最新版Web服务器项目详解 - 09 日志系统（上）

原创 互联网猿 两猿社 2020-04-29 08:32

基础知识

日志，由服务器自动创建，并记录运行状态，错误信息，访问数据的文件。

同步日志，日志写入函数与工作线程串行执行，由于涉及到I/O操作，当单条日志比较大的时候，同步模式会阻塞整个处理流程，服务器所能处理的并发能力将有所下降，尤其是在峰值的时候，写日志可能成为系统的瓶颈。

生产者-消费者模型，并发编程中的经典模型。以多线程为例，为了实现线程间数据同步，生产者线程与消费者线程共享一个缓冲区，其中生产者线程往缓冲区中push消息，消费者线程从缓冲区中pop消息。

阻塞队列，将生产者-消费者模型进行封装，使用循环数组实现队列，作为两者共享的缓冲区。

异步日志，将所写的日志内容先存入阻塞队列，写线程从阻塞队列中取出内容，写入日志。

单例模式，最简单也是被问到最多的设计模式之一，保证一个类只创建一个实例，同时提供全局访问的方法。

整体概述

本项目中，使用单例模式创建日志系统，对服务器运行状态、错误信息和访问数据进行记录，该系统可以实现按天分类，超行分类功能，可以根据实际情况分别使用同步和异步写入两种方式。

其中异步写入方式，将生产者-消费者模型封装为阻塞队列，创建一个写线程，工作线程将要写的内容push进队列，写线程从队列中取出内容，写入日志文件。

日志系统大致可以分成两部分，其一是单例模式与阻塞队列的定义，其二是日志类的定义与使用。

本文内容

本篇将介绍单例模式与阻塞队列的定义，具体的涉及到单例模式、生产者-消费者模型，阻塞队列的代码实现。

单例模式，描述懒汉与饿汉两种单例模式，并结合线程安全进行讨论。

生产者-消费者模型，描述条件变量，基于该同步机制实现简单的生产者-消费者模型。

代码实现，结合代码对阻塞队列的设计进行详解。

单例模式

单例模式作为最常用的设计模式之一，保证一个类仅有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。

实现思路：私有化它的构造函数，以防止外界创建单例类的对象；使用类的私有静态指针变量指向类的唯一实例，并用一个公有的静态方法获取该实例。

单例模式有两种实现方法，分别是懒汉和饿汉模式。顾名思义，懒汉模式，即非常懒，不用的时候不去初始化，所以在第一次被使用时才进行初始化；饿汉模式，即迫不及待，在程序运行时立即初始化。

经典的线程安全懒汉模式

单例模式的实现思路如前述所示，其中，经典的线程安全懒汉模式，使用双检测锁模式。

```
class single{
private:
    //私有静态指针变量指向唯一实例
    static single *p;

    //静态锁，是由于静态函数只能访问静态成员
    static pthread_mutex_t lock;

    //私有化构造函数
    single(){
        pthread_mutex_init(&lock, NULL);
    }
    ~single(){}

public:
    //公有静态方法获取实例
    static single* getinstance();

};

pthread_mutex_t single::lock;

single* single::p = NULL;
```

```
single* single::getinstance(){
    if (NULL == p){
        pthread_mutex_lock(&lock);
        if (NULL == p){
            p = new single;
        }
        pthread_mutex_unlock(&lock);
    }
    return p;
}
```

为什么要用双检测，只检测一次不行吗？

如果只检测一次，在每次调用获取实例的方法时，都需要加锁，这将严重影响程序性能。双层检测可以有效避免这种情况，仅在第一次创建单例的时候加锁，其他时候都不再符合NULL == p的情况，直接返回已创建好的实例。

局部静态变量之线程安全懒汉模式

前面的双检测锁模式，写起来不太优雅，《Effective C++》（Item 04）中的提出另一种更优雅的单例模式实现，使用函数内的局部静态对象，这种方法不用加锁和解锁操作。

```
class single{
private:
    single(){}
    ~single(){}

public:
    static single* getinstance();

};

single* single::getinstance(){
    static single obj;
    return &obj;
}
```

这时候有人说了，这种方法不加锁会不会造成线程安全问题？

其实，C++0X以后，要求编译器保证内部静态变量的线程安全性，故C++0x之后该实现是线程安全的，C++0x之前仍需加锁，其中C++0x是C++11标准成为正式标准之前的草案临时名字。

所以，如果使用C++11之前的标准，还是需要加锁，这里同样给出加锁的版本。

```
class single{
private:
    static pthread_mutex_t lock;
    single(){
        pthread_mutex_init(&lock, NULL);
    }
    ~single(){}

public:
    static single* getinstance();

};
pthread_mutex_t single::lock;
single* single::getinstance(){
    pthread_mutex_lock(&lock);
```

```
16      static single obj;
17      pthread_mutex_unlock(&lock);
18      return &obj;
19  }
```

