# 并发编程概述

作者：David W. Bustard

翻译：clark huang

## 基本概念

### 1.1 并发编程的本质

#### 1.1.1 显示及隐示并发

#### 1.1.2 进程以及并发程序：基本概念

#### 1.1.3 区分并发，并行，以及分布式的区别

#### 1.1.4 区分并行程序和并行系统之间的差异

#### 1.1.5 非确定性

#### 1.1.6 进程交互

### 1.2 并发编程中的问题

#### 1.2.1 违背共享祛除

#### 1.2.2 死锁

#### 1.2.3 无限延期（饥饿／锁定）

#### 1.2.4 不公平

#### 1.2.5 焦虑等待

#### 1.2.6 瞬时误差

### 1.3 并发编程的特性

#### 1.3.1 安全性

#### 1.3.2 活跃性

### 1.4 并发程序的执行

#### 1.4.1 并发的度量

#### 1.4.2 执行环境

#### 1.4.3 执行模式

#### 1.4.4 进程状态

#### 1.4.5 进程的计划

## 2. 程序结构

### 2.1 开发方式

### 2.2 一个必要的视图

#### 2.2.1 瞬时逻辑

#### 2.2.2 佩特里网

#### 2.2.3 进程模型

#### 2.2.4 有限状态机器

#### 2.2.5 数据流动诊断

### 2.3 设计案例

### 2.4 编码案例

#### 2.4.1 通过共享变量交互

#### 2.4.2 通过传递信息交互

### 2.5 并发编程解析

## 3. 常用应用

### 3.1 实时系统

### 3.2 通用的操作系统

### 3.3 即时系统

### 3.4 数据库系统

## 基本概念

这个部分提供对并发编程的主要概念和关系的介绍和讨论。更详细的解释，以及相关阐述，可以在其他的一些基础的论文中找到。并发的概念同样在很多操作系统的书中有所涉及到。

1.1 并发编程的本质

1.1.1显示及隐示并发

从原则上说，大多数程序对并发的理解是这样的：

* 包含独立的进程或线程（在阻塞，语句，或者表达式层级上）能够并行的执行程序。或者
* 触发设备并行的多线程执行程序

这些大致可以被定义为是隐式并发。然后显式的并发才是有程序设计者可以对并发行为进行操控的点。

通常来说，对于并发就意味着标志性的一串序列，或者按照算法进行的并行的操作。通过这样的行为来提高代码的执行效率。通常的提升点可通过，部分的潜在的优化可以通过对程序的重构实现。

这部分的重点在于对于显式的并发特性的运用。

1.1.2进程以及并发程序：基本概念

一个顺序程序通常被定义了一系列执行的语句，执行这个程序单元叫做线程／进程，一个并行的程序指定两个或者多个顺序程序，将它们同时执行，并且在并行的线程／进程中，在许多语言中，线程或者进程也被用于描述进程的行为。在Ada这门语言中，用task来描述进程／线程。

1.1.3区分并发，并行，以及分布式的区别

从字面上来说，并发通常被用混用在并行以及分布式的概念中，不幸的是，很少有作者能够对这几个概念的关系做一个明确的区分。总而言之，下面对这些概念的定义，得到了大多数人的肯定：

* 并发程序，表示行为同时发生。
* 并行程序，通常是表示多核系统中，程序在不同内核同时执行的情形。
* 分布式程序，指的是一个并行的程序在多个未知的不共享内存的系统中被调用。

因此，可以说并发是一个多并行和分布式的广义的定义。并行和分布式都是并发的子概念，都在特定的并行处理环境下的一种表现形式。

大家都知道，单个线程可以在并发环境中执行程序。但有时候会有假并发的现象。一个假并发的程序通常执行通过和协助进程相互分享控制权，来完成一段程序的执行。（这也是并发模型在编程语言中很基本的一个应用）

