**C++ 和 Scala 语言并行机制的比较**

张有杰

**摘要：**多核技术及并行编程已成为目前提升计算机系统处理能力的主要途径。并行编程语言与编程模型在近些年为人们所关注。越来越多的支持并行编程的语言如 C++、JAVA、Go 、Scala语言等得到了广泛的应用。在众多的语言中选取了 C++ 与 Scala 语言，从并行编程模型、并行机制、执行效率等多个方面对二者进行比较，说明不同语言在并行编程中的优势和劣势，对未来并行编程语言的发展提出了自己的想法。

**关键词：**并行机制，并行编程模型，线程，Actor 模型

一、引言

随着计算机硬件技术的日益成熟以及半导体技术的发展已日趋物理极限，处理器的处理能力已经不能够再简单地通过提高单个处理器的性能而得到提升。同时，在当前互联网时代的海量数据处理等对计算机的处理能力有着更高要求的需求面前，仅仅提升单个处理器的性能也显得杯水车薪。人们为了进一步提升计算机的处理能力，提出了在同一个处理器中加入更多的处理核心的解决方案，即并行处理技术。由于在并行编程中存在着许多需要面对的复杂问题，如多核心间协调访问共享资源以及访问资源时避免死锁等问题，如何对现实世界进行选择更好的并行编程模型及合适的并行编程语言成为了我们需要考虑和解决的问题。

因此，对于 C++ 以及 Scala 这两种目前应用广泛并且有一定代表作用的语言，本文对它们的并行机制作简单介绍，并对其并行机制在性能、开销及可写性等方面的表现进行比较，说明其优势和劣势，对并行编程语言的未来发展提出了自己的观点。

二、并行与并行编程模型

1. 并行

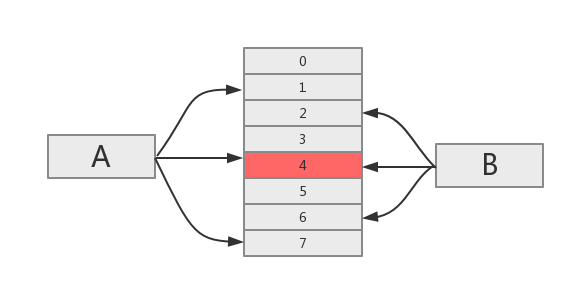
并行指计算机并行运行多个程序或者并行运行同一个程序中的多个部分的能力。通过并行编程，我们可以充分利用处理器多个核心的计算能力以大大提高程序的执行效率。[1]

并行编程主要需要解决的问题是如何划分执行单元以及执行单元之间如何协同工作。因此，每一个并行编程模型可以由并行执行的单元及其相互之间通信的方式所组成的二元组表示。并行模型指定了计算机系统中的执行单元如何协作以完成分配给他们的作业。不同的并行模型使用不同的方式拆分要完成的作业，同时执行的单元之间协作与交互方式也不同。按照执行单元间相互通信的方式分类，并行编程模型可以划分为两种主要的模型：共享内存模型与消息传递模型。[2]

2. 并行编程模型

2.1共享内存模型

在共享内存模型中，并行执行的执行单元之间通过读写共享的内存单元进行同步与通信。共享内存模型可以通过 (线程，共享内存单元) 二元组表示。[3]如图 2.1 所示，假设 A 与 B 分别是计算机中两个并行执行的线程。A 独占 1 号与 7 号内存单元， B 独占 2 号与 6 号内存单元。A 与 B 两个线程通过共享 4 号存储单元，两个线程之间的通信通过读写 4 号存储单元完成。

图 2.1 共享内存模型

可以看出，在不加限制的情况下， A 与 B 可以异步访问 4 号存储单元。当 A 与 B 并行地访问该存储单元时，可能会产生竞争条件等问题。因此我们需要同步机制来协调各个执行单元以避免竞争条件等问题。在共享内存模型下，同步机制主要通过信号量、锁等机制实现。这种并行模型被以 C++ 语言为代表的语言广泛采用。

