目录介绍

1.对外封闭的公共接口

工程实践中，总会出现一些common的地方，这些common设计是为了满足整个大的工程架构里，可以被其各个子模块共同调用，这些子模块只需要根据本身的需求，调用公共接口，返回自身需要的结果即可。

1.1.工程作用

公共接口是可以被所有子模块调用的，子模块中传入本地需要的特定参数，调用公共接口，由传入的参数决定返回的结果类型，特定参数一般设计为枚举类型，返回的结果是根据枚举的值进行判定输出，当然，也可以建立一个静态表，根据静态表的关系进行对外输出。它的工程作用主要有：

> 1，公共接口设计在common的仓库内，对外开放一个接口，满足开闭原则
>
> 2，避免子模块自己实现这个接口的繁琐和复杂，以及给代码带来的臃肿性
>
> 3，代码设计优雅，符合工程架构的规范，是中级工程师必备的技能素养

1.2.框架实现

公共接口总的架构实现，可以被嵌套在任何工程实践中，不管是算法，还是软件，都可以将其放在common的地方，等待子模块的调用。

1.2.1.静态外部接口

```
//1，对外只是开放这一个接口，所有的实现都在接口内部的调用中
//知识点：静态函数
static std::string getXX_entry(const XXType& type, int idx = -1)
{
    return XXManager::getInstace()->getXX(type,idx);
}

namespace project_XX{
    //知识点：enum用法
    typedef enum XXType{
        AA = 1,
        BB
    }XXTYPE;

    class XXManager{
        public:
           //知识点：最基本的单例模式
            static XXManager* getInstace()
            {
                return instance_;
            }

            std::string getXX(const XXTYPE& type, int idx) const;

        private:
            //知识点：禁止类的构造方式
            XXManager(const XXManager&) =delete;
            XXManager& operator=(const XXManager&) =delete;
```

```
            bool init();

            XXManager():is_init_(false){
                //这种写法也值得推荐，构造的时候直接把init也做了，并且返回init结果
                is_init_ = init();
            }

            bool is_init_;
            //知识点：静态成员对象
            static XXManager* instance_;

    };
}

namespace project_XX{

    XXManager* XXManager::instance_ = new (std::nothrow) XXManager();

    std::string XXManager::getXX(const XXTYPE& type, ind idx) const
    {
        if(!is_init_)
        {
            return " ";
        }

        /*
        主要逻辑自己实现

        */

        return " XX ";

    }

    bool XXManager::init()
    {
        /*

        1， 读取配置文件
        2，所有变量的一个初始化
        3， 自己想要实现的
        */
        return true;
    }

}

//举例说明
class AA{

    public:

        bool process(){
```

```
            //直接调用对外封装的接口
            std::string aa = getXX_entry(AA,10);
            /*


            */
        }

};


class BB{

    public:

        bool process(){
                //直接调用对外封装的接口
                std::string bb = getXX_entry(AA,10);
                /*


                */
            }
};
```

1.2.2.嵌套接口封装

```
#include <string>
#include <vector>
#include <unordered_map>
#include <iostream>

/*
@ 嵌套对外接口实现

1，封装一层静态单例的接口调用
2，由namespace 进行包装

公众号：码出名企路
工作中是怎么写C++的系列视频和代码，文档，入圈一起学习
*/

//this_node.h

//知识点：using的用法
using V_string = std::vector<std::string>;
using M_string = std::unordered_map<std::string,std::string>;

//知识点：命名空间用法
```

```
namespace this_node
{
    const std::string& getName();

    const std::string& getNamespace();
}

//this_node.cpp
namespace this_node
{
    class ThisNode{
        std::string name_;
        std::string namespace_;

        ThisNode() : name_("empty"),namespace_("namespace") {}

        public:
            //比较与上一个例子的单例的实现，返回一个静态引用对象
            static ThisNode& instance()
            {
                static ThisNode singleton;
                return singleton;
            }

            const std::string& getName() const
            {
                std::cout<<"get name: "<<name_<<std::endl;

                return name_;
            }

            const std::string& getNamespace() const
            {
                std::cout<<"get Namespace: "<<namespace_<<std::endl;
                return namespace_;
            }

            void init(const std::string& name, const M_string& remappings, uint32_t
options);
    };

    //对外包装了一层调用，外部只需要调用这个接口
    const std::string& getNamespace()
    {
        return ThisNode::instance().getNamespace();
    }

    //对外接口
    void init(const std::string& name, const M_string& remappings, uint32_t options)
    {
        ThisNode::instance().init(name, remappings, options);
    }
```

```
    void ThisNode::init(const std::string& name, const M_string& remappings, uint32_t
options)
    {
        std::cout<<"init ThisNode "<<std::endl;
    }
}


class Test{
    public:

        void test(){

            this_node::getNamespace();//外部只需要调用这个就行了

            this_node::ThisNode::instance().getName();
        }
};

int main()
{
    Test test_;
    test_.test();
}
```

1.3.知识点拆分

从写一个公共接口中，我们学到了一些C++知识点，正是这些知识点的组合，才得以使我们能够在工作中实现我们想要的工程架构。知识点主要涉及到：

1，静态函数

2，单例模式

3，using用法

4，namespace用法

5，静态成员对象

6，禁止构造函数delete

下面我们从框架实现的细节中具体讲解以上涉及的知识点，掌握了C++的基本语法的基础下，将以上知识点运用熟练，我们可以更好的发挥学以致用的优势。

1.3.1.静态函数和静态成员函数及对象

第一部分：《写一个公共接口》工程涉及到的代码如下：

```
static ThisNode& instance()
{
    static ThisNode singleton;
    return singleton;
}
```

第二部分：静态函数和静态成员函数及对象详细讲解如下：

在C++中，静态函数和静态成员函数是类的成员函数，声明时用关键字"static"来修饰。它们相对于普通成员函数的不同在于它们不需要依赖于类的对象来访问，因此可以通过类名访问。

静态函数是属于类的函数，它不会访问或修改任何与类实例相关的任何数据。因此，即使不创建类的实例，也可以使用静态函数。静态函数的定义形式为：

```
class MyClass{
public:
    static void myStaticFunc(){
        // 静态函数的函数体
    }
};
```

静态成员函数是属于类的成员函数，它可以访问所有静态成员变量和静态成员函数。静态成员函数的定义形式为：

```
class MyClass{
public:
    static int num;
    static void myStaticFunc(){
        // 静态成员函数的函数体
        cout << "num=" << num << endl;
    }
};

int MyClass::num = 0;
```

静态成员变量是指属于类的而不是属于类的实例的变量。它们可以在类声明中声明，并在类的外部进行初始化。静态成员变量的定义形式为：

```
class MyClass{
public:
    static int num;
};

int MyClass::num = 0;
```

要使用静态函数和静态成员变量，可以使用以下代码：

```
MyClass::num = 10;  // 静态成员变量的初始化
MyClass::myStaticFunc();  // 调用静态成员函数
```

静态函数和静态成员变量通常被用来存储和共享在类中所有实例都需要访问的数据，或者用于实现某些与类实例无关的公共操作。

1.3.2.using和namespace用法

第一部分：《写一个公共接口》工程涉及到的代码如下：

```
//知识点：using的用法
using V_string = std::vector<std::string>;
using M_string = std::unordered_map<std::string,std::string>;

//知识点：命名空间用法
namespace this_node
{
    const std::string& getName();

    const std::string& getNamespace();
}
```

第二部分：using和namespace在C++中的用法详细讲解如下：

在C++中，using和namespace是用于解决命名冲突的关键字。

using关键字被用来在程序中引入一个已经被定义的命名空间中的名称或者描述符。使用using关键字可以使得程序更加简洁明了，避免重复的代码。

例如，可以使用以下语句来使得std命名空间中的cout和endl可用：

```
using std::cout;
using std::endl;
```

这样，在程序中使用cout和endl时，就可以直接使用无需在前面添加std::前缀了，如下所示：

```
cout << "Hello World!" << endl;
```

namespace关键字被用来在程序中定义命名空间。使用namespace关键字可以在程序中定义属于自己的命名空间，从而避免命名空间冲突，使得程序的名称更加独特。

例如，可以使用以下语句在程序中定义一个命名空间：

```
namespace TestSpace {
    int num;
    void testFunction();
}
```

这样，在程序中就可以使用TestSpace::num和TestSpace::testFunction()来访问该命名空间中的变量和函数了。

需要注意的是，在使用using时要谨慎，因为有可能会引入意料之外的名称冲突问题。在命名空间冲突方面，可以通过使用命名空间限定符（::）解决不同命名空间中重名的对象或函数。

1.3.3.单例模式的实现技巧

第一部分：《写一个公共接口》工程涉及到的代码如下：

```
//比较与上一个例子的单例的实现，返回一个静态引用对象
static ThisNode& instance()
{
  static ThisNode singleton;
  return singleton;
}
```

第二部分：C++工程中常用的单例模式实现细节讲解如下：

单例模式是一种常用的设计模式，它保证了类只能有一个实例，并提供了对该实例的全局访问点。在C++中，单例模式的实现技巧主要包括以下几点：

1.在类中定义一个静态的私有实例，用于表示该类的唯一实例对象：

```
class Singleton{
private:
    static Singleton* instance;
    Singleton() {}
public:
    static Singleton* getInstance();
};
```

2.使用一个静态的公有方法，用于返回该类的唯一实例对象：

```
Singleton* Singleton::getInstance() {
    if (instance == nullptr) {
        instance = new Singleton();
    }
    return instance;
}
```

3.在静态公有方法中判断唯一实例对象是否已经被创建，如果已经被创建，则直接返回该实例对象；如果尚未被创建，则先创建实例对象，然后返回该实例对象。

4.可以使用私有的构造函数，以防止在类的外部进行实例化操作。

5.可以使用互斥锁（mutex）或者双检锁（double-check locking）等方式来确保线程安全。

综上所述，单例模式的实现技巧主要是在类中定义静态私有实例、使用静态公有方法来返回该实例、私有构造函数、线程安全等方面。注意，在实现单例模式时，需要时刻考虑线程安全问题，以确保程序的正确性。

1.3.4.禁止构造函数的几种写法

第一部分：《写一个公共接口》工程涉及到的代码如下：

```
XXManager(const XXManager&) =delete;
XXManager& operator=(const XXManager&) =delete;
```

第二部分：我们知道在工程实践中，经常需要规定一些构造函数不能再创建实例，这样的写法很多种。如下所示：

在C++中，禁止构造函数的几种写法主要有以下几种：

1. 私有构造函数

将构造函数声明为private可确保类的实例只能在类的内部创建，无法在类的外部进行实例化。私有构造函数通常用于实现单例模式，以确保只有一个实例存在。例如：

```
class Singleton {
public:
    static Singleton& getInstance() {
        static Singleton instance;
        return instance;
    }
private:
    Singleton() {}  //私有构造函数
};
```

2. 抽象类

抽象类不允许直接实例化对象，通常用于提供接口和基类。一个类如果包含至少一个纯虚函数，则被称为抽象类。例如：

```
class Shape {
public:
    virtual void draw() = 0;  //纯虚函数
};
```

3. delete关键字

C++11引入了delete关键字，用于删除一些不需要或不合法的函数。使用delete可以将函数声明为已删除，从而禁止该函数的调用。例如：

```
class MyClass {
public:
    MyClass() = delete;  //禁用默认构造函数
    MyClass(const MyClass&) = delete;  //禁用拷贝构造函数
};
```

4. 派生构造函数

在C++11中，可以使用派生构造函数来控制基类和成员变量，从而限制对象的构造方式。派生构造函数允许使用基类和成员变量的默认构造函数。例如：

```
class MyClass {
public:
    MyClass() {}
private:
    explicit MyClass(int x) {}
};

class Derived : public MyClass {
public:
    using MyClass::MyClass;  //继承基类构造函数
    Derived(int x, int y) : MyClass(x), m_y(y) {}
private:
    int m_y;
};
```

通过上述写法可以通过using关键字来继承基类构造函数，然后将MyClass(int)构造函数声明为私有的，从而禁止外部使用该构造函数。

以上是禁止构造函数的几种常用写法，通过这些方法可以实现对类的构造过程进行约束，从而提高程序的安全性和稳定性。

1.4.本节大作业

本节所讲说的工程种使用公共接口，是一个可扩展的模板框架，在实际的工作种，当我们遇到一个工程需要加载配置文件的时候，可以考虑使用此工程框架。具体来说就是：

1，配置文件的加载和获取独立成一个单例类，在这个单例类种管理所有的配置，主要实现配置文件对象的构造，文件的加载和获取方法，还可以根据工程需要自定义一些接口，比如配置文件的路径，属性等等

2，对外部或用户再封装一层静态函数，这些静态函数是对上述单例管理方法的进一步封装，保持软件设计模式种开放封闭的原则，对外只开放一个static接口，具体的内部实现只放在单例中去做，单例中的接口可以任意扩展，实现也可以任意扩展，对外是不可见的。static函数接口可以参照1.2节框架实现的1.2.1.静态外部接口的写法。

请用本节学习的知识，结合自己的工程或者学校里的项目，完成一个管理配置文件的工程，并书写一个完整的例子进行测试。

2.工厂与单例模式结合模板

在学校里或者工作中，你偶尔会翻翻设计模式相关的书籍，想必这个大多数同学都有学习，学习设计模式的初衷就是想改造一下现有的工程架构，封装类的设计，方便后续扩展和维护。其中，单例模式和工厂模式在工作中应用的比较广泛，而将两者结合起来又是什么样子呢？我们拭目以待。

2.1.工程作用

众所周知，软件设计模式中的单例模式是仅允许外部构造一个实例对象，所有的实现均有一个对象进行管理，这可以减少多次创建对象的资源消耗。工厂模式则是为了方便扩展，定义不同的工厂，对生产不同的产品。两者分开使用的例子很多，结合使用是如何做到的，它有什么工程作用，具体列出了几条：

> 1，单例对象管理工厂类的构建，只构造一个单例工厂，防止工厂与产品的错位；
>
> 2，两者结合，可以将工厂与产品对象用map哈希表的形式在编译时候构造出来，通过单例的方法返回对应的工厂类，不用在运行中构造，加快运行速度；
>
> 3，工厂模式体现了继承，封装多态的实现，真正实现面向对象的C++编程。

2.2.框架实现

本节基于上述工厂和单例的作用，实现两个在算法工程中广泛使用的架构，其中之一是区别于第一节简单单例的实现方法，本节用宏定义实现一个单例模式，给读者一种高大上的感觉，也体现代码的编码功底。另外实现了编译时期工厂和单例注册的结合模板，这个模板可以被大多数工程架构使用。

2.2.1.宏定义实现的单例模式

用宏定义实现一个单例模式，并且在类编译的时候已经将其实例化了，也就是说后面在引用被实例化类的成员函数或者成员变量的时候，不必再重新Instance，直接将其当作单例类来用就行。

```
#include <iostream>
```

```
#include <memory>
#include <mutex>
#include <type_traits>
#include <utility>

//知识点：宏定义函数， 模板结构体/类
#define DEFINE_TYPE_TRAIT(name, func)                     \
  template <typename T>                                   \
  struct name {                                           \
    template <typename Class>                             \
    static constexpr bool Test(decltype(&Class::func)*) { \
      return true;                                        \
    }                                                     \
    template <typename>                                   \
    static constexpr bool Test(...) {                     \
      return false;                                       \
    }                                                     \
                                                          \
    static constexpr bool value = Test<T>(nullptr);       \
  };                                                      \
                                                          \
  template <typename T>  \
  constexpr bool name<T>::value;
  //知识点：constexpr关键字

DEFINE_TYPE_TRAIT(HasShutdown, Shutdown)

//知识点：std::enable_if用法
template <typename T>
typename std::enable_if<HasShutdown<T>::value>::type CallShutdown(T *instance) {
  instance->Shutdown();
}

template <typename T>
typename std::enable_if<!HasShutdown<T>::value>::type CallShutdown(
    T *instance) {
  (void)instance;
}


#undef UNUSED
#undef DISALLOW_COPY_AND_ASSIGN

#define UNUSED(param) (void)param
//知识点：禁止构造函数，在第1节中已经学习
#define DISALLOW_COPY_AND_ASSIGN(classname) \
  classname(const classname &) = delete;    \
  classname &operator=(const classname &) = delete;
//知识点：std::call_once
#define DECLARE_SINGLETON(classname)                                \
 public:                                                            \
  static classname *Instance(bool create_if_needed = true) {        \
    static classname *instance = nullptr;                           \
```

```
    if (!instance && create_if_needed) {                                  \
      static std::once_flag flag;                                         \
      std::call_once(flag,                                                \
                     [&] { instance = new (std::nothrow) classname(); }); \
    }                                                                     \
    return instance;                                                      \
  }                                                                       \
                                                                          \
  static void CleanUp() {                                                 \
    auto instance = Instance(false);                                      \
    if (instance != nullptr) {                                            \
      CallShutdown(instance);                                             \
    }                                                                     \
  }                                                                       \
                                                                          \
 private:                                                                 \
  classname();                                                            \
  DISALLOW_COPY_AND_ASSIGN(classname)



class ClassWithShutdown {
    public:
        void Shutdown() {

            std::cout<<"shut down"<<std::endl;
            set_foo(1);
        }

        static int foo() { return foo_; }
        static void set_foo(int val) { foo_ = val; }

    private:
        static int foo_;
       //关键点：编译时候已经将单例进行注册
        DECLARE_SINGLETON(ClassWithShutdown)
};

int ClassWithShutdown::foo_ = 0;
inline ClassWithShutdown::ClassWithShutdown() {}

class ClassWithoutShutdown {
    private:
        DECLARE_SINGLETON(ClassWithoutShutdown)
};

inline ClassWithoutShutdown::ClassWithoutShutdown() {}

int main()
{

   std::cout<< HasShutdown<ClassWithShutdown>::value << std::endl;
   std::cout<< HasShutdown<ClassWithoutShutdown>::value << std::endl;
```

```
    std::cout<< ClassWithShutdown::foo() << std::endl;
    ClassWithShutdown::CleanUp();

    ClassWithoutShutdown::CleanUp();

}
```

2.2.2.编译注册的单例与工厂结合

以往我们对工厂模式的做法是，构造工厂和产品对象，两者一一对应，高级一点是手动写一个静态map表，将两者的关系维护起来，这样在运行的时候可以直接用工厂关键字去构造对应的产品对象。本节带来的写法是map不是静态建立，不可更改的，而是在编译时候自动创建，无需更改，快捷构造。结合单例模式，完美实现一个单例与工厂模式结合使用的工程架构模板。

```
#include<iostream>
#include<memory>
#include<thread>
#include<vector>
#include<ros/ros.h>
#include<unordered_map>

/*
1，单例
2，工厂
3，编译时注册
4，智能指针
5，宏定义函数
6，using / typedef
*/

struct InitOptions
{
    std::string conf_name;
    std::string topic_type;
};

class InitManager;

//知识点：using已经在第1节讲过
using  InitManagerPtr = std::shared_ptr<InitManager> ;
using  M_InitOptions = std::unordered_map<std::string, InitOptions>;
using  V_InitOptions = std::vector<InitOptions>

class BaseMode;

class InitManager
{
```

```
public:
    //知识点：默认构造写法
    InitManager()= default;
    ~InitManager() = default;

     //知识点：类中嵌套模板类，注册工厂
    template<typename Inherit>
    class Register_t{
        public:
            Register_t(const std::string& key){

                InitManager::getInstance_1()->map_.emplace(key,new Inherit());
            }
    };
    //知识点：智能指针std::unique_ptr
    std::unique_ptr<BaseMode> GetUniqueDevice(const std::string& type);
    BaseMode* Produce(const std::string& type);

    //普通单例第1节已经学习
    static InitManager& getInstance_0(){
        static InitManager instance;
        return instance;
    }

    static InitManager* getInstance_1(){

    }
    //加锁单例，返回指针，注意与下面做区分
    static InitManagerPtr getInstance_2(){
        if (m_pInstance == nullptr){
            std::lock_guard<std::mutex> lk(m_mutex);

            if (m_pInstance == nullptr){
                m_pInstance = std::shared_ptr<InitManager>(new InitManager);
            }
        }

        return m_pInstance;
    }
    //call_once单例，返回const指针引用
    static const InitManagerPtr& getInstance_3()
    {
        static InitManagerPtr instance = nullptr;
        static std::once_flag flag;
        std::call_once(flag, [&] { instance = std::make_shared<InitManager>();});
        return instance;
    }

    bool init(const std::string& conf_file);

    const M_InitOptions& config_prop_map()
    {
        return config_init_options_m;
```

```
    }

    const V_InitOptions& node_configs_map()
    {
        return config_init_options_v;
    }


private:

    InitOptions params_;

    M_InitOptions                   config_init_options_m_;
    V_InitOptions                   config_init_options_v_;

    static InitManagerPtr m_pInstance;
    static std::mutex m_mutex;
    std::map<std::string, InitManager*> map_; //{"A_Mode", A_MODE*}

    //知识点：宏定义的 ##使用
    #define MODE_FACTORY(T) v2xos_reg_func_str_##T##_
        /**
        * @brief      用于注册指定设备
        * @param[in]  T 具体设备子类型
        * @param[in]  key 注册的设备名字
        */
    #define MODE_FACTORY_FACTORY(T, key) static InitManager::Register_t<T> MODE_FACTORY(T)
(key);
};

InitManagerPtr InitManager::m_pInstance = nullptr;
std::mutex InitManager::m_mutex;

bool InitManager::init(const std::string& conf_file)
{
    /*

    @...................

    1, config_init_options_m_
    2, config_init_options_v_
    3, params_

    */
    return true;
}

std::unique_ptr<BaseMode> InitManager::GetUniqueDevice(const std::string& type)
{
    return std::unique_ptr<BaseMode>(Produce(type));
}

BaseMode* InitManager::Produce(const std::string& type)
```

```
{
    if (map_.find(type) == map_.end())
    {
        return nullptr;
    }
    return map_[type];
}

class BaseMode
{
    public:
        virtual bool int() = 0;
}

class A_Mode: BaseMode
{
    public:
        virtual bool int() override{
            std::cout<<"A_Mode: init"<<std::endl;
        }
    //宏定义编译时候注册了单例，构造了工厂map
    MODE_FACTORY(A_Mode,"A_Mode");
}

int main()
{

   if (!ConfigManager::getInstance_3()->init())
   {

   }
   //单例对象的方法获取工厂实例
   std::unique_ptr<BaseMode> mode_name =
    ConfigManager::getInstance_3()->GetUniqueDevice("A_Mode");

   mode_name->init();
}
```

