函数式语言和命令式语言的内存模型

Memory Models of Functional Languages and Imperative Languages

李博杰 PB10000603

在编写多线程程序时，函数式语言和命令式语言的主要区别是，函数式语言由于没有副作用，调度器可以轻松地同时执行多个没有依赖关系的任务；命令式语言的堆内存一般是共享的，需要程序员显式处理并行。

在命令式语言中，简单的顺序一致性内存模型（Sequential Consistency）限制了编译器优化；而Happen-Before模型又太弱，会出现数据凭空产生的问题，因此Java引入了复杂的Memory Model [4]。C++（不考虑其中的C子集）作为强类型的语言，在C++11标准 [3] 中借鉴了Java Memory Model。C++中的std::atomic<> 变量相当于Java中的atomic变量。这些模型在数学上并不优雅，而且经常产生违反直觉的结果。

但没有听说过Haskell之类的纯函数语言需要Memory Model，因为Haskell的函数都是没有副作用的，不存在多进程间共享内存，当然也就不存在竞争条件了。这样编译器就能充分发掘程序中的并行性，将不相互依赖的函数调用并行执行，充分利用CPU的计算能力。这样程序员就从锁和临界区的麻烦中解脱出来了，可以用更自然的方式表达程序逻辑。

然而，函数式语言并不是银弹，因为Haskell这样的函数式语言在处理输入输出和I/O这些与环境交互的事情时，由于不能有副作用，只好用Monad。Monad需要从程序的入口穿过所有处理逻辑，一直到程序的出口，就像一根管子，管子里流过的是I/O操作。即使一个函数与I/O毫无关系，这根管子也要从它那里穿过去，这看起来很不优雅。

由于现实世界的应用系统大部分是与外界环境（用户和持久化数据）打交道的，麻烦的I/O是纯函数语言的重要短板。Erlang就是吸收了命令式语言表达逻辑方便性与函数式语言的纯粹性。

Erlang中的线程开销很低，一个线程只需要512字节，而一个Java线程需要512KB，几乎是1000倍的差距 [2]。C++和Java都是从单线程语言演变而来的，因此在设计之初没有考虑到多线程环境。它们的线程是相对重量级的。

Erlang的内存模型是，每个线程有一个独立的堆和栈，进程间通信要通过messages（消息）或者特别分配的binary heap来进行。这样，线程内部的堆和栈处理不需要任何锁和同步机制，线程间通信是通过Erlang的消息队列，Erlang只需要保证消息队列的一致性，这是比较简单的。

由于Erlang的线程开销很小，每个线程一般只完成一件特定的事情，这样线程的存活时间就比较短。线程一旦结束，它的私有堆和栈就可以立即被销毁。这样就大大降低了垃圾回收算法的复杂度。例如，Erlang可以在内存不足时再运行标记回收的垃圾回收算法，而不需要采用引用计数的方法每时每刻维护内存使用情况，因为Erlang可以期望一些线程能在较短的时间内结束，这样就可以大大降低垃圾回收算法对程序正常执行带来的开销。

Java的内存模型是，所有线程的数据都分配在同一个堆里。此外，JVM还会将代码缓存等数据区域放在公共的堆里。这样堆里的数据就无法被各线程分开了，因此需要采用复杂的同步算法。即使如此，由于Java Memory Model允许多种编译器优化，Java程序员还要小心同步，避免一些违反直觉的奇怪情况，如 [6]。因此编写并行Java程序是比较困难的。

当然，Erlang通过消息传递进行进程间通信，有可能导致消息队列的长度不断增长，从而消耗掉大量的内存。在线程间速度不匹配时，这种问题尤为明显，而且队列中未处理的消息所占用的内存是不可能被垃圾回收的。

事实上，让每个线程使用自己的堆空间在其他语言中也被采用。PHP是用于Web的语言，PHP-fpm服务器有一个线程池，每个线程处理一个HTTP请求，PHP解释器的C代码中一般使用emalloc在内存池里分配内存。在HTTP请求处理完毕后，除了pemalloc分配的持久内存，用emalloc分配的内存就会被全部释放，这样即使运行库的C代码里有内存泄漏，只要PHP脚本的执行时间不是很长，就能在脚本结束时释放全部内存，从而不会影响到PHP服务器的正常运行。这种线程之间独立的模型也给程序员增加了一些限制，使得程序员更多考虑任务的分离，降低代码的耦合性。

在类型系统方面，以Erlang为代表的函数式语言多是动态类型的，Java、C++、Scala等语言都是静态类型的。静态类型是指对象的类型在编译时即可确定。动态类型语言编写起来更灵活，但一般来说无法生成像静态类型语言一样高效的代码。需要注意的是，动态类型语言并不等于语言不是类型安全的。强类型（类型安全）的语言可以在运行时检查对象的类型，不过类型检查是有一定开销的。因此动态类型语言一般都会在编译期做一些类型推断，将代码中占大部分的静态类型推断出来并进行检查，将为数不多的编译时无法确定的动态类型留待运行时确定和检查。从这个角度考虑，动态类型语言一般是强类型的 [6]，因为对象在内存中的表示本来就带有类型信息。当然Perl、PHP等动态类型的脚本语言是弱类型（非类型安全）的，它们的解释器会在类型不相符时自动进行隐式类型转换。

Java语言表面上是类型安全的。但由于Java的静态类型限制，程序员需要强制类型转换。例如collection的元素只能是同一种类型，而应用很可能需要多种类型的元素，于是只好把它们的父类塞进collection，从collection中取出元素时再将它们强制类型转换成子类。这样事实上丧失了类型安全的特性。[5] 从这个角度看，动态类型语言实现的类型安全比静态类型语言更符合程序员的实际需求。

内存模型会影响语言是否是类型安全的。[8]证明了Java Memory Model是类型安全的；但Happen-Before Model就不是类型安全的，因为可能有凭空产生的数据，这种凭空产生数据的类型是不确定的。因此要开发一种类型安全的语言，就必须选择合适的Memory Model。Java Memory Model是这方面的鼻祖，C++11模仿了Java，LLVM又模仿了C++。LLVM在底层拥有高级类型系统，可以表示基本类型、结构体类型、数组类型、指针类型等，足以表示类型安全语言的对象。[7] 一个在虚拟机层面支持不好的反例是JVM，它拒绝在虚拟机层支持Java的template，使得Java的模板只能像C++一样在编译期使用，再转换成不带模板的JVM代码，从而不能发挥模板的全部威力。[5] 可见合理的虚拟机指令系统和数据类型系统是很重要的，会影响到运行在其上的语言能否充分利用其语言结构。

参考文献：

[1] Efficient Memory Management for Message-Passing Concurrency --- part {I}: Single-threaded execution, Jesper Wilhelmsson, Department of Information Technology, Uppsala University

[2] Erlang memory architecture vs. Java memory architecture: <http://java-monitor.com/forum/showthread.php?t=890>

[3] The C1x and C++11 concurrency model, Mark Batty, University of Cambridge

[4] JSR-133: Java™ Memory Model and Thread Specification

[5] Beyond Java, Bruce A. Tate, O’Reilly Media, 2005

[6] Java Memory Model引发的血案 <http://blog.csdn.net/historyasamirror/article/details/6254820>

[7] LLVM Atomic Instructions and Concurrency Guide <http://llvm.org/docs/Atomics.html>

[8] The Java Memory Model is Type Safe, Andreas Lochbihler, Karlsruhe Reports in Informatics