1.1.4区分并行程序和并行系统之间的差异

一个并行程序是一个完整的软件一个必不可少的组成部分。如果两个用于交互的程序软件可以同时执行，相互之间没有沟通，则两者的交互是并行的。否则，这两个软件之间一定是通过某种双方同意的协议进行沟通的。通信程序然而却是通常是用并行程序构建的。

1.1.5 不确定性

一个顺序程序标志着这个程序的执行一定是按照程序写好的那样指令执行的。然而一个并行程序却标志着这个程序只有部分的顺序性，也就是说，有些事件的执行，并不能确定一定按照某种循序。这种特性就是我们说的不确定性。不确定性的后果是，有时候虽然我们的执行结果是一致的但是我们的执行过程可能却不是一样的。

程序的级别是作为不确定性还是确定性的都依赖于它们对于当前程序的重要性来判断的。大多数程序都是不确定性的当它们的执行次数在一个很少的数量级的时候，但是它们的不可遇行为决定了所有的程序等级。

1.1.6 进程交互

所有的并发程序必然涉及到线程或者进程的交互，这个发生是因为两个主要的原因：

* 进程间对于共享资源分享概念，例如物理设备或者数据。
* 进程间的数据交换。

在两种情况下，不论是从避免获取资源时候的冲突，或者是建立交互分享数据时候的，进程间的同步执行是有必要的。

进程能够交互通过其中某一种方式：共享数据，或者发送信息给其他人。进程的交互能够被显示在程序中，或者隐式执行。特别说明的是，有管理系统资源的隐式的管理方式。例如，进程活性，或者内存。被用于跑一个程序。

1.1.6.1.资源管理

一个线程想要使用共享的资源必须首先获取资源。通过获取权限获取它，并且当使用完毕后，不需要的时候，需要释放资源。当然一个线程也可以同时获取多个资源。这些资源也可以是同种资源也可以是不同中资源。这些资源可能是静态的共享资源，也可能是当时创建的资源，当程序启动的时候，当程序结束的时候，资源就会被销毁掉。也可能，一些资源也许是用的别的线程分享的资源。例如，可能有很多线程会同时获取一个共享的数据。但是能修改这个数据的一次只能是一个人。

1.1.6.2 建立连接

内部的线程沟通是通过下面的方式进行的：

* 同步（Synchronous）：交换数据的时候，同步进行。
* 异步（ASynchronous）：指的是一个线程提供的共享数据，就放在哪儿了，不会等待接受线程。数据可能会被储存在一个交互缓冲流种一个短暂的时间。并且其他的线程获取数据的顺序和它们到达的顺序是一致的。

1.2 并发编程中的问题

1.2.1违背共享祛除

在并发编程中，一些操作可能获取不到指定的效果。当多个线程操作同一块代码的时候，被称之为临界区或者是临界界面。如果其中的一个线程在这种临界区域的时候，其他所有的线程必须被祛除在外，直到这个线程完成对临界区域的操作为止。当我们构造并发程序的时候，十分有必要识别出共享祛除的状态，以及对其的控制。

大多数的共享祛除使用的一个例子如下：

x := x+1

假设x的初始值为12，现在可能发生两个线程同时对这句话进行操作，导致两个线程获取的x都为13。从而发生了错误。

对于单独的内存区域索引的共享祛除，通常是通过硬件来实现的。因此，如果两个线程想对同一段内存区域写入3和4，其中一个线程的进入必然会对另一个线程的祛除。

1.2.2 死锁

如果一个进程等待着某个资源，但是这个资源可能释放掉，或者他等待的周期是无限大的，这个时候我们说这个线程死锁了。死锁通常涉及到了多个进程，并且非常可能导致程序的中断。死锁常常出现线程交互的时候（两个线程尝试着发送信息给对方同时并且同步的时候就会出现死锁，双方都等待对方回复，但是这个时候双方都进入了block状态，所以谁都不会回复给任何人任何信息，就陷入无限等待中。），死锁更常见于对共享资源的处理上。