2.3.知识点拆分

在本节的工厂模式和单例模式的结合使用中，我们使用了很多C++的知识点，这些知识点也是在工程实践中经常需要用到的，能够熟练掌握并且快速应用到工作中，是我们当下最应该关注和学习的事情。本节主要设计的知识点如下：

1，宏定义函数传参

2，结构体模板和类模板

3，constexpr和std::enable_if关键字

4，std::call_once实现的单例

5，默认构造函数的作用及写法

6，类中嵌套类

7，智能指针std::unique_ptr

8，宏定义的##符合用法

下面我们就对框架实现的细节进行知识点的拆分，详细讲解以上知识点是如何在工程实践中应用并发挥它们价值的，只有在实践中才能更好的学习，并快速高效的学习。

2.3.1.宏定义函数传参

第一部分：本节工程框架里涉及到的应用如下：

```
#define DEFINE_TYPE_TRAIT(name, func)
```

第二部分：详细讲解宏定义是如何代替函数，并且进行形参传递的。

在C++中，宏定义函数传参可以使用#define指令来定义简单的宏（例如常见的宏定义），也可以使用宏函数（带参宏）来传递参数，这些宏定义函数可以在预处理器中使用，可以解决一些编译期间的问题。

使用宏函数的一般形式是：

```
#define macro(param1, param2, ...) expression
```

其中，param1、param2、...为参数列表，expression为表达式。

宏函数是一个简单的字符串替换，当编译器遇到宏调用时，编译器会将该宏调用替换为宏函数的实现代码，然后再编译程序。

例如，以下是一个将两个整数相加的宏函数的示例：

```
#define ADD(a, b) ((a) + (b))

int main(){
    int x = 10, y = 20;
    int sum = ADD(x, y);
    return 0;
}
```

在预处理时，编译器会将该代码替换为以下代码：

```
int main(){
    int x = 10, y = 20;
    int sum = ((x) + (y));
    return 0;
}
```

需要注意的是，使用宏函数的时候要注意传参的方式。由于宏函数仅仅是一个简单的字符串替换，所以在传参时要确保括号的正确使用，以避免出现意料之外的错误。例如，上述加法宏函数中，在进行加法运算时要确保传入参数被括号括起来，以避免运算顺序不正确，导致计算结果错误。

此外，在使用宏函数式需要注意宏函数调用时参数的类型，如果传的是一个复杂的类型，那么宏函数可能会出现一些没有预料到的问题，因为宏函数的参数并不会像程序中传入函数时那样得到严格控制。

2.3.2.类模板和结构体模板

第一部分：本节工程实践代码中涉及使用结构体模板的地方如下：

```
 template <typename T>                                    \
  struct name {                                            \
    template <typename Class>                              \
    static constexpr bool Test(decltype(&Class::func)*) { \
      return true;                                         \
    }
   }
```

第二部分：下面我们将详细讲解类模板的语法及实现细节，这个务必掌握，工程中经常用到。

C++中的类模板和结构体模板是可重用的代码模板，可以用来创建具有相似特征的不同对象。

类模板的定义与普通类的定义类似，只是在类名后加了一对尖括号，并在其中指定了一个或多个类型参数。

例如，下面的代码定义了一个实现了加法和减法功能的类模板：

```
template <typename T>
class Calculator {
  T value;
public:
  Calculator(T v) : value(v) {}
  T add(T v) { return value + v; }
  T subtract(T v) { return value - v; }
};
```

上面的代码中，T是类型参数。在定义类时，我们并没有指定T的具体类型。相反，每当我们使用这个类时，可以将任意类型作为T的类型参数传递进去。

下面的代码演示了如何使用Calculator模板类来进行加法和减法运算：

```
Calculator<int> intCalc(2);
cout << intCalc.add(3) << endl; // 输出 5
cout << intCalc.subtract(1) << endl; // 输出 1

Calculator<double> doubleCalc(2.4);
cout << doubleCalc.add(3.4) << endl; // 输出 5.8
cout << doubleCalc.subtract(1.2) << endl; // 输出 1.2
```

在上面的代码中，我们使用了int和double两种类型作为Calculator的类型参数，并对Calculator对象进行了加法和减法运算。

与类模板类似，结构体模板也是用尖括号来指定一个或多个类型参数的。下面的代码定义了一个结构体模板：

```
template<typename T>
struct Pair {
  T first, second;
  Pair(T f, T s) : first(f), second(s) {}
};
```

上述代码创建了一个名为Pair的结构体模板，其中包含两个成员变量first和second，它们都是类型为T的变量。这个结构体还提供了一个构造函数，用于初始化这两个成员变量。

以下是如何使用这个结构体模板的示例代码：

```
Pair<int> p1(1, 2);
Pair<double> p2(2.5, 5.7);
```

在第一行代码中，我们使用int作为类型参数来创建一个Pair对象。在第二行中，我们使用double作为类型参数来创建一个Pair对象。

这些示例说明了如何在C++中使用类模板和结构体模板来创建可重用的代码。这些模板大大提高了代码的可扩展性、复用性和代码质量。

2.3.3.constexpr关键字和std::enable_if模板类

第一部分：constexpr关键字和std::enable_if在本节中涉及的应用有：

```
constexpr bool name<T>::value;

//知识点：std::enable_if用法
template <typename T>
typename std::enable_if<HasShutdown<T>::value>::type CallShutdown(T *instance) {
  instance->Shutdown();
}
```

第二部分：我们详细分别讲解一下constexpr的作用，以及std::enable_if想表达的含义

下面是constexpr要表达的意思：

`constexpr`是C++11中引入的关键字，它用于声明函数或变量的值可以在编译时计算出结果，并且如果编译器能够在编译时计算出结果，它还可以用于替代`const`常量。

对于函数，`constexpr`声明可以指示编译器在编译时进行函数求值的优化。这些函数必须是在编译时可计算的，这意味着它们必须是短小精悍，没有循环、递归或运行时判断。

以下是一个使用`constexpr`的函数的例子：

```
constexpr int factorial(int n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}
```

这段代码定义了一个递归的`factorial`函数，计算`n!`的值。由于递归函数的性质，在编译时无法计算此函数的值。但是，由于它被标记为`constexpr`，如果我们调用`factorial`并将结果赋值给`constexpr`变量，编译器会尝试在编译时直接计算结果，而不是在运行时进行计算。

另外，`constexpr`变量就如同`const`变量一样，但是在编译时它们的值是已知的，这意味着可以像常量一样使用它们，而不需要进行运行时计算。以下是一个使用`constexpr`变量的例子：

```
constexpr int max(int a, int b) {
    return a > b ? a : b;
}

int main() {
    constexpr int x = 10;
    constexpr int y = 20;
    constexpr int z = max(x, y);
    return z;
}
```

这段代码定义了一个`max`函数，返回两个整数中的最大值。然后，它定义了三个`constexpr`变量`x`、`y`和`z`，其中`x`和`y`都是常量，而`z`是调用`max`函数计算出的值，由于`x`和`y`都是`constexpr`变量，编译器可以在编译时计算出`max(x, y)`的值，所以`z`也是可编译时常量。

总之，`constexpr`关键字可以使我们更轻松地编写可预测、高效、可重用和安全的代码，提高我们的程序性能和可读性。

下面是std::enable_if要表达的意思：

`std::enable_if`是C++标准库中定义的一个模板类，它用于根据某些条件，在编译时决定是否启用或禁用模板函数或类。

在函数模板中，我们通常使用`std::enable_if`来控制模板参数的类型，以便只有满足某种条件时才会启用该函数。以下是一个使用`std::enable_if`的示例：

```
#include <type_traits>

template<typename T>
typename std::enable_if<std::is_integral<T>::value, bool>::type
is_odd(T i) { return bool(i % 2); }

int main() {
    static_assert(is_odd(1), "1 is odd");
    static_assert(!is_odd(2), "2 is not odd");
    return 0;
}
```

在上面的代码中，`is_odd`是一个模板函数，它只在传递的参数是整数类型时才会被启用。它使用`std::enable_if`模板来检查模板参数是否是整数类型。如果是整数类型，则返回`bool`值`true`或`false`表示输入的整数是否为奇数。

在`std::enable_if`中，第一个模板参数是一个布尔表达式，它可以使用`std::is_integral`模板来检查模板参数是否是整数类型。第二个模板参数指定了`enable_if`中的`type`成员类型，它的值取决于第一个参数，通常是一个返回类型的模板别名。

通常，`std::enable_if`的第一个参数是一个表达式，这个表达式最终会得到一个布尔类型的值。如果这个值为`true`，那么`std::enable_if`的`type`类型定义将存在并且等于指定的类型，否则它将不存在。

除了函数模板中，`std::enable_if`也经常用于类模板中，以实现对模板参数的类型检查，从而保证代码的正确性和安全性。

2.3.4.std::call_once实现的单例

第一部分：本节工程代码中使用std::call_one实现单例涉及的地方是：

```
    //call_once单例，返回const指针引用
     static const InitManagerPtr& getInstance_3()
     {
         static InitManagerPtr instance = nullptr;
         static std::once_flag flag;
         std::call_once(flag, [&] { instance = std::make_shared<InitManager>();});
         return instance;
     }
```

第二部分：std::call_once具体细节讲解：

在 C++11 标准中，我们可以使用`std::call_once`和`std::once_flag`来实现线程安全的单例模式。

下面是一个简单的使用`std::call_once`的单例实现：

```
#include <iostream>
#include <mutex>
```

```
class Singleton {
public:
    static Singleton& getInstance() {
        static Singleton instance;   // 静态局部变量实现单例
        return instance;
    }

private:
    Singleton() {
        std::cout << "Singleton constructor called." << std::endl;
    }

    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
};

int main() {
    auto& s = Singleton::getInstance();
    return 0;
}
```

在上面的代码中，`getInstance`函数返回一个静态局部变量实例，确保只有一个实例被创建。但是，在多线程情况下，这种实现可能不安全。

为了确保线程安全，可以使用`std::call_once`函数，如下所示：

```
#include <iostream>
#include <mutex>

class Singleton {
public:
    static Singleton& getInstance() {
        static Singleton instance;            // 静态局部变量实现单例
        static std::once_flag onceFlag;       // std::once_flag 标记初始化是否完成
        std::call_once(onceFlag, []() {       // 使用 std::call_once 只执行一次
            std::cout << "Singleton constructor called." << std::endl;
        });
        return instance;
    }

private:
    Singleton() = default;
};

int main() {
    auto& s = Singleton::getInstance();
    return 0;
}
```

在这个实现中，我们使用了`std::once_flag`来标记初始化是否完成，并且在`getInstance`中使用`std::call_once`和 lambda 表达式，使得初始化操作只会被执行一次。这种实现方式能够保证在多线程情况下的线程安全。

需要注意，这里的 Singleton 类被禁止拷贝构造和赋值，以避免多个实例的产生。

总之，使用`std::call_once`和`std::once_flag`能够避免多线程情况下的竞态条件，从而实现线程安全的单例模式。

2.3.5.默认构造函数的作用

第一部分：本节工程实践代码中涉及默认构造的地方如下：

```
    //知识点：默认构造写法
    InitManager()= default;
    ~InitManager() = default;
```

第二部分：如何使用默认构造，默认构造在代码中发挥的作用是什么，具体细节如下：

在C++中，默认构造函数（default constructor)是一种无参构造函数，如果我们没有为一个类定义任何构造函数，则编译器会自动生成默认构造函数，我们也可以显示对其进行定义。默认构造函数可以用来初始化成员变量以及为派生类构造器提供构造函数。

默认构造函数的作用可以总结如下：

1. 创建对象：默认构造函数被用于创建对象，并将其初始化为其数据成员的默认值或零。
2. 初始化成员变量：类的每个成员变量需要被初始化，如果没有提供构造函数，编译器将使用默认构造函数来初始化对象的成员变量。
3. 方便各种形式的实例化：使用默认构造函数的类可以被各种形式的实例化。

下面是一个示例，它展示了如何使用默认构造函数：

```
#include <iostream>

class MyClass {
public:
    int x;

    MyClass() = default;  // 显式指示生成默认构造函数

    MyClass(int val) {
        std::cout << "Parameterized constructor called." << std::endl;
        x = val;
    }
};

int main() {
    MyClass obj1;           // 调用显式生成的默认构造函数
```

```
    MyClass obj2(42);       // 调用带参数的构造函数
    std::cout << obj1.x << std::endl; // 输出 0
    std::cout << obj2.x << std::endl; // 输出 42
    return 0;
}
```

在这个例子中，我们定义了一个MyClass类，它包含一个整数字段x。我们使用default关键字显示指示编译器生成默认构造函数，并用另一个带参数的构造函数重载了它。我们创建了obj1和obj2对象，其中obj1使用了默认构造函数，而obj2使用了带参数的构造函数。我们在main函数中输出这两个对象的值，以便验证它们的确被正确地初始化。

注意，这里我们重载了类的构造函数，然后使用default关键字显示要求编译器生成默认构造函数。稍有不同的场景是，如果我们没有重载类的构造函数，那么编译器将自动生成默认构造函数（无需使用default关键字）。

总之，使用default关键字可以轻松地让编译器生成默认构造函数，降低编写代码的难度，并保证默认构造函数的正确性和兼容性。

2.3.6.为什么会有类中嵌套类

第一部分：本节工程代码中实现了一个类，并且该类里又嵌套了另外一个类，涉及代码如下：

```
class InitManager
{
public:
    //知识点：默认构造写法
    InitManager()= default;
    ~InitManager() = default;

     //知识点：类中嵌套模板类，注册工厂
    template<typename Inherit>
    class Register_t{
        public:
            Register_t(const std::string& key){

                InitManager::getInstance_1()->map_.emplace(key,new Inherit());
            }
    };
};
```

第二部分：为什么会有类中嵌套类的实现呢？把类里的类拿出来不好吗，下面是具体讲法：

在C++程序设计中，类中嵌套类是一种定义在一个类的定义之内的类，被称为内部类。它们可以在外部类的作用域中被直接使用，但是不能在外部类之外的作用域中直接使用。

内部类是一种典型的抽象数据类型，它可以更好地封装数据和实现细节，进而提高代码的复用性、可读性和安全性。以下是内部类具体应用的一些场景：

1. 实现迭代器：当我们需要在迭代器中访问某些私有成员时，可以使用内部类来实现迭代器。

2. 实现模板元编程：在C++中，内部类可以声明成模板类，允许我们实现一些元编程技术，例如嵌套的SFINAE：嵌套类型可以用来计算一些表达式中的类型。
3. 访问外部类的私有成员：内部类可以访问包含它的外部类的私有成员。这允许我们在外部类的实现细节上进行抽象，而不会暴露其具体实现。
4. 与继承和多态结合使用：在继承和多态的情况下，内部类可用于实现辅助类的功能，例如实现缓存或工厂函数模式。
5. 实现状态机：内部类可以用于表示状态，从而可以实现状态机。

总之，内部类可以提高数据封装和抽象级别，提高代码的可读性、安全性和复用性。在许多情况下，使用内部类比使用单独的类更加合适，可以让我们更好地组织代码，并实现更高级别的抽象和封装。

下面是一个C++代码示例，演示了如何在内部类中访问外部类的私有成员：

```
#include <iostream>
using namespace std;

class OuterClass {
private:
    int privateVar;

public:
    OuterClass(int initialValue) {
        privateVar = initialValue;
    }

    class InnerClass {
    public:
        void printPrivateVar(OuterClass& outerObj) {
            cout << "The value of privateVar is: " << outerObj.privateVar << endl;
        }
    };
};

int main() {
    OuterClass outerObj(42);
    OuterClass::InnerClass innerObj;

    innerObj.printPrivateVar(outerObj);

    return 0;
}
```

在这个示例中，`OuterClass`是外部类，它有一个私有成员`privateVar`。在这个类内部，我们定义了一个公共内部类`InnerClass`，并在其中定义了一个函数`printPrivateVar()`。该函数接受一个外部类对象的引用，并打印出其私有成员变量`privateVar`的值。

在主函数中，我们首先实例化一个`OuterClass`对象`outerObj`，并使用`42`初始化了它的私有成员`privateVar`。然后，我们实例化了`OuterClass`的内部类`InnerClass`对象`innerObj`。最后，我们通过调用`innerObj.printPrivateVar()`从内部类中打印出外部类的私有成员变量。

当你编译并运行这段代码时，你将看到下面的输出：

```
The value of privateVar is: 42
```

这表明内部类成功地访问了外部类的私有成员。

2.3.7.智能指针std::unique_ptr

第一部分：本工程代码用智能指针构造了单例对象，如下：

```
    std::unique_ptr<BaseMode> mode_name =
     ConfigManager::getInstance_3()->GetUniqueDevice("A_Mode");
```

第二部分：智能指针std::unique_ptr在工程代码里应用的比较多，它实现的细节和原理如下：

`std::unique_ptr`是C++11中引入的智能指针，它提供了一个对动态分配的对象进行管理的方式，该方式自动管理所有权和内存释放。它是一个具有独占权的智能指针，即同一时间只有一个`std::unique_ptr`对象可以拥有指向某个动态分配对象的指针，保证了内存的正确释放。

`std::unique_ptr`的特点：

- 只能单个`std::unique_ptr`对象拥有所指向的对象
- 所有权不能以复制方式传递，只能以移动方式传递
- 在`std::unique_ptr`被销毁时，它所指向的动态对象也会被销毁
- 支持自定义删除器

下面是一个简单的例子：

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<int> ptr(new int(42)); // 用new分配内存
    std::cout << *ptr << std::endl; // 访问内存

    return 0;
}
```

在这个例子中，我们创建了一个指向`int`类型的`std::unique_ptr`对象，`new int(42)`动态分配了一个整数并返回指向该内存的指针。然后，我们使用`*`操作符访问`std::unique_ptr`所拥有的指针指向的值。

`std::unique_ptr`可以帮助我们避免内存泄漏或重复释放内存等内存管理问题，以保证程序的正确性和可靠性。

使用`std::unique_ptr`构造类对象与构造基本类型对象时类似。唯一的不同是，在将`std::unique_ptr`与类对象结合使用时，需要使用自定义删除器或删除函数，以便在析构时正确释放对象。

下面是一个创建带有自定义删除器的`std::unique_ptr`示例：

```
class MyClass {
public:
    void printMsg() const {
        std::cout << "Hello from MyClass!" << std::endl;
    }

    ~MyClass() {
        std::cout << "MyClass destroyed." << std::endl;
    }
};

void deleteMyClass(MyClass* ptr) {
    std::cout << "Custom deleter called." << std::endl;
    delete ptr;
}

int main() {
    std::unique_ptr<MyClass, void(*)(MyClass*)> ptr(new MyClass(), deleteMyClass);

    ptr->printMsg();

    return 0;
}
```

在这个例子中，我们定义了一个名为`MyClass`的类，并在其中定义了一个打印消息的成员函数`printMsg()`。然后，我们定义了一个名为`deleteMyClass`的自定义删除器，它在析构对象时打印一条消息并使用`delete`运算符释放内存。最后，我们实例化了一个指向`MyClass`类型对象的`std::unique_ptr`对象，使用`new`分配内存，并传递自定义删除器。

我们通过在主函数中调用`ptr->printMsg()`来访问该类的成员函数。

在删除器函数中，我们使用`delete`运算符释放了内存，并在析构时打印了一条消息。当程序退出时，会自动调用`std::unique_ptr`的析构函数来销毁对象，并且在该过程中自定义的删除器也会被调用，从而成功释放内存。

总之，使用`std::unique_ptr`可以确保对象在正确的时候被释放，避免了动态内存分配和释放时常见的问题。

2.3.8.宏定义中的##符使用

第一部分：宏定义中有时候会用到##或者#符号，它们在本节工程代码中涉及如下：

```
    //知识点：宏定义的 ##使用
    #define MODE_FACTORY(T) v2xos_reg_func_str_##T##_
        /**
        * @brief      用于注册指定设备
        * @param[in]  T 具体设备子类型
        * @param[in]  key 注册的设备名字
        */
    #define MODE_FACTORY_FACTORY(T, key) static InitManager::Register_t<T> MODE_FACTORY(T)
(key);
```

第二部分：为什么会有宏定义的#号出现呢，它在代码中发挥一个什么样的角色，下面是具体的细节说明：

宏定义是C++中的预处理指令，提供了一种让程序员自定义并使用自己的代码块的方法。在宏定义中，#符号和##符号都是特殊符号，用于构建更加灵活和可配置的宏定义。

#符号被称为字符串化操作符，它可以使宏的参数字符串化并带上双引号拼接成一个新字符串。例如：

