ZLToolKit笔记10：对象池

2024年12月28日 21:29

以下是ResourcePool.h中的代码：

```
#if (defined(__GNUC__) && (__GNUC__ >= 5 || (__GNUC__ >= 4 && __GNUC_MINOR__ >=
9))) || defined(__clang__)             \
    || !defined(__GNUC__)
#define SUPPORT_DYNAMIC_TEMPLATE
#endif
template <typename C>
class ResourcePool_l;
template <typename C>
class ResourcePool;
template <typename C>
class shared_ptr_imp : public std::shared_ptr<C> {
public:
    shared_ptr_imp() {}
    /**
     * 构造智能指针
     * @param ptr 裸指针
     * @param weakPool 管理本指针的循环池
     * @param quit 对接是否放弃循环使用
     * Constructs a smart pointer
     * @param ptr Raw pointer
     * @param weakPool Circular pool managing this pointer
     * @param quit Whether to give up circular reuse

     * [AUTO-TRANSLATED:5af6d6a5]
     */
    shared_ptr_imp(
        C *ptr, const std::weak_ptr<ResourcePool_l<C>> &weakPool,
std::shared_ptr<std::atomic_bool> quit,
        const std::function<void(C *)> &on_recycle);
    /**
     * 放弃或恢复回到循环池继续使用
     * @param flag
     * Abandon or recover to continue using in the circular pool
     * @param flag

     * [AUTO-TRANSLATED:eda3e499]
     */
    void quit(bool flag = true) {
        if (_quit) {
            *_quit = flag;
        }
    }
private:
    std::shared_ptr<std::atomic_bool> _quit;
```

```
};
template <typename C>
class ResourcePool_l : public std::enable_shared_from_this<ResourcePool_l<C>> {
public:
    using ValuePtr = shared_ptr_imp<C>;
    friend class shared_ptr_imp<C>;
    friend class ResourcePool<C>;
    ResourcePool_l() {
        _alloc = []() -> C * { return new C(); };
    }
#if defined(SUPPORT_DYNAMIC_TEMPLATE)
    template <typename... ArgTypes>
    ResourcePool_l(ArgTypes &&...args) {
        _alloc = [args...]() -> C * { return new C(args...); };
    }
#endif // defined(SUPPORT_DYNAMIC_TEMPLATE)
    ~ResourcePool_l() {
        for (auto ptr : _objs) {
            delete ptr;
        }
    }
    void setSize(size_t size) {
        _pool_size = size;
        _objs.reserve(size);
    }
    ValuePtr obtain(const std::function<void(C *)> &on_recycle = nullptr) {
        return ValuePtr(getPtr(), _weak_self, std::make_shared<std::atomic_bool>
(false), on_recycle);
    }
    std::shared_ptr<C> obtain2() {
        auto weak_self = _weak_self;
        return std::shared_ptr<C>(getPtr(), [weak_self](C *ptr) {
            auto strongPool = weak_self.lock();
            if (strongPool) {
                //放入循环池  [AUTO-TRANSLATED:5ec73a78]
                //Put into circular pool
                strongPool->recycle(ptr);
            } else {
                delete ptr;
            }
        });
    }
private:
    void recycle(C *obj) {
        auto is_busy = _busy.test_and_set();
        if (!is_busy) {
            //获取到锁  [AUTO-TRANSLATED:6eb7c6e9]
            //Acquired lock
            if (_objs.size() >= _pool_size) {
                delete obj;
            } else {
                _objs.emplace_back(obj);
            }
            _busy.clear();
        } else {
```

```
                //未获取到锁  [AUTO-TRANSLATED:2b5e8adb]
                //Failed to acquire lock
                delete obj;
            }
        }
    C *getPtr() {
        C *ptr;
        auto is_busy = _busy.test_and_set();
        if (!is_busy) {
            //获取到锁  [AUTO-TRANSLATED:6eb7c6e9]
            //Acquired lock
            if (_objs.size() == 0) {
                ptr = _alloc();
            } else {
                ptr = _objs.back();
                _objs.pop_back();
            }
            _busy.clear();
        } else {
            //未获取到锁  [AUTO-TRANSLATED:2b5e8adb]
            //Failed to acquire lock
            ptr = _alloc();
        }
        return ptr;
    }
    void setup() { _weak_self = this->shared_from_this(); }
private:
    size_t _pool_size = 8;
    std::vector<C *> _objs;
    std::function<C *(void)> _alloc;
    std::atomic_flag _busy { false };
    std::weak_ptr<ResourcePool_l> _weak_self;
};
/**
 * 循环池，注意，循环池里面的对象不能继承enable_shared_from_this!
 * @tparam C
 * Circular pool, note that objects in the circular pool cannot inherit from
enable_shared_from_this!
 * @tparam C

 * [AUTO-TRANSLATED:e08caac8]
 */
template <typename C>
class ResourcePool {
public:
    using ValuePtr = shared_ptr_imp<C>;
    ResourcePool() {
        pool.reset(new ResourcePool_l<C>());
        pool->setup();
    }
#if defined(SUPPORT_DYNAMIC_TEMPLATE)
    template <typename... ArgTypes>
    ResourcePool(ArgTypes &&...args) {
        pool = std::make_shared<ResourcePool_l<C>>(std::forward<ArgTypes>
(args)...);
```

```
        pool->setup();
    }
#endif // defined(SUPPORT_DYNAMIC_TEMPLATE)
    void setSize(size_t size) { pool->setSize(size); }
    //获取一个对象，性能差些，但是功能丰富些  [AUTO-TRANSLATED:88b9a207]
    //Get an object, performance is slightly worse, but with more features
    ValuePtr obtain(const std::function<void(C *)> &on_recycle = nullptr) { return
pool->obtain(on_recycle); }
    //获取一个对象，性能好些  [AUTO-TRANSLATED:0032c7ca]
    //Get an object, performance is slightly better
    std::shared_ptr<C> obtain2() { return pool->obtain2(); }
private:
    std::shared_ptr<ResourcePool_l<C>> pool;
};

template<typename C>
shared_ptr_imp<C>::shared_ptr_imp(C *ptr,
                                  const std::weak_ptr<ResourcePool_l<C> >
&weakPool,
                                  std::shared_ptr<std::atomic_bool> quit,
                                  const std::function<void(C *)> &on_recycle) :
    std::shared_ptr<C>(ptr, [weakPool, quit, on_recycle](C *ptr) {
            if (on_recycle) {
                on_recycle(ptr);
            }
            auto strongPool = weakPool.lock();
            if (strongPool && !(*quit)) {
                //循环池还在并且不放弃放入循环池  [AUTO-TRANSLATED:96e856da]
                //Loop pool is still in and does not give up putting into loop
pool
                strongPool->recycle(ptr);
            } else {
                delete ptr;
            }
        }), _quit(std::move(quit)) {}
```

