

C++八股文

更新记录

2022 年 8 月 4 日：在彭博 [C++.pdf](#) 的基础上完成初版

2022 年 8 月 7 日：加入 C++ 的引用部分

2022 年 8 月 12 日：加入尾递归优化的部分；加入 `push_back` 和 `emplace_back` 更细节的说明

2022 年 8 月 18 日：加入了 `sizeof` 关键词

2022 年 8 月 21 日：加入右左法则的部分

2022 年 10 月 18 日：更新了 `volatile` 的部分，需要注意 `store buffer`、`invalidate queue` 导致的弱一致性问题

右左法则判断变量声明

📖 右左法则

📖 有 `const int **pp2`，那么 `const` 的意思是不能更改哪个数据？

右左法则不是 C 标准里面的内容，它是从 C 标准的声明规定中归纳出来的方法。C 标准的声明规则，是用来解决如何创建声明的，而右左法则是用来解决如何辨识一个声明的。

究竟右左法则的规律是什么呢？顾名思义，从声明的右边看到左边，下面是左右法则的专业解释：

右左法则：首先从最里面的圆括号（应该是未定义的标识符）看起，然后往右看，再往左看。每当遇到圆括号时，就应该掉转阅读方向。一旦解析完圆括号里面所有的东西，就跳出圆括号。重复这个过程直到整个声明解析完毕。

尾递归优化

📖 尾递归为啥能优化？

尾递归：

若函数在尾位置调用自身（或是一个尾调用本身的其他函数等等），则称这种情况为尾递归。尾递归也是递归的一种特殊情形。尾递归是一种特殊的尾调用，即在尾部直接调用自身的递归函数。对尾递归的优化也是关注尾调用的主要原因。尾调用不一定是递归调用，但是尾递归特别有用，也比较容易实现。

特点：

尾递归在普通尾调用的基础上，多出了 2 个特征：

1. 在尾部调用的是函数自身 (Self-called) ;
2. 可通过优化, 使得计算仅占用常量栈空间 (Stack Space)。

— 维基百科尾调用词条

尾递归为什么可以优化

上面说了, 函数栈的目的是啥? 是保持入口环境。那么在什么情况下可以把这个入口环境给优化掉? 答案不言而喻, 入口环境没意义的情况下为啥要保持入口环境? 尾递归, 就恰好是这种情况。

因为尾递归的情况下, 我们保持这个函数的入口环境没意义, 所以我们就可以把这个函数的调用栈给优化掉。

执行的四个过程

- 1、预处理：条件编译，头文件包含，宏替换的处理，生成.i 文件。 -E
- 2、编译：将预处理后的文件进行词法分析、语法分析、语义分析然后转换成汇编语言，生成.s 文件 -S
- 3、汇编：汇编变为目标代码(机器代码) 生成.o 的文件 -c
- 4、链接：连接目标代码，生成可执行程序，链接是处理可重定位文件，把它们的各种符号引用和符号 定义转换为可执行文件中的合适信息(般是虚拟内存地址) 的过程。链接 分为静态链接(.a) 和动态链接(.so)， -o

extern

extern 可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。

extern "C"

extern "C" 的主要作用就是为了能够正确实现 C++ 代码调用其他 C 语言代码。加上 extern "C" 后, 会 指示编译器这部分代码按 C 语言的进行编译, 而不是 C++ 的。由于 C++ 支持函数重载, 因此编译器编译函数的过程中会将函数的参数类型也加到编译后的代码中, 而不仅仅是函数名; 而 C 语言并不支持 函数重载, 因此编译 C 语言代码的函数时不会带上函数的参数类型, 般只包括函数名。

static 关键字

- 1、全局静态变量：在全局变量前加上关键字 static，在静态存储区，整个程序运行期间 直存在。未初始化会被自动初始化为 0，作用域是声明它的源文件访问。
 - 2、局部静态变量：在代码块内部的声明的变量前面加上 static，由自动变量变为静态变量。作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域结束。但是当局部静态变量离开作用域后，并没有销毁，而是仍然驻留在内存当中，只不过不能再对它进行访问，直到该函数再次被调用，并且值不变。
 - 3、静态函数和全局静态变量类似只能在声明它的源文件类访问。
 - 4、类的静态成员：类的静态成员是类的所有对象中共享的成员，而不是某个对象的成员。对多个对象来说，静态数据成员只存储 处，供所有对象共用。
 - 5、类的静态函数：静态成员函数和静态数据成员 样，它们都属于类的静态成员，它们都不是对象成员。
- 全局变量、静态全局变量和静态局部变量 都存放在内存的 静态存储区域 (Bss、数据段、代码段)，局部变量 存放在 内存的栈区

c++中的四种类型转换

1. const_cast：用于去除变量的 const 属性
2. static_cast：用于基本数据类型之间的转换。用于基类和派生类之间指针或者引用的转换，进行派生类转基类是安全的，进行基类转派生类时，由于没有动态类型检查，是不安全的。
3. dynamic_cast：在进行基类转派生类的会对运行期类型的信息进行检查。(无法转换指针返回 NULL，引用返回异常) (依赖于 RTTI 机制)
4. reinterpret_cast：重新解释类型，用在任意指针 (或引用) 类型之间的转换，但没有进行二进制的转换

RTTI 机制

C++中的RTTI机制解析

概念：RTTI(Run Time Type Identification)即通过运行时类型识别，程序能够使用基类的指针或引用来检查着这些指针或引用所指的对象的实际派生类型。

产生的原因：为什么会出现 RTTI 这一机制，这和 C++语言本身有关系。和很多其他语言一样，C++是一种静态类型语言。其数据类型是在编译期就确定的，不能在运行时更改。然而由于面向对象程序设计中多态性的要求，C++中的指针或引用(Reference)本身的类型，可能与它实际代表(指向或引用)的类型并不一致。有时我们需要将一个多态指针转换为其实指向对象的类型，就需要知道运行时的类型信息，这就产生了运行时类型识别的要求。和 Java 相比，C++要想获得运行时类型信息，只能通过 RTTI 机制，并且 C++最终生成的代码是直接和机器

相关的。

主要操作符：typeid 操作符，返回指针和引用所指的 actual 类型；dynamic_cast 操作符，将基类类型的指针或引用安全地转换为其派生类类型的指针或引用。

通过运行时类型识别，程序能够使用基类的指针或引用来检查着这些指针或引用所指的对象的实际派生类型。

C++ 的多态性（运行时）是由虚函数实现的，对于多态性的对象，无法在程序编译阶段确定对象的类型。当类中含有虚函数时，其基类的指针就可以指向任何派生类的对象，这时就有可能不知道基类指针到底指向的是哪个对象的情况，类型的确定要在运行时利用运行时类型标识做出。为了获得一个对象的类型可以使用 typeid 函数，返回指针和引用所指的 actual 类型。

dynamic_cast 操作符，将基类类型的指针或引用安全地转换为其派生类类型的指针或引用。

C++ 中的智能指针

RAII (Resource Acquisition is Initialization)，也称为“资源获取就是初始化”，避免内存泄漏的方法，使用对象时先构造对象，最后析构对象。

智能指针的目的：

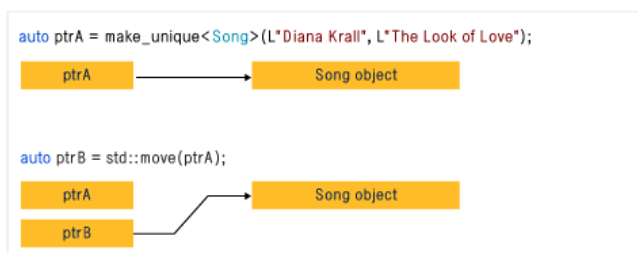
- 用来管理动态分配的内存的，它能够动态地分配资源且能够在适当的时候释放掉曾经动态分配的内存。

智能指针的两个原则：

- 智能指针本身不能是动态分配的，否则它自身有不被释放的风险，进而可能导致它所管理对象不能正确地被释放；
- 在栈上分配智能指针，让它指向堆上动态分配的对象，这样就能保证智能指针所管理的对象能够合理地被释放。

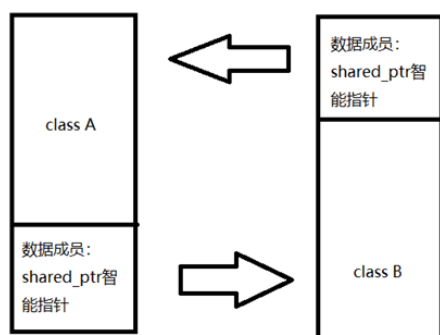
几种智能指针：

- auto_ptr：在 C++ 98 中引入的，在 C++ 17 中被移除掉
- shared_ptr：实现共享拥有的概念，多个智能指针可以指向相同的对象，它使用计数机制来表明资源被几个指针共享，该资源会在最后一个引用被销毁的时候释放。
- unique_ptr (参考)：实现独占拥有的概念，保证同一时间只能有一个智能指针可以指向该对象。无法拷贝，值只能转移