```
#define PRINT(x) cout << #x << ": " << x << endl;
int main() {
    int a = 100;
    PRINT(a); // 将输出"a: 100"
    return 0;
}
```

在这个示例中，我们定义了一个名为PRINT的宏，它对传递的参数x执行打印操作。在宏定义中，我们使用#号将传入的参数“x”变为一个字符串“a”，并将它添加到输出语句中。

##符号被称为连接操作符或令牌粘贴操作符，它可以将两个或多个元素连接成一个新元素。例如：

```
#define CREATE_VARIABLE(name, num) int name##num = 100;
int main() {
    CREATE_VARIABLE(a, 1); // 将创建一个名为a1的int变量，初始值为100
    return 0;
}
```

在这个示例中，我们定义了一个名为CREATE_VARIABLE的宏，它接受两个参数：name和num。在宏定义中，我们使用##符号将它们连接起来，并创建一个新的int变量，名字为“a1”（因为我们调用宏时传递的参数分别是"a"和"1"）并将值设置为100。

这些特殊的符号使宏定义更加灵活和可配置，可以帮助程序员更快地编写自己的代码块和工具，以满足项目的需求。

2.4.本节大作业

本节所讲的工厂模式和单例模式结合使用的模板务必牢牢掌握，它在工作中可应用的地方可能率达到90%，任何功能框架都可以封装成工厂模式，这样方便扩展，用单例模式进行管理节省了资源，并且本节给我们展现了宏定义实现的单例，可谓优雅高级，可直接copy复用到任何单例生成的地方。

很多场景都可以使用以上框架进行编写，比如在工作中你遇到这样一个需求：项目经历让你写一个相机驱动程序，相机种类分很多中，公司当前采购了XX和YY品牌的相机，后续还可能采购ZZ品牌相机，请用本节所学知识完成这个需求，满足可扩展的要求，并写一个测试用例进行测试。

3.原子操作实现对象池模板

在日常学习中，我们可能见到最多的是线程池，并且线程池也是应用最多的，或者我们也听过内存池，队列池，这些也有其价值和应用，今天我们讲一个比较模式的对象池，看看对象池在工作中是怎么发挥它的作用的。

3.1.工程作用

一提到池子，必然会和线程操作联系在一起，因为池子必然有入池和出池，这就涉及了两者操作的同步，本节实现的对象池模板是用原子操作进行同步的，当然也有其他的方式进行同步，比如加锁，和信号量，这里不是重点，重点来看对象池发挥作用的场合有哪些：

> 1，对象是对类的一个封装，每次构建对象的时候也仅仅只能构建一个，开辟一块空间，进行资源的回收；
>
> 2，如果我们需要多次创建对象，并且对象应用要求是不同的，这个时候每次都去创建一个新的对象，感觉是优点浪费空间，有没有一种可能，想队列一样，一次性创造多个对象，需要的时候去池里取，方便应用。
>
> 3，用这个思路，我们看看对象池是如何实现的?

3.2.框架实现

基于上述讨论，我们本节实现一个对象池，这个对象池里放多个对象，根据需要去插入或者取出，满足我们随时构建对象的要求，下面先来实现一个别开生面的for循环，这个for循环的实现是为了下一个框架对象池而生的。

3.2.1.重写编译器for循环模板

我们都知道，代码中用的最多的就是for循环，这个fox循环有没可能对其进行封装一下，用宏定义进行封装接口，任何框架都可以进行引用，并且里面应用到多个C++11的新特性。

```cpp
#include <cstdlib>
#include <new>
#include <type_traits>
#include <vector>
#include <iostream>

/*
1，宏定义
2，类型特征 type trait，用于判断一个类型是否满足某个条件
例如 std::is_pointer判断是否为指针类型

3，constexpr：表达式在编译期间就可以被求值，而不是在运行时才求值
将 value声明为编译器常量，保证在编译期间就可以被求值

4， std::enable_if

*/
//用来定义一个类型特征：判断一个类型是否包含名为 func的成员函数
/*
宏定义一个 name 结构体模板，测试一个类型T 是否满足特定条件的结果
1， Test 函数模板接受一个参数，判断参数是否为特定类的成员函数指针，并返回true或false
2，如果参数为 类 class 的成员函数指针，返回true，否则返回false
3，name结构体模板的静态成员变量 value 会被赋值 为 Test<T>(nullptr)的结果，用于判断 T 是否满足特定条件

*/

//知识点：宏定义的函数传参，前两节以已经讲过
#define DEFINE_TYPE_TRAIT(name, func)                    \
  template <typename T>                                  \
  struct name {                                          \
    template <typename Class>                            \
    static constexpr bool Test(decltype(&Class::func)*) { \
      return true;                                       \
    }                                                    \
    template <typename>                                  \
    static constexpr bool Test(...) {                    \
      return false;                                      \
    }                                                    \
                                                         \
    static constexpr bool value = Test<T>(nullptr);      \
  };                                                     \
                                                         \
  template <typename T>                                  \
  constexpr bool name<T>::value;

DEFINE_TYPE_TRAIT(HasLess,operator<)

/*
enable_if 一个模板元编程工具，在编译期间根据模板参数的条件选择是否实例化模块函数或类

如果 value和 end 都有 < 操作符的实现（HasLess返回true），则启动此函数，并将其返回类型设为 bool
```

```
*/

    //知识点：已经讲过
template<class Value,class End>
typename std::enable_if<HasLess<Value>::value && HasLess<End>::value, bool>::type
LessThan(const Value& val, const End& end){
    return val < end;
}

/*
如果 value和 end 任意一个没有 < 操作符的实现（HasLess返回false），则启动此函数，并将其返回类型设为 bool
*/
template <class Value, class End>
typename std::enable_if<!HasLess<Value>::value || !HasLess<End>::value, bool>::type
LessThan(const Value& val, const End& end){
    return val != end;
}

//真正调用的地方
#define FOR_EACH(i,begin,end) \
    for(auto i = (true? (begin) : (end)); \
        LessThan(i, (end)); ++i)


class Test{
  public:
    //测试
    void test(){
       std::vector<int> vec;
      FOR_EACH(i, 0, 100){
          vec.push_back(i);
      }

      int index = 0;
      FOR_EACH(it, vec.begin(),vec.end()){
          std::cout<<*it<<std::endl;
      }
    }

};


int main()
{
    Test test_;
    test_.test();
}
```

3.2.2.用原子操作实现的对象池

本小节是重点，利用上述实现的for循环模板，我们进行原子操作的对象池模板构建。

```
#include <cstdlib>
#include <new>
#include <type_traits>
#include <vector>
#include <iostream>
#include <memory>
#include <functional>
#include <mutex>
#include <atomic>

/*
此处省略了 3.2.1.小节中 FOR_EACH的实现

*/

/*
@ 构造一个对象池，自定义对象池可以存放对象的个数
@ 1，线程池
@ 2，内存池
@ 3，队列缓冲池
@ 4，对象池
读，写，加锁或者信号量，今天讲原子操作的应用

【 Object， Object，Object 】

1，define的字符串用法
2，inline用法， explicit用法
3，std::calloc
4，std::enable_shared_from_this
5，alignas
6，std::atomic的各种成员函数
       store(),std::memory_order_relaxed,
       load()
       compare_exchange_weak()
7，std::forward
8，多参数形参  template <typename... Args>
9，std::once_flag

10，std::function<void()>;

公众号： 码出名企路
工作中是怎么写C++系列?

**/

//知识点：宏定义的括号
#define cyber_unlikely(x) (x)
```

```cpp
//知识点: inline函数
inline void* CheckedCalloc(size_t num, size_t size) {
  //知识点: std::calloc/ std::bad_calloc
  void* ptr = std::calloc(num, size);
  if (!ptr) {
    throw std::bad_alloc();
  }
  return ptr;
}

/*
@ 构造 objectPool
主要实现construct,get和release object对象
*/

//知识点: std::;enbale_shared_from_this
template<typename T>
class CCObjectPool:public std::enable_shared_from_this<CCObjectPool<T>>
{
    public:
        explicit CCObjectPool(uint32_t size);
        virtual ~CCObjectPool();

        //知识点: 对参数传参
        template <typename... Args>
        void ConstructAll(Args &&... args);

        template <typename... Args>
        std::shared_ptr<T> ConstructObject(Args &&... args);

        std::shared_ptr<T> GetObject();
        void ReleaseObject(T *);

    private:
        struct Node{
            T object;
            Node *next;
        };
        //知识点: alignas内存对奇
        struct alignas(2 * sizeof(Node*)) Head{
            uintptr_t count;
            Node *node;
        };

    private:
        CCObjectPool(CCObjectPool &) = delete;
        CCObjectPool &operator=(CCObjectPool &) = delete;
        bool FindFreeHead(Head *head);

        //知识点: 原子操作
        std::atomic<Head> free_head_;
        Node *node_arena_ = nullptr;
        uint32_t capacity_ = 0;
```

```
};

template <typename T>
CCObjectPool<T>::CCObjectPool(uint32_t size):capacity_(size)
{
    node_arena_ = static_cast<Node*>(CheckedCalloc(capacity_,sizeof(Node)));
    FOR_EACH(i,0,capacity_){
        node_arena_[i].next = node_arena_ + 1 + i;
    }

    node_arena_[capacity_ - 1].next = nullptr;
    //知识点：原子操作 store
    free_head_.store({0,node_arena_}, std::memory_order_relaxed);
}

template <typename T>
template <typename... Args>
void CCObjectPool<T>::ConstructAll(Args &&... args)
{
    FOR_EACH(i,0,capacity_){
        //知识点：完美转发 std::forward
        new (node_arena_ + i) T(std::forward<Args>(args)...);
    }
}

template <typename T>
CCObjectPool<T>::~CCObjectPool()
{
    std::free(node_arena_);
}

template <typename T>
bool CCObjectPool<T>::FindFreeHead(Head *head)
{
    Head new_head;
    //知识点：原子操作 load
    Head old_head = free_head_.load(std::memory_order_acquire);

    do{
        if(cyber_unlikely(old_head.node == nullptr)){
            return false;
        }

        new_head.node = old_head.node->next;
        new_head.count = old_head.count + 1;
        //知识点：原子操作compare_exchange_weak
    }while(!free_head_.compare_exchange_weak(old_head, new_head,
                                             std::memory_order_acq_rel,
                                             std::memory_order_acquire));

    *head = old_head;
    return true;
}
```

```
template <typename T>
std::shared_ptr<T> CCObjectPool<T>::GetObject()
{
    Head free_head;
    if (cyber_unlikely(!FindFreeHead(&free_head))){
        return nullptr;
    }
    //知识点：shared_from_this和this区别
    auto self = this->shared_from_this();
    //知识点：自定义的std::shared_ptr析构
    return std::shared_ptr<T>(reinterpret_cast<T*>(free_head.node),
            [self](T *object){
                self->ReleaseObject(object);
            });
}

template <typename T>
template <typename... Args>
std::shared_ptr<T> CCObjectPool<T>::ConstructObject(Args &&... args)
{
    Head free_head;
    if (cyber_unlikely(!FindFreeHead(&free_head))){
        return nullptr;
    }

    auto self = this->shared_from_this();
    T *ptr = new (free_head.node) T(std::forward<Args>(args)...);

    return std::shared_ptr<T>(ptr, [self](T *object){
        object->~T();
        self->ReleaseObject(object);
    });
}


template <typename T>
void CCObjectPool<T>::ReleaseObject(T *object)
{
    Head new_head;
    Node *node = reinterpret_cast<Node*>(object);

    Head old_head = free_head_.load(std::memory_order_acquire);
    do{
        node->next = old_head.node;
        new_head.node = node;
        new_head.count = old_head.count + 1;
    }while(!free_head_.compare_exchange_weak(old_head, new_head,
                                             std::memory_order_acq_rel,
                                             std::memory_order_acquire));
}
```

```
//进行测试
constexpr size_t STACK_SIZE = 2 * 1024 * 1024;
struct RoutineContext {
  char stack[STACK_SIZE];
  char* sp = nullptr;
};


//test
using RoutineFunc = std::function<void()>;
class Test{
    public:
      //知识点: explicit
        explicit Test(const RoutineFunc &func):func_(func){
            std::call_once(pool_init_flag, [&]() {
                /*
                  自定义单例对象构造
                */
                 context_pool.reset(new CCObjectPool<RoutineContext>(10));
            });

            context_ = context_pool->GetObject();
            if (context_ == nullptr) {
                std::cout << "Maximum routine context number exceeded! Please check "
                        "[routine_num] in config file.";
                context_.reset(new RoutineContext());
            }

            func_();
        }
        virtual ~Test(){context_ = nullptr;}


    private:

        std::shared_ptr<RoutineContext> context_;
        RoutineFunc func_;
        std::once_flag pool_init_flag;
        std::shared_ptr<CCObjectPool<RoutineContext>> context_pool = nullptr;

};

void function()
{
    printf("i am lyy");
}

int main()
{
    std::shared_ptr<Test> cr = std::make_shared<Test>(function);
}
```

3.3.知识点拆分

在本节原子操作实现对象池模板中，我们首先学习了一种for循环新的写法和实现，这种宏定义的封装给for循环带来恰当的好处，可以满足大量工程的调用，其中之一的用处我们看到它被用在了对象池模板里，这两大模块的实现引用了C++11的许多新特性，主要涉及的知识点如下：

1，宏定义加括号的用法

2，inline和explicit关键字

3，std::calloc分配内存

4，std::enable_shared_from_this用法

5，内存对齐alignas

6，原子操作std::atomic各种成员函数

7，完美转发std::forward

8，模板多参数传参

9，std::function函数模板

下面我们分小节将以上知识点带入到框架中去，学习其详细的用法。

3.3.1.宏定义为什么要加括号

第一部分：这种用法在框架中的体现如下：

```
#define cyber_unlikely(x) (x)
```

第二部分：如果不加括号，会出现什么问题，我们带着这个思考，看一下这种写法背后的原因。

在宏定义中加括号是一个良好的编程习惯，可以避免一些常见的预处理器错误。以下是需要加括号的几种情况：

1. 宏定义中有表达式

如果宏定义中包含运算符和表达式，则应该将表达式放在括号中。否则，可能会由于优先级不当导致预期外的结果。例如，以下代码可能无法正确编译：

```
#define ADD(a, b) a + b

int sum = ADD(1, 2) * 3;
```

因为预处理器将其展开为：

```
int sum = 1 + 2 * 3;
```

正确的做法是将ADD的表达式放在括号中：

```
#define ADD(a, b) ((a) + (b))

int sum = ADD(1, 2) * 3;
```

这样，预处理器将其展开为：

```
int sum = (1 + 2) * 3;
```

2. 宏定义中有参数

如果宏定义中有参数，则应该将参数放在括号中，以避免出现歧义。例如：

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int x = 2, y = 3;
int max = MAX(x++, y++);
```

如果没有加括号，则条件表达式可能不会按预期计算。在这个例子中，由于MAX的展开式中使用了两次参数，每次使用时都会自增，导致最终结果为4，而不是预期的3。因此应该这样定义MAX：

```
#define MAX(a, b) (((a) > (b)) ? (a) : (b))
```

在宏定义中加括号是一个容易遵循的编程习惯，它可以有效避免一些常见的预处理器错误，提高代码的可读性和可维护性。

3.3.2.关键字inline和explicit什么时候用

第一部分：我们看到了这两个关键字在实际框架中的使用如下：

```
inline void* CheckedCalloc(size_t num, size_t size)
```

```
explicit CCObjectPool(uint32_t size);
```

第二部分：inline理解为内敛的意思，什么时候用内敛函数呢?

`inline`是C++中的一个关键字，用于修饰函数或类中的成员函数。它的作用是向编译器建议将函数或成员函数的代码插入到调用处，而不是像普通函数一样在程序中生成一个函数的实现，并通过跳转来调用。这种方式可以显著减少函数调用的开销，提高程序的执行效率。

需要注意的是，在某些情况下，编译器可能不会将`inline`函数或成员函数的代码插入到调用处。例如：

1. 函数体过于复杂或函数调用的频率很低。
2. `inline`函数或成员函数中包含循环或递归。
3. 在不同的编译单元中对同一个`inline`函数或成员函数进行不一致的定义。

在使用`inline`关键字时，需要注意以下几点：

1. `inline`关键字应该放在函数或成员函数的定义处，而不是声明处。因为编译器需要函数或成员函数的定义体才能将其展开。
2. `inline`关键字是一个建议性的修饰词，编译器可以忽略它。因此，在使用`inline`时应该确保其确实能提供性能上的改进，并在必要时使用其他优化技术。
3. 频繁地使用`inline`可能会导致代码数量的增加，从而增加可执行文件的大小。因此，在使用`inline`时需要谨慎考虑。

总的来说，使用`inline`关键字可以提高程序的执行效率，但需要根据实际情况进行权衡和选择。一般来说，适用于内联的函数和成员函数包括较短的、频繁调用的函数或成员函数，避免使用循环或递归等复杂结构的函数。

当函数被定义为inline函数时，编译器将尝试将函数的代码直接插入到调用函数的位置，以避免函数调用的开销。这种方法在函数代码较短且频繁调用时可以提高程序的性能。以下是几个使用C++的inline函数的示例：

1. 数值平方的计算

```
inline int square(int num)
{
   return num * num;
}

int main()
{
   int num = 5;
   int result = square(num);
   return 0;
}
```

在这个例子中，`square`函数计算传递的数值的平方，使用`inline`关键字强制将函数内联。由于函数的代码非常简单，编译器很可能将代码插入到调用`square`的位置，从而避免了函数调用的开销。

2. 字符串查找

```
inline int find(char* str, char c)
{
   int i = 0;
   while (str[i] != '\0') {
      if (str[i] == c) return i;
      i++;
   }
   return -1; // not found
}

int main()
{
   char* str = "hello world";
```

```
    char c = 'w';
    int result = find(str, c);
    return 0;
}
```

这个例子中，`find`函数使用循环在传递的字符串`str`中查找指定的字符`c`。由于函数的代码较短，我们可以使用`inline`关键字来将其强制内联，以提升性能。

3. 时间戳计算

```
inline long timestamp()
{
    std::chrono::milliseconds ms = std::chrono::duration_cast<std::chrono::milliseconds>
(std::chrono::system_clock::now().time_since_epoch());
    return ms.count();
}

int main()
{
    std::cout << "Current timestamp: " << timestamp() << std::endl;
    return 0;
}
```

这个例子中，`timestamp`函数返回当前时间戳的毫秒级数值。由于函数内部使用了C++11的时间库，代码较长。但由于该函数的代码不复杂，我们可以使用`inline`关键字将其强制内联，以减少函数调用的开销。

需要注意的是，使用`inline`关键字并不能保证函数一定会被内联，这取决于编译器的实现。在大多数情况下，编译器会根据代码复杂性和程序性能等考虑因素，自动选择是否将函数内联。因此，只有在函数的代码非常简单和需要频繁调用时，才有必要使用`inline`关键字。

explicit又是什么含义得呢，它具体体现在什么地方，下面我们具体来看一下：

在C++中，`explicit`关键字用于避免隐式类型转换，例如构造函数中的参数类型转换。以下是几个使用`explicit`关键字的示例：

1. 构造函数的显式声明

```
class MyInt
{
public:
    explicit MyInt(int num) : m_num(num) {}
    int getNum() const { return m_num; }
private:
    int m_num;
};

void displayNum(MyInt num)
{
    std::cout << "Number: " << num.getNum() << std::endl;
}
```

```
int main()
{
    // MyInt num = 42; // Error: implicit conversion from int to MyInt
    MyInt num(42); // OK: explicit conversion from int to MyInt
    displayNum(num);
    return 0;
}
```

在这个例子中，`MyInt`类的构造函数使用`explicit`关键字显式声明，表示不允许参数的隐式类型转换。因此，如果我们试图通过将一个整数隐式转换为`MyInt`对象来创建一个新的`MyInt`对象，则会得到一个编译时错误。相反，我们必须显式调用构造函数来创建一个新的对象。

2. 类型转换函数的显式声明

```
class MyFloat
{
public:
    explicit MyFloat(float num) : m_num(num) {}
    float getNum() const { return m_num; }
private:
    float m_num;
};

class MyDouble
{
public:
    explicit MyDouble(double num) : m_num(num) {}
    double getNum() const { return m_num; }
    operator MyFloat() const { return static_cast<MyFloat>(m_num); }
private:
    double m_num;
};

void displayNum(MyFloat num)
{
    std::cout << "Number: " << num.getNum() << std::endl;
}

int main()
{
    MyDouble num(3.1415);
    // displayNum(num); // Error: no matching function for call to 'displayNum'
    displayNum(static_cast<MyFloat>(num)); // OK: explicit conversion from MyDouble to
MyFloat
    return 0;
}
```

在这个例子中，`MyDouble`类有一个类型转换函数，它将`MyDouble`对象转换为`MyFloat`对象。但由于该函数的返回类型不是`MyFloat`，而是`MyDouble`本身，所以在调用`displayNum`函数时会出现编译时错误。为了解决这个问题，我们需要将`MyDouble`对象转换为`MyFloat`对象，然后将其传递给`displayNum`函数。由于`MyFloat`的构造函数被声明为`explicit`，我们无法将`MyDouble`对象直接传递给`displayNum`函数，而必须显式地使用`static_cast`来进行类型转换。