以下的四个状态会导致死锁的出现：

1. 线程间存在对共享资源的共享祛除
2. 线程在等待其他人的时候，自己手上必须拿着共享资源（获取资源断断续续）
3. 等待线程中的等待资源，不能够被祛除掉（资源不能够别提前释放）
4. 存在环状的线程链表，并且其中每个线程拥有者一个或者多个资源是其他的线程所急需的。

避免死锁或者从死锁中恢复的手段，基于至少避免上述的其中一种情况。对于死锁的描述的其中比较好的文章是Dijkstra’s Banker的算法。Dijkstra’s 也提出了一个经典的例子用来解释死锁的问题。就是我们所说的哲学家就餐问题。

1.2.3 无限延期（饥饿／锁定）

一个线程被称之为饥饿是指这个线程在等待某个事件，但是这个事件可能永远不会发生。这种情形通常出现在如果通过一个算法执行对一个资源的请求，但是没有给涉及到的线程任何等待的时间。系统化技术用于避免这种问题的方式，是将运算的线程放入一个优先级顺序中，如此等待时间较长的线程会给予较高的优先级。处理线程严格按照它们的延迟顺序是一个煎蛋的解决方式对于大多数的程序环境来说都是适用的。

1.2.4 不公平

通常来说（但是不是所有情况），大家认为并发中进程的竞争状态都是相同的，所以我们应当确保保证线程的公正性。公正性在并发系统中，被认为是涉及以及系统级别的实现需要考虑的问题。对于一个设计者来说，这是一条需要遵守的守则。任何不公正的现象都可能导致饥饿的产生导致程序的不正确。对于一个系统实现者来说同样也是一条守则。大多数的编程语言都是不考虑公平性的。而是吧这个问题留给了编译器实现者或者是开发运行时程序的软件开发者。

总的来说，相同的选择行为反复重复出现在并发程序中，我们一定不能忽视任何的无线延期的出现。这是一个对于不公正常有的现象。正常的情形应当是所有的线程都拥有同样的机会。

1.2.5 忙等待

不论并行程序的执行环境是怎样的。如果其中任何一个线程执行一个循环的时候是需要等待程序的状态的的话，都是不被接受的。这种情形称之为忙等待。其中的状态变量涉及到创建一个旋转锁（自旋锁最多被一个线程持有，如果一个执行线程试图获取一个被争用的自旋锁，那么该线程就会进行忙循环直到该锁可用），这不是他本身发生了什么错误，而是他浪费了线程的性能，这样会导致程序的性能不满足要求。理想状态是，这个程序应当被悬挂起来，直到所有的状态全部满足之后，才能继续进行。

1.2.6 瞬时误差

由于不确定性，可能在并行程序中出现瞬时误差的情形，指的是一个错误可能或者也可能不出现，根据的是程序执行的轨迹在特定的激活程序的过程中。造成瞬时误差的原因是很难发现的，因为这个发送错误的情形是十分难以复原的。因此，设计并行程序的其中一个手段或能力就是保证程序的正确性，并且程序的执行是按照时间依赖的。

1.3 并发特性

并发程序的一些要求可以被定义成一些必须有的特性。

1.3.1 安全性

线程安全的特性规范了一个程序什么能做什么不能做，下面的一些情况是必须规避的。

* 共享祛除：不能有多余一个线程同时处理临界区域
* 无死锁：没有线程回去等待一个永远不会执行的事件
* 正确性：如果一个程序执行完毕，那么她的输出必须是要求一定的。

安全的特性被表述为不变的计算。这是一种状态表示程序不论在哪种情况下都能够正确不变的执行。不变性必须保持正确对任何初始状态或者交互状态。

1.3.2 活跃度

活跃度特性标志着一个程序应该做什么，在程序计算的过程中会发生什么，例如：

* 公平性：一个可执行的程序会被执行
* 可靠的交互：信息发送从一个线程到另一个线程会被接收
* 可靠的正确性：如果程序执行结束，输出会输出必定要求的输出

活跃性特性能够被表述成为一系列活跃原理，这些活跃性会被证明活跃特性。但是活跃特性的证明是基于安全性的基础上的。

1.4 并发程序的执行

1.4.1 度量并发

并发行为可以通过多种方式度量，在实际运用中，度量用于初步度量程序的行为。这些度量的名字下面会给出，当然对于名字也没有一致的建议。在实际中，对于并发的粒度通过以下的一些方式进行赋予定义的含义：

