CN43-1258/TP
ISSN 1007-130X

计算机工程与科学
COMPUTER ENGINEERING & SCIENCE

2008 年第 30 卷第 12 期
Vol. 30, No. 12, 2008

文章编号:1007-130X(2008)12-0142-04

一种面向分布式系统的程序设计模式*

A Programming Pattern for Distributed Systems

李慧霸,彭宇行,卢锡城
LI Hui-ba, PENG Yu-xing, LU Xi-cheng
(并行与分布处理国家重点实验室,湖南 长沙 410073)
(National Laboratory for Parallel and Distributed Processing, Changsha 410073, China)

摘 要:分布式系统的程序设计模式主要包括多线程模式和事件驱动模式,其中事件驱动模式占据了主导地位。本文讨论了这两种模式的不足,以及 Coroutine 模式相对它们的优点,并认为 Coroutine 模式是最适合分布式系统的程序设计模式。本文在此基础上提出了 Libresync,它是一个基于 Coroutine 模式的分布式系统基础支持库。它既能给应用开发带来非常清晰的控制流程,又具有很高的灵活性和表达力,性能也能满足大多数需求。

Abstract: Programming patterns for distributed systems mainly includes multi-threaded and event-driven patterns, and the event-driven pattern is the major one. This paper, however, discusses their disadvantages, and the Coroutine pattern's advantages over them. We think Coroutine is the most suitable pattern for distributed system programming. And we present Libresync, a Coroutine-based supporting library for distributed systems, which brings us clear control flow, high flexibility and expressiveness. The performance of Libresync is also reasonably good enough for most purposes.

关键词:分布式系统;并发;多线程;事件驱动;Coroutine

Key words: distributed system; concurrency; multi-threaded; event-driven; Coroutine

中图分类号:TP311.5 **文献标识码**:A

1 引言

分布式系统具有开发难度高、维护复杂度高的特点。分布式操作系统、网格、中间件等研究工作分别采用不同的方法,试图构建一个理想的分布式计算环境。然而这仍然是一个远未解决的问题。

Raymond[1]将软件开发的复杂度分为本质复杂度、选择复杂度和偶然复杂度。本质复杂度是问题本质具有的复杂度,是任何方法都不能降低的复杂度;选择复杂度是由于一些人为的额外要求(即所谓"选择")所带来的复杂度;偶然复杂度则是因为开发人员"偶然"地采用了不合适的方法所带来的复杂度。分布式系统一方面具有很高的本质复杂度,另一方面开发人员又引入了很高的偶然复杂度,这两者掺杂在一起,相互之间具有非常"微妙的联系"[2],使得我们难以同时将这些问题解决。

分布式系统本质上是并发的,系统中的各个节点并发地与其他节点进行交互。并发问题是本质复杂问题,而我们当前描述、处理并发的方式则带来了额外的偶然复杂度。多数研究人员采用多线程模式或者事件驱动模式来描述、处理并发,然而我们认为这两种模式都会在分布式系统的开发过程中引发很高的偶然复杂度。

多线程模式会带来各种竞态条件问题,为解决竞态条件问题而引入的锁又会导致死锁、活锁问题,因而研究人员普遍认为这种模式在许多方面不适合用来支持分布式系统中的并发任务[3]。

事件驱动模式会带来"stack ripping"[4]问题,这也是一种偶然复杂度,其具体表现包括:(1)在概念逻辑上原本一体的函数将被事件拆分成为多个实际函数;(2)许多原本概念函数的局部变量不得不移动到一个动态分配的状态结构体中,以使其生命周期能跨越多个拆分后的函数;(3)函数调用堆栈必须手动记录,以便在调试的时候能够查询。这些问题还会进一步导致软件模块之间的可组合性降低,单个模块的可演化性降低等高层问题。

为了避免现有模式引发的这些不必要的偶然复杂度,我们基于 Coroutine 模式设计并实现了 Libresync——一种

* 收稿日期:2008-04-13;修订日期:2008-07-10
基金项目:国家 973 计划资助项目(2005CB321800)
作者简介:李慧霸(1980-),男,四川仁寿人,博士生,研究方向为分布式计算;彭宇行,博士,研究员,博士生导师,研究方向为多媒体系统;卢锡城,教授,博士生导师,中国工程院院士,研究方向为计算机网络、大规模并行与分布处理等。
通讯地址:410073 湖南省长沙市砚瓦池正街 47 号并行与分布处理国家重点实验室;Tel:13467531920;E-mail:lihuiba@163.com
Address: National Laboratory for Parallel and Distributed Processing, 47 Yanwachi St, Changsha, Hunan 410073, P. R. China

