C++超全基础知识-2024

一、虚函数

https://zhuanlan.zhihu.com/p/546108835

秋招全面的八股总结：

1.多态的实现：

主要分为静态多态和动态多态，静态多态主要是重载，在编译的时候就已经确定；动态多态是用虚函数机制 实现的，在运行期间动态绑定

举个例子：

（1）一个父类类型的指针指向一个子类对象时候

（2）使用父类的指针去调用子类中重写了的父类中的虚函数的时候

（3）会调用子类重写过后的函数

动态绑定:符合以下三个条件：通过指针来调用函数,指针 upcast 向上转型,调用的是虚函数

```
1 B bObject;
2 A *p = & bObject;
3 p->vfunc1();
```

多态的好处：

1. 简化代码：多态允许我们通过一个通用的接口处理不同类型的对象，从而减少代码重复和提高代码的可 读性和可维护性。
2. 扩展性：由于多态允许基类指针或引用可以指向派生类对象，我们可以方便地向程序添加新的派生类，

而无需修改已有代码。这有助于实现开放/封闭原则，即对扩展开放，对修改封闭。

3. 解耦：多态有助于实现类之间的解耦，因为它使得类之间的交互是通过通用接口进行的。这可以提高代 码的模块化，使得各个类可以独立地开发和测试。

2.虚函数作用及底层原理：

虚函数的实现：在有虚函数的类中，类的最开始部分是一个虚函数表的指针，这个指针指向一个虚函数表， 表中放了虚函数的地址，实际的虚函数在代码段(.text)中。当子类继承了父类的时候也会

继承其虚函数表， 当子类重写父类中虚函数时候，会将其继承到的虚函数表中的地址替换为重新写的函数地址。使用了虚函数，会增加访问内存开销，降低效率 。

纯虚函数是一种特殊的虚函数，它在基类中没有实现（或者说实现为空），并且在声明时被赋予一个 **"=0"**

的初始值。纯虚函数的存在意味着派生类必须为这个函数提供实现。如果一个类包含至少一个纯虚函数，那

么这个类被称为抽象类。抽象类不能实例化，只能作为基类为其他派生类提供接口。纯虚函数的目的是为了 定义一个通用的接口，让派生类提供具体的实现。

2.1 访问普通成员函数和虚函数哪快?

访问普通成员函数更快，因为普通成员函数的地址在编译阶段就已确定，因此在访问时直接调 用对应地址的函数，而虚函数在调用时，需要首先在虚函数表中寻找虚函数所在地址

2.2 析构函数是虚函数?

若存在类继承关系并且析构函数中需要析构某些资源时，析构函数需要是虚函数，否则当使用父类指针指向

子类对象，在delete时只会调用父类的析构函数，而不能调用子类的析构函数，造成内存泄露等问题。

将可能会被继承的父类的析构函数设置为虚函数，可以保证当我们new一个子类，然后使用基类指针指向该

子类对象，释放基类指针时可以释放掉子类的空间，防止内存泄漏。

C++默认的析构函数不是虚函数是因为虚函数需要额外的虚函数表和虚表指针，占用额外的内存。而对于不会被继承的类来说，其析构函数如果是虚函数，就会浪费内存。因此C++默认的析构函数不是虚函数，而是 只有当需要当作父类时，设置为虚函数。

2.3 内联函数、构造函数、静态成员函数是虚函数吗?

都不可以。**内联函数**需要在编译阶段展开，而虚函数是运行时动态绑定的，编译时无法展开；

构造函数在进行调用时还不存在父类和子类的概念，父类只会调用父类的构造函数，子类调用子类,因此不存在动态绑定的概念；没有动态绑定的效果，父类构造函数中调用的仍然是父类版本的函数，子类中调用的仍然是子类 版本的函数。

静态成员函数是以类为单位的函数，与具体对象无关，虚函数是与对象动态绑定的，因此是两个不冲突的概念；

2.4 静态函数和虚函数的区别

静态函数没有**this指**针，**static成员不属于任何类对象或类实例**

虚函数需要通过**this ->vptr->vtable->func**

https://www.cnblogs.com/lakeone/p/5967548.htm

2.5 虚函数弊端

https://www.cnblogs.com/findumars/p/4021383.html

问题就是在**子类很少覆盖基类函数实现的时候**内存开销太大，再加上象界面编程这样子类众多的情况（大概 是都不覆盖）

2.6 虚函数可以不加virtul吗

不可以，虚函数必须使用 `virtual` 关键字进行声明，否则编译器将不会将其视为虚函数。否则动态绑定的时候会出错误。

2.7 虚函数表是怎么实现的

- 编译器为每个包含虚函数的类生成一个虚函数表，其中存储了该类中所有虚函数的地址。虚函数表通常是一个数组，每个元素存储一个虚函数的地址。
- 在对象的内存布局中添加一个指向虚函数表的指针，称为虚函数指针（VPointer）。虚函数指针通常是**对象的第一个成员变量**，因为它需要在**对象的构造函数中进行初始**化。
- 在调用虚函数时，编译器会根据**虚函数指针**找到**对应的虚函数表**，并根据函数在虚函数表中的位置来调用正确的函数。具体来说，编译器会将虚函数调用转换为对虚函数指针所指向的虚函数表中的函数地址的调用。

2.8 类的成员模版函数可以是虚函数吗

是的，**类的成员模板函数可以是虚函数**。在C++中，虚函数是一种特殊的成员函数，用于实现多态性。如果一个类中的函数被声明为虚函数，那么在该类的派生类中，可以通过重写该函数来实现不同的行为。而成员模板函数是一种泛型编程机制，可以根据不同的类型参数生成不同的代码。因此，成员模板函数可以与虚函数结合使用，实现更加灵活和通用的代码。

2.9 虚表存放的位置；虚表的大小?

虚表存放的位置和虚表的大小是由编译器和操作系统决定的，不同的编译器和操作系统可能有不同的实现方式。一般来说，**虚表存放在对象的内存布局中，通常是在对象的开头或结尾**处。虚表的大小取决于类中声明的虚函数的数量和类型，每个虚函数在虚表中占用一个指针大小的空间。

在使用虚函数的类中，每个对象都包含一个指向虚表的指针，该指针指向类的虚表。当调用虚函数时，编译器会根据对象的虚表指针找到对应的虚表，并根据函数的索引在虚表中查找函数的地址，然后调用该函数。由于虚表是在编译时生成的，因此虚函数的调用是静态绑定的，而不是动态绑定的。

3.0 虚函数表存在哪里？一个类一个还是一个对象一个?

虚函数表存放在类的内存布局中，通常是在类的开头或结尾处。每个对象都包含一个指向虚函数表的指针，该指针指向类的虚函数表。当调用虚函数时，编译器会根据对象的虚函数表指针找到对应的虚函数表，并根据函数的索引在虚函数表中查找函数的地址，然后调用该函数。

因为每个类都有一个对应的虚函数表，所以虚函数表是类级别的，而不是对象级别的。也就是说，所有属于同一个类的对象共享同一个虚函数表。这是因为虚函数表中存储的是类的虚函数地址，而不是对象的虚函数地址。因此，无论创建多少个对象，它们都共享同一个虚函数表。

二 、内存布局、内存管理

1. C++ 内存布局，类的内存分布

每个对象所占用的存储空间只是该对象的**数据部分**（虚函数指针和虚基类指针也属于数据部分）所占用的存储空间，而不包括函数代码所占用的存储空间。

C++分为四个存储区域：

全局数据区：全局变量，静态数据和常量

代码区：类成员函数和非成员函数代码

栈区：为运行函数而分配的局部变量、函数参数、返回数据、返回地址等；由编译器自动分配释放, 存放函数的参数值,局部变量等，由编译器自动分配释放。

堆区：由程序员分配和释放,若程序员不释放,程序结束时由操作系统回收；

类的定义时，类成员函数是被放在**代码区**，而类的静态成员变量在类定义时就已经在**全局数据区**分配了内存，因而它是**属于类**的。对于**非静态成员变量**，我们是在类的实例化过程中(构造对象)才在栈区或者堆区为其分配内存，是为每个对象生成一个拷贝，所以它是**属于对象**的。

静态成员函数和非静态成员函数都是在类的定义时放在内存的代码区，类可以调用静态成员函数，类的对象可以调用非静态成员函数，因为**this**指针。

程序在加载到内存前，代码区和全局区(`data` 和 `bss`)的大小就是固定的，程序运行期间不能改变。然后，运行可执行程序，系统把程序加载到内存，除了根据可执行程序的信息分出代码区（`text`）、数据区（`data`）和未初始化数据区（`bss`）之外，还额外增加了`栈区`、`堆区`。

2. C++编译过程：

- 预处理(Preprocessing)：将所有的#include头文件以及宏定义替换成其真正的内容
- 编译(Compilation)：预处理之后的程序转换成特定汇编代码**(assembly code)**
- 汇编(Assemble)：汇编代码转换成机器码**(machine code)**
- 链接(Linking)：多个目标文以及所需的库文件**(.so**等**)**链接成最终的可执行文件(