* ”并发行为单元“ 适用于描述程序行为的基本定义。他可能是表达式中的一个元素，他也可能是一个程序语句，但是更常见的他是一个程序阻塞。
* “并发级别”指的是在程序的执行过程中同时并发的线程数量
* “并发的规模”：执行一个程序的生命周期。有一个天花板效应在初始化并发行为的时候，所以理想化的情况是，让这个初始化的生命周期尽可能的短，以至于可以忽略这个天花板效应。
* “并发的粒度”：指程序执行交互的交流时间。这个并发力度应该相对较大，如果涉及到分布式的程序的执行的话。
  + 1. 执行环境

程序涉及到大规模的并发行为的时候通常被执行在一个单核处理器的电脑上，多线程分享单核内存。这个被称之为多任务编程，多核编程指的是并行程序运行在多核处理器的电脑上共享主内存。

联机电脑使用多个不同的主内存，它们的执行称之为分布式处理。紧密耦合的联机电脑拥有快速并且可靠的点对点的内部处理器链接。解藕的不同系统间通信，跨网络的通信不仅慢并且还不值得信赖。联机的电脑组件可能在相互相隔临近，这些被称之为工作站点。

小规模的并发程序，通常指的是向量处理器，对一些数据进行同样的操作同时进行的程序行为。这也叫做同步处理。数据流和减少机器数量用于不同的操作于不同的数据的同时执行。

1.4.3 执行模式

大多数的并发程序，启动时候，都是单线程的，然后开始被划分为多线程程序在程序执行的某个环节中，这些孵化的线程可能被单独激活执行程序，或者以几何的形式激活。这些激活的新线程也可以以同样的方式进行扩展。下面有两种主要的执行模型：

* 使用孵化线程，当它们被激活的时候，它们会按照自己的运行法则进行触发执行
* 触发的进程，派生出多个进程，并且折线进程完成任务后，又连接到单主线程。

大多数编程语言中都支持fork-and-join（叉接） 模型

1.4.4 线程状态

线程存在以下三种状态之一：

* 清醒状态，表示当前线程能够完成执行任务
* 睡眠（活着说阻塞状态），当前线程被挂起来了，等待一个特殊的事件唤醒它。
* 终结状态。意味着当前线程的执行已经完成了。

当然清醒状态的线程可以进一步被分为正在执行的状态和准备执行的状态（只要处理器分配片段时间了）

1.4.5 进程计划

在特殊情形下，一个并发程序可能直接派在单独的硬件上。更常见的情况是，程序会在在基于硬件的抽象接口上执行程序的。这个被称之为系统内核。系统内核的其中一个组件就是计划调度器。这个计划调度器负责分配处理器给不同线程／进程的时间。这也是当线程数量和处理器数量不匹配时候的解决方案，通过合理的分配大家的处理时间。当然在分布是系统中，计划调度器本身也是分布式的。

在一些并发编程语言中，进程能够被指定到特定的处理器进行处理。然而更常见的是让计划调度器隐式的去分配。

线程通常的执行是有优先级的。计划调度器的其中一个目标是保证所有的正在运行的线程拥有一个较低的优先级相较于那些已经在等待状态的进程。优先级可以被显示的指定，也能够被隐式的指定被计划调度器。

编译并发程序，大多数情况下是去调用内核完成的，这可能触发计划调度器的操作。任何对内核的访问都可能导致当前程序被挂起，而去执行其他的线程。当，处理能力很低的时候，对所有的线程进行处理器的时间分割让线程都到达稳定状态也是极好的。这个通常的实现是在系统始终的协助下打断某一个线程以一个连续的片段实现的。

