使用的块的总数仍不到最大的块数目 1024 的一半，我们可以选择推迟垃圾收集。首先我们将所有的状态为 spent_nursery 的块转移到 step 中，图 15.20 描述了这个重要的转移函数。新增和原来同样多的 nursery，这些 nursery 状态都是 free_uncleared_nursery，这样再做 get_free_nursery()就不会返回 NULL 了。

```
bool move_spent_nurseries_into_step(){
    for every nursery block this_nursery {
        block_info *new_nursery = get_new_nursery();
        //新获得一个 nursery 块，用来替换这个已使用完的 nursery 块
        if (new_nursery == NULL ) return false; //不能成功完成
        nurseries[i] = new_nursery;//修改 nurseries 数组
        this_nursery->c_area_p = false;//这个块并不是被收集区域
        this_nursery->in_step_p = true;//这个块已经属于 step 了
        this_nursery->in_nursery_p = false;//此块不属于 nursery
        this_nursery->nursery_status = bogus_nursery;//我们用这个状态表示它原来是个 nursery 块
        gc_block_status_table[this_nursery->block_status_table_index] = block_in_step;
        this_nursery->list_info = step;//从 list_info 我们可以找到此块属于的 step 的信息
        this_nursery->next = step->blocks;//将这个块添加到表头
        step->blocks = this_nursery;//更新属于这个 step 的第一个块
        this_nursery->scan = this_nursery->free;
    }
}
```

图 15.20 spent_nursery 块到 step 块的移动过程

15.3.4 垃圾收集过程

垃圾收集是 GC 最重要的功能，也是关系到 GC 性能的一个重要因素。在 13，14 章中，已经介绍了各种垃圾收集算法，每种方法在性能，实现难易方面有各自的特点。ORP 的 GC 模块的垃圾收集算法采用了拷贝，分代，sapphire，train 多种算法。用户可以通过命令行参数，也可以通过改变预编译选项定制适合自己的 GC 模块。下面我们要讨论的是其中一种 GC 模块，它采用的垃圾收集算法结合了拷贝，分代，train 等算法，体现了实现复杂度和性能的一个折中。虽然当前的实现引入了记忆集，但是在收集过程中并没有用到它们，可能留待以后实现；当前实现只采用页标记来减少需扫描的页个数。

垃圾收集主要分如下几个阶段进行：

1. train 调整。这个阶段的主要任务为，保证有两个 train。并根据 train 中 car 的状况以及调整次数决定是否把注意点放在收集一个 car。如果调整函数的返回值为 true，我们将得到一个需要重点注意的 car，叫做 focus_car，它等于第一个 train 的第一个 car。对第一次垃圾收集，可以看作是 MOS 空间的初始化。

我们来看看在做垃圾收集前，gc 是怎样调整 train 的。首先我们介绍一下描述 train 的数据结构。

```
struct train_info{
    car_info *cars;
    car_info *last_car;
    POINTER_SIZE_INT birthday;
    train_info *next;
    train_info *previous;
};
train_info *trains;
```

train_info 是个双向链表的节点结构。car_info 参见图 15.10，cars 域指向属于这个 train 的 car 链表的表头，last_car 域指向 car 链表的表尾，birthday 表示这个 train 的生日，值越大表示越年轻，next 域指向更年轻的 train，previous 指向更老的 train。trains 是这个 train 链表的头指针。图 15.21 给出了 trains 的一个示例。

图 15.22 的代码描述了调整过程。

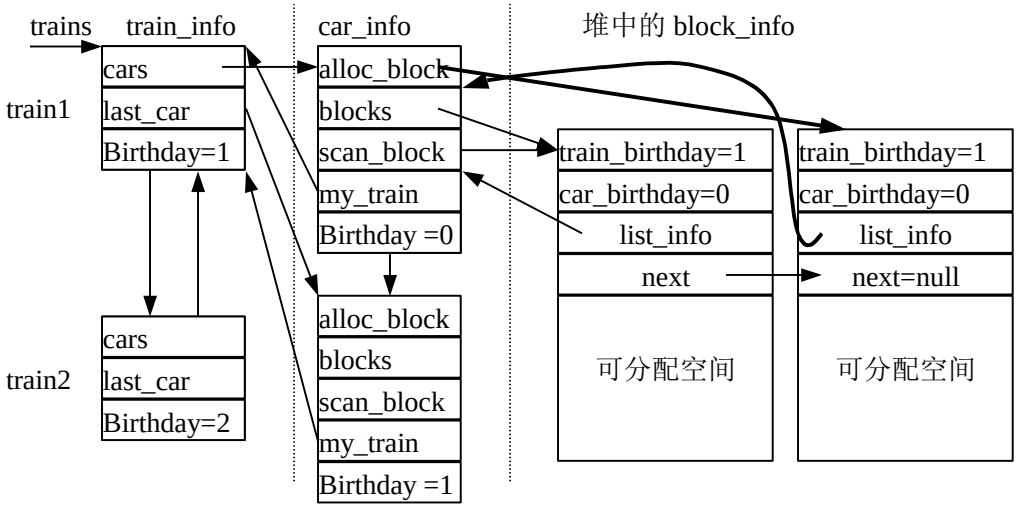


图 15.21 train 示意图

```
boolean train_adjustment()//如果返回 true，引起调用者只收集一个 car
{
    boolean do_gc = false;
    if (!trains) add_train();//保证有一个 train

    if (!trains->next) add_train();//保证总有两个 train

    对每一个 train，我们统计它的最后一个 car 中有多少块；如果数目大于一个预定值，在这个 train
```

```
        中增加一个 car;并且 do_gc = true;

        //我们增加一个 car，所以也要收集一个 car
        if (adjustment_count++ > 8) {
            //这次 train 调整是第 10 次，因为调整计数是从 0 开始

            do_gc = true;//此时要收集一个 car

            adjustment_count = 0;
        }
        return do_gc;
    }
```