需要注意的是，在构造函数或类型转换函数中使用`explicit`关键字可能会导致程序的可读性和灵活性受到影响。因此，应该仔细考虑是否需要使用该关键字。通常情况下，只有在明确需要避免隐式类型转换时才使用`explicit`关键字。

3.3.3.内存分配std::calloc用法

第一部分：它是用来进行内存分配的，主要用法体现在：

```
  void* ptr = std::calloc(num, size);
  if (!ptr) {
    throw std::bad_alloc();
  }
```

第二部分：new和malloc还不够用吗，为什么有出现这个？具体用法如下：

`std::calloc`是一个C++标准库函数，用于在内存中分配一块指定数量的连续空间，并初始化为0。其用法与C标准库函数`calloc`相似。以下是几个使用`std::calloc`函数的示例：

1. 分配整型数组内存空间

```
#include <iostream>
#include <cstdlib>

int main()
{
    int size = 10;
    int* arr = static_cast<int*>(std::calloc(size, sizeof(int)));

    if (arr != nullptr)
    {
        for (int i = 0; i < size; ++i)
        {
            std::cout << arr[i] << " ";
        }
        std::cout << std::endl;

        std::free(arr);
    }

    return 0;
}
```

在这个例子中，我们使用`std::calloc`函数分配了一个包含`10`个整数的数组，并用`0`进行初始化。如果内存分配成功，我们遍历数组并输出其初始化后的值，最后使用`std::free`函数释放内存空间。

2. 分配字符数组内存空间

```
#include <iostream>
#include <cstdlib>
#include <cstring>

int main()
{
    const char* str = "Hello, world!";
    int len = std::strlen(str);
    char* buffer = static_cast<char*>(std::calloc(len + 1, sizeof(char)));

    if (buffer != nullptr)
    {
        std::strcpy(buffer, str);
        std::cout << buffer << std::endl;

        std::free(buffer);
    }

    return 0;
}
```

在这个例子中，我们使用`std::calloc`函数分配了一个大小为`len+1`的字符数组，并将其初始化为`0`。我们将字符指针变量`str`指向的字符串复制到分配的缓冲区中，并输出该字符串。最后，我们使用`std::free`函数释放内存空间。

3. 分配二维数组内存空间

```
#include <iostream>
#include <cstdlib>

int main()
{
    int rows = 3, cols = 4;
    int** matrix = static_cast<int**>(std::calloc(rows, sizeof(int*)));

    for (int i = 0; i < rows; ++i)
    {
        matrix[i] = static_cast<int*>(std::calloc(cols, sizeof(int)));
    }

    if (matrix != nullptr)
    {
        for (int i = 0; i < rows; ++i)
        {
            for (int j = 0; j < cols; ++j)
            {
                std::cout << matrix[i][j] << " ";
            }
            std::cout << std::endl;
            std::free(matrix[i]);
```

```
        }

        std::free(matrix);
    }

    return 0;
}
```

在这个例子中，我们使用`std::calloc`函数分配一个大小为`rows x cols`的二维整型数组，并将其初始化为`0`。我们遍历该数组并输出其初始化后的值。最后，我们释放每行的内存空间，并使用`std::free`函数释放二维数组的内存空间。

需要注意的是，使用`std::calloc`函数分配的内存空间需要使用`std::free`函数来释放，否则会出现内存泄漏。此外，在使用前必须检查分配是否成功，否则会出现程序崩溃等问题。

3.3.4.std::enable_shared_from_this真正作用

第一部分：它用在类得声明处，告诉这个类可以返回自己得this指针，在本框架中用法体现在下面：

```
class CCObjectPool:public std::enable_shared_from_this<CCObjectPool<T>>
{}
```

第二部分：什么情况下需要使用std::enable_shared_from_this呢，它和this指针有什么区别，返回对象得时候直接返回this指针不就好了，为什么还需要这个？带着这些疑问，我们看下std::enable_shared_from_this的具体用法。

`std::enable_shared_from_this` 是一个 C++11 的模板类，主要用于解决在对象的共享指针中，需要获得该对象的另一个共享指针的问题。举个例子，如果有一个对象 `obj`，并且有多个共享指针 `ptr1`、`ptr2` 等等指向这个对象，如果我们需要在 `obj` 的成员函数中获取一个指向 `obj` 的共享指针，那么我们可以使用 `std::enable_shared_from_this`。

使用方法很简单：需要在你的类中继承 `std::enable_shared_from_this<ClassName>`，然后在成员函数中使用 `shared_from_this()` 获取指向该对象的共享指针。

需要注意的是，只有使用共享指针（`std::shared_ptr<T>`）才可以使用 `std::enable_shared_from_this`，使用普通指针或者其他智能指针是无法使用的。同时需要保证在调用 `shared_from_this()` 时，该对象已经被至少一个共享指针管理，否则会抛出 `std::bad_weak_ptr` 异常。

例子如下：

```
#include <memory>
#include <iostream>

class MyClass : public std::enable_shared_from_this<MyClass> {
public:
    void execute() {
```

```
        std::shared_ptr<MyClass> ptr = shared_from_this();
        std::cout << "shared_ptr count: " << ptr.use_count() << std::endl;
    }
};

int main() {
    std::shared_ptr<MyClass> ptr(new MyClass());
    ptr->execute();
    std::cout << "shared_ptr count: " << ptr.use_count() << std::endl;
    return 0;
}
```

输出结果为：

```
shared_ptr count: 2
shared_ptr count: 1
```

可以看到，通过 `shared_from_this()` 获取到了另一个指向 `MyClass` 的共享指针，并且共享指针的计数器为 2。

3.3.5.内存对齐的alignas

第一部分：内存对齐在定义结构体中是需要经常考虑的问题，其中alignas的用法在工程中体现如下：

```
struct alignas(2 * sizeof(Node*)) Head{
uintptr_t count;
Node *node;
};
```

第二部分：alignas可以自己进行内存字节数对齐吗？如果可以，我们是不是就不用考虑结构体定义变量顺序的问题了，参考如下具体解释：

`alignas` 是 C++11 中引入的一个关键字，用于指定类型或变量的对齐方式。在 C++中，对齐是为了最大程度地利用计算机硬件的特性，因为如果一个数据类型没有按照特定的字节边界对齐，处理器就需要进行额外的处理才能将数据读出，这就会降低程序执行的效率，特别是在处理大量数据时。因此，可以使用`alignas`关键字在编译时控制数据的对齐方式，从而提高程序的性能。

`alignas` 关键字的语法如下：

```
alignas(n) variable;
alignas(n) type;
```

其中，`n` 指定对齐方式，可以是 2、4、8、16 或其他 2 的幂。`variable` 是变量名，`type` 是类型名。

以下是一个示例：

```
#include <iostream>

int main()
{
    alignas(32) unsigned char buffer[128];
    std::cout <<"buffer alignment: "<<std::alignment_of<decltype(buffer)>::value<<
std::endl;
    return 0;
}
```

在这个示例中，我们创建了一个大小为 128 字节的无符号字符数组，并使用 `alignas(32)` 声明其对齐方式为 32 字节。然后使用 `std::alignment_of` 获取 `buffer` 的对齐方式，并输出结果，输出结果应该是：

```
buffer alignment: 32
```

由此可以看出，`buffer` 确实按照 32 字节对齐，并且可以使用相应的指令集读写该数据缓存，从而提高程序执行效率。

需要注意的是，使用 `alignas` 指定了数据的对齐方式后，就需要自行确保该数据的对齐方式，否则可能会导致程序崩溃或产生未定义的行为。

3.3.6.原子操作atomic的几大成员函数

第一部分：原子操作中有load,store等，还要一些我们不常见的交换函数，具体涉及代码如下：

```
free_head_.compare_exchange_weak(old_head, new_head,
                                                 std::memory_order_acq_rel,
                                                 std::memory_order_acquire));
```

第二部分：我们具体看看atomic的几个函数的应用。

std::atomic 是 C++11 引入的一个模板类，用于实现原子类型，支撑多线程环境下的原子操作。 操作以原子方式执行，不会被中断，并且在所有线程上看起来是单个、无中间状态的操作。

std::atomic 包含了多个成员函数，下面是其中几个常用的：

1. load()：以原子方式读取 atomic 变量的值。

```
std::atomic<int> x {42};
int a = x.load();
```

2. store()：以原子方式存储 val 到 atomic 变量中。

```
std::atomic<int> x {42};
x.store(43);
```

3. exchange()：以原子方式将 atomic 变量中的值替换为 val，并返回替换前 atomic 变量的值。

```
std::atomic<int> x {42};
int a = x.exchange(43);
```

4. compare_exchange_weak()：比较 atomic 变量的值和期望值，如果相等，以原子方式存储新值，否则存储原值。该函数是弱 CAS，可能会失败并重试，因此应该在循环中使用。

```
std::atomic<int> x {42};
int expected = 42;
int desired = 43;
bool success = x.compare_exchange_weak(expected, desired);
```

5. compare_exchange_strong()：和 compare_exchange_weak() 函数类似，但是该函数是强 CAS，在失败时不能重试，应该直接返回。

```
std::atomic<int> x {42};
int expected = 42;
int desired = 43;
bool success = false;
while (!success) {
    success = x.compare_exchange_strong(expected, desired);
}
```

6. fetch_add()：以原子方式将 atomic 变量增加 val，并返回增加前 atomic 变量的值。

```
std::atomic<int> x {42};
int a = x.fetch_add(1);
```

7. fetch_sub()：以原子方式将 atomic 变量减少 val，并返回减少前 atomic 变量的值。

```
std::atomic<int> x {42};
int a = x.fetch_sub(1);
```

8. is_lock_free()：返回该 atomic 类型是否为无锁类型。如果为无锁类型，则 Atomic 操作不会调用内核中的锁定机制，通常更快，但是不保证原子操作的有序性。

```
std::atomic<int> x {42};
bool b = x.is_lock_free();
```

除了以上提到的成员函数，std::atomic 还提供了更多的原子操作，这里只是列出了常用的几个。需要注意的是，std::atomic 不是万能的，对于一些复杂的数据结构或业务场景，还需要使用更细粒度的同步原语，例如 std::mutex。

3.3.7.完美转发std::forward什么时候用

第一部分：完美转发和右值引用有什么区别，本文涉及完美转发的代码如下：

```
  auto self = this->shared_from_this();
    T *ptr = new (free_head.node) T(std::forward<Args>(args)...);
```

第二部分：在实际工作中，什么时候考虑用std::forward，是我们应该思考的问题，下面我们看看它具体的用法

`std::forward` 是一个模板函数，通常在实现完美转发时使用。在 C++ 中，一个函数当前接受的参数在传递给另一个函数时，这些参数的类型会发生改变或丢失。为了保持参数的类型完整性，C++11 引入了右值引用，但是右值引用的引入增加了代码的复杂性，实现也比较困难。为了解决这个问题，C++11 还引入了完美转发。

完美转发又分为两种，一种是从函数中传递参数到其他函数，另一种是从函数模板中传递参数到其他函数模板。

前者中，完美转发指的是在接受参数时使用右值引用（`&&`）来保证原始参数的值类别被保留（即左值传左值、右值传右值）。具体而言，我们可以使用如下方式进行：

```
template <typename T>
void f(T&& arg) {
    g(std::forward<T>(arg)); // forward arg's value category
}
```

这里 `std::forward` 就是用来保持参数类型和值类别的。在传递参数的时候，我们通常会将参数转发给其他函数，但是我们不知道该参数的值类别，这时候可以使用 `std::forward` 来保持原参数的值类别，就可以保证转发后的参数类型和值类别都是一致的，从而达到完美转发的效果。

另外，需要注意的是，只有当传入的参数是一个右值时，使用 `std::forward` 才有意义。由于完美转发的本质是保持参数的值类别，因此传入的参数值类别与模板参数不匹配时，此时使用 `std::forward` 不会有任何作用。

总之，如果你需要写有参数转发功能的模板函数，那么就需要考虑使用 `std::forward` 进行完美转发，以保持参数的类型和取值类别在传递到下一个函数或模板的过程中不被丢失。

3.3.8.模板函数多参数传参Args

第一部分：多参数形参在模板编程中应用广泛，一个函数最多传入的参数没有限制，但是传入太多也不好看，本章涉及的Args代码如下：

```
template <typename T>
template <typename... Args>
void CCObjectPool<T>::ConstructAll(Args &&... args)
{
    FOR_EACH(i,0,capacity_){
        //知识点：完美转发 std::forward
        new (node_arena_ + i) T(std::forward<Args>(args)...);
    }
}
```

第二部分：什么时刻考虑Args的用法，具体细节如下：

在 C++ 的模板编程中，`Args` 通常被用作可变参数模板中的参数列表类型。下面举几个例子说明 `Args` 在模板编程中的应用：

1. 实现可变参数函数模板

可变参数函数模板是指接受可变数量的参数并提供通用、灵活的实现方式的函数模板。它可以使用 `Args...` 语法来接受任意数量的参数，例如：

```
template <typename... Args>
void foo(Args... args) {
    // ...
}
```

这里的 `Args` 表示参数的类型列表，`...` 表示参数包扩展语法，用于展开参数包中的参数。在函数体中，可以使用 `args...` 语法来展开参数列表，例如：

```
template <typename... Args>
void foo(Args... args) {
    ((std::cout << args << " "), ...);
}
```

这样就可以将参数列表展开为表达式 `(cout << args) , ...)`，其中 `,` 表示将多个表达式组合在一起，从而实现参数列表的输出。

2. 实现可变参数模板类

可变参数模板类是指接受可变数量的模板参数并提供通用、灵活的实现方式的模板类。它可以使用 `Args...` 语法来接受任意数量的模板参数，例如：

```
template <typename... Args>
class Foo {
    // ...
};
```

这里的 `Args` 表示模板参数的类型列表，`...` 表示参数包扩展语法，用于展开参数包中的参数。在类中可以使用 `Args...` 语法来展开参数列表，例如：

```
template <typename... Args>
class Foo {
public:
    using Tuple = std::tuple<Args...>;
};
```

这里利用 `std::tuple` 来创建了一个模板类型别名 `Tuple`，其中的 `Args...` 表示所有传递进来的类型都会被放入一个 `std::tuple` 中。

3. 实现可变参数模板函数的转发

有时候，在实现可变参数模板函数时，需要将参数转发给其他函数进行处理。这时候可以使用 `std::forward` 和 `Args...` 语法来实现，例如：

```
template <typename... Args>
void foo(Args&&... args) {
    bar(std::forward<Args>(args)...);
}
```

这里的 `Args` 表示参数的类型列表，`...` 表示参数包扩展语法，用于展开参数包中的参数。在调用函数 `bar` 时，使用 `std::forward` 函数对参数 `args` 进行了完美转发，以保留其原始值类别。

总之，`Args` 是一个非常有用的概念，可以在模板编程中实现许多灵活的功能，例如实现可变参数函数模板、可变参数模板类以及可变参数模板函数的转发等。它能够在模板的实现中提高代码的通用性和灵活性，避免重复代码的出现，使得代码更加简洁、易于维护。

3.3.9.std::function函数模板

第一部分：有时候做函数绑定的时候，都会用到函数模板std::function，本工程框架中用到如下：

```
using RoutineFunc = std::function<void()>;
```

第二部分：函数模板std::function一般与std:::bind结合使用，目的是注册函数，细节如下：

`std::bind` 是 C++11 中的一个标准库函数，用于将可调用对象和一部分参数绑定起来，生成一个新的可调用对象，其中绑定的参数成为该可调用对象的一部分参数。`std::function` 则可以用于存储任何可调用对象，包括通过 `std::bind` 绑定产生的对象。

下面是一些示例，演示了如何使用 `std::bind` 和 `std::function` 结合，实现更加灵活和可读性强的代码：

1. 使用 `std::bind` 将参数绑定到一个函数上，创建一个新的函数对象：

```
#include <iostream>
#include <functional>

void func(int x, int y, int z) {
    std::cout << x << " + " << y << " + " << z << " = " << (x + y + z) << std::endl;
}

int main() {
    auto f = std::bind(func, 1, 2, std::placeholders::_1);
    f(3);
    return 0;
}
```

这个示例使用了 `std::bind` 将参数 `1` 和 `2` 绑定到函数 `func` 上，将第三个参数作为占位符，然后创建了一个新的函数对象 `f`。调用 `f(3)` 就相当于调用 `func(1, 2, 3)`，输出为 `1 + 2 + 3 = 6`。

2. 使用 `std::function` 存储一个 `std::bind` 绑定产生的对象：

```
#include <iostream>
#include <functional>

void func(int x, int y, int z) {
    std::cout << x << " * " << y << " * " << z << " = " << (x * y * z) << std::endl;
}

int main() {
    auto f = std::function<void(int)>(std::bind(func, 2, std::placeholders::_1, 3));
    f(4);
    return 0;
}
```

这个示例使用了 `std::bind` 将参数 `2` 和 `3` 绑定到函数 `func` 上，将第二个参数作为占位符，然后使用 `std::function` 来存储 `std::bind` 绑定产生的对象，并创建了一个新的函数对象 `f`。调用 `f(4)` 就相当于调用 `func(2, 4, 3)`，输出为 `2 * 4 * 3 = 24`。

3. 使用 `std::bind` 将成员函数绑定到对象上：

```
#include <iostream>
#include <functional>

class Foo {
public:
    void func(int x, int y, int z) {
        std::cout << x << " - " << y << " - " << z << " = " << (x - y - z) << std::endl;
    }
};

int main() {
    Foo foo;
    auto f = std::bind(&Foo::func, &foo, 5, std::placeholders::_1, 6);
    f(4);
    return 0;
```

```
}
```

这个示例使用了 `std::bind` 将成员函数 `Foo::func` 绑定到一个对象 `foo` 上，并将参数 `5` 和 `6` 绑定到函数中，将第二个参数作为占位符。然后创建了一个新的函数对象 `f`。调用 `f(4)` 就相当于调用 `foo.func(5, 4, 6)`，输出为 `5 - 4 - 6 = -5`。

总之，`std::bind` 和 `std::function` 可以结合使用，实现更加灵活和可读性强的代码。例如可以使用 `std::bind` 将参数绑定到一个函数上创建一个新的函数对象，也可以使用 `std::function` 存储 `std::bind` 绑定产生的对象，或者使用 `std::bind` 将成员函数绑定到对象上实现更加灵活的逻辑。

3.4.本节大作业

通过本节的学习，我们知道了对象池是一种提高程序性能和减少内存分配与释放的技术，其原理是在程序启动的时候预先分配一些对象，然后将它们存储在一个池子中，当需要使用某个对象的时候，从池子里取出一个可用的对象，并在使用完成后对象归还到池子中，避免反复内存和释放导致的开销。主要可以用在以下几个地方：

1. 游戏和图形处理

在游戏和图形处理中，通常需要创建大量的对象，如粒子、特效、音效、模型等等。如果每次需要某个物体时都要重新创建一次，而且这些物体又有比较长的生命周期，那么每次创建和销毁都会花费较多时间和性能。因此采用对象池技术可以有效地优化这个问题。

1. 网络编程

在网络编程中，每次接收到数据时都需要创建一个缓冲区对象来存储数据，如果每次都重新创建和销毁缓冲区对象，那么会产生大量的开销，尤其是在高并发的情况下，可能会消耗大量的 CPU 时间。因此使用对象池技术来存储和复用这些缓冲区对象可以有效地提高性能和稳定性。

1. 多线程编程

在多线程编程中，由于线程之间的竞争和协调，可能会导致一些对象频繁地被创建和销毁。如果每次都重新创建和销毁对象，那么可能会产生线程冲突和性能问题。因此使用对象池技术来存储和复用这些对象可以有效地避免这个问题。

请利用本节的框架，假设你是一位游戏开发者，请开放一款游戏，这个游戏包含背景音乐“海阔天空”，动作特效“王者荣耀”等，还可以发挥你的想象，增加任意功能。

4.封装信号槽的观察者模板

在学习的过程中，我们很好学习到信号槽signal的概念，当遇到多线程编程的时候，用到最多的就是互斥锁，高级一点，来个信号量，那么什么是信号槽呢，今天我们就看实际看一下，并看下在工作中它发挥怎么样的作用。

4.1.工程作用

信号槽，顾名思义，是通过信号的方式通知一个任务开始或者结束，信号的发起者至关重要，当然，任务的执行者也必不可少。在工程中，既然是多任务，肯定用在多线程编程中，信号和槽机制通过信号和槽的连接，实现了对象之间的松耦合，既一个对象发出 信号，其他对象接收到该信号并执行相应的槽函数。具体步骤如下：

> 1. 在对象中定义信号，可以使用Q_OBJECT宏来声明对象类，来支持信号和槽机制。
> 2. 在对象中定义槽函数，用来处理信号。
> 3. 在对象中实现信号的发射函数，用来发出信号。
> 4. 在对象之间建立信号和槽连接，可以使用connect()函数将信号和槽连接起来。