- weak_ptr：weak_ptr 是一种不控制对象生命周期的智能指针，它指向一个 shared_ptr 管理的对象，weak_ptr 只是提供了对管理对象的一个访问段，他不会增加

shared_ptr 的引用计数。可以用来解决 [shared_ptr 中的相互引用时的死锁问题](#)。通常不单独使用（没有实际用处），只能和 shared_ptr 类型指针搭配使用。甚至于，我们可以将 weak_ptr 类型指针视为 shared_ptr 指针的一种辅助工具，借助 weak_ptr 类型指针，我们可以获取 shared_ptr 指针的一些状态信息，比如有多少指向相同的 shared_ptr 指针、shared_ptr 指针指向的堆内存是否已经被释放等等。除此之外，weak_ptr<T> 模板类中没有重载 * 和 -> 运算符，这也就意味着，weak_ptr 类型指针只能访问所指的堆内存，而无法修改它。



https://blog.csdn.net/weixin_45590473

new 和 malloc 区别

本质不同：

- new/delete 是 C++ 的关键字，而 malloc/free 是库函数，new/delete 可以进行重载，而 malloc/free 不行；

行为不同（后者被前者调用）：

- new 会先调用 operator new 函数，申请足够的内存（通常底层使用 malloc 实现）。然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete 先调用析构函数，然后调用 operator delete 函数释放内存（通常底层使用 free 实现）。

返回类型不同：

- new 操作符成功时，返回的是对象类型的指针。而 malloc 内存分配成功则是返回 void*，需要通过强制类型转换将 void* 指针转换成需要的类型。

malloc 之下的系统调用：

- 当开辟的空间小于 128K 时，调用 brk() 函数，malloc 的底层实现是系统调用函数 brk()，其主要移动指针 _enddata(此时的 _enddata 指的是 Linux 地址空间中堆段的末尾地址，不是数据段的末尾地址)
- 当开辟的空间大于 128K 时，mmap() 系统调用函数来在虚拟地址空间中（堆和栈中间，称为“文件映射区域”的地方）找一块空间来开辟。

c++中的多态

- 静态多态：重载。静态多态是编译器在编译期间完成的，编译器会根据实参类型来选择调用合适的函数，如果有合适的函数可以调用就调，没有的话就会发出警告或者报错
- 动态多态：动态多态是用虚函数机制实现的，在运行期间动态绑定。虚函数的实现：在有虚函数的类中，类中有一个指向虚函数表的指针，表中放了虚函数的地址。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。

虚函数表实现运行时多态

虚函数表是一个类的虚函数的地址表，每个对象在创建时，都会有一个指针指向该类虚函数表，每个类的虚函数表，按照函数声明的顺序，会将函数地址存在虚函数表中，当子类对象重写父类的虚函数的时候，父类的虚函数表中对应的位置会被子类的虚函数地址覆盖。

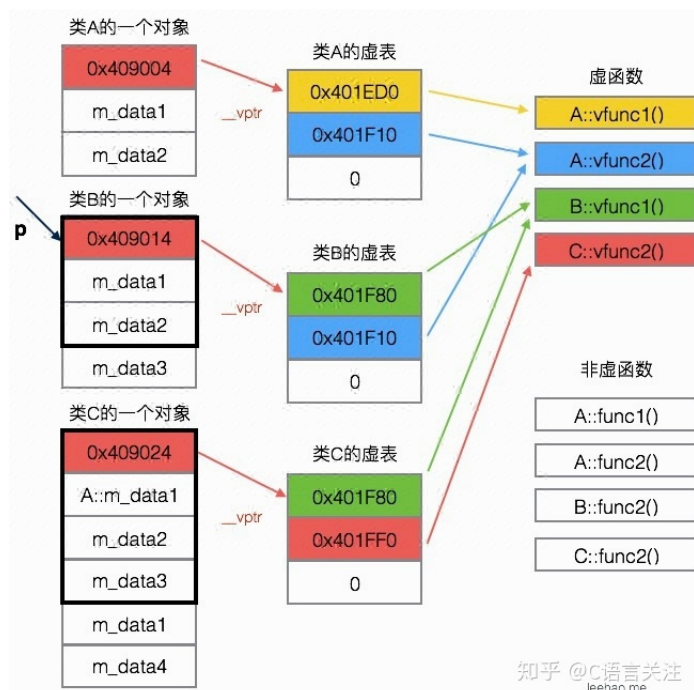
C++ 虚函数表剖析

C++虚函数表剖析

虚表是属于类的，而不是属于某个具体的对象，一个类只需要一个虚表即可。同一个类的所有对象都使用同一个虚表。

一个静态例子：

类 A 是基类，类 B 继承类 A，类 C 又继承类 B。类 A，类 B，类 C，其对象模型如下图所示。



- 由于这三个类都有虚函数，故编译器为每个类都创建了一个虚表，即类 A 的虚表 (A vtbl)，类 B 的虚表 (B vtbl)，类 C 的虚表 (C vtbl)。类 A，类 B，类 C 的对象都拥有一

个虚表指针，*__vptr，用来指向自己所属类的虚表。

- 类 A 包括两个虚函数，故 A vtbl 包含两个指针，分别指向 A::vfunc1()和 A::vfunc2()。
- 类 B 继承于类 A，故类 B 可以调用类 A 的函数，但由于类 B 重写了 B::vfunc1()函数，故 B vtbl 的两个指针分别指向 B::vfunc1()和 A::vfunc2()。
- 类 C 继承于类 B，故类 C 可以调用类 B 的函数，但由于类 C 重写了 C::vfunc2()函数，故 C vtbl 的两个指针分别指向 B::vfunc1()（指向继承的最近的一个类的函数）和 C::vfunc2()。

虽然图看起来有点复杂，但是只要抓住“对象的虚表指针用来指向自己所属类的虚表，虚表中的指针会指向其继承的最近的一个类的虚函数”这个特点，便可以快速将这几个类的对象模型在自己的脑海中描绘出来。

非虚函数的调用不用经过虚表，故不需要虚表中的指针指向这些函数。

一个动态调用的例子：

- 程序在执行 p->vfunc1()时，会发现 p 是个指针，且调用的函数是虚函数，接下来便会进行以下的步骤。
- 首先，根据虚表指针 p->__vptr 来访问对象 bObject 对应的虚表。虽然指针 p 是基类 A*类型，但是 *__vptr 也是基类的一部分，所以可以通过 p->__vptr 可以访问到对象对应的虚表。
- 然后，在虚表中查找所调用的函数对应的条目。由于虚表在编译阶段就可以构造出来了，所以可以根据所调用的函数定位到虚表中的对应条目。对于 p->vfunc1()的调用，B vtbl 的第一项即是 vfunc1 对应的条目。
- 最后，根据虚表中找到的函数指针，调用函数。从图 3 可以看到，B vtbl 的第一项指向 B::vfunc1()，所以 p->vfunc1()实质会调用 B::vfunc1()函数。

构造函数不能是虚函数的原因

构造一个对象时，必须知道对象实际类型，而虚函数是在运行期间确定实际类型的。而在构造一个对象时，由于对象还未构造成功，编译器就无法知道对象的实际类型，是该类本身，还是派生类，还是其他。

虚函数相应一个指向 vtable 虚函数表的指针，但是这个指向 vtable 的指针事实上是存储在对象的内存空间的。假设构造函数是虚的，就须要通过 vtable 来调用，但是对象还没有实例化，也就是内存空间还没有，找不到虚表。所以构造函数不能是虚函数。

75、(超重要)构造函数为什么不能为虚函数？析构造函数为什么要虚函数？

1. 从存储空间角度，虚函数相应一个指向 vtable 虚函数表的指针，这大家都知道，但是这个指向 vtable 的指针事实上是存储在对象的内存空间的。问题出来了，假设构造函数是虚的，就须要通过 vtable 来调用，但是对象还没有实例化，也就是内存空间还没有，怎么找 vtable 呢？所以构造函数不能是虚函数。

2. 从使用角度，虚函数主要用于在信息不全的情况下，能使重载的函数得到相应的调用。构造函数本身就是初始化实例，那使用虚函数也没有实际意义呀。所以构造函数没有必要是虚函数。虚函数的作用在于通过父类的指针或者引用来调用它的时候可以变成调用子类的那个成员函数。而构造函数是在创建对象时自己主动调用的，不可能通过父类的指针或者引用去调用，因此也就规定构造函数不能是虚函数。
3. 构造函数不须要是虚函数，也不同意是虚函数，由于创建一个对象时我们总是要明白指定对象的类型，虽然我们可能通过实验室的基类的指针或引用去访问它但析构却不一定，我们往往通过基类的指针来销毁对象。这时候假设析构函数不是虚函数，就不能正确识别对象类型从而不能正确调用析构函数。
4. 从实现上看，vbt1 在构造函数调用后才建立，因而构造函数不可能成为虚函数从实际含义上看，在调用构造函数时还不能确定对象的真实类型（由于子类会调父类的构造函数）；
5. 并且构造函数的作用是提供初始化，在对象生命期仅仅运行一次，不是对象的动态行为，也没有必要成为虚函数。

模板成员函数不可以是虚函数

编译器在处理类的定义的时候就能确定这个类的虚函数表的大小，如果允许有类的虚成员模板函数，那么就要求编译器提前知道程序中所有对该类的该虚成员模板函数的调用，而这是不可行的。