/*在往 train 中加入 car 时，新 car 中至少有一个块。此时，堆中空闲块可能没有，如何获得一个空闲块呢？在图 15.8 中，我们说到如果不能正常获得一个空闲块，只有触发垃圾收集，或扩展堆。现在我们为做垃圾收集请求新块，不可能再次触发垃圾收集，所以只有扩展堆。*/

```
    struct block_group_link {
        block_info *block_list;
        block_group_link *next;
    };//当我们分配一组新的块时，我们用这个结构将各个组连接起来

    bool _extend_heap(int block_size_bytes){//参数为我们想扩展多少个字节
        if (delta_bytes < block_size_bytes)
            delta_bytes = block_size_bytes;//如果想扩展的字节数比缺省的要小，就按缺省值扩展

        如果扩展后的堆第一次超出了最大堆的大小，可以接受，如果再次超出，则返回 false;

        将堆的大小增加 delta_bytes 个字节；

        将增加的空间分成块，并初始化每块的 block_info;

        link_free_blocks ((block_info *)p_new_region, block_count) ;
        //将从 p_new_region 开始的 block_count 个块加入到空闲块表中

        block_group_link *new_group_link = (block_group_link *)malloc (sizeof(block_group_link));
        将这个新的组加到 block_group_list 中去；// block_group_list 为块组链表的头指针

        return true;//如果返回 false,调用者会处理;目前没考虑返回 false 的情况
    }
```

```
void add_car(train_info *a_train){//往 train 中加入一节新的 car
    car_info *new_car = (car_info *)malloc(sizeof (car_info));
    new_car->alloc_block = get_new_block(true, false, true);
    //第一个 true 表示可以通过扩展堆来获得一个新空闲块，false 表示不能返回空，第三个 true 表
    //示调用者正在做垃圾收集
    new_car->alloc_block->list_info = (block_list_info *)new_car;
    new_car->alloc_block->train_birthday = a_train->birthday;
    gc_block_status_table[new_car->alloc_block->block_status_table_index] = block_in_mos;
    //train 中的块都属于 MOS
    new_car->blocks = new_car->alloc_block;
    new_car->my_train = a_train;

    如果 a_train 中还没有 car，则 new_car 既是第一个 car，也是最后一个 car，它的生日为 0;

    如果不是，则 new_car 的生日比 a_train 的最后一个 car 的生日大 1，并把它加到 a_train 的表尾；

    new_car->alloc_block->in_free_p = false;//被用做 car 的块已不是空闲的了
}
void add_train(){
    train_info *this_train = (train_info *)malloc(sizeof (train_info));
    将 this_train 清零;
    this_train->birthday = current_train_creation_date++;
    // current_train_creation_date 是当前 train 的创建日期，每增加一个 train，要加 1
    //初始值为 1，而不是 0; 0 被 YOS 使用，包括 nurseries 和 steps
    add_car (this_train);
    if (!trains) trains = this_train;//这是第一个 train
    else
        从 trains 中找到最年轻的 train；并将 this_train 加到最年轻的 train 之后；
}
```

图 15.22 train 调整过程

2. 设置 step 中的 cheney 空间。此次垃圾收集之前我们要保证 step 中所有对象都被扫描过了，因为上次的垃圾收集一定扫描了所有对象。具体表现为：step-> alloc_block 等于 step->scan_block；step->alloc_block->free 等于 step->scan_block->scan。所谓 cheney 空间，就是需要做 cheney 扫描的空间。

当前的 step 中包含上次垃圾收集存放活对象的块，也包含本次垃圾收集前从 nursery 中提升上来的

块。step 中的现有块即将成为垃圾收集的目标，我们要给 step 一个新块，以便存放从此次收集从 nursery 中存活下来的对象，这个块也就是 cheney 扫描的目标；并且将 step 中原有块用 pp_step_from_blocks 指示出来，以便下一步将它们标记为收集区（块信息头的域 c_area_p 为真），并在所有活对象被移出之后，它们能够被回收。图 15.23 描述了整个过程。

```
void set_up_step_cheney_spaces(step_info *step, block_info **pp_step_from_blocks){

    *pp_step_from_blocks = step->blocks;//要收集的块在这个链表中，通过 block_info 的 next 链接

    step->blocks = p_add_new_step_block(step);//取一个新的空闲块作为 step 的第一个块

    step->scan_block = step->blocks;
    step->alloc_block = step->blocks;

    //step 中的块全部更换，原有的块*pp_step_from_blocks 来指示

}
```

图 15.24 设置 step 中的 cheney 空间

3. 设置我们要收集的范围。主要工作包括将所有的 nursery 块标记为被收集区；将第 2 步得到的来自于 step 的块(step_from_blocks)标记为被收集区；可选择将 MOS 块（即 trains 中的块）标记为被收集区，或只标记第 1 步得到的 focus_car 中的块。对于 nursery 块的标记，我们要做的只是将它的块信息的 c_area_p 标记为 true 即可。对来自 step 的块，我们通过第 2 步返回的表头指针和块信息的 next 域来遍历整个链表，并将每个块信息的 c_area_p 标记为 true。如果以强制的方式（forced=true）回收整个堆，我们也要将 trains 中的块标记；如果不是强制方式，我们只标记 focus_car 中的块。但是当前的实现 forced 在收集开始前总被置成 true ，所以当前实现 trains 总是被收集目标，trains 中存活的对象被移入更年轻的一个 train（这个 train 就在这一步被添加到 trains）中。所以 train 的标记稍微不同，见图 15.25。对 focus_car 的标记与 step_from_blocks 的标记相同。

```
void set_up_c_areas_in_all_trains ()
{

    将 train 中所有块标记为被收集区；//它们在此次收集的后段会被回收

    add_train();//其他 train 都是被收集区，所以再加入一个 train 以便移入存活的对象

}
```

图 15.25 将所有 train 设置成被收集区

4. 获得根集。这一步要调用 orp 提供的服务。除此之外，这一步还包括清空一些 hash 表，他们分别是指向 YOS 的引用的记忆集，所有弱引用的记忆集，指向 focus_car 的引用的记忆集，这些 Remembered_Set 的数据结构是用来为了性能更高实现也更复杂的 gc 算法准备的，但是 ORP 现有版本没有实现。15.3.5 节中，我们会简单说明这些数据结构的作用。

