转 生产者/消费者模式

分类： 设计模式 2011-09-08 09:42 16238人阅读 评论(3) 收藏 举报

socket 存储 通讯 语言 编程 shell

[0]：概述

今天打算来介绍一下"生产者／消费者模式"，这玩意儿在很多开发领域都能派上用场。由于该模式很重要，打算分几个帖子来介绍。今天这个帖子先来扫盲一把。如果你对这个模式已经比较了解，请跳过本扫盲帖，直接看下一个帖子（关于该模式的具体应用）。

看到这里，可能有同学心中犯嘀咕了：在四人帮（GOF）的23种模式里面似乎没听说过这种嘛！其实GOF那经典的23种模式主要是基于OO的（从书名《Design Patterns: Elements of Reusable Object-Oriented Software》就可以看出来）。而Pattern实际上即可以是OO的Pattern，也可以是非OO的Pattern的。

★简介

言归正传！在实际的软件开发过程中，经常会碰到如下场景：某个模块负责产生数据，这些数据由另一个模块来负责处理（此处的模块是广义的，可以是类、函数、线程、进程等）。产生数据的模块，就形象地称为生产者；而处理数据的模块，就称为消费者。

单单抽象出生产者和消费者，还够不上是生产者／消费者模式。该模式还需要有一个缓冲区处于生产者和消费者之间，作为一个中介。生产者把数据放入缓冲区，而消费者从缓冲区取出数据。大概的结构如下图。

为了不至于太抽象，我们举一个寄信的例子（虽说这年头寄信已经不时兴，但这个例子还是比较贴切的）。假设你要寄一封平信，大致过程如下：

1、你把信写好——相当于生产者制造数据

2、你把信放入邮筒——相当于生产者把数据放入缓冲区

3、邮递员把信从邮筒取出——相当于消费者把数据取出缓冲区

4、邮递员把信拿去邮局做相应的处理——相当于消费者处理数据

★优点

可能有同学会问了：这个缓冲区有什么用捏？为什么不让生产者直接调用消费者的某个函数，直接把数据传递过去？搞出这么一个缓冲区作甚？

其实这里面是大有讲究的，大概有如下一些好处。

◇解耦

假设生产者和消费者分别是两个类。如果让生产者直接调用消费者的某个方法，那么生产者对于消费者就会产

生依赖（也就是耦合）。将来如果消费者的代码发生变化，可能会影响到生产者。而如果两者都依赖于某个缓冲区，两者之间不直接依赖，耦合也就相应降低了。

接着上述的例子，如果不使用邮筒（也就是缓冲区），你必须得把信直接交给邮递员。有同学会说，直接给邮递员不是挺简单的嘛？其实不简单，你必须得认识谁是邮递员，才能把信给他（光凭身上穿的制服，万一有人假冒，就惨了）。这就产生和你和邮递员之间的依赖（相当于生产者和消费者的强耦合）。万一哪天邮递员换人了，你还要重新认识一下（相当于消费者变化导致修改生产者代码）。而邮筒相对来说比较固定，你依赖它的成本就比较低（相当于和缓冲区之间的弱耦合）。

◇支持并发（concurrency）

生产者直接调用消费者的某个方法，还有另一个弊端。由于函数调用是同步的（或者叫阻塞的），在消费者的方法没有返回之前，生产者只好一直等在那边。万一消费者处理数据很慢，生产者就会白白糟蹋大好时光。

使用了生产者／消费者模式之后，生产者和消费者可以是两个独立的并发主体（常见并发类型有进程和线程两种，后面的帖子会讲两种并发类型下的应用）。生产者把制造出来的数据往缓冲区一丢，就可以再去生产下一个数据。基本上不用依赖消费者的处理速度。

其实当初这个模式，主要就是用来处理并发问题的。

从寄信的例子来看。如果没有邮筒，你得拿着信傻站在路口等邮递员过来收（相当于生产者阻塞）；又或者邮递员得挨家挨户问，谁要寄信（相当于消费者轮询）。不管是哪种方法，都挺土的。

