线程池

由 enlai.feng创建, 最后修改于二月 05, 2024

- •

  线程（thread）——共享内存
- •

  进程（process）——非共享内存

我们使用线程的时候就去创建一个线程，这样实现起来非常简便，但是就会有一个问题：如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间。

线程池是一种多线程处理形式，处理过程中将**任务添加到队列**，然后在创建线程后自动启动这些任务。线程池线程都是后台线程。每个线程都使用默认的堆栈大小，以默认的优先级运行，并处于多线程单元中。如果某个线程在托管代码中空闲（如正在等待某个事件），则线程池将插入另一个辅助线程来使所有处理器保持繁忙。如果所有线程池线程都始终保持繁忙，但队列中包含挂起的工作，则线程池将在一段时间后创建另一个辅助线程但线程的数目永远不会超过最大值。超过最大值的线程可以排队，但他们要等到其他线程完成后才启动。

线程池的使用主要分为3个部分，这三个部分配合就可以得到一个完整的线程池：

（1）任务队列：存储需要处理的任务，由工作的线程来处理这些任务

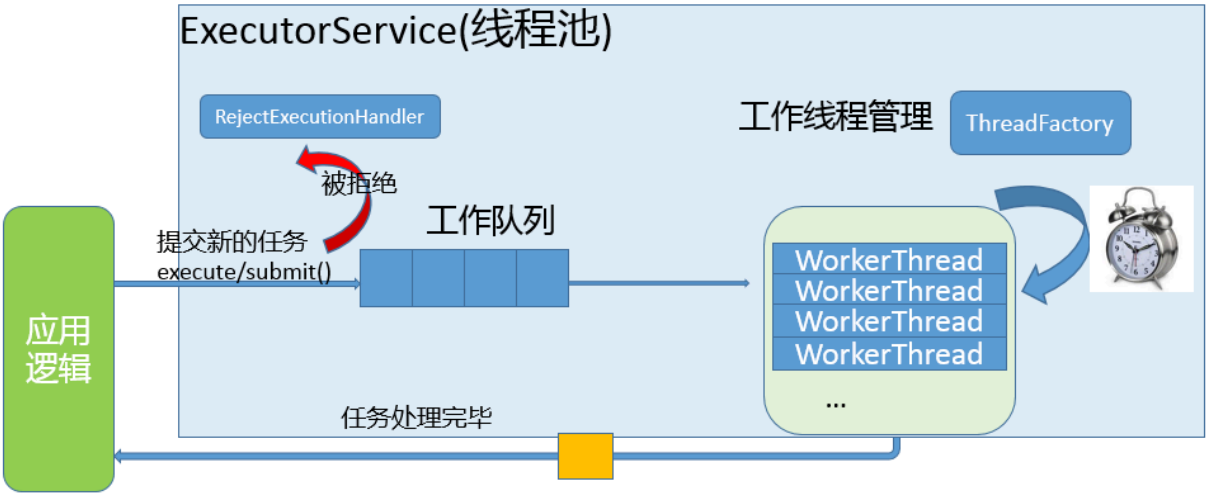
- 通过线程池提供的API函数，将一个待处理的任务添加到任务队列，或者从任务队列中删除
- 已处理的任务会被从任务队列中删除
- 线程池的使用者，也就是调用线程池的函数向任务队列中添加任务的线程，就是生产者线程。

（2）工作的线程（任务队列任务的消费者），N个

- 线程池中维护了一定数量的工作线程。不停读取任务队列，从中取出任务并进行处理
- 工作的线程相当于任务队列中的消费者角色
- 如果任务队列为，工作的线程将被阻塞。
- 如果阻塞之后有了新的任务，将由生产者将阻塞消除，工作线程开始工作

（3）管理者线程

- 周期性的对任务队列中的任务数量以及处于忙状态的工作线程个数进行检测
- 当任务过多的时候，可以适当创建一些新的工作线程
- 当任务过少的时候，可以适当销毁一些工作线程



任务队列

```
// 任务结构体
typedef struct Task
{
    void (*function)(void* arg);
    void* arg;
```

```
}Task;
```

线程池定义

```
// 线程池结构体
struct ThreadPool
{
    // 任务队列
    Task* taskQ;
    int queueCapacity;  // 容量
    int queueSize;      // 当前任务个数
    int queueFront;     // 队头 -> 取数据
    int queueRear;      // 队尾 -> 放数据

    pthread_t managerID;    // 管理者线程ID
    pthread_t *threadIDs;   // 工作的线程ID
    int minNum;             // 最小线程数量
    int maxNum;             // 最大线程数量
    int busyNum;            // 忙的线程的个数
    int liveNum;            // 存活的线程的个数
    int exitNum;            // 要销毁的线程个数
    pthread_mutex_t mutexPool;  // 锁整个的线程池
    pthread_mutex_t mutexBusy;  // 锁busyNum变量
    pthread_cond_t notFull;     // 任务队列是不是满了
    pthread_cond_t notEmpty;    // 任务队列是不是空了

    int shutdown;           // 是不是要销毁线程池, 销毁为1, 不销毁为0
};
```