所有的根集都放在 Root_List 的对象中，我们维持两个这样的对象，名字分别为 p_root_set，和 p_verify_root_set。所有指向 YOS（包括 nursery 和 step，也包括 LOS）的引用可以用类 Remembered_Set 的对象存放，地址为_p_refs_to_young_object_space；所有弱引用，指向 focus_car 的引用也用这个类的对象存放，地址分别为_p_weak_references，_p_refs_to_focus_car。首先我们将所有这些对象清空，然后调用 ORP 的接口函数 orp_enumerate_root_set_all_threads()（见 15.2.1 节）。当返回 gc 时，p_root_set 和 p_verify_root_set 中已有同样的根集了。当前的实现中，其他的引用记忆集集没有被枚举出来。这一部分的具体实现请参考 ORP，JIT 的相关内容。

5. 确认堆的状况。包括确认所有的块表，nurseries，step，LOS，MOS，根集。这一步不会在

真正的垃圾收集代码中出现，讲解这一步的目的是为了了解整个堆的状况，加深对分代概念和页标记的理解，深入理解一些全局变量和成员变量的作用。这一步对垃圾收集无任何意义。

我们通过 block_group_list 来访问所有的块链表，并统计块链表中的块的使用情况。对于 nurseries，step，LOS 等确认的讲解，我们首先介绍一些基本例程 verify_slot，verify_card_mark，verify_object，verify_simple_block，verify_single_object_blocks，如图 15.26 所示。

```
void verify_slot(Java_java_lang_Object **p_slot)
{
    if(*p_slot == NULL) return;//如果引用成员为 NULL，无须做其他工作
    block_info *target_block_info = GC_BLOCK_INFO(*p_slot);
    //target_block_info 为引用成员所指对象所在的块的信息
    Java_java_lang_Object *the_obj = *p_slot;
    首先 the_obj->vt 不能为 NULL;
    the_obj->vt->clss 所在块一定在 LOS 中；
    the_obj->vt->clss 所在块一定是一个单块;//单块是指 number_of_blocks 等于 1 的块
    if(!target_block_info->in_los_p)
    {//LOS 中的对象不是按地址顺序分配的，所以不在此确认之内
        目标块的 free 指针一定比 the_obj 大；
        if (!target_block_info->in_nursery_p) 目标块的 scan 指针一定等于 free 指针；
        //这表示上次垃圾收集此块中的所有对象都被扫描过了，而 nursery 中的块在垃圾收集后还
        //会被分配出去，free 指针增大，所以不在此确认之内
    }
}
void verify_card_mark (Java_java_lang_Object *source, Java_java_lang_Object *target)
{
    //所谓的 source，target 是指 source 对象中 slot 指向的对象。当 source，target 之间存在
    //跨代引用关系，并且必须是前者属于 MOS 时，我们要检查 source 对象所在块的标记情况
    if (target == NULL) return;
    boolean mark = false;
    当 source 所在的块属于 MOS，并且 source 所在块的 train 比 target 的年轻（即 train_birthday 大，
    另外 nursery，step 块的 train_birthday 为 0）时，mark = true;
```

```
    如果两者的 train_birthday 相等，并且前者所在 car 比后者的年轻，则 mark = true;
    if(mark) {
      cardindex = GC_CARD_INDEX(source);//求出 source 对象在块中属于第几个页
      source 对象所在块的 card_table[cardindex]被标记
    }
    //综上所述，如果块中某个页有个对象有上述跨代引用行为，则整个页都被标记。这也是
    //card_table 的作用，在扫描 MOS 块时，我们就不必扫描所有页，只需扫描被标记的页
}
void verify_object(Java_java_lang_Object *p_object)
{
    //主要是对对象中的 slot 和对象所在块的 card_mark 进行确认
    p_object->vt 一定不能为空；
    if (is_array(p_object)) {//如果是数组
        if (is_array_of_primitives(p_object)) return;//它是一个基本类型的数组，没有什么可确认
        array_offset = init_array_scanner(p_object);//获取数组中第一个引用的偏移量
        while ((pp_target_object = p_get_array_ref(p_object, array_offset)) != NULL) {
            //p_get_array_ref 得到数组对象中一个 slot 的地址
            array_offset = next_array_ref (array_offset);//下一个引用的偏移量
            verify_slot (pp_target_object);//确认这个 slot
            verify_card_mark (p_object, *pp_target_object);//确认响应的 card_mark
        } // end while
        return;
    }
    unsigned int * offset_scanner = init_object_scanner (p_object);//对象中第一个 slot 的偏移量
    while ((pp_target_object = p_get_ref(offset_scanner, p_object)) != NULL) {
        //p_get_ref 得到一个普通对象中 slot 的地址
        offset_scanner = p_next_ref (offset_scanner);
        verify_slot (pp_target_object);
        verify_card_mark (p_object, *pp_target_object);
```

```
    }
}
void verify_simple_block (block_info *the_block){
    对这个块中的每个对象做 verify_object；
}
void verify_single_object_blocks(){
    block_info *a_single_object_blocks = single_object_blocks;//全局变量存放了连续块代表的地址
    while (a_single_object_blocks) {
        首先确认 a_single_object_blocks 中的 mark_table 条目都为假；
        //此时我们还不知道这个 LOS 对象是不是活的，所以标记为假
        verify_object(object); //多个块中仅有一个对象，它就是 object
        a_single_object_blocks = a_single_object_blocks->next;//下一个连续块的代表
    }
}
```

图 15.26 一些基本确认例程

对 nursery 的确认，我们只要对 nursery 中的每个块做 verify_simple_block，并且要求每个块的 train_birthday 和 car_birthday 都为 0（因为它们属于 YOS）。对于 step，同样是对每个块做 verify_simple_block，其他一些要求是每个块的 list_info 域必须指回 step，in_step_p 域为真，train_birthday 和 car_birthday 都为 0（step 也属于 YOS）等等。

对于 LOS 中的块，它的 block_info 中有一个 mark_table 数组，块中每 64 个字节对应表中一个条目，在做垃圾收集前所有条目都为假。一旦扫描时发现对象是活的时，我们就将它在 mark_table 中对应的条目置为真，这样所有为假的条目对应的空间都会被回收。所以对 LOS 块的确认，要求 mark_table 的所有条目为假。根据 los_free_link 链表，我们可以得到块中哪些段已经存放了 LOS 对象，对每一个对象做 verify_object。LOS 中的块还包括那些存放了一个大对象的多个连续块，我们用 verify_single_object_blocks 来确认。