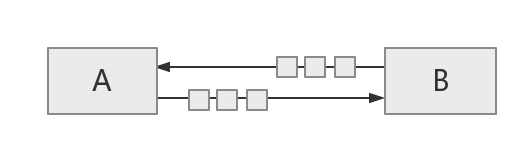
共享内存模型的优点主要是程序中的每一条用户线程直接映射到操作系统提供的一条内核线程，因此程序的执行速度较快。

而其缺点则表现为：1. 程序代码复杂，并且容易出错。 2. 由于每一条用户线程对应一条内核线程，创建线程的开销非常大，限制了程序中最多拥有的线程数，可扩展性较差。

3. 执行过程复杂，不易调试。

2.2 消息传递模型

**在消息传递模型中，并行执行的执行单元之间通过通信信道相互发送消息以同步和通信。每个执行单元向其他单元发送消息，同时接收从其他单元发来的消息。对于外部接收到的消息，每个执行单元对感兴趣的外部消息进行响应处理，忽略不感兴趣的消息。[4]**

 **图 2.2 消息传递模型**

**消息的接收与发送既可以是异步的，即在接收方准备好接收消息之前发送方就可以发送消息，也可以是同步的，即只有在接收方准备好接收消息时发送方才能够发送消息。[5]以 Google 推出的** Go 语言为代表的语言使用同步通信channel来连接进程。以 Scala 为代表的语言则使用 Actor模型进行异步消息传递。Actor 模型则可以使用 (actor，消息)

二元组进行表示。

消息传递模型的优点在于：1. 将用户程序的执行模块与共享资源抽象为可以收发消息的对象，在面向对象编程的思想中使得并行编程更为直观。2. 多个执行单元，如 Actor 模型中的 actor，可以复用同一条线程。这使得每一个执行单元更为轻量级，意味着其可扩展性更好。

而其缺点则表现在：1. 消息传递的方式效率较低，一定程度上影响了程序的并行性能。2. 目前消息传递模型的库与解决方案仍不成熟，仍然存在很多问题。

三、C++ 与 Scala 介绍

1. C++ 介绍

C++ 是 Bjarne Stroustrup 在 C 语言的基础上继承及发展得到的编译型语言。作为一种通用的程序设计语言， C++ 支持面向过程、基于对象和面向对象三种编程模式。即，C++ 可以进行 C 语言的过程化程序设计，也可以进行以抽象数据类型为特点的基于对象的程序设计，还可以进行以继承和多态为特点的面向对象的程序设计。[6]

在 C++ 11 之前的标准中，C++ 程序员只能通过使用 Windows API 中对多线程的支持以完成并行编程。而 Windows API 的调用是十分繁琐与复杂的，并且其对多线程的支持并不能够更好地开发多核心处理器的并行性。同时，使用 Windows API 使得编写后的多线程程序只能够在 Windows 平台下运行，阻碍了程序在不同平台之间的可移植性。在 2011 年新出版的 C++ 11 标准中，C++ 标准协会引入了 thread 等头文件，内容包括了管理线程、保护共享数据、线程间的同步操作、低级原子操作等以支持并行编程，使得 C++程序员从语言层面就可以完成并行编程，提高了多线程程序的可移植性。

2. Scala 介绍

Scala 是一门运行在 Java 虚拟机(JVM) 上的编译型语言，它整合了面向对象编程及函数式编程的特性。作为一门纯面向对象语言，Scala 中的每个值都是对象。Scala 通过子类继承以及灵活的混入机制完成类抽象机制的扩展。作为一门函数式语言，Scala 提供了十分轻量级的语法以定义匿名函数、支持高级函数以及多层函数的嵌套等。[7]

在对于并行编程的支持上，Scala 使用 Actor 模型作为其并行模型，并以 Akka 作为其默认的 Actor 模型实现。Actor是类似线程的实体，通过邮箱收发消息以互相协调工作。Actor 可以复用线程，因此可以在程序中可以使用数百万个Actor,而线程只能创建数千个。基于 Actor 模型的 Scala语言，简化了并行编程，更好地开发了多核 CPU的并行处理能力。

