手把手教你写Linux线程池

线程池

> 顾名思义，存储线程的池子。线程池是线程的一种使用模式。在平常业务开发中常规的逻辑是遇到任务然后创建线程去执行。但是线程的频繁创建就类似于内存的频繁申请和销毁，会给操作系统带来大的压力，进而影响整体的性能。所以我们一次申请好一定数量而定线程，然后将线程的管理操作交给线程池，就避免了在短时间内不断创建与销毁线程的代价，线程池不但能够保证内核的充分利用，还能防止过分调度，并根据实际业务情况进行修改。

使用线程池的好处

- 任务到来后立马就有线程去执行任务，节省了创建线程的时间
- 防止服务器线程过多导致的系统过载问题
- 相对于进程池，线程池资源占用较少，但是健壮性很差
- 降低资源消耗，通过重用已经创建的线程来降低线程创建和销毁的消耗
- 提高线程的可管理性，线程池可以统一管理、分配、调优和监控其中的线程

什么情况下使用线程池

- 需要大量的线程来完成任务，且完成任务的时间比较短
- 对性能要求苛刻的应用
- 接收突发性的大量请求，但不至于使服务器因此产生大量线程的应用

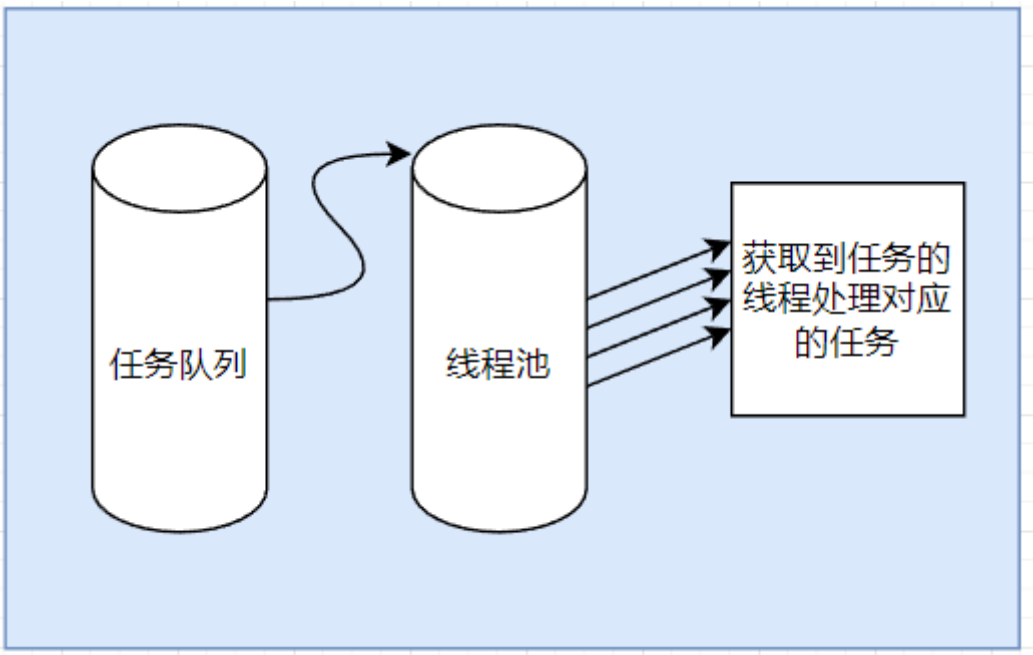
线程不是越多越好

> 线程的越多，可能会导致线程切换越频繁， 进而还有可能导致程序运行效率降低。多线程程序的运行效率， 是一个正态分布的结果， 线程数量从1开始增加， 随着线程数量的增加， 程序的运行效率逐渐变高， 直到线程数量达到一个临界值， 再次增加线程数量时， 程序的运行效率会减小（主要是由于频繁的线程切换影响线程运行效率）。

