技术文档

类的介绍

Any
用于存储任意类型数据。例如，int、long、short 以及其他自定义类型。

设计思路

在 Any 中定义一个基类 Base，然后定义一个继承基类 Base 的派生类 Derived。派生类 Derived 使用模板并用模板参数定义成员变量。
Any 中的成员变量是 Base 类型的智能指针(std::unique_ptr)，这样保证 Any 内部值的唯一性。因此，在 Any 内部不会定义拷贝构造、赋值构造函数，而是会定义移动构造、移动赋值(默认)。

代码实现

```
class Any {
public:
    Any() = default;
    ~Any() = default;
    Any(const Any& v) = delete;
    Any& operator=(const Any& v) = delete;
    Any(Any&&) = default;
    Any& operator=(Any&&) = default;

    template<typename T>
    Any(T v) :data(std::make_unique<Derived<T>>(v)) {}

    template<typename T>
    T cast_(){
        Derived<T>* derived = dynamic_cast<Derived<T>*>(data.get());
        return derived->val_;
    }
private:
    class Base {
    public:
```

```
        virtual ~Base() = default;
    };
    template<typename T>
    class Derived : public Base {
    public:
        Derived(T val) :val_(val) {
        }
        ~Derived() {
        }
        T val_;
    };
    std::unique_ptr<Base> data;
};
```

Semaphore

多线程编程中，线程中实现同步机制的方式。

设计思路

使用互斥锁与条件变量进行设计。当信号量进行等待的时候，线程抢占锁并对成员变量自减，之后使用条件变量进行判断。当信号量增添之后通知其他线程。

代码实现

```
class Semaphore {
public:
    Semaphore(int val = 0) : val_(val) {}
    ~Semaphore() = default;

    void wait() {
        std::unique_lock<std::mutex> lock(mtx_);
        val_--;
        con_.wait(lock, [&]()->bool { return val_ > 0; });
        if (val_ > 0)
        {
            con_.notify_all();
        }
    }
    void post() {
        std::unique_lock<std::mutex> lock(mtx_);
```

```
        val_++;
        con_.notify_all();
    }
private:
    std::mutex mtx_;
    std::condition_variable con_;
    int val_;

};
```

Task

抽象类 Task 内部定义一个纯虚函数 void run()，用于子类进行重写实现多态。

代码实现

```
class Task {
public:
    Task();
    ~Task();
    void setResult(Result* res);
    void exec();
    virtual Any run() = 0;
    Result* res_;};
```

Result

多线程编程时用于异步接收线程处理的返回值。

设计思路

Result 内部会有 std::shared_ptr<Task>任务、信号量、以及 Any。Result 与 Task 会相互成为对方的成员变量，当 Task 在进行执行的时候 Result 会设置返回值 Any 并使用信号量同步。

代码实现

```
class Result {
public:
    Result(const std::shared_ptr<Task> task, bool isValid = true):task_(task),
isValid_(isValid){
```

```
        task_->setResult(this);
    }
    ~Result(){}
    Result(const Result&) = default;
    void setVal(Any val);
    Any get();
private:
    Any val_;
    bool isValid_;
    Semaphore sem_;
    std::shared_ptr<Task> task_;
};
```

Thread

线程类

设计思路

线程类在初始化的时候会为每个对象赋值一个 id 作为唯一标识；同时使用 std::bind 绑定线程函数作为线程类的成员对象；内部定义 void start()函数开启线程。

代码实现

```
class Thread {

public:
    using Func = std::function<void(int)>;

    Thread(Func f);
    ~Thread();

    void start();
    int  getId();

private:
    Func f;
    int id;
    static int threadId;

};
```

ThreadPool

线程池

设计思路

ThreadPool 有两种模式分别为 FIXED 和 CACHCED。
FIXED 模式下线程数量是固定的。

线程的启动

使用 void start(int threadNum), 启动子线程，首先按照所给的线程数绑定对应数量的线程函数并将其放入 Thread 中，按照 Thread 的 id 值已经 Thread 对象本身组成 Key、Value 将其组成一对放入 std::unorder_map<int,Thread> threadsMap_中。之后循环遍历 threadsMap_逐一对线程启动。

任务提交

主线程使用 Master-Slave 模式对任务进行分配，使用 void submitTask(std::make_shared<Task> task) 对任务进行提交，tasksNotEmpty_去唤醒等待线程。然后使用 Result 对返回值进行接收，假如 Result 使用 get()接收返回值 Any 的时候，会进行阻塞等待直到有返回值。当其他子线程运行线程函数的时候，会有一个 for(;;) 循环使得每个子线程一直运行，保持在任务队列 tasksQue_取任务。使用互斥锁 tasksQueMtx_维护任务队列的线程安全，获取锁之后，tasksnotEmpty_进行条件判断，之后从任务队列取出任务并维护剩余任务的成员变量 currentTasksSize_之后使用条件变量 tasksNotFull.notify_all 去通知提交任务的线程。

当任务队列已满的时候，提交任务线程不会一直卡在 tasksNotFull.wait 中，而是会设置等待时长，时间一到就进行返回，保证提交效率。

CACHED 模式

CACHED 模式是为了应对任务数急剧上升之后又下降的情况，所以应该添加线程等到线程空闲的时候又回收线程。当任务数急剧上升时，CACHED 模型下，会维护一个空闲线程 idleThreadSize_，并用该成员变量与当前任务数 currenntTaskSize_判断将当前线程数量维持在最大线程阈值之下。在这种情况下会创建新的线程并且启动去应对新的任务。当子线程在执行线程函数的时候，当某个线程一直空闲，会等待一定时间后会对该线程进行回收。具体办法就是在线程函数最开始执行的时候以及执行完成一次任务的时候进行时间记录，然后对当前时间与上一次时间做差，统计该时间是否达到指定的时间阈值。假如达到该时间阈值，就会依据线程 id 删除掉保存在 threadsMap_中的 Thread 对象并进行返回。

资源回收

ThreradPool 的析构函数中，会维护一个 threadsRunning_变量将其置为 false，之后会获取锁 taskQueMtx_,并且唤醒等待的线程，使用 exitCon_.wait 进行条件判断当前任务队列 tasksQue_ 是否为空，假如为空则执行完析构函数，否则等待阻塞并释放锁。子线程中被唤醒，进行循环当 threadsRunning_ 为 false,则会将该线程从线程 Map hreadsMap_中擦除并且返回。

并且整个线程池是需要任务执行完毕。所以在生存周期内，析构函数执行完毕之前是需要将所有任务执行完毕的。