饿汉模式

饿汉模式不需要用锁，就可以实现线程安全。原因在于，在程序运行时就定义了对象，并对其初始化。之后，不管哪个线程调用成员函数getinstance()，都只不过是返回一个对象的指针而已。所以是线程安全的，不需要在获取实例的成员函数中加锁。

```
1   class single{
2   private:
3       static single* p;
4       single(){}
5       ~single(){}
6
7   public:
8       static single* getinstance();
9
10  };
11  single* single::p = new single();
12  single* single::getinstance(){
13      return p;
14  }
15
16  //测试方法
17  int main(){
18
19      single *p1 = single::getinstance();
20      single *p2 = single::getinstance();
21
22      if (p1 == p2)
23          cout << "same" << endl;
24
25      system("pause");
26      return 0;
27  }
```

饿汉模式虽好，但其存在隐藏的问题，在于非静态对象（函数外的static对象）在不同编译单元中的初始化顺序是未定义的。如果在初始化完成之前调用 getInstance() 方法会返回一个未定义的实例。

条件变量与生产者-消费者模型

条件变量API与陷阱

条件变量提供了一种线程间的通知机制，当某个共享数据达到某个值时,唤醒等待这个共享数据的线程。

基础API

- pthread_cond_init函数，用于初始化条件变量
- pthread_cond_destory函数，销毁条件变量
- pthread_cond_broadcast函数，以广播的方式唤醒**所有**等待目标条件变量的线程

- pthread_cond_wait函数，用于等待目标条件变量。该函数调用时需要传入 **mutex参数(加锁的互斥锁)**，函数执行时，先把调用线程放入条件变量的请求队列，然后将互斥锁mutex解锁，当函数成功返回为0时，表示重新抢到了互斥锁，互斥锁会再次被锁上， **也就是说函数内部会有一次解锁和加锁操作**.

使用pthread_cond_wait方式如下：

```
pthread _mutex_lock(&mutex)

while(线程执行的条件是否成立){
    pthread_cond_wait(&cond, &mutex);
}

pthread_mutex_unlock(&mutex);
```

pthread_cond_wait执行后的内部操作分为以下几步：

- 将线程放在条件变量的请求队列后，内部解锁
- 线程等待被pthread_cond_broadcast信号唤醒或者pthread_cond_signal信号唤醒，唤醒后去竞争锁
- 若竞争到互斥锁，内部再次加锁

陷阱一

使用前要加锁，为什么要加锁？

多线程访问，为了避免资源竞争，所以要加锁，使得每个线程互斥的访问公有资源。

pthread_cond_wait内部为什么要解锁？

如果while或者if判断的时候，满足执行条件，线程便会调用pthread_cond_wait阻塞自己，此时它还在持有锁，如果他不解锁，那么其他线程将会无法访问公有资源。

具体到pthread_cond_wait的内部实现，当pthread_cond_wait被调用线程阻塞的时候，pthread_cond_wait会自动释放互斥锁。

为什么要把调用线程放入条件变量的请求队列后再解锁？

线程是并发执行的，如果在把调用线程A放在等待队列之前，就释放了互斥锁，这就意味着其他线程比如线程B可以获得互斥锁去访问公有资源，这时候线程A所等待的条件改变了，但是它没有被放在等待队列上，导致A忽略了等待条件被满足的信号。

倘若在线程A调用pthread_cond_wait开始，到把A放在等待队列的过程中，都持有互斥锁，其他线程无法得到互斥锁，就不能改变公有资源。

为什么最后还要加锁？

将线程放在条件变量的请求队列后，将其解锁，此时等待被唤醒，若成功竞争到互斥锁，再次加锁。

陷阱二

为什么判断线程执行的条件用while而不是if？

一般来说，在多线程资源竞争的时候，在一个使用资源的线程里面（消费者）判断资源是否可用，不可用，便调用pthread_cond_wait，在另一个线程里面（生产者）如果判断资源可用的话，则调用pthread_cond_signal发送一个资源可用信号。

在wait成功之后，资源就一定可以被使用么？答案是否定的，如果同时有两个或者两个以上的线程正在等待此资源，wait返回后，资源可能已经被使用了。

再具体点，有可能多个线程都在等待这个资源可用的信号，信号发出后只有一个资源可用，但是有A，B两个线程都在等待，B比较速度快，获得互斥锁，然后加锁，消耗资源，然后解锁，之后A获得互斥锁，但A回去发现资源已经被使用了，它便有两个选择，一个是去访问不存在的资源，另一个就是继续等待，那么继续等待下去的条件就是使用while，要不然使用if的话pthread_cond_wait返回后，就会顺序执行下去。

所以，在这种情况下，应该使用while而不是if:

```
while(resource == FALSE)
    pthread_cond_wait(&cond, &mutex);
```

如果只有一个消费者，那么使用if是可以的。

生产者-消费者模型

这里摘抄《Unix 环境高级编程》中第11章线程关于pthread_cond_wait的介绍中有一个生产者-消费者的例子P311，其中，process_msg相当于消费者，enqueue_msg相当于生产者，struct msg* workq作为缓冲队列。