以上是ResourcePool.h中的全部内容了。

尝试从BufferRaw: :create函数入手:

```
BufferRaw::Ptr BufferRaw::create() {
#if 0
    static ResourcePool<BufferRaw> packet_pool;
    static onceToken token([]() {
        packet_pool.setSize(1024);
    });
    auto ret = packet_pool.obtain2();
    ret->setSize(0);
    return ret;
#else
    return Ptr(new BufferRaw);
#endif
}
```

这段代码展示了如何通过ResourcePool管理BufferRaw对象，它使用静态资源池packet_pool

存储BufferRaw对象，从而避免频繁的动态内存分配。

函数内static修饰变量，说明该变量只会初始化一次。初始化之后，在整个程序的生命周期该变量会一直存在，且只能在create这个函数中进行访问。

```
    static ResourcePool<BufferRaw> packet_pool;
```

这句代码定义了一个BufferRaw类型的静态资源池。所有线程都共享同一个这样的资源池。反过来我们来看这个BufferRaw类型的静态资源池ResourcePool是如何初始化的。在模板类ResourcePool实例化为真正的类之后，会调用其构造函数构造一个ResourcePool的对象：

```
     ResourcePool() {
        pool.reset(new ResourcePool_l<C>());
        pool->setup();
    }
```

pool是一个管理真正资源池对象的智能指针。

```
    std::shared_ptr<ResourcePool_l<C>> pool;
```

在构造函数中，new了一个C类型（这里是BufferRaw类）的ResourcePool_l对象，并用智能指针pool加以管理。然后调用pool->setup方法

```
    void setup() { _weak_self = this->shared_from_this(); }
```