STL 的 allocator

STL allocator 将两个阶段操作区分开来：内存配置有 `alloc::allocate()` 负责，内存释放由 `alloc::deallocate()` 负责；对象构造由 `alloc::construct()` 负责，对象析构由 `alloc::destroy()` 负责。

同时为了提升内存管理的效率，减少申请小内存造成的内存碎片问题，SGI STL 采用了两级配置器，当分配的空间大小超过 128B 时，会使用第一级空间配置器；当分配的空间大小小于 128B 时，将使用第二级空间配置器。第一级空间配置器直接使用 `malloc()`、`realloc()`、`free()` 函数进行内存空间的分配和释放，而第二级空间配置器采用了内存池技术，通过空闲链表来管理内存。

Vector 和 list 的区别

vector 拥有一段连续的内存空间，支持高效的随机访问，但是对非尾节点进行删除和插入需要内存拷贝。

List 拥有的是多段不连续的内存空间，可以进行高效的插入和删除操作，但是随机访问效率较差。

迭代器

迭代器不是指针，是类模版，表现的像指针，本质是封装了原生指针，提供了比指针更高级的行为。迭代器的访问方式就是把不同集合类的访问逻辑抽象出来，使得不用暴露集合内部的结构而达到遍历集合的效果。

<https://www.zhihu.com/question/54047747/answer/137783330>

有利于不同容器选型变化时代码重用

resize 和 reserve 的区别

Resize 既分配了空间，也创建了对象，可以通过下标访问。Reserve 只是修改了容器的 capacity 大小，不修改 size 大小。

[vector中resize\(\)和reserve\(\)区别](#)

size 指容器当前拥有的元素个数；

而 capacity 则指容器在必须分配新存储空间之前可以存储的元素总数。

resize()函数和容器的 size 息息相关。调用 resize(n)后，容器的 size 即为 n。至于是否影响 capacity，取决于调整后的容器的 size 是否大于 capacity。

reserve()函数和容器的 capacity 息息相关。调用 reserve(n)后，若容器的 capacity < n，则重新分配内存空间，从而使得 capacity 等于 n。如果 capacity ≥ n 呢？capacity 无变化。

从两个函数的用途可以发现，容器调用 resize()函数后，所有的空间都已经初始化了，所以可以直接访问。

而 reserve()函数预分配出的空间没有被初始化，所以不可访问。

c++11 新特性

- auto 关键字：编译器可以根据初始值自动推导出类型。但是不能用于函数传参以及数组类型的推导。可以把 auto 关键字看成是一个变量定义中的数据类型占位符，它占据了原来应该是具体数据类型的位置。而在编译的时候，编译器会根据这个变量的初始值，推断出这个变量应有的具体数据类型，然后替换掉 auto 关键字，就成为一个普通的带有具体数据类型的变量定义了。用 auto 关键字定义变量时，变量必须有初始值。
- 右值引用：消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率
- atomic 原子操作用于多线程资源互斥操作
- Lambda 表达式 可以方便的定义和创建匿名函数。[capture list] (params list) mutable

exception-> return type { function body }

- [] 不截取任何变量
- [&] 截取外部作用域中所有变量，并作为引用在函数体中使用
- [=] 截取外部作用域中所有变量，并拷贝一份在函数体中使用
- [=, &foo] 截取外部作用域中所有变量，并拷贝一份在函数体中使用，但是对 foo 变量使用引用
- [bar] 截取 bar 变量并且拷贝一份在函数体中使用，同时不截取其他变量
- [this] 截取当前类中的 this 指针。如果已经使用了&或者=就默认添加此选项。

左值右值

c/c++左值和右值

例如：

- ++a 为将 a 进行自加，然后返回 a，a 本身就有内存地址，为一个左值，因此 (++a)++ 正确；
- 而 a++ 是后自增，在表达式里的值仍然为原来的 a，返回 a，然后 a 再被赋值为 a+1。而返回的值为一个临时变量，是一个右值，不能被改变，不存在内存地址。所以 (a++)++ 错误。

C++右值引用

拷贝构造函数和移动构造函数的调用关系也在这个链接中

↓被-初始化	non-const 左值	const 左值	non-const 右值	const 右值
non-const 左值	√	×	×	×
const 左值	√	√	√	√
non-const 右值引用	×	×		
const 右值引用	×	×		

一旦你初始化一个右值引用变量，该变量就成为了一个左值，可以被赋值

```
1  int x = 20; // 左值
2  int&& rx = x * 2; // x*2 的结果是一个右值，rx 延长其生命周期
3  int y = rx + 2; // 因此你可以重用它：42
4  rx = 100;      // 一旦你初始化一个右值引用变量，该变量就成为了一个左值，可以被赋值
```

万能引用与引用折叠

引用折叠

<https://www.zhihu.com/question/40346748/answer/86100193>

根本原因是因为 C++ 中禁止 reference to reference，所以编译器需要对四种情况(也就是 L2L, L2R, R2L, R2R)进行处理，将他们“折叠”(也可说是“坍缩”)成一种单一的 reference。

<https://www.zhihu.com/question/40346748/answer/88672920>

1. 所有右值引用折叠到右值引用上仍然是一个右值引用。(A&& && 变成 A&&)
2. 所有的其他引用类型之间的折叠都将变成左值引用。(A& & 变成 A&; A& && 变成 A&; A&& & 变成 A&)

万能引用

[C++ 的万能引用解析](#)

C++11 除了带来了右值引用以外，还引入了一种称为“万能引用”的语法；通过“万能引用”，对某型别的引用 T&&，既可以表达右值引用，也可以表达左值引用。

该语法有两种使用场景，最常见的一种是作为函数模板的形参，第二个场景则是 auto 声明

```
1  template<typename T>
2  void f(T&& param);
3  auto&& var2 = var1;
```

这两种情况都涉及到了型别的推导，也就是说，如果你虽然遇到了 T&& 的形式，但是不涉及型别推导，那么它只是一个右值引用

话说到这里，我们就可以更深入地理解万能引用，其实它就是满足了下面两个条件的语境中的右值引用：

1. 型别推导的过程会区别左值和右值。T 型别的左值推导结果为 T&，而 T 型别的右值推导结果为 T。
2. 会发生引用折叠。

[万能引用，引用折叠，移动构造函数，emplace_back 及其实现，完美转发及其实现](#)

Forwarding Reference 是 C++ 语言的规定的一个例外规则 (C++ Primer 原话)：

- T && 碰到右值 int &&，T 匹配成 int；
- T && 遇到左值 int，也能匹配，T 此时是 int &。
- T && 碰到左值 const int，T 匹配为 const int &。

- T &&碰到左值 `const int *` (指针类型), T 匹配为 `const int *&` (下略)
- T &&碰到左值 `const int * const` (指针类型), T 匹配为 `const int *const &` (下略)

move 和 forward

浅谈C++11中的move和forward

move 和 forward 都是 C++11 中引入的，它们是移动语义和完美转发实现的基石。

- move：不能移动任何东西，它唯一的功能是将一个左值强制转化为右值引用，继而可以通过右值引用使用该值，以用于移动语义从实现上讲，`std::move` 基本等同于一个类型转换：`static_cast<T&&>(lvalue)`
- forward：不转发任何东西，也是执行左值到右值的强制类型转换，只是仅在特定条件满足时才执行该转换

典型使用场景：某个函数模板取用了万能引用类型为形参，随后把它传递给了另一个函数

move 和移动语义

浅谈C++11中的move和forward

那么问题来了，既然 move 不移动而只是转换类型，为什么还把它命名为 move，而不是一个类似 `rvalue_cast` 之类的名字？

首先再重复一下，move 做的是强制类型转换，把左值转换为右值，不做移动

而右值是可以进行移动的，所以 move 一个对象，就是告诉编译器该对象具备可移动的条件，这样，move 表述了这样一个事实：该对象可以移动了。move = make it movable

在下面的情况下，移动语义并不会带来什么好处：

- 没有移动操作：待移动对象没有提供移动操作，这时移动请求就变成了复制请求
- 移动不是更快：待移动对象有移动操作，但并不比复制操作更快，如采用了小型字符串优化 (SSO) 的小 string (容器不超过 15 的字符串)，或者使用 array 容器而非 vector
- 移动不可用：在移动本可以发生的语境下，要求移动操作不可抛出异常，但该操作未加上 `noexcept` 声明
- 源对象是个左值：除极少数例外，只有右值可以作为移动对象的源

这个示例说明了几个问题：

- 针对常量对象执行移动操作将被偷偷地变换成复制操作，这可能会给性能调试带有困扰
- 如果想取得对某个对象执行移动操作的能力，不要将其声明为常量
- move 不权不实际移动任何东西，甚至不保证经过其强制类型转换后的对象具备可移动的能力

- 针对 move 的结果，唯一可确定的是，结果是个右值（但可能是常量右值）

forward 和完美转发

```
1  template <typename T>
2  T&& forward(typename std::remove_reference<T>::type& param)
3  {
4      return static_cast<T&&>(param);
5  }
```

总的来说相当于仅在传入万能引用的类型是右值的情况下，T 会被实例化为非引用类型，此时传入 forward 中，最后返回 T&&而不发生折叠。如果传入万能引用的类型是左值，T 会被实例化为左值引用类型，传到 forward 中发生引用折叠，返回 T&（= T& &&）类型。

 谈谈完美转发(Perfect Forwarding)：完美转发 = 引用折叠 + 万能引用 + std::forward

 [C++][标准库] 完美转发 = 引用折叠 + 万能引用 + std::forward

完美转发是一个比较简单，却又比较复杂的东西。

简单之处在于理解**动机**：C++为什么需要完美转发？

复杂之处在于理解**原理**：完美转发基于万能引用，引用折叠以及 std::forward 模板函数。

...