◇支持忙闲不均

缓冲区还有另一个好处。如果制造数据的速度时快时慢，缓冲区的好处就体现出来了。当数据制造快的时候，消费者来不及处理，未处理的数据可以暂时存在缓冲区中。等生产者的制造速度慢下来，消费者再慢慢处理掉。

为了充分复用，我们再拿寄信的例子来说事。假设邮递员一次只能带走1000封信。万一某次碰上情人节（也可能是圣诞节）送贺卡，需要寄出去的信超过1000封，这时候邮筒这个缓冲区就派上用场了。邮递员把来不及带走的信暂存在邮筒中，等下次过来时再拿走。

费了这么多口水，希望原先不太了解生产者／消费者模式的同学能够明白它是怎么一回事。然后在下一个帖子中，我们来说说如何确定数据单元。

另外，为了方便阅读，把本系列帖子的目录整理如下：

1、如何确定数据单元

2、队列缓冲区

3、队列缓冲区

4、双缓冲区

5、.....

[1]：如何确定数据单元？

既然前一个帖子已经搞过扫盲了，那接下来应该开始聊一些具体的编程技术问题了。不过在进入具体的技术细节之前，咱们先要搞明白一个问题：如何确定数据单元？只有把数据单元分析清楚，后面的技术设计才好搞。

★啥是数据单元

何谓数据单元捏？简单地说，每次生产者放到缓冲区的，就是一个数据单元；每次消费者从缓冲区取出的，也是一个数据单元。对于前一个帖子中寄信的例子，我们可以把每一封单独的信件看成是一个数据单元。

不过光这么介绍，太过于简单，无助于大伙儿分析出这玩意儿。所以，后面咱们来看一下数据单元需要具备哪些特性。搞明白这些特性之后，就容易从复杂的业务逻辑中分析出适合做数据单元的东西了。

★数据单元的特性

分析数据单元，需要考虑如下几个方面的特性：

2013年08月 (2)
2013年07月 (5)
展开

阅读排行
FragmentPagerAdapter (42486)
tomcat奇怪错误之A child (38084)
父类引用指向子类对象 (21610)
org.springframework.cor (18151)
生产者/消费者模式 (16224)
关于/dev/null及用途 (12462)
pthread_kill和pthread_ca (11913)
ERROR:TNS-12535: TN (10862)
kill用法详细解释（特别是 (9501)
spring security设置（spr (9184)

评论排行
父类引用指向子类对象 (16)
spring security设置（spr (8)
tomcat奇怪错误之A child (7)
pthread_kill和pthread_ca (7)
启动init.rc文件中的servic (6)
FragmentPagerAdapter (6)
android sqlite db-journal (3)
生产者/消费者模式 (3)
过桥问题和倒水问题算法 (3)
oracle11g安装过程中的问 (3)

推荐文章
*在R中使用支持向量机（SVM）进行数据挖掘（上）
* 你不再需要动态网页——编辑-发布-开发分离
*Android性能优化之使用线程池处理异步任务
* Nginx初探
*编译器架构的王者LLVM——（6）多遍翻译的宏翻译系统
* 我的第一个Apple Watch小游戏——猜数字（Swift）

最新评论
Android的交叉编译工具
wyqwh: 您好，我想请问您一个问题，如何使用android里面的交叉编译工具编译C程序，谢谢。
父类引用指向子类对象
_许青: 顶一下 对多态有新的理解。
android sqlite db-journal文件产生
cheetah747: 原来如此。。。
org.springframework.context.Ap
指尖残雪: 学习了，谢谢
FragmentPagerAdapter API
HanKaiDaHaoRen: sb
父类引用指向子类对象

◇关联到业务对象

首先，数据单元必须关联到某种业务对象。在考虑该问题的时候，你必须深刻理解当前这个生产者 / 消费者模式所对应的业务逻辑，才能够作出合适的判断。

由于“寄信”这个业务逻辑比较简单，所以大伙儿很容易就可以判断出数据单元是啥。但现实生活中，往往没这么乐观。大多数业务逻辑都比较复杂，当中包含的业务对象是层次繁多、类型各异。在这种情况下，就不易作出决策了。