更适合分布式系统的底层支持库，它具有如下特点：

(1)通过协作式的任务调度既消除了“stack ripping”[4]问题，又有效避免了多线程模式中的竞态条件和死锁问题；

(2)仅仅提供了几个函数作为接口，不对上层应用强加任何程序设计风格，使得应用模块具有更好的组合性和演化性。

2 相关研究现状

分布式系统的并发模式大体上可以分为三类：多线程模式、事件驱动模式和 Coroutine 模式。其中 Coroutine 在本质上可以说是协作式线程，在这里，我们为了强调两者的区别，以“线程”特指抢占式线程，以“Coroutine”特指协作式线程。

多线程模式已被广泛认为不适合分布式系统[3]，因而针对本问题的研究工作主要集中在其他两种模式上。文献[4]从理论角度将这个问题领域划分为几乎正交的五个方面，分析了这两种模式在这五个方面的各种特性，指出 Coroutine 模式是解决问题的“甜区”，并详细讨论了 Coroutine 与事件驱动的混合模式。文献[5]在文献[4]的基础上进一步论证了 Coroutine 模式的优点，并以实验验证了这种模式可以满足高性能的要求。Capriccio[6]是这种模式的一个实现，它提供了一套与 POSIX 兼容的接口，并通过一些特别技巧截获 libc 对内核的系统调用，可以对多数应用程序实现无修改移植。这种非常底层的实现方式严重影响了 Capriccio 的可移植性，此外它也没有考虑 C++异常相关的问题，使得 C++程序可能会出现不可预料的行为。

另一方面，也有许多研究工作致力于减少“stack ripping”带来的问题。例如，Mace[2]提出了一种针对分布式系统开发的 DSL，可以极大简化分布式应用开发的工作量。然而 Mace 语言采用了显式的状态机模型，并不适合人类的思维方式，所以并不是最好的方法。

Libeel[7]除了实现一个代码库，还包含一整套图形化的分析、调试工具，可以清晰地显示事件之间的各种关联，辅助事件驱动程序的开发。但是，Libeel 并没有给事件驱动带来实质性的变化。

Tame[8]对 C++语言进行了大幅扩充，引入第一类的事件、第一类的闭包以及一些语法糖，通过源代码到源代码(Tame 到 C++)的翻译，显著地简化了事件驱动程序的开发，并且保持了事件模式的灵活性。

Tame 的本质是将 Coroutine 风格的代码翻译成为事件驱动的代码，这样的工作，正如 Behren 在文献[5]中所述，“在实际效用上只是重复了线程(译著：此处指 Coroutine)的语法和运行时行为”。下面[8]是 Tame 文章中的例子，其代码与 Coroutine 模式具有十分类似的结构。

```
// 线程代码
void wait_then_print_threads() {
  sleep(10); // 将本函数及其调用者阻塞
  printf("Done!");
}
// Tame 代码(使用语法糖)
tamed wait_then_print_simple_tame() {
  twait { timer(10, mkevent()); }
  printf("Done!");
}
```

Tame 的理想目标是：像线程代码一样简洁直观，像事件代码一样灵活。我们认为 Tame 的简洁直观性源于它对 Coroutine 的模仿，或者说它给应用开发人员提供的模式在概念上与 Coroutine 相当；而 Tame 又兼具灵活性的特点，实际上是因为 Tame 模式是一种事件与 Coroutine 的混合模式。也就是说，若 Libresync 也采用这样的混合模式，则可以提供与 Tame 同构的简洁直观性与灵活性。这一点可以从图 1[8]以及本文第 4 节的对比中得出。

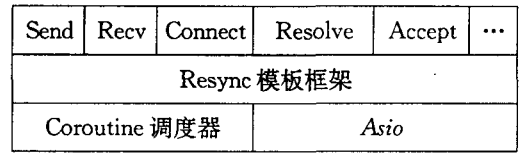

图 1 Libresync 组成与结构

3 原理与实现

如图 1 所示，Libresync 的组成可以分为上、中、下三层，其中下层为基础支撑部件，中层为 Libresync 的核心机理框架，上层是对应用提供的各种接口函数。