四、C++ 与 Scala 的并行机制及其比较分析

1. C++ 的并行机制

C++ 使用了共享内存模型作为其并行编程模型，以线程为其执行单元，以锁机制同步线程。公共的数据结构所在的内存空间由多个线程之间共享。由于这种共享所支持的对内存单元的读写是异步的，因此需要额外的同步机制控制线程对共享内存空间的访问，消除异步访问可能带来的风险。在这种模型下，多个线程之间没有直接的通信，所有的通信通过共享内存空间的读写隐式地完成。

以图 3.1 为例 。假设我们使用多个并行执行的线程产生数据，同时需要统计所产生数据的数量。我们使用共享变量 count 记录产生数据的总量。每当一个线程产生数据后，它将其产生的数据的数量写入共享变量 count。同时，每个线程都可以通过访问共享变量 count 来获取当前系统产生数据的总量。如果每个线程都可以异步访问并更改共享变量 count，count最终的结果可能是错误的。

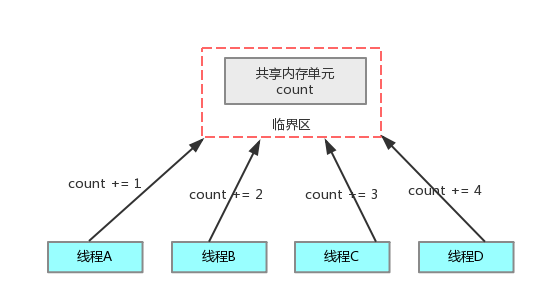


图3.1 C++ 共享内存模型举例

在 Windows 10 系统中，使用 Visual Studio 2017编写附录中所示的

[**<tryThread.cpp>**](#程序1) 程序进行对上述过程进行简单地模拟：

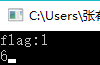
  

图3.2 实验结果

该程序并没有严格模拟上述例子，只是意图说明当多个线程同一时刻异步访问并更改共享内存单元时可能造成错误。可以看出程序几次执行的结果都不相同。值得注意的是，在第三次实验中，count 的值是正确的。在多线程程序的编写中，可能在测试阶段程序运行的结果都是正常的，但是最终程序正式发布运行后由于偶尔出现的并行访问情况导致意想不到的错误。所以在多线程序的编写时，应该仔细设计出共享变量访问的同步机制，以保证程序的正确性。修改上述程序中的 add 函数，即通过使用互斥锁的方式进行线程同步的过程如下：

void add(int num) {

while (flag != 1)//等到所有线程创建完之后一起开始相加

;

myLock.lock();//互斥锁

count += num;

myLock.unlock();

}

在使用互斥锁后，每个线程想要操作 count 所在的内存单元，首先需要获得互斥锁 myLock。如果别的线程正持有互斥锁，线程则需要挂起等待。并且在修改完 count 之后，线程需要释放互斥锁。通过这样的机制，程序避免了多个线程同时操作共享变量 count，保证了程序的正确性。在使用锁以同步线程时，我们还需要避免死锁等问题。

因此，使用线程和锁进行并行编程时，代码通常远比上例中的代码复杂。并且当代码较为复杂时，由于线程的调度具有一定的随机性，程序出现问题之后不易调试。另外，线程是一种有限的资源。以 32 位机为例，其内存空间一共有 4 GB。由于每个线程都需要一个额外的栈空间，在很多操作系统上通常为 1MB，那么一个程序在不存储程序代码及静态数据的情况下最多只能支持 4096 个线程。同时，每次 CPU 在不同的线程间进行上下文的切换时同样要消耗时间及空间资源。这一定程度上限制了并行程序的处理性能及其扩展性。

2. Scala 的并行机制

Scala 使用了消息传递模型作为其并行编程模型。在Scala 中消息传递模型的具体实现是 Actor 模型。该模型通过 Scala 中的 Akka 库实现。在 Actor 模型中，一切都是 actor。也就是说，不仅仅用户程序的执行单元被看作是 actor ，每一个共享的资源也被看作是 actor。所有的actor之间的通信都通过传递消息实现。Actor 将收到的交流消息映射为由发送给其他actor的有限消息集合、新的行为（比如发送消息给下一个actor），新创建的有限个 actor组成的三元组，如图 3.3 所示。[8]

