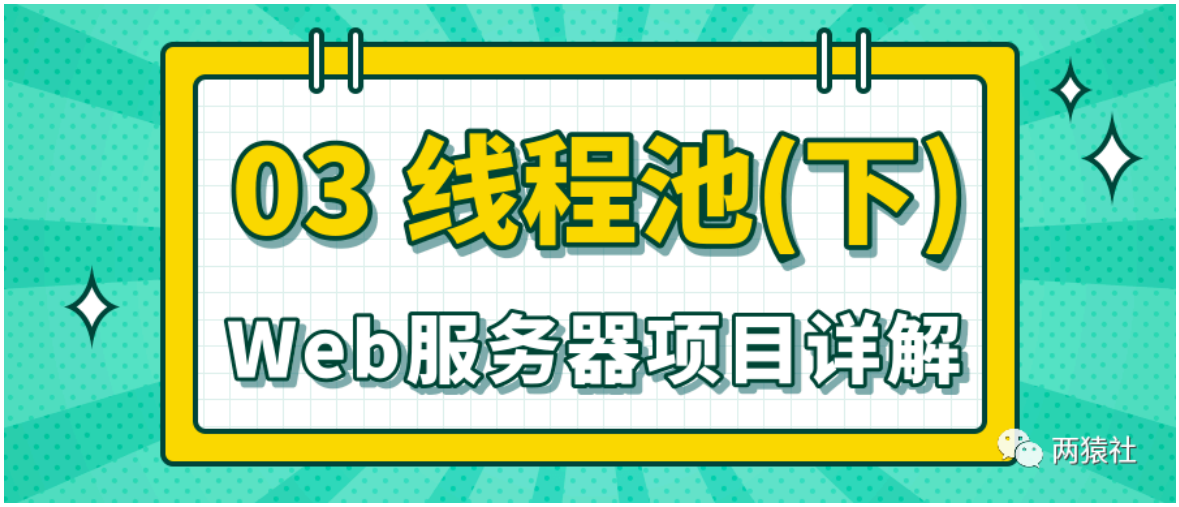
最新版Web服务器项目详解 - 03 半同步半反应堆线程池（下）

原创 互联网猿 两猿社 2020-04-21 14:30



互联网猿 | 两猿社

基础知识

静态成员变量

将类成员变量声明为static，则为静态成员变量，与一般的成员变量不同，无论建立多少对象，都只有一个静态成员变量的拷贝，静态成员变量属于一个类，所有对象共享。

静态变量在编译阶段就分配了空间，对象还没创建时就已经分配了空间，放到全局静态区。

- 静态成员变量
  - 最好是类内声明，类外初始化（以免类名访问静态成员访问不到）。
  - 无论公有，私有，静态成员都可以在类外定义，但私有成员仍有访问权限。
  - 非静态成员类外不能初始化。
  - 静态成员数据是共享的。

静态成员函数

将类成员函数声明为static，则为静态成员函数。

- 静态成员函数
  - 静态成员函数可以直接访问静态成员变量，不能直接访问普通成员变量，但可以通过参数传递的方式访问。
  - 普通成员函数可以访问普通成员变量，也可以访问静态成员变量。

- 静态成员函数没有this指针。非静态数据成员为对象单独维护，但静态成员函数为共享函数，无法区分是哪个对象，因此不能直接访问普通变量成员，也没有this指针。

pthread_create陷阱

首先看一下该函数的函数原型。

```
#include <pthread.h>
int pthread_create (pthread_t *thread_tid,                    //返回新生成的线程的id
                    const pthread_attr_t *attr,              //指向线程属性的指针,通常设置
                    void * (*start_routine) (void *),        //处理线程函数的地址
                    void *arg);                              //start_routine()中的参数
```

函数原型中的第三个参数，为函数指针，指向处理线程函数的地址。该函数，要求为静态函数。如果处理线程函数为类成员函数时，需要将其设置为**静态成员函数**。

this指针的锅

pthread_create的函数原型中第三个参数的类型为函数指针，指向的线程处理函数参数类型为（void *），若线程函数为类成员函数，则this指针会作为默认的参数被传进函数中，从而和线程函数参数（void*）不能匹配，不能通过编译。

静态成员函数就没有这个问题，里面没有this指针。

线程池分析

线程池的设计模式为半同步/半反应堆，其中反应堆具体为Proactor事件处理模式。

具体的，主线程为异步线程，负责监听文件描述符，接收socket新连接，若当前监听的socket发生了读写事件，然后将任务插入到请求队列。工作线程从请求队列中取出任务，完成读写数据的处理。

线程池类定义

具体定义可以看代码。需要注意，线程处理函数和运行函数设置为私有属性。

```cpp
template<typename T>
class threadpool{
    public:
        //thread_number是线程池中线程的数量
        //max_requests是请求队列中最多允许的、等待处理的请求的数量
        //connPool是数据库连接池指针
        threadpool(connection_pool *connPool, int thread_number = 8, int max_reque
        ~threadpool();

        //像请求队列中插入任务请求
        bool append(T* request);

    private:
        //工作线程运行的函数
        //它不断从工作队列中取出任务并执行之
        static void *worker(void *arg);

        void run();

    private:
        //线程池中的线程数
        int m_thread_number;

        //请求队列中允许的最大请求数
        int m_max_requests;

        //描述线程池的数组，其大小为m_thread_number
        pthread_t *m_threads;

        //请求队列
        std::list<T *>m_workqueue;

        //保护请求队列的互斥锁
        locker m_queuelocker;

        //是否有任务需要处理
        sem m_queuestat;

        //是否结束线程
        bool m_stop;

        //数据库连接池
        connection_pool *m_connPool;
};
```