Libresync 下层包括 Coroutine 调度器和异步事件分派器两部分。其中 Coroutine 调度器实现了 Coroutine 的创建、挂起等基本操作，和互斥、信号量等基本同步原语。而异步事件分派器则直接使用了现有 Boost[10]库中的异步 I/O 组件 Asio。由于 Asio 是重用现有工作，在图 1 中以斜体文字来表达。Libresync 的上层提供了各种接口函数，其设计原则为清晰、简练，而各种接口功能的共同部分则包含在抽象化的中层框架内。Libresync 的中层框架是最重要的部分，它最能体现核心思想：通过显式地、确定性地调度 Coroutine，将所有异步操作在概念上被重新同步化(即“Resync”)，如图 2 所示。

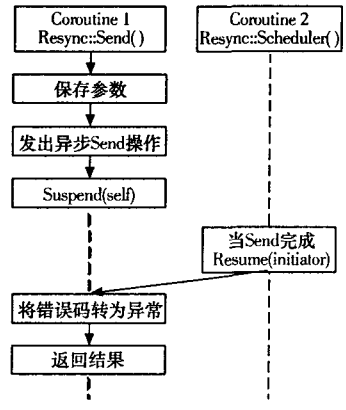

图 2 Libresync 工作原理(以 Send 为例)

Libresync 在发出 Send 操作命令之后，主动将自己所在 Coroutine 挂起，以等待 Send 的执行。当调度 Coroutine 收到异步 Send 完成消息的时候，将唤醒发出该操作的 Coroutine，以使其继续原有的执行。所有异步操作的变换过程十分类似，区别仅仅在于操作类型、参数和返回值。

很多面向 C 语言的 Coroutine 库并没有把 C++ Exception 纳入考虑范围，使得基于它们的 C++程序在遇到 Exception 时可能会出现不可预料的行为。例如，g++编译器支持两种异常方式，其中一种是基于 setjmp / longjmp

函数对实现的，这种方式就需要 Coroutine 在调度的时候显式地将 Exception 的状态作为执行上下文一起切换。VC++编译器使用的结构化异常处理机制与 Win32 的 fiber API 之间的兼容性非常好，因此我们首先在这个环境实现我们的 Libresync，但在原理上 Libresync 是平台无关的。

Coroutine 模式虽然没有竞态条件的问题，但在实际当中仍需要同步操作。例如，整个应用程序可能共享一个日志输出对象“Logger”，各个任务都将日志信息写入 Logger。我们有时会希望将某个任务的一些连续写入动作组成一个原子事务，不被其他任务的写入所搅乱。这时最合适的方法就是给 Logger 附加一个 Mutex 同步原语，在必要时任务可以“锁定”Logger。

Libresync 也可以看做是一个桥接器，它主要的工作是实现 Coroutine 模式代码调用事件模式代码（Boost::Asio）。我们同时也实现了反向的桥接，即事件模式代码调用 Coroutine 模式代码。其原理在于创建一个新的 Coroutine 执行被调用的函数，并且在该函数返回时通知调用方。Asio 采用泛型回调函数的方式进行事件通知。然而，当被包装函数执行完毕的时候，桥接器不能直接调用回调函数，必须使用 Asio::io_service 中提供的 Post() 函数在原始 Coroutine 上下文中发送完成消息。

4 实践验证

Libresync 最大的特点在于简化分布式系统开发，同时能够满足大多数情况下的性能需求。我们针对这两方面分别进行了三类对比：首先是我们根据自身经历，对各种模式进行评述；其次是将 Libresync 与 Tame 在灵活性方面进行对比；最终以 Web 服务器为例进行性能测试。

4.1 我们的经历

在一个内部的 P2P 项目中，我们先后尝试了这三种模式，对它们各自的特点深有体会。这个项目始于 2006 年初，当时我们几个参与人员都对网络编程没有多少经验，于是“偶然”地采用了貌似简单的多线程模式。然而，程序最终的稳定性甚至很难达到内部测试的要求，于是不得不寻求其他途径。

我们转为使用 Asio 库进行事件驱动的异步 I/O 编程。这次的尝试非常成功，我们很快就可以在学校范围内进行公开测试。然而，我们在这期间也深刻体会到了“stack ripping”问题。原本在概念上一体的函数不得不被拆分成多个实际函数；原本的局部变量不得不放置在一个状态结构体中，并且把状态对象在各个函数之间传递；甚至为了调试方便，原本不是状态的变量也需要纳入状态范畴。程序的正常流程被严重搅乱，我们甚至很难改进其中的 P2P 数据调度策略。