不难发现，本质问题在于，左值右值在函数调用时，都转化成了左值，使得函数转调用时无法判断左值和右值。

在 STL 中，随处可见这种问题。比如 C++11 引入的 `emplace_back`，它接受左值也接受右值作为参数，接着，它转调用了空间配置器的 `construct` 函数，而 `construct` 又转调用了 `placement new`，`placement new` 根据参数是左值还是右值，决定调用拷贝构造函数还是移动构造函数。

这里要保证从 `emplace_back` 到 `placement new`，参数的左值和右值属性保持不变。这其实不是一件简单的事情。

...

`std::forward` 不是独自运作的，在我的理解里，完美转发 = `std::forward` + 万能引用 + 引用折叠。三者合一才能实现完美转发的效果。

`std::forward` 的正确运作的前提，是引用折叠机制，为 T &&类型的万能引用中的模板参数 T 赋了一个恰到好处的值。我们用 T 去指明 `std::forward` 的模板参数，从而使得 `std::forward` 返回的是正确的类型

...

右值的概念其实很微妙，一旦某个右值，有了名字，也就在内存中有了位置，它就变成了 1 个左值。但它又是一个很有用的概念，允许程序员更加细粒度的处理对象拷贝时的内存分配问题，提高了对临时对象和不需要的对象的利用率，极大提高程序的效率。当然，也会引入更多的 bug。不过，这就是 C++ 的哲学，什么都允许你做，但出了问题，可别赖 C++ 这门语言。

...

完美转发基于万能引用，引用折叠以及 `std::forward` 模板函数。据我所知，STL 出现 `std::forward`，一定出现万能引用。其实这也很好理解，完美转发机制，是为了将左值和右值统一处理，节约代码量，而只有万能引用会出现同时接受左值和右值的情况，所以完美转发只存在于万能引用中。

 万能引用，引用折叠，移动构造函数，`emplace_back` 及其实现，完美转发及其实现

`push_back` 是 `vector` 的一个普通成员函数，有 2 个重载，分别接受左值和右值。

`emplace_back` 是 `vector` 的一个模板成员函数，没有重载，接受左值和右值。只有将 `emplace_back` 写成模板成员函数，它的参数才可以写成 `Args &&` 的形式，才可以激活万能引用，这样才既可以接受左值，又可以接受右值。

又因为 `emplace_back` 既可以接受左值，又可以接受右值，所以转调用其他函数时，要进行完美转发，见下文。

...

右值在传参的过程中变为左值

使用 `std::move` 会使得左值变为右值（因为 `T &&` 是万能引用，所以可能本来应该是左值）

我们希望的是维持左值或右值属性不变，完美转发可以做到这点

push_back 和 emplace_back

 c++ 中有什么 `push` 和 `insert` 可以但 `emplace` 做不到的操作么？

```
1  struct S {
2      int value;
3  };
4
5  std::vector<S> v;
6  v.push_back({0});    // 编译成功
7  v.emplace_back({0}); // 编译不过
```

`emplace_back` 是模板函数，对于 `{0}` 它无法确定是啥东东。`push_back` 不是模板函数，它会直接构造一个对应的对象拷过去。

C++ 移动(move) 右值引用和 forwad(完美转发)

`std::move` : 将对象的状态或者所有权从一个对象转移到另一个对象, 只是转移, 没有内存拷贝。移动语义可以将资源(堆、系统对象等)通过浅拷贝的方式从一个对象转移到另一个对象, 减少不必要的临时对象的创建、拷贝与销毁, 提高了 c++ 应用程序的性能。

`std::forward<T>(u)` 有两个参数: `T` 与 `u`。当 `T` 为左值引用类型时, `u` 将被转换为 `T` 类型的左值, 否则 `u` 将被转换为 `T` 类型右值

C++ 内存分配机制(高到低)

- 栈: 存放函数的局部变量、函数参数、返回地址等, 由编译器自动分配和释放。
- 堆: 由 `new` 分配的内存块, 一般一个 `new` 就要对应一个 `delete`。如果程序员没有释放掉, 那么在程序结束后, 操作系统会自动回收。
- 静态存储区: 全局变量和静态变量被分配到同一块内存中,
- 常量存储区: 里面存放的是常量, 不允许修改, 程序运行结束自动释放。包括字符串常量区和常变量区
- 代码区: 存放代码(如函数), 不允许修改但可以执行

目标文件分段

- 数据段 (`.data`): 初始化了的全局静态变量和局部静态变量
- 只读数据段 (`.rodata`): 只读数据, 一般是程序里面的只读变量 (`const`) 和字符串常量。可以将其映射成值得读内存, 保证了程序的安全性
- BSS 段 (`.bss`): 未初始化的全局变量和局部静态变量

c++ 内存对齐方式

1. 第一个数据成员放在 `offset` 为 0 的地方, 以后每个数据成员的对齐按照 `pragma pack` 指定的数值和这个数据成员自身长度中, 比较小的那个进行。
2. 在数据成员完成各自对齐之后, 类(结构或联合)本身也要进行对齐, 对齐将按照 `pragma pack` 指定的数值和结构(或联合)最大数据成员长度中, 比较小的那个进行。

pragma pack 编译宏

 [#pragma pack\(\)用法](#)

#pragma pack 的主要作用就是改变编译器的内存对齐方式，这个指令在网络报文的处理中有着重要的作用

- #pragma pack(n)：最基本的用法，其作用是改变编译器的对齐方式，不使用这条指令的情况下，编译器默认采取#pragma pack(8)也就是 8 字节的默认对齐方式，n 值可以取 (1 , 2 , 4 , 8 , 16) 中任意一值。
- #pragma pack(show)：显示当前内存对齐的字节数会在编译阶段提出一个警告，说明当前对齐字节数。
- #pragma pack(push)：会将当前的对齐字节数压入栈顶，并设置这个值为新的对齐字节数，就是说不会改变这个值
- #pragma pack(push, n)：会将当前的对齐字节数压入栈顶，并设置 n 为新的对齐字节数。
- #pragma pack(push, identifier [, n])：会在上面的操作基础上为这个对齐字节数附上一个标识符，这里注意这个标识符只能以 (\$、_、字母) 开始，标识符中可以有 (\$、_、字母、数字)，并且标识符不能是关键字(push，pop 可以作为标识符)。
- #pragma pack(pop)：会弹出栈顶对齐字节数，并设置其为新的内存对齐字节数
- #pragma pack(pop, n)：会弹出栈顶并直接丢弃，设置 n 为其新的内存对齐字节数。
- #pragma pack(pop, identifier [, n])较为复杂，编译器执行这条执行时会从栈顶向下顺序查找匹配的 identifier，找到 identifier 相同的这个数之后将从栈顶到 identifier，包括找到 identifier 全部 pop 弹出，若没有找到则不进行任何操作。

(全局范围)

```
#pragma pack(push, 4)
#pragma pack(push, 4)
#pragma pack(show)
#pragma pack(pop, 16)
#pragma pack(show)
#pragma pack(pop)
#pragma pack(show)
int main() {
    return 0;
}
```

0 个错误 3 个警告 0 个消息

	说明	文件	行	列	项目
1	warning C4810: 杂注 pack(show) 的值 == 4	源.cpp	3	1	Project1
2	warning C4810: 杂注 pack(show) 的值 == 16	源.cpp	5	1	Project1
3	warning C4810: 杂注 pack(show) 的值 == 8	源.cpp	7	1	Project1

内存对齐的作用

1. 平台原因(移植原因)：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。
2. 性能原因：经过内存对齐后，CPU 的内存访问速度大大提升。
 - a. CPU 把内存当成是一块一块的，块的大小可以是 2，4，8，16，甚至 32 字节大小，因此 CPU 在读取内存时是一块一块进行读取的。块大小称为 memory access granularity (粒度)。对齐后可以减少读取次数和拼接成本。[一文读懂内存对齐的规则和作用](#)