对 MOS 块的确认，主要对 train 中的所有块做 verify_simple_block。具体包括检查 card_mark 数组的正确性，检查 train，car 中的信息是否满足各自的要求，比如 car_info 中的 train_birthday 一定要和它所属 train 的 train_birthday 一致。

最后是对根集的确认。我们只要对根集指明的每一个活对象做 verify_object。

6. 前面各步可以说是垃圾收集的准备阶段，这里是垃圾收集执行阶段。

首先我们要做的是将根集的对象引用指明的活对象转移到适当的块中，这是对根集的第一层扫描。如果这个活对象在 nursery 中，转移的目标块应该是 step 中的 alloc_block；如果在 LOS 中，则无须移动，但要做一些标记和对象扫描工作；除此之外，目标块为最年轻的 train 中最年轻的 car 的 alloc_block。图 15.27 说明了转移过程和几个通用例程。对象转移后，我们根据转移函数的返回地址更新运行栈中的对象引用到对象的新地址。

```
inline bool is_object_forwarded(IN Java_java_lang_Object *p_obj)
{
    //这个内联函数判断一个对象是否被转移过
```

```
    Object_Gc_Header *p_gc_hdr = P_OBJ_INFO(p_obj);
    //获得对象的 gc 头部，实现和对象布局有关,一般为 p_obj+4
    如果它的 gc 头部的最高位为 1，表明已被移动过，返回真，否则返回假；
    //如果被移动过，gc 头部的后 31 位即为对象的新地址
}
inline Java_java_lang_Object *p_get_already_forwarded_object(Java_java_lang_Object *p_obj)
{//如果对象已经被移动过，通过这个内联函数获得新地址
    Object_Gc_Header *p_gc_hdr = P_OBJ_INFO(p_obj);//获得对象的 gc 头部
    清除*p_gc_hdr 的最高位，即得到对象新地址 p_new_obj；
    return p_new_obj;
}
Java_java_lang_Object *p_get_forwarded_object (block_info *target, Java_java_lang_Object *p_obj)
{
    //返回对象的新地址；如果对象已经被移动过了，我们要做的仅仅是返回新地址
    如果 target 不为空，则要求 target 块必定不在被收集范围内；
    如果 p_obj 所在的块不在被收集范围内，我们只要返回 p_obj 即可；
    //MOS 中的块可能不在被收集范围内，也可能都在，它依赖与第 3 阶段做的设置
    if(GC_BLOCK_INFO(p_obj)->in_los_p)//如果要转移对象所在块在 LOS 中
    {//我们要做的是标记并扫描这个对象，将原地址返回
        Java_java_lang_Object *temp = mark_and_scan_object(p_obj);//temp 一定是原地址
        return p_obj;
    }
    if (is_object_forwarded(p_obj)) {
        //如果对象已经被转移过了，则这个对象所在块一定在被收集范围之内，一定不在 LOS 中
        return p_get_already_forwarded_object(p_obj);//只要返回它的新地址
    }
    //下面的代码处理要转移的情况
    if   target 中剩余空间还能容纳 p_obj 对象{
```

```
        对象的新地址为 p_obj_start = target->free；
        修改 target->free；
        修改 target 的 card_last_object_table 的对应条目到 p_obj_start；
        将对象拷贝到新地址，并修改老地址处的 gc 头部信息为最高位为 1 的新地址；
        return p_obj_start;
    }
    //如果 target 中的剩余空间不够，执行下面的代码
    if(target->train_birthday) {//如果目标块是一个 car 中的块
        首先找到这个块所属的 car 和所属的 this_train，并统计这个 car 中块的数目 block_count;
        if (block_count > GC_MAX_BLOCKS_PER_CAR) {
            //如果 car 中块数目较多，不能再加块，我们只有增加一个 car
            add_car(this_train);
            return p_get_forwarded_object(this_train->last_car->alloc_block, p_obj);//转移到新 car 中
        }
    }
    //如果目标块不属于一个 car，或者属于但 car 中还能加一些块，执行如下代码
    获得一个新的空闲块 new_alloc_block；//如果已无空闲块，获取的过程只能是扩展堆
    new_alloc_block 的属性应该和 target 的一样，所以根据 target 修改 new_alloc_block 的信息；
    //属性包括 list_info,所属 car，train 的生日
    根据 target 来修改 new_alloc_block 在 gc_block_status_table 中对应的条目；
    target->next = new_alloc_block；//将新块加在表尾
    target->list_info->alloc_block = new_alloc_block;//修改所属块表的表尾信息
    target = target->next；
    return p_get_forwarded_object(alloc_block, p_obj);//重试转移
}
```

图 15.27 对象转移过程

从转移函数可以看到，对除 LOS 中以外的对象，我们仅要做转移就可以了，无须扫描对象中的 slot；

但是对于 LOS 中的对象我们不仅要做标记还要扫描 LOS 对象中的 slot，转移它的后代对象（它的 slot 指明的对象）。现在我们来看看 LOS 对象的 mark_and_scan_object。首先介绍几个简单例程，is_object_marked 返回对象所在块的 mark_table 中对应条目的值。如果对象被标记过了，说明这个对象已经访问过了，无须再做标记和扫描。set_object_marked 将对象所在块的 mark_table 的对应条目置为真，说明此 LOS 对象已经被访问过了。从对象地址找到 mark_table 的对应条目很简单，因为对象在 LOS 中要满足 64 字节的边界条件，这里不在给出 set_object_marked 函数的代码。图 15.28 说明了 mark_and_scan_object 的主要过程。

```
Java_java_lang_Object *mark_and_scan_object(Java_java_lang_Object *p_obj)
{
    if (is_object_marked(p_obj)) return p_obj;
    set_object_marked(p_obj);
    scan_object(p_obj);
    return p_obj;
}
```