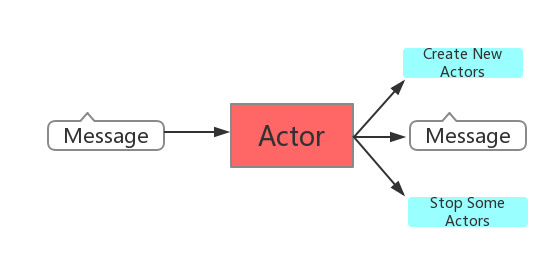


图 3.3 Actor 映射

Actor 模型是事件驱动及异步的。它是事件驱动的。这表现在actor响应事件而不是简单的睡眠或者被唤醒，这里的事件指与其他 actor之间的相互作用，即消息。当actor处理完当前消息并且没有新的消息时会进入消极状态，而当收到感兴趣的消息时它们立刻变得活跃起来。它是异步的。这表现在每个 actor 都有一个存储未处理消息的邮箱。每当收到一个新的消息时，如果actor仍在处理之前的消息，则该消息存在邮箱中。而当actor处理完当前消息并开始等待新消息时，它会首先查看自己的邮箱。不合适的类型的消息会被忽略，第一条合适类型的消息被处理。[9]

以图 3.4 为例，我们使用 Actor 模型重复之前的实验。假设有多个 actor 产生数据，一个 actor负责统计产生数据的总量。每个负责产生数据的 actor 产生数据后，将其产生数据的数量以消息的形式发往统计数据 actor。发送消息无需同步。

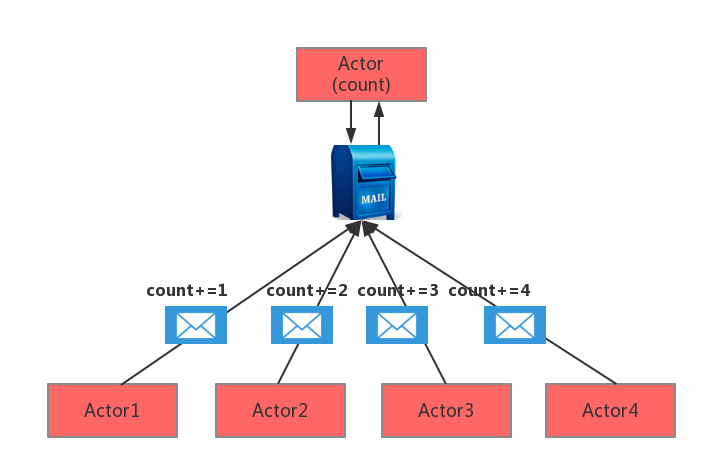


图 3.4 Scala Actor 模型举例

在 Windows 10 系统中，使用 IntelliJ IDEA (JDK版本：1.8.0 Scala版本:2.12.2)编写附录中 [**<AkkaTry.scala>**](#程序2)所示程序模拟上述过程。与之前实验相同，为了模拟多个 actor 同时修改 count 的情况，所有的 actor 在flag = 1时一起对 count 更新。实验结果如图 3.5 所示。由于每一个更新 count 的请求都作为消息存储在 Counter 的邮箱中，Counter 对其邮箱中每一条消息逐一处理，避免了同时修改 count 导致的数据一致性的破坏，使得正确地对 count 进行修改。



图 3.5 实验结果

在 Actor 模型中，程序中完成业务逻辑的单元与数据模型单元都被抽象为了 actor。这体现出了面向对象编程以及 Actor 模型的优势。我们可以直观地将 actor 看作是互相沟通的人，使得在并行编程中我们更加专注于业务逻辑的描述，而不是底层数据的访问控制。

3. 执行效率比较

通过使用 C++ 语言和 Scala 语言编写定积分的方法计算 π 的程序来进行实验以分析使用两种语言开发出的并行程序性能。以下实验在 Windows 10 系统、CPU:Intel I5 四处理核心的电脑上进行。