- b. 人为控制对齐粒度可以一定程度上优化缓存性能，避免两个无关的频繁修改变量被放在一个缓存行中发生乒乓效应

const 关键字

1. 防止被修饰的成员的内容被改变。
2. 修饰类的成员函数时，表示其为一个常函数，意味着成员函数将不能修改类成员变量的值。
3. 在函数声明时修饰参数，表示在函数访问时参数（包括指针和实参）的值不会发生变化。
4. 对于指针而言，可以指定指针本身为 const，也可以指定指针所指的数据为 const，const int *b = &a;或者 int* const b = &a;修饰的都是后面的值，分别代表*b 和 b 不能改变。

const 和 define 区别

- 编译器处理方式
 - define – 在预处理阶段进行替换
 - const – 在编译时确定其值
- 类型检查
 - define – 无类型，不进行类型安全检查，可能会产生意想不到的错误（比如著名的 NULL 问题，见 [C++笔记](#)）
 - const – 有数据类型，编译时会进行类型检查
- 内存空间
 - define – 不分配内存，给出的是立即数，有多少次使用就进行多少次替换，在内存中会有多个拷贝，消耗内存大
 - const – 在静态存储区中分配空间，在程序运行过程中内存中只有一个拷贝

inline 和宏

inline 作为函数定义的关键字，说明该函数是内联函数。内联函数会将代码块嵌入到每个调用该函数的地方。内联函数减少了函数的调用，使代码执行的效率提高，但是会增加目标代码的

大小，最终会使程序的代码段占有大量的内存，进而可能导致 d-cache miss 增加有损效率。

两者的区别

1. 内联函数在编译时展开，宏在预处理时展开；
2. 内联函数直接嵌入到目标代码中，宏是简单的做文本替换；
3. 内联函数有类型检测、语法判断等功能，宏没有；
4. inline 函数是函数，宏不是；
5. 宏定义时要注意书写（参数要括起来）否则容易出现歧义，内联函数不会产生歧义；

inline 相比宏定义有哪些优越性

1. inline 函数代码是被放到符号表中，使用时像宏一样展开，没有调用的开销，效率很高；
2. inline 函数是真正的函数，所以要进行一系列的数据类型检查；
3. inline 函数作为类的成员函数，可以使用类的保护成员及私有成员；

为什么不能把所有的函数写成 inline 函数？

内联函数以代码复杂为代价，它以省去函数调用的开销来提高执行效率。

- 一方面如果内联函数体内代码执行时间相比函数调用开销较大没有太大的意义；
- 另一方面每处内联函数的调用都要复制代码，消耗更多的内存空间，因此以下情况不宜使用内联函数：
 - 函数体内的代码比较长，将导致内存消耗代价；
 - 函数体内有循环，函数执行时间要比函数调用开销大；

STL sort 的策略

STL 的 sort 算法，数据量大时采用 QuickSort 快排算法。一旦分段后的数据量小于某个门槛，为避免 QuickSort 快排的递归调用带来过大的额外负荷，就改用 Insertion Sort 插入排序。如果递归层次过深，还会改用 HeapSort 堆排序。

volatile 的作用

☞ 既然CPU有缓存一致性协议（MESI），为什么JMM还需要volatile关键字？ - 码海的回...

☞ 请问，多个线程可以读一个变量，只有一个线程可以对这个变量进行写，到底要不要加...

8.2.3.5 Intra-Processor Forwarding Is Allowed

The memory-ordering model allows concurrent stores by two processors to be seen in different orders by those two processors; specifically, **each processor may perceive its own store occurring before that of the other**. This is illustrated by the following example:

Example 8-5. Intra-Processor Forwarding is Allowed

Processor 0	Processor 1
mov [_x], 1 mov r1, [_x] mov r2, [_y]	mov [_y], 1 mov r3, [_y] mov r4, [_x]
Initially x = y = 0 r2 = 0 and r4 = 0 is allowed	

The memory-ordering model imposes no constraints on the order in which the two stores appear to execute by the two processors. This fact allows processor 0 to see its store before seeing processor 1's, while processor 1 sees its store before seeing processor 0's. (Each processor is self consistent.) This allows r2 = 0 and r4 = 0.

In practice, the reordering in this example can arise **as a result of store-buffer forwarding**. While a store is temporarily held in a processor's store buffer, it can satisfy the processor's own loads but is not visible to (and cannot satisfy) loads by other processors.

store buffer

如果一个基本变量被 volatile 修饰，编译器将不会把它保存到寄存器中，而是每次都去访问内存中实际保存该变量的位置上。

C/C++ 中 volatile 关键字详解

通常用于建立语言级别的 memory barrier。volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。

注意：

1. 可以把一个非 volatile int 赋给 volatile int，但是不能把非 volatile 对象赋给一个 volatile 对象。
2. 除了基本类型外，对用户定义类型也可以用 volatile 类型进行修饰。
3. C++中一个有 volatile 标识符的类只能访问它接口的子集，一个由类的实现者控制的子集。用户只能用 const_cast 来获得对类型接口的完全访问。
4. 此外，volatile 向 const 一样会从类传递到它的成员。

需要用到 volatile 的地方

其实不只是内嵌汇编操纵栈"这种方式属于编译无法识别的变量改变，另外更多的可能是多线程并发访问共享变量时，一个线程改变了变量的值，怎样让改变后的值对其它线程 visible。一般说来，volatile 用在如下的几个地方：

1. 中断服务程序中修改的供其它程序检测的变量需要加 volatile；
2. 多任务环境下各任务间共享的标志应该加 volatile；（否则会产生短暂的不一致性，见[本](#)

文)

3. 存储器映射的硬件寄存器通常也要加 volatile 说明，因为每次对它的读写都可能由不同意义；

volatile 指针

和 const 修饰词类似，const 有常量指针和指针常量的说法，volatile 也有相应的概念

volatile 关键字不能保证线程安全

当一个变量定义为 volatile 后，该变量对所有线程均可。即当一条线程修改了这个变量的值，新值对于其他线程可以是立即得知的。但是这并不意味着 volatile 变量在并发下是线程安全的。这是因为 volatile 只保证当前线程在读取这个变量时，变量的值与其他所有线程一致。当前线程把读取到的致的（在读的时候致）值压入栈顶进行计算时，由于该计算可能不具备原子性。在执行该计算的过程中，变量可能被其他线程改变。而这个改变并不会影响当前线程已压入栈顶的数据，这时栈顶的数据就成了过期的数据，从而发生线程安全问题。

C++的引用和指针

[C++中“引用”的底层实现原理详解](#)

引用的本质就是所引用对象的地址，但是一些操作被包装了

引用占用内存吗

通过上面的分析，我们得出了引用本身存放的是引用对象的地址，通俗点理解就是引用就是通过指针来实现的，所以，应用所占的内存大小就是指针的大小。

引用的地址

在最开始，我们写过一段代码来测试引用的地址，发现引用的地址和变量的地址是一样的。但是，在后面对引用的底层分析后发现，它本身又存放的是变量的地址，即引用的值是地址，那么这不是很冲突吗？

事实上，b 的地址我们没法通过 &b 获得，因为编译器会将 &b 解释为：&(*b) = &x，所以 &b 将得到 &x。也验证了对所有的 b 的操作，和对 x 的操作等同。

为什么要设计引用

[C++语言的历史和演化\(读书笔记\)-运算符重载催生了引用](#)

很多人要求 C++ 应该有运算符重载的能力，因为这可以让代码看起来很舒服。

比如：Matrix a,b,c; a = b * c;

但是 C++ 是沿用 C 的函数参数默认传值语义的，上面的代码本质上是下面的函数：


```
Matrix operator * (Matrix lhs, Matrix rhs);
```

但是上面的实现会发生拷贝，这对复杂对象来说是不可接受的，那就是要传指针：

```
Matrix operator (Matrix * lhs, Matrix * rhs);
```

这样以来乘法就会写成下面这样：

```
Matrix a,b,c; a = &b * &c;
```

上面这样一写的话，看起来就很丑了，没人会接受，太麻烦，重载的目标就是简洁优雅，整出这么个玩意儿出来，实在说不过去。何况上面的写法在 C 里面已经有意义了。那就只能让 C++ 支持引用了：

```
Matrix operator * (const Matrix & lhs, const Matrix & rhs);
```

有了上面的引用语义，一开始的 `a = b * c`；就变得又优雅，又没有成本了。

有了引用，将引用作为返回值也是很重要的，尤其是像 `string` 这种类型的下标运算成员函数：

```
char& operator [] (int index);
```

```
string s;
```

```
s[i] = 'c';//这里的赋值变得简洁明了
```

堆和栈的比较

- 申请方式：栈是系统自动分配，堆是程序员主动申请。
- 申请后系统响应：
 - 分配栈空间，如果剩余空间大于申请空间则分配成功，否则分配失败栈溢出；
 - 申请堆空间，堆在内存中呈现的方式类似于链表（记录空闲地址空间的链表），在链表上寻找第一个大于申请空间的节点分配给程序，将该节点从链表中删除，大多数系统中该块空间的地址存放的是本次分配空间的大小，便于释放，将该块空间上的剩余空间再次连接在空闲链表上。
 - 栈在内存中是连续的块空间（向低地址扩展）最大容量是系统预定好的，堆在内存中的空间（向高地址扩展）是不连续的。
- 申请效率：栈是有系统自动分配，申请效率高，但程序员无法控制；堆是由程序员主动申请，效率低，使用起来方便但是容易产生碎片。
- 存放的内容：栈中存放的是局部变量，函数的参数；堆中存放的内容由程序员控制。

C++ 中内存泄漏的几种情况

1. `new` 和 `delete` 函数调用不匹配
2. 释放对象数组时在 `delete` 中没有使用方括号
3. 没有将基类的析构函数定义为虚函数

内存问题

1. 动态内存泄露；
2. 资源泄露，比如文件指针不关闭；
3. 动态内存越界；
4. 数组内存越界；
5. 动态内存 double free；
6. 使用野指针，即未初始化的指针；
7. 释放野指针，即未初始化的指针。