## 2. 程序结构

大部分方面并发编程被其他的课程模块涵盖到了，这个部分提供对那些部分知识进行以下介绍，然后在特定的添加一些补充的信息。

2.1 开发方法

并发和顺序编程的开发基本上是同样的方式进行的。有以下三个方面需要注意：需求分析，软件设计，软件编写。在实际运用中，这三个部分并没有做一个很明确的区分。

2.2 需求视角

在需求这个层级上，程序的属性相对于它们的结构更被注重，这个部分会去指定一些技术用于描述并发，从数学的角度来讲，然而有一些重要的特性不能被表达出来。包括对于性能的要求比如对一个时间的反应时间。

2.2.1 时间逻辑

时间逻辑用于描述并发或者是顺序程序。他是传统逻辑架上时间的分析分析方式。

每一段程序的执行都会产生一系列的计算，状态和事件。这个被称之为轨迹。

时间逻辑用于特定化一段轨迹。两个时间逻辑的常量经常被使用。

* 线性时态逻辑 ：被认为是单独的轨迹
* 分支时间时序逻辑： 认为可能的轨迹可能形成一个树状的结构包含着信息和状态以及不确定的选择。

不管是哪种一系列的操作行为都会形成一系列的指令和行为和状态在时间逻辑中。

2.2.2佩特里网

佩里提供了一种对并发系统建立信息流动模型的方式。对于一张网状，上面有几个变量。在一个状态／事件的网状结构中，时间相互连接起来是通过状态的。每一个事件都有一系列的输入输出状态。一个输出状态可能服务于多个输入状态。应此形成了一个网状结构。一个特定事件的发生，当所有指定的输入状态满足条件后，事件也可能同时发生。这样的事件激活了他的输出状态。同理递归对其他的事件产生影响。对于简单的系统的话，网状能够时分的有效，在一个图形化的形式中。

2.2.3 处理模式

对于并发程序的行为关系到进程的通信，每一个进程都关系到一系列事件，总体来说有三种事件：

* 一个存在于进程内部的事件
* 单个线程与多个线程的同步
* 特殊系统环境下的单个线程与多个线程的同步

进程将不同的执行事件按照指定的顺序执行下去。通常来说，进程被定义成递归式的。例如：时钟的行为可以被描述成无限次的tick和tock事件，因此：

clock：（tick；tock；clock）

精准的符号以及一系列的指令用于描述进程都是相互极其不同的。但是每一个都支持基本的对系统的定制化描述

2.2.4 有限状态机器

一个有限状态机器可被用于描述进程的一些行为。并发系统能够用于描述交互的FSM（finite state machines）

FSM通过一系列的状态量和转变量定义。一个状态转变量描述图，描述了FSM的一种工作状态。

2.2.5 数据流程图

DFD（data flow diagram）一张用于描述数据流转和存储的图形。这样的场景对于描述并发程序时分有意义

2.3 设计视角

Gomma的模型主要用于考虑实时系统，他总结出了可适用于下面的五个方面：

* 结构分析和设计实时系统
* 海军研究实验室
* 面向对象编程
* 杰克森实时系统
* 设计倾向实时系统

2.4 编码视角

2.4.1 通过分享共享变量完成交互

这种情形下的共享祛除是必要的。具体做法上面已经给出下面就不再做具体的介绍了。

2.4.1.1 状态共享变量

对临界区的进入通常是被一段内存区或者说是一个共享变量标志当前临界区是否正在被使用，这个共享状态本身也是需要祛除状体的。这个可以通过原子性的操作完成，包括设置变量设值的过程不被打断。大多数电脑提供这样的用法。

状态行为，就是简单的理解和实现。然而这个也是不安全并且是十分浪费进程效率的，因为他鼓励了忙等待。

2.4.1.2 信号量

信号量可以被描述成是前面说的状态共享变量的高级抽象。进入活着是退出临界区通过P和V操作进行控制。这些操作可以被解释为“如果有必要刻意等待”和“发送信号”（这两个字母实际是表示通过或者是释放）。一些信号量被定义给完成的进程权限按照进入的顺序。最初的定义，然而，却没有规定任何的顺序，较少的要求给予了实现开发者更多的灵活性，但是却要求程序设计者自己找到方法去处理等待的线程队列。