除此之外，程序还有一个严重的可演化问题：我们发现有一个相对底层的函数需要从同步操作变成异步操作。但出乎我们意料的是，这个微小变动的影响十分巨大，因为所有依赖这个函数的代码都必须从同步变为异步，并且这个影响还会递归地传递出去。因此，程序的可演化性十分的低下。

2007 年我们开始尝试 Coroutine 模式，随后开发出 Libresync。在移植应用代码的过程中，我们发现多线程模式和事件驱动模式中的各种问题都不存在了，控制流程非常清晰，对代码的调优也进展得十分顺利。因此，我们认为这种模式是最适合分布式系统的程序设计模式。

4.2 与其他工作的对比

我们认为 Tame[8] 在简化分布式系统开发方面做出了非常实质性的工作，因为它思路新颖，并且同时做到了事件驱动模式的灵活性和 Coroutine/多线程模式的表达力。Tame 最具特色的例子是并发 dns 查询（multidns_par），如下所示。

```
tamed multidns_par( char * name[], ipaddr a[],int n, event<> done) {
  twait { //使用 twait 语法糖
    for (int i = 0; i < n; i++)
      gethost_ev(name[i], mkevent(a[i]));
  }
  done.trigger();
}
```

Tame 的这个例子充分利用了“twait{…}”语法糖，实现解析完成的消息聚合，体现了 Tame 的灵活性与表达力。这种灵活性在 Coroutine 模式中也可以很容易地获得。如下所示：

```
void multidns_par(char * name[], ipaddr a[], int n) {
  Coroutine::group group;
  for (int i=0; i<n; ++i) {//仅使用函数库
    group.create(&gethostbyname,
      make_arg(&name[i], &a[i]));
  }
  group.join();
}
```

在合适的同步原语（join）支持下，Coroutine 模式的代码同样十分简洁，并且 Libresync 的工作属于运行时库，具有更低的代价和更好的移植性。

4.3 性能测试

Libresync 最大的特色在于表达清晰，代码简洁，并且性能可以达到理想的水平。我们用 Libresync 编写了一个简单的 Web 服务器 Reweb，只用了百余行代码，并将其与 Apache 进行了性能对比。测试选取最普通的计算环境，详情如表 1 所示。

表 1 性能测试环境

	Web 服务器	测试机器
主机	普通笔记本	普通台式机
网卡	Broadcom NetLink 千兆	NVIDIA 千兆
网络连接	网线直连	网线直连
处理器	Core2 T5500 1.66GHz	Althon X2 1.9GHz
内存	1.5GB	1GB
操作系统	Windows XP	Debian 4.0r1 Linux 2.6.18

测试使用 WebBench 1.5，分别针对大小为 1KB，4KB，…，1 024KB 的文件请求，和并发量为 1，2，4，…，1 024 个的情况进行测试。同时，我们自行制作了一个小程序 TCP-Bench，可以在不同并发规模下对 TCP 协议的实际吞吐率进行测试。这个测试结果（图中虚线）可以近似看做实验环境中 Web 服务器吞吐率的上界。

如图 3 所示，当单次请求数据量超过 64KB 时，Reweb 的吞吐率接近 TCP 在实验环境中的极限吞吐率 480Mbps。我们认为，如果换用服务器级别的配置，Reweb 应能够取得

更好的成绩。此外，我们在测试中发现，Reweb 在运行过程中会占用大量的地址空间，以至于我们目前无法测试并发量为 2 048 情况，然而 Reweb 占用的物理内存数量却在合理范围内。我们认为虚地址空间的浪费现象是由于 Coroutine 模式自身特点决定的，我们可以通过进一步优化 Libresync 以实现更高的并发性，我们也可以换用 64 位系统以解决这个问题。

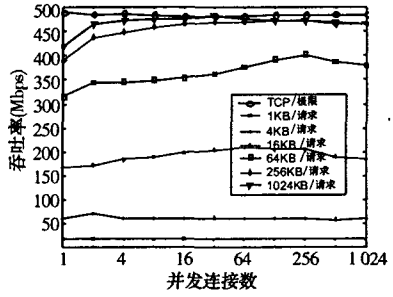

图 3　Reweb 吞吐率测试

测试过程中，始终有一个 CPU 核心处于全负荷状态，另外一个核心只有少量占用。

我们可以从图 4 中看出 Apache 在理想工作状态下，吞吐率能达到 430Mbps，较 Reweb 低一些。Apache 的负载曲线与 Reweb 呈现出了不同的特点：并发请求能够更显著地提高 Apache 的吞吐率。我们猜测这是由于 Apache 在内部对数据进行了缓冲，使得大部分请求无需再从文件读取数据，而 Reweb 并没有做这种工作，完全依赖操作系统的磁盘缓冲。