3. 堆区和栈区的区别：

- **分配方式**：栈区的内存分配是由编译器自动完成的，而堆区的内存分配是由程序员手动完成的。
- **内存大小**：栈区的内存大小是固定的，通常在编译时就已经确定了，而堆区的内存大小是动态的，可以根据程序的需要进行调整。
- **内存分配方式**：栈区的内存分配是按照先进后出的原则进行的，也就是说，最后分配的内存最先释放，而堆区的内存分配和释放是由程序员手动控制的。
- **内存访问速度**：栈区的内存访问速度比堆区的内存访问速度要快，因为栈区的内存是连续的，而堆区的内存是离散的。
- **生命周期**：栈区的变量的生命周期是由其所在的函数决定的，函数结束时，栈区的变量会自动被释放，而堆区的变量的生命周期是由程序员手动控制的，需要手动释放。

总的来说，栈区的内存分配和释放是由编译器自动完成的，适用于一些生命周期短、内存需求量小的变量；而堆区的内存分配和释放是由程序员手动控制的，适用于一些生命周期长、内存需求量大的变量。在使用堆区时，需要注意内存泄漏和内存溢出等问题。

4.malloc 和 new 区别

malloc 不调用构造函数，不会对类成员进行初始化，仅仅是分配空间。

内存分配： malloc() 只是分配一段内存，而 new 不仅分配内存，还会调用对象的**构造函数**来初始化

内存。

大小指定： malloc() 函数的大小参数表示要分配的字节数，而 new 运算符的大小参数表示要分配的

对象的数量。

类型安全： new 运算符是类型安全的，因为它返回适当类型的指针，而 malloc() 则返回 void 指针，需要强制转换为适当类型，容易出现类型错误。

扩展性： new 运算符可以重载，允许为自定义类型提供自定义的分配和释放方法，而 malloc() 函数 不能重载

特征	new/delete	malloc/free
分配内存的位置	自由存储区	堆
内存分配失败	抛出异常	返回NULL
分配内存的大小	编译器根据类型计算得出	显式指定字节数
处理数组	有处理数组的new版本new[]	需要用户计算数组的大小后进行内存分配
已分配内存的扩张	不支持	使用realloc完成
分配内存时内存不足	可以指定处理函数或重新制定分配器	无法通过用户代码进行处理
是否可以重载	可以	不可以
构造函数与析构函数	调用	不调用

4.结构体字节对齐：

为了使**CPU**能够对变量进行快速的访问

主要是为了访存效率，因为对齐的字节访存效率更高。计算机底层**存储硬件**比如**内存、CPU cache、寄存器**

等的访问都不是一次一个字节而是**一次一批/一组**。

举例来说：一个4字节的整除原本只需要一次访存，如果组织不当比如**跨过了两个cache line**的边界则需要两次存，开销高了一倍！

字节对齐的本质就是在**内存空间占用和访存效率之间做折中**。C/C++编译器会自动处理struct的内对齐，同

时提供了一些机制让程序员手动控制内存对齐。

5.拷贝构造函数、移动构造、深拷贝、浅拷贝、

5.1构造函数

使用初始化列表，避免重复赋值。

多个参数时，explicit 的作用体现在禁止从一个 {} 表达式初始化。
如果你希望在一个返回 Pig 的函数里用：return {“佩奇”, 80};的话，就不要加 explicit。

int(3.14f) 不会出错，但是 int{3.14f} 会出错，因为 {} 是非强制转换。

使用explicit避免隐式转换。

编译器默认生成的构造函数：无参数，int、float、doubel vodi*这些POD类（完全由类组成的类）不会被初始化为0。

5.2 拷贝构造和拷贝赋值

5.2.1 **拷贝赋值函数≈解构函数+拷贝构造函数**

拷贝构造函数的作用是在Pig尚未初始化时，将另一个Pig拷贝进来，以初始化当前Pig。

Pig pig = pig2; // 拷贝构造

拷贝赋值函数的作用是在**Pig已经初始化时，将当前Pig销毁，同时将另一个Pig拷贝**进来。

Pig pig;　　// 无参构造

pig = pig2;　　// 拷贝赋值

追求性能时推荐用拷贝构造，因为可以避免一次无参构造，拷贝赋值是出于需要临时修改对象的灵活性需要。

5.2.2 **拷贝构造的使用场景**：

- 直接使用另一个对象初始化新对象：
- 将对象作为值传递给函数：
- 从函数返回值为对象的情况

5.2.3 MyClass(const MyClass& other);

- **&和cosnt ,**避免无限递归；不要的拷贝；const避免拷贝对象被修改

5.3编译器自动生成的函数

```
1   Example()构造函数的实现
2   Example(const Example& other) 拷贝构造函数的实现
3   Example(Example&& other) noexcept 移动构造函数的实现
4   Example& operator=(const Example& other)拷贝赋值运算符
5   Example& operator=(Example&& other) noexcept 移动赋值运算符
6   析构函数 ~Example()
```

三五法则：

如果一个类定义了解构函数，那么您必须同时定义或删除拷贝构造函数和拷贝赋值函数，否则出错。

如果一个类定义了拷贝构造函数，那么您必须同时定义或删除拷贝赋值函数，否则出错，删除可导致低效。

如果一个类定义了移动构造函数，那么您必须同时定义或删除移动赋值函数，否则出错，删除可导致低效。

如果一个类定义了拷贝构造函数或拷贝赋值函数，那么您必须最好同时定义移动构造函数或移动赋值函数，否则低效。

5.4 std::move 移动构造

`std::move` 函数可以将一个对象转换为右值引用，从而允许移动语义的使用。当你需要将一个对象的所有权从一个对象转移到另一个对象时，可以使用 `std::move` 函数来实现。

移动是o(1)，拷贝是o(n)。

```
这些情况下编译器会调用移动:
return v2                                                    // v2 作返回值
v1 = std::vector<int>(200)                                   // 就地构造的 v2
v1 = std::move(v2)                                           // 显式地移动
这些情况下编译器会调用拷贝:
return std::as_const(v2)                                     // 显式地拷贝
v1 = v2                                                      // 默认拷贝
注意，以下语句没有任何作用:
std::move(v2)                                                // 不会清空 v2，需要清空可以用 v2 = {} 或 v2.clear()
std::as_const(v2)                                            // 不会拷贝 v2，需要拷贝可以用 { auto _ = v2; }
这两个函数只是负责转换类型，实际产生移动/拷贝效果的是在类的构造/赋值函数里。
移动构造≈拷贝构造+他解构+他默认构造
移动赋值≈拷贝赋值+他解构+他默认构造
```

5.5 深拷贝、浅拷贝

浅拷贝 (shallow copy) 只是对指针的拷贝, 拷贝够两个指针指向同一个内存空间.默认的拷贝行为就是浅拷

贝。

深拷贝 (deep copy) ，不但对指针进行拷贝, 而且对指针指向的内容进行拷贝. 经过深拷贝后的指针是指向两个

不同地址的指针.。

当类持有其它资源时，例如动态分配的内存、指向其他数据的指针等，默认的拷贝构造函数就不能拷贝这些

资源了，我们必须显式地定义拷贝构造函数，以完整地拷贝对象的所有数据。

当对象具有指针类型的成员变量，且需要避免潜在的浅拷贝问题时，应自定义拷贝构造函数来实现深拷贝。

5.6 左值右值

- 左值（lvalue）：指向具有标识符的内存位置的表达式，或者可以引用这样的表达式的表达式。左值可以出现在赋值语句的左边或右边。
- 右值（rvalue）：指向没有标识符的内存位置的表达式，或者可以引用这样的表达式的表达式。右值只能出现在赋值语句的右边。

因为右值引用（rvalue reference）是C++11中的一个新特性，用于提高代码的性能和效率。右值引用只能绑定到右值，而不能绑定到左值。

```
// 移动拷贝构造函数
```

```
    MyString(MyString&& other) {
        // 将other的资源转移给this
        data_ = other.data_;
        size_ = other.size_;
        other.data_ = nullptr;
        other.size_ = 0;
    }
```

5.7 拷贝构造、移动构造区别

拷贝构造函数和移动构造函数是C++中的两个特殊的构造函数，用于创建新对象并将其初始化为另一个对象的副本。它们之间的主要区别在于它们如何处理源对象。

拷贝构造函数会创建一个新的对象，并将其初始化为另一个对象的副本。它通常会被调用在以下情况下：

- 通过值传递参数时，会调用拷贝构造函数来创建参数的副本。
- 在函数返回对象时，会调用拷贝构造函数来创建返回值的副本。
- 在创建对象数组时，会调用拷贝构造函数来创建数组元素的副本。

移动构造函数也会创建一个新的对象，但是它会从一个临时对象中“窃取”资源，而不是创建一个新的副本。它通常会被调用在以下情况下：

- 当一个临时对象被移动到一个新的对象时。
- 当一个对象被转移所有权时，比如使用std::move()函数。

移动构造函数比拷贝构造函数更高效，因为它避免了不必要的复制操作。然而，只有当对象拥有资源时，移动构造函数才有意义。如果对象只是简单的值类型，那么拷贝构造函数和移动构造函数的效果是一样的。

6.内存池、内存泄漏、智能指针

6.1内存池

内存池的基本思想是在程序启动时预先分配一定数量的内存块，并将这些内存块存储在一个池中。当程序需要分配内存时，可以从内存池中获取一个内存块，而不是使用标准库中的 `new` 运算符来分配内存。当程序需要释放内存时，可以将内存块返回给内存池，而不是使用标准库中的 `delete` 运算符来释放内存。