线程池创建与回收

构造函数中创建线程池,pthread_create函数中将类的对象作为参数传递给静态函数(worker),在静态函数中引用这个对象,并调用其动态方法(run)。

具体的，类对象传递时用this指针，传递给静态函数后，将其转换为线程池类，并调用私有成员函数run。

```
template<typename T>
threadpool<T>::threadpool( connection_pool *connPool, int thread_number, int max_r

    if(thread_number<=0||max_requests<=0)
        throw std::exception();

    //线程id初始化
    m_threads=new pthread_t[m_thread_number];
    if(!m_threads)
        throw std::exception();
    for(int i=0;i<thread_number;++i)
    {
        //循环创建线程，并将工作线程按要求进行运行
        if(pthread_create(m_threads+i,NULL,worker,this)!=0){
            delete [] m_threads;
            throw std::exception();
        }

        //将线程进行分离后，不用单独对工作线程进行回收
        if(pthread_detach(m_threads[i])){
            delete[] m_threads;
            throw std::exception();
        }
    }
}
```

向请求队列中添加任务

通过list容器创建请求队列，向队列中添加时，通过互斥锁保证线程安全，添加完成后通过信号量提醒有任务要处理，最后注意线程同步。

```
template<typename T>
bool threadpool<T>::append(T* request)
{
    m_queuelocker.lock();

    //根据硬件，预先设置请求队列的最大值
    if(m_workqueue.size()>m_max_requests)
    {
        m_queuelocker.unlock();
        return false;
    }

    //添加任务
    m_workqueue.push_back(request);
    m_queuelocker.unlock();

    //信号量提醒有任务要处理
    m_queuestat.post();
    return true;
}
```

线程处理函数

内部访问私有成员函数run，完成线程处理要求。

```
template<typename T>
void* threadpool<T>::worker(void* arg){

```

```
    //将参数强转为线程池类，调用成员方法
    threadpool* pool=(threadpool*)arg;
    pool->run();
    return pool;
}
```

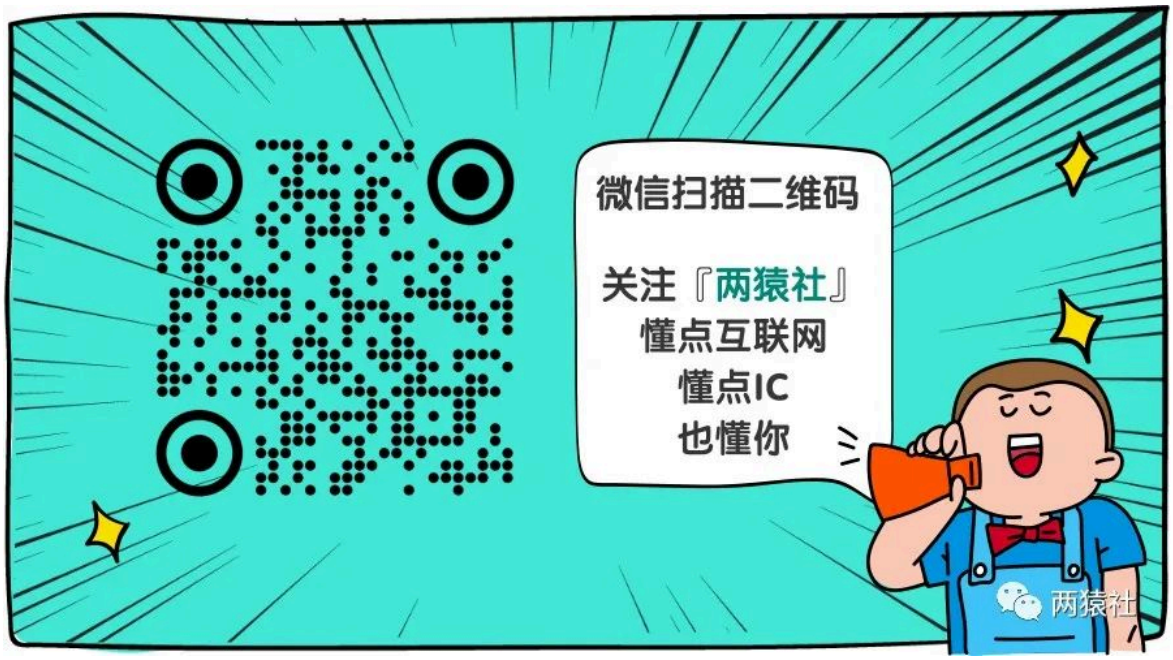
run执行任务

主要实现，工作线程从请求队列中取出某个任务进行处理，注意线程同步。

```
template<typename T>
void threadpool<T>::run()
{
    while(!m_stop)
    {
        //信号量等待
        m_queuestat.wait();

        //被唤醒后先加互斥锁
        m_queuelocker.lock();
        if(m_workqueue.empty())
        {
            m_queuelocker.unlock();
            continue;
        }

        //从请求队列中取出第一个任务
        //将任务从请求队列删除
        T* request=m_workqueue.front();
        m_workqueue.pop_front();
        m_queuelocker.unlock();
        if(!request)
            continue;

        //从连接池中取出一个数据库连接
        request->mysql = m_connPool->GetConnection();

        //process(模板类中的方法,这里是http类)进行处理
        request->process();

        //将数据库连接放回连接池
        m_connPool->ReleaseConnection(request->mysql);
    }
}
```

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。



Web服务器-原始版本 13

Web服务器-原始版本 · 目录

上一篇
最新版Web服务器项目详解 - 02 半同步半反应堆线程池（上）

下一篇
最新版Web服务器项目详解 - 04 http连接处理（上）

阅读原文