这一步很重要，如果选错了业务对象，会导致后续程序设计和编码实现的复杂度大为上升，增加了开发和维护成本。

◇完整性

所谓完整性，就是在传输过程中，要保证该数据单元的完整。要么整个数据单元被传递到消费者，要么完全没有传递到消费者。不允许出现部分传递的情形。

对于寄信来说，你不能把半封信放入邮筒；同样的，邮递员从邮筒中拿信，也不能只拿出信的一部分。

◇独立性

所谓独立性，就是各个数据单元之间没有互相依赖，某个数据单元传输失败不应该影响已经完成传输的单元；也不应该影响尚未传输的单元。

为啥会出现传输失败捏？假如生产者的生产速度在一段时间内一直超过消费者的处理速度，那就会导致缓冲区不断增长并达到上限，之后的数据单元就会被丢弃。如果数据单元相互独立，等到生产者的速度降下来之后，后续的数据单元继续处理，不会受到牵连；反之，如果数据单元之间有某种耦合，导致被丢弃的数据单元会影响到后续其它单元的处理，那就会使程序逻辑变得非常复杂。

对于寄信来说，某封信弄丢了，不会影响后续信件的送达；当然更不会影响已经送达的信件。

◇颗粒度

前面提到，数据单元需要关联到某种业务对象。那么数据单元和业务对象是否要一一对应捏？很多场合确实是一一对应的。

不过，有时出于性能等因素的考虑，也可能会把N个业务对象打包成一个数据单元。那么，这个N该如何取值就是颗粒度的考虑了。颗粒度的大小是有讲究的。太大的颗粒度可能会造成某种浪费；太小的颗粒度可能会造成性能问题。颗粒度的权衡要基于多方面的因素，以及一些经验值的考量。

还是拿寄信的例子。如果颗粒度过小（比如设定为1），那邮递员每次只取出1封信。如果信件多了，那就得来回跑好多趟，浪费了时间。

如果颗粒度太大（比如设定为100），那寄信的人得等到凑满100封信才拿去放入邮筒。假如平时很少写信，就得等上很久，也不太爽。

可能有同学会问：生产者和消费者的颗粒度能否设置成不同大小（比如对于寄信人设置成1，对于邮递员设置成100）。当然，理论上可以这么干，但是在某些情况下会增加程序逻辑和代码实现的复杂度。后面讨论具体技术细节时，或许会聊到这个问题。

好，数据单元的话题就说到这。希望通过本帖子，大伙儿能够搞明白数据单元到底是怎么一回事。下一个帖子，咱们来聊一下“基于队列的缓冲区”，技术上如何实现。

[2]：队列缓冲区

经过前面两个帖子的铺垫，今天终于开始聊一些具体的编程技术了。由于不同的缓冲区类型、不同的并发场景对于具体的技术实现有较大的影响。为了深入浅出、便于大伙儿理解，咱们先来介绍最传统、最常见的方式。也就是单个生产者对应单个消费者，当中用队列（FIFO）作缓冲。

关于并发的场景，在之前的帖子“进程还线程？是一个问题！”中，已经专门论述了进程和线程各自的优缺点，两者皆不可偏废。所以，后面对各种缓冲区类型的介绍都会同时提及进程方式和线程方式。

★线程方式

先来说一下并发线程中使用队列的例子，以及相关的优缺点。

◇内存分配的性能

在线程方式下，生产者和消费者各自是一个线程。生产者把数据写入队列头（以下简称push），消费者从队列尾部读出数据（以下简称pop）。当队列为空，消费者就稍息（稍事休息）；当队列满（达到最大长度），生产者就稍息。整个流程并不复杂。

那么，上述过程会有什么问题捏？一个主要的问题是关于内存分配的性能开销。对于常见的队列实现：在每次push时，可能涉及到堆内存的分配；在每次pop时，可能涉及堆内存的释放。假如生产者和消费者都很勤快，频繁地push、pop，那内存分配的开销就很可观了。对于内存分配的开销，用Java的同学可以参见前几天的帖子“Java性能优化[1]”；对于用C/C++的同学，想必对OS底层机制会更清楚，应该知道分配堆内存（new或malloc）会有加锁的开销和用户态／核心态切换的开销。