图 15.28 LOS 对象的标记和扫描

在介绍 scan_object 例程之前，我们来介绍有趣 slot 的概念。有趣 slot 是指这样的 slot，其中的指针从老的一代指向年轻的一代（从块的 birthday 来看，应该是年轻的指向年老的）。如果有趣 slot 包含一个指向当前正被收集的代的指针，这个 slot 就会被处理，如果指向的代不在被收集之列，这个 slot 的地址就会被插入这个代的记忆集中，以备以后的收集使用。scan_object 所要做的就是扫描对象中 slot，看这个 slot 是否为有趣的，并且转移 slot 所引用的对象。图 15.29 说明了 scan_object 的过程。

```
void scan_object(Java_java_lang_Object *p_object)
{
    mark_card = false;
    根据 p_object 很容易得到所在的块信息 slot_block_info；
    //如果我们在对象中找到一个有趣 slot，我们就将它置成 true，并决定是否标记对象所在的 card
    for 对象中的每一个 slot{
        //通过 p_obj->vt->gc_information 可得到每个 slot，如果对象是指针数组，更易取得每个 slot
        if (scan_slot (pp_target_object, slot_block_info))//如果此 slot 为有趣的，返回 true
            //pp_target_object 即为 slot
            mark_card = true;//只要此对象有一个有趣 slot，它所在的 card 就要被标记
    }
    if (mark_card) slot_block_info->card_table[GC_CARD_INDEX(p_object)] = true;
    //这个 card 被标记后，下一次这个 card 就会被扫描到
    return;
}
inline boolean scan_slot(Java_java_lang_Object **pp_target_object, block_info *slot_block_info){
    boolean mark_card = false;
```

```
    if (*pp_target_object == NULL) return mark_card;//slot 内容为空，说明这个 slot 不含后代对象
    if (target_block_info->c_area_p) {//后代对象所在块在收集范围内
        if (target_block_info->in_nursery_p)//如果后代对象在 nursery 中，转移到 step
            *pp_target_object = p_get_forwarded_object (step->alloc_block, *pp_target_object);
            //上面语句转移后代对象，并且修改父对象的 slot
        }
        else{
            如果父对象所在块 slot_block_info 在 car 中，则后代对象的目的地为同列 train 的最年
            轻 car 的最后一个块；
            如果父对象在 step 中 则后代对象的目的地为最年轻 train 的最年轻 car 的最后一个块；
            目的地块为 to_alloc_block；
            *pp_target_object = p_get_forwarded_object (to_alloc_block, *pp_target_object);
        }
    }
    //下面是后代对象已经被转移过或者后代对象不在收集范围内，看此 slot 是否属于有趣 slot
    target_block_info = GC_BLOCK_INFO(*pp_target_object);//移动后的后代对象的块信息
    /*由以下几条，判断是否为有趣 slot：只有 MOS 中的 slot 才是有趣 slot;YOS 和 LOS 对象都是
    在日期 0(所在块的 train_birthday=0)时出生，总是比较老的;父对象在 train 中，并且后代对象年
    老（由 train_birthday 决定）,包含后代对象的 slot 就是有趣 slot，这种情况包含后代对象在 YOS
    或 LOS 的情况，因为它们总是年老的；父对象和后代对象都在一个 train 中，但父对象所在 car
    要年轻（由 car_birthday 决定）*/
    如果这个 slot 是有趣的，则 mark_card = true;
    return mark_card;
}
```

图 15.29 对象扫描过程

讲过对 LOS 对象做标记和扫描之后，我们再回过头来看 p_get_forwarded_object。我们可以很容易看出这是一个递归函数。递归只发生在要转移的对象是个 LOS 对象或者选定的目的块不够存放这个对象。对于后者，我们只要更改目的块重试；对于前者我们首先要标记这个对象为已访问，然后扫描这个对象

的各个 slot，对各个 slot 包含的后代对象根据情况再做转移，扫描结束后，如果发现有一个有趣 slot（这一步不可能存在有趣 slot，因为有趣 slot 只存在于 MOS 中），则标记这个对象所在 card。如果我们把这一步看做以根集为根的搜索的话，那么除了 LOS 对象，我们只做了一层搜索；对 LOS 对象，我们一直搜索，直到它的后代不是 LOS 对象为止。转移过程的大致效果图见 15.30。