C++成员初始化列表

初始化类的成员有两种方式：一是使用初始化列表，二是在构造函数体内进行赋值操作。使用初始化列表主要是基于性能问题，对于内置类型，如 int, float 等，使用初始化类表和在构造函数体内初始化差别不是很大，但是对于类类型来说，最好使用初始化列表，为什么呢？由上面的测试可知，使用初始化列表少了一次调用默认构造函数的过程，这对于数据密集型的类来说，是非常高效的。

除了性能问题之外，有些场景初始化列表是不可或缺的，以下几种情况时必须使用初始化列表：

- 常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面
- 引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面
- 没有默认构造函数的 class type，因为使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化。
- 当调用一个基类的构造函数时

成员是按照他们在类中出现的顺序进行初始化的，而不是按照他们在初始化列表出现的顺序初始化的

C++构造函数逻辑执行顺序

1. 创建派生类的对象，基类的构造函数优先被调用，（基类的构造函数也优先于派生类里的成员属性类）；
2. 如果类里面有成员属性类，成员属性类的构造函数优先被调用；(成员属性类的构造函数也优先于该类本身的构造函数)
3. 基类构造函数如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序而不是它们在成员初始化表中的顺序；

4. 成员类对象构造函数如果有多个成员类对象，则构造函数的调用顺序是对象在类中被声明的顺序而不是它们出现在成员初始化表中的顺序；
5. 派生类构造函数，作为一般规则，派生类构造函数应该不能直接向一个基类数据成员赋值而是把值传递给适当的基类构造函数，否则两个类的实现变成紧耦合的（tightly coupled）将更加难于正确地修改或扩展基类的实现。（基类设计者的责任是提供一组适当的基类构造函数）

用 include 引用头文件时，双引号和尖括号的区别：

1. 双引号：引用非标准库的头文件，编译器 先在程序源文件所在目录查找，如果未找到，则去系统默认目录查找，通常用于引用用户自定义的头文件。
2. 尖括号：只在系统默认目录（在 Linux 系统中通常为/usr/include 目录）或者尖括号内的路径查找，通常用于引用标准库中自带的头文件。

静态链接和动态链接

静态链接：静态库是在编译期完成的，程序在运行时对函数库再无关系，移植方便。但是浪费空间和资源，因为所有相关的目标文件与相关的函数库都会被链接成一个可执行文件。

动态链接：动态库把对一些库函数的链接推迟到程序运行的时期，可以实现进程之间的资源共享。

二者的优缺点：

静态链接：浪费空间，每个可执行程序都会有目标文件的一个副本，这样如果目标文件进行了更新操作，就需要重新进行编译链接生成可执行程序（更新困难）；优点就是执行的时候运行速度快，因为可执行程序具备了程序运行的所有内容（可移植性强）。

动态链接：节省内存、更新方便，但是动态链接是在程序运行时，每次执行都需要链接，相比静态链接会有一定的性能损失。

valgrind

Valgrind 是个开源的工具，功能很多。例如检查内存泄漏 工具---memcheck。常用命令
valgrind --tool=memcheck --leak-check=full ./test

Memcheck 可以检测到内存问题未释放内存的使用：

1. 对释放后内存的读/写

2. 对已分配内存块尾部的读/ 写
3. 内存泄露 (使用 malloc 没使用 free)
4. 不匹配的使用 malloc/new/new[] 和 free/delete/delete[] (malloc 和 delete 一起使用)
5. 重复释放内存

AddressSanitizer

valgrind 的性能开销太大了 (x20), 所以后来我用谷歌的 sanitizers , 开销更小 (x2)

构造函数如何做到线程安全

对象构造要做到线程安全, 唯一的要求是在构造期间不要泄露 this 指针, 因为在构造函数执行期间对象还没有完成初始化, 如果 this 被泄露 (escape) 给了其他对象 (其自身创建的子对象除外), 那么别的线程有可能访问这个半成品对象, 这会造成难以预料的后果。

析构函数如何做到线程安全

(shared_ptr 和内置类型 一样) 用智能指针: 一个 shared_ptr 实体可被多个线程同时读取;

- 两个的 shared_ptr 实体可以被两个线程同时写入, “析构” 算写操作;
- 如果要从多个线程读写同一个 shared_ptr 对象, 那么需要加锁。

muduo之当析构函数遇见线程安全

作为 class 数据成员的 mutex_ 只能同步本 class 的其他数据成员的读和写, 不能保护安全的析构。

解决方案的核心要点是: 析构过程不需要保护, 所有人都用不到的东西一定是垃圾, 垃圾自动回收的原理是“引用计数为 0 时, 该对象就是垃圾, 需要销毁”, 引出了带引用计数的智能指针。

既然通过 weak_ptr 的 expired 函数可以探查对象的生死, 那么 Observer 的竞态问题就很容易解决, 只要让 Observer 保存 weak_ptr<Observer> 即可。

两种单例模式

为什么单例模式要起名叫饿汉式和懒汉式? - fhyPayaso的回答 - 知乎

不管程序是否需要这个对象的实例, 总是在类加载的时候就先创建好实例, 理解起来就像不管一个人想不想吃东西都把吃的先买好, 如同饿怕了一样。

如果一个对象使用频率不高, 占用内存还特别大, 明显就不合适用饿汉式了, 这时就需要一种

懒加载的思想，当程序需要这个实例的时候才去创建对象，就如同一个人懒的饿到不行了才去吃东西

🔗 懒汉模式和饿汉模式的区别

懒汉模式：在类加载的时候不被初始化。

饿汉模式：在类加载时就完成了初始化，但是加载比较慢，获取对象比较快。

饿汉模式是线程安全的，在类创建好一个静态对象提供给系统使用，懒汉模式在创建对象时不加上 synchronized，会导致对象的访问不是线程安全的。

饿汉模式不需要用锁，就可以实现线程安全。

饿汉模式虽好，但其存在隐藏的问题，在于非静态对象（函数外的 static 对象）在不同编译单元中的初始化顺序是未定义的。如果在初始化完成之前调用 getInstance() 方法会返回 一个未定义的实例。

IO 复用模型

说明 1

🔗 [bilibili]【并发】IO多路复用select/poll/epoll介绍

服务器会维持多个客户链接。如果使用多线程，可能会导致线程太多，带来上下文切换的开销。单线程用户态轮询效率低

select

1980 年就有了

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(2000);
addr.sin_addr.s_addr = INADDR_ANY;
bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));
listen(sockfd, 5);

for (i=0; i<5; i++)
{
    memset(&client, 0, sizeof(client));
    addrlen = sizeof(client);
    fds[i] = accept(sockfd, (struct sockaddr*)&client, &addrlen);
    if (fds[i] > max)
        max = fds[i];
}

while(1){
    FD_ZERO(&rset);
    for (i = 0; i < 5; i++) {
        FD_SET(fds[i], &rset);
    }

    puts("round again");
    select(max+1, &rset, NULL, NULL, NULL);

    for (i=0; i<5; i++) {
        if (FD_ISSET(fds[i], &rset)){
            memset(buffer, 0, MAXBUF);
            read(fds[i], buffer, MAXBUF);
            puts(buffer);
        }
    }
}
```

fds
max

① 1024 bitmap
② FDset 不可重用
③ 用↔内开销
④ 0(n)再次遍历

用户态
vset
00000
内核态
00000

有数据: ① FD置位 (数据来的) ② select返回

bitmap 1024
0110010101000...

读处理

过程：

- 服务器的每一个允许的用户连接会被定义为一个 fd（文件描述符）

- fd 会组成一个数组，同时记录数组中的最大值 max。max 的目的是避免 select 过程中对 bitmap 过多的检查，约定一个有效区间（比如 max=10 的时候检查到第 10 位就可以了）
- 利用 FD_SET 接口将 fd 数组转化为一个 1024 位的 bitmap（rset），调用 select 接口
 - 将 rset 拷贝到内核态，让内核来检查文件（网络传输）的有效数据情况。因为用户检查也需要陷入内核态，直接让内核做可以更高效
 - 程序会阻塞在 select 中，直到有数据就绪
 - 检查到就绪数据后，会将对应的 FD 置位，并返回。此时的 FD 指的是 rset 而不是 fd 数组。如果本 fd 没有消息，select 会主动把 rset 的那一位变 0，如果有数据就维持 1。
 - 检查 rset 中的置位情况，并相应处理

优点：内核轮询，快

缺点：

- bitmap 默认大小 1024，可调，但存在上限
- FD_SET 产生的 rset 不可重用，因为会被内核修改。每次都要重新清零+初始化
- rset 从用户态拷贝到内核态存在开销（虽然比每次用户自己检查要少）
- select 返回后，虽然知道一定有数据就绪，但是不能直接指导是哪一个，需要再次 O(n) 遍历

poll

同样把数组信息拷贝到内核，让内核代为轮询。但是比起 select 中使用的 bitmap，这次使用了特制的 pollfd 数据结构，一切的优化围绕着这个数据结构展开