那该怎么办捏？请听下文分解，关于“生产者／消费者模式[3]：环形缓冲区”。

◇同步和互斥的性能

另外，由于两个线程共用一个队列，自然就会涉及到线程间诸如同步啊、互斥啊、死锁啊等等劳心费神的事情。好在"操作系统"这门课程对此有详细介绍，学过的同学应该还有点印象吧？对于没学过这门课的同学，也不必难过，网上相关的介绍挺多的（比如"这里"），大伙自己去瞅一瞅。关于这方面的细节，咱今天就不多啰嗦了。

这会儿要细谈的是，同步和互斥的性能开销。在很多场合中，诸如信号量、互斥量等玩意儿的使用也是有不小的开销的（某些情况下，也可能导致用户态／核心态切换）。如果像刚才所说，生产者和消费者都很勤快，那这些开销也不容小觑啊。

这又该咋办捏？请听下文的下文分解，关于“生产者／消费者模式[4]：双缓冲区”。

◇适用于队列的场合

刚才尽批判了队列的缺点，难道队列方式就一无是处？非也。由于队列是很常见的数据结构，大部分编程语言都内置了队列的支持（具体介绍见"这里"），有些语言甚至提供了线程安全的队列（比如JDK 1.5引入的ArrayBlockingQueue）。因此，开发人员可以捡现成，避免了重新发明轮子。

所以，假如你的数据流量不是很大，采用队列缓冲区的好处还是很明显的：逻辑清晰、代码简单、维护方便。比较符合KISS原则。

★进程方式

说完了线程的方式，再来介绍基于进程的并发。

跨进程的生产者／消费者模式，非常依赖于具体的进程间通讯（IPC）方式。而IPC的种类名目繁多，不便于挨个列举（毕竟口水有限）。因此咱们挑选几种跨平台、且编程语言支持较多的IPC方式来说事儿。

◇匿名管道

感觉管道是最像队列的IPC类型。生产者进程在管道的写端放入数据；消费者进程在管道的读端取出数据。整个的效果和线程中使用队列非常类似，区别在于使用管道就无需操心线程安全、内存分配等琐事（操作系统暗中都帮你搞定了）。

管道又分命名管道和匿名管道两种，今天主要聊匿名管道。因为命名管道在不同的操作系统下差异较大（比如Win32和POSIX，在命名管道的API接口和功能实现上都有较大差异；有些平台不支持命名管道，比如Windows CE）。除了操作系统的问题，对于有些编程语言（比如Java）来说，命名管道是无法使用的。所以我一般不推荐使用这玩意儿。

其实匿名管道在不同平台上的API接口，也是有差异的（比如Win32的CreatePipe和POSIX的pipe，用法就很不一样）。但是我们可以仅使用标准输入和标准输出（以下简称stdio）来进行数据的流入流出。然后利用shell的管道符把生产者进程和消费者进程关联起来（没听说过这种手法的同学，可以看"这里"）。实际上，很多操作系统（尤其是POSIX风格的）自带的命令都充分利用了这个特性来实现数据的传输（比如more、grep等）。

这么干有几个好处：

1、基本上所有操作系统都支持在shell方式下使用管道符。因此很容易实现跨平台。

2、大部分编程语言都能够操作stdio，因此跨编程语言也就容易实现。

3、刚才已经提到，管道方式省却了线程安全方面的琐事。有利于降低开发、调试成本。

当然，这种方式也有自身的缺点：

1、生产者进程和消费者进程必须得在同一台主机上，无法跨机器通讯。这个缺点比较明显。

2、在一对一的情况下，这种方式挺合用。但如果要扩展到一对多或者多对一，那就有点棘手了。所以这种方式的扩展性要打个折扣。假如今后要考虑类似的扩展，这个缺点就比较明显。

3、由于管道是shell创建的，对于两边的进程不可见（程序看到的只是stdio）。在某些情况下，导致程序不便于对管道进行操纵（比如调整管道缓冲区尺寸）。这个缺点不太明显。

4、最后，这种方式只能单向传数据。好在大多数情况下，消费者进程不需要传数据给生产者进程。万一你确实需要信息反馈（从消费者到生产者），那就费劲了。可能得考虑换种IPC方式。