使用 C++ 语言(IDE：Visual Studio 2017)编写附录中 [<computePi.cpp>](#程序3) 所示的程序进行实验。使用 Scala 语言(IDE: IntelliJ IDEA 2017.1.3 JDK版本：1.8.0 Scala版本:2.12.2 )编写附录中 [**<PI.scala>**](#程序4) 所示的程序进行实验[10]。两个程序所使用的算法及实现逻辑基本相同。

在使用 C++ 语言编写的程序中，我们将积分段分为 20 段，分配给 20 个线程进行处理。每个线程在分配给它的积分段中求得相应的积分值，计算完成后将结果在互斥锁同步的情况下写入 sum 变量。在所有的线程都执行结束后，使用 sum 变量求得最终的 π 值。

在使用 Scala 语言编写的程序中，我们同样将积分段分为 20 段，分配给 20 个 actor 进行处理。处理过程同上。区别在于，在 Scala 中任务分配以消息的形式发送给 actor ，每个 actor 处理的结果也以消息的形式发回给负责统计结果的 actor；而在 C++ 中任务分配以函数参数的形式发送给线程，每个线程处理的结果在互斥锁保证同步的前提下写入结果变量中。

两段程序的执行结果如图 3.5 与图 3.6 所示：

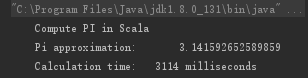


图 3.5 Scala 程序执行结果

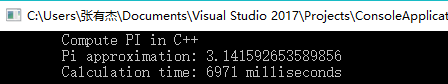


图 3.6 C++ 程序执行结果

结合附录中程序与程序执行结果，总结得到实验结果如下表所列：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 可读性与可写性 | 空间 | 编译时间 | 执行时间 |
| C++ | 基于 C++ 11标准的多线程程序，可读性和可写性较好。但是当程序逻辑较为复杂且需要多个线程协调工作时，同步机制不易设计，程序不易理解。 | 0.9 MB | 2.1 s | 6971 ms |
| Scala | 结合函数式编程及面向对象编程特性，所以可读性与可写性较好。基于 actor 模型的实现较为直观，易于理解。 | 约为 800 MB(包括JVM 等运行时环境在内) | 31 s | 3114ms |

4. 二者并行机制比较总结

在完成上述对两种语言的并行机制、并行编程模型、同步机制、通信机制以及执行效率的分析后，总结如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **并行模型** | **并行模型** | **通信机制** | **同步机制** | **空间** | **时间** |
| C++ | 共享内存模型 | 执行单元之间通过读写内存中的公共存储单元来隐式进行通信。 | 同步是显式进行的。通过互斥锁等机制完成。 | 程序所占空间较小 | 编译时间较短，执行时间较长。 |
| Scala | 消息传递模型（actor 模型) | 执行单元之间必须通过显式地发送消息来进行通信。 | 可异步执行。也可由程序员加入同步策略。 | 程序(包括运行时环境)所占空间较大。 | 编译时间较长，执行时间较短。 |

五、结束语

本文分析比较了 C++ 语言和 Scala 语言的并行机制，并编写实例程序说明其并行机制和比较了二者的执行效率。两种语言都从语言层面提供了对并行机制的支持，能够较好地开发多核计算机系统的计算能力。Scala 语言具有较好的可读性与可写性，同时其 actor 模型更为直观，但是编译时间较长并且所占内存空间较大。而 C++ 语言在完成具有复杂逻辑的多线程编程时，可读性与可写性较差，但是编译时间较短并且所占内存空间较小。多核技术及并行编程使我们目前提高计算机系统执行性能的主要途径。因此，在开发过程中，要选择合适的并行编程模型及编程语言。目前支持并行编程的语言如文中比较的 C++ 语言与 Scala 语言都各有优劣，在实际应用中应结合实际情况综合考虑后进行选择。并且，今后的并行编程语言应该能够提供更高层次的抽象，以使得底层内存单元的存在对程序员透明化，使得程序员避免对同步等问题过多的关注，而专注于业务逻辑的实现上。

参考文献:

1. JAVA并行编程 [EB/OL] <http://ifeve.com/java-concurrency-cookbook/>
2. 并行编程模型 [EB/OL] [http://ifeve.com/%E5%B9% B6%E5%8F%91%E7%BC%96%E](http://ifeve.com/%E5%B9%B6%E5%8F%91%E7%BC%96%E7%A8%8B%E6%A8%A1%E5%9E%8B)

[7%A8%8B%E6%A8%A1%E5%9E%8B](http://ifeve.com/%E5%B9%B6%E5%8F%91%E7%BC%96%E7%A8%8B%E6%A8%A1%E5%9E%8B)

1. 共享内存模型 [EB/OL] <https://en.wikipedia.org/wiki/Shared_memory>
2. 并行模型 [EB/OL] <https://stackoverflow.com/questions/4153118/list-of-concurrency-models>)
3. 并行简介 [EB/OL] <http://web.mit.edu/6.005/www/fa14/classes/17-concurrency/>
4. C++ [EB/OL] <https://en.wikipedia.org/wiki/C%2B%2B>
5. Scala [EB/OL] <https://zh.wikipedia.org/wiki/Scala>
6. ACTOR-BASED MODEL FOR CONCURRENT PROGRAMMING [EB/OL] [http://ds.cs.ut.ee/](http://ds.cs.ut.ee/courses/previous/seminar-materials/SanderS6najalg-slides.pdf)

[courses/previous/seminar-materials/SanderS6najalg-slides.pdf](http://ds.cs.ut.ee/courses/previous/seminar-materials/SanderS6najalg-slides.pdf)

1. G Agha; Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism [C] 1985
2. Getting-started-first-scala [EB/OL] [http://doc.akka.io/docs/akka/2.0.1/](http://doc.akka.io/docs/akka/2.0.1/intro/getting-started-first-scala.html)

[intro/getting-started-first-scala.html](http://doc.akka.io/docs/akka/2.0.1/intro/getting-started-first-scala.html)

附录：

1. <tryThread.cpp>

#include<iostream>

#include<thread>

int count = 0;

int flag = 0;//标志所有线程是否都建立成功

void add(int num) {

while (flag != 1)//等到所有线程创建完之后一起开始相加

;

count += num;

}

int main() {

std::thread \* worker[4];

for (int i = 0; i < 4; ++i) {//建立线程

worker[i] = new std::thread(add,i+1);

}

std::cout << "flag:";

std::cin >> flag;//所有线程建立完毕

for (int i = 0; i < 4; ++i) {//等待所有线程结束

worker[i]->join();

}

std::cout << count;//输出count值

std::cin >> count;

return 0;

}

2. <AkkTry.scala>

import akka.actor.{Actor, ActorRef,ActorSystem,Props,ActorLogging}

object Flag{

var flag = 0

}

object Counter {

def props = Props[Counter]

final case class show()

final case class add(num : Int)

}

class Counter extends Actor {

import Counter.\_

var count : Int = 0

def receive = {

case show() =>

println("Current count:" + count)

case add(num) =>

count += num

}

}

object Worker {

def props(counter : ActorRef) : Props = Props(new Worker(counter))

final case class work(num : Int)

final case class produce()

}

class Worker (counter : ActorRef)extends Actor {

import Worker.\_

import Counter.\_

import Flag.\_

var numEach : Int = 0//每次产生数据的数量

def receive = {

case work(num) => numEach = num

case produce() =>

while(flag == 0){}//等待 flag 变为 1

counter ! add(numEach)

}

}

object AkkaTry extends App{

import Worker.\_

import Counter.\_

import Flag.\_

val system : ActorSystem = ActorSystem("MyActorSystem")

try {

val counter : ActorRef = system.actorOf(Counter.props,"Counter")

val worker1 : ActorRef = system.actorOf(Worker.props(counter),"Worker1")

val worker2 : ActorRef = system.actorOf(Worker.props(counter),"Worker2")

val worker3 : ActorRef = system.actorOf(Worker.props(counter),"Worker3")

val worker4 : ActorRef = system.actorOf(Worker.props(counter),"Worker4")