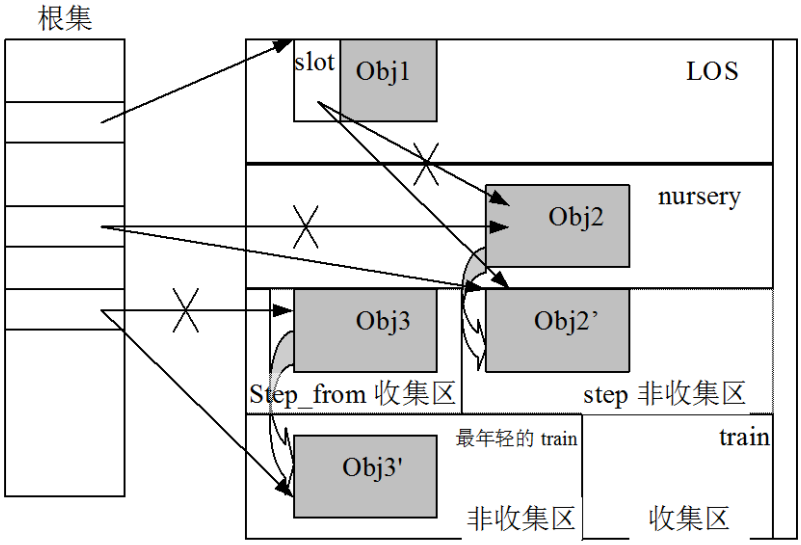


图 15.30 转移过程效果图

根集的一次扫描之后，从根集可达的第一层对象（除 LOS 对象，LOS 活对象也作了标记）都到了非收集区。它们的后代对象是一定不能被清除的，所以我们还要做搜索，这次搜索的起点就应该是非收集区内的对象。对每一个 train（从年轻的到年老的），首先要扫描 train 中的块，然后作一次 cheney 扫描。前一步的工作，使后一步 cheney 扫描起点对象集合尽可能大，从而效率高。扫描块使用 scan_cards_in_block，见图 15.31；cheney 扫描见图 15.32。对前一步我们并不是扫描 train 中所有的块，我们只扫描在非收集区中的块，因为收集区中的对象不一定存活，它的后代对象也不一定存活，扫描收集区中的块引起的后代对象转移是无意义的，很可能将死对象转移到非收集区中，从而使后面的 cheney 扫描做了无用功。扫描 train 中的块的作用就是将从 train 中非收集区可达的对象（由有趣 slot 包含）移出收集区。cheney 扫描是从非收集区内的对象出发，将一步或多步可达的对象移出收集区。使用页标记技术，我们不用扫描整个块，而只要扫描块中标记过的页。

```
void scan_cards_in_block (block_info *this_block){

    求出第一个 card 中第一个对象 first_obj；

    for every card_index

        求出此 card 最后一个对象 last_obj；

        if (this_block->card_table[card_index]) {
            this_block->card_table[card_index] = false;
            scan_card(first_obj, last_obj);
        }
        first_obj = last_obj;

        //对象属于哪个 card 由对象的起始地址决定；如果一个对象跨两个 card，则是前一个 card

        //最后一个对象，又是后一 card 的第一个对象

}
```

```
void scan_card (Java_java_lang_Object *first_obj, Java_java_lang_Object *last_obj){
    for every object between first_object and last_object do
        scan_object (this_obj);
    return;
}
```

图 15.31 扫描 train 中非收集区块中页的过程

```
void do_cheney_scan (){

    bool object_scanned = true;// 表明扫描未结束

    while (object_scanned) {

        object_scanned = false;//如果一遍扫描不会增加新的扫描对象，说明扫描完毕

        //首先扫描 step

        block_info *the_scan_block = the_step->scan_block;//从 step 的扫描指针指向的块开始扫描

        for every block in step do{

            scan_ptr = (Object_Gc_Header *)the_scan_block->scan;//块内的扫描指针

            while (scan_ptr < the_scan_block->free) {
                scan_object(scan_ptr));

                //可能会引起后代对象的转移，the_scan_block->free 也可能变化

                计算下一个对象的地址到 scan_ptr;

                object_scanned = true;//扫描对象后，后代对象仍需扫描

            }
            the_scan_block->scan = scan_ptr;
            the_step->scan_block = the_scan_block;
            the_scan_block = the_scan_block->next;
        }

        //扫描 step 可能引起一些后代对象转移到 train 中，所以我们对 train 做 cheney 扫描

        for train 中所有块 the_can_block do{

            //无论对 train 还是 car 都从年老的开始，因为后代对象的转移只会影响年轻的块

            while (scan_ptr < the_scan_block->free) {//一开始 scan_ptr 是块中第一个对象的地址

            //如果还未进入循环扫描指针和空闲指针就相等说明这个 train 中的块一定属于收集区

                scan_object( (scan_ptr));

                求块中下一个对象的地址 scan_ptr;
```

```
                object_scanned = true;
            }
        }
    }
    return;
    /*cheney 扫描之后能保证尽可能多的活对象都到了非收集区，收集区中是死对象，或是活对象
    的拷贝*/
}
```

图 15.32 cheney 扫描的过程

有了图 15.31 和图 15.32 的过程，我们回过头来看看根集第一层搜索后的整个扫描过程。如图 15.33 所示。

```
train_info *this_train = get_youngest_train();//从最年轻的 train 开始搜索循环
while (this_train){
    对此 train 中的所有非收集区块 this_block，做 scan_cards_in_block (this_block);
    do_cheney_scan()；
    this_train = this_train->previous;
}
```

图 15.33 第一层搜索后的扫描过程

7. 将所有活对象拷贝到非收集区后，收集区中的对象不是死对象，就是活对象的无用拷贝，整个收集区此时就能被清除和再利用。整个收集区包括 YOS（nursery，step_from 区），部分 MOS，LOS。

首先来看 YOS 的清除。对于每个 nursery 块，它的 c_area_p 标志都应该设为 false；如果块的状态原为 spent_nursery，则现在状态设为 free_uncleared_nursery，因为我们还未将这个块清零；如果原状态为 active_nursery，则将从块可分配地址开始到 free 指针所指的空间清零，free 指针指向块的可分配地址。

再来看 MOS 的清除。MOS 的清除实现同样依赖于 forced 参数（见第 3 阶段）。对于强制方式的回收，上次垃圾收集后保留下来的 train 都是收集区，所以需要清除这些收集区，而不是仅仅一个 car。做 train 的收集区设置时我们有了 3 列 train。前两列是收集区，后一列是非收集区，所以现在我们将前两列 train 中的所有块释放，用 link_free_blocks 将释放的块连接到空闲块表中，并将前两列 train 中的所有 train_info，car_info 数据释放（它们不属于堆空间）。清除结束时 train 表中应该只有一列 train。对于非强制方式的回收，如果要收集 MOS，也只是收集 focus_car 中的块，focus_car 如果存在，一定是第一列 train 的第一节 car（因为它最老）。将此 car 中的块释放并连接到空闲链表中后，释放这个 car 占的空间，如果第一列 train 中已无 car，则释放第一列 train，第二列成了第一列。