头文件声明

```
#ifndef _THREADPOOL_H
#define _THREADPOOL_H

typedef struct ThreadPool ThreadPool;
// 创建线程池并初始化
ThreadPool *threadPoolCreate(int min, int max, int queueSize);

// 销毁线程池
int threadPoolDestroy(ThreadPool* pool);

// 给线程池添加任务
void threadPoolAdd(ThreadPool* pool, void(*func)(void*), void* arg);

// 获取线程池中工作的线程的个数
int threadPoolBusyNum(ThreadPool* pool);

// 获取线程池中活着的线程的个数
int threadPoolAliveNum(ThreadPool* pool);

/////////////////////
// 工作的线程(消费者线程)任务函数
void* worker(void* arg);
// 管理者线程任务函数
void* manager(void* arg);
// 单个线程退出
void threadExit(ThreadPool* pool);
```

```
#endif  // _THREADPOOL_H
```

源文件定义

```
ThreadPool* threadPoolCreate(int min, int max, int queueSize)
{
    ThreadPool* pool = (ThreadPool*)malloc(sizeof(ThreadPool));
    do
    {
        if (pool == NULL)
        {
            printf("malloc threadpool fail...\n");
            break;
        }

        pool->threadIDs = (pthread_t*)malloc(sizeof(pthread_t) * max);
        if (pool->threadIDs == NULL)
        {
            printf("malloc threadIDs fail...\n");
            break;
        }
        memset(pool->threadIDs, 0, sizeof(pthread_t) * max);
        pool->minNum = min;
        pool->maxNum = max;
        pool->busyNum = 0;
        pool->liveNum = min;    // 和最小个数相等
        pool->exitNum = 0;

        if (pthread_mutex_init(&pool->mutexPool, NULL) != 0 ||
            pthread_mutex_init(&pool->mutexBusy, NULL) != 0 ||
            pthread_cond_init(&pool->notEmpty, NULL) != 0 ||
            pthread_cond_init(&pool->notFull, NULL) != 0)
        {
            printf("mutex or condition init fail...\n");
            break;
        }

        // 任务队列
        pool->taskQ = (Task*)malloc(sizeof(Task) * queueSize);
        pool->queueCapacity = queueSize;
        pool->queueSize = 0;
        pool->queueFront = 0;
        pool->queueRear = 0;

        pool->shutdown = 0;

        // 创建线程
        pthread_create(&pool->managerID, NULL, manager, pool);
        for (int i = 0; i < min; ++i)
        {
            pthread_create(&pool->threadIDs[i], NULL, worker, pool);
        }
        return pool;
    } while (0);

    // 释放资源
    if (pool && pool->threadIDs) free(pool->threadIDs);
    if (pool && pool->taskQ) free(pool->taskQ);
    if (pool) free(pool);
```

```
    return NULL;
}

int threadPoolDestroy(ThreadPool* pool)
{
    if (pool == NULL)
    {
        return -1;
    }

    // 关闭线程池
    pool->shutdown = 1;
    // 阻塞回收管理者线程
    pthread_join(pool->managerID, NULL);
    // 唤醒阻塞的消费者线程
    for (int i = 0; i < pool->liveNum; ++i)
    {
        pthread_cond_signal(&pool->notEmpty);
    }
    // 释放堆内存
    if (pool->taskQ)
    {
        free(pool->taskQ);
    }
    if (pool->threadIDs)
    {
        free(pool->threadIDs);
    }

    pthread_mutex_destroy(&pool->mutexPool);
    pthread_mutex_destroy(&pool->mutexBusy);
    pthread_cond_destroy(&pool->notEmpty);
    pthread_cond_destroy(&pool->notFull);

    free(pool);
    pool = NULL;

    return 0;
}


void threadPoolAdd(ThreadPool* pool, void(*func)(void*), void* arg)
{
    pthread_mutex_lock(&pool->mutexPool);
    while (pool->queueSize == pool->queueCapacity && !pool->shutdown)
    {
        // 阻塞生产者线程
        pthread_cond_wait(&pool->notFull, &pool->mutexPool);
    }
    if (pool->shutdown)
    {
        pthread_mutex_unlock(&pool->mutexPool);
        return;
    }
    // 添加任务
    pool->taskQ[pool->queueRear].function = func;
    pool->taskQ[pool->queueRear].arg = arg;
    pool->queueRear = (pool->queueRear + 1) % pool->queueCapacity;
    pool->queueSize++;
```

```
    pthread_cond_signal(&pool->notEmpty);
    pthread_mutex_unlock(&pool->mutexPool);
}
```

