线程池

简介

线程池（英语：thread pool）：一种线程使用模式。线程过多会带来调度开销，进而影响缓存局部性和整体性能。而线程池维护着多个线程，等待着监督管理者分配可并发执行的任务。这避免了在处理短时间任务时创建与销毁线程的代价。线程池不仅能够保证内核的充分利用，还能防止过分调度。可用线程数量应该取决于可用的并发处理器、处理器内核、内存、网络sockets等的数量。 例如，线程数一般取cpu数量+2比较合适，线程数过多会导致额外的线程切换开销。
任务调度以执行线程的常见方法是使用同步队列，称作任务队列。池中的线程等待队列中的任务，并把执行完的任务放入完成队列中。
线程池模式一般分为两种：

1. HS/HA半同步/半异步模式
    - 半同步/半异步模式又称为生产者消费者模式，是比较常见的实现方式，比较简单。分为同步层、队列层、异步层三层。同步层的主线程处理工作任务并存入工作队列，工作线程从工作队列取出任务进行处理，如果工作队列为空，则取不到任务的工作线程进入挂起状态。由于线程间有数据通信，因此不适于大数据量交换的场合。
2. L/F领导者与跟随者模式
    - 领导者跟随者模式，在线程池中的线程可处在3种状态之一：领导者leader、追随者follower或工作者processor。任何时刻线程池只有一个领导者线程。事件到达时，领导者线程负责消息分离，并从处于追随者线程中选出一个来当继任领导者，然后将自身设置为工作者状态去处置该事件。处理完毕后工作者线程将自身的状态置为追随者。这一模式实现复杂，但避免了线程间交换任务数据，提高了CPU cache相似性。在ACE(Adaptive Communication Environment)中，提供了领导者跟随者模式实现。

注

线程池的伸缩性对性能有较大的影响。

- 创建太多线程，将会浪费一定的资源，有些线程未被充分使用。
- 销毁太多线程，将导致之后浪费时间再次创建它们。
- 创建线程太慢，将会导致长时间的等待，性能变差。
- 销毁线程太慢，导致其它线程资源饥饿。

组成部分

1. 线程池管理器（ThreadPoolManager）
    - 用于创建并管理线程池

```
typedef list<IWorkItem*> WorkItemQueue;


class CMyThreadPool
{
public:
```

```
    CMyThreadPool();
    virtual ~CMyThreadPool();
public:
    int Create(int MaxThreads);  //创建线程池
    int Destroy(); //销毁线程池
    int InsertWorkItem(IWorkItem* WorkItem); //插入任务
private:
    static DWORD __stdcall WorkerThread(LPVOID lpParam);
    IWorkItem* RemoveWorkItem();  //获取任务
private:
    int     mIdleCount;           //空闲的线程数
    int     mCurrentThreads;      //当前线程数
    int     mMaxThreads;          //最大线程数
    HANDLE *mThreads;
    WorkItemQueue mWorkItemQueue; //工作项队列
    CriticalSection mCS;          //队列同步对象
    CMySemaphore mCMySemaphore;   //任务信号量
    bool    mIsDestroy;
};
```

2. 工作线程（WorkThread）
    - 线程池中线程
3. 任务接口（Task）
    - 每个任务必须实现的接口，以供工作线程调度任务的执行。

```
class IWorkItem
{
public:
    virtual int Execute() = 0;  //执行任务
    virtual int Abort() = 0;    //终止任务
};

// 具体任务派生自IWorkItem
class WorkItem1 :public IWorkItem
{
public:
    WorkItem1(int number)
    {
        mNumber = number;
    }

    virtual int Execute()
    {
        printf("id:%d num:%d WorkItem1 Execute\n",
GetCurrentThreadId(), mNumber);
        return 0;
    }
    virtual int Abort()
    {
        printf("id:%d num:%d WorkItem1 Abort\n", GetCurrentThreadId(),
mNumber);
        return 0;
    }
private:
```

```
    int mNumber;
};
```

4. 任务队列
    - 用于存放没有处理的任务。提供一种缓冲机制。

应用范围

- 需要大量的线程来完成任务，且完成任务的时间比较短。 WEB服务器完成网页请求这样的任务，使用线程池技术是非常合适的。因为单个任务小，而任务数量巨大，你可以想象一个热门网站的点击次数。 但对于长时间的任务，比如一个Telnet连接请求，线程池的优点就不明显了。因为Telnet会话时间比线程的创建时间大多了。
- 对性能要求苛刻的应用，比如要求服务器迅速响应客户请求。
- 接受突发性的大量请求，但不至于使服务器因此产生大量线程的应用。