LOS 的清除和其他的都不一样，因为 LOS 对象不是顺序分配的（参见 15.1.5）。LOS 分两部分，一部分为 LOS 桶，一部分为单对象块。首先我们要清除所有 LOS 桶，然后清除单对象块。清除过程如图 15.34 的代码所示。

```
void sweep_los_bucket(block_info *bucket){//我们需要对每一个 los_buckets[i:0..9]调用这个函数
    bool whole_block_free = true;//如果整个块都没有活对象，则整个块就可以成为空闲块
    block_info *this_block = bucket;
```

```
    unsigned int size = this_block->los_object_size;//段的大小
    while (this_block) {//对桶中的每一个块，将死对象清除
        求出第一个对象的地址 an_obj_start 和最后一个可能对象的地址 last_possible_obj_start；
        while(an_obj_start <=last_possible_obj_start){
            if(!GC_LOS_MARK(an_obj_start)){//通过查 mark_table 可以得到对象是否被标记
                memset (an_obj_start, 0x0, size);
                ((los_free_link *)an_obj_start)->next = (los_free_link *)(this_block->free);
                this_block->free = (void *)an_obj_start;//将释放的段插入到空闲段表的头部
            }else {
                whole_block_free = false;//至少有一个活对象
                set_object_unmarked(an_obj_start);//将这个对象的标记去掉，为下一轮收集做准备
            }
            an_obj_start = an_obj_start + size；//下一个对象
        }
        if ((whole_block_free) && (bucket != this_block)){
            //当整个块都没有活对象，并且不是 los_buckets[i]的第一个块时，可以释放这个块
            首先将这个块从以 los_buckets[i]为头的链表中断开，然后连接到空闲块表中；
        }
        this_block = this_block->next;
        whole_block_free = true;
    }
}
void sweep_single_object_blocks (){
    block = single_object_blocks;//所有单对象块都可通过 single_object_blocks 访问
    while (block != NULL) {
        block = block->next;
        几个连续块中只有一个对象，所以对象地址 obj_header 应该是块代表的首地址；
        if 对象未被标记
            将这几个连续块连到空闲块表；single_object_blocks 也要改变
        else set_object_unmarked(obj_header);
    }
```

}

图 15.34 清除 LOS 中的死对象

step_from 区的清除很简单，我们只要将每一个块都连接到空闲块表就可以了。

8. 可选的块合并阶段。

无论是 nusery，step 还是 train 中的块都是单块，它代表它自己。这样当这些块被回收后，整个堆都被分成了一个个的单块。如果现在有一个占好几个块的大对象要分配空间，我们在堆中再也找不到几个连续的空闲块。几个单块在地址上是连续的，并不意味着它们是连续空闲块（原因参见 15.1.2）。所以此时我们要把连续的单块合并成一个块区。

实际实现中，我们给回收过程一个参数 size 表明是因为 nursery 空间不够还是 LOS 空间不够引起的垃圾收集。如果 size 等于 0，表明 nursery 空间不够，这样在这一阶段我们无需做合并，因为现在已经有足够的单块分配给 nursery；如果 size 大于 0，表明 LOS 空间不够，即使现在有足够的单块，也不见得有足够大的块区给大对象使用，这时我们只有将一些连续的单块合并成一个大块区。合并算法如图 15.35 所示。

```
void coalesce_free_blocks(){

    首先将 free_blocks 各个条目清空;//这个函数后来会重建 free_blocks

    block_group_link *the_group_list = block_group_list;//通过 block_group_list 可访问所有块

    while (the_group_list) {

        the_blocks = the_group_list->block_list;//第一个块组的块链

        while (the_blocks) {

            next_block = the_blocks->all_blocks_next;//紧邻的下一个块区，当然可能是单块

            if (the_blocks->in_free_p) {
                if (next_block) {
                    combine = next_block->in_free_p
                }else combine = false;

                //下面的 while 循环将最大化 the_blocks 代表的块区大小

                while (combine) {
                    the_blocks->number_of_blocks =
                        the_blocks->number_of_blocks + next_block->number_of_blocks;
                    next_block = next_block->all_blocks_next;
                    the_blocks->all_blocks_next = next_block;
                    if (next_block) combine = next_block->in_free_p;
                    else combine = false;
                }

                将 the_blocks 代表的一个块区链接到 free_blocks 中；

            }
            the_blocks = next_block;
        }
        the_group_list = the_group_list->next;
    }
```

}

图 15.35 块区合并

15.3.5 对现有垃圾收集过程改进的讨论

从 15.3.4 垃圾收集的各个阶段来看，代价最高的就是第 4 和第 6 阶段了。第 4 阶段依赖于 JIT，第 6 阶段依赖于 GC 的自身实现。第 6 阶段是一个搜索后代对象的过程，也就是如何将所有的活对象从收集区转移到非搜集区。这个搜索过程从根集开始，第一层的搜索的复杂度是线性相关于根集规模的，以后的搜索从非收集区开始，我们要扫描非收集区内的每个对象（包括因为扫描而引进的新对象），这个搜索的复杂度是跟活对象的数目成正比的。当前的实现由于将整个 MOS 空间都设为收集区，以致每个活对象经过一次垃圾收集肯定要被转移一次。可以说，这不是一个真正意义上的分代算法，不过它能比较容易地升级成分代算法。

再来讨论页标记，我们在做 gc_write_barrier 时会标记页，在 scan_object 时也可能要标记页，但我们仅仅在扫描 train 非收集区块中的页的过程（图 15.31）中才用到页标记。这个过程首先作用于最年轻 train 中的块，此时这个块中为从第一层搜索而转移过来的对象，那次转移没有做对象扫描所以没有页标记，做 cheney 扫描后，这个块中有些页被标记，但下一次这个块中再也不会被图 15.31 的过程扫描到。下一次垃圾收集这个最年轻的 train 在做 train 调整时就不是最年轻的了，并且接下来会被设成被收集区，同样也用不到页标记。所以我们有理由相信页标记和记忆集在这个 GC 模块中未使用。正是有了这个缺陷，预编译选项为 GC_GEN_V3 的 GC 模块不是真正的分代算法。

我们试着在此基础上实现一个真正的分代算法。首先为了减少对象拷贝，我们只收集 MOS 中的一部分，或根本就不对 MOS 进行收集，只有当 MOS 大到一定程度或做了几次垃圾收集后才对整个 MOS 收集。图 15.36 说明了这个收集过程。