信号量是最常见的用于共享祛除的方式，然而它也是不安全的并且太严格了应为它们不能够被直接检测。

2.4.1.3 有状态的临界区

有状态的临界区被提出用于提高对信号量的缺点的认识。在临界区的变量被叫做资源。只有完成对资源获取的认证和资源状态的全部满足才能够对资源进行获取。

这个技术虽然知识实验上被证实了，但是对这个的讨论是十分多的。这个对现代编程语言的影响是十分巨大的。

2.4.1.4 监控器

监控进程对临界区有状态资源的获取和释放。

2.4.2 通信交互

相对共享变量的交互方式，通信的交互方式更流行。交互可以是同步的也可以是异步的，交互的线程相互是可以识别的。但是这个并没有在编程语言中体现，应为这会使线程库极度复杂。取而代之的是：

* 进程可以命名一个共享的通信通道
* 通信可以系统性的完成，只是，发送的线程指定接收的线程。例如在，Ada中，触发线程调用过一个进入口，使得他和接收方能够通信。如果多个线程相同一个接受方发送信息的时候，就需要等待一个队列中。这种基础的交换信息的场景叫做约会。并发C也支持约会的概念，但是它们之间的线程沟通并不是同步的，而是异步的。

2.5 并发编程计量

## 3. 常用应用

3.1 实时通信系统

实时系统的六个特点：

* 一个实时交互系统可能是一个大型的软件／硬件系统的一部分，这个被称为为嵌入式程序
* 一个实时交互系统一般情况下和外部环境交互，交互对象可能是人类，但是也可能是其他人
* 所有的实时交互系统都有时间限制，意味着它们必须在指定时间内完成指定的行为
* 一个实时交互系统会根据输入作出控制决定
* 通常是基于事件的
* 通常是基于顺序反馈，控制或者是发生事件的

在设计实时系统的时候，也会用一些线程去对一些资源进行管理。

3.2 操作系统

常见的操作系统会管理大量的资源并提供给一个或多个用户。并且也是实时交互系统的一种，进程通常对用户的行为进行反馈。基本上没一个用户对应的就是一个进程，并且这个进程完成一些常见的行为。

资源管理作为操作系统中主要的一环，在设备昂贵的年代，对这些资源性能的控制十分容易造成死锁。现在来说，设备便宜了所以分享的地方需要的就少了，程序好些多了。确实对于很多的操作系统来说都是分布式的系统，发生死锁，活锁等问题的概率变的低了很多。

3.3 仿真系统

在实时交互系统，活着是操作系统用于实际生产前，必然是会进行一些模拟的操作。这样的系统就是仿真系统。

3.4 数据库系统

数据哭是一个处理数据集合的工具。数据库通常来说是多线程的，因为可能多用户对数据库的同时操作，一个事务指的是某个线程对某段数据的处理，并且将数据从一个状态达到另一个状态。多个事务的同步执行，其实和按照一个执行的是一样的。

对事务的共享祛除是保证数据一致性的保证。然而，为了保证用户的反馈在接受的级别，通常有必要实现读和写锁在小的数据单元中。数据单元的大小其实就是我们的说的锁的粒度。有时候可能一个线程不知需要获得某一个数据，可能获取数据是多个方面的，所以对于死锁的预防也是需要的。（通常情况涉及到多资源获取的时候十分容易产生死锁）

通常情况下，是不会对死锁做任何的预防的。相反，是通过终止或者是多个锁着的事务，并且回滚做的一些改变。

另一个方式（优化，计划），不涉及到锁，是第一个获取锁的让改变锁。那个时候，同步的行为已经得到了commit。所以冲突不会发生。其他的事务也会遭到拒绝，数据库也得到了更新。