```

1  struct pollfd {
2      int fd;
3      short events; // 在意的事件，由宏描述 POLLIN / POLLOUT
4      short revents; // 对 events 的回馈
5  }
```



```

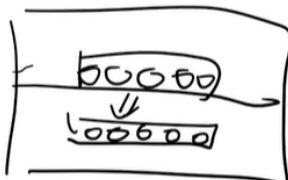
for (i=0; i<5; i++)
{
    memset(&client, 0, sizeof (client));
    addrlen = sizeof(client);
    pollfds[i].fd = accept(sockfd, (struct sockaddr*)&client, &addrlen);
    pollfds[i].events = POLLIN;
}
sleep(1);
while(1){
    puts("round again");
    poll(pollfds, 5, 50000);
    for(i=0; i<5; i++) {
        if (pollfds[i].revents & POLLIN){
            pollfds[i].revents = 0;
            memset(buffer, 0, MAXBUF);
            read(pollfds[i].fd, buffer, MAXBUF);
            puts(buffer);
        }
    }
}

```

```

struct pollfd {
    int fd;
    short events;
    short revents;
};

```



有数据口. pollfd. revents 置位.

② poll 返回.

过程：

- socket 链接生成文件描述符，组成数组 pollfds
- 阻塞调用 poll 接口（过程和 select 一样）
 - 有数据后对 pollfd.revents 置位（确保可重用性），并返回
- 检查 pollfds 中的 revents，并将其归零

相比于 select 的改进：

- 避免了数据结构对并发度的限制
- 增加了重用性减少拷贝（select 使用的 fdset，还是 poll 使用的 pollfds，用户态和内核态切换时都要用到 copy_from_user 和 copy_to_user，无论是，从用户态到内核态一次拷贝，从内核态到用户态到一次拷贝，它们都经历了两次拷贝。换句话说，要在用户态和内核态流转的东西，它们都是不可重用的，也就没有是不是缺点这一说。）
- select 之所以慢于 poll，是因为 select 要返回 3 个 fd_set 给用户态，分别为 readfds, writefds, exceptfds，然后用户态要 loop 三遍，才能进行完本轮可进行的 IO 操作。而 poll 只需要 loop 一遍 pollfds 就可以了。

epoll



(好像在瞎讲)

相比于 poll 的优化：

- 数组在内核态和用户态共享，避免拷贝开销（存疑）
- 置位方法改变，通过重排而非 flag 置位，让数据就绪的 fd 排在最前面

说明 2

[select、poll、epoll之间的区别总结\[整理\]](#)

select, poll, epoll 都是 IO 多路复用的机制。I/O 多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但 select, poll, epoll 本质上都是同步 I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步 I/O 则无需自己负责进行读写，异步 I/O 的实现会负责把数据从内核拷贝到用户空间。

select 的几大缺点：

1. 每次调用 select，都需要把 fd 集合从用户态拷贝到内核态，这个开销在 fd 很多时会很大
2. 同时每次调用 select 都需要在内核遍历传递进来的所有 fd，这个开销在 fd 很多时也很大
3. select 支持的文件描述符数量太小了，默认是 1024

epoll 既然是对 select 和 poll 的改进，就应该能避免上述的三个缺点。那 epoll 都是怎么解决的呢？在此之前，我们先看一下 epoll 和 select 和 poll 的调用接口上的不同，select 和 poll 都只提供了一个函数——select 或者 poll 函数。而 epoll 提供了三个函数，epoll_create, epoll_ctl 和 epoll_wait，epoll_create 是创建一个 epoll 句柄；epoll_ctl 是注册要监听的事件类型；epoll_wait 则是等待事件的产生。

对于第一个缺点，epoll 的解决方案在 `epoll_ctl` 函数中。每次注册新的事件到 epoll 句柄中时（在 `epoll_ctl` 中指定 `EPOLL_CTL_ADD`），会把所有的 fd 拷贝进内核，而不是在 `epoll_wait` 的时候重复拷贝。epoll 保证了每个 fd 在整个过程中只会拷贝一次。

对于第二个缺点，epoll 的解决方案不像 select 或 poll 一样每次都把 current 轮流加入 fd 对应的设备等待队列中，而只在 epoll_ctl 时把 current 挂一遍（这一遍必不可少）并为每个 fd 指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调函数，而这个回调函数会把就绪的 fd 加入一个就绪链表）。epoll_wait 的工作实际上就是在这个就绪链表中查看有没有就绪的 fd（利用 schedule_timeout() 实现睡一会，判断一会的效果，和 select 实现中的第 7 步是类似的）。

对于第三个缺点，epoll 没有这个限制，它所支持的 FD 上限是最大可以打开文件的数目，这个数字一般远大于 2048,举个例子,在 1GB 内存的机器上大约是 10 万左右，具体数目可以 cat /proc/sys/fs/file-max 察看,一般来说这个数目和系统内存关系很大。

总结：

1. `select`, `poll` 实现需要自己不断轮询所有 `fd` 集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而 `epoll` 其实也需要调用 `epoll_wait` 不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪 `fd` 放入就绪链表中，并唤醒在 `epoll_wait` 中进入睡眠的进程。虽然都要睡眠和交替，但是 `select` 和 `poll` 在“醒着”的时候要遍历整个 `fd` 集合，而 `epoll` 在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的 CPU 时间。这就是回调机制带来的性能提升。
2. `select`, `poll` 每次调用都要把 `fd` 集合从用户态往内核态拷贝一次，并且要把 `current` 往设备等待队列中挂一次，而 `epoll` 只要一次拷贝，而且把 `current` 往等待队列上挂也只挂一次（在 `epoll_wait` 的开始，注意这里的等待队列并不是设备等待队列，只是一个 `epoll` 内部定义的等待队列）。这也能节省不少的开销。

说明 3

 [\[bilibili\] select/poll/epoll原理](#)

I/O 多路复用模型-epoll示例

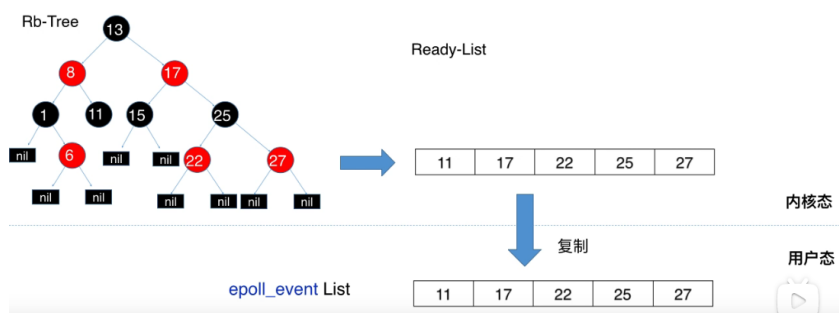
```
int fds[] = ...;
int efd = epoll_create(...);          → 内核态创建epoll实例（红黑树rbr和就绪链表rdlist）

for(int i=0; i < fds.count; i++) {
    epoll_ctl(efd, ..., fds[i], ...); → 对红黑树操作，添加所有socket节点
}

struct epoll_event events[MAX_EVENTS];