冯

```
int threadPoolBusyNum(ThreadPool* pool)
{
    pthread_mutex_lock(&pool->mutexBusy);
    int busyNum = pool->busyNum;
    pthread_mutex_unlock(&pool->mutexBusy);
    return busyNum;
}

int threadPoolAliveNum(ThreadPool* pool)
{
    pthread_mutex_lock(&pool->mutexPool);
    int aliveNum = pool->liveNum;
    pthread_mutex_unlock(&pool->mutexPool);
    return aliveNum;
}

void* worker(void* arg)
{
    ThreadPool* pool = (ThreadPool*)arg;

    while (1)
    {
        pthread_mutex_lock(&pool->mutexPool);
        // 当前任务队列是否为空
        while (pool->queueSize == 0 && !pool->shutdown)
        {
            // 阻塞工作线程
            pthread_cond_wait(&pool->notEmpty, &pool->mutexPool);

            // 判断是不是要销毁线程
            if (pool->exitNum > 0)
            {
                pool->exitNum--;
                if (pool->liveNum > pool->minNum)
                {
                    pool->liveNum--;
                    pthread_mutex_unlock(&pool->mutexPool);
                    threadExit(pool);
                }
            }
        }

        // 判断线程池是否被关闭了
        if (pool->shutdown)
        {
            pthread_mutex_unlock(&pool->mutexPool);
            threadExit(pool);
        }

        // 从任务队列中取出一个任务
        Task task;
        task.function = pool->taskQ[pool->queueFront].function;
        task.arg = pool->taskQ[pool->queueFront].arg;
        // 移动头结点
        pool->queueFront = (pool->queueFront + 1) % pool->queueCapacity;
        pool->queueSize--;
        // 解锁
```

```
        pthread_cond_signal(&pool->notFull);
        pthread_mutex_unlock(&pool->mutexPool);

        printf("thread %ld start working...\n", pthread_self());
        pthread_mutex_lock(&pool->mutexBusy);
        pool->busyNum++;
        pthread_mutex_unlock(&pool->mutexBusy);
        task.function(task.arg);
        free(task.arg);
        task.arg = NULL;

        printf("thread %ld end working...\n", pthread_self());
        pthread_mutex_lock(&pool->mutexBusy);
        pool->busyNum--;
        pthread_mutex_unlock(&pool->mutexBusy);
    }
    return NULL;
}

void* manager(void* arg)
{
    ThreadPool* pool = (ThreadPool*)arg;
    while (!pool->shutdown)
    {
        // 每隔3s检测一次
        sleep(3);

        // 取出线程池中任务的数量和当前线程的数量
        pthread_mutex_lock(&pool->mutexPool);
        int queueSize = pool->queueSize;
        int liveNum = pool->liveNum;
        pthread_mutex_unlock(&pool->mutexPool);

        // 取出忙的线程的数量
        pthread_mutex_lock(&pool->mutexBusy);
        int busyNum = pool->busyNum;
        pthread_mutex_unlock(&pool->mutexBusy);

        // 添加线程
        // 任务的个数>存活的线程个数 && 存活的线程数<最大线程数
        if (queueSize > liveNum && liveNum < pool->maxNum)
        {
            pthread_mutex_lock(&pool->mutexPool);
            int counter = 0;
            for (int i = 0; i < pool->maxNum && counter < NUMBER
                && pool->liveNum < pool->maxNum; ++i)
            {
                if (pool->threadIDs[i] == 0)
                {
                    pthread_create(&pool->threadIDs[i], NULL, worker, pool);
                    counter++;
                    pool->liveNum++;
                }
            }
            pthread_mutex_unlock(&pool->mutexPool);
        }
        // 销毁线程
        // 忙的线程*2 < 存活的线程数 && 存活的线程>最小线程数
        if (busyNum * 2 < liveNum && liveNum > pool->minNum)
        {
            pthread_mutex_lock(&pool->mutexPool);
```

```
            pool->exitNum = NUMBER;
            pthread_mutex_unlock(&pool->mutexPool);

            for (int i = 0; i < NUMBER; ++i)
            {
                pthread_cond_signal(&pool->notEmpty);
            }
        }
    }
    return NULL;
}

void threadExit(ThreadPool* pool)
{
    pthread_t tid = pthread_self();
    for (int i = 0; i < pool->maxNum; ++i)
    {
        if (pool->threadIDs[i] == tid)
        {
            pool->threadIDs[i] = 0;
            printf("threadExit() called, %ld exiting...\n", tid);
            break;
        }
    }
    pthread_exit(NULL);
}
```


测试代码

```
void taskFunc(void* arg)
{
    int num = *(int*)arg;
    printf("thread %ld is working, number = %d\n",
        pthread_self(), num);
    sleep(1);
}

int main()
{
    // 创建线程池
    ThreadPool* pool = threadPoolCreate(3, 10, 100);
    for (int i = 0; i < 100; ++i)
    {
        int* num = (int*)malloc(sizeof(int));
        *num = i + 100;
        threadPoolAdd(pool, taskFunc, num);
    }

    sleep(30);

    threadPoolDestroy(pool);
    return 0;
}
```