顺便补充几个注意事项，大伙儿留意一下：

1、对stdio进行读写操作是以阻塞方式进行。比如管道中没有数据，消费者进程的读操作就会一直停在哪儿，直到管道中重新有数据。

2、由于stdio内部带有自己的缓冲区（这缓冲区和管道缓冲区是两码事），有时会导致一些不太爽的现象（比如生产者进程输出了数据，但消费者进程没有立即读到）。具体的细节，大伙儿可以看"这里"。

◇SOCKET（TCP方式）

基于TCP方式的SOCKET通讯是又一个类似于队列的IPC方式。它同样保证了数据的顺序到达；同样有缓冲的机制。而且这玩意儿也是跨平台和跨语言的，和刚才介绍的shell管道符方式类似。

SOCKET相比shell管道符的方式，有啥优点捏？主要有如下几个优点：

1、SOCKET方式可以跨机器（便于实现分布式）。这是主要优点。

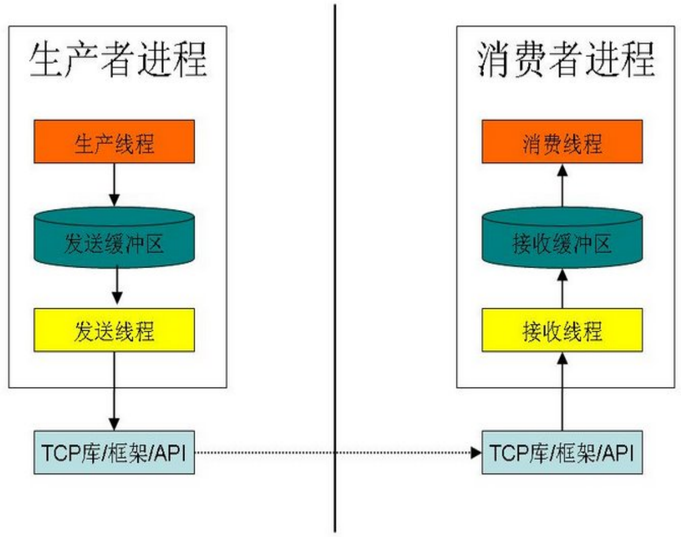
2、SOCKET方式便于将来扩展成为多对一或者一对多。这也是主要优点。

3、SOCKET可以设置阻塞和非阻塞方法，用起来比较灵活。这是次要优点。

4、SOCKET支持双向通讯，有利于消费者反馈信息。

当然有利就有弊。相对于上述shell管道的方式，使用SOCKET在编程上会更复杂一些。好在前人已经做了大量的工作，搞出很多SOCKET通讯库和框架给大伙儿用（比如C++的ACE库、Python的Twisted）。借助于这些第三方的库和框架，SOCKET方式用起来还是比较爽的。由于具体的网络通讯库该怎么用不是本系列的重点，此处就不细说了。

虽然TCP在很多方面比UDP可靠，但鉴于跨机器通讯先天的不可预料性（比如网线可能被某傻X给拔错了，网络的忙闲波动可能很大），在程序设计上我们还是要多留一手。具体该如何做捏？可以在生产者进程和消费者进程内部各自再引入基于线程的"生产者／消费者模式"。这话听着像绕口令，为了便于理解，画张图给大伙儿瞅一瞅。



这么做的关键点在于把代码分为两部分：生产线程和消费线程属于和业务逻辑相关的代码（和通讯逻辑无关）；发送线程和接收线程属于通讯相关的代码（和业务逻辑无关）。

这样的好处是很明显的，具体如下：

1、能够应对暂时性的网络故障。并且在网络故障解除后，能够继续工作。

2、网络故障的应对处理方式（比如断开后的尝试重连），只影响发送和接收线程，不会影响生产线程和消费线程（业务逻辑部分）。

3、具体的SOCKET方式（阻塞和非阻塞）只影响发送和接收线程，不影响生产线程和消费线程（业务逻辑部分）。

4、不依赖TCP自身的发送缓冲区和接收缓冲区。（默认的TCP缓冲区的大小可能无法满足实际要求）

5、业务逻辑的变化（比如业务需求变更）不影响发送线程和接收线程。

针对上述的最后一条，再多啰嗦几句。如果整个业务系统中有多个进程是采用上述的模式，那或许可以重构一把：在业务逻辑代码和通讯逻辑代码之间切一刀，把业务逻辑无关的部分封装成一个通讯中间件（说中间件显得比较牛X :-）。如果大伙儿对这玩意儿有兴趣，以后专门开个帖子聊。