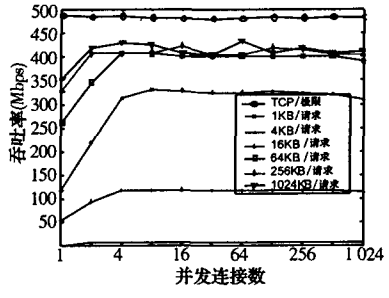

图 4　Apache(v2.2.8)吞吐率测试

在测试中我们还发现，Apache 的服务进程开启了数十个线程，很多时候会将处理器的两个核心全部占用。然而，Apache 对地址空间以及物理内存的占用都比 Reweb 低很多。

Reweb 和 Apache 的性能对比结果绘制在图 5 中。可以看出，对于规模很小(1K)的请求，Reweb 的效率是 Apache 的 2.5 倍左右，我们猜测 Apache 此时的低效率是因为它处理请求时的额外工作(如日志等)比 Reweb 多，请求规模越小则有效负载比重越低，也就是效率越低。对于中小规模(4～64K)的请求，Reweb 效率总体反而比 Apache 低一些，可能这种负载正是 Apache 着力优化的目标。对于中等及更大规模的请求，Reweb 的效率总体上是 Apache 的 1.3 倍左右。

Reweb 仅仅只是一个实验用的小程序，Apache 是一个功能十分完备的 Web 服务器，会有更多的代码去完成 Reweb 没有考虑的工作，这是 Apache 在本次测试中的劣势。另一方面，Reweb 以及 Libresync 还是新生事物，并没有充足的优化工作，而且性能也不是 Libresync 主要的目标；Apache

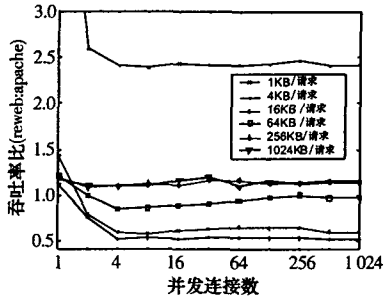

图 5　吞吐率比较(Reweb : Apache)

则是非常成熟的产品，代码经过高度的优化，这是 Apache 在本次测试中的优势。综合考虑这些因素，我们认为 Reweb 和 Libresync 可以判定为在网络 I/O 方面微弱胜出。

5　结束语

Coroutine 与事件在能力上是等价的，但在其他方面各有所长，谁也不能取代对方。应用开发人员应该根据实际需求选择最合适的方法。

Libresync 最大的优势在于既具有非常好的可读性，又具有足够的灵活性，此外它还能够满足大多数的性能需求。我们认为在分布式系统的开发中，Libresync 适用于大多数场合；少数必须采用事件模式的代码，或者遗留的事件模式代码可以很方便地与 Libresync 进行桥接。我们希望 Libresync 能够在更多的分布式系统中发挥基础性作用。

参考文献：

[1] Raymond E S. The Art Of Unix Programming[M]. Addison-Wesley, 2003.

[2] Killian C, Anderson J W, Braud R, et al. Mace: Language Support for Building Distributed Systems[C]// Proc of Programming Languages Design and Implementation, 2007.

[3] Ousterhout J K. Why Threads Are a Bad Idea (for Most Purposes)[C]// Proc of the USENIX Annual Technical Conf, 1996.

[4] Adya A, Howell J, Theimer M, et al. Cooperative Task Management without Manual Stack Management[C]// Proc of the USENIX Annual Technical Conf, 2002.

[5] von Behren R, Condit J, Brewer E. Why Events Are a Bad Idea (for High-Concurrency Servers[C]// Proc of HotOS'03, 2003.

[6] von Behren R, Condit J, Zhou F, et al. Capriccio: Scalable Threads for Internet Services[C]// Proc of the 19th Int'l Symp on Operating Systems, 2003.

[7] Cunningham R, Kohler E. Making Events Less Slippery with eel[C]// Proc of HotOS'05, 2005.

[8] Krohn M, Kohler E, Frans K M. Events Can Make Sense[C]// Proc of the USENIX Annual Technical Conf, 2007.

[9] Lauer H C, Needham R M. On the Duality of Operating System Structures[C]// Proc of the 2nd Int'l Symp on Operating Systems, 1978.

[10] The Boost Library[DB/OL]. [2007-12-10]. http://www.boost.org.