使得_weak_self与这个管理Resource_l对象的share_ptr绑定（即与pool绑定）。避免多次shared_ptr引用资源池而无法正确析构释放资源。

之后：

```
     static onceToken token([]() {
        packet_pool.setSize(1024);
    });
```

当第一次调用BufferRaw::create时，token的构造函数将被执行。它是一个静态局部变量，这就保证了在整个程序的生命周期内该变量只会被初始化一次，即构造一次。
onceToken的设计作用是一种单次执行控制机制，其设计的目的是让某些操作只在需要时执行一次。
如下所示：

```
class onceToken {
public:
    using task = std::function<void(void)>;
    template<typename FUNC>
    onceToken(const FUNC &onConstructed, task onDestructed = nullptr) {
        onConstructed();
        _onDestructed = std::move(onDestructed);
    }
    onceToken(std::nullptr_t, task onDestructed = nullptr) {
        _onDestructed = std::move(onDestructed);
    }
    ~onceToken() {
        if (_onDestructed) {
            _onDestructed();
        }
    }
private:
    onceToken() = delete;
    onceToken(const onceToken &) = delete;
    onceToken(onceToken &&) = delete;
    onceToken &operator=(const onceToken &) = delete;
```

```
    onceToken &operator=(onceToken &&) = delete;
private:
    task _onDestructed;
};
```

构造函数接受一个可调用对象（如函数或lambda表达式），在构造时立即执行。onceToken的实例token是静态变量，所以其构造函数仅在token第一次被访问时执行，保证传入的函数[]() {......}仅调用一次。

C++11及以后的标准中，静态局部变量的初始化是线程安全的。编译器会确保多个线程同时访问静态局部变量时，初始化过程只有一个线程能成功执行，其他线程会等待初始化完成后再使用该变量。

之后：

```
    auto ret = packet_pool.obtain2();
```

首先调用ResourcePool中的obtain2方法：

```
    std::shared_ptr<C> obtain2() { return pool->obtain2(); }
```

它封装了真正从资源池中获取对像的过程：

```
std::shared_ptr<C> obtain2() {
        auto weak_self = _weak_self;
        return std::shared_ptr<C>(getPtr(), [weak_self](C *ptr) {
            auto strongPool = weak_self.lock();
            if (strongPool) {
                //放入循环池  [AUTO-TRANSLATED:5ec73a78]
                //Put into circular pool
                strongPool->recycle(ptr);
            } else {
                delete ptr;
            }
        });
    }
```

返回的是一个智能指针，他管理的对象由getPtr()方法返回，析构方法由第二个参数（匿名函数）指定（主要调用了recycle( )方法，决定是否放回资源池中还是真正的析构掉。

```
    void recycle(C *obj) {
        auto is_busy = _busy.test_and_set();
        if (!is_busy) {
            //获取到锁  [AUTO-TRANSLATED:6eb7c6e9]
            //Acquired lock
                // 池已满直接析构，池未满将对象放回池中。
            if (_objs.size() >= _pool_size) {
                delete obj;
            } else {
                _objs.emplace_back(obj);
            }
            _busy.clear();
        } else {
            //未获取到锁  [AUTO-TRANSLATED:2b5e8adb]
            //Failed to acquire lock
            delete obj;
        }
    }
    C *getPtr() {
```

```
        C *ptr;
        auto is_busy = _busy.test_and_set();
        if (!is_busy) {
            //获取到锁 [AUTO-TRANSLATED:6eb7c6e9]
            //Acquired lock
            if (_objs.size() == 0) {
                ptr = _alloc();    //如果资源池中没有可用对象，则创建一个
            } else {
                ptr = _objs.back();   //资源池中有可用对象，则返回一个可用对象（直接复用）
                _objs.pop_back();
            }
            _busy.clear();
        } else {
            //未获取到锁 [AUTO-TRANSLATED:2b5e8adb]
            //Failed to acquire lock
            ptr = _alloc();
        }
        return ptr;
    }
```

有关原子锁的内容参见笔记：C++：std::atomic&&std::atomic_flag

```
    std::atomic_flag _busy { false };
```