//设定每次产出的数据量

worker1 ! work(1)

worker2 ! work(2)

worker3 ! work(3)

worker4 ! work(4)

//生产

worker1 ! produce()

worker2 ! produce()

worker3 ! produce()

worker4 ! produce()

flag = 1

scala.io.StdIn.readLine()

counter ! show()

println(">>> Press ENTER to exit <<<")

scala.io.StdIn.readLine()

}

finally{

system.terminate()

}

}

3. <PI.scala>

import akka.actor.\_

import scala.concurrent.duration.\_

import akka.routing.RoundRobinPool

sealed trait PiMessage

case object Calculate extends PiMessage

case class Work(start: Int, nrOfElements: Int) extends PiMessage

case class Result(value: Double) extends PiMessage

case class PiApproximation(pi: Double, duration: Duration)

object PI extends App {

calculate(nrOfWorkers = 4, nrOfElements = 10000, nrOfMessages = 10000)

def calculate(nrOfWorkers: Int, nrOfElements: Int, nrOfMessages: Int) {

val system = ActorSystem("PiSystem")

val listener = system.actorOf(Props[Listener], name = "listener")

val master = system.actorOf(Props(new Master(

nrOfWorkers, nrOfMessages, nrOfElements, listener)),

name = "master")

master ! Calculate

}

}

//计算PI值

class Worker extends Actor {

def calculatePiFor(start: Int, nrOfElements: Int): Double = {

var acc = 0.0

for (i ← start until (start + nrOfElements))

acc += 4.0 \* (1 - (i % 2) \* 2) / (2 \* i + 1)

acc

}

def receive = {

case Work(start, nrOfElements) ⇒

sender ! Result(calculatePiFor(start, nrOfElements)) // perform the work

}

}

//主Actor

class Master(nrOfWorkers: Int, nrOfMessages: Int, nrOfElements: Int, listener: ActorRef)

extends Actor {

var pi: Double = \_

var nrOfResults: Int = \_

val start: Long = System.currentTimeMillis

//设置nrOfWorkers个Workder计算PI值

val workerRouter = context.actorOf(

Props[Worker].withRouter(RoundRobinPool(nrOfWorkers)), name = "workerRouter")

def receive = {

case Calculate ⇒

for (i ← 0 until nrOfMessages) workerRouter ! Work(i \* nrOfElements, nrOfElements)

case Result(value) ⇒

pi += value

nrOfResults += 1

if (nrOfResults == nrOfMessages) {

listener ! PiApproximation(pi, duration = (System.currentTimeMillis - start).millis)

context.stop(self)

}

}

}

//监听作用的Actor，记录打印

class Listener extends Actor {

def receive = {

case PiApproximation(pi, duration) ⇒

println("\n\tPi approximation: \t\t%s\n\tCalculation time: \t%s"

.format(pi, duration))

context.system.terminate()

}

}

4. <computePi.cpp>

#include <iostream>

#include <thread>

#include <mutex>

#include <ctime>

#include <iomanip>

long num\_step = 1000000000;

long double step = 1.0 / (long double)num\_step;

std::mutex myLock;

long double sum = 0.0;

int FindPi(long long t1)

{

long long t2 = t1 + 50000000;

long double tmp = 0;

for (t1; t1<t2; t1++)

{

long double x;

x = (t1 + 0.5)\*step;

tmp += 4.0 / (1.0 + x\*x);

}

myLock.lock();

sum += tmp;

myLock.unlock();

return 0;

}

int main()

{

time\_t start = clock();

std::thread \* thread[20];

long long st[20];

long long i;

for (i = 0; i<20; i++)

{

st[i] = i \* 50000000;

thread[i] = new std::thread(FindPi, st[i]);

thread[i]->join();

}

long double pi = step\*sum;

std::cout << "\tCompute PI in C++"<<std::endl;

std::cout << std::setprecision(16) << "\tPi approximation: " << pi << std::endl;

std::cout <<"\tCalculation time: "<< clock() - start <<" milliseconds"<< std::endl;

return 0;

}