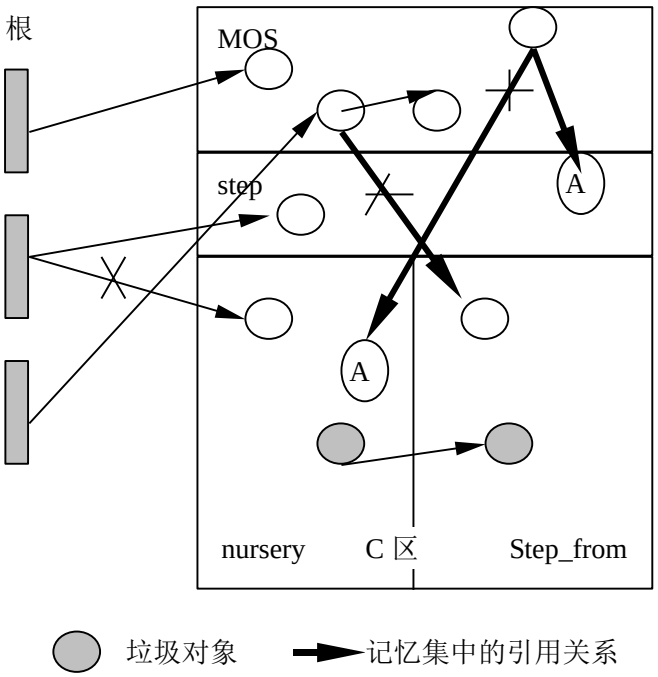


图 15.36 一种分代算法实现

从图中我们可以看出我们在记忆集中记录下所有跨代的引用关系。在作收集时，我们遍历根集和记忆集,将它们所指的对象转移到合适的区中如 MOS 或 step，再对 MOS 和 step 作 cheney 扫描。这样作可能将垃圾对象作了拷贝，如对象 A。但对 MOS 的研究发现 MOS 对象的死亡率很低，它们的后代对象时垃圾的概率也很小。所以总的看来，仍然大大减少了对象拷贝的代价。我们作对象转移时同样要跟踪新的引用关系，向记忆集增减引用关系。维护记忆集增加了代价，但是对于活对象很多的情况，拷贝代价

的减小相对于维护记忆集代价还是相当显著的。对于 MOS 的收集，我们可以采用 train 算法。

15.3.6 关于栅栏

栅栏是一种记录技术，用来收集有助于垃圾收集，或垃圾收集必需的信息。栅栏有读，写栅栏之分，顾名思义，就是分别在读写一个对象之前，记录下相关信息。栅栏的作用有些类似于分页系统的脏位的作用，一旦发现一个页帧被修改，就置这个位，在换出这个页帧时，如果发现已被修改，则写回磁盘，否则无需写回。栅栏也是一样。以写栅栏为例，当我们将一个引用写入一个对象时，首先需要知道我们所用的 GC 算法要求哪种栅栏。可以通过 GC 提供的 gc_requires_barriers 接口函数获知要求哪种栅栏。如果需要栅栏，ORP 调用 gc_write_barrier 来记录写操作引起的效应。不同的 GC 算法需要的栅栏是不同的。对于 GC_GEN_V3 的简单分代算法，它的栅栏仅仅将被写对象所在页在 card_table 的对应条目置位（card_table 的元素大小为一个字节，除此之外它很象一个脏位）。card_table 的作用在讲述垃圾收集过程时已有说明，这里不再赘述。因为写对象的操作在程序中是很频繁的，所以栅栏的代价一定要小，否则会影响程序运行的整体性能。

15.3.7 ORP 中的 GC 算法与拷贝算法的区别

GC_GEN_V3 实现的 GC 算法最基本的技术就是拷贝，但它没有 fromspace 和 tospace，并且由于引入了对象分代的概念，使得数据的聚集性比简单的拷贝算法有了很大的提高。在拷贝算法中，如果 fromspace 的大小为 x，则 tospace 的大小也要为 x，为了完成垃圾收集过程，总共需要 2x 大小的空间。而在我们的算法中，被收集区大小为 x，存活的对象大小总共为 y，我们可以动态地增大收集区地大小以满足活对象移动地要求，所以所需要的空间为 x+y。由于拷贝算法的 cheney 扫描宽度优先地搜索活对象，造成父对象和后代对象处于不同的页中，影响了对象访问的速度。而 GC_GEN_V3 引入了分代的概念，虽然它不是一个真正的分代算法，但它的引入还是提高了数据的聚集。因为同一代的对象在垃圾收集后仍处于同一代，显然能提高物理内存的命中率，减少页帧的换入换出。另外，LOS 空间的引入，消除了大对象拷贝代价。

参考文献：

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*,21(4):280-294,April 1978.

[Bis77] Peter B. Bishop. Computer Systems with a Very Large Address Space and Garbage Collection. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1977.

[Bro84] R.A.Brooks. Trading data space for reduced time and code space in real time garbage collection on stock hardware.In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*,pages 256-262,ACM,1984.

[CH98] P.Cheng,R.Harper,and P.Lee. Generational stack collection and profile-driven pretenuring.In *1998 ACM SiGPLAN Conference on Programming Language Design and Implementation*,pages 162-173,Montreal,Quebec,June 1998.Acm Press.

[Cha92] Craig Chambers. The Design and Implementation of the SELF compiler, an Optimizing Compiler for an Object-Oriented Programming Language. *PhD thesis, Stanford University, March 1992.*

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677-678,November 1970.

[Coh81] Jacques Cohen. Garbage Collection of linked data structures. *Computing Surveys*，13（3）：341－367，September1981.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A.J.Martin, C.S.Scholten, and E.F.M.Steffens. On-the-fly garbage collection: An exercise in cooperation. Communications of the ACM,21(11):966-975,November

1978.

[HH93] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. *ACM OOPSLA'93 Workshop on Memory Management and Garbage Collection.*

[HM01] Richard L. Hudson and J. Eliot B. Moss Sapphhire : Copying GC without Stopping the World. *SIGPLAN : ACM Special Interest Group on Programming Languages*. Pages: 48 - 57 ACM press.

[HM92] Incremental Collection of Mature Objects, *Proceedings of the International Workshop on Memory Management*,*September 1992*,pp.388-403.

[Hudson's guide] Hudson 的关于 ORP 的 GC 实现部分的 User Guide.

[Urs93] Urs Holzle. A Fast Write Barrier for Generational Garbage Collectors. In *OOPSLA '93 Garage Collection Workshop*, 1993.

[Wil92] Paul Wilson. Uniprocessor Garbage Collection Techniques. Submitted to ACM Computing Surveys, 1994.

[Yua90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*,11:181-198,1990.

Sapphire: Copying GC Without Stopping the World**,** Hudson and Moss. Java Grande 2001.

Cycles to Recycle: Garbage Collection on the IA-64, Hudson, Moss,Subramoney and Washburn. ISMM 2000.