- 线程若不限制数量的创建，线程创建过多，资源耗尽，有程序崩溃的风险
- 处理一个短时间任务时，线程会频繁创建和销毁，占用系统资源，降低系统性能

Linux如何实现一个线程池

- 每个任务在放入任务队列时设置好需要处理的数据和处理函数
- 线程池中的线程负责从任务队列当中获取任务，并进行处理



代码实现过程

- 创建一个任务类CTask,可以设置处理任务的回调func以及需要处理的数据m_data
- 创建一个线程池类CThreadPool，成员变量有设置线程池中线程的最大数量thr_max，任务缓冲队列m_queue，互斥量m_mutex，用于实现对缓冲队列的安全性，条件变量m_cond，用于实现线程池中线程的同步

> 创建任务类

```
/* 任务类 */
class CTask
{
public:
    CTask(){}
    ~CTask(){}

    void SetTask(int data, func handler) // 设置数据和处理接口
    {
        m_data = data;
        m_handler = handler;
    }

    void Do() // 执行任务
    {
        return m_handler(m_data);
    }

private:
    int  m_data;     // 数据
    func m_handler;  // 处理接口
};
```

> 创建线程池类

- 1. 创建线程池

```
    /* 创建线程池 */
    for (int i = 0; i < m_SumMax; i++)
    {
        pthread_t tid;
        int ret = pthread_create(&tid, NULL, ThrPoolRun, this);
        if (ret != 0)
        {
            printf("thread create error\n");
        }
    }
```

- 2. 任务放入队列

```
bool TaskPush(CTask &task)
{
    pthread_mutex_lock(&m_Mutex);
    m_Queue.push(task);
    pthread_mutex_unlock(&m_Mutex);
    pthread_cond_signal(&m_Mond);
    return true;
}
```

- 3. 线程池空闲线程从队列获取任务并处理

```
CThreadPool *p = (CThreadPool*)arg;

while (p->m_bIsRun)
{
    pthread_mutex_lock(&p->m_Mutex);

    /* 等待任务到来 */
    while (p->m_Queue.empty())
    {
        pthread_cond_wait(&p->m_Mond, &p->m_Mutex);
    }

    /* 取出任务 */
    CTask Task;
    Task =p->m_Queue.front();
    p->m_Queue.pop();
    pthread_mutex_unlock(&p->m_Mutex);

    /* 处理任务 */
    Task.Do();
}
```

> main函数

```
void TaskFunc1(int nData)
{
    printf("TaskFunc1, ThreadId: %p, nData:%d\n", pthread_self(), nData);

    sleep(1);
}

void TaskFunc2(int nData)
{
    printf("TaskFunc2, ThreadId: %p, nData:%d\n", pthread_self(), nData);

    sleep(1);
}

int main(int argc, char const *argv[])
{
    CThreadPool ThreadPool;

    for (size_t i = 0; i < 10; i++)
    {
        CTask Task;

        (0 == (i % 2)) ? Task.SetTask(i, TaskFunc1) : Task.SetTask(i, TaskFunc2);

        ThreadPool.TaskPush(Task);  // 放入任务队列
    }

    sleep(3);
    return 0;
}
```

运行结果

- 10个任务被5个线程分别处理完

```
TaskFunc1, ThreadId: 0x7ff8210b7700, nData:0
TaskFunc2, ThreadId: 0x7ff81f8b4700, nData:1
TaskFunc1, ThreadId: 0x7ff8218b8700, nData:2
TaskFunc2, ThreadId: 0x7ff8200b5700, nData:3
TaskFunc1, ThreadId: 0x7ff8208b6700, nData:4
TaskFunc2, ThreadId: 0x7ff8218b8700, nData:5
TaskFunc1, ThreadId: 0x7ff8208b6700, nData:6
TaskFunc2, ThreadId: 0x7ff81f8b4700, nData:9
TaskFunc2, ThreadId: 0x7ff8200b5700, nData:7
TaskFunc1, ThreadId: 0x7ff8210b7700, nData:8
```

总结

至此，一个简单的线程池实例就完成了。实际工作中我们可以根据实际的业务量来初始化线程池中线程的个数，并根据任务量的多少动态的增加或减少线程池中的线程。好了，现在让我们行动起来吧，自己编写一个线程池。

如果需要线程池源码，关注 `Linux兵工厂` ，回复**1024**获取，并由大量Linux资料赠送。