生产者和消费者是互斥关系，两者对缓冲区访问互斥，同时生产者和消费者又是一个相互协作与同步的关系，只有生产者生产之后，消费者才能消费。

```
#include <pthread.h>
struct msg {
  struct msg *m_next;
  /* value...*/
};

struct msg* workq;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void
process_msg() {
  struct msg* mp;
  for (;;) {
    pthread_mutex_lock(&qlock);
    //这里需要用while，而不是if
    while (workq == NULL) {
      pthread_cond_wait(&qread, &qlock);
    }
    mq = workq;
    workq = mp->m_next;
    pthread_mutex_unlock(&qlock);
    /* now process the message mp */
  }
}
```

```
26
27  void
28  enqueue_msg(struct msg* mp) {
29      pthread_mutex_lock(&qlock);
30      mp->m_next = workq;
31      workq = mp;
32      pthread_mutex_unlock(&qlock);
33      /** 此时另外一个线程在signal之前，执行了process_msg，刚好把mp元素拿走*/
34      pthread_cond_signal(&qready);
35      /** 此时执行signal，在pthread_cond_wait等待的线程被唤醒，
36          但是mp元素已经被另外一个线程拿走，所以，workq还是NULL ,因此需要继续等待*/
37  }
```

阻塞队列代码分析

阻塞队列类中封装了生产者-消费者模型，其中push成员是生产者，pop成员是消费者。

阻塞队列中，使用了循环数组实现了队列，作为两者共享缓冲区，当然了，队列也可以使用STL中的queue。

自定义队列

当队列为空时，从队列中获取元素的线程将会被挂起；当队列是满时，往队列里添加元素的线程将会挂起。

阻塞队列类中，有些代码比较简单，这里仅对push和pop成员进行详解。

```
1   class block_queue
2   {
3   public:
4
5       //初始化私有成员
6       block_queue(int max_size = 1000)
7       {
8           if (max_size <= 0)
9           {
10              exit(-1);
11          }
12
13          //构造函数创建循环数组
14          m_max_size = max_size;
15          m_array = new T[max_size];
16          m_size = 0;
17          m_front = -1;
18          m_back = -1;
19
20          //创建互斥锁和条件变量
21          m_mutex = new pthread_mutex_t;
22          m_cond = new pthread_cond_t;
23          pthread_mutex_init(m_mutex, NULL);
24          pthread_cond_init(m_cond, NULL);
25      }
26
27      //往队列添加元素，需要将所有使用队列的线程先唤醒
28      //当有元素push进队列，相当于生产者生产了一个元素
29      //若当前没有线程等待条件变量,则唤醒无意义
30      bool push(const T &item)
31      {
32          pthread_mutex_lock(m_mutex);
```

```
33          if (m_size >= m_max_size)
34          {
35              pthread_cond_broadcast(m_cond);
36              pthread_mutex_unlock(m_mutex);
37              return false;
38          }
39
40          //将新增数据放在循环数组的对应位置
41          m_back = (m_back + 1) % m_max_size;
42          m_array[m_back] = item;
43          m_size++;
44
45          pthread_cond_broadcast(m_cond);
46          pthread_mutex_unlock(m_mutex);
47
48          return true;
49      }
50
51      //pop时，如果当前队列没有元素,将会等待条件变量
52      bool pop(T &item)
53      {
54          pthread_mutex_lock(m_mutex);
55
56          //多个消费者的时候，这里要是用while而不是if
57          while (m_size <= 0)
58          {
59              //当重新抢到互斥锁，pthread_cond_wait返回为0
60              if (0 != pthread_cond_wait(m_cond, m_mutex))
61              {
62                  pthread_mutex_unlock(m_mutex);
63                  return false;
64              }
65          }
66
67          //取出队列首的元素，这里需要理解一下，使用循环数组模拟的队列
68          m_front = (m_front + 1) % m_max_size;
69          item = m_array[m_front];
70          m_size--;
71          pthread_mutex_unlock(m_mutex);
72          return true;
73      }
74
75      //增加了超时处理，在项目中没有使用到
76      //在pthread_cond_wait基础上增加了等待的时间，只指定时间内能抢到互斥锁即可
77      //其他逻辑不变
78      bool pop(T &item, int ms_timeout)
79      {
80          struct timespec t = {0, 0};
81          struct timeval now = {0, 0};
82          gettimeofday(&now, NULL);
83          pthread_mutex_lock(m_mutex);
84          if (m_size <= 0)
85          {
86              t.tv_sec = now.tv_sec + ms_timeout / 1000;
87              t.tv_nsec = (ms_timeout % 1000) * 1000;
88              if (0 != pthread_cond_timedwait(m_cond, m_mutex, &t))
89              {
90                  pthread_mutex_unlock(m_mutex);
91                  return false;
92              }
93          }
94
95          if (m_size <= 0)
96          {
97              pthread_mutex_unlock(m_mutex);
98              return false;
99          }
```

```
100
101             m_front = (m_front + 1) % m_max_size;
102             item = m_array[m_front];
103             m_size--;
104             pthread_mutex_unlock(m_mutex);
105             return true;
106         }
107     };
108
109     #endif
```

如果本文对你有帮助， 阅读原文 star一下服务器项目，我们需要你的星星^_^.

完。