[3]：环形缓冲区

前一个帖子提及了队列缓冲区可能存在的性能问题及解决方法：环形缓冲区。今天就专门来描述一下这个话题。

为了防止有人给咱扣上“过度设计”的大帽子，事先声明一下：只有当存储空间的分配／释放非常频繁并且确实产生了明显的影响，你才应该考虑环形缓冲区的使用。否则的话，还是老老实实用最基本、最简单的队列缓冲区吧。还有一点需要说明一下：本文所提及的“存储空间”，不仅包括内存，还可能包括诸如硬盘之类的存储介质。

★环形缓冲区 vs 队列缓冲区

◇外部接口相似

在介绍环形缓冲区之前，咱们先来回顾一下普通的队列。普通的队列有一个写入端和一个读出端。队列为空的

时候，读出端无法读取数据；当队列满（达到最大尺寸）时，写入端无法写入数据。

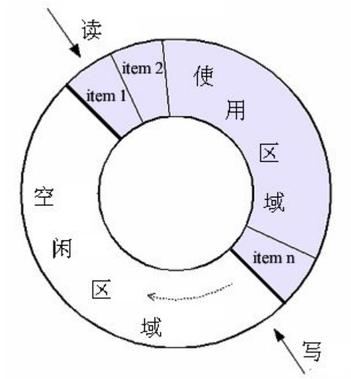
对于使用者来讲，环形缓冲区和队列缓冲区是一样的。它也有一个写入端（用于push）和一个读出端（用于pop），也有缓冲区“满”和“空”的状态。所以，从队列缓冲区切换到环形缓冲区，对于使用者来说能比较平滑地过渡。

◇内部结构迥异

虽然两者的对外接口差不多，但是内部结构和运作机制有很大差别。队列的内部结构此处就不多啰嗦了。重点介绍一下环形缓冲区的内部结构。

大伙儿可以把环形缓冲区的读出端（以下简称R）和写入端（以下简称W）想象成是两个人在体育场跑道上追逐（R追W）。当R追上W的时候，就是缓冲区为空；当W追上R的时候（W比R多跑一圈），就是缓冲区满。

为了形象起见，去找来一张图并略作修改，如下：



从上图可以看出，环形缓冲区所有的push和pop操作都是在一个固定的存储空间内进行。而队列缓冲区在push的时候，可能会分配存储空间用于存储新元素；在pop时，可能会释放废弃元素的存储空间。所以环形方式相比队列方式，少掉了对于缓冲区元素所用存储空间的分配、释放。这是环形缓冲区的一个主要优势。

★环形缓冲区的实现

如果你手头已经有现成的环形缓冲区可供使用，并且你对环形缓冲区的内部实现不感兴趣，可以跳过这段。

◇数组方式 vs 链表方式

环形缓冲区的内部实现，即可基于数组（此处的数组，泛指连续存储空间）实现，也可基于链表实现。

数组在物理存储上是一维的连续线性结构，可以在初始化时，把存储空间一次性分配好，这是数组方式的优点。但是要使用数组来模拟环，你必须在逻辑上把数组的头和尾相连。在顺序遍历数组时，对尾部元素（最后一个元素）要作一下特殊处理。访问尾部元素的下一个元素时，要重新回到头部元素（第0个元素）。如下图所示：

使用链表的方式，正好和数组相反：链表省去了头尾相连的特殊处理。但是链表在初始化的时候比较繁琐，而且在有些场合（比如后面提到的跨进程的IPC）不太方便使用。

◇读写操作

环形缓冲区要维护两个索引，分别对应写入端（W）和读取端（R）。写入（push）的时候，先确保环没满，然后把数据复制到W所对应的元素，最后W指向下一个元素；读取（pop）的时候，先确保环没空，然后返回R对应的元素，最后R指向下一个元素。

◇判断“空”和“满”