4.2.框架实现

基于上述讨论，本节我们自己实现一个信号和槽函数，链接的方式也自己实现，并可以自定义扩展，并给出多个测试用例，这种实现方式在工程实践中可以被实际需求扩展，发挥重要作用。

```
#include <algorithm>
#include <functional>
#include <list>
#include <memory>
#include <mutex>
#include <iostream>
/*

封装信号槽接口实践，用于观察者模式

//各个子类可以调用

1，模板参数，多参数传递
2，模板类和类模板
3，std::function
4，callback用法
5，智能指针
6，运算符重载
7，std::remove_if
8，观察者模式
9，operator()
10，using

公众号：码出名企路
```

```
视频，代码，加群和小伙伴一起学习

工作中是怎样写 C++的?

*/

//知识点：类的前置声明
template <typename... Args>
class Slot;
// using SlotPtr = std::shared_ptr<Slot<Args...>>;

template <typename... Args>
class Connection;

template <typename... Args>
class Signal{ //封装信号类
    public:

        using Callback = std::function<void(Args...)>;//函数模板，已经讲过
        //知识点：shared_ptr
        using SlotPtr = std::shared_ptr<Slot<Args...>>;
        using SlotList = std::list<SlotPtr>;
        using ConnectionType = Connection<Args...>;

        Signal(){

        }

        virtual ~Signal(){
            this->DisconnectAllSlots();
        }

        void DisconnectAllSlots(){
           std::lock_guard<std::mutex> lock(mutex_);
           for(auto& slot:slots_){
            slot->Disconnect();
           }

           slots_.clear();
        }
        //知识点：括号运算符
        void operator()(Args... args){
            SlotList local;
            //知识点：函数中的括号阈
            {
                std::lock_guard<std::mutex> lock(mutex_);
                for(auto& slot:slots_){
                    local.emplace_back(slot);
                }
            }

            if (!local.empty()){
                for(auto& slot:local){
```

```
                (*slot)(args...);
            }
        }

        ClearDisconnectedSlots();
    }


    ConnectionType Connect(const Callback& cb){
        auto slot = std::make_shared<Slot<Args...>>(cb);
        {
           //锁的作用域限制
            std::lock_guard<std::mutex> lock(mutex_);
            slots_.emplace_back(slot);
        }

        return ConnectionType(slot,this);
    }
    //信号与槽断开连接
    bool Disconnect(const ConnectionType& conn){
        bool find  =false;
        {
            std::lock_guard<std::mutex> lock(mutex_);
            for(auto& slot:slots_){
                if(conn.HasSlot(slot)){
                    find = true;
                    slot->Disconnect();
                }
            }
        }

        if(find){
            ClearDisconnectedSlots();
        }

        return find;
    }

private:
    //清楚所有的连接
    void  ClearDisconnectedSlots(){
        std::lock_guard<std::mutex> lock(mutex_);
        //知识点: list的erase函数, std::remove_if
        slots_.erase(std::remove_if(slots_.begin(), slots_.end(),[](
            const SlotPtr& slot){
                return !slot->connected();
            }), slots_.end());
    }

    SlotList slots_;
    std::mutex mutex_;
```

```
};

template <typename... Args>
class Connection{ //信号与槽函数连接
    public:
        using SlotPtr = std::shared_ptr<Slot<Args...>>;
        using SignalPtr = Signal<Args...>*;

        Connection():slot_(nullptr),signal_(nullptr){

            std::cout<<"Conn struct()"<<std::endl;

        }
        Connection(const SlotPtr& slot, const SignalPtr&
signal):slot_(slot),signal_(signal){
            std::cout<<"Conn struct(X,X)"<<std::endl;
        }

        virtual ~Connection(){
            slot_ = nullptr;
            signal_ = nullptr;
        }
        //知识点：重载运算符号 =
        Connection& operator=(const Connection& another){
            if (this != &another){
                this->slot_ = another.slot_;
                this->signal_ = another.signal_;
            }
            return *this;
        }

        bool HasSlot(const SlotPtr& slot) const{
            if(slot != nullptr && slot_ !=nullptr){
                return slot_.get() == slot.get();
            }

            return false;
        }

        bool IsConnected()const{
            if(slot_){
                return slot_->connected();
            }

            return false;
        }

        bool Disconnect(){
            if(signal_ && slot_){
                return signal_->Disconnect(*this);
            }
            return false;
        }
```

```
    private:
        SlotPtr slot_;
        SignalPtr signal_;
};



template <typename... Args>
class Slot{//信号槽
    public:
        using Callback = std::function<void(Args...)>;
        Slot(const Slot& another):cb_(another.cb_), connected_(another.connected_){
            std::cout<<"slot struct"<<std::endl;
        }

        explicit Slot(const Callback& cb, bool connected =
true):cb_(cb),connected_(connected){

            std::cout<<"explicit slot struct"<<std::endl;

        }

        virtual ~Slot(){

        }
        //知识点：重载括号
        void operator()(Args... args){
            if (connected_ && cb_){
                cb_(args...);//回调函数执行
            }
        }

        void Disconnect(){
            connected_ =false;
        }

        bool connected() const{
            return connected_;
        }


    private:
        Callback cb_;
        bool connected_ = true;
};

//真正将信号和槽函数进行统一管理的实现
class TopologyManager{
    public:
        using ChangeSignal = Signal<const std::string&>;
        using ChangeFunc = std::function<void(const std::string&)>;
```

```
        using ChangeConnection = Connection<const std::string&>;

        TopologyManager(){

        }
        virtual ~TopologyManager(){
            this->Shutdown();
        }
        void Shutdown(){
            /*
            自定义一些实现
            */
           change_signal_.DisconnectAllSlots();
        }
        //信号与槽函数建立关联关系
        ChangeConnection AddChangeListener(const ChangeFunc& func){
            return change_signal_.Connect(func);
        }

       //移除关联关系
        void RemoveChangerListener(const ChangeConnection& conn){
            auto local_conn = conn;
            local_conn.Disconnect();
        }

    private:
        ChangeSignal change_signal_;

};

class Test{
    public:
        Test():topologt_(new TopologyManager()){

        }
        ~Test(){

        }
        void test_listener(){
            std::string sum = "lyy";
            std::cout<<"topologt_: "<<topologt_<<std::endl;
            auto conn = topologt_->AddChangeListener([&sum](const std::string& msg){
                sum = sum + msg;
            });

            /*
             增加多个观察者
            */
            std::cout<<"add "<<sum<<std::endl;

            topologt_->RemoveChangerListener(conn);

        }
```

```
void slotTest_zeroInputParam(){

    char ch = '0';
    Slot<> slot_a([&ch](){
        ch = 'a';
    });

    slot_a.connected();

    slot_a();

    std::cout<<"test 1-1: "<<ch<<std::endl;

    slot_a.Disconnect();

    slot_a.connected();
    ch = '0';
    slot_a();
    std::cout<<"test 1-2: "<<ch<<std::endl;

    Slot<> slot_b([&ch](){
        ch = 'b';
    }, false);
    slot_b.connected();
    std::cout<<"test 1-3: "<<ch<<std::endl;

    ch = '0';
    slot_b();
    std::cout<<"test 1-4: "<<ch<<std::endl;

    Slot<> slot_c(nullptr);
    slot_c();
}

void slotTest_twoInputParam(){
    int sum = 0;
    Slot<int,int> slot_a([&sum](int lhs, int rhs){
        sum = lhs + rhs;
    });
    slot_a.connected();

    int lhs =1, rhs =2;
    slot_a(lhs,rhs);
    std::cout<<"test 2-1: "<<sum<<" "<<lhs + rhs<<std::endl;

    Slot<int,int> slot_b(slot_a);
    lhs =3;
    rhs =4;
    slot_b(lhs, rhs);
    std::cout<<"test 2-2: "<<sum<<" "<<lhs + rhs<<std::endl;

    slot_b.Disconnect();
    slot_b.connected();
```

```
            sum = 0;
            lhs = 5;
            rhs = 6;
            slot_b(lhs,rhs);
            std::cout<<"test 2-3: "<<sum<<" "<<lhs + rhs<<std::endl;

        }

        void connTest_nullSignal(){
            Connection<> conn_a;
            std::cout<<conn_a.IsConnected()<<std::endl;
            std::cout<<conn_a.Disconnect()<<std::endl;
            std::cout<<conn_a.HasSlot(nullptr)<<std::endl;

            auto slot = std::make_shared<Slot<>>([](){

            });

            Connection<> conn_b(slot,nullptr);
            std::cout<<conn_b.IsConnected()<<std::endl;
            std::cout<<conn_b.Disconnect()<<std::endl;
            std::cout<<conn_b.HasSlot(slot)<<std::endl;

            std::cout<<conn_a.HasSlot(slot)<<std::endl;

            conn_b  =conn_b;
            conn_a = conn_a;
            std::cout<<conn_a.IsConnected()<<std::endl;
            std::cout<<conn_a.Disconnect()<<std::endl;
            std::cout<<conn_a.HasSlot(slot)<<std::endl;

            Signal<> sig;
            Connection<> conn_c(nullptr, &sig);
            std::cout<<conn_c.Disconnect()<<std::endl;

        }

        void signalTest_module(){
                Signal<int, int> sig;

                int sum_a = 0;
                auto conn_a = sig.Connect([&sum_a](int lhs, int rhs) { sum_a = lhs + rhs;
});

                int sum_b = 0;
                auto conn_b = sig.Connect([&sum_b](int lhs, int rhs) { sum_b = lhs + rhs;
});

                int lhs = 1, rhs = 2;
                sig(lhs, rhs);

                std::cout<<"test 4-1: "<<sum_a<<" "<<lhs + rhs<<std::endl;
```

```
                std::cout<<"test 4-2: "<<sum_b<<" "<<lhs + rhs<<std::endl;


                Connection<int, int> conn_c;
                std::cout<<sig.Disconnect(conn_c)<<std::endl;
                std::cout<<sig.Disconnect(conn_b)<<std::endl;
                sum_a = 0;
                sum_b = 0;
                lhs = 3;
                rhs = 4;
                sig(lhs, rhs);

                std::cout<<"test 4-3: "<<sum_a<<" "<<lhs + rhs<<std::endl;
                std::cout<<"test 4-4: "<<sum_b<<" "<<lhs + rhs<<std::endl;
                sig.DisconnectAllSlots();
                sum_a = 0;
                sum_b = 0;
                lhs = 5;
                rhs = 6;
                sig(lhs, rhs);

                std::cout<<"test 4-5: "<<sum_a<<" "<<lhs + rhs<<std::endl;
                std::cout<<"test 4-6: "<<sum_b<<" "<<lhs + rhs<<std::endl;
        }

    private:
        TopologyManager* topologt_;
};


int main()
{
    Test test_;
    test_.slotTest_zeroInputParam();
    test_.slotTest_twoInputParam();
    test_.connTest_nullSignal();
    test_.signalTest_module();
    test_.test_listener();
}
```

4.3.知识点拆分

我们通过本节的学习，发现了一种信号与槽函数建立连接的新方式，可以不用信号量了，本节涉及的知识点大多数在前面章节中已经接触，根据读者的掌握程度，相信在阅读和理解代码方面不是问题。下面我们主要学习涉及到的新的知识点：

1，类的前置声明

2，智能指针std::shared_ptr

3，重载括号（）运算符

4，函数中加{}作用域

5，std::remove_if函数

6，重载等号=运算符

下面我们对以上知识点在本案例中的实际应用和具体的细节进行展开说明，帮助读者更好将其用在实际工作中去。

4.3.1.为什么要类的前置声明

第一部分：类的前置声明在工程中经常用到，本案例的用处在：

```
template <typename... Args>
class Slot;
```

第二部分：实际工作中什么时候用类的前置声明呢？在声明之后是不是就可以任意调用类的成员函数了呢？下面给出具体讲解：

类的前置声明是一种在使用类之前声明该类的方式。这种声明只会告诉编译器有这么一个类的存在，而不会提供有关该类的详细信息。因此，在使用前置声明声明了一个类后，不能直接使用该类的成员函数或数据成员。如果你尝试这么做，编译器会报错，因为它不知道该类的详细信息。

但是，你可以使用指向该类对象的指针或引用，因为指针或引用本质上只是一个内存地址，不需要知道类的详细信息。在这种情况下，编译器只需要知道这个类的存在就可以了。

要在类的前置声明后使用类的成员函数或数据成员，你需要将类的定义放在使用之前。这样，编译器就能够访问该类的详细信息。

以下是示例代码：

```
// MyClass.h
#ifndef MYCLASS_H
#define MYCLASS_H

class MyOtherClass; // 前置声明

class MyClass
{
public:
    void doSomething();
private:
    MyOtherClass* pointer; // 使用指针不需要知道类的详细信息
};

#endif

// MyOtherClass.h
```

```
#ifndef MYOTHERCLASS_H
#define MYOTHERCLASS_H

class MyOtherClass
{
public:
    void doSomethingElse();
};

#endif

// MyClass.cpp
#include "MyClass.h"
#include "MyOtherClass.h" // 在使用前包含类的定义

void MyClass::doSomething()
{
    MyOtherClass other; // 创建对象需要知道类的详细信息
    other.doSomethingElse();
}
```

在这个示例代码中，我们前置声明了MyOtherClass，这样ClassA中就可以使用指向MyOtherClass对象的指针或引用。在MyClass.cpp中包含了MyOtherClass.h头文件，这样我们就可以在其中使用MyOtherClass的详细定义了。

4.3.2.智能指针std::shared_ptr

第一部分：智能指针发挥的作用无处不在，std::unique_ptr已经讲过，std::shared_ptr的用处在本案例中体现如下：

```
ConnectionType Connect(const Callback& cb){
auto slot = std::make_shared<Slot<Args...>>(cb);
```

第二部分：std::shared_ptr的具体用法如下：

在C++中，智能指针是一种通过模板类来实现的指针类型，它允许你动态地分配内存，并自动进行资源管理，从而避免了内存泄漏和悬挂指针等问题。

std::shared_ptr是一种常用的智能指针类型，它可以用来管理动态分配的内存。std::shared_ptr使用引用计数来跟踪有多少个std::shared_ptr指向同一个对象。当最后一个指向该对象的std::shared_ptr被销毁时，其所管理的内存也会自动被释放。这样，就可以避免出现内存泄漏的问题。

以下是一个简单的示例代码：

```
#include <iostream>
#include <memory>

int main()
```

```
{
    std::shared_ptr<int> sp1(new int(1)); // 创建std::shared_ptr对象
    std::cout << "sp1: " << *sp1 << " use_count: " << sp1.use_count() << std::endl;

    {
        std::shared_ptr<int> sp2 = sp1; // 复制构造函数，引用计数加1
        std::cout << "sp1: " << *sp1 << " use_count: " << sp1.use_count() << std::endl;
        std::cout << "sp2: " << *sp2 << " use_count: " << sp2.use_count() << std::endl;
    } // sp2离开作用域，引用计数减1

    std::cout << "sp1: " << *sp1 << " use_count: " << sp1.use_count() << std::endl;

    return 0; // sp1离开作用域，内存自动销毁
}
```

在这个示例代码中，我们首先创建了一个std::shared_ptr对象sp1，并通过new运算符动态分配了一个int类型的内存，值为1。接着打印了sp1的值和引用计数。

然后，我们创建了另一个std::shared_ptr对象sp2，将其初始化为sp1的值。注意，这里使用的是复制构造函数，而不是赋值运算符。因为复制构造函数会将引用计数加1，而赋值运算符则不会。

接着，我们打印了sp1和sp2的值和引用计数，并输出包含这两个std::shared_ptr对象的作用域。当sp2离开作用域时，引用计数减1。

最后，我们再次打印了sp1的值和引用计数。因为此时没有其他std::shared_ptr对象指向这个内存，所以其引用计数为1。当sp1离开作用域时，其所管理的内存也会被自动释放。

我们在看一个例子：

当使用C++类时，std::shared_ptr可以用来管理动态分配的类实例的内存，从而简化内存管理的过程。下面是一个示例：

```
#include <memory>
#include <iostream>

class MyClass {
public:
    MyClass() {
        std::cout << "MyClass constructed." << std::endl;
    }
    ~MyClass() {
        std::cout << "MyClass destroyed." << std::endl;
    }
};

void doSomething() {
    std::shared_ptr<MyClass> ptr(new MyClass);
    // 这里可以使用ptr来使用MyClass的成员函数或访问MyClass对象中的数据成员。
}
```

```
int main() {
    doSomething();
    return 0;
}
```

在这个示例代码中，我们定义了一个MyClass类用于管理一些数据，然后我们使用std::shared_ptr来动态地分配一个MyClass对象的内存。在代码中，我们调用doSomething函数，在该函数中创建了一个std::shared_ptr对象ptr，它指向一个新的MyClass对象。

由于std::shared_ptr是一个智能指针，因此当这个std::shared_ptr对象ptr超出其作用域时会自动销毁。这意味着MyClass对象也会被销毁，并释放其内存。同时，当多个std::shared_ptr对象指向同一个MyClass对象时，引用计数会自动跟踪管理，并在没有任何std::shared_ptr对象引用该对象时自动释放其内存。这样，就可以避免内存泄漏等问题。

这只是std::shared_ptr的应用之一，它还有许多其他应用。因此，在C++中使用std::shared_ptr可以提供一种有效的方式来管理动态分配的内存。

4.3.3.重载括号（）运算符

第一部分：重载运算符还是很常见的，主要有比较运算符，等号运算符，括号运算符号等，在本案例中有两处用到了括号运算符号：

```
void operator()(Args... args){
if (connected_ && cb_){
cb_(args...);
}
}
```

第二部分：其中括号运算符号什么时候发挥它的作用呢？下面具体讲解一下：

在C++中，可以通过重载括号（）运算符来实现类的实例对象像函数一样被调用的效果。通过重载括号运算符，可以将类的实例对象作为函数使用，并访问对象的内部成员变量和方法。在C++中，重载括号运算符是通过在类的定义中定义一个名为operator()的函数实现的。

以下是一个示例代码：

```
#include <iostream>

class DemoClass {
public:
    DemoClass(int value) {
        this->value = value;
    }

    int operator()(int x) {
        return this->value + x;
    }
```

```
private:
    int value;
};

int main() {
    DemoClass obj(7);
    std::cout << "The output is: " << obj(3) << std::endl;
    return 0;
}
```

在这个示例代码中，我们定义了一个名为DemoClass的类，并在其中实现了一个operator()函数。这个函数接收一个整数x，并返回对象内部成员变量value和x的和。这个operator()函数的实现使得DemoClass类的实例可以像函数一样被调用。在main函数中，我们创建了一个名为obj的DemoClass实例对象，并将其值初始化为7。然后我们调用这个对象，将参数设置为3，并将结果打印出来，输出结果为10。

重载括号运算符可以让类的实例对象像函数一样被调用，使得代码更加简洁易懂。它还可以隐式地把类的实例对象作为函数传递，使得在调用函数时省略对象参数。但是，在使用重载括号运算符时，需要小心处理参数，并避免与其他重载运算符造成混淆。

4.3.4.函数中加 {} 的块作用域

第一部分：很多时候，我们会看到一个函数体内经常用多个{}的添加，在这些 {}中，作者实现一些简单的代码，在本例子中的应用如下：

```
bool Disconnect(const ConnectionType& conn){
bool find =false;
{
std::lock_guard<std::mutex> lock(mutex_);
for(auto& slot:slots_){
if(conn.HasSlot(slot)){
find = true;
slot->Disconnect();
}
}
}
if(find){
ClearDisconnectedSlots();
}
return find;
}
```

第二部分：{}作用域实际发生了什么样的作用呢，为什么会出现它，它内部的变量作用域存活周期有多长，带着这些疑问我们看下具体解释：

在C++中，{}（花括号）通常用作代码块的开始和结束标记，它们用于定义作用域。

C++中的作用域是指一个代码块内声明的变量或函数，在代码块内部是可见的，但在代码块外部不可见。同时，代码块可以用于限制其内部变量和函数在外部所需的可见性。

因此，当在函数中定义了一个局部代码块时，可以使用花括号将其括起来，在该代码块内部声明的变量和函数仅在该块内部可见。这通常被称为局部作用域（或者称为块作用域）。

以下是一个示例：

```
#include <iostream>

void testFunc() {
    int x = 5;
    std::cout << "x in the outer scope: " << x << '\n';
    {
        int x = 7;
        std::cout << "x in the inner scope: " << x << '\n';
    }
    std::cout << "x in the outer scope again: " << x << '\n';
}

int main() {
    testFunc();
    return 0;
}
```

在这个示例代码中，我们定义了一个testFunc函数，并在其内部定义了一个名为x的变量。接着，在函数内部又定义了一个局部作用域，其中重新声明了一个名为x的变量，其值为7。在该代码块内部使用的x变量不会影响testFunc函数外部的x变量，它只在该代码块内部有效。在该代码块结束之后，testFunc函数在外部的x变量的值仍旧是5。

因此，在函数中使用{}（花括号）作为代码块的开始和结束标记，可以限制代码块内部的变量和函数的可见性，并且使代码更加易于理解和维护。具体可以解释为以下三种应用：

> 1. 限制变量作用域：在代码块中声明的变量仅在该块内部有效，这使得代码更加易于理解和维护。例如，如果有一个复杂的逻辑，其中需要定义大量的变量，可以将这些变量的声明放在一个代码块中，从而使代码更加可读且易于维护。
> 2. 限制函数命名空间：在函数内部，可以使用块作用域来限制一组函数的名称空间，这有助于防止不同函数之间的命名冲突。例如，一个函数集合可能有多个函数名相同或相似，需要在不同上下文中使用它们。
> 3. 隐藏实现细节：在类的实现中，可以使用块作用域来将部分函数的实现细节隐藏在类的实现中，仅暴露必要的接口。这可以使类更加封装，更容易维护和修改。