内存池的优点是可以减少内存分配和释放的次数，从而提高程序的性能和效率。由于内存块已经预先分配，因此内存分配的时间和空间开销都可以得到优化。此外，内存池还可以减少内存碎片的产生，从而提高内存的利用率。

内存池的缺点是需要预先分配一定数量的内存块，因此可能会浪费一些内存空间。此外，内存池的实现可能比较复杂，需要考虑线程安全、内存泄漏等问题。

总之，内存池是一种内存管理技术，它可以提高内存分配和释放的效率，适用于需要频繁分配和释放内存的场景。

6.2 C++内存管理，如何避免没有delete

- RALL思想，在构造函数中获取资源，在析构函数中释放资源，保证对象在生命周期结束的时候自动释放内存。
- 使用stl 容器类：容器类都是给封装好的，可以避免手动分配和释放内存
- 避免使用new/delete
- 使用智能指针

6.3 内存泄漏

6.3.1 智能指针

- unique_ptr： 一个独占所有权的智能指针，它确保在其生命周期内只有一个指针可以指向该对象。当 unique_ptr 被销毁时，它所拥有的对象也会被销毁。
- shared_ptr： 一个共享所有权的智能指针，可以让多个指针指向同一个对象。 shared_ptr 维护一个引用计数，当引用计数为 0 时，对象会被销毁。可以使用 make_shared 函数来创建 shared_ptr 。最大的陷阱是循环引用，循环引用会导致堆内存无法正确释放
- weak_ptr： 也是一个共享所有权的指针，但它不会增加引用计数，只是提供了对所指对象的一种弱引用。当 weak_ptr 所指的对象已经被销毁时，它会自动将自己置为 nullptr ，以避免悬挂指针的问题。

  面试回答：

  从unique_ptr说起，可以体现C++封装的思想，unique_ptr禁止拷贝，直接把一个指针传入会报错，所以可以用std::move夺取指针的生命周期大权。或者使用p.get（）获取原始指针。

  unique_ptr 解决**重复释放**的方式是禁止拷贝，这样效率高，但导致使用困难，容易犯错等。相比之下， shared_ptr允许拷贝，解决重复释放的方式是通过引用计数，从而可以保证只要还有存在一个指针指向该对象，这个对象就不会被析构。

  有时候我们希望维护一个 shared_ptr 的**弱引用 weak_ptr**，即：**弱引用的拷贝与解构不影响其引用计数器**。
但有需要时，可以通过 **lock()** 随时产生一个新的 **shared_ptr** 作为强引用。但不 **lock 的时候不影响计数**。
如果失效（计数器归零）则 expired() 会返回 false，且 lock() 也会返回 nullptr。**可以避免循环引用的问题。**

- shared_ptr 管理的对象生命周期，取决于所有引用中，最长寿的那一个。
- unique_ptr 管理的对象生命周期长度，取决于他所属的唯一一个引用的寿命。

6.3.2 shared_ptr怎么保障智能指针线程安全?

- 互斥锁
- 对引用计数进行原子操作
- 使用std::shared_mutex对shared_ptr的构造和析构函数进行加锁

6.3.3 如何知道shared_ptr指针是无效的

当指针所指的对象被删除后，它就变得无效了。可以通过使用**weak_ptr**和**lock()**函数来判断，如果无效将返回一个空的shared_ptr

6.3.4 如何管理引用计数

如果想手动管理引用计数，可以 使用**std::enable_shared_from_this**和**shared_from_this()**函数来实现

6.3.5 unique_ptr 、shared_ptr实现

通过模版类，std::unique_ptr依靠C++11中的移动语义， std::shared_ptr依靠引用计数。

```
template <typename T>
class shared_ptr {
public:
    // 构造函数
    shared_ptr(T* ptr = nullptr) : ptr_(ptr), ref_count_(new int(1)) {}
    // 拷贝构造函数
    shared_ptr(const shared_ptr& other) : ptr_(other.ptr_),
ref_count_(other.ref_count_) {
        ++(*ref_count_);
    }
    // 析构函数
    ~shared_ptr() {
        if (--(*ref_count_) == 0) {
            delete ptr_;
            delete ref_count_;
        }
    }
    // 重载 * 和 -> 运算符
    T& operator*() const {
        return *ptr_;
    }
    T* operator->() const {
        return ptr_;
    }
    // 获取引用计数
    int use_count() const {
        return *ref_count_;
    }
private:
```

```
29      T* ptr_;
30      int* ref_count_; //是一个指针
31  };
32
33  template <typename T>
34  class unique_ptr {
35  public:
36      // 构造函数
37      unique_ptr(T* ptr = nullptr) : ptr_(ptr) {}
38
39      // 析构函数
40      ~unique_ptr() {
41          delete ptr_;
42      }
43
44      // 禁止拷贝构造函数和拷贝赋值运算符
45      unique_ptr(const unique_ptr&) = delete;
46      unique_ptr& operator=(const unique_ptr&) = delete;
47
48      // 移动构造函数和移动赋值运算符
49      unique_ptr(unique_ptr&& other) : ptr_(other.ptr_) {
50          other.ptr_ = nullptr;
51      }
52      unique_ptr& operator=(unique_ptr&& other) {
53          if (this != &other) {
54              delete ptr_;
55              ptr_ = other.ptr_;
56              other.ptr_ = nullptr;
57          }
58          return *this;
59      }
60      // 重载 * 和 -> 运算符
61      T& operator*() const {
62          return *ptr_;
63      }
64      T* operator->() const {
65          return ptr_;
66      }
67
68      // 获取指针
69      T* get() const {
70          return ptr_;
71      }
72
73      // 释放指针
74      T* release() {
75          T* tmp = ptr_;
```

```
76          ptr_ = nullptr;
77          return tmp;
78      }
79
80      // 重置指针
81      void reset(T* ptr = nullptr) {
82          delete ptr_;
83          ptr_ = ptr;
84      }
85
86  private:
87      T* ptr_;
88  };
```

6.3.6智能指针一定能防止内存泄露嘛

智能指针也有一些限制。例如，如果在使用智能指针时没有考虑到对象之间的所有权关系，可能会导致内存泄漏。此外，如果使用不当，智能指针也可能会导致其指向的对象过早释放或重复释放，从而导致程序崩溃或其他问题。

因此，在使用智能指针时，需要仔细考虑对象之间的所有权关系，以及智能指针的生命周期和作用域。同时，也需要遵循良好的编程实践，如避免循环引用、正确使用 move 语义等，以确保智能指针能够正确地管理内存，从而避免内存泄漏问题。

eg:

假设有一个类 A 和一个类 B，它们之间有一个所有权关系，即 A 拥有一个指向 B 对象的指针。如果在使用智能指针时没有考虑到这个所有权关系，可能会导致内存泄漏问题。

例如，如果使用 shared_ptr 来管理 B 对象的内存，但在 A 对象被销毁时没有正确地释放 shared_ptr，就会导致 B 对象的内存泄漏。这是因为 shared_ptr 会在没有任何指向它的智能指针时自动释放内存，但如果 A 对象持有的 shared_ptr 没有被正确释放，那么 B 对象的内存就无法被释放。

为了避免这种问题，可以使用 unique_ptr 或 weak_ptr 来管理 B 对象的内存。unique_ptr 只能有一个指向它的智能指针，因此可以确保 A 对象拥有 B 对象的所有权。而 weak_ptr 则不会增加 B 对象的引用计数，因此可以避免循环引用问题。

6.3.7 unique_ptr 是线程安全的嘛

unique_ptr 本身的线程安全性仅限于其所有权和内存管理。在多线程环境下，如果多个线程同时访问同一个 unique_ptr 对象，可能会导致竞争条件和数据竞争问题。unique_ptr 并不提供内置的线程同步机制，所以在多线程环境下使用时需要注意以下几点：

1. 如果多个线程需要访问同一个 unique_ptr 对象，应当使用互斥锁或其他同步机制来保护对 unique_ptr 对象的访问，确保同一时间只有一个线程能够访问它。
2. 避免在多个线程之间共享 unique_ptr 对象。尽量将 unique_ptr 对象的生命周期限定在单个线程中，或者将 unique_ptr 对象移动到另一个线程中。

需要注意的是，即使 unique_ptr 对象本身的所有权和内存管理是线程安全的，它所管理的对象本身可能并不是线程安全的。在多线程环境下使用 unique_ptr 时，需要仔细考虑对象的线程安全性，并采取相应的措施来保证程序的正确性。

6.3.8 vector 是线程安全的嘛，怎么解决

`std::vector` 本身并不是线程安全的。在多线程环境下，如果有多个线程同时访问和修改同一个 `std::vector` 对象，可能会导致数据竞争和未定义行为。

为了在多线程环境下使用 `std::vector`，你可以采取以下措施来解决线程安全问题：

a. 使用互斥锁（例如 `std::mutex`）或其他同步机制来保护对 `std::vector` 对象的访问。在访问和修改 `std::vector` 对象之前，线程需要先锁定互斥锁。这样可以确保同一时间只有一个线程能够访问 `std::vector` 对象。

b. 如果只有一个线程负责写入 `std::vector`，而其他线程仅负责读取，那么可以使用 `std::shared_mutex` 和 `std::shared_lock` 来实现读写锁。这允许多个线程同时读取 `std::vector`，但在写入时会阻止其他线程访问

c. 避免在多个线程之间共享 `std::vector` 对象。尽量将 `std::vector` 对象的生命周期限定在单个线程中，或者将 `std::vector` 对象移动到另一个线程中。

6.3.9 shared_ptr 存在性能问题嘛

`std::shared_ptr` 是 C++ 标准库提供的一种引用计数智能指针，它可以自动管理动态分配的内存，确保在不再需要时正确地释放。相比于裸指针和 `std::unique_ptr`，`std::shared_ptr` 存在一定程度的性能开销，主要原因如下：

1. 引用计数开销：`std::shared_ptr` 需要维护一个引用计数，以记录有多少个 `shared_ptr` 对象指向同一个资源。每当创建、复制、移动或销毁一个 `shared_ptr` 对象时，都需要更新这个引用计数。这会导致一定的性能开销。
2. 原子操作开销：为了确保在多线程环境下的线程安全性，`std::shared_ptr` 需要使用原子操作来更新引用计数。原子操作通常比普通操作更耗时，尤其是在高度竞争的场景中。
3. 内存开销：`std::shared_ptr` 需要额外的内存来存储引用计数。这意味着，对于同一个资源，`std::shared_ptr` 会比裸指针和 `std::unique_ptr` 占用更多的内存。

尽管 `std::shared_ptr` 存在一定的性能开销，但在许多场景下，这些开销是可以接受的。`std::shared_ptr` 提供了自动内存管理和线程安全性，这可以帮助避免许多常见的内存泄漏和竞争条件问题。

然而，在性能关键的场景中，你可能需要权衡 `std::shared_ptr` 带来的性能开销和便利性。在这些场景下，可以考虑使用其他智能指针（如 `std::unique_ptr`）或自定义内存管理策略来提高性能。需要注意的是，这可能会增加程序的复杂性和出错的风险。因此，在选择合适的内存管理策略时，需要根据具体的需求和场景进行权衡。

6.4 coredump由什么原因导致的

6.4.1 coredump文件找不到报错为什么

在多线程程序中，当你使用 coredump 文件进行 backtrace 时，可能会遇到无法找到错误点的问题。这可能是由以下几个原因导致的：

a. 多线程调度：当程序崩溃时，可能只有一个线程导致了错误，而其他线程仍在正常运行。coredump 文件捕获了程序崩溃时的整个进程状态，包括所有线程的状态。当你进行 backtrace 时，你可能会看到所有线程的栈信息，而不仅仅是导致错误的线程。你需要仔细检查所有线程的栈信息，以找到导致错误的线程。

b. 栈损坏：如果程序崩溃是由栈损坏（例如栈溢出）引起的，那么在进行 backtrace 时，可能无法准确地找到错误点。这是因为栈损坏可能导致栈帧信息丢失或混乱，从而使 backtrace 变得不可靠。在这种情况下，你可能需要使用其他调试方法，如日志记录、断点调试或代码审查，来找到错误原因。

c. 优化：编译器的优化可能会导致 backtrace 信息不准确。例如，编译器可能会内联函数、移除未使用的变量或重新排序指令。这可能会使得栈帧信息与源代码不完全匹配，从而导致在 backtrace 时无法准确地找到错误点。为了获得更准确的 backtrace 信息，你可以尝试在编译程序时禁用优化（例如，使用 `-O0` 选项）。

d. 符号信息丢失：如果你的程序没有包含调试符号信息，那么在进行 backtrace 时，可能无法看到源代码中的函数名和行号。这会使得找到错误点变得困难。为了解决这个问题，你可以在编译程序时启用调试符号（例如，使用 `-g` 选项）。

总之，在多线程程序中进行 coredump backtrace 时，可能会遇到无法找到错误点的问题。这可能是由多种原因导致的，如多线程调度、栈损坏、编译器优化或符号信息丢失。你需要仔细检查 backtrace 信息，并尝试使用其他调试方法来找到错误原因。

6.4.2 如何排查内存泄露、有什么工具，如何定位

内存泄漏是指程序在分配内存后未能正确释放，导致内存资源的浪费。排查内存泄漏通常需要使用专门的工具来分析程序的内存使用情况。以下是一些常用的内存泄漏检测工具和方法，以及如何使用它们定位问题：

a. Valgrind：Valgrind 是一个强大的内存调试和分析工具，它可以帮助你检测内存泄漏、内存访问错误等问题。要使用 Valgrind 检测内存泄漏，你可以运行 `valgrind --leak-check=full your_program`，其中 `your_program` 是你的可执行程序。Valgrind 会在程序运行时监测内存分配和释放，当程序退出时报告潜在的内存泄漏。报告中会包括泄漏的内存大小、分配内存的函数调用栈等信息，这有助于定位问题。

b. AddressSanitizer：AddressSanitizer（ASan）是一个内存错误检测器，它可以检测内存泄漏、越界访问等问题。要使用 AddressSanitizer，你需要在编译程序时启用 `-fsanitize=address` 选项。例如，使用 g++ 编译器时，可以运行 `g++ -fsanitize=address your_program.cpp -o your_program`。然后运行程序，AddressSanitizer 会在程序运行时监测内存使用情况，并在检测到内存泄漏时报告错误。报告中会包括泄漏的内存大小、分配内存的函数调用栈等信息。

c. 使用操作系统工具：某些操作系统提供了内存分析工具，可以帮助你检测内存泄漏。例如，在 Linux 系统中，你可以使用 `ps`、`top` 或 `htop` 等工具来监测程序的内存使用情况。如果你发现程序的内存使用量持续上升，这可能是内存泄漏的迹象。要定位问题，你可能需要结合其他工具（如 Valgrind 或 AddressSanitizer）或代码审查。

d. 使用智能指针和 RAII：C++ 提供了智能指针（如 `std::shared_ptr` 和 `std::unique_ptr`）和资源获取即初始化（RAII）技术，可以帮助你自动管理内存并避免内存泄漏。在编写程序时，尽量使用智能指针和 RAII 来管理内存，以减少内存泄漏的风险。

总之，排查内存泄漏通常需要使用专门的工具，如 Valgrind、AddressSanitizer 或操作系统工具。这些工具可以帮助你监测程序的内存使用情况，并定位潜在的内存泄漏问题。同时，在编写程序时，使用智能指针和 RAII 技术可以有效地避免内存泄漏。

7.并行 锁

7.1 进程与线程区别

进程是一个程序被操作系统拉起加载到内存之后，从开始执行到结束的过程。线程是进程中的一个实体，是被系统独立分配和调度的基本单位。线程是 CPU 执行调度的最小单位。也就是说进程本身并不能获取 CPU 时间，它的线程才可以。

进程>线程，一个进程可以有多个线程。

每个线程共享同样的内存内存空间，开销小。

每个进程拥有独立的内存空间，开销大。高性能并行计算，更好的是多线程。

7.2 多线程 std::thread

多线程不一定分配到分个核，还可以做异步、无阻塞任务。C++11 提供std::thread，是语言级别的多线程是，之前是操作系统级别的pthread。

7.3 t1.join()

线程 `join` 的作用是等待一个线程完成执行。当你调用一个线程对象的 `join` 方法时，当前线程（调用 `join` 方法的线程）会阻塞，直到被 `join` 的线程完成执行。这使得你可以确保某个线程在继续执行其他操作之前已经完成了它的任务。

`join` 的一个典型用途是在主线程中等待其他线程完成执行。例如，假设你有一个程序，它在多个线程中执行一些计算任务。在这些线程完成计算之前，你可能不希望主线程退出，因为这可能导致

计算结果丢失或其他问题。通过在主线程中调用其他线程的 `join` 方法，你可以确保主线程在其他线程完成执行之前不会退出。

t1.join() 主线程等待子线程结束再退出。

```
std::thread t1([n]() {
        for (int i = 1; i <= n; i += 2) {
            std::cout << i << " ";
        }
    });
  t1.join();
```

7.4 t1.detach()

std::thread的解构函数会销毁线程，比如std::thread又被包在一个函数内，函数的花括号结束就自动销毁了。

t1.detach()可以分离该线程-意味这线程的生命周期不再由当前std::thread管理，而是在线程退出后自动销毁自己，不会还是会在进程退出时自动退出。

```
void myfunc(){
    std::thread t1([n]() {
            for (int i = 1; i <= n; i += 2) {
                std::cout << i << " ";
            }
        });
     t1.detach()
}
```

7.5 移动到全局线程池

detach问题是进程退出时，不会等待所有子线程执行完毕。可以把t1移动一个全局变量去，延长它的生命周期到myfunc函数体外面。

也可以封装一个**类 ThreadPool**，创建一个全部变量，join的操作放在析构函数里面。

```
std::vector<std::thread> pool;
void myfunc(){
    std::thread t1([n]() {
            for (int i = 1; i <= n; i += 2) {
                std::cout << i << " ";
            }
        });
     pool.push_back(std::move(t1));
```

```
}
int main(){
    myfunc();
    for (auto &t:pool) t.join();//等待线程全部执行完毕
}
```

7.6 std::async异步

多线程是一种并发执行的方式，适用于需要同时执行多个任务的场景；**异步是一种非阻塞的执行方式**，适用于需要等待某个操作完成后再执行其他任务的场景。在实际应用中，多线程和异步可以结合使用，以达到更好的性能和效果。

接受一个带有返回值的lambda，返回一个std::future对象。

lambda函数体将在另一个线程中执行。

std::wai()

std::wati_for()等待

```
std::future<int> fret = std::async([&] {
    return download();
});

int ret = fret.get(); //等待
```

7.7 std::lock_guard 和 std::unique_lock

`std::lock_guard` 和 `std::unique_lock` 都是用于管理互斥锁（如 `std::mutex`）的 RAII（资源获取即初始化）包装器。它们都会在构造时自动锁定互斥锁，并在析构时自动解锁。它们的主要区别在于灵活性和功能。

`std::lock_guard` 是一个简单的 RAII 包装器，它只提供了基本的互斥锁管理功能。当你创建一个 `std::lock_guard` 对象时，它会自动锁定给定的互斥锁。当 `std::lock_guard` 对象销毁时（例如在离开作用域时），它会自动解锁互斥锁。`std::lock_guard` 不支持显式地解锁互斥锁或在构造时不锁定互斥锁。

`std::unique_lock` 提供了更多的灵活性和功能。与 `std::lock_guard` 类似，它也会在构造时自动锁定互斥锁并在析构时自动解锁。但是，`std::unique_lock` 还允许在运行时显式地解锁互斥锁、重新锁定互斥锁、或在构造时不锁定互斥锁。这使得 `std::unique_lock` 更适合用于条件变量和更复杂的同步场景。例如：

可以保证mtx.lock()和mtx.unlock（）之前的代码段同一时间只有一个线程在执行。

std::lock_guard，符合 RAII 思想的上锁和解锁，解构函数自动调用mtx.unlock()，自动解锁，可以用 {}来限制std::lock_guard的作用域。

```
std::mutex mtx;
mtx.lock();
mtx.unlock(); 解锁
```

std::unique_lock，可以提前解锁，可以使用std::defer_lock做参数，需要手动调用lock。

无阻塞的try_lock()，在上锁失败的时候不会陷入等待，直接返回false。

try_lock_for()，指定等待一段时间。

7.8 死锁

两个线程可能不是同步的，双方都在等待对方释放锁，但是因为等待而无法释放，无限制的等待

```
t1 mtx1.lock
t2 mtx2.lock
t1 mtx2.lock 失败，陷入等待
t2 mtx1.lock 失败，陷入等待
```

解决办法：

- 同一个线程永远不要持有两个锁
- 保持双方上锁的顺序一致
- 使用std::lock(mtx1, mtx2)，同时对多个上锁

同一个线程，重复锁，也有可能造成死锁：

std::recursive_mutex 信号量，lock+1,unlock-1,为 0才能解锁。

7.9 读写锁 std::shared_mutex

读可以共享、写必须独占，读和写不能共存

std::shared_mutex

读写锁：shared_mutex

- 为此，标准库提供了 std::shared_mutex。
- 上锁时，要指定你的需求是**拉**还是**喝**，负责调度的读写锁会帮你判断要不要等待。
- 这里 push_back() 需要修改数据，因需求此为**拉**，使用 lock() 和 unlock() 的组合。
- 而 size() 则只要读取数据，不修改数据，因此可以和别人共享一起**喝**，使用 lock_shared() 和 unlock_shared() 的组合。

```
[100%] Built target cpptest
2000
bate@archrtx ~/Codes/course/05/05_struct/04 (master) $
```

```cpp
#include <vector>
#include <shared_mutex>

class MTVector {
    std::vector<int> m_arr;
    mutable std::shared_mutex m_mtx;

public:
    void push_back(int val) {
        m_mtx.lock();
        m_arr.push_back(val);
        m_mtx.unlock();
    }

    size_t size() const {
        m_mtx.lock_shared();
        size_t ret = m_arr.size();
        m_mtx.unlock_shared();
        return ret;
    }
};

int main() {
    MTVector arr;

    std::thread t1([&] () {
        for (int i = 0; i < 1000; i++)
            arr.push_back(i);
    });

    std::thread t2([&] () {
        for (int i = 0; i < 1000; i++)
            arr.push_back(1000 + i);
    });

    t1.join();
    t2.join();

    std::cout << arr.size() << std::en

    return 0;
```


7.10 条件变量 condition_variable

- **用于实现生产者、消费者队列的形式**

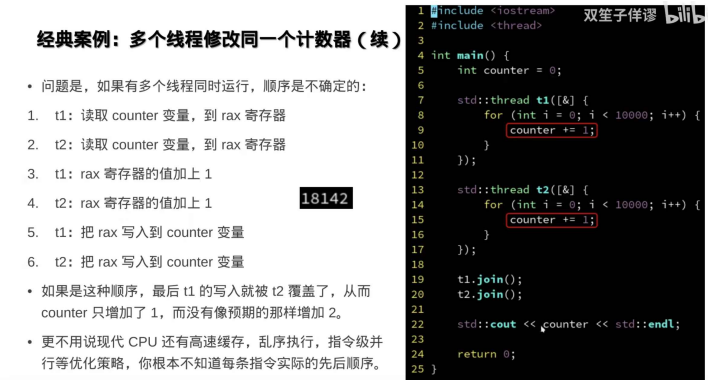
在这个示例中，我们定义了一个线程函数 `worker`，它会一直等待条件变量 `cv` 满足某个条件后再继续执行。在主线程中，我们创建了一个新线程 `t` 并让它执行 `worker` 函数。然后我们等待 2 秒钟，模拟某个条件满足后通知 `worker` 线程。在主线程中，我们使用一个互斥锁 `mtx` 来保护共享变量 `ready`，并使用条件变量 `cv` 等待 `ready` 变为 `true`。当 `ready` 变为 `true` 时，我们通过 `cv.notify_one()` 通知 `worker` 线程条件变量已经满足。

在这个示例中，我们使用了互斥锁 `mtx` 和条件变量 `cv` 来实现线程间同步。当 `ready` 变为 `true` 时，`worker` 线程会被唤醒，并继续执行。如果 `ready` 没有变为 `true`，`worker` 线程会一直等待，直到被唤醒。

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void worker() {
    std::unique_lock<std::mutex> lck(mtx);
    while (!ready) {
        cv.wait(lck);
    }
```

```
    std::cout << "Worker thread is running..." << std::endl;
}

int main() {
    std::thread t(worker);

    std::this_thread::sleep_for(std::chrono::seconds(2));

    {
        std::lock_guard<std::mutex> lck(mtx);
        ready = true;
        cv.notify_one();
    }

    t.join();

    return 0;
}
```

7.11 原子操作



暴力，使用 mutex上锁，太过重量级，会让线程挂起，从而需要系统调用，进入内核层，调度到其他线程执行，有很大的开销，影响效率。

```
lock xadd %eax,(%rdx)
```

建议用 atomic：有专门的硬件指令加持

- 因此可以用更轻量级的 atomic，对他的 += 等操作，会被编译器转换成专门的指令。
- CPU 识别到该指令时，会锁住内存总线，放弃乱序执行等优化策略（将该指令视为一个同步点，强制同步掉之前所有的内存操作），从而向你保证该操作是**原子 (atomic)** 的（取其不可分割之意），不会加法加到一半另一个线程插一脚进来。
- 对于程序员，只需把 int 改成 atomic<int> 即可，也不必像 mutex 那样需要手动上锁解锁，因此用起来也更直观。

20000

```cpp
#include <iostream>
#include <thread>
#include <atomic>

int main() {
    std::atomic<int> counter = 0;

    std::thread t1([&] {
        for (int i = 0; i < 10000; i++) {
            counter += 1;
        }
    });

    std::thread t2([&] {
        for (int i = 0; i < 10000; i++) {
            counter += 1;
        }
    });

    t1.join();
    t2.join();

    std::cout << counter << std::endl;

    return 0;
}
```

注意：请用 +=，不要让 + 和 = 分开

- 不过要注意了，这种写法：

11854

- counter = counter + 1; // 错，不能保证原子性
- counter += 1; // OK，能保证原子性
- counter++; // OK，能保证原子性

三、关键字

1.谈谈static和const理解

static关键字用于定义静态变量，即在程序运行期间只分配一次内存，不会随着函数的调用而被多次创建和销毁。静态变量可以在函数内部或者类的成员函数中定义，也可以在全局作用域中定义。静态变量的作用域和生命周期与定义的位置有关，如果在函数内部定义，则只能在该函数内部访问；如果在类的成员函数中定义，则可以被该类的所有对象共享。

const关键字用于定义常量，即在程序运行期间不可修改的变量。常量可以在函数内部或者类的成员函数中定义，也可以在全局作用域中定义。常量的值在定义时就已经确定，不能被修改。需要注意的是，const关键字还可以用于函数的参数和返回值类型中，用于指定函数的输入和输出参数不可修改。

1.1为何static成员函数不能为const函数

在 C++ 中，`const` 关键字用于修饰成员函数，表示该函数不会修改对象的状态。而 `static` 关键字用于修饰静态成员函数，表示该函数不依赖于任何对象，只能访问静态成员变量和静态成员函数。

由于 `const` 成员函数是针对对象的，而 `static` 成员函数是针对类的，因此 `static` 成员函数不能被声明为 `const`。如果将 `static` 成员函数声明为 `const`，编译器会报错，因为 `const` 成员函数需要访问对象的状态，而 `static` 成员函数没有对象的状态可供访问。

2.static

- 函数中的静态变量：仅初始化一次，保留上次的调用结果
- 静态类对象：分配到静态存储区，一定要程序结束
- 类中的静态成员函数：只能访问静态成员变量、静态成员函数
- 类中的静态成员变量：被所有类对象共享

3.const 和constexp

- 修饰参数、返回值、类函数
- 修饰类成员函数，必须在构造函数中列表初始化
- const的函数修改某个成员变量，可以将变量声明为mutable
- const对象不可以调用非const成员函数，非const对象可以调用const成员函数

const：

const 关键字用于定义一个不可变的常量。一旦声明为 const 的变量被初始化，它的值就不能再更改。

const 可以用于修饰变量、函数参数、成员函数（例如在成员函数后面加 const 表示该成员函数不会修改类的成员变量）等。

const 常量在编译时或运行时进行初始化。因此，可以使用运行时计算的值对其进行初始化。

constexpr：

constexpr 关键字用于定义在编译时确定值的常量。这意味着 constexpr 常量的值在编译时就已经计算出来，并且在整个程序执行期间保持不变。

constexpr 可以用于变量、函数（此时表示函数的返回值在编译时确定）等。

constexpr 常量的计算和初始化只能在编译时进行，因此不能使用运行时计算的值对其进行初始化

4.noexcept

在C++11中，引入了noexcept关键字，用于指示函数是否可能抛出异常。noexcept关键字可以用于函数声明和函数定义中，用于表示函数是否会抛出异常。如果一个函数被声明或定义为noexcept，则表示该函数不会抛出异常；如果一个函数没有被声明或定义为noexcept，则表示该函数可能会抛出异常。

一般析构函数、const函数、移动构造/赋值函数、swap函数声明为noexcept

5.override

在C++11中，引入了override关键字，用于指示派生类中的虚函数覆盖了基类中的虚函数。override关键字可以用于类的成员函数声明中，用于表示该函数是一个虚函数，并且覆盖了基类中的同名虚函数。

四、C++11

1.lamda函数

Lambda函数是C++11引入的一种匿名函数，也称为闭包（Closure）。Lambda函数可以在需要函数对象的任何地方使用，例如作为函数参数、返回值、变量等。Lambda函数的语法类似于函数定义，但可以在函数内部定义局部变量，并可以访问外部作用域的变量。

[capture-list] (parameter-list) -> return-type { function-body }

值捕获（a）、引用捕获（&）、隐式捕获（使用this指针）

2.vector和 list 和 map

- 底层数据结构

`vector` 底层使用的是**动态数组**（dynamic array），即连续的内存空间，可以通过下标访问元素。当 `vector` 的大小超过了当前分配的内存空间时，会重新分配一块更大的内存空间，并将原有的元素复制到新的内存空间中。因此， `vector` 的插入和删除操作可能会导致内存重新分配和元素复制，时间复杂度为O(n)。

`list` 底层使用的是双向链表（doubly linked list），即每个元素都包含一个指向前驱和后继元素的指针，可以通过迭代器访问元素。由于 `list` 的元素不是连续的，因此插入和删除操作不需要移动其他元素，时间复杂度为O(1)。

- 操作复杂度

`vector` 支持随机访问，可以通过下标访问元素，时间复杂度为O(1)。但是插入和删除操作可能会导致内存重新分配和元素复制，时间复杂度为O(n)。

`list` 不支持随机访问，只能通过迭代器访问元素，时间复杂度为O(n)。但是插入和删除操作不需要移动其他元素，时间复杂度为O(1)。

- 适用场景。

应根据具体的场景选择合适的容器。如果需要频繁访问元素，可以选择 `vector` ；如果需要频繁插入和删除元素，可以选择 `list` 。

2.1 reserse 和 resize

`reserve()` 和 `resize()` 函数的主要区别在于它们是否改变容器的大小。 `reserve()` 函数仅预留空间，而 `resize()` 函数则改变容器的大小。

2.2 vector 在栈还是堆上，里面的 array 在哪

`std::vector` 是一个动态数组容器，它的内部结构通常包括一个指针、一个大小（size）和一个容量（capacity）。 `std::vector` 对象本身可以存储在栈上或堆上，具体取决于它是如何声明和分配的。然而， `std::vector` 管理的动态数组（即存储元素的内存区域）始终位于堆上。

3.map 和 unordered_map

https://blog.csdn.net/qq_21989927/article/details/109690980

内部实现结构不同： map 使用红黑树实现， unordered_map 使用哈希表实现。

有序性： map 内部以红黑树为基础实现，保证元素有序。而 unordered_map 是无序的，元素存储的位置由哈希函数计算而来，没有特定的顺序。

查找效率： unordered_map 的查找时间复杂度是O(1)，也就是常数级别的。而 map 的查找时间复杂度是O(log n)，与元素个数成对数关系，因为它是基于红黑树实现的。

插入和删除元素： unordered_map 的插入和删除元素的时间复杂度也是O(1)，而 map 的插入和删除元素的时间复杂度是O(log n)。

内存占用：由于哈希表的冲突处理和节点的指针等因素， unordered_map 相对于 map 需要更多的内存。

因此，如果需要对元素进行快速的查找，可以考虑使用 unordered_map ，如果需要有序的存储和访问元素，则可以使用 map 。

数据量不是特别大,两者差距不是特别大。

`set` 是一个基于红黑树实现的有序集合，它可以自动对元素进行排序，并且支持快速的插入、删除和查找操作。由于 `set` 是基于红黑树实现的，因此它的插入、删除和查找操作的时间复杂度都是O(log n)。

4.模板是什么，底层怎么实现的?

- **编译器并不是把函数模板处理成能够处理任意类的函数**；编译器从函数模板通过具体类型产生不同的函数；
- 编译器会对函数模板进行两次编译：
  - 在声明的地方对模板代码本身进行编译，
  - 在调用的地方对参数替换后的代码进行编译。
- 这是因为**函数模板要被实例化后才能成为真正的函数**，在使用函数模板的源文件中包含函数模板的头文件，如果该头文件中只有声明，没有定义，那编译器无法实例化该模板，最终导致链接错误。
- 模板可以重载返回值, 函数重载不行
- 如果我们试图通过在头文件中定义函数模板, 在 `cpp` 文件中实现函数模板, 那么我们必须在在实现的那个 `cpp` 文件中**手动实例化**, 也就是使用你需要使用的参数替换模板, 从而使得编译器为你编译生成相应参数的模板函数.

4.1 编译器如何去实现对C++模板的编译

编译器会**解析模板代码、实例化模板、生成代码、编译代码**。

模板的编译过程可以分为两个阶段：模板定义和模板实例化。

模板定义:编译器会对模板进行语法和语义分析，并生成模板的抽象语法树（AST）。模板定义阶段不会生成任何代码，只是对模板进行检查和解析，以确保模板的正确性和合法性。

模板实例化:在模板实例化阶段，编译器会根据模板定义中的类型参数生成具体的代码。具体来说，编译器会根据模板定义中的类型参数，将模板中的类型替换为实际的类型，并生成对应的函数或类。这个过程称为模板实例化。

模板实例化可以分为两种类型：显式实例化和隐式实例化。显式实例化是指在代码中显式地声明模板的类型参数，并要求编译器生成对应的代码。隐式实例化是指在代码中使用模板时，编译器自动根据实际的类型参数生成对应的代码。

在生成代码后，编译器会将代码编译成目标代码，以便在程序运行时执行。在编译代码时，编译器会对代码进行优化，以提高代码的执行效率。

需要注意的是，C++模板的编译过程比较复杂，可能会导致编译时间较长。为了提高编译效率，编译器通常会采用一些优化策略，例如**模板代码的预编译、模板代码的缓存**等。此外，为了避免模板代码的重复实例化，C++11引入了**extern template**机制，可以在**模板实例化时避免重复生成代码**，从而提高编译效率。

4.2 模板函数可以偏特化吗

C++中的模板函数可以**进行特化**，但是不能进行偏特化。

偏特化只适用于模板类，而不适用于模板函数。

需要注意的是，**模板类的偏特化必须在全局作用域**中进行定义，而不能在类或函数内部进行定义。C++限制。

模板函数的特化必须在全局作用域中进行定义，而不能在类或函数内部进行定义。

4.3 类的成员函数可以偏特化吗

C++中的类成员函数不能进行偏特化，只能进行特化。**特化的成员函数必须在全局作用域中进行定义**

```
template<typename T>
class MyClass {
public:
    void foo(T t);
};

template<typename T>
void MyClass<T>::foo(T t) {
    // ...
}

// 特化的成员函数必须在全局作用域中进行定义
template<>
void MyClass<int>::foo(int t) {
    // ...
}
————————————————————————————————————————
// 原始的模板类
template<typename T, typename U>
class MyPair {
public:
    MyPair(T t, U u) : first(t), second(u) {}
private:
    T first;
    U second;
};

// 偏特化的模板类
template<typename T>
class MyPair<T, T> {
public:
    MyPair(T t1, T t2) : first(t1), second(t2) {}
private:
    T first;
    T second;
};
```

五、设计模式

1.工厂模式

C++中的工厂模式是一种创建型设计模式，用于创建对象而不需要暴露对象的创建逻辑。工厂模式将对象的创建过程封装在一个工厂类中，客户端只需要通过工厂类来创建对象，而不需要直接调用对象的构造函数。工厂模式可以提高代码的可维护性和可扩展性，因为它将对象的创建过程与客户端代码分离开来，使得客户端代码更加简洁和易于维护。

简易工厂模式、工厂方法模式。

工厂方法模式是工厂模式的一种更加灵活和可扩展的实现方式，它使用一个抽象工厂类来定义对象的创建接口，然后由具体的工厂类来实现对象的创建。工厂方法模式可以根据需要添加新的工厂类来支持新的对象类型，而不需要修改抽象工厂类的代码。

2.单例模式

用于确保一个类只有一个实例，并提供全局访问点。单例模式可以避免多个实例之间的冲突和资源浪费，同时也可以提高代码的可维护性和可扩展性。

静态局部变量来创建单例对象，可以使用工厂模式来创建单例对象，并将单例对象保存在工厂类中，以确保在任何时候都只有一个实例。

```
class Config {
public:
    static Config& getInstance() {
        static Config instance;
        return instance;
    }
private:
    Config() {}
    Config(const Config&) = delete;
    Config& operator=(const Config&) = delete;
};

class ConfigFactory {
public:
    static Config& getConfig() {
        static Config config;
        return config;
    }
};

int main() {
    Config& config1 = ConfigFactory::getConfig();
    Config& config2 = ConfigFactory::getConfig();
```

```
    assert(&config1 == &config2);
    return 0;
}
```

3.观察者模式

观察者模式是一种行为型设计模式，用于在对象之间建立一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会收到通知并自动更新。观察者模式可以提高代码的可维护性和可扩展性，因为它将对象之间的依赖关系解耦，使得对象之间的交互更加灵活和可扩展。

六、遇到的面试问题

1.静态库和动态库区别：

有时候我们会有多个可执行文件，他们之间用到的某些功能是相同的，我们想把这些共用的功能做成一个库，方便大家一起共享。库中的函数可以被可执行文件调用，也可以被其他库文件调用。

库文件又分为**静态库文件**和**动态库文件**。

其中静态库相当于直接把代码**插入到生成的可执行文件**中，会导致体积变大，但是**只需要一个文件**即可运行。

而动态库则只在生成的可执行文件中**生成“插桩”函数**，当可执行文件被加载时会读取指定目录中的.dll文件，加载到内存中空闲的位置，并且替换**相应的“插桩”指向的地址为加载后的地址**，这个过程称为**重定向**。这样以后函数被调用就会跳转到动态加载的地址去。

静态库的优点：

- 简单易用：静态库是将代码编译成一个独立的文件，可以直接链接到应用程序中，使用起来非常简单。
- 性能高：静态库在编译时会被完全链接到应用程序中，因此在运行时不需要额外的加载和链接过程，可以提高应用程序的性能。
- 稳定可靠：静态库的代码是固定的，不会受到外部环境的影响，因此更加稳定可靠。

静态库的缺点：

- 占用空间大：静态库会将所有代码都包含在应用程序中，因此会占用较大的空间。
- 更新困难：如果静态库的代码发生了更新，需要重新编译和链接应用程序才能使用新的代码。

动态库的优点：

- 节省空间：动态库的代码是独立于应用程序的，多个应用程序可以共享同一个动态库文件，因此可以节省空间。
- 更新方便：如果动态库的代码发生了更新，只需要替换动态库文件即可，不需要重新编译和链接应用程序。
- 共享性强：多个应用程序可以共享同一个动态库文件，可以提高代码的重用性和维护性。

动态库的缺点：

- 性能稍低：动态库在运行时需要进行加载和链接的过程，因此相比静态库会稍微降低一些性能。
- 依赖性强：动态库的使用需要依赖于操作系统的支持，不同操作系统可能有不同的动态库格式，因此在跨平台开发时需要注意兼容性问题。

综上所述，静态库适用于对性能要求较高、独立部署的场景，而动态库适用于节省空间、方便更新和共享的场景。具体选择哪种库取决于项目的需求和实际情况。

库文件 iddefine避免重复定义**

2.C++三大特性：封装、继承、多态

封装，将多个逻辑上相关的变量包装为一个类，比如把一个array 起始指针等打包为一个vector，避免程序员出错。

3.C++思想 RALL

获取资源就被视为初始化，释放就销毁，用了构造函数和析构函数，C++ 标准保证当异常发生时，会调用已创建对象的解构函数。

4.inline函数的作用

inline关键字用于定义内联函数，即在函数调用处直接将函数体展开，而不是通过函数调用的方式执行函数，为了解决一些频繁调用的小函数大量消耗栈空间（**栈内存**）的问题。

inline的使用时**有所限制的**，inline只适合函数体内部代码简单的函数使用，不能包含复杂的结构控制语句例如while、switch，并且不能内联函数本身不能是直接递归函数。

建议：inline函数的定义放在头文件中

内联是以**代码膨胀（复制）为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率**。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。

4.1宏定义和inline函数区别

宏定义是在预处理阶段进行替换，而inline函数是在编译阶段进行展开。宏定义是通过简单的文本替换来实现的，而inline函数是通过编译器进行展开来实现的。因此，宏定义的展开是在预处理阶段进行的，而inline函数的展开是在编译阶段进行的。

宏定义没有类型检查，而inline函数有类型检查。宏定义是简单的文本替换，没有类型检查，因此可能会导致一些类型错误。而inline函数是在编译阶段进行展开，有类型检查，可以避免这些问题。

宏定义可以定义任何代码，而inline函数只能定义函数。宏定义可以定义任何代码，包括变量、表达式、语句等，而inline函数只能定义函数。

宏定义可以定义递归函数，而inline函数不能定义递归函数。宏定义可以定义递归函数，因为宏定义是在预处理阶段进行展开的，可以展开任意次数。而inline函数不能定义递归函数，因为在展开时可

能会导致无限递归。

宏定义可以定义带有副作用的代码，而inline函数不能定义带有副作用的代码。宏定义可以定义带有副作用的代码，例如修改全局变量、调用其他函数等，而inline函数不能定义带有副作用的代码，因为在展开时可能会导致意外的行为。

5.class与struct的区别

在 `class` 中，默认的访问权限是 `private` ，而在 `struct` 中，默认的访问权限是 `public` 。这意味着，在 `class` 中定义的成员变量和成员函数默认是私有的，而在 `struct` 中定义的成员变量和成员函数默认是公有的。

在 `class` 中，继承默认是 `private` 继承，而在 `struct` 中，继承默认是 `public` 继承。这意味着，在 `class` 中定义的基类成员和成员函数默认是私有的，而在 `struct` 中定义的基类成员和成员函数默认是公有的。

在 `class` 中，使用 `typedef` 或 `using` 定义类型别名时，默认的访问权限是 `private` ，而在 `struct` 中，默认的访问权限是 `public` 。

6.网络字节序，怎么判断大小端，不同 CPU 芯片下怎么样

网络字节序（Network Byte Order）是一种通信协议中使用的字节序，它是大端字节序（Big-Endian）。

在大端字节序中，数据的高位字节存储在低地址，低位字节存储在高地址。例如，整数0x12345678在大端字节序中的存储顺序为0x12 0x34 0x56 0x78。

判断CPU的字节序可以通过查看CPU的架构和操作系统的类型来确定。例如，x86架构的CPU和Windows操作系统使用的是小端字节序（Little-Endian），而PowerPC架构的CPU和Mac OS X操作系统使用的是大端字节序（Big-Endian）。

在网络通信中，由于不同的计算机可能使用不同的字节序，因此需要将数据转换为网络字节序进行传输。

7.C++用户态线程和内核态线程是一一对应的吗

C++ 中的线程通常是指操作系统中的内核态线程。C++11 标准库提供了 `std::thread` 类来表示和管理线程。当你使用 `std::thread` 创建一个线程时，实际上是在操作系统中创建了一个内核态线程。这些内核态线程由操作系统调度和管理。

用户态线程（也称为绿色线程或轻量级线程）是在用户程序中实现的线程，而不是由操作系统内核管理。用户态线程的调度和管理由用户程序（例如，一个线程库）负责。用户态线程通常比内核态线程更轻量级，因为它们不需要进行系统调用来进行上下文切换。然而，在 C++ 标准库中没有直接提供用户态线程的支持。

C++ 中的内核态线程和用户态线程之间并不是一一对应的。一个内核态线程可以与多个用户态线程关联，这种情况下，用户态线程的调度和管理由用户程序（例如，一个线程库）负责。同样，一个

用户态线程也可以在多个内核态线程上运行，例如在一些协程库中，协程（可以看作是用户态线程）可以在不同的内核态线程上切换执行。

总之，C++ 中的内核态线程和用户态线程并不是一一对应的。具体的对应关系取决于你使用的线程库和编程模型。

8.C++有守护线程的概念嘛

C++ 本身没有守护线程（daemon thread）的概念，但你可以用 C++ 实现类似守护线程的功能。

守护线程通常是指在后台运行的线程，它们执行一些辅助任务，例如日志记录、垃圾回收或系统监控。守护线程通常在程序启动时创建，并在整个程序运行期间一直运行。当主程序结束时，守护线程会自动退出。

在 C++ 中，你可以创建一个线程，并将其分离（detach）以实现类似守护线程的功能。分离线程会在后台运行，不会阻止主线程退出。当主线程退出时，分离线程会自动被终止。

9.一个线程挂了有什么影响

一个线程挂掉（意外终止）可能会对程序产生以下影响：

a. 资源泄漏：挂掉的线程可能无法正确地释放其占用的资源，例如内存、文件句柄或其他系统资源。这可能导致资源泄漏，从而降低程序的性能或导致其他问题。

b. 数据不一致：如果一个线程在执行临界区代码时挂掉，可能会导致共享数据处于不一致或不完整的状态。这可能会影响其他线程的执行，导致程序错误或崩溃。

c. 程序崩溃：如果挂掉的线程是程序的主线程，或者是执行关键任务的线程，其异常终止可能导致整个程序崩溃。

d. 任务未完成：挂掉的线程可能无法完成其分配的任务，从而影响程序的整体功能或性能。

为了减轻线程异常终止的影响，可以采取以下措施：

a. 在线程中使用异常处理，确保异常不会导致线程意外终止。

b. 使用资源管理类（例如智能指针、RAII 包装器等），确保线程在退出时正确地释放资源。

c. 在线程中使用同步机制（例如互斥锁、信号量等），确保线程在访问共享数据时不会导致数据不一致。

d. 对于关键任务，可以使用监控线程来检测线程的状态，并在必要时重新启动挂掉的线程。

总之，一个线程挂掉可能会对程序产生严重的影响。在编写多线程程序时，需要考虑线程的异常处理、资源管理和同步机制，以确保线程在意外终止时能够最小化对程序的影响。

10. 并行和并发的区别

并行（Parallelism）和并发（Concurrency）是两个经常被混淆的概念，它们都涉及到多个任务的执行，但在实现方式和目标上有所不同。

并发（Concurrency）是指多个任务在同一时间段内交替执行，它们可能是在单个处理器上通过时间片切换（time slicing）来共享 CPU 时间，也可能是在多个处理器上同时执行。并发的主要目标是提高资源利用率和提高程序的响应性。在并发执行的任务之间，可能需要进行同步和协调，以确保正确的执行顺序和数据一致性。

并行（Parallelism）是指多个任务在同一时刻同时执行，它通常需要多个处理器或多核 CPU。并行的主要目标是提高程序的执行速度和处理能力。在并行执行的任务之间，可能需要进行数据拆分和结果合并，以确保正确的计算结果。

以下是一个简单的例子来说明并发和并行的区别：

假设你有一个程序，它需要处理大量的数据。在并发模式下，程序可以同时处理多个任务，例如在处理一部分数据时，还可以接收新的数据请求。这样可以确保程序在处理数据时仍具有良好的响应性。然而，并发模式下的任务在单个处理器上交替执行，因此总体处理速度可能受到限制。

在并行模式下，程序可以将数据拆分为多个部分，并在多个处理器上同时处理这些部分。这样可以显著提高程序的处理速度和吞吐量。然而，并行模式下的任务可能需要额外的同步和协调机制，以确保正确的计算结果。

总之，并发和并行是两个相关但不同的概念。并发关注多个任务在同一时间段内交替执行，以提高资源利用率和程序响应性；而并行关注多个任务在同一时刻同时执行，以提高程序的执行速度和处理能力。在实际应用中，这两种模式可以结合使用，以实现高效且响应迅速的程序。

11.为什么切换线程有额外的开销

切换线程（线程切换或上下文切换）是指操作系统在多个线程之间切换执行的过程。线程切换会产生额外的开销，主要原因如下：

a. 保存和恢复寄存器状态：在线程切换时，操作系统需要保存当前线程的寄存器状态，以便在恢复执行时可以从上次暂停的地方继续。同样，操作系统还需要恢复即将执行的线程的寄存器状态。这些操作涉及到许多寄存器的读取和写入，会产生一定的开销。

b. TLB（Translation Lookaside Buffer）失效：TLB 是一种高速缓存，用于加速虚拟地址到物理地址的转换。在线程切换时，操作系统可能需要刷新 TLB 或更新 TLB 条目，以确保新线程可以访问正确的内存地址。这会导致 TLB 失效，从而降低内存访问的性能。

c. 缓存失效：现代处理器通常具有多级缓存（如 L1、L2 和 L3 缓存），用于加速内存访问。在线程切换时，新线程可能需要访问不同的内存区域，这会导致缓存失效。缓存失效会降低内存访问的性能，从而增加线程切换的开销。

d. 内核态和用户态切换：线程切换通常需要在内核态和用户态之间切换。内核态切换涉及到特权级别的更改、栈切换等操作，这会产生额外的开销。

e. 调度器开销：在线程切换时，操作系统的调度器需要确定下一个要执行的线程。调度器可能需要遍历线程列表、更新线程优先级等操作，这会产生一定的开销。

线程切换的开销可能会降低多线程程序的性能。为了减轻线程切换的影响，可以采取以下措施：

a. 限制线程数量：尽量将线程数量限制在处理器核心数的范围内，以减少线程切换的频率。

b. 使用线程池：线程池可以复用已创建的线程，从而减少线程创建和销毁的开销。

c. 优化同步机制：避免不必要的锁争用和阻塞，以减少线程切换的频率。

d. 使用协程或轻量级线程：协程（如 C++20 的 `std::coroutine`）或轻量级线程（如用户态线程）可以在用户空间进行上下文切换，通常比内核态线程切换具有更低的开销。

总之，线程切换会产生额外的开销，这可能会影响多线程程序的性能。在编写多线程程序时，需要考虑线程切换的开销，并采取相应的措施来减轻其影响。

12.grpc相关

gRPC（gRPC Remote Procedure Calls）是一个高性能、开源的远程过程调用（RPC）框架，由 Google 开发。gRPC 使用 HTTP/2 作为传输协议，使用 Protocol Buffers 作为数据序列化格式。gRPC 的网络架构主要包括以下组件：

a. 服务端（Server）：gRPC 服务端是一个监听客户端连接和请求的应用程序。服务端实现了特定的服务接口，用于处理客户端的 RPC 调用。服务端可以使用多种编程语言和库来实现，例如 C++、Java、Python、Go 等。

b. 客户端（Client）：gRPC 客户端是一个向服务端发起连接和请求的应用程序。客户端使用服务端提供的服务接口进行 RPC 调用。客户端同样可以使用多种编程语言和库来实现。

c. 服务定义（Service Definition）：gRPC 使用 Protocol Buffers（protobuf）定义服务接口和消息格式。服务定义文件（通常以 `.proto` 为扩展名）描述了服务的方法、输入参数和输出结果。通过编译服务定义文件，可以生成对应编程语言的代码和数据结构，以便在客户端和服务端实现 RPC 调用。

d. HTTP/2：gRPC 使用 HTTP/2 作为传输协议。HTTP/2 是一个现代的、二进制协议，提供了多路复用（multiplexing）、头部压缩（header compression）和流量控制（flow control）等功能。这使得 gRPC 具有高性能、低延迟和高吞吐量的特点。

e. Protocol Buffers：gRPC 使用 Protocol Buffers 作为数据序列化格式。Protocol Buffers 是一种高效、紧凑的二进制格式，可以在不同编程语言和平台之间进行交换。与 JSON、XML 等文本格式相比，Protocol Buffers 具有更小的数据大小和更快的编解码速度。

f. 双向流（Bidirectional Streaming）：gRPC 支持双向流，即客户端和服务端可以在一个连接上同时发送和接收数据。这使得 gRPC 能够实现实时通信、长连接等应用场景。

总之，gRPC 的网络架构包括服务端、客户端、服务定义、HTTP/2、Protocol Buffers 和双向流等组件。这些组件共同构成了一个高性能、灵活的 RPC 框架，适用于微服务、云原生应用和跨平台通信等场景。

13.tfsavemodel api 内部如何把静态图加载起来的

a. 解析计算图：在加载 SavedModel 文件时，TensorFlow 会解析 `saved_model.pb` 文件中的计算图。计算图是一个有向无环图（DAG），表示了模型中各个操作（Ops）之间的依赖关系。静态图是在模型构建阶段定义的，与动态图（Eager Execution）相比，静态图具有更好的性能和优化潜力。

b. 恢复权重：TensorFlow 会从 `variables` 目录中加载模型的权重，并将它们恢复到计算图的变量节点中。这样，在执行模型推理时，可以直接使用这些权重进行计算。

c. 使用函数签名（Function Signatures）：SavedModel 包含了一个或多个函数签名，它们定义了模型的输入和输出。在加载 SavedModel 时，TensorFlow 会创建一个包装函数（Wrapper Function），用于调用这些函数签名。你可以使用这个包装函数来执行模型推

超全面经：

https://wangpengcheng.github.io/2019/12/11/interview_c_plusplus/#14c%E6%A8%A1%E6%9D%BF%E6%98%AF%E4%BB%80%E4%B9%88%E5%BA%95%E5%B1%82%E6%80%8E%E4%B9%88%E5%AE%9E%E7%8E%B0%E7%9A%84