while(true) {
    int n = epoll_wait(efd, &events, ...); → 
        1. 阻塞线程
        2. 内核查找红黑树中ready的socket,放入就绪列表
        3. 就绪列表中内容复制到events
    if(n > 0) {
        for(i=0; i<n; i++){
            events[i].data.fd;           → 所有需要处理的socket
        }
    }
}
```

I/O 多路复用模型-epoll工作图解



I/O 多路复用模型-Select vs Poll vs Epoll

	select	poll	epoll
Fd 数量	1024	无限制	无限制
Fd状态感知	轮询[O(N)]	轮询[O(N)]	事件通知[O(1)]
重置原数据	需要	不需要 (event/revent)	不需要 (只通知就绪的)
运行模式	条件触发(LT)	条件触发(LT)	边缘触发(ET)/条件触发(LT)

假设某次读取只读取了一部分数据：

- 条件触发情况下，下次还会通知事件（redis）
- 边缘事件情况下，下次就不会通知了，高效，但可能会遗漏事件（Nginx）

[IO复用\(较详细\)](#)

[深入理解 Epoll](#)

epoll 相比于 select 并不是在所有情况下都要高效，例如在如果有少于 1024 个文件描述符监听（链接少），且大多数 socket 都是处于活跃繁忙的状态，这种情况下，select 要比 epoll 更为高效，因为 epoll 会有更多次的系统调用，用户态和内核态会有更加频繁的切换。

说明 4

[图文详解 epoll 原理【Redis，Netty，Nginx实现高性能IO的核心原理】epoll 详解](#)

epoll 与 select、poll 的对比

- 用户态将文件描述符传入内核的方式
 - select：创建 3 个文件描述符集并拷贝到内核中，分别监听读、写、异常动作。这里受到单个进程可以打开的 fd 数量限制，默认是 1024。
 - poll：将传入的 struct pollfd 结构体数组拷贝到内核中进行监听。
 - epoll：执行 epoll_create，会在内核的高速 cache 区中，建立一颗红黑树以及就绪链

表(该链表存储已经就绪的文件描述符)。接着用户执行的 `epoll_ctl` 函数，添加文件描述符会在红黑树上增加相应的结点。

- 内核态检测文件描述符读写状态的方式
 - `select`：采用轮询方式，遍历所有 `fd`，最后返回一个描述符读写操作是否就绪的 `mask` 掩码，根据这个掩码给 `fd_set` 赋值。
 - `poll`：同样采用轮询方式，查询每个 `fd` 的状态，如果就绪则在等待队列中加入一项并继续遍历。
 - `epoll`：采用事件回调机制。在执行 `epoll_ctl` 的 `add` 操作时，不仅将文件描述符放到红黑树上，而且也注册了回调函数；内核在检测到某文件描述符可读/可写时会调用回调函数，该回调函数将文件描述符放在就绪链表中。
- 找到就绪的文件描述符并传递给用户态的方式
 - `select`：将之前传入的 `fd_set` 拷贝传出到用户态并返回就绪的文件描述符总数。用户态并不知道是哪些文件描述符处于就绪态，需要遍历来判断。
 - `poll`：将之前传入的 `fd` 数组拷贝传出用户态，并返回就绪的文件描述符总数。用户态并不知道是哪些文件描述符处于就绪态，需要遍历来判断。
 - `epoll`：`epoll_wait` 只用观察就绪链表中有无数据即可，最后将链表的数据返回给数组，并返回就绪的数量。内核，将就绪的文件描述符，放在传入的数组中。然后，依次遍历，处理即可。这里返回的文件描述符，是通过 `mmap()`，让内核和用户空间，共享同一块内存实现传递的，减少了不必要的拷贝。
- 重复监听的处理方式
 - `select`：将新的监听文件描述符集合拷贝传入内核中，继续以上步骤。
 - `poll`：将新的 `struct pollfd` 结构体数组拷贝传入内核中，继续以上步骤。
 - `epoll`：无需重新构建红黑树，直接沿用已存在的即可。

彭

`select` 需要提供可读可写和异常事件集合的三个参数，它不能处理更多类型的事件，并且下次调用 `select` 需要重置这三个集合，而 `poll` 把文件描述符和事件都定义在 `pollfd` 结构体中，内核每次修改的都是 `pollfd` 中的 `revents` 成员，下次调用 `poll` 时不需要重置 `pollfd` 事件参数。`select` 和 `poll` 调用返回的是整个用户注册的事件集合，采用轮询的方式检测就绪事件。而 `epoll` 采用不同的方式管理用户注册的事件，它在内核中维护一个事件表，采用回调的方式。内核检测到就绪的文件描述符，将触发回调函数，回调函数将该文件描述符上对应的事件插入到内核就绪的事件队列。然后内核将该就绪事件队列中的内容拷贝到用户空间。

当我们调用 `epoll_ctl` 往里塞入百万个 `fd` 时，`epoll_wait` 仍然可以快速的返回，并有效的将发生事件的 `fd` 给我们用户。这是由于我们在调用 `epoll_create` 时，内核除了帮我们在 `epoll` 文件系统里建了个 `file` 结点，在内核 `cache` 里建了个红黑树用于存储以后 `epoll_ctl` 传来的 `fd` 外，还会再建立一个 `list` 链表，

用于存储准备就绪的事件，当 `epoll_wait` 调用时，仅仅观察这个 list 链表里有没有数据即可。有数据就返回，没有数据就 sleep，等到 timeout 时间到后即使链表没数据也返回。所以，`epoll_wait` 非常高效。而且，通常情况下即使我们要监控百万计的 fd，大多次也只返回很少量的准备就绪 fd 而已，所以，`epoll_wait` 仅需要从内核态 copy 少量的 fd 到用户态而已。那么，这个准备就绪 list 链表是怎么维护的呢？当我们执行 `epoll_ctl` 时，除了把 fd 放到 `epoll` 文件系统里 file 对象对应的红黑树上之外，还会给内核中断处理程序注册一个回调函数，告诉内核，如果这个 fd 的中断到了，就把它放到准备就绪 list 链表里。所以，当一个 fd（例如 socket）上有数据到了，内核在把设备（例如网卡）上的数据 copy 到内核中后就将来把 fd（socket）插入到准备就绪 list 链表里了。

LT 模式：当 `epoll_wait` 检测到其上有事件发生并将此事件通知应用程序后，应用程序可以不立即处理该事件。这样当应用程序下次调用 `epoll_wait` 时，`epoll_wait` 还会再次向应用程序通知此事件，直到有该事件被处理。而对于采用 **ET 模式** 的文件描述符，当 `epoll_wait` 检测当其上有事件发生时并将此事件通知应用程序后，应用程序必须立即处理该事件，因为后续的 `epoll_wait` 调用不再将此事件通知应用程序。ET 模式在很大程度上降底了同一个 `epoll` 事件被重复触发的次数，因此效率要比 LT 模式高。

```
int select(int maxfdp, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout);
```

微秒

```
int poll(struct pollfd fd[], nfds_t nfds, int timeout);
```

 第二个参数是要监听的文件数目 毫秒

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

select 原理概述

调用 select 时，会发生以下事情：

1. 从用户空间拷贝 fd_set 到内核空间；
2. 注册回调函数 `__pollwait`；
3. 遍历所有 fd，对全部指定设备做 次 poll（这里的 poll 是一个文件操作，它有两个参数，一个是文件 fd 本身，一个是当设备尚未就绪时调用的回调函数 `pollwait`，这个函数把设备自 特有的等待队列传给内核，让内核把当前的进程挂载到其中）；
4. 当设备就绪时，设备就会唤醒在自 特有等待队列中的【所有】节点，于是当前进程就获取到了完成的信号。poll 文件操作返回的是 组标准的掩码，其中的各个位指示当前的不同的就绪状态（全 0 为没有任何事件触发），根据 mask 可对 fd_set 赋值；
5. 如果所有设备返回的掩码都没有显示任何的事件触发，就去掉回调函数的函数指针，进入有限时的睡眠状态，再恢复和不断做 poll，再作有限时的睡眠，直到其中一个设备有事件触发为止。
6. 只要有事件触发，系统调用返回，将 fd_set 从内核空间拷贝到用户空间，回到用户态，用户就可以对相关的 fd 作进一步的读或者写操作了。

epoll 原理概述

调用 `epoll_create` 时，做了以下事情：

1. 内核在 `epoll` 文件系统里建了个 `file` 结点；
2. 在内核 `cache` 里建了个红黑树用于存储以后 `epoll_ctl` 传来的 `socket`；
3. 建立一个 `list` 链表，用于存储准备就绪的事件。

调用 `epoll_ctl` 时，做了以下事情：

1. 把 `socket` 放到 `epoll` 文件系统里 `file` 对象对应的红黑树上；
2. 给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪 `list` 链表里。

调用 `epoll_wait` 时，做了以下事情：

观察 `list` 链表里有没有数据。有数据就返回，没有数据就 `sleep`，等到 `timeout` 时间到后即使链表没数据也返回。而且，通常情况下即使我们要监控百万计的句柄，大多次也只返回很少量的准备就绪句柄而已，所以，`epoll_wait` 仅需要从内核态 `copy` 少量的句柄到用户态而已。

`select` 缺点：

1. 最大并发数限制：使用 32 个整数的 32 位，即 $32 \times 32 = 1024$ 来标识 `fd`，虽然可修改，但是有以下第二点的瓶颈；
2. 效率低：每次都会线性扫描整个 `fd_set`，集合越大速度越慢；
3. 内核/用户空间内存拷贝问题。

`epoll` 的提升：

1. 本身没有最大并发连接的限制，仅受系统中进程能打开的最大文件数目限制；
2. 效率提升：只有活跃的 `socket` 才会主动的去调用 `callback` 函数；
3. 省去不必要的内存拷贝：`epoll` 通过内核与用户空间 `mmap` 同一块内存实现

sizeof 关键词

问题来自 `uthash` 中的一个例子：

```
1 struct my_struct *s, *tmp, *users = NULL;
2 s = (struct my_struct *)malloc(sizeof *s);
```

`s` 没有被初始化就被解引用，而且 `sizeof` 没有加括号。

这是因为 `sizeof` 是一个关键字不是一个函数，而且 `sizeof` 会影响之后元素的编译过程

sizeof 的一个有趣的问题

在 C 编译器的实现中，在产生语法树的时候，当扫描到 `sizeof` 的时候，实际上对 `sizeof` 后面的一整个元素，会单独产生一个小的语法树，但是这个语法树不参与代码生成，仅仅做类型判断。而后面的 `(x=2)`，它所生成的一个小的语法树的父节点是一个 `int` 类型（是整个表达式的类

型是 int，不单单指 x)，然后对于整个 sizeof 的子节点，在大的语法树的生成上，直接用了这个类型的大小来替换，这里就是 4。

如果 sizeof 的运算对象是类型的名字就必须加括号，如果是一个具体的数据加不加都可以

sizeof x 是变量，sizeof (y) 是类型。

sizeof x 实际上等同于 sizeof (decltype(x))。

而众所周知 decltype 是不计算表达式的值，只推导表达式结果类型的。

C/C++:sizeof数组与指针

sizeof 是编译时进行的，也就是说，其值的大小，是在运行之前就已经决定好的，不像函数调用，是在运行期间决定的。