4.3.5.std::remove_if删除函数

第一部分：删除函数有许多，能想到的有remove, remove_if,erase,它们之间有哪些区别呢。本案例中实际用到了std::remove_if如下：

```
slots_.erase(std::remove_if(slots_.begin(), slots_.end(),[](
const SlotPtr& slot){
return !slot->connected();
}), slots_.end());
```

第二部分：std::remove_if的具体用法如下：

在C++中，std::remove_if是一个可以在容器中删除指定元素的算法。它可以用来删除满足一定条件的容器元素。std::remove_if算法接受三个参数：第一个参数是要操作的容器的起始迭代器；第二个参数是要操作的容器的结束迭代器，指向容器中的第一个未处理元素；第三个参数是一个函数对象或函数指针，用于确认每个元素是否需要被删除。当元素需要被删除时，std::remove_if会将其移到容器末尾，并返回指向新的末尾元素的迭代器，新的末尾元素之后的部分则保持不变。

> 1. 自定义删除谓词：除了lambda表达式，还可以使用函数对象来作为删除谓词。这些函数对象可以有自定义的操作和删除逻辑，从而满足更加特定和需求。
> 2. 删除重复元素：当需要删除容器中的重复元素时，可以使用std::remove_if配合容器的sort函数进行实现。
> 3. 区分符合条件和不符合条件的元素：有时候需要找出容器中符合条件的元素，同时不删除其他元素。在这种情况下，可以使用std::partition和std::partition_point算法。

下面是一个使用std::remove_if算法的示例：

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> nums{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // 删除所有偶数
    auto end = std::remove_if(nums.begin(), nums.end(), [](int i) { return i % 2 == 0; });

    // 输出删除后的容器元素
    for (auto it = nums.begin(); it != end; ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

在这个示例中，我们创建了一个std::vector容器nums，其中包含了1-10的整数。然后，我们使用std::remove_if算法删除了所有偶数。在这个算法中，我们使用一个lambda表达式来判断一个元素是否为偶数。最后，我们使用循环输出删除后的容器元素，其中包含了所有的奇数。

需要注意的是，在使用std::remove_if算法时，元素并没有真正被删除。相反，它们被移动到容器末尾，并返回指向新的末尾元素的迭代器。如果需要，可以使用容器的erase函数来真正地删除这些元素。

总之，std::remove_if算法是一个方便且常用的算法，它提供了一个简单的方法来删除容器中的特定元素。

4.3.6.重载赋值 = 运算符

第一部分：接着上面我们继续讲解重载等号 = 运算符号，先看看在代码中的实际应用：

```
Connection& operator=(const Connection& another){
if (this != &another){
this->slot_ = another.slot_;
this->signal_ = another.signal_;
}
return *this;
}
```

第二部分：看下具体细节：

在C++中，重载运算符可以用来改变运算符与类对象之间的行为。对于赋值运算符=，可以通过重载运算符来改变对象之间的赋值行为。因为C++中默认的赋值操作是浅拷贝，它只会复制对象的地址，而没有复制对象的内容，造成在多个对象中共享同一块内存，容易出现潜在的问题。因此，通过重载赋值运算符=实现深拷贝，即从一个对象复制到另一个对象时，会复制内容，并且不会共享同一块内存，可以避免潜在的问题。

要想在自定义类中重载赋值运算符=，需要按照以下方式实现：

```
class MyClass {
private:
    int* pData;

public:
    MyClass(const MyClass& other);  // 拷贝构造函数
    MyClass& operator=(const MyClass& other);  // 赋值运算符=
    ~MyClass();  // 析构函数

    // 实现其他类方法
};

MyClass::MyClass(const MyClass& other) {
    // 实现拷贝构造函数
}

MyClass& MyClass::operator=(const MyClass& other) {
    if (this != &other) { // 避免自赋值
        delete pData;  // 清除现有的内存资源

        // 深度拷贝other中的值，分配内存资源
        pData = new int(*other.pData);
```

```
        // 实现其他赋值逻辑
    }
    return *this;
}

MyClass::~MyClass() {
    delete pData;  // 释放对象的内存资源
}
```

在这个示例代码中，我们首先定义了一个MyClass类，并在其中实现了一个int类型的指针pData。接着，我们定义了拷贝构造函数和析构函数，用于对象的拷贝和释放内存资源。

然后，我们重载了赋值运算符=，在其中实现了自定义的赋值逻辑。在这个例子中，我们使用if语句来避免自赋值，因为自赋值会导致对象资源的争夺。接着，我们删除旧的对象，为新的对象分配新的内存空间，并将副本值复制到新的内存中，并返回当前对象的指针。当我们调用MyClass类的赋值运算符时，会执行自定义的赋值逻辑。

总之，重载赋值运算符=可以改变类对象之间的赋值行为，实现深拷贝，以避免潜在的内存问题。需要在赋值运算符=实现中记得避免自赋值。

4.4.本节大作业

在大型C++项目中，信号和槽函数的使用可以提高代码的可读性和模块之间的通信效率，使得项目开发更加灵活和高效，模块之间的通信更加灵活和有效。 假设你正在为学校图书馆开发一款图书管理系统，该系统管理查阅，借书和还书至少三大模块，其余模块可以自定义扩展，这些模块之间是可以进行通信连接的，可以互相知道各自模块的状态，请利用本节工程架构，开发完成此此项目，并给出测试用例。

5.自定义删除器应用与设计

我们都知道stl容器有自己的删除原则，默认不写，同样，智能指针也有自己的删除规则，默认不写，但是如果我们想打破固有的删除或者析构规则，就需要自己定义删除器了。

5.1.工程作用

自定义删除器是在C++中使用智能指针管理资源时非常重要的概念。在默认情况下，智能指针使用操作符 delete 来释放对应资源的内存，但在一些情况下，你可能需要使用自定义删除器来管理资源的释放过程。

自定义删除器可以是一个函数指针、一个函数对象（如 Lambda 表达式）或者一个类的成员函数指针。它负责在智能指针释放资源之前执行一些额外的操作，如日志记录、资源计数、或者调用有特殊释放方法的系统函数等。

5.2.四种实现方式

自定义删除器是C++中使用智能指针管理资源时非常重要的一部分，它可以实现对资源的更加精细的管理，以及为资源释放操作增加额外的操作。你需要根据具体情况，合理选择自定义删除器，从而让程序更加健壮、易用。本节我们将学习自定义删除器的几种方式。读者可以根据自己的需要去随机利用以下几种方式的一种或者多种。

```
#include <string>
#include <iostream>
#include <memory>
#include <vector>
#include <functional>

/*
1,  自定义的删除器的3种方法
2,  std::move转移权的交接
3,  std::vector初始化定义大小
4,  extern C
5,  std::function, std::bind
6,  static_cast , reinterpret_cast
7,  __func__
8,  自定义删除程序将c样式
    内存管理包装到std::unique_ptr中的一种方法

WXGZH:  码出名企路

*/

//方式三：仿函数
class Info{

    public:
        Info(){}
        ~Info(){}
};

struct uniqDeleter
{
    public:
    //知识点，前面章节已经学过
        void operator()(Info* info){

            if (info != nullptr)
            {
                delete info;
                info = nullptr;
                std::cout<<"ptr del 3"<<std::endl;
            }
```

```
        }
};

//方式四
extern "C" {

struct ABC { };

enum free_type_e {
    FREE_ALL,
    FREE_SOME
};

typedef void (free_f)(enum free_type_e free_type, void *ptr);
struct some_c_ops { free_f* free_op; };

void MY_free(enum free_type_e free_type, void *ptr)
{
    printf("%s:%d ptr=%ld \n", __func__, __LINE__, (long)ptr);
    (void)free_type;
    free(ptr);
}

}; // extern"C"

template<typename T>
using c_unique_ptr = std::unique_ptr<T,std::function<void(T*)>>;

template <typename T>
c_unique_ptr< T > make_c_unique(some_c_ops* op, free_type_e free_type)
{
    return c_unique_ptr< T >(static_cast<T*>(calloc(1, sizeof(T))),
                             std::bind(op->free_op, free_type, std::placeholders::_1));
}

void foo(c_unique_ptr<ABC> ptr)
{
    std::cout << __func__ <<":" << __LINE__
        <<" ptr=" << reinterpret_cast<size_t>(ptr.get()) <<std::endl;
}


int main()
{
    //方式一: 普通函数
    {
        std::unique_ptr<int[], void(*)(int *)> ptr(new int[10],
                    [](int *p) {delete []p;
                    std::cout<<"ptr del 1"<<std::endl; });
        ptr[9] = 9;
    }

    //方式二: functional
```

```
    {
        std::unique_ptr<int[], std::function<void(int *)>> ptr(new int[10],
                 [&](int *p) {delete  []p;
                std::cout<<"ptr del 2"<<std::endl;});
        ptr[9] = 9;

    }

    //方式三：仿函数
    std::vector<int> v1[5];
    v1->push_back(11);

    std::vector<std::unique_ptr<Info,uniqDeleter> > v2[10];
    std::unique_ptr<Info,uniqDeleter> info(new Info);
    v2->push_back(std::move(info));
    //v2->push_back(info);

   //方式四
    some_c_ops ops = { MY_free };
    c_unique_ptr<ABC> ptr = make_c_unique<ABC>(&ops, FREE_ALL);
    std::cout << __func__ <<":" << __LINE__
        <<" ptr=" << reinterpret_cast<size_t>(ptr.get()) << std::endl;

    foo(std::move(ptr));

    std::cout << __func__ <<":" << __LINE__
        <<" ptr=" << reinterpret_cast<size_t>(ptr.get()) << std::endl;
}
```

5.3.知识点拆分

本节我们主要讲解了如何自定义删除器，框架不是标准统一的，因此也没比较用统一架构，只要记住以上四种实现方式，我们就可以进行扩展应用。它们涉及的主要知识点如下：

1，**func**的意义

2，extern "C"的应用

3，calloc的作用

4，std::bind函数绑定

5，右值引用std::move

下面我们结合本章节的举例案例来详细说明以下上面知识点的细节。

5.3.1.func特殊标识符

第一部分：经常看到在工程中打印的信息里有**func**这个东西，它会打印什么信息呢，在本节的应用如下：

```
std::cout << __func__ <<":" << __LINE__
<<" ptr=" << reinterpret_cast<size_t>(ptr.get()) <<std::endl;
```

第二部分：下面我们认识一下它的具体作用。

`__func__`是C++语言中的一个特殊标识符，它用于返回当前函数的名称，是一个预定义的局部静态变量。在C++11标准中，该标识符已经升级为 `constexpr` 表达式。

`__func__`的作用包括：

1. 调试：在代码中输出当前函数的名称是很常见的调试手段，可帮助开发人员快速地识别问题代码段和问题来源。例如，在抛出异常时，通常需要将异常信息和当前函数的名称一起输出以方便调试。
2. 代码审查：对于需要代码审查的项目，使用 `__func__` 标识符可以增强代码的可读性和可维护性，因为审查人员可以更容易地理解代码中使用的函数及其作用。
3. 记录日志：在日志记录器中，使用 `__func__` 标识符可以自动记录每条日志所在的函数名称，方便后续的问题排查和调试。

下面是一个简单的示例代码，展示了如何在程序中使用 `__func__` 标识符：

```
#include <iostream>

void myFunc() {
    std::cout << "Function name: " << __func__ << std::endl;
}

int main() {
    myFunc();  // 打印当前函数名称

    return 0;
}
```

在此示例代码中，我们定义了一个名为myFunc的函数，它使用 `__func__` 标识符来输出自己的名称。在主函数中，我们调用了myFunc函数，并输出了函数的名称。

总之，`__func__` 标识符是在C++语言中用于返回当前函数名称的特殊标识符。它可以用于调试、代码审查和日志记录等方面，提高代码的可读性和可维护性。

5.3.2.extern "C"的应用

第一部分：在C++和C混合使用的时候，会加入extern "C"符号，本节代码如下：

```
extern "C" {

}
```

第二部分：实际作用如下：

`extern "C"` 是C++中一种特殊的语法标记，它用于将符号名称修饰为C语言风格的符号，以便C++代码与C代码之间进行链接和交互。在C++中，每个符号（函数、变量等）的名称被翻译为一个新的、没有特殊字符的二进制名称，这样就保证了不同编译单元中定义的名称不会产生冲突。

举个例子，如果我们在C++中定义如下的函数：

```c++
// my_function.cpp
int my_function(int a, int b) {
    return a + b;
}
```

那么编译器将生成一个符号名为 `_Z11my_functionii` 的函数名。而在C语言中，该函数名将会以可读性更高的方式表示：

```c
// my_function.c
int my_function(int a, int b) {
    return a + b;
}
```

为了允许C++代码和C代码的链接和交互，我们可以在C++代码中使用 `extern "C"` 标记。

下面是一个示例演示如何在C++中使用 `extern "C"` 标记修饰一个函数：

```c++
// my_function.cpp
#include <iostream>

extern "C" {
    int my_function(int a, int b) {
        return a + b;
    }
}

int main() {
    std::cout << my_function(3, 4) << std::endl;  // 调用 C 风格的 my_function 函数

    return 0;
}
```

在这个例子中，我们使用 `extern "C"` 标记将 C 风格的 my_function 函数导出到 C++。当代码被编译时，my_function 函数将使用 C 语言的名称规则，因此可以与 C 代码之间进行交互。

需要注意的是，`extern "C"` 标记只适用于 C 风格的符号名。根据这个标记，编译器搜索定义了带有 C 风格命名的符号的库。若找到，则使用符号名进行链接。因此，C++函数不能使用 `extern "C"` 标记，以及任何其他旨在重载函数名的修饰符，因为它们与 C 风格的符号不兼容。

总之，`extern "C"` 是C++中一种用于将符号名称修饰为C语言风格的符号的标记。它使得C++代码能够与C代码之间进行链接和交互。

5.3.3.calloc的作用

第一部分：C语言标准库的应用，区别与std::calloc，它在本节中的应用如下：

```
return c_unique_ptr< T >(static_cast<T*>(calloc(1, sizeof(T))),
std::bind(op->free_op, free_type, std::placeholders::_1));
```

第二部分：具体细节看下面解释。

`calloc` 是一个 C 语言标准库函数，用于在内存中动态分配指定数量和大小的字节，并将它们初始化为0。在 C++ 语言中也可以使用 `calloc` 函数，不过更常见的是使用 `new` 运算符或 `std::malloc` 函数。

下面是一个示例演示如何使用 `calloc` 函数在 C++ 中进行动态内存分配：

```
#include <iostream>
#include <cstdlib>

int main() {
    int* p;
    p = (int*) calloc(5, sizeof(int));  // 申请并初始化为 0 的 5 个 int 类型的内存空间

    if (p == nullptr) {
        std::cout << "Memory allocation failed." << std::endl;
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < 5; i++) {
        std::cout << "p[" << i << "] = " << p[i] << std::endl;
    }

    free(p);  // 释放动态分配的内存空间

    return 0;
}
```

在这个例子中，我们使用 `calloc` 函数分配了 5 个 int 类型的内存空间，并将这些内存空间初始化为 0。如果内存分配失败，程序会通过 `exit()` 函数终止运行。接着我们通过循环打印出刚刚分配的内存空间内容，并最后使用 `free()` 函数释放内存空间。

需要注意的是，尽管 `calloc` 在 C++ 中仍然可用，但优先使用 `new` 和 `std::malloc` 是比较好的选择。因为 `new` 运算符和 `std::malloc` 函数在 C++ 中是更常见的动态内存分配方式，它们更加面向对象和更容易与 C++ 标准库及其容器类一起使用。

下面是一个示例演示如何使用 `new` 运算符在 C++ 中动态分配内存：

```
#include <iostream>

int main() {
    int* p = new int[5]{0};  // 申请并初始化为 0 的 5 个 int 类型的内存空间

    for (int i = 0; i < 5; i++) {
        std::cout << "p[" << i << "] = " << p[i] << std::endl;
    }

    delete[] p;  // 释放动态分配的内存空间

    return 0;
}
```

在这个例子中，我们使用 `new` 运算符动态分配了 5 个 int 类型的内存空间，并将这些内存空间初始化为 0。最后，我们使用 `delete[]` 运算符来释放内存空间。

5.3.4.std::bind函数绑定

第一部分：函数绑定std::bind在工程中应用很广泛，本节应用如下：

```
return c_unique_ptr< T >(static_cast<T*>(calloc(1, sizeof(T))),
std::bind(op->free_op, free_type, std::placeholders::_1));
```

第二部分：std::bind的详细用法如下：

```
`std::bind` 是 C++11 标准库中的一个函数模板，它可以用于将可调用对象、函数指针或成员函数指针绑定到特定的参数上，生成一个新的可调用对象。其功能与 std::function 有所重叠，但它可以更灵活地对可调用对象进行绑定和调用。

`std::bind` 的语法如下：

```C++
template<class Fn, class... Args>
std::bind(Fn&& fn, Args&&... args);
```
```

```

其中，`Fn` 表示要进行绑定的可调用对象、函数指针或成员函数指针；`Args` 表示绑定的参数列表。`std::bind` 函数的返回值是一个新的可调用对象，如果需要，可以使用 `std::placeholders::_1`、`std::placeholders::_2` 等占位符来代表绑定参数。

下面是一个示例演示如何使用 `std::bind` 绑定一个二元函数：

```c++
#include <iostream>
#include <functional>

int add(int a, int b) {
    return a + b;
}

int main() {
    auto f = std::bind(add, 2, std::placeholders::_1);

    std::cout << f(3) << std::endl;

    return 0;
}
```

在这个例子中，我们定义了一个二元函数 add，然后使用 `std::bind` 将其与 `2` 绑定在一起，生成一个新的可调用对象f。该可调用对象需要再次调用时，传入一个整数参数，这个整数参数与占位符 `_1` 相绑定，即 `_1` 的值为 `3`。因此，最终通过调用 f 函数，返回 `add(2, 3)`，即输出结果为 `5`。

值得注意的是，`std::bind` 生成的可调用对象不再是完全通用的，因为它已经被绑定到了特定的参数上。因此，不同于 `std::function` 对象，它不能临时绑定到不同的参数上，也不能将可调用对象转换为 `bool` 类型，但是这种限制使得 `std::bind` 更加轻量、更适用于一些特定的场景。

总之，`std::bind` 是 C++11 标准库中非常有用的一个函数模板，它可以将可调用对象、函数指针或成员函数指针绑定到特定的参数上，生成一个新的可调用对象，这样就使得可调用对象更加灵活、更加适应不同场景的使用。

5.3.5.右值引用std::move

第一部分：右值引用在本节应用：

```
foo(std::move(ptr));
```

第二部分：什么时候会用到std::move，我们给出了如下解释：

`std::move` 是 C++11 标准库中一个非常有用的函数模板，它用于将对象转移为右值，从而可以转移控制权、避免不必要的拷贝和提高程序性能。`std::move` 通常与 move 语义和右值引用一起使用，以提高程序的效率。
```

`std::move` 的语法如下：

```
template<class T>
constexpr std::remove_reference_t<T>&& move(T&& t) noexcept;
```

在这个语法中，`T` 表示要转移的对象的类型，`std::remove_reference_t<T>` 用于返回对象的准确类型，`&&` 表示返回一个右值引用，`noexcept` 表示该操作不会引发异常。

在实际使用中，`std::move` 通常用于将对象的所有权转移给另一个对象或函数。例如：

```
#include <utility>
#include <vector>

int main() {
    std::vector<int> v1{1,2,3,4,5};
    std::vector<int> v2 = std::move(v1);    // 将 v1 中的所有元素移动到 v2 中
    return 0;
}
```

在这个例子中，我们使用 `std::move` 将 vector 对象 `v1` 中的所有元素转移给 `v2`，在转移后，`v1` 变为空。因为 `std::move` 的本质是将对象的引用转换为右值引用，所以使用 `std::move` 时需要注意：

1. 被移动的对象在其转移到另一个对象后，其值的状态就会不确定，不应再次使用它，除非它被赋予新的值或重新构造。否则将出现未定义行为。
2. 即使使用 `std::move` 将对象转移为右值，目标对象本身不一定是一个右值引用。如果目标对象是一个左值引用，其值也可能在移动后发生更改。

总之，`std::move` 是 C++11 标准库中非常有用的一个函数模板，用于将对象转移为右值，从而转移到另一个对象或函数。使用 `std::move` 可以避免不必要的拷贝，提高程序性能，并避免一些潜在的编程错误。

5.4.本节大作业

请利用本节所有的知识，实现一个图像预处理的功能开发，具体功能自己定义，但是有一点，需要用到自定义的删除器，这个删除器的方法实现主要包括删除次数，图像处理时间等打印，还要有自己思考细节的实现，请完成这项功能。

6.缓冲队列和工作队列

缓冲队列和工作队列，顾名思义，是把需要应用到的元素先放入队列中进行缓存，保证一定数量的数据存在，等需要使用的时候，随时从缓冲队列中取出即可。

6.1.工程作用

专业点讲，缓冲队列适用于需要两个或者多个处理单元之间进行数据异步传输，而避免内存泄露的场景下。它可以减少生产者和消费者之间的竞争关系，提高系统的并发程度和性能，主要发挥的作用如下：

> 1. 多线程数据传输：多线程环境下，线程之间需要进行数据传输或共享数据时可以使用缓冲队列。生产者将数据写入队列，消费者从队列中读取数据，通过缓冲队列可以安全、高效的实现线程之间的通讯。
> 2. 网络数据传输：网络应用中，缓冲队列常用于实现一些常见的网络模型，如 Reactor 和 Proactor 模型。通过缓冲队列，可以有效地处理从网络中读取和写入数据的请求。
> 3. 音视频流处理：在音视频流处理等场景下，需要使用缓冲队列来缓存音视频帧数据，确保数据帧的传输顺序和实时性，提高音视频数据的处理效率和稳定性。
> 4. 数据库操作：在数据库操作中，缓冲队列可以用于异步写入数据库操作，将需要写入数据库的数据缓存到队列中，提高数据的处理速度。

6.2.框架实现

下面，我们就来实现一个工作中使用缓冲队列和工作队列的框架，这种框架用的最多的地方就是在处理视频流的时候，生产者生产出来的视频流先预存在缓冲队列里，消费者去队列中取出来，进行视频的进一步处理。

6.2.1.双队列应用

首先，我们实现一个工作队列和空闲队列相互结合的案例，它的而原理很简单，数据依次送入空闲队列，进行缓存，需要的时候数据从工作队列中取出来进行消费。

```
#include  <deque>
#include <iostream>
#include  <string>
#include <thread>
#include <pthread.h>
#include <time.h>


/*
缓冲队列：工作队列和空闲队列

作用：
(1)把数据缓存起来，用的时候再用：相机的驱动数据，视频流的帧

1，数据依次送入空闲队列，进行缓存；需要用到数据时候从工作队列取出

  1.1  在捕获xx数据的回调的时候将 其放入空闲队列
```

1.2 开启一个线程每隔 50 从工作队列中取出 xxbuf，在此对数据处理
处理完后将节点放回空闲队列

2，原理：
..: 初始化固定数据量的节点放入空闲队列，当 xx 回调产生数据后，从空闲队列 头部 取出一个节点 将产生的每一帧图像 buffer装入，然后入队到工作队列 尾部

..: 处理buffer的线程从工作队列的 头部 取出一个节点中的 buffer进行处理处理完成后将装有 次 buffer的节点中 data设空 并重新放入空闲队列的 头部供下次使用

3，解析

..: 空闲队列：每次从空闲队列中取出一个空节点来装入 xx 数据，所以没必要按照尾进头出的方式保证节点的顺序； 头进头出

..: 工作队列：尾进头出，确保从相机中捕获的数据是连续的，以便后期我们播放出的画面也是连续的，必须保证尾进头出

空闲队列做缓冲，正常情况下，在 33ms内对数据进行的操作可正常完成，则工作队列始终为 0 或1，但是如果长期工作或遇到某一帧数据处理较慢的情况，处理时间大于33ms，则工作队列的长度会增加，我们使用了这样的队列会保护那一帧处理慢的数据在，仍然能够正常处理完

```
/*

1，  线程
2，static 函数
3，队列应用

公众号：码出名企路

*/
//知识点：enum class
typedef enum{
    WORK_QUEUE = 0,
    FREE_QUEUE
}QueueType;

typedef struct NodeQueue
{
    void *data; //存放数据
    size_t size;
    long index;//保证按照顺序索引
    struct NodeQueue *next;
}NODEQUEUE;

//消费队列。队头队尾
typedef struct CustomQueue{
    int size;
    QueueType type;
```

```
    NodeQueue *front;
    NodeQueue *rear;
}CUSTOMQUEUE;//工作队列和空闲队列和始终不变

class ProcessQueue{
    public:
       //两个队列
        CustomQueue *free_queue_;
        CustomQueue  *work_queue_;

        ProcessQueue();
        ~ProcessQueue(){}

        void Init_Queue(CustomQueue *queue,QueueType type);
        //入队
        void  Enque_Queue(CustomQueue *queue, NodeQueue *node);
        //出队
        NodeQueue *Deque_Queue(CustomQueue  *queue);

        void Clear_Custom_Queue(CustomQueue *queue);
        void Free_Node(NodeQueue *node);
        void Reset_Free_Queue(CustomQueue *work_queue,CustomQueue *free_queue);
    private:
         //两个线程锁
        pthread_mutex_t free_queue_mutex;
        pthread_mutex_t work_queue_mutex;

};

const int CustomQueueSize = 3;

//直接分配工作队列和空闲队列的大小
ProcessQueue::ProcessQueue()
{
    free_queue_ = (CustomQueue*)malloc(sizeof(struct CustomQueue));
    work_queue_ =(CustomQueue*)malloc(sizeof(struct CustomQueue));

    Init_Queue(free_queue_, FREE_QUEUE);
    Init_Queue(work_queue_,WORK_QUEUE);

    for (int i=0; i < CustomQueueSize; i++){
        NodeQueue *node = (NodeQueue*)malloc(sizeof(struct NodeQueue));
        node->data = NULL;
        node->size = 0;
        node->index = 0;
        //入队
        this->Enque_Queue(free_queue_,node);
    }

    pthread_mutex_init(&free_queue_mutex,NULL);
    pthread_mutex_init(&work_queue_mutex,NULL);

    printf("Init finish!");
```

```
}

void ProcessQueue::Init_Queue(CustomQueue *queue,QueueType type)
{
    if(queue != NULL){
        queue->type = type;
        queue->size = 0;
        queue->front = 0;
        queue->rear = 0;
    }
}

void  ProcessQueue::Enque_Queue(CustomQueue *queue, NodeQueue *node)
{
    if (queue == NULL){
        printf("current queue is null\n");
        return;
    }

    if (node == NULL){
        printf("current node is null\n");
        return;
    }

    if (FREE_QUEUE == queue->type){
        pthread_mutex_lock(&free_queue_mutex);

        if(queue->front == NULL){
            queue->front = node;
            queue->rear = node;
        }
        else{
            //head in ,head out
            node->next = queue->front;
            queue->front = node;

            //tail in, had out
            /*
                @TODO
            */
        }
        //队列大小
        queue->size += 1;
        printf("free queue size=%d\n",queue->size);
        pthread_mutex_lock(&free_queue_mutex);
    }

    if (WORK_QUEUE == queue->type){
        pthread_mutex_lock(&work_queue_mutex);

        static long nodeIndex = 0;
        node->index = (++nodeIndex);
        if(queue->front == NULL){
```

```
            queue->front = node;
            queue->rear = node;
        }
        else{
            //head in ,head out
            /*
                @TOD
            */
            //tail in, had out
            queue->rear->next = node;
            queue->rear = node;
        }
        queue->size += 1;
        printf("work queue size=%d\n",queue->size);
        pthread_mutex_lock(&work_queue_mutex);
    }
}


 NodeQueue* ProcessQueue::Deque_Queue(CustomQueue  *queue)
 {
    if(queue == NULL){
        printf("current queue is null\n");
        return NULL;
    }

    const char *type = queue->type == WORK_QUEUE?"work queue":"free queue";
    pthread_mutex_t *queue_mutex = ((queue->type == WORK_QUEUE) ? &work_queue_mutex:
&free_queue_mutex);

    NodeQueue *element = NULL;

    pthread_mutex_lock(queue_mutex);
    //队头开始出来
    element = queue->front;
    if(element == NULL){
        pthread_mutex_unlock(queue_mutex);
        printf("The node is NULL\n");
        return NULL;
    }

    queue->front = queue->front->next;
    queue->size -= 1;
    pthread_mutex_unlock(queue_mutex);

    printf("type=%d, size=%d \n", queue->type, queue->size);

    return element;
 }

void ProcessQueue::Clear_Custom_Queue(CustomQueue *queue)
{
    while(queue->size){
```

```
        NodeQueue *node = this->Deque_Queue(queue);
        this->Free_Node(node);
    }
    printf("clear process queue\n");
}
void ProcessQueue::Free_Node(NodeQueue *node)
{
    if(node !=NULL){
        free(node->data);
        free(node);
    }
}
void ProcessQueue::Reset_Free_Queue(CustomQueue *work_queue,CustomQueue *free_queue)
{
    if(work_queue == NULL){
        printf("the workqueue is null\n");
        return;
    }
    if(free_queue == NULL){
        printf("the free queue is null\n");
        return;
    }

    int work_queue_size = work_queue->size;
    if(work_queue_size > 0){
        for (int i=0; i < work_queue_size; i++){
            NodeQueue *node = Deque_Queue(work_queue);
            free(node->data);
            node->data = NULL;
            Enque_Queue(free_queue,node);
        }
    }

    printf("reset free queue: work queue size is %d, free queue size is %d\n",
        work_queue->size, free_queue->size);
}


typedef struct TestBuffer
{

}TESTBUFFAE;


class Test{
    public:
        Test(){

            process_queue_ = new ProcessQueue();
            pthread_create(&process_queue_thread_, NULL,startTask,this);
        }
        ~Test(){
            if(process_queue_ !=NULL){
```

```
                delete process_queue_;
                process_queue_ = NULL;
            }
        }

        //知识点：起线程用static，形参可以转成this
        static void* startTask( void* arg ){
            ((Test*)arg)->handleTask();
            return NULL;
        }

        void handleTask(){
            while (true){
                //取出对头元素，进行处理
                NodeQueue *node = process_queue_->Deque_Queue(process_queue_->work_queue_);
                if(node ==NULL){
                    printf("data node is null");
                    //sleep(1);
                    continue;
                }
                TestBuffer* test_buffer = (TestBuffer*)node->data;
                //如果 index是连续的，则test_buffer放入的也是连续的
                printf("test index: %ld", node->index);
                /*

                用完记得释放内存，并将节点重新放入空闲队列
                */
                free(test_buffer);
                node->data = NULL;
                process_queue_->Enque_Queue(process_queue_->free_queue_, node);
            }
        }


    private:

        ProcessQueue *process_queue_;
        pthread_t process_queue_thread_;

};




int main(){
    Test test_;
}
```

6.2.2.缓冲池队列应用

接着，我们实现一个缓冲池队列，这个缓冲池队列是一个循环队列，通过信号量的方式，监控数据的写入和读取，下面我们来学习一下它的具体实现。

```
#include <iostream>
#include <functional>
#include <condition_variable>
#include <string>
#include <thread>
#include <mutex>

/*
@ 循环队列，缓冲池中保存缓存区模板

1,信号量
2,缓存区

公众号：码出名企路

*/

/*
@ 信号量

*/
class Semaphore{
    public:
        Semaphore(int count =0){
            if (count <0){
                printf("used count < 0 \n");
            }

            this->count_ = count;
        }

        //释放资源
        void signal(){
            std::unique_lock<std::mutex> lock(this->mutex_);
            ++this->count_;
            //知识点：notify_one和wait的对应关系
            this->cv_.notify_one();
        }

        //申请资源
        void wait(){
            std::unique_lock<std::mutex> lock(this->mutex_);
            this->cv_.wait(lock, [=]{
                return count_ > 0;
```

```
            });
            --this->count_;
        }

    private:

        std::mutex mutex_;
        std::condition_variable cv_;
        int count_;
};

/*
@ 缓冲区
*/
typedef struct Data{

}DATA;

class Buffer{
    public:
        Buffer(int buffer_size);
        ~Buffer();

        void setEffectSize(int size);

        //将数据写入缓冲区
        void write(std::function<int(Data*,int)> const& writeBuffer);

        //从缓冲区读取数据
        void read(std::function<int(Data*,int,int)> const& readBuffer);

    private:

        //缓冲区数据
        Data *data_;
        int max_size_;
        int effect_size_;

        //信号量
        Semaphore* sem_;
};

Buffer::Buffer(int buffer_size)
{
    this->max_size_ = buffer_size;
    this->effect_size_ = 0;
    this->data_ = new Data[buffer_size];

    this->sem_ = new Semaphore(1);
}

Buffer::~Buffer()
{
```

```
}

void Buffer::setEffectSize(int size)
{
    this->effect_size_ = size;
}

//将数据写入缓冲区
void Buffer::write(std::function<int(Data*,int)> const& writeBuffer)
{
    this->sem_->wait();
    this->effect_size_ = writeBuffer(this->data_,this->max_size_);
    this->sem_->signal();
}

//从缓冲区读取数据
void Buffer::read(std::function<int(Data*,int,int)> const& readBuffer)
{
    this->sem_->wait();
    readBuffer(this->data_,this->effect_size_,this->max_size_);
    this->sem_->signal();
}


/*
@ 缓冲池
缓冲区的集合，通过缓冲池来分配缓冲区

要保证按数据写入的顺序读出，应该把缓冲池设计为队列，保证读取时缓冲区总是最先写入的，
同时写入时获得的缓冲区是最末尾的缓冲区。同时为了保证缓冲区循环利用，将缓冲池设计为循环队列。

对于循环队列，当头指针和尾指针相等时，有两种情况。一种是队列为空(未分配缓冲区)，另一种是队列已满(所有缓冲区都被分配)。
解决方法一般有两种，第一种是牺牲一个存储空间，当尾指针指向的下一位为头指针时，即队列为满。
另一种是增加标志位来判断当头尾指针相同时，当前队列的状态。
*/
class BufferPool{
    public:

        BufferPool(int count =10, int buffersize = 10);

        ~BufferPool();
        //获取一个缓冲区
        Buffer* getBuffer();

        //取得头缓冲区并弹出
        Buffer* popBuffer();

        bool empty();

        //缓冲池是否已满
        bool full();
```

```
    private:

        //头尾指针
        int head,tail;
        //缓冲池
        Buffer** buffers;
        //缓冲区总个数和使用个数
        int total, lenth;
};


BufferPool::BufferPool(int count, int bufferSize) {
    this->head = 0;
    this->tail = 0;
    this->lenth = 0;
    this->total = count;

    this->buffers = new Buffer*[count];
    for (int i = 0; i < count; i ++) {
        this->buffers[i] = new Buffer(bufferSize);
    }
}

/**
* 获取一个缓冲区 当缓冲池已满时，覆盖旧数据
*/
Buffer* BufferPool::getBuffer() {

    Buffer* buffer = this->buffers[this->tail];
    // tail指针指向下一个缓冲区，如果当前缓冲池已满，头指针下移
    this->tail = (this->tail + 1) % this->total;
    this->lenth++;
    if (this->lenth > this->total) {
        this->head = (this->head + 1) % this->total;
        this->lenth = this->total;
    }
    return buffer;
}

/**
* 获取头缓冲区并弹出
*/
Buffer* BufferPool::popBuffer() {
    if (this->lenth == 0) {
        throw "缓冲池为空";
    }
    Buffer* buffer = this->buffers[this->head];
    this->head = (this->head + 1) % this->total;
    this->lenth--;
    return buffer;
}
```

```
BufferPool::~BufferPool() {
    for (int i = 0; i < this->total; i ++) {
        delete this->buffers[i];
        this->buffers[i] = NULL;
    }
    delete this->buffers;
    this->buffers = NULL;
}

bool BufferPool::empty() {
    return this->lenth == 0;
}

bool BufferPool::full() {
    return this->lenth == this->total;
}


/*
测试用例:

*/
class Test{

    public:

        void test(){
            //缓冲池保存数据
            BufferPool* bufferPool = new BufferPool();

            //获取一个缓冲区，将数据写入
            int read_total_count = 0;
            bufferPool->getBuffer()->write([&](Data* data,int maxSize){

                /*
                    具体采集数据得程序
                    read_total_count 在此更新
                */
                return read_total_count;
            });

            //获取缓冲区数据

            while(!bufferPool->empty()){
                bufferPool->popBuffer()->read([&](Data* data,int eff,int maxSize){
                    /*

                     read and use it
                    */
                    for(int index = 0; index < maxSize; index++){
                        printf("index: %d, data: %s \n",index,data[index]);
                    }
```

```
                    return 0;
                });
            }
        }
};


int main()
{
    Test test_;
    test_.test();
}
```

6.3.知识点拆分

随着我们学习的增加，遇到的陌生知识点越来越少。本节缓冲队列用的都是我们比较熟悉的数据结构和知识点，关键在于，当我们遇到一个大型问题的时候，思考如何设计架构是关键。当架构思考结束之后，就应用想到需要用到哪些知识进行实现。本节主要涉及的知识点如下：

1，enum class和enum有什么区别

2，线程池为什么必须是static函数

3，信号量notify_one和wait的对应关系

下面我们具体看一下以上知识点在工程框架中的应用以及具体的知识细节。

6.3.1.enum class和enum有什么区别

第一部分：enum就是列举枚举的意思，在本节框架中的应用如下：

```
typedef enum{
    WORK_QUEUE = 0,
    FREE_QUEUE
}QueueType;
```

第二部分：有时候，我们或许可以看到enum calss的写法，这样的写法和enum有什么不一样的地方或者好处呢，具体解释如下：

在 C++11 中，引入了一种新的枚举类型：enum class，它和原来的枚举类型 enum 有以下区别：

1. 范围：enum 类型的常量值在同一范围内，即使它们有不同的名称也能互相比较，而 enum class 常量值的作用域为其所在的枚举类，因此它们不能与其它枚举类常量值进行比较或者赋值操作，必须显式转换为其它整数类型。
2. 名称空间：enum 类型的枚举值被强制转换为 int 类型，并且它们存在于全局命名空间中，可能导致名称重复的问题。而 enum class 解决了这个问题，每个标识符都属于其所在的枚举类的作用域，不会造成命名冲突。
3. 类型安全：enum class 可以避免未初始化类型的变量和因为枚举值循环或溢出引起的不确定行为，使得编译时错误更加容易发现。

下面我给出一些 C++ 例子来解释 enum class 和 enum 之间的区别。

1. 名称空间的区别

枚举类型的名称可以在全局作用域中使用，因此可能导致命名冲突。

```
enum Color {
    RED,
    GREEN,
    BLUE
};

enum Fruit {
    APPLE,
    ORANGE,
    BANANA
};

int main() {
    int c = RED;          // OK
    int a = APPLE;        // OK
    return 0;
}
```

而 enum class 可以将枚举类型的名称限制在命名空间内。

```
enum class Color {
    RED,
    GREEN,
    BLUE
};

enum class Fruit {
    APPLE,
    ORANGE,
    BANANA
};

int main() {
    int c = Color::RED;     // 编译错误：必须显式转换
    int a = Fruit::APPLE;   // 编译错误：必须显式转换
    return 0;
}
```

2. 类型安全性的区别

枚举类型的值可以赋给一个 int 类型的变量，也可以相互之间进行比较，但枚举类型的值之间没有明确的类型差异。

```
enum Color {
    RED,
    GREEN,
    BLUE
};

void printColor(Color c) {
    std::cout << "The color is " << c << std::endl;
}

int main() {
    Color c = RED;
    int i = c;              // OK，转换为了 int 类型
    std::cout << (c == i) << std::endl;  // 输出 1，可以相互之间进行比较

    printColor(1);          // 也可以传入 int 类型的参数

    return 0;
}
```

enum class 在比较和赋值时需要使用静态转换，同时这两种类型之间不能进行几何操作。

```
enum class Color {
    RED,
    GREEN,
    BLUE
};

void printColor(Color c) {
    std::cout << "The color is " << static_cast<int>(c) << std::endl;
}

int main() {
    Color c = Color::RED;
    int i = static_cast<int>(c);    // 必须使用静态转换
    std::cout << (c == i) << std::endl;  // 编译错误：不能相互之间进行比较

    printColor(static_cast<Color>(1));  // 必须使用静态转换

    return 0;
}
```

3. 变量初始化和默认值的区别

枚举类型的变量可以直接使用默认值，而 enum class 必须给每个变量都明确的赋值。

```
enum Color {
    RED = 0,
    GREEN = 2,
    BLUE = 4
};

int main() {
    Color c = BLUE;    // 可以直接使用默认值
    return 0;
}
```

```
enum class Color {
    RED = 0,
    GREEN = 2,
    BLUE = 4
};

int main() {
    Color c = Color::RED;    // 必须赋明确的值
    return 0;
}
```

总之，enum class 和 enum 在 C++ 中都是枚举类型，但它们之间有很大的区别，主要体现在名称空间的冲突、类型的安全性、变量初始化和默认值等方面。

6.3.2.线程必须是static函数吗

第一部分：线程static函数，形参是一个void*的指针，可以转换成this指针，调用自身的成员函数，本节应用如下：

```
    static void* startTask( void* arg ){
            ((Test*)arg)->handleTask();
            return NULL;
        }
```

第二部分：线程函数必须是static函数吗？还有其他方式实现没？

在 C++ 中，线程函数不必是 static 函数，其实任何函数都可以被用作线程函数，只需要满足以下两个条件：

1. 该函数是可调用的(callable)，可以是普通函数、静态函数、成员函数或者 lambda 表达式等。
2. 该函数的参数类型、个数、返回值类型与 std::thread 构造函数中指定的参数类型、个数、返回值类型一致。

下面分别给出实现一个普通函数、静态函数和成员函数作为线程函数的例子：

1. 实现一个普通函数作为线程函数

```
void threadFunc(int arg1, double arg2) {
    // do something
}

int main() {
    std::thread t(threadFunc, 42, 3.14);
    // do something in main thread
    t.join();
    return 0;
}
```

2. 实现一个静态函数作为线程函数

```
class Foo {
public:
    static void threadFunc(int arg1, double arg2) {
        // do something
    }

    void startThread() {
        std::thread t(&Foo::threadFunc, this, 42, 3.14);
        // do something in main thread
        t.join();
    }
};

int main() {
    Foo f;
    f.startThread();
    return 0;
}
```

3. 实现一个成员函数作为线程函数

```
class Bar {
public:
    void threadFunc(int arg1, double arg2) {
        // do something
    }

    void startThread() {
        std::thread t(&Bar::threadFunc, this, 42, 3.14);
        // do something in main thread
        t.join();
    }
};

int main() {
    Bar b;
    b.startThread();
    return 0;
}
```

如上所示，我们可以使用普通函数、静态函数、成员函数或者 lambda 表达式等作为线程函数，只需要满足可调用的要求，即参数类型、个数、返回值类型和 std::thread 构造函数中指定的参数一致即可。

6.3.3.信号量notify和wait的对应关系

第一部分：互斥锁和信号量是多线程编程中的必备知识，其中信号量一般也要和锁结合使用，本节使用案例如下：

```
        //释放资源
        void signal(){
            std::unique_lock<std::mutex> lock(this->mutex_);
            ++this->count_;
            //知识点：notify_one和wait的对应关系
            this->cv_.notify_one();
        }

        //申请资源
        void wait(){
            std::unique_lock<std::mutex> lock(this->mutex_);
            this->cv_.wait(lock, [=]{
                return count_ > 0;
            });
            --this->count_;
        }
```

第二部分：notify_one,notify_all,wait,wait_for这些都是在什么时候应用，我们具体学习一下：

`notify_one`、`notify_all`、`wait` 和 `wait_for` 属于 C++ 中的线程同步机制。它们主要用于实现线程间的通信和同步，常常用于单线程等待多线程通知，或多个线程等待某个事件的发生等场景。

具体来说，可以在互斥量（`std::mutex`）和条件变量（`std::condition_variable`）上使用这些函数，其中互斥量用于在多个线程之间提供互斥访问同一个共享变量，条件变量用于在同步线程之间的信号传递。在使用条件变量时，wait 用于等待其他线程的通知，notify_one 和 notify_all 用于发出通知。

1. `std::condition_variable::notify_one()`：唤醒一个等待此条件变量的线程，如果没有线程等待，则该方法不执行任何操作。
2. `std::condition_variable::notify_all()`：唤醒所有等待此条件变量的线程。
3. `std::condition_variable::wait(lock)`：等待一个信号或通知，在等待过程中会释放锁，并进入休眠状态。当其它线程发出通知后，该方法会尝试重新获取锁，并继续访问共享变量。其中，`lock` 为一个 std::unique_lockstd::mutex 类型的互斥锁。
4. `std::condition_variable::wait_for(lock, timeout)`：等待一个信号或通知，在等待过程中会释放锁，并进入休眠状态，但是如果在指定的时间内没有接收到信号或通知，该方法会返回，继续访问共享变量。其中，`timeout` 为等待的时间，超时时间之后，该方法不会等待任何其他的通知。

下面是一个简单的例子，演示了使用互斥量和条件变量实现两个线程间的同步：

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void worker_thread() {
    std::unique_lock<std::mutex> lck(mtx);  // 获得互斥锁
    while (!ready) {  // 如果条件未满足
        cv.wait(lck);  // 睡眠等待信号唤醒
    }
    // 条件满足，执行操作
    std::cout << "Worker thread is processing data..." << std::endl;
}

int main() {
    std::thread worker(worker_thread);

    // 假设在这里进行一些计算任务
    std::this_thread::sleep_for(std::chrono::milliseconds(10));

    {
        std::lock_guard<std::mutex> lck(mtx);  // 获得互斥锁
        ready = true;  // 设定条件已满足
    }
    cv.notify_one();  // 发送信号

    worker.join();
    std::cout << "Done" << std::endl;

    return 0;
}
```

该例子中，主线程在向 worker 线程发送信号之前等待 10ms，模拟一些计算任务的执行，之后发送通知唤醒 worker 线程执行任务，worker 线程在条件判断不满足时等待信号，条件满足后继续执行任务。

6.4.本节大作业

本节我们学习的缓冲队列应用较为广泛，当然这里说一个小技巧，利益stl中的deque或者queue也可以实现上述功能，但是既然是学习框架，就带大家看一下具体原理。本节有两个作业：

1，用stl中的deqeue去代替本节的具体实现，选择其中一个架构进行改写，并编译测试通过

2，这个作业是重点，现在需要你开发一个相机驱动程序，需要满足视频流和处理视频流两大块逻辑不能相互阻塞，并且保证处理视频流的工作是定频的。请选择本节框架进行实现。

本节参考

https://segmentfault.com/a/1190000040005802?sort=votes

7.大型监控系统

大型监控系统一般包括如下功能，数据采集和处理，从多个系统中采集数据，包括CPU使用率，内存使用率等，然后对采集到的数据进行处理转换成监控数据。一般，我们会自己定义出错的报警信息，这种信息在整个工程框架中是可以被复用的，工程里进行调用既可以完成实时监控。

7.1.工程应用

本节我们重点实现一个大型监控系统，这个监控系统我们可以应用在各种场合：

> 1，自己负责的模块设计的的各种报错信息，比如初始化失败，正常处理时失败等
>
> 2，整条链路上的报警，所有模块都可以进行插桩，每个模块用户自定义插桩位置，完成系统监控的功能。

7.2.框架实现

本节我们重点实现一个大型监控系统，这个监控系统我们用来进行搜集错误信息，统计链路出错结果等。它可以被应用在各种场合，读者可以拿来放在自己的工程里，实践出真知。

```
#include <string>
#include <iostream>
#include <memory>

/*
@ 一个埋点系统，所有节点可以调用埋点接口，统一把结果传给埋点系统

a，判断当前节点的运行状态
b.输出各个节点的错误或者正确信息
```

```
c 帮助你快速定位整个工程架构的运行信息，错误日志，定位问题等等
d, .......

1，定义错误日志信息代表的含义，enum / enum class
2，埋点系统类设计    class / single  / 重载
3，宏定义接口设计    define  __##
4，测试类用例        class /  埋点
5，C++ 14的unique_ptr    make_uniqued
*/

/*
@ 1,  枚举表示错误日志信息
*/
//知识点：enum calss
enum InfoType
{
    DEFAULT = 0,
    INIT_ERROR = 1,
    INIT_SUCESS = 2,
    CALL_ERROR = 3,
    CALL_SUCESS = 4
};

inline std::string getInfoType(InfoType type)
{
    switch (type)
    {
    case INIT_ERROR:
        return "INIT_ERROR";
        break;

    default:
        return " ";
        break;
    }
}

class ReporterInfo{

    public:
       //知识点：单例模式
        static ReporterInfo &getInstance()
        {
            static ReporterInfo reporter_info;
            return reporter_info;
        }


        ~ReporterInfo();

        bool init(const std::string &conf = "");

        bool callDump(InfoType type,const std::string &re_info = " ")
```

```
        {
            return callDump_Main(getInfoType(type),re_info);
        }

        bool callDump_Main(const std::string &node, InfoType type)
        {
            return callDump_Main(node, getInfoType(type));
        }

        bool callDump_Main(const std::string &node, const std::string &type);

    private:
        ReporterInfo(){}
        ReporterInfo(const ReporterInfo &) = delete;
        ReporterInfo &operator=(const ReporterInfo &) = delete;
};

//知识点：static全局变量和成员变量
static bool s_is_init = false;

ReporterInfo::~ReporterInfo()
{

}

bool ReporterInfo::init(const std::string &conf)
{
    std::cout<<"INFO_REPORT_INIT conf: "<<conf<<std::endl;
    /*


    */

    s_is_init = true;
    return true;
}

bool ReporterInfo::callDump_Main(const std::string &node, const std::string &type)
{
    std::cout<<"node: "<<node<<"type: "<<type<<std::endl;

    if (!s_is_init)
    {
        return false;
    }

    /*
    自己实现需求

    */
    return true;
}
```

```
/*
3，宏定义触发事件发生
*/
#define INFO_REPORT_INIT() ReporterInfo::getInstance().init()
#define INFO_REPORT_INIT_CONF(name) ReporterInfo::getInstance().init(name)

//知识点：宏定义多参数...
#define INFO_REPORT_CALL(type,...) ReporterInfo::getInstance().callDump(type,
##__VA_ARGS__)
#define INFO_REPORT_CALL_MAIN(name,type,...)
ReporterInfo::getInstance().callDump_Main(name,type,##__VA_ARGS__)


/*
@测试用例
*/
class Test_Mode{
    public:
        Test_Mode(){}
        ~Test_Mode(){}
        bool init(const std::string &conf_name);

        bool process();

    private:

};

bool Test_Mode::init(const std::string &conf_name)
{
    INFO_REPORT_INIT_CONF(conf_name);
    if (1)
    {
        /*
        自己实现
        */

        INFO_REPORT_CALL(InfoType::CALL_SUCESS);
    }

    return true;
}

bool Test_Mode::process()
{
    if (1)
    {
        /*
        自己实现
        */

        INFO_REPORT_CALL_MAIN("test",InfoType::CALL_SUCESS);
    }
```

```
    return true;
}

/*

C++14 make_unique

用C++11 不支持 用C++11实现

*/
#if __cplusplus >= 201402L
using std::make_unique;
#else
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args){
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
#endif

int main()
{
   //std::shared_ptr<Test_Mode> test_mode = std::make_shared<Test_Mode>();

   std::unique_ptr<Test_Mode> test_mode = make_unique<Test_Mode>();

   test_mode->init("/data/init_txt.txt");
   test_mode->process();

}
```

7.3.知识点拆分

大型监控系统利用的知识点比较常见，大多数是我们之前学习过的，这里就不多做说明了。我在这里主要说明之前项目中没有讲到的知识点，具体涉及如下，后面章节我们遇到新的知识点会越来越少，这也正是我们坚持下来的结果。

1，static全局变量和成员变量

2，宏定义define的多参数...传参

3，**VA_ARGS**的应用

4，C++11实现的make_unique

下面我们就具体讲解一下以上知识点在本节大型监控系统中的应用，以及我们必须掌握的知识点。

7.3.1.static全局变量和成员变量

第一部分：本节我们定义了一个sttaic的全局变量，如下所示：

```
static bool s_is_init = false;
```

第二部分：为什么要有static全局变量，与普通变量有什么区别，下面我们具体看一下：

静态全局变量和静态成员变量的区别和普通全局变量或成员变量的区别是类似的。

1. 静态全局变量和静态成员变量在内存中只有一份实例，而普通全局变量或成员变量在每个引用它们的文件中都有一份实例。因此，静态变量的生命周期和作用域都是全局的，并且在多个文件中的访问是通过extern关键字进行声明（也可以被封装在类内，实现静态成员变量）。
2. 静态全局变量和静态成员变量默认初始化为0，而普通全局变量不会隐式地初始化为0（它们可能包含任何值）。
3. 静态成员变量可以直接通过类名访问，而不需要先创建类的实例。而普通成员变量的访问必须通过类的实例进行。
4. 静态成员变量可以被继承和覆盖，而普通成员变量不能被继承和覆盖。

在使用静态变量时，需要注意以下几点：

1. 静态变量的初始化顺序是不确定的，因此不应该依赖于其他静态变量的初始化状态。
2. 静态变量的生命周期同程序运行期间，因此长时间持有静态变量可能会影响性能和内存使用。
3. 静态变量可能存在线程安全性问题，因此在多线程程序中使用时需要进行同步。

下面是几个C++例子，演示了静态全局变量和普通全局变量、静态成员变量和普通成员变量的区别：

1. 静态全局变量和普通全局变量

静态全局变量和普通全局变量的区别在于静态全局变量只能在当前文件中访问，而普通全局变量可以在其他文件中访问。

```
// file1.cpp
#include <iostream>
static int static_global_var = 1;
int normal_global_var = 2;

void function1() {
  std::cout << "static_global_var = " << static_global_var << std::endl;
  std::cout << "normal_global_var = " << normal_global_var << std::endl;
}

// file2.cpp
#include <iostream>
extern int normal_global_var;
void function2() {
  std::cout << "normal_global_var = " << normal_global_var << std::endl;
  // 编译出错，static_global_var 不能在此处访问
  std::cout << "static_global_var = " << static_global_var << std::endl;
}

// main.cpp
```

```
void function1();
void function2();

int main() {
  function1();
  function2();
  return 0;
}
```

在上面的示例中，静态全局变量static_global_var只能在当前文件中访问，因此在function2中无法编译通过。而普通全局变量normal_global_var可以在其他文件中访问，因此在function2中可以正常访问。

2. 静态成员变量和普通成员变量

静态成员变量和普通成员变量的区别在于静态成员变量是类的所有实例共享的，而普通成员变量是每个实例独立的。

```
#include <iostream>

class MyClass {
public:
  int normal_member_var = 1;
  static int static_member_var;
};

int MyClass::static_member_var = 2;

int main() {
  // 创建两个 MyClass 实例
  MyClass obj1, obj2;

  // 修改普通成员变量
  obj1.normal_member_var = 3;

  // 修改静态成员变量
  obj1.static_member_var = 4;

  // 打印变量的值
  std::cout << "obj1.normal_member_var = " << obj1.normal_member_var << std::endl;
  std::cout << "obj2.normal_member_var = " << obj2.normal_member_var << std::endl;
  std::cout << "obj1.static_member_var = " << obj1.static_member_var << std::endl;
  std::cout << "obj2.static_member_var = " << obj2.static_member_var << std::endl;

  return 0;
}
```

上面的示例中，定义了一个类MyClass，类中包括一个普通成员变量normal_member_var和一个静态成员变量static_member_var。在main函数中，创建了两个MyClass实例：obj1和obj2，并修改了它们的normal_member_var和static_member_var成员变量的值。可以看到，虽然obj1的成员变量值被修改了，但obj2的静态成员变量static_member_var的值也被改变，这是因为它们共享同一份静态成员变量。而普通成员变量normal_member_var的值是独立的，obj1和obj2的值是不相同的。

需要注意的是，静态成员变量需要在类外部进行定义和初始化。在上面的示例中，为了初始化静态成员变量，需要加上int MyClass::static_member_var = 2;这句定义。

7.3.2.宏定义define的多参数...传参

第一部分：define应用广泛，一般作为对外接口，用宏定义封装内部实现，有时候会用到...形参，如本节所示：

```
#define INFO_REPORT_CALL_MAIN(name,type,...)
ReporterInfo::getInstance().callDump_Main(name,type,##__VA_ARGS__)
```

第二部分：define用的...发挥怎么样的作用，下面我们具体学习一下。

可以使用可变参数宏（variadic macro）来实现宏定义中的多参数传递，具体方法是在宏定义中使用省略号(...)。

以下是一个可变参数宏的示例，可以将多个参数相加：

```
#include <iostream>
#define ADD(...) add(__VA_ARGS__)
int add(int a, ...) {
  int sum = a;
  va_list args;
  va_start(args, a);
  while (int arg = va_arg(args, int)) {
    sum += arg;
  }
  va_end(args);
  return sum;
}

int main() {
  std::cout << ADD(1, 2, 3, 4, 5) << std::endl;
  std::cout << ADD(1, 2, 3) << std::endl;
  std::cout << ADD(1, 2) << std::endl;
  std::cout << ADD(1) << std::endl;
  return 0;
}
```

在上面的示例中，DEFINE(...)是可变参数宏的定义，其中add(...)是一个函数，使用了可变参数列表，并返回所有参数的总和。在main函数中，将多个参数传递给ADD(...)宏，宏会将所有参数转发给add(...)函数，并输出它们的总和。

需要注意的是，在使用可变参数宏时，不要使用逗号操作符、括号或分号将参数组合在一起，因为这可能会与可变参数列表的语法相冲突。

以下是另一个可变参数宏的示例，用于打印输出信息：

```
#include <iostream>
#include <string>
#include <sstream>

#define LOG(...) log(__FILE__, __LINE__, __VA_ARGS__)
```

```
void log(const char* file, int line, const char* format, ...) {
  va_list args;
  va_start(args, format);

  std::ostringstream ss;
  ss << file << ":" << line << " - " << format << std::endl;

  std::vprintf(ss.str().c_str(), args);
  va_end(args);
}

int main() {
  int x = 42;
  std::string name = "Alice";
  LOG("x = %d, name = %s", x, name.c_str());
  LOG("Hello, World!");
  return 0;
}
```

在上面的示例中，DEFINE(...)是可变参数宏的定义，其中log(...)是一个函数，它接受一个文件名、行号以及可变数量的参数，把它们格式化成一个字符串，最终通过标准输出打印出来。

在main函数中，通过LOG(...)宏打印了两个消息。第一个LOG(...)使用了格式化字符串来输出局部变量x和name的值，第二个LOG(...)只是简单地输出了"Hello, World!"。

使用宏定义的好处在于，它能够使代码更加简洁易懂，并且可以减少代码中的重复。在上面的示例中，由于LOG(...)宏的使用，可以在输出的消息中包含文件名和行号，这是一项非常有用的调试功能。

需要注意的是，使用可变参数宏时，要注意参数列表的数量和类型，以及它们的匹配关系。在可变参数宏的定义和使用中，需要使用"..."指明参数列表的可变部分。`__VA_ARGS__`用于引用这些可变参数列表。另外，可变参数列表应该以一个特殊值（如0或nullptr）结尾，以便在函数中结束可变参数的循环。例如，在上面的log函数中，`va_arg(args, int)`就使用了这种结尾方式，当遇到0时中止参数列表的处理。

7.3.3.VA_ARGS的应用

第一部分：**VA_ARGS**这种关键字很少见，本节带领大家学习一下：

```
#define INFO_REPORT_CALL_MAIN(name,type,...)
ReporterInfo::getInstance().callDump_Main(name,type,##__VA_ARGS__)
```

第二部分：什么时候需要用到这个关键字呢？

VA_ARGS 是 C++ 中的一个特殊关键字，用于可变参数的宏定义中。

在 C++ 的宏定义中，如果希望宏可以接受可变数量的参数，可以使用省略号（...）来定义宏参数。那么，在宏的正文中，如何引用可变数量的参数呢？这时就需要 **VA_ARGS** 了。

VA_ARGS 实际上代表了所有可变参数的序列，在宏的定义中可以将其引用。下面是一个简单的示例：

```
#include <iostream>

#define PRINT_VALUES(...) printValues(__VA_ARGS__)

void printValues(const char* title, ...) {
  std::cout << title << ": ";
  va_list arglist;
  va_start(arglist, title);
  for (;;) {
    const char* value = va_arg(arglist, const char*);
    if (!value) break;
    std::cout << value << ", ";
  }
  va_end(arglist);
  std::cout << std::endl;
}

int main() {
  PRINT_VALUES("Integers", "one", "two", "three", nullptr);
  PRINT_VALUES("Strings", "a", "b", "c", "d", "e", nullptr);
  return 0;
}
```

在上面的示例中，使用 PRINT_VALUES 宏定义来代替 printValues 函数调用，并传入可变数量的参数来自动调用函数。

在 PRINT_VALUES 的宏定义中，**VA_ARGS** 就代表了可变数量的参数，在宏扩展时，将可变数量的参数按照传入参数列表的顺序展开。

需要注意的是，在使用 **VA_ARGS** 时，要确保以nullptr、0或其他可靠的值来结尾。这样可以使函数了解何时到达参数列表的结尾，以便正确处理参数。

除了 **VA_ARGS** 之外，C++ 标准库还提供了其他处理可变数量参数的方法，如 std::initializer_list、std::tuple 等等，可以根据需要选择使用。

7.3.4.C++11实现的make_unique

第一部分：熟悉新特性的同学应该知道C++14中才允许使用make_unique，然后我们大多数工程都是在C++11里编译的，如何在C++11里自己实现一个make_unique呢？本节目给出了答案：

C++11 中引入了 make_unique 函数模板，用于实现返回一个 std::unique_ptr 的工厂函数，用于动态分配对象。

make_unique 函数模板的定义非常简单，不过需要注意在传递参数时，std::make_unique 不支持 std::initializer_list。

以下是一个示例：

```
#include <iostream>
#include <memory>

std::unique_ptr<int> make_unique(int value) {
    return std::unique_ptr<int>(new int(value));
}

int main() {
    std::unique_ptr<int> ptr = make_unique(42);
    std::cout << *ptr << std::endl;
    return 0;
}
```

在上面的示例中，make_unique 函数模板以参数 value 为输入，返回一个指向 int 对象的 std::unique_ptr。

值得注意的是，在 C++11 标准中，std::make_unique 函数可供使用，可以用作 std::unique_ptr 技术中对象体的构造。

同样，以下是 std::make_unique 的一个示例：

```
#include <iostream>
#include <memory>

int main() {
    auto ptr = std::make_unique<int>(42);
    std::cout << *ptr << std::endl;
    return 0;
}
```

在上述示例中，我们使用 std::make_unique 来初始化一个指向 int 对象的 unique_ptr，而不是显式调用 new 函数。这是一个更方便、简单和安全的方法，避免使用 new 和 delete 操作符引入的错误，尤其是在内存泄漏和悬垂指针方面。

在使用 std::make_unique 时，不需要显式指定对象的类型，类型是通过 std::make_unique 自动推断得到的。std::make_unique 函数模板还可支持构造对象、数组和使用自定义释放器。

需要注意的是，在许多实现中，std::make_unique 函数是以 C++14 的模板参数推导功能实现的，而不是 C++11 的环境。在使用时，需要确认当前环境支持 C++14。

7.4.本节大作业

本节完成了一个大型监控系统的设计与开发，这可以说是一个模板，自己可以定义任意的参数和接口去扩展这个监控系统。下面，请你在自己前几节作用完成的基础上，或者你目前正在用的框架，将这个监控系统嵌入进去，满足可以实时报警的功能，具体报警的信息和错误日志，由你自己来定。同时，扩展一下，将这些日志记录在本地，供开发调试查看。