上述的操作并不复杂，不过有一个小小的麻烦：空环和满环的时候，R和W都指向同一个位置！这样就无法判

断到底是“空”还是“满”。大体上有两种方法可以解决该问题。

办法1：始终保持一个元素不用

当空环的时候，R和W重叠。当W比R跑得快，追到距离R还有一个元素间隔的时候，就认为环已经满。当环内元素占用的存储空间较大的时候，这种办法显得很土（浪费空间）。

办法2：维护额外变量

如果不喜欢上述办法，还可以采用额外的变量来解决。比如可以用一个整数记录当前环中已经保存的元素个数（该整数>=0）。当R和W重叠的时候，通过该变量就可以知道是“空”还是“满”。

◇元素的存储

由于环形缓冲区本身就是要降低存储空间分配的开销，因此缓冲区中元素的类型要选好。尽量存储值类型的数据，而不要存储指针（引用）类型的数据。因为指针类型的数据又会引起存储空间（比如堆内存）的分配和释放，使得环形缓冲区的效果打折扣。

★应用场合

刚才介绍了环形缓冲区内部的实现机制。按照前一个帖子的惯例，我们来介绍一下在线程和进程方式下的使用。

如果你所使用的编程语言和开发库中带有现成的、成熟的环形缓冲区，强烈建议使用现成的库，不要重新制造轮子；确实找不到现成的，才考虑自己实现。如果你纯粹是业余时间练练手，那另当别论。

◇用于并发线程

和线程中的队列缓冲区类似，线程中的环形缓冲区也要考虑线程安全的问题。除非你使用的环形缓冲区的库已经帮你实现了线程安全，否则你还是得自己动手搞定。线程方式下的环形缓冲区用得比较多，相关的网上资料也多，下面就大致介绍几个。

对于C++的程序员，强烈推荐使用boost提供的circular_buffer模板，该模板最开始是在boost 1.35版本中引入的。鉴于boost在C++社区中的地位，大伙儿应该可以放心使用该模板。

对于C程序员，可以去看看开源项目circbuf，不过该项目是GPL协议的，不太爽；而且活跃度不太高；而且只有一个开发人员。大伙儿慎用！建议只拿它当参考。

对于C#程序员，可以参考CodeProject上的一个示例。

◇用于并发进程

进程间的环形缓冲区，似乎少有现成的库可用。大伙儿只好自己动手、丰衣足食了。

适用于进程间环形缓冲的IPC类型，常见的有共享内存和文件。在这两种方式上进行环形缓冲，通常都采用数组的方式实现。程序事先分配好一个固定长度的存储空间，然后具体的读写操作、判断“空”和“满”、元素存储等细节就可参照前面所说的来进行。

共享内存方式的性能很好，适用于数据流量很大的场景。但是有些语言（比如Java）对于共享内存不支持。因此，该方式在多语言协同开发的系统中，会有一定的局限性。

而文件方式在编程语言方面支持很好，几乎所有编程语言都支持操作文件。但它可能会受限于磁盘读写（Disk I/O）的性能。所以文件方式不太适合于快速数据传输；但是对于某些“数据单元”很大的场合，文件方式是值得考虑的。

对于进程间的环形缓冲区，同样要考虑好进程间的同步、互斥等问题，限于篇幅，此处就不细说了。

下一个帖子，咱们来聊一下双缓冲区的使用。

上一篇 java.net.InetAddress类的使用
下一篇 接循环遍历整个hashMap,hashTable

顶 0 踩 0

猜你在找

- 360度解析亚马逊AWS数据存储服务
- 大数据之编程语言：Scala
- 大数据编程语言：Java基础
- 4.7.存储类&作用域&生命周期&链接属性-C语言高级专题
- Ceph—分布式存储系统的另一个选择

查看评论

3楼 cyclone409 2015-08-12 15:11发表

受教了

2楼 zhaozhichenghpu 2015-08-08 15:29发表

学习了，生产者消费者设计模式

1楼 卢凤文 2014-08-20 10:55发表

文章不错。但在队列缓冲区节介绍队列时好像错了，“生产者把数据写入队列头（以下简称push），消费者从队列尾部读出数据（以下简称pop）”，这应该反了吧。

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC
coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap