

一、作者介绍

大家好，我是小贺，北交大师兄，目前就职于宇宙中心五道口，是这本 C++ 八股文 PDF 的作者，电子书的内容整理于我公众号「**herongwei**」里的图解文章和 [GitHub](#) 仓库的学习笔记。

还没关注的朋友，可以微信搜索「**herongwei**」，或者扫码关注我的公众号，后续最新版本的 PDF 会在我的公众号第一时间发布，而且会有更多其他系列的图解文章，比如数据库、算法等等。



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复 「大礼包」
白嫖编程学习资料

关注公众号回复 「加群」 拉
你进百人技术交流群

简单介绍下这份 C++ 八股文 PDF

背景：最近秋招找工作，很多读者都来问我学习和面试的资料，因此整理了下常见的 C++ 高频面试知识点。分享给你。

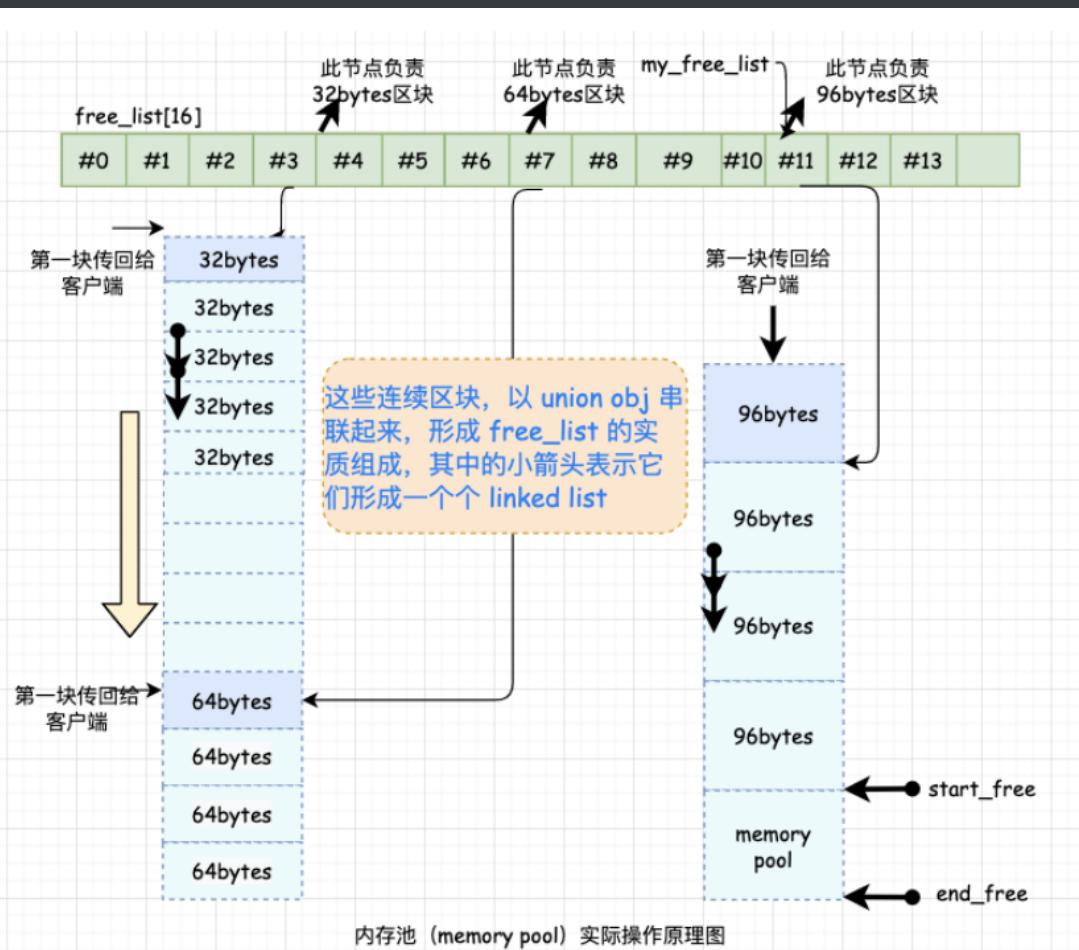
这本电子书总共有 **20W 多字 +100 多张图**，文字都是小贺一个字一个字敲出来的，很多图片都是小贺一个点一条线画出来的，非常的不容易。

这本书整理的知识主要是面向程序员的，因为小贺本身也是个程序员，所以涉及到的知识主要是关于程序员日常工作或者面试的 C++ 知识。

适合有一点 C++ 基础，但是又不怎么扎实，或者知识点串不起来的同学，说白了，这本电子书就是为了拯救面试突击的你，适合面试突击 C++ 后台岗位知识时拿来看，不敢说 100 % 涵盖了面试问题，但是至少 90% 是有的。

这里允许小贺自卖自夸一下：其中加餐篇-图解 STL 源码（第十三章-第十七章）小贺强烈推荐大家看一看，虽然只有五篇，但是耗费了小贺几个月的时间输出，真的是 STL 源码剖析的精华中的精华了 

举个栗子，随便拿出一张图，你都能看的出小贺的用心，是真的是图文并茂！就是为了初学者能明明白白的搞定它！



NOTE: 上述就是 STL 源码当中实际内存池的操作原理，我们可以看到其实以共用体串联起来共享内存形成了 `free_list` 的实质组成。

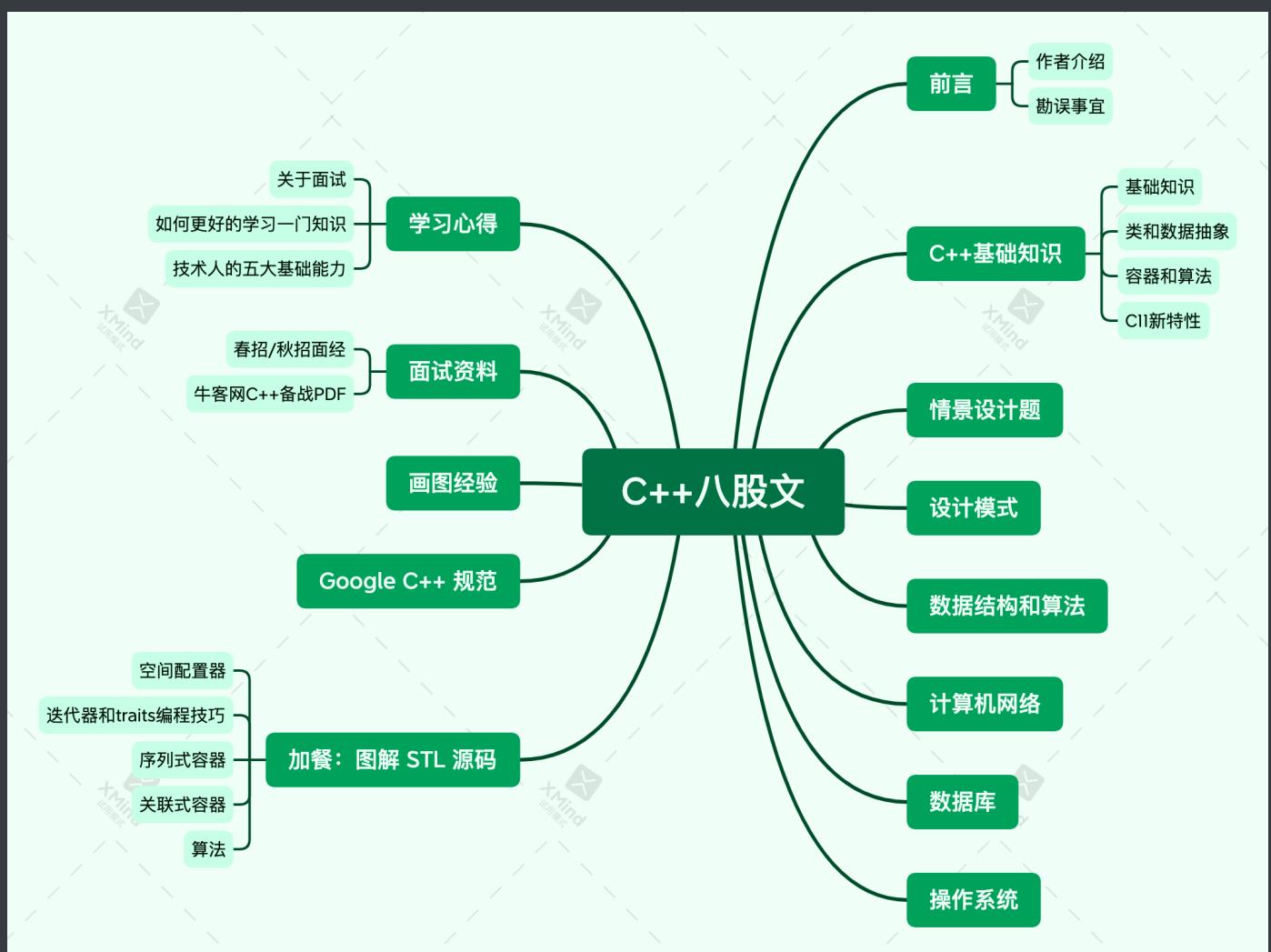
那这本书要怎么阅读呢？

这本电子书不是教科书，肯定没有教科书的知识点全面，是一本为了准备面试的辅助突击复习资料。

如果你还不是到即将面试的时间点，建议多花时间系统学习底层知识，同时扩充知识面，才是长久之计。这个时间你需要的是加速度，而不是速度。

阅读的顺序可以不用从头读到尾，你可以根据你想要了解的知识点，去看哪个章节的文章就好，可以随意阅读任何章节的文章。

下面这张思维导图是整个电子书的目录结构：



二、勘误事宜

小贺整理时间仓促，难免会有很多错别字，所以在学习这份电子书的同学，如果你发现有任何错误或者疑惑的地方，欢迎你通过下方的邮箱反馈给小贺，小贺会逐个修正，然后发布新版本的 C++ PDF，一起迭代出更好的 C++ PDF！

勘误邮箱：1952281585@qq.com

三、C++ 语言基础篇

1、说一下你理解的 C++ 中的四种智能指针



面试官你好，首先，说一下为什么要使用智能指针：智能指针其作用是管理一个指针，避免咱们程序员申请的空间在函数结束时忘记释放，造成内存泄漏这种情况滴发生。

然后使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

常用接口

```
T* get();  
T& operator*();  
T* operator->();  
T& operator=(const T& val);  
T* release();  
void reset (T* ptr = nullptr);
```

- T 是模板参数, 也就是传入的类型；
- get() 用来获取 auto_ptr 封装在内部的指针, 也就是获取原生指针；
- operator() 重载 , operator->() 重载了->, operator=()重载了=;
- realease() 将 auto_ptr 封装在内部的指针置为 nullptr, 但并不会破坏指针所指向的内容, 函数返回的是内部指针置空之前的值；
- 直接释放封装的内部指针所指向的内存, 如果指定了 ptr 的值, 则将内部指针初始化为该值 (否则将其设置为nullptr;

下面分别说一下哪四种：

1、**auto_ptr** (C++98 的方案, C11 已抛弃) 采用所有权模式。

```
auto_ptr<std::string> p1 (new string ("hello"));  
auto_ptr<std::string> p2;  
p2 = p1; //auto_ptr 不会报错.
```

此时不会报错, p2 剥夺了 p1 的所有权, 但是当程序运行时访问 p1 将会报错。所以 auto_ptr 的缺点是：存在潜在的内存崩溃问题！

2、**unique_ptr** (替换 auto_ptr)

`unique_ptr` 实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露特别有用。

采用所有权模式，还是上面那个例子

```
unique_ptr<string> p3 (new string (auto)); //#4  
unique_ptr<string> p4; //#5  
p4 = p3; //此时会报错
```

编译器认为 `p4=p3` 非法，避免了 `p3` 不再指向有效数据的问题。

因此，`unique_ptr` 比 `auto_ptr` 更安全。

3、`shared_ptr`（共享型，强引用）

`shared_ptr` 实现共享式拥有概念，多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。从名字 `share` 就可以看出资源可以被多个指针共享，它使用计数机制来表明资源被几个指针共享。

可以通过成员函数 `use_count()` 来查看资源的所有者个数，除了可以通过 `new` 来构造，还可以通过传入 `auto_ptr`, `unique_ptr`, `weak_ptr` 来构造。当我们调用 `release()` 时，当前指针会释放资源所有权，计数减一。当计数等于 0 时，资源会被释放。

`shared_ptr` 是为了解决 `auto_ptr` 在对象所有权上的局限性 (`auto_ptr` 是独占的)，在使用引用计数的机制上提供了可以共享所有权的智能指针。

4、`weak_ptr`（弱引用）

`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。进行该对象的内存管理的是那个强引用的 `shared_ptr`。

`weak_ptr` 只是提供了对管理对象的一个访问手段。`weak_ptr` 设计的目的是为配合 `shared_ptr` 而引入的一种智能指针来协助 `shared_ptr` 工作，它只可以从一个 `shared_ptr` 或另一个 `weak_ptr` 对象构造，它的构造和析构不会引起引用计数的增加或减少。

`weak_ptr` 是用来解决 `shared_ptr` 相互引用时的死锁问题，如果说两个 `shared_ptr` 相互引用，那么这两个指针的引用计数永远不可能下降为0，也就是资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用计数，和 `shared_ptr` 之间可以相互转化，`shared_ptr` 可以直接赋值给它，它可以通过调用 `lock` 函数来获得 `shared_ptr`。

当两个智能指针都是 `shared_ptr` 类型的时候，析构时两个资源引用计数会减一，但是两者引用计数还是为 1，导致跳出函数时资源没有被释放（的析构函数没有被调用），解决办法：把其中一个改为 `weak_ptr` 就可以。

2、C++ 中内存分配情况

栈：由编译器管理分配和回收，存放局部变量和函数参数。

堆：由程序员管理，需要手动 `new malloc` `delete free` 进行分配和回收，空间较大，但可能会出现内存泄漏和空闲碎片的情况。

全局/静态存储区：分为初始化和未初始化两个相邻区域，存储初始化和未初始化的全局变量和静态变量。

常量存储区：存储常量，一般不允许修改。

代码区：存放程序的二进制代码。

3、C++ 中的指针参数传递和引用参数传递

指针参数传递本质上是值传递，它所传递的是一个地址值。值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，会在栈中开辟内存空间以存放由主调函数传递进来的实参值，从而形成了实参的一个副本（替身）。值传递的特点是，被调函数对形式参数的任何操作都是作为局部变量进行的，不会影响主调函数的实参变量的值（形参指针变了，实参指针不会变）。

引用参数传递过程中，被调函数的形式参数也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的**实参变量的地址**。被调函数对形参（本体）的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量（根据别名找到主调函数中的本体）。因此，被调函数对形参的任何操作都会影响主调函数中的实参变量。

引用传递和指针传递是不同的，虽然他们都是在被调函数栈空间上的一个局部变量，但是任何对于引用参数的处理都会通过一个间接寻址的方式操作到主调函数中的相关变量。而对于指针传递的参数，如果改变被调函数中的指针地址，它将应用不到主调函数的相关变量。如果想通过指针参数传递来改变主调函数中的相关变量（地址），那就得使用指向指针的指针或者指针引用。

从编译的角度来讲，程序在编译时分别将指针和引用添加到符号表上，符号表中记录的是变量名及变量所对应地址。指针变量在符号表上对应的地址值为指针变量的地址值，而引用在符号表上对应的地址值为引用对象的地址值（与实参名字不同，地址相同）。符号表生成之后就不会再改，因此指针可以改变其指向的对象（指针变量中的值可以改），而引用对象则不能修改。

4、C++ 中 **const** 和 **static** 关键字（定义，用途）

static 作用：控制变量的存储方式和可见性。

作用一：修饰局部变量：一般情况下，对于局部变量在程序中是存放在栈区的，并且局部的生命周期在包含语句块执行结束时便结束了。但是如果用 **static** 关键字修饰的话，该变量便会存放在静态数据区，其生命周期会一直延续到整个程序执行结束。但是要注意的是，虽然用 **static** 对局部变量进行修饰之后，其生命周期以及存储空间发生了变化，但其作用域并没有改变，作用域还是限制在其语句块。

作用二：修饰全部变量：对于一个全局变量，它既可以在本文件中被访问到，也可以在同一个工程中其它源文件被访问(添加 **extern** 进行声明即可)。用 **static** 对全局变量进行修饰改变了其作用域范围，由原来的整个工程可见变成了本文件可见。

作用三：修饰函数：用 **static** 修饰函数，情况和修饰全局变量类似，也是改变了函数的作用域。

作用四：修饰类：如果 C++ 中对类中的某个函数用 **static** 修饰，则表示该函数属于一个类而不是属于此类的任何特定对象；如果对类中的某个变量进行 **static** 修饰，则表示该变量以及所有的对象所有，存储空间中只存在一个副本，可以通过类和对象去调用。

（补充：静态非常量数据成员，其只能在类外定义和初始化，在类内仅是声明而已。）

作用五：类成员/类函数声明 static

- 函数体内 static 变量的作用范围为该函数体，不同于 auto 变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值；
- 在模块内的 static 全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；
- 在模块内的 static 函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模块内；
- 在类中的 static 成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；
- 在类中的 static 成员函数属于整个类所拥有，这个函数不接收 this 指针，因而只能访问类的 static 成员变量。
- static 类对象必须要在类外进行初始化，static 修饰的变量先于对象存在，所以 static 修饰的变量要在类外初始化；
- 由于 static 修饰的类成员属于类，不属于对象，因此 static 类成员函数是没有 this 指针，this 指针是指向本对象的指针，正因为没有 this 指针，所以 static 类成员函数不能访问非 static 的类成员，只能访问 static 修饰的类成员；
- static 成员函数不能被 virtual 修饰，static 成员不属于任何对象或实例，所以加上 virtual 没有任何实际意义；静态成员函数没有 this 指针，虚函数的实现是为每一个对象分配一个 vptr 指针，而 vptr 是通过 this 指针调用的，所以不能为 virtual；虚函数的调用关系，`this->vptr->ctable->virtual function。`

const 关键字：含义及实现机制

const 修饰基本类型数据类型：基本数据类型，修饰符 const 可以用在类型说明符前，也可以用在类型说明符后，其结果是一样的。在使用这些常量的时候，只要不改变这些常量的值即可。

const 修饰指针变量和引用变量：如果 const 位于小星星的左侧，则 const 就是用来修饰指针所指向的变量，即指针指向为常量；如果 const 位于小星星的右侧，则 const 就是修饰指针本身，即指针本身是常量。

const 应用到函数中：作为参数的 const 修饰符：调用函数的时候，用相应的变量初始化 const 常量，则在函数体中，按照 const 所修饰的部分进行常量化，保护了原对象的属性。

[注意]：参数 const 通常用于参数为指针或引用的情况；作为函数返回值的 const 修饰符：声明了返回值后，const 按照“修饰原则”进行修饰，起到相应的保护作用。

const 在类中的用法：const 成员变量，只在某个对象生命周期内是常量，而对于整个类而言是可以改变的。因为类可以创建多个对象，不同的对象其 const 数据成员值可以不同。所以不能在类的声明中初始化 const 数据成员，因为类的对象在没有创建时候，编译器不知道 const 数据成员的值是什么。const 数据成员的初始化只能在类的构造函数的初始化列表中进行。

const 成员函数：const 成员函数的主要目的是防止成员函数修改对象的内容。要注意，const 关键字和 static 关键字对于成员函数来说是不能同时使用的，因为 static 关键字修饰静态成员函数不含有 this 指针，即不能实例化，const 成员函数又必须具体到某一个函数。

const 修饰类对象，定义常量对象：常量对象只能调用常量函数，别的成员函数都不能调用。

补充：const 成员函数中如果实在想修改某个变量，可以使用 mutable 进行修饰。成员变量中如果想建立在整个类中都恒定的常量，应该用类中的枚举常量来实现或者 static const。

C ++ 中的 const 类成员函数（用法和意义）

常量对象可以调用类中的 const 成员函数，但不能调用非 const 成员函数；（原因：对象调用成员函数时，在形参列表的最前面加一个形参 this，但这是隐式的。this 指针是默认指向调用函数的当前对象的，所以，很自然，this 是一个常量指针 test * const，因为不可以修改 this 指针代表的地址。但当成员函数的参数列表（即小括号）后加了 const 关键字（void print() const;），此成员函数为常量成员函数，此时它的隐式 this 形参为 const test * const，即不可以通过 this 指针来改变指向对象的值。）

非常量对象可以调用类中的 const 成员函数，也可以调用非 const 成员函数。

5、C 和 C++ 区别（函数/类/struct/class）

首先，C 和 C++ 在基本语句上没有过大的区别。

C++ 有新增的**语法和关键字**，语法的区别有头文件的不同和命名空间的不同，C++ 允许我们自己定义自己的空间，C 中不可以。关键字方面比如 C++ 与 C 动态管理内存的方式不同，C++ 中在 malloc 和 free 的基础上增加了 new 和 delete，而且 C++ 中在指针的基础上增加了引用的概念，关键字例如 C++ 中还增加了 auto，explicit 体现显示和隐式转换上的概念要求，还有 dynamic_cast 增加类型安全方面的内容。

函数方面 C++ 中有重载和虚函数的概念：C++ 支持函数重载而 C 不支持，是因为 C++ 函数的名字修饰与 C 不同，C++ 函数名字的修饰会将参数加在后面，例如，int func(int,double) 经过名字修饰之后会变成_func_int_double，而 C 中则会变成 _func，所以 C++ 中会支持不同参数调用不同函数。

C++ 还有虚函数概念，用以实现多态。

类方面，C 的 struct 和 C++ 的类也有很大不同：C++ 中的 struct 不仅可以有成员变量还可以有成员函数，而且对于 struct 增加了权限访问的概念，struct 的默认成员访问权限和默认继承权限都是 public，C++ 中除了 struct 还有 class 表示类，struct 和 class 还有一点不同在于 class 的默认成员访问权限和默认继承权限都是 private。

C++ 中增加了模板还重用代码，提供了更加强大的 STL 标准库。

最后补充一点就是 C 是一种结构化的语言，重点在于算法和数据结构。C 程序的设计首先考虑的是如何通过一个代码，一个过程对输入进行运算处理输出。而 C++ 首先考虑的是如何构造一个对象模型，让这个模型能够契合与之对应的问题领域，这样就能通过获取对象的状态信息得到输出。

C 的 struct 更适合看成是一个数据结构的实现体，而 C++ 的 class 更适合看成是一个对象的实现体。

6、C++ 和 Java 区别（语言特性，垃圾回收，应用场景等）

指针：Java 语言让程序员没法找到指针来直接访问内存，没有指针的概念，并有内存的自动管理功能，从而有效的防止了 C++ 语言中的指针操作失误的影响。但并非 Java 中没有指针，Java 虚拟机内部中还是用了指针，保证了 Java 程序的安全。

多重继承：C++ 支持多重继承但 Java 不支持，但支持一个类继承多个接口，实现 C++ 中多重继承的功能，又避免了 C++ 的多重继承带来的不便。

数据类型和类：Java 是完全面向对象的语言，所有的函数和变量必须是类的一部分。除了基本数据类型之外，其余的都作为类对象，对象将数据和方法结合起来，把它们封装在类中，这样每个对象都可以实现自己的特点和行为。Java 中取消了 C++ 中的 struct 和 union。

自动内存管理：Java 程序中所有对象都是用 new 操作符建立在内存堆栈上，Java 自动进行无用内存回收操作，不需要程序员进行手动删除。而 C++ 中必须由程序员释放内存资源，增加了程序设计者的负担。Java 中当一个对象不再被用到时，无用内存回收器将给他们加上标签。Java 里无用内存回收程序是以线程方式在后台运行的，利用空闲时间工作来删除。

Java 不支持操作符重载。操作符重载被认为是 C++ 的突出特性。

Java 不支持预处理功能。C++ 在编译过程中都有一个预编译阶段，Java 没有预处理器，但它提供了 import 与 C++ 预处理器具有类似功能。

类型转换：C++ 中有数据类型隐含转换的机制，Java 中需要限时强制类型转换。

字符串：C++ 中字符串是以 Null 终止符代表字符串的结束，而 Java 的字符串 是用类对象（string 和 stringBuffer）来实现的。

Java 中不提供 goto 语句，虽然指定 goto 作为关键字，但不支持它的使用，使程序简洁易读。

Java 的异常机制用于捕获例外事件，增强系统容错能力。

7、说一下 C++ 里是怎么定义常量的？常量存放在内存的哪个位置？

对于局部常量，存放在栈区；

对于全局常量，编译期一般不分配内存，放在符号表中以提高访问效率；

字面值常量，比如字符串，放在常量区。

8、C++ 中重载和重写，重定义的区别

重载

翻译自 overload，是指同一可访问区内被声明的几个具有不同参数列表的同名函数，依赖于 C++ 函数名字的修饰会将参数加在后面，可以是参数类型，个数，顺序的不同。根据参数列表决定调用哪个函数，重载不关心函数的返回类型。

重写

翻译自 override，派生类中重新定义父类中除了函数体外完全相同的虚函数，注意被重写的函数不能是 static 的，一定要是虚函数，且其他一定要完全相同。要注意，重写和被重写的函数是在不同的类当中的，重写函数的访问修饰符是可以不同的，尽管 virtual 中是 private 的，派生类中重写可以改为 public。

重定义（隐藏）

派生类重新定义父类中相同名字的非 virtual 函数，参数列表

和返回类型都可以不同，即父类中除了定义成 virtual 且完全相同的同名函数才不会被派生类中的同名函数所隐藏（重定义）。

9、介绍 C++ 所有的构造函数

类的对象被创建时，编译系统为对象分配内存空间，并自动调用构造函数，由构造函数完成成员的初始化工作。

即构造函数的作用：初始化对象的数据成员。

无参数构造函数：即默认构造函数，如果没有明确写出无参数构造函数，编译器会自动生成默认的无参数构造函数，函数为空，什么也不做，如果不想使用自动生成的无参构造函数，必须要自己显示写出一个无参构造函数。

一般构造函数：也称重载构造函数，一般构造函数可以有各种参数形式，一个类可以有多个一般构造函数，前提是参数的个数或者类型不同，创建对象时根据传入参数不同调用不同的构造函数。

拷贝构造函数：拷贝构造函数的函数参数为对象本身的引用，用于根据一个已存在的对象复制出一个新的该类的对象，一般在函数中会将已存在的对象的数据成员的值一一复制到新创建的对象中。如果没有显示的写拷贝构造函数，则系统会默认创建一个拷贝构造函数，但当类中有指针成员时，最好不要使用编译器提供的默认的拷贝构造函数，最好自己定义并且在函数中执行深拷贝。

类型转换构造函数：根据一个指定类型的对象创建一个本类的对象，也可以算是一般构造函数的一种，这里提出来，是想说有的时候不允许默认转换的话，要记得将其声明为 explicit 的，来阻止一些隐式转换的发生。

赋值运算符的重载：注意，这个类似拷贝构造函数，将 = 右边的本类对象的值复制给 = 左边的对象，它不属于构造函数，= 左右两边的对象必需已经被创建。如果没有显示的写赋值运算符的重载，系统也会生成默认的赋值运算符，做一些基本的拷贝工作。

这里区分

```
A a1, A a2; a1 = a2; //调用赋值运算符  
A a3 = a1; //调用拷贝构造函数，因为进行的是初始化工作，a3 并未存在
```

10、C++ 的四种强制转换

C++ 的四种强制转换包括：**static_cast, dynamic_cast, const_cast, reinterpret_cast**

- **static_cast**：明确指出类型转换，一般建议将隐式转换都替换成显示转换，因为没有动态类型检查，上行转换（派生类->基类）安全，下行转换（基类->派生类）不安全，所以主要执行非多态的转换操作；
- **dynamic_cast**：专门用于派生类之间的转换，type-id 必须是类指针，类引用或 void*，对于下行转换是安全的，当类型不一致时，转换过来的是空指针，而 static_cast，当类型不一致时，转换过来的事错误意义的指针，可能造成非法访问等问题。
- **const_cast**：专门用于 const 属性的转换，去除 const 性质，或增加 const 性质，是四个转换符中唯一一个可以操作常量的转换符。
- **reinterpret_cast**：不到万不得已，不要使用这个转换符，高危操作。使用特点：从底层对数据进行重新解释，依赖具体的平台，可移植性差；可以将整形转换为指针，也可以把指针转换为数组；可以在指针和引用之间进行肆无忌惮的转换。

11、指针和引用的区别

指针和引用都是一种内存地址的概念，区别呢，指针是一个实体，引用只是一个别名。

在程序编译的时候，将指针和引用添加到符号表中。

指针它指向一块内存，指针的内容是所指向的内存的地址，在编译的时候，则是将“指针变量名-指针变量的地址”添加到符号表中，所以说，指针包含的内容是可以改变的，允许拷贝和赋值，有 const 和非 const 区别，甚至可以为空，`sizeof` 指针得到的是指针类型的大小。

而对于引用来说，它只是一块内存的别名，在添加到符号表的时候，是将"引用变量名-引用对象的地址"添加到符号表中，符号表一经完成不能改变，所以引用必须而且只能在定义时被绑定到一块内存上，后续不能更改，也不能为空，也没有 `const` 和非 `const` 区别。

`sizeof` 引用得到代表对象的大小。而 `sizeof` 指针得到的是指针本身的大小。另外在参数传递中，指针需要被解引用后才可以对对象进行操作，而直接对引用进行的修改会直接作用到引用对象上。

作为参数时也不同，传指针的实质是传值，传递的值是指针的地址；传引用的实质是传地址，传递的是变量的地址。

12、野(wild)指针与悬空(dangling)指针有什么区别？如何避免？

野指针(wild pointer)：就是没有被初始化过的指针。用 `gcc -Wall` 编译，会出现 `used uninitialized` 警告。

悬空指针：是指针最初指向的内存已经被释放了的一种指针。

无论是野指针还是悬空指针，都是指向无效内存区域(这里的无效指的是"不安全不可控")的指针。访问"不安全可控"(invalid)的内存区域将导致"Undefined Behavior"。

如何避免使用野指针？在平时的编码中，养成在定义指针后且在使用之前完成初始化的习惯或者使用智能指针。

13、说一下 `const` 修饰指针如何区分？

下面都是合法的声明，但是含义大不同：

`const int * p1; //指向整形常量 的指针，它指向的值不能修改`

`int * const p2; //指向整形的常量指针，它不能在指向别的变量，但指向（变量）的值可以修改。`

`const int *const p3; //指向整形常量 的 常量指针。它既不能再指向别的常量，指向的值也不能修改。`

理解这些声明的技巧在于，查看关键字const右边来确定什么被声明为常量，如果该关键字的右边是类型，则值是常量；如果关键字的右边是指针变量，则指针本身是常量。

14、简单说一下函数指针

从定义和用途两方面来说一下自己的理解：

首先是定义：函数指针是指向函数的指针变量。函数指针本身首先是一个指针变量，该指针变量指向一个具体的函数。这正如用指针变量可指向整型变量、字符型、数组一样，这里是指向函数。

在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。有了指向函数的指针变量后，可用该指针变量调用函数，就如同用指针变量可引用其他类型变量一样，在这些概念上是大体一致的。

其次是用途：调用函数和做函数的参数，比如回调函数。

示例：

```
char * fun(char * p) { ... } // 函数fun
char * (*pf)(char * p); // 函数指针pf
pf = fun; // 函数指针pf指向函数fun
pf(p); // 通过函数指针pf调用函数fun
```

15、堆和栈区别

栈

由编译器进行管理，在需要时由编译器自动分配空间，在不需要时候自动回收空间，一般保存的是局部变量和函数参数等。

连续的内存空间，在函数调用的时候，首先入栈的主函数的下一条可执行指令的地址，然后是函数的各个参数。

大多数编译器中，参数是从右向左入栈（原因在于采用这种顺序，是为了让程序员在使用C/C++的“函数参数长度可变”这个特性时更方便。如果是从左向右压栈，第一个参数（即描述可变参数表各变量类型的那个参数）将被放在栈底，由于可变参的函数第一步就需要解析可变

参数表的各参数类型，即第一步就需要得到上述参数，因此，将它放在栈底是很不方便的。) 本次函数调用结束时，局部变量先出栈，然后是参数，最后是栈顶指针最开始存放的地址，程序由该点继续运行，不会产生碎片。

栈是高地址向低地址扩展，栈低高地址，空间较小。

堆

由程序员管理，需要手动 new malloc delete free 进行分配和回收，如果不进行回收的话，会造成内存泄漏的问题。

不连续的空间，实际上系统中有一个空闲链表，当有程序申请的时候，系统遍历空闲链表找到第一个大于等于申请大小的空间分配给程序，一般在分配程序的时候，也会空间头部写入内存大小，方便 delete 回收空间大小。当然如果有剩余的，也会将剩余的插入到空闲链表中，这也是产生内存碎片的原因。

堆是低地址向高地址扩展，空间交大，较为灵活。

16、函数传递参数的几种方式

值传递：形参是实参的拷贝，函数内部对形参的操作并不会影响到外部的实参。

指针传递：也是值传递的一种方式，形参是指向实参地址的指针，当对形参的指向操作时，就相当于对实参本身进行操作。

引用传递：实际上就是把引用对象的地址放在了开辟的栈空间中，函数内部对形参的任何操作可以直接映射到外部的实参上面。

17、new / delete , malloc / free 区别

都可以用来在堆上分配和回收空间。new /delete 是操作符，malloc/free 是库函数。

执行 new 实际上执行两个过程：1.分配未初始化的内存空间（malloc）；2.使用对象的构造函数对空间进行初始化；返回空间的首地址。如果在第一步分配空间中出现问题，则抛出 std::bad_alloc 异常，或被某个设定的异常处理函数捕获处理；如果在第二步构造对象时出现异常，则自动调用 delete 释放内存。

执行 delete 实际上也有两个过程：1. 使用析构函数对对象进行析构；2. 回收内存空间（free）。

以上也可以看出 new 和 malloc 的区别，new 得到的是经过初始化的空间，而 malloc 得到的是未初始化的空间。所以 new 是 new 一个类型，而 malloc 则是 malloc 一个字节长度的空间。delete 和 free 同理，delete 不仅释放空间还析构对象，delete 一个类型，free 一个字节长度的空间。

为什么有了 malloc / free 还需要 new / delete？因为对于非内部数据类型而言，光用 malloc / free 无法满足动态对象的要求。对象在创建的同时需要自动执行构造函数，对象在消亡以前要自动执行析构函数。由于 malloc / free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行的构造函数和析构函数的任务强加于 malloc / free，所以有了 new / delete 操作符。

18、volatile 和 extern 关键字

volatile 三个特性

易变性：在汇编层面反映出来，就是两条语句，下一条语句不会直接使用上一条语句对应的 volatile 变量的寄存器内容，而是重新从内存中读取。

不可优化性：volatile 告诉编译器，不要对我这个变量进行各种激进的优化，甚至将变量直接消除，保证程序员写在代码中的指令，一定会被执行。

顺序性：能够保证 volatile 变量之间的顺序性，编译器不会进行乱序优化。

extern

在 C 语言中，修饰符 extern 用在变量或者函数的声明前，用来说明“此变量/函数是在别处定义的，要在此处引用”。

注意 extern 声明的位置对其作用域也有关系，如果是在 main 函数中进行声明的，则只能在 main 函数中调用，在其它函数中不能调用。其实要调用其它文件中的函数和变量，只需把该文件用 #include 包含进来即可，为啥要用 extern？因为用 extern 会加速程序的编译过程，这样能节省时间。

在 C++ 中 `extern` 还有另外一种作用，用于指示 C 或者 C++ 函数的调用规范。比如在 C++ 中调用 C 库函数，就需要在 C++ 程序中用 `extern "C"` 声明要引用的函数。这是给链接器用的，告诉链接器在链接的时候用 C 函数规范来链接。主要原因是 C++ 和 C 程序编译完成后在目标代码中命名规则不同，用此来解决名字匹配的问题。

19、`define` 和 `const` 区别（编译阶段、安全性、内存占用等）

对于 `define` 来说，宏定义实际上是在预编译阶段进行处理，没有类型，也就没有类型检查，仅仅做的是遇到宏定义进行字符串的展开，遇到多少次就展开多少次，而且这个简单的展开过程中，很容易出现边界效应，达不到预期的效果。因为 `define` 宏定义仅仅是展开，因此运行时系统并不为宏定义分配内存，但是从汇编的角度来讲，`define` 却以立即数的方式保留了多份数据的拷贝。

对于 `const` 来说，`const` 是在编译期间进行处理的，`const` 有类型，也有类型检查，程序运行时系统会为 `const` 常量分配内存，而且从汇编的角度讲，`const` 常量在出现的地方保留的是真正数据的内存地址，只保留了一份数据的拷贝，省去了不必要的内存空间。而且，有时编译器不会为普通的 `const` 常量分配内存，而是直接将 `const` 常量添加到符号表中，省去了读取和写入内存的操作，效率更高。

20、计算下面几个类的大小

```
class A{}; sizeof(A) = 1; //空类在实例化时得到一个独一无二的地址，所以为 1.  
class A{virtual Fun(){} }; sizeof(A) = 4(32bit)/8(64bit) //当 C++ 类中有虚  
函数的时候，会有一个指向虚函数表的指针 (vptr)  
class A{static int a; }; sizeof(A) = 1;  
class A{int a; }; sizeof(A) = 4;  
class A{static int a; int b; }; sizeof(A) = 4;
```

21、面向对象的三大特性，并举例说明

C++ 面向对象的三大特征是：封装、继承、多态。

所谓封装

就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让信任的类或者对象操作，对不可信的进行信息隐藏。一个类就是一个封装了数据以及操作这些数据的代码的逻辑实体。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

所谓继承

是指可以让某个类型的对象获得另一个类型的对象的属性的方法。它支持按级分类的概念。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。通过继承创建的新类称为“子类”或者“派生类”，被继承的类称为“基类”、“父类”或“超类”。继承的过程，就是从一般到特殊的过程。要实现继承，可以通过“继承”和“组合”来实现。

继承概念的实现方式有两类：

实现继承：实现继承是指直接使用基类的属性和方法而无需额外编码的能力。

接口继承：接口继承是指仅使用属性和方法的名称、但是子类必需提供实现的能力。

所谓多态

就是向不同的对象发送同一个消息，不同对象在接收时会产生不同的行为（即方法）。即一个接口，可以实现多种方法。

多态与非多态的实质区别就是函数地址是早绑定还是晚绑定的。如果函数的调用，在编译器编译期间就可以确定函数的调用地址，并产生代码，则是静态的，即地址早绑定。而如果函数调用的地址不能在编译器期间确定，需要在运行时才确定，这就属于晚绑定。

22、多态的实现

多态其实一般就是指**继承加虚函数实现的多态**，对于重载来说，实际上基于的原理是，编译器为函数生成符号表时的不同规则，重载只是一种语言特性，与多态无关，与面向对象也无关，但这又是 C++ 中增加的新规则，所以也算属于 C++，所以如果说重载算是多态的一种，那就可以说：**多态可以分为静态多态和动态多态**。

静态多态其实也就是重载，因为静态多态是指在编译时期就决定了调用哪个函数，根据参数列表来决定：

动态多态是指通过子类重写父类的虚函数来实现的，因为是在运行期间决定调用的函数，所以称为动态多态，

一般情况下我们不区分这两个时所说的多态就是指动态多态。

动态多态的实现与虚函数表，虚函数指针相关。

扩展：子类是否要重写父类的虚函数？子类继承父类时，父类的纯虚函数必须重写，否则子类也是一个虚类不可实例化。定义纯虚函数是为了实现一个接口，起到一个规范的作用，规范继承这个类的程序员必须实现这个函数。

23、虚函数相关（虚函数表，虚函数指针），虚函数的实现原理

首先我们来说一下，C++中多态的表象，在基类的函数前加上 virtual 关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数，如果是基类，就调用基类的函数。

实际上，当一个类中包含虚函数时，编译器会为该类生成一个虚函数表，保存该类中虚函数的地址，同样，派生类继承基类，派生类中自然一定有虚函数，所以编译器也会为派生类生成自己的虚函数表。当我们定义一个派生类对象时，编译器检测该类型有虚函数，所以为这个派生类对象生成一个虚函数指针，指向该类型的虚函数表，这个虚函数指针的初始化是在构造函数中完成的。

后续如果有一个基类类型的指针，指向派生类，那么当调用虚函数时，就会根据所指真正对象的虚函数表指针去寻找虚函数的地址，也就可以调用派生类的虚函数表中的虚函数以此实现多态。

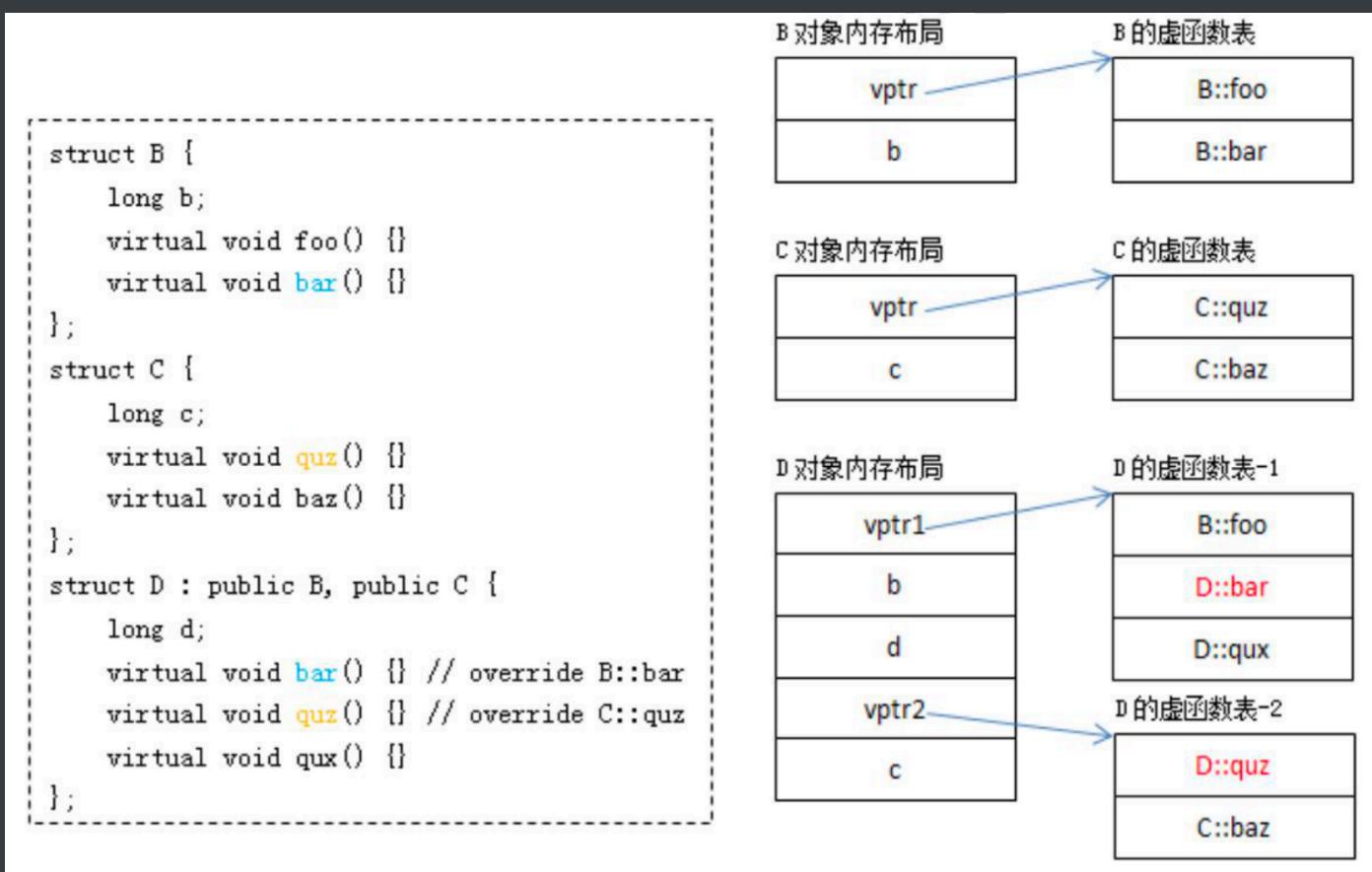
补充：如果基类中没有定义成 virtual，那么进行 Base B; Derived D; Base *p = D; p->function(); 这种情况下调用的则是 Base 中的 function()。因为基类和派生类中都没有虚函数的定义，那么编译器就会认为不用留给动态多态的机会，就事先进行函数地址的绑定（早绑定），详述过程就是，定义了一个派生类对象，首先要构造基类的空间，然后构造派生类的自身内容，形成一个派生类对象，那么在进行类型转换时，直接截取基类的部分的内存，编译器认为类型就是基类，那么（函数符号表 [不同于虚函数表的另一个表] 中）绑定的函数地址也就是基类中函数的地址，所以执行的是基类的函数。

24、编译器处理虚函数表应该如何处理

对于派生类来说，编译器建立虚函数表的过程其实一共是三个步骤：

- 拷贝基类的虚函数表，如果是多继承，就拷贝每个有虚函数基类的虚函数表
- 当然还有一个基类的虚函数表和派生类自身的虚函数表共用了一个虚函数表，也称为某个基类为派生类的主基类
- 查看派生类中是否有重写基类中的虚函数，如果有，就替换成已经重写的虚函数地址；
查看派生类是否有自身的虚函数，如果有，就追加自身的虚函数到自身的虚函数表中。

Derived *pd = new D(); B *pb = pd; C *pc = pd; 其中 pb, pd, pc 的指针位置是不同的，要注意的是派生类的自身的内容要追加在主基类的内存块后。



25、析构函数一般写成虚函数的原因

直观的讲：是为了降低内存泄漏的可能性。举例来说就是，一个基类的指针指向一个派生类的对象，在使用完毕准备销毁时，如果基类的析构函数没有定义成虚函数，那么编译器根据指针类型就会认为当前对象的类型是基类，调用基类的析构函数（该对象的析构函数的函数地址早就被绑定为基类的析构函数），仅执行基类的析构，派生类的自身内容将无法被析构，造成内存泄漏。

如果基类的析构函数定义成虚函数，那么编译器就可以根据实际对象，执行派生类的析构函数，再执行基类的析构函数，成功释放内存。

26、构造函数为什么一般不定义为虚函数

- 虚函数调用只需要知道“部分的”信息，即只需要知道函数接口，而不需要知道对象的具体类型。但是，我们要创建一个对象的话，是需要知道对象的完整信息的。特别是，需要知道要创建对象的确切类型，因此，构造函数不应该被定义成虚函数；
- 而且从目前编译器实现虚函数进行多态的方式来看，虚函数的调用是通过实例化之后对象的虚函数表指针来找到虚函数的地址进行调用的，如果说构造函数是虚的，那么虚函数表指针则是不存在的，无法找到对应的虚函数表来调用虚函数，那么这个调用实际上也是违反了先实例化后调用的准则。

27、构造函数或析构函数中调用虚函数会怎样

实际上是不应该在构造函数或析构函数中调用虚函数的，因为这样的调用其实并不会带来所想要的效果。

举例来说就是，有一个动物的基类，基类中定义了一个动物本身行为的虚函数 `action_type()`，在基类的构造函数中调用了这个虚函数。

派生类中重写了这个虚函数，我们期望着根据对象的真实类型不同，而调用各自实现的虚函数，但实际上当我们创建一个派生类对象时，首先会创建派生类的基类部分，执行基类的构造函数，此时，派生类的自身部分还没有被初始化，对于这种还没有初始化的东西，C++选择当它们还不存在作为一种安全的方法。

也就是说构造派生类的基类部分是，编译器会认为这就是一个基类类型的对象，然后调用基类类型中的虚函数实现，并没有按照我们想要的方式进行。即对象在派生类构造函数执行前并不会成为一个派生类对象。

在析构函数中也是同理，派生类执行了析构函数后，派生类的自身成员呈现未定义的状态，那么在执行基类的析构函数中是不可能调用到派生类重写的方法的。所以说，我们不应该在构造函数或析构函数中调用虚函数，就算调用一般也不会达到我们想要的结果。

28、析构函数的作用，如何起作用？

构造函数只是起初始化值的作用，但实例化一个对象的时候，可以通过实例去传递参数，从主函数传递到其他的函数里面，这样就使其他的函数里面有值了。规则，只要你一实例化对象，系统自动回调用一个构造函数，就是你不写，编译器也自动调用一次。

析构函数与构造函数的作用相反，用于撤销对象的一些特殊任务处理，可以是释放对象分配的内存空间；特点：析构函数与构造函数同名，但该函数前面加~。

析构函数没有参数，也没有返回值，而且不能重载，在一个类中只能有一个析构函数。当撤销对象时，编译器也会自动调用析构函数。每一个类必须有一个析构函数，用户可以自定义析构函数，也可以是编译器自动生成默认的析构函数。一般析构函数定义为类的公有成员。

29、构造函数的执行顺序？析构函数的执行顺序？

构造函数顺序

- 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。
- 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。
- 派生类构造函数。

析构函数顺序

- 调用派生类的析构函数；
- 调用成员类对象的析构函数；
- 调用基类的析构函数。

30、纯虚函数（应用于接口继承和实现继承）

实际上，纯虚函数的出现就是为了让继承可以出现多种情况：

- 有时我们希望派生类只继承成员函数的接口
- 有时我们又希望派生类既继承成员函数的接口，又继承成员函数的实现，而且可以在派生类中可以重写成员函数以实现多态
- 有的时候我们又希望派生类在继承成员函数接口和实现的情况下，不能重写缺省的实现。

其实，声明一个纯虚函数的目的就是为了让派生类只继承函数的接口，而且派生类中必需提供一个这个纯虚函数的实现，否则含有纯虚函数的类将是抽象类，不能进行实例化。

对于纯虚函数来说，我们其实是可以给它提供实现代码的，但是由于抽象类不能实例化，调用这个实现的唯一方式是在派生类对象中指出其 class 名称来调用。

31、静态绑定和动态绑定的介绍

说起静态绑定和动态绑定，我们首先要知道静态类型和动态类型，静态类型就是它在程序中被声明时所采用的类型，在编译期间确定。动态类型则是指“目前所指对象的实际类型”，在运行期间确定。

静态绑定，又名早绑定，绑定的是静态类型，所对应的函数或属性依赖于对象的静态类型，发生在编译期间。

动态绑定，又名晚绑定，绑定的是动态类型，所对应的函数或属性依赖于动态类型，发生在运行期间。

比如说，`virtual` 函数是动态绑定的，非虚函数是静态绑定的，缺省参数值也是静态绑定的。这里呢，就需要注意，我们不应该重新定义继承而来的缺省参数，因为即使我们重定义了，也不会起到效果。因为一个基类的指针指向一个派生类对象，在派生类的对象中针对虚函数的参数缺省值进行了重定义，但是缺省参数值是静态绑定的，静态绑定绑定的是静态类型相关的内容，所以会出现一种派生类的虚函数实现方式结合了基类的缺省参数值的调用效果，这个与所期望的效果不同。

32、深拷贝和浅拷贝的区别（举例说明深拷贝的安全性）

当出现类的等号赋值时，会调用拷贝函数，在未定义显示拷贝构造函数的情况下，系统会调用默认的拷贝函数—即浅拷贝，它能够完成成员的一一复制。当数据成员中没有指针时，浅拷贝是可行的。

但当数据成员中有指针时，如果采用简单的浅拷贝，则两类中的两个指针指向同一个地址，当对象快要结束时，会调用两次析构函数，而导致指野指针的问题。

所以，这时必需采用深拷贝。深拷贝与浅拷贝之间的区别就在于**深拷贝会在堆内存中另外申请空间来存储数据，从而也就解决来野指针的问题**。简而言之，当数据成员中有指针时，必需要用深拷贝更加安全。

33、什么情况下会调用拷贝构造函数(三种情况)

类的对象需要拷贝时，拷贝构造函数将会被调用，以下的情况都会调用拷贝构造函数：

- 一个对象以值传递的方式传入函数体，需要拷贝构造函数创建一个临时对象压入到栈空间中。
- 一个对象以值传递的方式从函数返回，需要执行拷贝构造函数创建一个临时对象作为返回值。
- 一个对象需要通过另外一个对象进行初始化。

34、为什么拷贝构造函数必需时引用传递，不能是值传递？

为了防止递归调用。当一个对象需要以值方式进行传递时，编译器会生成代码调用它的拷贝构造函数生成一个副本，如果类 A 的拷贝构造函数的参数不是引用传递，而是采用值传递，那么就又需要为了创建传递给拷贝构造函数的参数的临时对象，而又一次调用类 A 的拷贝构造函数，这就是一个无限递归。

35、结构体内存对齐方式和为什么要进行内存对齐？

首先我们来说一下结构体中**内存对齐**的规则：

- 对于结构体中的各个成员，第一个成员位于偏移为 0 的位置，以后的每个数据成员的偏移量必须是 `min(#pragma pack())` 制定的数，(数据成员本身长度) 的倍数。
- 在所有的数据成员完成各自对齐之后，结构体或联合体本身也要进行对齐，整体长度是 `min(#pragma pack())` 制定的数，(长度最长的数据成员的长度) 的倍数。

那么**内存对齐的作用**是什么呢？

- 经过内存对齐之后，CPU 的内存访问速度大大提升。因为 CPU 把内存当成是一块一块的，块的大小可以是 2, 4, 8, 16 个字节，因此 CPU 在读取内存的时候是一块一块进行读取的，块的大小称为内存读取粒度。比如说 CPU 要读取一个 4 个字节的数据到寄存器中（假设内存读取粒度是 4），如果数据是从 0 字节开始的，那么直接将 0-3 四个字节完全读取到寄存器中进行处理即可。
- 如果数据是从 1 字节开始的，就首先要将前 4 个字节读取到寄存器，并再次读取 4-7 个字节数据进入寄存器，接着把 0 字节, 5, 6, 7 字节的数据剔除，最后合并 1, 2, 3, 4 字节的数据进入寄存器，所以说，当内存没有对齐时，寄存器进行了很多额外的操作，大大降低了 CPU 的性能。

- 另外，还有一个就是，有的 CPU 遇到未进行内存对齐的处理直接拒绝处理，不是所有的硬件平台都能访问任意地址上的任意数据，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。所以内存对齐还有利于平台移植。

36、内存泄漏的定义，如何检测与避免？

定义：内存泄漏简单的说就是申请了一块内存空间，使用完毕后没有释放掉。它的一般表现方式是程序运行时间越长，占用内存越多，最终用尽全部内存，整个系统崩溃。由程序申请的一块内存，且没有任何一个指针指向它，那么这块内存就泄漏了。

如何检测内存泄漏

- 首先可以通过观察猜测是否可能发生内存泄漏，Linux 中使用 swap 命令观察还有多少可用的交换空间，在一两分钟内键入该命令三到四次，看看可用的交换区是否在减少。
- 还可以使用其他一些 /usr/bin/stat 工具如 netstat、vmstat 等。如发现某段有内存被分配且从不释放，一个可能的解释就是有个进程出现了内存泄漏。
- 当然也有用于内存调试，内存泄漏检测以及性能分析的软件开发工具 valgrind 这样的工具来进行内存泄漏的检测。

37、说一下平衡二叉树、高度平衡二叉树（AVL）

二叉树：任何节点最多只允许有两个子节点，称为左子节点和右子节点，以

递归的方式定义二叉树为，一个二叉树如果不为空，便是由一个根节点和左右两个子树构成，左右子树都可能为空。

二叉搜索树：二叉搜索树可以提供对数时间的元素插入和访问。节点的放置规则是：任何节点的键值一定大于其左子树的每一个节点的键值，并小于其右子树中的每一个节点的键值。因此一直向左走可以取得最小值，一直向右走可以得到最大值。插入：从根节点开始，遇键值较大则向左，遇键值较小则向右，直到尾端，即插入点。删除：如果删除点只有一个子节点，则直接将其子节点连至父节点。如果删除点有两个子节点，以右子树中的最小值代替要删除的位置。

平衡二叉树：其实对于树的平衡与否没有一个绝对的标准，“平衡”的大致意思是：没有任何一个节点过深，不同的平衡条件会造就出不同的效率表现。以及不同的实现复杂度。有数种特殊结构例如 AVL-tree, RB-tree, AA-tree，均可以实现平衡二叉树。

AVL-tree：高度平衡的平衡二叉树（严格的平衡二叉树）AVL-tree 是要求任何节点的左右子树高度相差最多为 1 的平衡二叉树。当插入新的节点破坏平衡性的时候，从下往上找到第一个不平衡点，需要进行单旋转，或者双旋转进行调整。

38、说一下红黑树（RB-tree）

红黑树的定义：

性质1：每个节点要么是黑色，要么是红色。

性质2：根节点是黑色。

性质3：每个叶子节点（NIL）是黑色。

性质4：每个红色结点的两个子结点一定都是黑色。

性质5：任意一结点到每个叶子结点的路径都包含数量相同的黑结点。

39、说一下 **define**、**const**、**typedef**、**inline** 使用方法？

1、**const** 与 **#define** 的区别

const 定义的常量是变量带类型，而 **#define** 定义的只是个常数不带类型；

define 只在预处理阶段起作用，简单的文本替换，而 **const** 在编译、链接过程中起作用；

define 只是简单的字符串替换没有类型检查。而**const**是有数据类型的，是要进行判断的，可以避免一些低级错误；

define 预处理后，占用代码段空间，**const** 占用数据段空间；

const 不能重定义，而 **define** 可以通过 **#undef** 取消某个符号的定义，进行重定义；

define 独特功能，比如可以用来防止文件重复引用。

2、**#define** 和别名 **typedef** 的区别

执行时间不同，**typedef** 在编译阶段有效，**typedef** 有类型检查的功能；**#define** 是宏定义，发

生在预处理阶段，不进行类型检查；

功能差异，`typedef` 用来定义类型的别名，定义与平台无关的数据类型，与 `struct` 的结合使用等。

`#define` 不只是可以为类型取别名，还可以定义常量、变量、编译开关等。

作用域不同，`#define` 没有作用域的限制，只要是之前预定义过的宏，在以后的程序中都可以使用。

而 `typedef` 有自己的作用域。

3、`define` 与 `inline` 的区别

`#define` 是关键字，`inline` 是函数；

宏定义在预处理阶段进行文本替换，`inline` 函数在编译阶段进行替换；

`inline` 函数有类型检查，相比宏定义比较安全；

扩展：

40、预处理，编译，汇编，链接程序的区别

一段高级语言代码经过四个阶段的处理形成可执行的目标二进制代码。

预处理器→编译器→汇编器→链接器：最难理解的是编译与汇编的区别。

这里采用《深入理解计算机系统》的说法。

预处理阶段：写好的高级语言的程序文本比如 `hello.c`，预处理器根据 `#` 开头的命令，修改原始的程序，如`#include<stdio.h>` 将把系统中的头文件插入到程序文本中，通常是以 `.i` 结尾的文件。

编译阶段：编译器将 `hello.i` 文件翻译成文本文件 `hello.s`，这个是汇编语言程序。高级语言是源程序。所以注意概念之间的区别。汇编语言程序是干嘛的？每条语句都以标准的文本格式确切描述一条低级机器语言指令。不同的高级语言翻译的汇编语言相同。

汇编阶段：汇编器将 hello.s 翻译成机器语言指令。把这些指令打包成可重定位目标程序，即 .o 文件。hello.o 是一个二进制文件，它的字节码是机器语言指令，不再是字符。前面两个阶段都还有字符。

链接阶段：比如 hello 程序调用 printf 程序，它是每个 C 编译器都会提供的标准库 C 的函数。这个函数存在于一个名叫 printf.o 的单独编译好的目标文件中，这个文件将以某种方式合并到 hello.o 中。链接器就负责这种合并。得到的是可执行目标文件。

41、说一下 fork, wait, exec 函数

父进程产生子进程使用 fork 拷贝出来一个父进程的副本，此时只拷贝了父进程的页表，两个进程都读同一块内存。

当有进程写的时候使用写实拷贝机制分配内存，exec 函数可以加载一个 elf 文件去替换父进程，从此父进程和子进程就可以运行不同的程序了。

fork 从父进程返回子进程的 pid，从子进程返回 0，调用了 wait 的父进程将会发生阻塞，直到有子进程状态改变，执行成功返回 0，错误返回 -1。

exec 执行成功则子进程从新的程序开始运行，无返回值，执行失败返回 -1。

42、动态编译与静态编译

静态编译，编译器在编译可执行文件时，把需要用到的对应动态链接库中的部分提取出来，连接到可执行文件中去，使可执行文件在运行时不需要依赖于动态链接库；

动态编译，可执行文件需要附带一个动态链接库，在执行时，需要调用其对应动态链接库的命令。所以其优点一方面是缩小了执行文件本身的体积，另一方面是加快了编译速度，节省了系统资源。缺点是哪怕是很简单的程序，只用到了链接库的一两条命令，也需要附带一个相对庞大的链接库；二是如果其他计算机上没有安装对应的运行库，则用动态编译的可执行文件就不能运行。

43、动态链接和静态链接区别

静态连接库就是把 (lib) 文件中用到的函数代码直接链接进目标程序，程序运行的时候不再需要其它的库文件；动态链接就是把调用的函数所在文件模块（DLL）和调用函数在文件中的位置等信息链接进目标程序，程序运行的时候再从 DLL 中寻找相应函数代码，因此需要相应 DLL 文件的支持。

静态链接库与动态链接库都是共享代码的方式，如果采用静态链接库，则无论你愿不愿意， lib 中的指令都全部被直接包含在最终生成的 EXE 文件中了。但是若使用 DLL，该 DLL 不必被包含在最终 EXE 文件中， EXE 文件执行时可以“动态”地引用和卸载这个与 EXE 独立的 DLL 文件。

静态链接库和动态链接库的另外一个区别在于静态链接库中不能再包含其他的动态链接库或者静态库，而在动态链接库中还可以再包含其他的动态或静态链接库。

动态库就是在需要调用其中的函数时，根据函数映射表找到该函数然后调入堆栈执行。如果在当前工程中有多少处对 dll 文件中同一个函数的调用，那么执行时，这个函数只会留下一份拷贝。但如果有多处对 lib 文件中同一个函数的调用，那么执行时该函数将在当前程序的执行空间里留下多份拷贝，而且是一处调用就产生一份拷贝。

44、动态联编与静态联编

在 C++ 中，联编是指一个计算机程序的不同部分彼此关联的过程。按照联编所进行的阶段不同，可以分为静态联编和动态联编；

静态联编是指联编工作在编译阶段完成的，这种联编过程是在程序运行之前完成的，又称为早期联编。要实现静态联编，在编译阶段就必须确定程序中的操作调用（如函数调用）与执行该操作代码间的关系，确定这种关系称为束定，在编译时的束定称为静态束定。静态联编对函数的选择是基于指向对象的指针或者引用的类型。其优点是效率高，但灵活性差。

动态联编是指联编在程序运行时动态地进行，根据当时的情况来确定调用哪个同名函数，实际上是在运行时虚函数的实现。这种联编又称为晚期联编，或动态束定。动态联编对成员函数的选择是基于对象的类型，针对不同的对象类型将做出不同的编译结果。

C++ 中一般情况下的联编是静态联编，但是当涉及到多态性和虚函数时应该使用动态联编。动态联编的优点是灵活性强，但效率低。动态联编规定，只能通过指向基类的指针或基类对象的引用来调用虚函数，其格式为：指向基类的指针变量名->虚函数名（实参表）或基类对象的引用名.虚函数名（实参表）

实现动态联编三个条件：

必须把动态联编的行为定义为类的虚函数；

类之间应满足子类型关系，通常表现为一个类从另一个类公有派生而来；

必须先使用基类指针指向子类型的对象，然后直接或间接使用基类指针调用虚函数；

哈喽，我是小贺哥，就爱分享编程知识，如果觉得文章对你有帮助，别忘记关注我哦！



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复「大礼包」
白嫖编程学习资料

关注公众号回复「加群」拉
你进百人技术交流群

四、类和数据抽象

1、什么是类的继承？

类与类之间的关系

has-A 包含关系，用以描述一个类由多个部件类构成，实现 has-A 关系用类的成员属性表示，即一个类的成员属性是另一个已经定义好的类；

use-A, 一个类使用另一个类，通过类之间的成员函数相互联系，定义友元或者通过传递参数的方式来实现；

is-A, 继承关系，关系具有传递性；

继承的相关概念

所谓的继承就是一个类继承了另一个类的属性和方法，这个新的类包含了上一个类的属性和方法，被称为子类或者派生类，被继承的类称为父类或者基类；

继承的特点

子类拥有父类的所有属性和方法，子类可以拥有父类没有的属性和方法，子类对象可以当做父类对象使用；

继承中的访问控制

`public`、`protected`、`private`

继承中的构造和析构函数

继承中的兼容性原则

2、什么是组合？

一个类里面的数据成员是另一个类的对象，即内嵌其他类的对象作为自己的成员；创建组合类的对象：首先创建各个内嵌对象，难点在于构造函数的设计。创建对象时既要对基本类型的成员进行初始化，又要对内嵌对象进行初始化。

创建组合类对象，构造函数的执行顺序：先调用内嵌对象的构造函数，然后按照内嵌对象成员在组合类中的定义顺序，与组合类构造函数的初始化列表顺序无关。然后执行组合类构造函数的函数体，析构函数调用顺序相反。

3、构造函数析构函数可否抛出异常

C++ 只会析构已经完成的对象，对象只有在其构造函数执行完毕就算是完全构造妥当。在构造函数中发生异常，控制权转出构造函数之外。因此，在对象 b 的构造函数中发生异常，对象b的析构函数不会被调用。因此会造成内存泄漏。

用 auto_ptr 对象来取代指针类成员，便对构造函数做了强化，免除了抛出异常时发生资源泄漏的危机，不再需要在析构函数中手动释放资源；

如果控制权基于异常的因素离开析构函数，而此时正有另一个异常处于作用状态，C++ 会调用 terminate 函数让程序结束；

如果异常从析构函数抛出，而且没有在当地进行捕捉，那个析构函数便是执行不全的。如果析构函数执行不全，就是没有完成他应该执行的每一件事情。

4、类如何实现只能静态分配和只能动态分配

前者是把 new、delete 运算符重载为 private 属性。

后者是把构造、析构函数设为 protected 属性，再用子类来动态创建

建立类的对象有两种方式：

- 静态建立，静态建立一个类对象，就是由编译器为对象在栈空间中分配内存；
- 动态建立，`A *p = new A();` 动态建立一个类对象，就是使用 new 运算符为对象在堆空间中分配内存。这个过程分为两步，第一步执行 operator new() 函数，在堆中搜索一块内存并进行分配；第二步调用类构造函数构造对象；
- 只有使用 new 运算符，对象才会被建立在堆上，因此只要限制 new 运算符就可以实现类对象只能建立在栈上。可以将 new 运算符设为私有。

5、何时需要成员初始化列表？过程是什么？

当初始化一个引用成员变量时；

初始化一个 const 成员变量时；

当调用一个基类的构造函数，而构造函数拥有一组参数时；

当调用一个成员类的构造函数，而他拥有一组参数；

编译器会一一操作初始化列表，以适当顺序在构造函数之内安插初始化操作，并且在任何显示用户代码前。list中的项目顺序是由类中的成员声明顺序决定的，不是初始化列表中的排列顺序决定的。

6、程序员定义的析构函数被扩展的过程？

析构函数函数体被执行；

如果 class 拥有成员类对象，而后者拥有析构函数，那么它们会以其声明顺序的相反顺序被调用；

如果对象有一个 vptr，现在被重新定义

如果有任何直接的上一层非虚基类拥有析构函数，则它们会以声明顺序被调用；

如果任何虚基类拥有析构函数

7、构造函数的执行算法？

在派生类构造函数中，所有的虚基类及上一层基类的构造函数调用；

对象的 vptr 被初始化；

如果有成员初始化列表，将在构造函数体内扩展开来，这必须在 vptr 被设定之后才做；

执行程序员所提供的代码；

8、构造函数的扩展过程？

记录在成员初始化列表中的数据成员初始化操作会被放在构造函数的函数体内，并与成员的声明顺序为顺序；

如果一个成员并没有出现在成员初始化列表中，但它有一个默认构造函数，那么默认构造函数必须被调用；

如果 class 有虚表，那么它必须被设定初值；

所有上一层的基类构造函数必须被调用；

所有虚基类的构造函数必须被调用。

9、哪些函数不能是虚函数

构造函数，构造函数初始化对象，派生类必须知道基类函数干了什么，才能进行构造；当有虚函数时，每一个类有一个虚表，每一个对象有一个虚表指针，虚表指针在构造函数中初始化；

内联函数，内联函数表示在编译阶段进行函数体的替换操作，而虚函数意味着在运行期间进行类型确定，所以内联函数不能是虚函数；

静态函数，静态函数不属于对象属于类，静态成员函数没有this指针，因此静态函数设置为虚函数没有任何意义。

友元函数，友元函数不属于类的成员函数，不能被继承。对于没有继承特性的函数没有虚函数的说法。

普通函数，普通函数不属于类的成员函数，不具有继承特性，因此普通函数没有虚函数。

哈喽，我是小贺哥，就爱分享编程知识，如果觉得文章对你有帮助，别忘记关注我哦！



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复 「大礼包」
白嫖编程学习资料

关注公众号回复 「加群」 拉
你进百人技术交流群

五、STL 容器和算法

具体图解见加餐篇：第十三章-第十七章的图解 STL 源码。

1、C++ 的 STL 介绍（内存管理，allocator，函数，实现机理，多线程实现等）

STL 一共提供六大组件，包括容器，算法，迭代器，仿函数，配接器和配置器，彼此可以组合套用。容器通过配置器取得数据存储空间，算法通过迭代器存取容器内容，仿函数可以协助算法完成不同的策略变化，配接器可以应用于容器、仿函数和迭代器。

容器：各种数据结构，如 vector, list, deque, set, map，用来存放数据，从实现的角度来讲是一种类模板。

算法：各种常用的算法，如 sort（插入，快排，堆排序），search（二分查找），从实现的角度来讲是一种方法模板。

迭代器：从实现的角度来看，迭代器是一种将 operator*, operator->, operator++, operator-- 等指针相关操作赋予重载的类模板，所有的 STL 容器都有自己的迭代器。

仿函数: 从实现的角度看，仿函数是一种重载了 operator() 的类或者类模板。可以帮助算法实现不同的策略。

配接器: 一种用来修饰容器或者仿函数或迭代器接口的东西。

配置器: 负责空间配置与管理，从实现的角度讲，配置器是一个实现了动态空间配置、空间管理，空间释放的类模板。

扩展: 内存管理 allocator

SGI 设计了**双层级配置器**，第一级配置器直接使用 malloc() 和 free() 完成内存的分配和回收。第二级配置器则根据需求量的大小选择不同的策略执行。

对于**第二级配置器**，如果需求块大小大于 128bytes，则直接转而调用第一级配置器，使用 malloc() 分配内存。如果需求块大小小于 128bytes，第二级配置器中维护了 16 个自由链表，负责 16 种小型区块的次配置能力。

即当有小于 128bytes 的需求块要求时，首先查看所需需求块大小所对应的链表中是否有空闲空间，如果有则直接返回，如果没有，则向内存池中申请所需需求块大小的内存空间，如果申请成功，则将其加入到自由链表中。如果内存池中没有空间，则使用 malloc() 从堆中进行申请，且申请到的大小是需求量的二倍（或二倍 + n 附加量），一倍放在自由空间中，一倍（或一倍 + n）放入内存池中。

如果 malloc() 也失败，则会遍历自由空间链表，四处寻找“尚有未用区块，且区块够大”的 freelist，找到一块就挖出一块交出。如果还是没有，仍交由 malloc() 处理，因为 malloc() 有 out-of-memory 处理机制或许有机会释放其他的内存拿来用，如果可以就成功，如果不可以就报 bad_alloc 异常。

STL 中序列式容器的实现:

vector

是动态空间，随着元素的加入，它的内部机制会自行扩充空间以容纳新元素。vector 维护的是一个连续的线性空间，而且普通指针就可以满足要求作为 vector 的迭代器 (RandomAccessIterator)。

`vector` 的数据结构中其实就是三个迭代器构成的，一个指向目前使用空间头的 `iterator`，一个指向目前使用空间尾的 `iterator`，一个指向目前可用空间尾的 `iterator`。当有新的元素插入时，如果目前容量够用则直接插入，如果容量不够，则容量扩充至两倍，如果两倍容量不足，就扩张至足够大的容量。

扩充的过程并不是直接在原有空间后面追加容量，而是重新申请一块连续空间，将原有的数据拷贝到新空间中，再释放原有空间，完成一次扩充。需要注意的是，每次扩充是重新开辟的空间，所以扩充后，原有的迭代器将会失效。

list

与 `vector` 相比，`list` 的好处就是每次插入或删除一个元素，就配置或释放一个空间，而且原有的迭代器也不会失效。`STL list` 是一个双向链表，普通指针已经不能满足 `list` 迭代器的需求，因为 `list` 的存储空间是不连续的。`list` 的迭代器必需具备前移和后退功能，所以 `list` 提供的是 `BidirectionalIterator`。`list` 的数据结构中只要一个指向 `node` 节点的指针就可以了。

deque

`vector` 是单向开口的连续线性空间，`deque` 则是一种双向开口的连续线性空间。所谓双向开口，就是说 `deque` 支持从头尾两端进行元素的插入和删除操作。相比于 `vector` 的扩充空间的方式，`deque` 实际上更加贴切的实现了动态空间的概念。`deque` 没有容量的概念，因为它是动态地以分段连续空间组合而成，随时可以增加一段新的空间并连接起来。

由于要维护这种整体连续的假象，并提供随机存取的接口（即也提供 `RandomAccessIterator`），避开了“重新配置，复制，释放”的轮回，代价是复杂的迭代器结构。也就是说除非必要，我们应该尽可能的使用 `vector`，而不是 `deque`。

那么我们回过来具体说 `deque` 是如何做到维护整体连续的假象的，`deque` 采用一块所谓的 `map` 作为主控，这里的 `map` 实际上就是一块大小连续的空间，其中每一个元素，我们称之为节点 `node`，都指向了另一段连续线性空间称为缓冲区，缓冲区才是 `deque` 的真正存储空间主体。

SGI STL 是允许我们指定 缓冲区的大小的，默认 0 表示使用 512bytes 缓冲区。当 `map` 满载时，我们选用一块更大的空间来作为 `map`，重新调整配置。`deque` 另外一个关键的就是它的 `iterator` 的设计，`deque` 的 `iterator` 中有四个部分，`cur` 指向缓冲区现行元素，`first` 指向缓冲区的头，`last` 指向缓冲区的尾（有时会包含备用空间），`node` 指向管控中心。所以总结来说，`deque`的数据结构中包含了，指向第一个节点的 `iterator start`，和指向最后一个节点的 `iterator`

`finish`, 一块连续空间作为主控 `map`, 也需要记住 `map` 的大小, 以备判断何时配置更大的 `map`。

stack

是一种先进后出的数据结构, 只有一个出口, `stack` 允许从最顶端新增元素, 移除最顶端元素, 取得最顶端元素。`deque` 是双向开口的数据结构, 所以使用 `deque` 作为底部结构并封闭其头端开口, 就形成了一个 `stack`。

queue

是一种先进先出的数据结构, 有两个出口, 允许从最底端加入元素, 取得最顶端元素, 从最底端新增元素, 从最顶端移除元素。`deque` 是双向开口的数据结构, 若以 `deque` 为底部结构并封闭其底端的出口, 和头端的入口, 就形成了一个 `queue`。 (其实 `list` 也可以实现 `deque`)

heap

堆并不属于 STL 容器组件, 它是个幕后英雄, 扮演 `priority_queue` 的助手, `priority_queue` 允许用户以任何次序将任何元素推入容器内, 但取出时一定是从优先权最高 (数值最高) 的元素开始取。大根堆 (binary max heap) 正具有这样的性质, 适合作为 `priority_queue` 的底层机制。

大根堆, 是一个满足每个节点的键值都大于或等于其子节点键值的二叉树 (具体实现是一个 `vector`, 一块连续空间, 通过维护某种顺序来实现这个二叉树), 新加入元素时, 新加入的元素要放在最下一层为叶节点, 即具体实现是填补在由左至右的第一个空格 (即把新元素插入在底层 `vector` 的 `end()`), 然后执行一个所谓上溯的程序: 将新节点拿来与父节点比较, 如果其键值比父节点大, 就父子对换位置, 如此一直上溯, 直到不需要对换或直到根节点为止。当取出一个元素时, 最大值在根节点, 取走根节点, 要割舍最下层最右边的右节点, 并将其值重新安插至最大堆, 最末节点放入根节点后, 进行一个下溯程序: 将空间节点和其较大的节点对调, 并持续下方, 直到叶节点为止。

priority_queue

底层时一个 `vector`, 使用 `heap` 形成的算法, 插入, 获取 `heap` 中元素的算法, 维护这个 `vector`, 以达到允许用户以任何次序将任何元素插入容器内, 但取出时一定是从优先权最高 (数值最高) 的元素开始取的目的。

slist: STL list 是一个双向链表， slist 是一个单向链表。

2、vector 使用的注意点及其原因，频繁对 vector 调用 push_back() 性能影响

使用注意点：

注意插入和删除元素后迭代器失效的问题；

清空 vector 数据时，如果保存的数据项是指针类型，需要逐项 delete，否则会造成内存泄漏。

频繁调用 push_back() 影响：

向 vector 的尾部添加元素，很有可能引起整个对象存储空间的重新分配，重新分配更大的内存，再将原数据拷贝到新空间中，再释放原有内存，这个过程是耗时耗力的，频繁对 vector 调用 push_back() 会导致性能的下降。

在 C++11 之后，vector 容器中添加了新的方法：emplace_back()，和 push_back() 一样的是都是在容器末尾添加一个新的元素进去，不同的是 emplace_back() 在效率上相较于 push_back() 有了一定的提升。

emplace_back() 函数在原理上比 push_back() 有了一定的改进，包括在内存优化方面和运行效率方面。内存优化主要体现在使用了就地构造（直接在容器内构造对象，不用拷贝一个复制品再使用）+强制类型转换的方法来实现，在运行效率方面，由于省去了拷贝构造过程，因此也有一定的提升。

想看图解源码剖析文章看这里：

3、map 和 set 有什么区别，分别又是怎么实现的？

map 和 set 都是 C++ 的关联容器，其底层实现都是红黑树（RB-Tree）。

由于 map 和 set 所开放的各种操作接口，RB-tree 也都提供了，所以几乎所有的 map 和 set 的操作行为，都只是转调 RB-tree 的操作行为。

map 和 set 区别在于：

- (1) map 中的元素是 key-value (关键字—值) 对：关键字起到索引的作用，值则表示与索引相关联的数据；Set与之相对就是关键字的简单集合，set 中每个元素只包含一个关键字。
- (2) set 的迭代器是 const 的，不允许修改元素的值；map 允许修改 value，但不允许修改 key。其原因是因为 map 和 set 是根据关键字排序来保证其有序性的，如果允许修改 key 的话，那么首先需要删除该键，然后调节平衡，再插入修改后的键值，调节平衡，如此一来，严重破坏了 map 和 set 的结构，导致 iterator 失效，不知道应该指向改变前的位置，还是指向改变后的位置。所以 STL 中将 set 的迭代器设置成 const，不允许修改迭代器的值；而 map 的迭代器则不允许修改 key 值，允许修改 value 值。
- (3) map 支持下标操作，set 不支持下标操作。map 可以用 key 做下标，map 的下标运算符 [] 将关键码作为下标去执行查找，如果关键码不存在，则插入一个具有该关键码和 mapped_type 类型默认值的元素至 map 中，因此下标运算符 [] 在 map 应用中需要慎用，const_map 不能用，只希望确定某一个关键值是否存在而不希望插入元素时也不应该使用，mapped_type 类型没有默认值也不应该使用。如果 find 能解决需要，尽可能用 find。

4、请你来说一说 STL 迭代器删除元素

这个主要考察的是迭代器失效的问题。

对于序列容器 vector, deque 来说，使用 `erase(iterator)` 后，后边的每个元素的迭代器都会失效，但是后边每个元素都会往前移动一个位置，但是 `erase` 会返回下一个有效的迭代器；

对于关联容器 map set 来说，使用了 `erase(iterator)` 后，当前元素的迭代器失效，但是其结构是红黑树，删除当前元素的，不会影响到下一个元素的迭代器，所以在调用 `erase` 之前，记录下一个元素的迭代器即可。

对于 list 来说，它使用了不连续分配的内存，并且它的 `erase` 方法也会返回下一个有效的 iterator，因此上面两种正确的方法都可以使用。

5、请你来说一下 STL 中迭代器的作用，有指针为何还要迭代器

迭代器

`Iterator` (迭代器) 模式又称 `Cursor` (游标) 模式，用于提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。或者这样说可能更容易理解：`Iterator`模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序（由`iterator`提供的方法）访问聚合对象中的各个元素。

由于`Iterator`模式的以上特性：与聚合对象耦合，在一定程度上限制了它的广泛运用，一般仅用于底层聚合支持类，如STL的`list`、`vector`、`stack`等容器类及`ostream_iterator`等扩展`iterator`。

迭代器和指针的区别

迭代器不是指针，是类模板，表现的像指针。他只是模拟了指针的一些功能，通过重载了指针的一些操作符，`->`、`*`、`++`、`--`等。迭代器封装了指针，是一个“可遍历STL（Standard Template Library）容器内全部或部分元素”的对象，本质是封装了原生指针，是指针概念的一种提升（lift），提供了比指针更高级的行为，相当于一种智能指针，他可以根据不同类型的数据结构来实现不同的`++`、`--`等操作。

迭代器返回的是对象引用而不是对象的值，所以`cout`只能输出迭代器使用`*`取值后的值而不能直接输出其自身。

迭代器产生原因

`Iterator`类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循环遍历集合的效果。

6、回答一下 STL 里 `resize` 和 `reserve` 的区别

`resize()`：改变当前容器内含有元素的数量(`size()`)，eg: `vector v; v.resize(len);` v的`size`变为`len`,如果原来v的`size`小于`len`，那么容器新增`(len-size)`个元素，元素的值为默认为0.当`v.push_back(3);`之后，则是3是放在了v的末尾，即下标为`len`，此时容器是`size`为`len+1`；

`reserve()`：改变当前容器的最大容量（`capacity`），它不会生成元素，只是确定这个容器允许放入多少对象，如果`reserve(len)`的值大于当前的`capacity()`，那么会重新分配一块能存`len`个对象的空间，然后把之前`v.size()`个对象通过 `copy constructor` 复制过来，销毁之前的内存；



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复 「大礼包」
白嫖编程学习资料

关注公众号回复 「加群」 拉
你进百人技术交流群

六、情景设计题

1、HelloWorld 程序开始到打印到屏幕上的全过程？

- 用户告诉操作系统执行 HelloWorld 程序（通过键盘输入等）；
- 操作系统：找到 HelloWorld 程序的相关信息，检查其类型是否是可执行文件；并通过程序首部信息，确定代码和数据在可执行文件中的位置并计算出对应的磁盘块地址；
- 操作系统：创建一个新进程，将 HelloWorld 可执行文件映射到该进程结构，表示由该进程执行 HelloWorld 程序；
- 操作系统：为 HelloWorld 程序设置 cpu 上下文环境，并跳到程序开始处；
- 执行 HelloWorld 程序的第一条指令，发生缺页异常；
- 操作系统：分配一页物理内存，并将代码从磁盘读入内存，然后继续执行 HelloWorld 程序；
- HelloWorld 程序执行 puts 函数（系统调用），在显示器上写一字符串；
- 操作系统：找到要将字符串送往的显示设备，通常设备是由一个进程控制的，所以，操作系统将要写的字符串送给该进程；
- 操作系统：控制设备的进程告诉设备的窗口系统，它要显示该字符串，窗口系统确定这是一个合法的操作，然后将字符串转换成像素，将像素写入设备的存储映像区；

- 视频硬件将像素转换成显示器可接收和一组控制数据信号;
- 显示器解释信号，激发液晶屏;
- OK，我们在屏幕上看到了 HelloWorld;

2、手写实现智能指针类

```

template<typename T>
class SharedPtr {
private:
    size_t* m_count_;
    T* m_ptr_;
public:
    //构造函数
    SharedPtr(): m_ptr_(nullptr), m_count_(new size_t) {}
    SharedPtr(T* ptr): m_ptr_(ptr), m_count_(new size_t) { m_count_ = 1; }
    //析构函数
    ~SharedPtr() {
        --(*m_count_);
        if (*m_count_ == 0) {
            delete m_ptr_;
            delete m_count_;
            m_ptr_ = nullptr;
            m_count_ = nullptr;
        }
    }
    //拷贝构造函数
    SharedPtr(const SharedPtr& ptr) {
        m_count_ = ptr.m_count_;
        m_ptr_ = ptr.m_ptr_;
        ++(*m_count_);
    }
    //拷贝赋值运算
    void operator=(const SharedPtr& ptr) { SharedPtr(std::move(ptr)); }
    //移动构造函数
    SharedPtr(SharedPtr&& ptr) : m_ptr_(ptr.m_ptr_),
    m_count_(ptr.m_count_) { ++(*m_count_); }
}

```

```

//移动赋值运算
void operator=(SharedPtr&& ptr) { SharedPtr(std::move(ptr)); }

//解引用
T& operator*() { return *m_ptr_; }

//箭头运算
T* operator->() { return m_ptr_; }

//重载bool操作符
operator bool() {return m_ptr_ == nullptr;}
T* get() { return m_ptr_; }

size_t use_count() { return *m_count_; }

bool unique() { return *m_count_ == 1; }

void swap(SharedPtr& ptr) { std::swap(*this, ptr); }

};

```

3、手写字符串函数 `strcat`, `strcpy`, `strncpy`, `memset`, `memcpy`实现

```

//把 src 所指向的字符串复制到 dest, 注意: dest定义的空间应该比src大。
char* strcpy(char *dest,const char *src) {
    char *ret = dest;
    assert(dest!=NULL); //优化点1: 检查输入参数
    assert(src!=NULL);
    while(*src!='\0')
        *(dest++)=*(src++);
    *dest='\0'; //优化点2: 手动地将最后的'\0'补上
    return ret;
}

//考虑内存重叠的字符串拷贝函数 优化的点
char* strcpy(char *dest,char *src) {
    char *ret = dest;
    assert(dest!=NULL);
    assert(src!=NULL);
    memmove(dest,src,strlen(src)+1);
    return ret;
}

//把 src 所指向的字符串追加到 dest 所指向的字符串的结尾。
char* strcat(char *dest,const char *src) {

```

```
//1. 将目的字符串的起始位置先保存，最后要返回它的头指针
//2. 先找到dest的结束位置，再把src拷贝到dest中，记得在最后要加上'\0'
char *ret = dest;
assert(dest!=NULL);
assert(src!=NULL);
while(*dest]!='\0')
    dest++;
while(*src]!='\0')
    *(dest++)=*(src++);
*dest='\0';
return ret;
}

//把 str1 所指向的字符串和 str2 所指向的字符串进行比较。
//该函数返回值如下：
//如果返回值 < 0, 则表示 str1 小于 str2。
//如果返回值 > 0, 则表示 str1 大于 str2。
//如果返回值 = 0, 则表示 str1 等于 str2。
int strcmp(const char *s1,const char *s2) {
    assert(s1!=NULL);
    assert(s2!=NULL);
    while(*s1!='\0' && *s2!='\0') {
        if(*s1>*s2)
            return 1;
        else if(*s1<*s2)
            return -1;
        else {
            s1++,s2++;
        }
    }
    //当有一个字符串已经走到结尾
    if(*s1>*s2)
        return 1;
    else if(*s1<*s2)
        return -1;
    else
        return 0;
}
```

```
}

//在字符串 str1 中查找第一次出现字符串 str2 的位置，不包含终止符 '\0'。
char* strstr(char *str1, char *str2) {
    char* s = str1;
    assert(str1 != '\0');
    assert(str2 != '\0');
    if(*str2 == '\0')
        return NULL; //若str2为空，则直接返回空
    while(*s != '\0') { //若不为空，则进行查询
        char* s1 = s;
        char* s2 = str2;
        while(*s1 != '\0' && *s2 != '\0' && *s1 == *s2)
            s1++, s2++;
        if(*s2 == '\0')
            return str2; //若s2先结束
        if(*s2 != '\0' && *s1 == '\0')
            return NULL; //若s1先结束而s2还没结束，则返回空
        s++;
    }
    return NULL;
}

//模拟实现memcpy函数 从存储区 str2 复制 n 个字符到存储区 dst。
void* memcpy(void* dest, void* src, size_t num) {
    void* ret = dest ;
    size_t i = 0 ;
    assert(dest != NULL) ;
    assert(src != NULL) ;
    for(i = 0; i < num; i++) {
        //因为void* 不能直接解引用，所以需要强转成char*再解引用
        //此处的void*实现了泛型编程
        *(char*) dest = *(char*) src ;
        dest = (char*) dest + 1 ;
        src = (char*) src + 1 ;
    }
    return ret ;
}
```

```
//考虑内存重叠的memcpy函数 优化的点
void* memmove(void* dest, void* src, size_t num) {
    char* p1 = (char*)dest;
    char* p2 = (char*)src;
    if(p1<p2) {//p1低地址p2高地址
        for(size_t i=0; i!=num; ++i)
            *(p1++) = *(p2++);
    }
    else {
        //从后往前赋值
        p1+=num-1;
        p2+=num-1;
        for(size_t i=0; i!=num; ++i)
            *(p1--) = *(p2--);
    }
    return dest;
}
```

4、C++ 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。

5、请你来写个函数在 main 函数执行前先运行

```
//第一种：gcc扩展，标记这个函数应当在main函数之前执行。同样有一个
__attribute__((destructor))，标记函数应当在程序结束之前（main结束之后，或者调用了
exit后）执行；
__attribute__((constructor))void before() {
    printf("before main 1\n");
}
```

```
//第二种：全局 static 变量的初始化在程序初始阶段，先于 main 函数的执行
int test1(){
    cout << "before main 2" << endl;
    return 1;
}
static int i = test1();
// 第三种：知乎大牛 Milo Yip 的回答利用 lambda 表达式
int a = []() {
    cout << "before main 3" << endl;
    return 0;
}();
int main(int argc, char** argv) {
    cout << "main function" << endl;
    return 0;
}
```

输出：

```
before main 1
before main 2
before main 3
main function
```

6、请你来说一下 fork 函数

#Fork：创建一个和当前进程映像一样的进程可以通过 fork() 系统调用：

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

成功调用 fork() 会创建一个新的进程，它几乎与调用 fork() 的进程一模一样，这两个进程都会继续运行。在子进程中，成功的 fork() 调用会返回0。在父进程中 fork() 返回子进程的 pid。

如果出现错误，fork() 返回一个负值。

最常见的 `fork()` 用法是创建一个新的进程，然后使用 `exec()` 载入二进制映像，替换当前进程的映像。这种情况下，派生 (`fork()`) 了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。

在早期的 Unix 系统中，创建进程比较原始。当调用 `fork` 时，内核会把所有的内部数据结构复制一份，复制进程的页表项，然后把父进程的地址空间中的内容逐页的复制到子进程的地址空间中。但从内核角度来说，逐页的复制方式是十分耗时的。现代的 Unix 系统采取了更多的优化，例如 Linux，采用了写时复制的方法，而不是对父进程空间进程整体复制。

7、说一下 `++i` 和 `i++` 的区别

菜鸡小贺： 这题我会！`++i`（前置加加）先自增 1 再返回，`i++`（后置加加）先返回 `i` 再自增 1。

前置加加不会产生临时对象，后置加加必须产生临时对象，临时对象会导致效率降低

`++i` 实现：

```
int& int::operator++ (){
    *this += 1;
    return *this;
}
```

`i++` 实现：

```
const int int::operator++ (int) {
    int oldValue = *this;
    ++ (*this);
    return oldValue;
}
```

8、简单说一下 `printf` 实现原理？

在 C/C++ 中，对函数参数的扫描是从后向前的。C/C++ 的函数参数是通过压入堆栈的方式来给函数传参数的（堆栈是一种先进后出的数据结构）。

最先压入的参数最后出来，在计算机的内存中，数据有2块，一块是堆，一块是栈（函数参数及局部变量在这里），而栈是从内存的高地址向低地址生长的，控制生长的就是堆栈指针了，最先压入的参数是在最上面，就是说在所有参数的最后面，最后压入的参数在最下面，结构上看起来是第一个，所以最后压入的参数总是能够被函数找到。

因为它就在堆栈指针的上方。printf的第一个被找到的参数就是那个字符指针，就是被双引号括起来的那一部分，函数通过判断字符串里控制参数的个数来判断参数个数及数据类型，通过这些就可算出数据需要的堆栈指针的偏移量了。

9、讲讲大端小端，如何检测

大端模式：是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址端。

小端模式，是指数据的高字节保存在内存的高地址中，低位字节保存在内存的低地址端。

直接读取存放在内存中的十六进制数值，取低位进行值判断

```
int a = 0x12345678;
int *c = &a;
c[0] == 0x12 大端模式
c[0] == 0x78 小端模式
```

用共同体来进行判断

union 共同体所有数据成员是共享一段内存的，后写入的成员数据将覆盖之前的成员数据，成员数据都有相同的首地址。Union 的大小为最大数据成员的大小。

union 的成员数据共用内存，并且首地址都是低地址首字节。Int i=1时：大端存储1放在最高位，小端存储1放在最低位。当读取char ch时，是最低地址首字节，大小端会显示不同的值。

```
#include <stdio.h>
int main() {
    union {
        int a; //4 bytes
        char b; //1 byte
    } data;
```

```
data.a = 1; //占4 bytes, 十六进制可表示为 0x 00 00 00 01

//b因为是char型只占1Byte, a因为是int型占4Byte
//所以, 在联合体data所占内存中, b所占内存等于a所占内存的低地址部分
if(1 == data.b) {
    //走到这里意味着说明a的低字节, 被取给到了b
    //即a的低字节存在了联合体所占内存的(起始)低地址, 符合小端模式特征
    printf("Little_Endian\n");
} else {
    printf("Big_Endian\n");
}
return 0;
}
```

10、分别写出 bool, int, float, 指针类型的变量 a 与“零”的比较语句。

```
bool: if ( !a ) or if(a)

int: if ( a == 0)

float: const EXPRESSION EXP = 0.000001 if ( a <= EXP && a >= -EXP)

pointer : if ( a != NULL) or if(a == NULL)
```

无论是 float 还是 double 类型的变量, 都有精度限制。所以一定要避免将浮点变量用“==”或“!=”与数字比较, 应该设法转化成“>=”或“<=”形式。

11、回调函数的作用

当发生某种事件时, 系统或其他函数将会自动调用你定义的一段函数;

回调函数就相当于一个中断处理函数, 由系统在符合你设定的条件时自动调用。为此, 你需要做三件事: 1, 声明; 2, 定义; 3, 设置触发条件, 就是在你的函数中把你的回调函数名称转化为地址作为一个参数, 以便于系统调用;

回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用为调用它所指向的函数时，我们就说这是回调函数；

因为可以把调用者与被调用者分开。调用者不关心谁是被调用者，所有它需知道的，只是存在一个具有某种特定原型、某些限制条件（如返回值为int）的被调用函数。

哈喽，我是小贺哥，就爱分享编程知识，如果觉得文章对你有帮助，别忘记关注我哦！



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复 「大礼包」
白嫖编程学习资料

关注公众号回复 「加群」 拉
你进百人技术交流群

七、C++11 新特性

C++11 的特性主要包括下面几个方面：

- 提高运行效率的语言特性：右值引用、泛化常量表达式
- 原有语法的使用性增强：初始化列表、统一的初始化语法、类型推导、范围 for 循环、Lambda 表达式、final 和 override、构造函数委托
- 语言能力的提升：空指针 nullptr、default 和 delete、长整数、静态 assert
- C++ 标准库的更新：智能指针、正则表达式、哈希表等

1、空指针 nullptr

nullptr 出现的目的是为了替代 NULL。

在某种意义上来说，传统 C++ 会把 NULL、0 视为同一种东西，这取决于编译器如何定义 NULL，有些编译器会将 NULL 定义为 ((void*)0)，有些则会直接将其定义为 0。C++ 不允许直接将 void * 隐式转换到其他类型，但如果 NULL 被定义为 ((void*)0)，那么当编译 char *ch = NULL; 时，NULL 只好被定义为 0。而这依然会产生问题，将导致了 C++ 中重载特性会发生混乱，考虑：

```
void func(int);
void func(char *);
```

对于这两个函数来说，如果 NULL 又被定义为了 0 那么 func(NULL) 这个语句将会去调用 func(int)，从而导致代码违反直观。

为了解决这个问题，C++11 引入了 nullptr 关键字，专门用来区分空指针、0。nullptr 的类型为 nullptr_t，能够隐式的转换为任何指针或成员指针的类型，也能和他们进行相等或者不等的比较。

当需要使用 NULL 时候，养成直接使用 nullptr 的习惯。

2、Lambda 表达式

Lambda 表达式实际上就是提供了一个类似匿名函数的特性，而匿名函数则是在需要一个函数，但是又不想费力去命名一个函数的情况下去使用的。

利用 lambda 表达式可以编写内嵌的匿名函数，用以替换独立函数或者函数对象，并且使代码更可读。

从本质上讲，lambda 表达式只是一种语法糖，因为所有其能完成的工作都可以用其它稍微复杂的代码来实现，但是它简便的语法却给 C++ 带来了深远的影响。

从广义上说，lambda 表达式产生的是函数对象。在类中，可以重载函数调用运算符()，此时类的对象可以将具有类似函数的行为，我们称这些对象为 **函数对象（Function Object）** 或者 **仿函数（Functor）**。相比 lambda 表达式，函数对象有自己独特的优势。

`lambda` 表达式一般是从方括号`[]`开始，然后结束于花括号`{}`，花括号里面就像定义函数那样，包含了 `lambda` 表达式体，一个最简单的例子如下：

```
// 定义简单的lambda表达式
auto basicLambda = [] { cout << "Hello, world!" << endl; };
basicLambda(); // 输出: Hello, world!
```

上面是最简单的 `lambda` 表达式，没有参数。如果需要参数，那么就要像函数那样，放在圆括号里面，如果有返回值，返回类型要放在`->`后面，即拖尾返回类型，当然你也可以忽略返回类型，**lambda会帮你自动推断出返回类型**：

```
// 指明返回类型，托尾返回类型
auto add = [](int a, int b) -> int { return a + b; };
// 自动推断返回类型
auto multiply = [](int a, int b) { return a * b; };

int sum = add(2, 5); // 输出: 7
int product = multiply(2, 5); // 输出: 10
```

最前边的 `[]` 是 `lambda` 表达式的一个很重要的功能，就是 闭包。

先说明一下 `lambda` 表达式的大致原理：每当你定义一个 `lambda` 表达式后，编译器会自动生成一个匿名类（这个类当然重载了 `()` 运算符），我们称为闭包类型（closure type）。

那么在运行时，这个 `lambda` 表达式就会返回一个匿名的闭包实例，其实一个右值。所以，我们上面的 `lambda` 表达式的结果就是一个个闭包实例。

闭包的一个强大之处是其可以通过传值或者引用的方式捕捉其封装作用域内的变量，前面的方括号就是用来定义捕捉模式以及变量，我们又将其称为 `lambda` 捕捉块。例子如下：

```

int main() {
    int x = 10;

    auto add_x = [x](int a) { return a + x; };           // 复制捕捉x, lambda
表达式无法修改此变量
    auto multiply_x = [&x](int a) { return a * x; }; // 引用捕捉x, lambda
表达式可以修改此变量

    cout << add_x(10) << " " << multiply_x(10) << endl;
    // 输出: 20 100
    return 0;
}

```

捕获的方式可以是引用也可以是复制，但是具体说来会有以下几种情况来捕获其所在作用域中的变量：

- []: 默认不捕获任何变量；
- [=]: 默认以值捕获所有变量；
- [&]: 默认以引用捕获所有变量；
- [x]: 仅以值捕获x，其它变量不捕获；
- [&x]: 仅以引用捕获x，其它变量不捕获；
- [=, &x]: 默认以值捕获所有变量，但是x是例外，通过引用捕获；
- [&, x]: 默认以引用捕获所有变量，但是x是例外，通过值捕获；
- [this]: 通过引用捕获当前对象（其实是复制指针）；
- [*this]: 通过传值方式捕获当前对象；

而 **lambda 表达式一个更重要的应用是其可以用于函数的参数，通过这种方式可以实现回调函数。**其实，最常用的是在STL算法中，比如你要统计一个数组中满足特定条件的元素数量，通过 **lambda 表达式给出条件，传递给 count_if 函数：**

```
int val = 3;
vector<int> v {1, 8, 5, 3, 6, 10};

int count = std::count_if(v.begin(), v.end(), [val](int x) { return x > val; });
// v中大于3的元素数量
```

最后给出 `lambda` 表达式的完整语法：

```
[ capture-list ] ( params ) mutable(optional) constexpr(optional)(c++17)
exception attribute -> ret { body }

// 可选的简化语法
[ capture-list ] ( params ) -> ret { body }
[ capture-list ] ( params ) { body }
[ capture-list ] { body }
```

- `capture-list`: 捕捉列表，这个不用多说，前面已经讲过，它不能省略；
- `params`: 参数列表，可以省略（但是后面必须紧跟函数体）；
- `mutable`: 可选，将 `lambda` 表达式标记为 `mutable` 后，函数体就可以修改传值方式捕获的变量；
- `constexpr`: 可选，C++17，可以指定 `lambda` 表达式是一个常量函数；
- `exception`: 可选，指定 `lambda` 表达式可以抛出的异常；
- `attribute`: 可选，指定 `lambda` 表达式的特性；
- `ret`: 可选，返回值类型；
- `body`: 函数执行体。

3、右值引用

C++03 及之前的 standard 中，右值是不允许被改变的，实践中也通常使用 `const T&` 的方式传递右值。然而这是效率低下的做法，例如：

```
Person get(){
    Person p;
    return p;
}

Person p = get();
```

上述获取右值并初始化 p 的过程包含了 Person 的3个构造过程和2个析构过程。这是 C++ 广受诟病的一点，但C++11 的右值引用特性允许我们对右值进行修改。借此可以实现 **move语义**，即从右值中直接拿数据过来初始化或修改左值，而不需要重新构造左值后再析构右值。一个 move 构造函数是这样声明的：

```
class Person{
public:
    Person(Person&& rhs){...}
    ...
};
```

4、泛化的常量表达式

还记得刚开始学习 C++ 给你的苦恼吗？你看：

```
int N = 5;
int arr[N];
```

编译器会报错： error: variable length array declaration not allowed at file scope int arr[N]；，但 N 就是5，不过编译器不知道这一点，于是我们需要声明为 const int N = 5 才可以。但C++11的**泛化常数**给出了解决方案：

```
constexpr int N = 5;      // N 变成了一个只读的值
int arr[N];              // OK
```

constexpr 告诉编译器这是一个编译期常量，甚至可以把一个函数声明为编译期常量表达式。

```
constexpr int getFive(){ return 5; }
int arr[getFive() + 1];
```

5、初始化列表

接下来几个特性属于原有语言特性的使用性增强。这意味着这些操作原来也是可以实现的，不过现在语法上更加简洁。比如首先要介绍的**初始化列表**。

而 C++11 提供了 `initializer_list` 来接受变长的对象初始化列表：

```
class A{
public:
    A(std::initializer_list<int> list);
};

A a = {1, 2, 3};
```

注意**初始化列表**特性只是现有语法增强，并不是提供了动态的可变参数。该列表只能静态地构造。

6、统一的初始化语法

不同的数据类型具有不同的初始化语法。如何初始化字符串？如何初始化数组？如何初始化多维数组？如何初始化对象？

C++11给出了统一的初始化语法：均可使用“{}-初始化变量列表”：

```
X x1 = X{1,2};
X x2 = {1,2};      // 此处的'='可有可无
X x3{1,2};
X* p = new X{1,2};

struct D : X {
    D(int x, int y) : X{x,y} { /* ... */ };
};
```

```
struct S {  
    int a[3];  
    // 对于旧有问题的解决方案  
    S(int x, int y, int z) :a{x,y,z} { /* ... */ };  
};
```

7、类型推导

C++ 提供了 `auto` 和 `decltype` 来静态推导类型，在我们知道类型没有问题但又不想完整地写出类型的时候，便可以使用静态类型推导。

```
for(vector<int>::const_iterator it = v.begin(); it != v.end(); ++it);  
// 可以改写为  
for(auto it = v.begin(); it != v.end(); ++it);
```

虽然写起来和动态语言（如JavaScript的 `var`）很像，但C++仍然是强类型的，会执行静态类型检查的语言。这只是语法上的简化，并未改变C++的静态类型检查。

`decltype` 用于获取一个表达式的类型，而不对表达式进行求值（类似于 `sizeof`）。`decltype(e)` 规则如下：

- 若 `e` 为一个无括号的变量、函数参数、类成员，则返回类型为该变量/参数/类成员在源程序中的声明类型；
- 否则的话，根据表达式的值分类（value categories），设设 `T` 为 `e` 的类型：
的类型：
 - 若 `e` 是一个左值（lvalue，即“可寻址值”），返回 `T&`；
 - 若 `e` 是一个临时值（xvalue），则返回值为 `T&&`；
 - 若 `e` 是一个纯右值（prvalue），则返回值为 `T`。

```
const std::vector<int> v(1);
const int&& foo();           // 返回临终值：生命周期已结束但内存还未拿走

auto a = v[0];                // a 为 int
decltype(v[0]) b = 0;         // b 为 const int&
                             // 即 vector<int>::operator[](size_type) const 的
返回值类型
auto c = 0;                   // c, d 均为 int
auto d = c;
decltype(c) e;                // e 为 int, 即 c 的类型
decltype((c)) f = e;          // f 为 int&, 因为 c 是左值
decltype(0) g;                // g 为 int, 因为 0 是右值
```

8、基于范围的for循环

Boost 中定义了很多"范围", 很多标准库函数都使用了范围风格的实现。这一概念被C++11提了出来:

```
int arr[5];
std::vector<int> v;

for(int x: arr);
for(const int& x: arr);
for(int x: v);
for(auto &x: v);
```

9、构造函数委托

在 C# 和 Java 中, 一个构造函数可以调用另一个来实现代码复用, 但 C++一直不允许这样做。

现在可以了, 这使得构造函数可以在同一个类中一个构造函数调用另一个构造函数, 从而达到简化代码的目的:

```
class myBase {
    int number; string name;
    myBase( int i, string& s ) : number(i), name(s){}
public:
    myBase( )           : myBase( 0, "invalid" ){}
    myBase( int i )     : myBase( i, "guest" ){}
    myBase( string& s ) : myBase( 1, s ){ PostInit(); }
};
```

10、final 和 override

C++ 借由虚函数实现运行时多态，但 C++ 的虚函数又很多脆弱的地方：

- 无法禁止子类重写它。可能到某一层级时，我们不希望子类继续来重写当前虚函数了。
- 容易不小心隐藏父类的虚函数。比如在重写时，不小心声明了一个签名不一致但有同样名称的新函数。

C++11 提供了 `final` 来禁止虚函数被重写/禁止类被继承，`override` 来显示地重写虚函数。这样编译器给我们不小心的行为提供更多有用的错误和警告。

```
struct Base1 final { };
struct Derived1 : Base1 {}; // 编译错：Base1不允许被继承

struct Base2 {
    virtual void f1() final;
    virtual void f2();
};

struct Derived2 : Base2 {
    virtual void f1(); // 编译错：f1不允许重写
    virtual void f2(int) override; // 编译错：父类中没有 void f2(int)
};
```

11、default 和 delete

我们知道编译器会为类自动生成一些方法，比如构造和析构函数（完整的列表见 Effective C++: Item 5）。

现在我们可以显式地指定和禁止这些自动行为了。

```
struct classA {  
    classA() = default;      // 声明一个自动生成的函数  
    classA(T value);  
    void *operator new(size_t) = delete;    // 禁止生成new运算符  
};
```

在上述 classA 中定义了 classA(T value) 构造函数，因此编译器不会默认生成一个无参数的构造函数了，如果我们需要可以手动声明，或者直接 = default。

12、静态 assertion

C++ 提供了两种方式来 assert：一种是 assert 宏，另一种是预处理指令 #error。前者在运行期起作用，而后者是预处理期起作用。它们对模板都不好使，因为模板是编译期的概念。`static_assert` 关键字的使用方式如下：

```
template< class T >  
struct Check {  
    static_assert( sizeof(int) <= sizeof(T), "T is not big enough!" );  
};
```

13、智能指针

接下来介绍 C++11 对于 C++ 标准库的变更。C++11 把 TR1 并入了进来，废弃了 C++98 中的 `auto_ptr`，同时将 `shared_ptr` 和 `unique_ptr` 并入 `std` 命名空间。

智能指针在 [Effective C++: Item 13] 中已经有不少讨论了。这里给一个例子：

```
int main(){
    std::shared_ptr<double> p_first(new double);
    {
        std::shared_ptr<double> p_copy = p_first;
        *p_copy = 21.2;
    } // p_copy 被销毁, 里面的 double 还有一个引用因此仍然保持
    return 0; // p_first 及其里面的 double 销毁
}
```

14、正则表达式

这个任何一门现代的编程语言都会提供的特性终于进标准:

```
const char *reg_esp = "[ ,.\t\n;:]";
std::regex rx(reg_esp) ;
std::cmatch match ;
const char *target = "Polytechnic University of Turin" ;

if( regex_search( target, match, rx ) ) {
    const size_t n = match.size();
    for( size_t a = 0 ; a < n ; a++ ) {
        string str( match[a].first, match[a].second ) ;
        cout << str << "\n" ;
    }
}
```

上述代码段来自Wikipedia: <https://zh.wikipedia.org/wiki/C%2B%2B11>

15、增强的元组

在 C++ 中本已有一个 `pair` 模板可以定义二元组, C++11 更进一步地提供了边长参数的 `tuple` 模板:

```
typedef std::tuple< int , double, string > tuple_1 t1;
typedef std::tuple< char, short , const char * > tuple_2 t2 ('X', 2,
"Hello! ");
t1 = t2 ;           // 隐式类型转换
```

16、哈希表

C++ 的 `map`，`multimap`，`set`，`multiset` 使用红黑树实现，插入和查询都是 $O(\lg n)$ 的复杂度，

但 C++11 为这四种模板类提供了（底层哈希实现）以达到 $O(1)$ 的复杂度：

散列表类型	有无关系值	接受相同键值
<code>std::unordered_set</code>	否	否
<code>std::unordered_multiset</code>	否	是
<code>std::unordered_map</code>	是	否
<code>std::unordered_multimap</code>	是	是

参考：<https://harttle.land/2015/10/08/cpp11.html>

哈喽，我是小贺哥，就爱分享编程知识，如果觉得文章对你有帮助，别忘记关注我哦！



一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复「大礼包」
白嫖编程学习资料

关注公众号回复「加群」拉
你进百人技术交流群

扫一扫，关注「herongwei」公众号

八、数据结构和算法

1、十大排序算法及其时间和空间复杂度

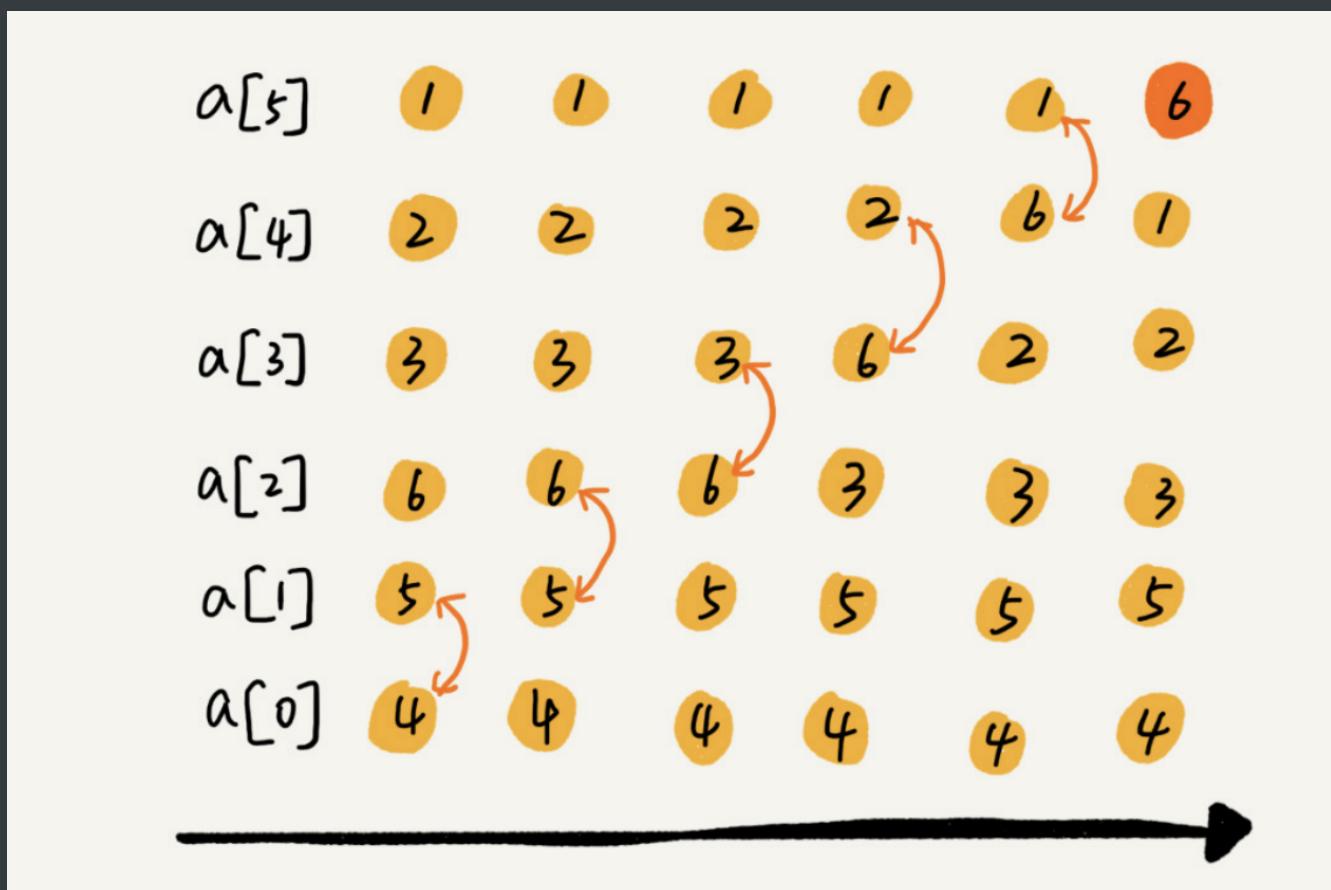
排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

(1) 冒泡排序

算法描述：

- 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
- 针对所有的元素重复以上的步骤，除了最后一个；
- 重复步骤 1~3，直到排序完成。

用一个例子，带你看下冒泡排序的整个过程。我们要对一组数据 4, 5, 6, 3, 2, 1，从小到大进行排序。第一次冒泡操作的详细过程就是这样



可以看出，经过一次冒泡操作之后，6 这个元素已经存储在正确的位置上。要想完成所有数据的排序，我们只要进行 6 次这样的冒泡操作就行了。

冒泡次数	冒泡后结果	有序度
初始状态	4 5 6 3 2 1	3
第一次冒泡	4 5 3 2 1 6	6
第二次冒泡	4 3 2 1 5 6	9
第三次冒泡	3 2 1 4 5 6	12
第四次冒泡	2 1 3 4 5 6	14
第五次冒泡	1 2 3 4 5 6	15

下面代码中 `std::swap` 函数的源代码如下，可以看到有三个赋值操作：

```
template<class T>
void swap(T &a, T &b) {
    T temp = a;
    a = b;
    b = temp;
}
```

```
void BubbleSort(std::vector<int> &nums, int n) {
    if (n <= 1) return;
    bool is_swap;
    for (int i = 1; i < n; ++i) {
        is_swap = false;
        //设定一个标记，若为false，则表示此次循环没有进行交换，也就是待排序列已经有
        //序，排序已经完成。
        for (int j = 1; j < n - i + 1; ++j) {
            if (nums[j] < nums[j-1]) {
                std::swap(nums[j], nums[j-1]);
                is_swap = true;//表示有数据交换
            }
        }
        if (!is_swap) break;//没有数据交集，提前退出
    }
}
```

```
    }  
}
```

```
int main() {  
    int a[] = {34,66,2,5,95,4,46,27};  
    BubbleSort(a,sizeof(a)/sizeof(int)); //cout => 2 4 5 27 34 46 66 95  
    return 0;  
}
```

(2) 插入排序

算法描述：分为已排序和未排序 初始已排序区间只有一个元素 就是数组第一个 遍历未排序的每一个元素在已排序区间里找到合适的位置插入并保证数据一直有序。



```
void InsertSort(std::vector<int> &nums, int n) {  
    if (n <= 1) return;  
    for(int i = 0; i < n; ++i) {  
        for (int j = i; j > 0 && nums[j] < nums[j-1]; --j) {  
            std::swap(nums[j], nums[j-1]);  
        }  
    }  
}
```

```
int main() {  
    std::vector<int> nums = {4, 6, 5, 3, 2, 1};  
    InsertSort(nums, 6); // cout => 1, 2, 3, 4, 5, 6  
    return 0;  
}
```

(3) 选择排序

算法描述：分已排序区间和未排序区间。每次会从未排序区间中找到最小的元素，将其放到已排序区间的末尾。

选择排序原理示意图



```
void SelectSort(std::vector<int> &nums, int n) {  
    if (n <= 1) return;  
    int mid;  
    for (int i = 0; i < n - 1; ++i) {  
        mid = i;  
        for (int j = i + 1; j < n; ++j) {  
            if (nums[j] < nums[mid]) {  
                mid = j;  
            }  
        }  
        std::swap(nums[mid], nums[i]);  
    }  
}
```

```
int main() {  
    std::vector<int> nums = {4, 6, 5, 3, 2, 1};  
    SelectSort(nums, 6); // cout => 1, 2, 3, 4, 5, 6  
    return 0;  
}
```

【时间,空间复杂度/是否稳定?】

首先, 选择排序空间复杂度为 $O(1)$, 是一种原地排序算法。选择排序的最好情况时间复杂度、最坏情况和平均情况时间复杂度都为 $O(n)$ 。你可以自己来分析看看。

那选择排序是稳定的排序算法吗? 答案是否定的, 选择排序是一种不稳定的排序算法。从图中, 你可以看出来, 选择排序每次都要找剩余未排序元素中的最小值, 并和前面的元素交换位置, 这样破坏了稳定性

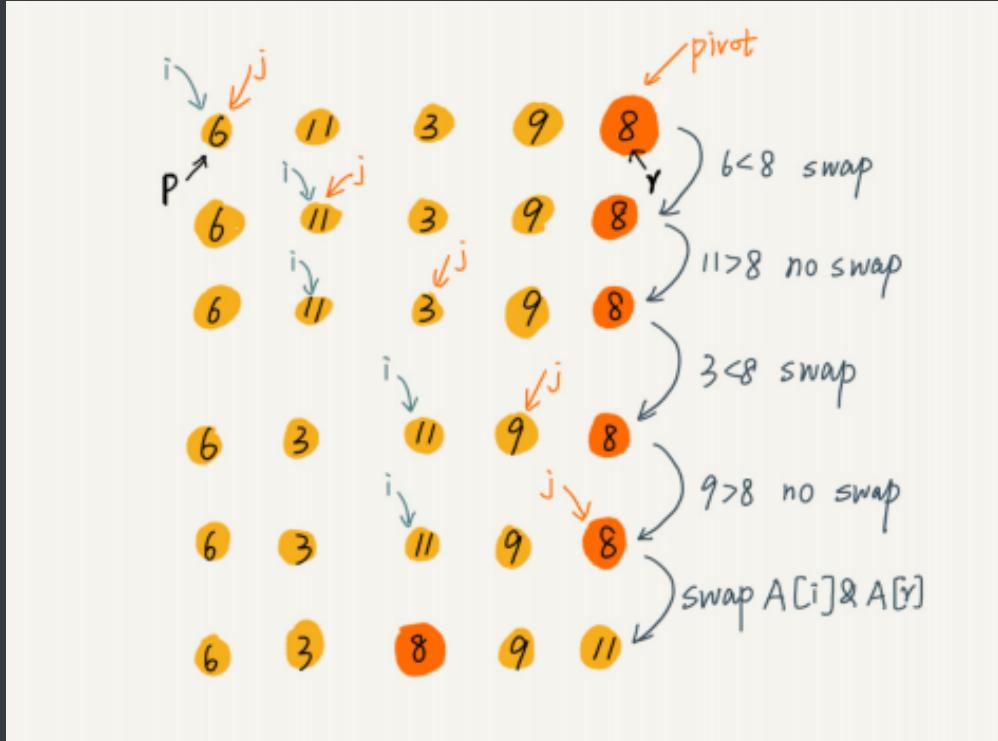
【思考】冒泡排序和插入排序的时间复杂度都是 $O(n)$, 都是原地排序算法, 为什么插入排序要比冒泡排序更受欢迎呢?

【思路】冒泡排序不管怎么优化, 元素交换的次数是一个固定值, 是原始数据的逆序度。插入排序是同样的, 不管怎么优化, 元素移动的次数也等于原始数据的逆序度。但是, 从代码实现上来看, 冒泡排序的数据交换要比插入排序的数据移动要复杂, 冒泡排序需要 3 个赋值操作, 而插入排序只需要 1 个。把执行一个赋值语句的时间粗略地计为单位时间, 处理相同规模的数, 插入排序比冒泡排序减少三倍的单位时间!

(4) 快排

算法描述: 先找到一个枢纽; 在原来的元素里根据这个枢纽划分 比这个枢纽小的元素排前面; 比这个枢纽大的元素排后面; 两部分数据依次递归排序下去直到最终有序。

手撕快排讲解: <https://www.bilibili.com/video/BV1mE411M7SH>



```

void QuickSort(std::vector<int> &nums, int l, int r) {
    if (l + 1 >= r) return;
    int first = l, last = r - 1, key = nums[first];
    while (first < last) {
        while (first < last && nums[last] >= key) last--;//右指针 从右向左
        //扫描 将小于piv的放到左边
        nums[first] = nums[last];
        while (first < last && nums[first] <= key) first++;//左指针 从左向右
        //扫描 将大于piv的放到右边
        nums[last] = nums[first];
    }
    nums[first] = key;//更新piv
    quick_sort(nums, l, first);//递归排序 //以L为中间值，分左右两部分递归调用
    quick_sort(nums, first + 1, r);
}

```

```

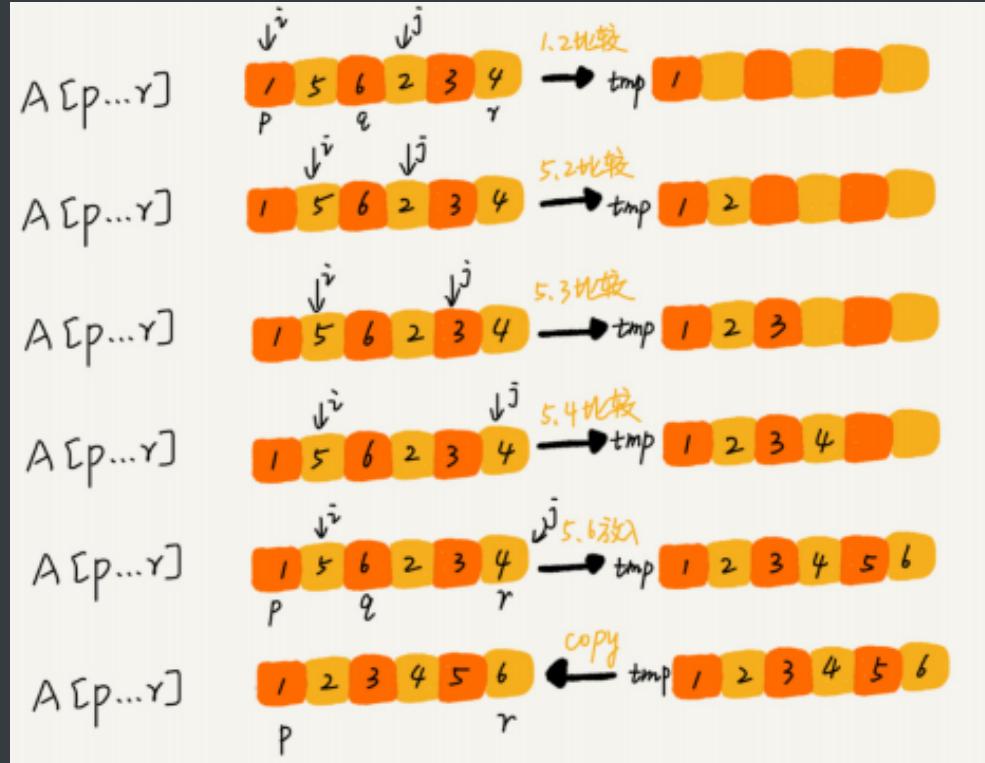
int main() {
    int a[] = {0,34,66,2,5,95,4,46,27};
    QuickSort(a, 0, sizeof(a)/sizeof(int));
    for(int i=0; i<=8; ++i) {
        std::cout<<a[i]<<" ";
        // print => 0 2 4 5 27 34 46 66 95
    }
    std::cout<<endl;
    return 0;
}

```

(5) 归并排序

算法描述：归并排序是一个稳定的排序算法，归并排序的时间复杂度任何情况下都是 $O(n \log n)$ ，归并排序不是原地排序算法

用两个游标 i 和 j ，分别指向 $A[p \dots q]$ 和 $A[q+1 \dots r]$ 的第一个元素。比较这两个元素 $A[i]$ 和 $A[j]$ ，如果 $A[i] \leq A[j]$ ，我们就把 $A[i]$ 放入到临时数组 tmp ，并且 i 后移一位，否则将 $A[j]$ 放入到数组 tmp ， j 后移一位。



```

void mergeCount(int a[], int L, int mid, int R) {
    int *tmp = new int[L+mid+R];

```

```
int i=L;
int j=mid+1;
int k=0;
while( i<=mid && j<=R ) {
    if(a[i] < a[j])
        tmp[k++] = a[i++];
    else
        tmp[k++] = a[j++];
}
///判断哪个子数组中有剩余的数据
while( i<=mid )
    tmp[k++] = a[i++];
while( j<=R)
    tmp[k++] = a[j++];
/// 将 tmp 中的数组拷贝回 A[p...r]
for(int p=0; p<k; ++p)
    a[L+p] = tmp[p];
delete tmp;
}

void mergeSort(int a[],int L,int R) {
    ///递归终止条件 分治递归
    /// 将 A[L...m] 和 A[m+1...R] 合并为 A[L...R]
    if( L>=R ) { return; }
    int mid = L + (R - L)/2;
    mergeSort(a,L,mid);
    mergeSort(a,mid+1,R);
    mergeCount(a,L,mid,R);
}
```

```

int main() {
    int a[] = {0,34,66,2,5,95,4,46,27};
    mergeSort(a, 0, sizeof(a)/sizeof(int));
    for(int i=0; i<=8; ++i) {
        std::cout<<a[i]<<" "; // print => 0 2 4 5 27 34 46 66 95
    }
    std::cout<<endl;
    return 0;
}

```

(6) 堆排序

算法描述：利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。堆排序可以用到上一次的排序结果，所以不像其他一般的排序方法一样，每次都要进行 $n-1$ 次的比较，复杂度为 $O(n \log n)$ 。

算法步骤：

- 1、利用给定数组创建一个堆 $H[0..n-1]$ （我们这里使用最小堆），输出堆顶元素
- 2、以最后一个元素代替堆顶，调整成堆，输出堆顶元素
- 3、把堆的尺寸缩小 1
- 4、重复步骤 2，直到堆的尺寸为 1

建堆：将数组原地建成一个堆，不借助额外的空间，采用从上往下的堆化（对于完全二叉树来说，下标是 $n/2+1$ 到 n 的节点都是叶子节点，不需要堆化）。

排序：“删除堆顶元素”：当堆顶元素移除之后，把下标为 n 的元素放到堆顶，然后在通过堆化的方法，将剩下的 $n - 1$ 个元素重新构建成堆，堆化完成之后，在取堆顶的元素，放到下标为 $n-1$ 的位置，一直重复这个过程，直到最后堆中只剩下标 1 的一个元素。

```
/*
```

优点： $O(n \log n)$ ，原地排序，最大的特点：每个节点的值大于等于（或小于等于）其子树节点

缺点：相比于快排，堆排序数据访问的方式没有快排友好；数据的交换次数要多于快排。

```
/*
void HeapSort(int a[], int n) {
    for(int i=n/2; i>=1; --i) {
        Heapify(a, n, i);
    }
    int k = n;
    while( k > 1) {
        swap(a[1],a[k]);
        --k;
        Heapify(a,k,1);
    }
}

void Heapify(int a[], int n, int i) {
    while (1) {
        int maxPos = i;
        if (i*2 <= n && a[i] < a[i*2]) { maxPos = i*2; }
        if (i*2+1 <= n && a[maxPos] < a[i*2+1]) { maxPos = i*2+1; }
        if (maxPos == i) break;
        std::swap(a[i], a[maxPos]);
        i = maxPos;
    }
}
void
```

(7) 桶排序

算法描述：将数组分到有限数量的桶里。每个桶再个别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排序）。

```
void bucketSort(std::vector<int>& nums) {
    if (nums.empty()) return ;
    int low = *std::min_element(nums.begin(), nums.end());
    int high = *std::max_element(nums.begin(), nums.end());
    int n = high - low + 1;
    std::vector<int> buckets(n);
```

```

std::vector<int> res;
for (auto x : nums) ++buckets[x - low];
for (int i = 0; i < n; ++i) { //实现了按桶下标从小到达输出
    for (int j = 0; j < buckets[i]; ++j) { //重复的输出
        res.push_back(i + low);
    }
}
for (int i=0; i<res.size(); ++i) {
    std::cout<<res[i]<<" ";
}

```

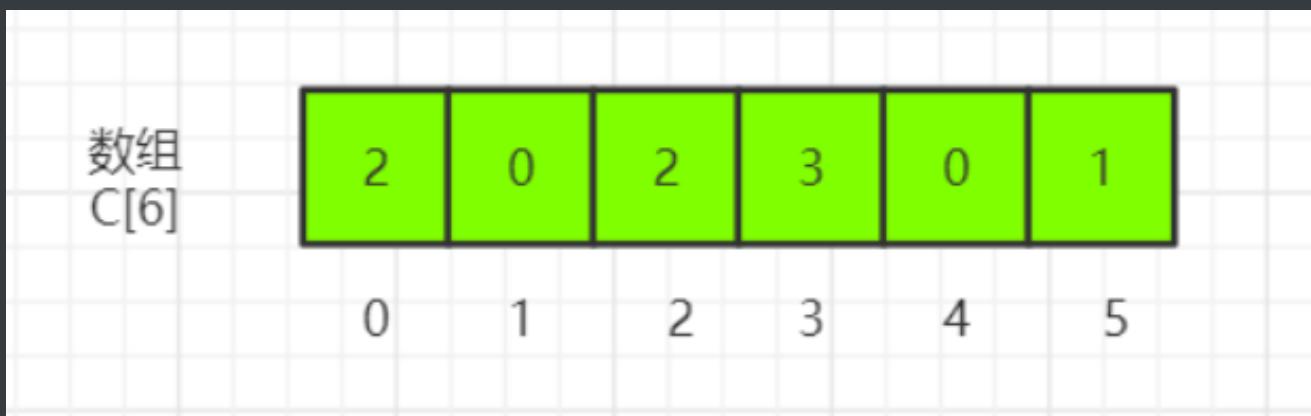
(8) 计数排序

扩展：如果在面试中有面试官要求你写一个 $O(n)$ 时间复杂度的排序算法，可不要傻乎乎的说这不可能！虽然前面基于比较的排序的下限是 $O(n \log n)$ 。但是确实也有线性时间复杂度的排序，只不过有前提条件，就是待排序的数要满足一定的范围的整数，而且计数排序需要比较多的辅助空间。

算法描述：其基本思想是，用待排序的数作为计数数组的下标，统计每个数字的个数。然后依次输出即可得到有序序列。

假设有 8 个考生，分数在 0 到 5 分之间。这 8 个考生的成绩我们放在一个数组 $A[8]$ 中，它们分别是：2, 5, 3, 0, 2, 3, 0, 3。

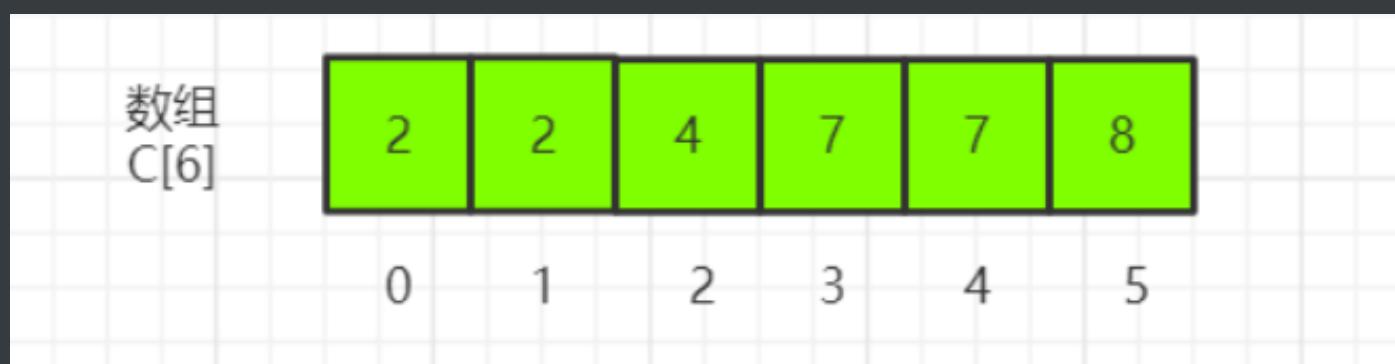
考生的成绩从 0 到 5 分，我们使用大小为 6 的数组 $C[6]$ 表示桶，其中下标对应分数。不过， $C[6]$ 内存储的并不是考生，而是对应的考生个数。像我刚刚举的那个例子，我们只需要遍历一遍考生分数，就可以得到 $C[6]$ 的值。



这是我们的数组，从图中可以看出，分数为 3 分的考生有 3 个，小于 3 分的考生有 4 个，所以，成绩为 3 分的考生在排序之后的有序数组 R[8] 中，会保存下标 4, 5, 6 的位置。

那我们如何快速计算出，每个分数的考生在有序数组中对应的存储位置呢？

我们对 C[6] 数组顺序求和，C[6] 存储的数据就变成了下面这样子。C[k] 里存储小于等于分数 k 的考生个数。



我们从后到前依次扫描数组 A。比如，当扫描到 3 时，我们可以从数组 C 中取出下标为 3 的值 7，也就是说，到目前为止，包括自己在内，分数小于等于 3 的考生有 7 个，也就是说 3 是数组 R 中的第 7 个元素（也就是数组 R 中下标为 6 的位置）。当 3 放入到数组 R 中后，小于等于 3 的元素就只剩下了 6 个了，所以相应的 C[3] 要减 1，变成 6。

以此类推，当我们扫描到第 2 个分数为 3 的考生的时候，就会把它放入数组 R 中的第 6 个元素的位置（也就是下标为 5 的位置）。当我们扫描完整个数组 A 后，数组 R 内的数据就是按照分数从小到大有序排列的了。

注意：计数排序只能用在数据范围不大的场景中，如果数据范围 k 比要排序的数据 n 大很多，就不适合用计数排序了。而且，计数排序只能给非负整数排序，如果要排序的数据是其他类型的，要将其在不改变相对大小的情况下，转化为非负整数。

```
void countSort(int *a, int n){  
    int maxV=a[0];  
    for(int i=1; i<n; ++i){  
        maxV=max(maxV,a[i]);  
    }  
    int c[maxn];  
    memset(c,0,sizeof(c));  
    for(int i=0; i<n; ++i){
```

```

        c[a[i]]++;
    }

    for(int i=1; i<=maxV; ++i){
        c[i]+=c[i-1];
    }

    int r[maxn];
    memset(r, 0, sizeof(r));

    for(int i=n-1; i>=0; --i){
        int index = c[a[i]]-1;
        r[index]=a[i];
        c[a[i]]--;
    }

    for(int i=0; i<n; ++i){
        a[i]=r[i];
    }
}

```

(9) 基数排序

算法描述：基数排序对要排序的数据是有要求的，需要可以分割出独立的“位”来比较，而且位之间有递进的关系，如果 a 数据的高位比 b 数据大，那剩下的低位就不用比较了。除此之外，每一位的数据范围不能太大，要可以用线性排序算法来排序，否则，基数排序的时间复杂度就无法做到 $O(n)$ 了。

基数排序相当于通过循环进行了多次桶排序。

```

int getDigit(int x,int d){
    int t[]={1,1,10,100};
    return (x/t[d])%10;
}

void RadixSort(int *a,int begin,int end,int d){
    const int radix = 10;
    int c[maxn];
    int bucket[maxn];
    for(int k=1; k<=d; ++k){

```

```

memset(c, 0, sizeof(c));
for(int i=begin; i<=end; ++i){
    c[getDigit(a[i], k)]++; //计算i号桶里要放多少数
}
for(int i=1; i<radix; ++i) c[i] += c[i-1];
// 把数据依次装入桶 (注意: 装入时的分配技巧)
for(int i=end; i>=begin; --i){
    int j=getDigit(a[i], k); //求出关键码的第k位的数值, 例如: 576的第3
位是5
    bucket[c[j]-1]=a[i]; //放入对应的桶中(count[j]-1)表示第k位数值为j
    的桶底索引
    --c[j]; //当前第k位数值为j的桶底边界索引减一
}
for(int i=begin, j=0; i<=end; ++i, ++j){ // 从各个桶中收集数据
    a[i]=bucket[j];
}
}

```

(10) 希尔排序

算法描述：通过将比较的全部元素分为几个区域来提升插入排序的性能。这样可以让一个元素可以一次性地朝最终位置前进一大步。然后算法再取越来越小的步长进行排序，算法的最后一步就是普通的插入排序，但是到了这步，需排序的数据几乎是已排好的了。

```
// 希尔排序
void shellSort(vector<int>& nums) {
    for (int gap = nums.size() / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < nums.size(); ++i) {
            for (int j = i; j - gap >= 0 && nums[j - gap] > nums[j]; j -= gap)
            {
                swap(nums[j - gap], nums[j]);
            }
        }
    }
}
```

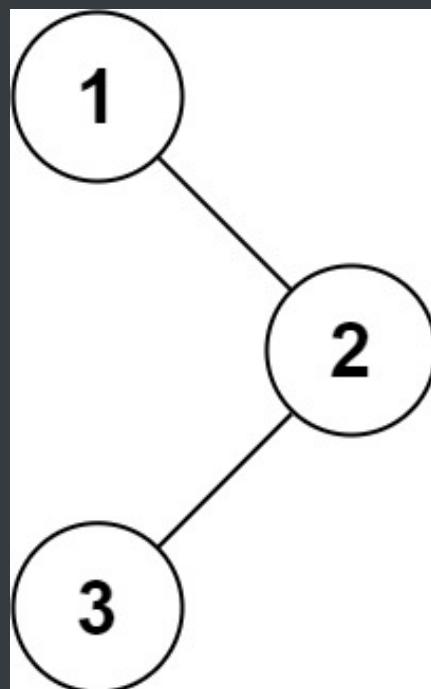
2、二叉树前中后遍历手撕代码（递归和非递归）

LeetCode 144. 二叉树的前序遍历

难度简单609

给你二叉树的根节点 `root`，返回它节点值的 **前序** 遍历。

示例 1：



输入: root = [1,null,2,3]

输出: [1,2,3]

示例 2:

输入: root = []

输出: []

示例 3:

输入: root = [1]

输出: [1]

【思路】

由于“中左右”的访问顺序正好符合根结点寻找子节点的顺序，因此每次循环时弹栈，输出此弹栈结点并将其右结点和左结点按照叙述顺序依次入栈。至于为什么要右结点先入栈，是因为栈后进先出的特性。右结点先入栈，就会后输出右结点。

初始化：

一开始让root结点先入栈，满足循环条件

步骤：

- 弹栈栈顶元素，同时输出此结点
- 当前结点的右结点入栈
- 当前结点的左结点入栈
- 重复上述过程
- 结束条件：
- 每次弹栈根结点后入栈子结点，栈为空时则说明遍历结束。

【代码】

```
//递归版
/* Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 * }
```

```

*     TreeNode *left;
*     TreeNode *right;
*     TreeNode() : val(0), left(nullptr), right(nullptr) {}
*     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
*     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),
left(left), right(right) {}
* };
*/
class Solution {
public:
    vector<int> ans;
    vector<int> preorderTraversal(TreeNode* root) {
        // 为空则直接返回
        if(root == NULL)
            return ans;
        ans.push_back(root->val);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
        return ans;
    }
}

```

```

//非递归
class Solution {
public:
    std::vector<int> preorderTraversal(TreeNode* root) {
        std::vector<int> res;
        if (!root) return res;
        stack<TreeNode*> st;
        TreeNode* node = root;
        while (!st.empty() || node) {
            while (node) {
                st.push(node);
                res.push_back(node->val);
                node = node->left;
            }

```

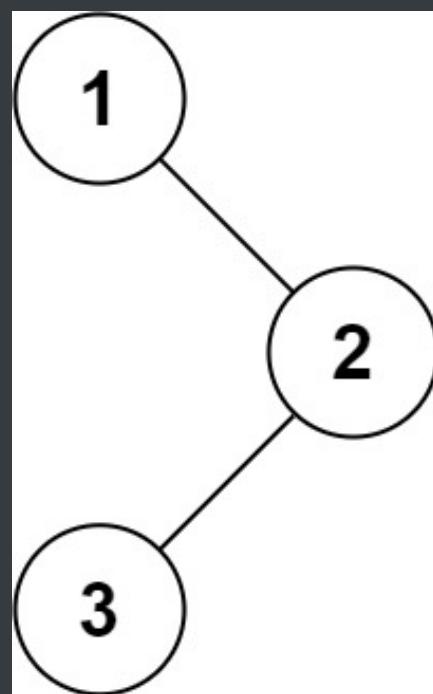
```
    node = st.top();
    st.pop();
    node = node->right;
}
return res;
};
```

LeetCode 94. 二叉树的中序遍历

难度简单1035

给定一个二叉树的根节点 `root`，返回它的 **中序** 遍历。

示例 1：



输入: `root = [1,null,2,3]`

输出: `[1,3,2]`

示例 2：

输入: root = []

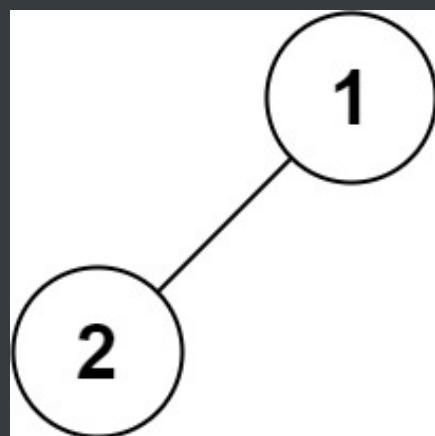
输出: []

示例 3:

输入: root = [1]

输出: [1]

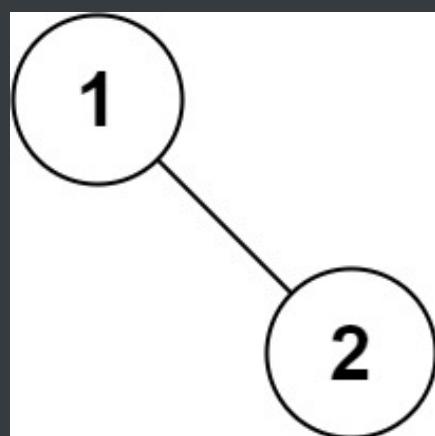
示例 4:



输入: root = [1,2]

输出: [2,1]

示例 5:



输入: root = [1,null,2]

输出: [1,2]

【思路】

中序遍历思路相较于前序遍历有很大的改变。前序遍历遇到根结点直接输出即可，但中序遍历“左中右”需先找到此根结点的左结点，因此事实上第一个被输出的结点会是整个二叉树的最左侧结点。

依据这一特性，我们每遇到一个结点，首先寻找其最左侧的子结点，同时用栈记录寻找经过的路径结点，这些是输出最左侧结点之后的返回路径。

之后每次向上层父结点返回，弹栈输出上层父结点的同时判断此结点是否含有右子结点，如果存在则此右结点入栈并到达新一轮循环，对此右结点也进行上述操作。

初始化：

curr定义为将要入栈的结点，初始化为root

top定义为栈顶的弹栈结点

步骤：

- 寻找当前结点的最左侧结点直到curr为空（此时栈顶结点即为最左侧结点）
 弹栈栈顶结点top并输出
- 判断top是否具有右结点，如果存在则令curr指向右结点，并在下一轮循环入栈
- 重复上述过程
- 结束条件：这里可以看到结束条件有两个：栈为空，curr为空。这是因为中序遍历优中后右的特性，会有一个时刻栈为空但右结点并未被遍历，因此只有在curr也为空证明右结点不存在的情况下，才能结束遍历。

【代码】

```
/**  
 * Definition for a binary tree node.  
 * struct TreeNode {  
 *     int val;  
 *     TreeNode *left;  
 *     TreeNode *right;  
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}  
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}  
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),  
 *         left(left), right(right) {}
```

```
* };  
*/  
  
//递归法  
// class Solution {  
// public:  
//     std::vector<int> ret;  
//     vector<int> inorderTraversal(TreeNode* root) {  
//         postOrder(root);  
//         return ret;  
//     }  
//     void postOrder(TreeNode* root) {  
//         if (root == nullptr) return;  
//         inorderTraversal(root->left);  
//         ret.push_back(root->val);  
//         inorderTraversal(root->right);  
//     }  
// };
```

```
//迭代法  
class Solution {  
public:  
    vector<int> inorderTraversal(TreeNode* root) {  
        vector<int> ans;  
        stack<TreeNode*> stk;  
        TreeNode* curr = root;  
        while(!stk.empty() || curr != NULL) {  
            // 找到节点的最左侧节点，同时记录路径入栈  
            while(curr != NULL) {  
                stk.push(curr);  
                curr = curr->left;  
            }  
            // top 定义是此刻的弹栈元素  
            TreeNode* top = stk.top();  
            ans.push_back(top->val);  
            stk.pop();  
        }  
    }  
};
```

```
stk.pop();
// 处理过最左侧结点后，判断其是否存在右子树
if(top->right != NULL)
    curr = top->right;
}
return ans;
};
```

LeetCode 145. 二叉树的后序遍历

难度简单630

给定一个二叉树，返回它的 *后序遍历*。

示例：

输入： [1,null,2,3]

```
1
 \
 2
 /
3
```

输出： [3,2,1]

进阶：递归算法很简单，你可以通过迭代算法完成吗？

【思路】

- 1、前序遍历的过程是中左右。
- 2、将其转化成中右左。也就是压栈的过程中优先压入左子树，再压入右子树。
- 3、在弹栈的同时将此弹栈结点压入另一个栈，完成逆序。
- 4、对新栈中的元素直接顺序弹栈并输出。

【代码】

```
/**  
 * Definition for a binary tree node.  
 * struct TreeNode {  
 *     int val;  
 *     TreeNode *left;  
 *     TreeNode *right;  
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}  
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}  
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x),  
 * left(left), right(right) {}  
 * };  
 */  
/*  
//递归版  
class Solution {  
public:  
    vector<int> ans;  
    vector<int> preorderTraversal(TreeNode* root) {  
        // 为空则直接返回  
        if(root == NULL)  
            return ans;  
        preorderTraversal(root->left);  
        preorderTraversal(root->right);  
        ans.push_back(root->val);  
        return ans;  
    }  
};  
*/  
//非递归版  
class Solution {  
public:  
    std::vector<int> postorderTraversal(TreeNode* root) {  
        std::vector<int> res;  
        if (!root) return res;
```

```
    std::stack<TreeNode*> st1;
    std::stack<TreeNode*> st2;
    st1.push(root);
    // 栈一顺序存储
    while (!st1.empty()){
        TreeNode* node = st1.top();
        st1.pop();
        st2.push(node);
        if (node->left) st1.push(node->left);
        if (node->right) st1.push(node->right);
    }
    // 栈二直接输出
    while (!st2.empty()) {
        res.push_back(st2.top()->val);
        st2.pop();
    }
    return res;
}
};


```

具体图解可参考这位老哥的思路：<https://leetcode-cn.com/problems/binary-tree-preorder-traversal/solution/cer-cha-shu-san-chong-bian-li-qian-zhong-erk2/>

哈喽，我是小贺哥，就爱分享编程知识，如果觉得文章对你有帮助，别忘记关注我哦！



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复 「大礼包」
白嫖编程学习资料

关注公众号回复 「加群」 拉
你进百人技术交流群

九、计算机网络

关于 面试经常问的 TCP 和 UDP ， 我之前在知乎上写了一篇文章：参考：<https://zhuanlan.zhihu.com/p/108822858>

目录：

- 1、UDP 和 TCP 的特点与区别
- 2、UDP 、TCP 首部格式
- 3、TCP 的三次握手和四次挥手
- 4、TCP 的三次握手（为什么三次？）
- 5、TCP 的四次挥手（为什么四次？）
- 6、TCP 长连接和短连接的区别
- 7、TCP 粘包、拆包及解决办法
- 8、TCP 可靠传输
- 9、TCP 滑动窗口
- 10、TCP 流量控制
- 11、TCP 拥塞控制
- 12、提供网络利用率

打开知乎，搜索关键字「一文搞定 TCP UDP」排名第一的答案就是了。

The screenshot shows a Zhihu post with the title '一文搞定 TCP UDP' highlighted with a red arrow. Below the title, a red arrow points to the text '第一个就是我写得答案了'. The post content includes a thumbnail image of a globe with network connections, the author's note '请耐心阅读。目录：1、UDP 和 TCP 的特点与区别 2、UDP、TCP 首部格式 3、TCP 的三次握手和四次挥手 4、TCP 的三次握手（为什么三次？）', and interaction metrics: 1653 upvotes, 37 comments, and the date 04-07.

重点在 TCP/IP 协议和 HTTP 协议。

1、OSI 七层协议模型

OSI 模型（Open System Interconnection Model）是一个由 ISO 提出得到概念模型，试图提供一个使各种不同的的计算机和网络在世界范围内实现互联的标准框架。

虽然 OSI 参考模型在实际中的应用意义并不是很大，但是它对于理解网络协议内部的运作很有帮助，为我们学习网络协议提供了一个很好的参考。它将计算机网络体系结构划分为 7 层，每层都为上一层提供了良好的接口。以下将具体介绍各层结构及功能。

2、分层结构

OSI 参考模型采用分层结构，如图所示。附上一张经典图。

TCP/IP

第7层 应用层

各种应用程序协议，如 HTTP、FTP、SMTP、POP3。



7

第6层 表示层

信息的语法语义以及它们的关联，如加密解密、转换翻译、压缩解压缩。

6

第5层 会话层

不同机器上的用户之间建立及管理会话。

5

第4层 传输层

接受上一层的数据，在必要的时候把数据进行分割，并将这些数据交给网络层，且保证这些数据段有效到达对端。

4

TCP 传输控制协议
UDP 用户数据报协议



第3层 网络层

控制子网的运行，如逻辑编址、分组传输、路由选择。

3



第2层 数据链路层

物理寻址，同时将原始比特流转变为逻辑传输线路。

2



1 IEEE 802.2
Ethernet v.2
Internetwork

主要分为以下七层（从下至上）：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层。

3、各层功能

■ 物理层

简单的说，物理层（Physical Layer）确保原始的数据可在各种物理媒体上传输。在这一层上面规定了激活，维持，关闭通信端点之间的机械性，电气特性，功能特性，为上层协议提供了一个传输数据的物理媒体，这一层传输的是 bit 流。

■ 数据链路层

数据链路层（Data Link Layer）在不可靠的物理介质上提供可靠的传输。该层的作用包括：物理地址寻址、数据的成帧、流量控制、数据的检错、重发等。这一层中将 bit 流封装成 frame 帧。

■ 网络层

网络层（Network Layer）负责对子网间的数据包进行路由选择。此外，网络层还可以实现拥塞控制、网际互连等功能。在这一层，数据的单位称为数据包（packet）。

■ 传输层

传输层是第一个端到端，即主机到主机的层次。传输层负责将上层数据分段并提供端到端的、可靠的或不可靠的传输。此外，传输层还要处理端到端的差错控制和流量控制问题。在这一层，数据的单位称为数据段（segment）。

■ 会话层

这一层管理主机之间的会话进程，即负责建立、管理、终止进程之间的会话。会话层还利用在数据中插入校验点来实现数据的同步，访问验证和会话管理在内的建立和维护应用之间通信的机制。如服务器验证用户登录便是由会话层完成的。使通信会话在通信失效时从校验点继续恢复通信。**比如说建立会话，如 session 认证、断点续传。**

■ 表示层

这一层主要解决用户信息的语法表示问题。它将欲交换的数据从适合于某一用户的抽象语法，转换为适合于OSI系统内部使用的传送语法。即提供格式化的表示和转换数据服务。数据的压缩和解压缩，加密和解密等工作都由表示层负责。比如**图像、视频编码解，数据加密。**

■ 应用层

这一层为操作系统或网络应用程序提供访问网络服务的接口。

4、各层传输协议、传输单元、主要功能性设备比较

名称	传输协议	传输单元	主要功能设备/接口
物理层	IEEE 802.1A、IEEE 802.2	bit-flow 比特流	光纤, 双绞线, 中继器, 集线器, 网线接口
数据链路层	ARP、MAC、FDDI、Ethernet、Arpanet、PPP、PDN	frame 帧	网桥、二层交换机
网络层	IP、ICMP、ARP、RARP	数据包 (packet)	路由器、三层交换机
传输层	TCP、UDP	Segment/Datagram	四层交换机
会话层	SMTP、DNS	报文	QoS
表示层	Telnet、SNMP	报文	-
应用层	FTP、TFTP、Telnet、HTTP、DNS	报文	-

5、描述TCP头部？

- 序号 (32bit) : 传输方向上字节流的字节编号。初始时序号会被设置一个随机的初始值 (ISN) , 之后每次发送数据时, 序号值 = ISN + 数据在整个字节流中的偏移。假设A -> B且ISN = 1024, 第一段数据512字节已经到B, 则第二段数据发送时序号为1024 + 512。用于解决网络包乱序问题。
- 确认号 (32bit) : 接收方对发送方TCP报文段的响应, 其值是收到的序号值 + 1。
- 首部长 (4bit) : 标识首部有多少个4字节 * 首部长, 最大为15, 即60字节。
- 标志位 (6bit) :
 - URG: 标志紧急指针是否有效。
 - ACK: 标志确认号是否有效 (确认报文段) 。用于解决丢包问题。
 - PSH: 提示接收端立即从缓冲读走数据。

- RST：表示要求对方重新建立连接（复位报文段）。
- SYN：表示请求建立一个连接（连接报文段）。
- FIN：表示关闭连接（断开报文段）。
- 窗口（16bit）：接收窗口。用于告知对方（发送方）本方的缓冲还能接收多少字节数据。用于解决流控。
- 校验和（16bit）：接收端用CRC检验整个报文段有无损坏。

6、TCP三次握手和挥手

3. 三次握手过程？

- 第一次：客户端发含SYN位， $SEQ_NUM = S$ 的包到服务器。（客 -> SYN_SEND）
- 第二次：服务器发含ACK，SYN位且 $ACK_NUM = S + 1$ ， $SEQ_NUM = P$ 的包到客户机。（服 -> SYN_RECV）
- 第三次：客户机发送含ACK位， $ACK_NUM = P + 1$ 的包到服务器。（客 -> ESTABLISH，服 -> ESTABLISH）

4. 四次挥手过程？

- 第一次：客户机发含FIN位， $SEQ = Q$ 的包到服务器。（客 -> FIN_WAIT_1）
- 第二次：服务器发送含ACK且 $ACK_NUM = Q + 1$ 的包到服务器。（服 -> CLOSE_WAIT，客 -> FIN_WAIT_2）
 - 此处有等待
- 第三次：服务器发送含FIN且 $SEQ_NUM = R$ 的包到客户机。（服 -> LAST_ACK，客 -> TIME_WAIT）
 - 此处有等待
- 第四次：客户机发送最后一个含有ACK位且 $ACK_NUM = R + 1$ 的包到客户机。（服 -> CLOSED）

5. 为什么握手是三次，挥手是四次？

- 对于握手：握手只需要确认双方通信时的初始化序号，保证通信不会乱序。（第三次握手必要性：假设服务端的确认丢失，连接并未断开，客户机超时重发连接请求，这样服务器会对同一个客户机保持多个连接，造成资源浪费。）
- 对于挥手：TCP是双工的，所以发送方和接收方都需要FIN和ACK。只不过有一方是被动的，所以看上去就成了4次挥手。

6. TCP连接状态？

- CLOSED：初始状态。
- LISTEN：服务器处于监听状态。

- SYN_SEND：客户端socket执行CONNECT连接，发送SYN包，进入此状态。
- SYN_RECV：服务端收到SYN包并发送服务端SYN包，进入此状态。
- ESTABLISH：表示连接建立。客户端发送了最后一个ACK包后进入此状态，服务端接收到ACK包后进入此状态。
- FIN_WAIT_1：终止连接的一方（通常是客户机）发送了FIN报文后进入。等待对方FIN。
- CLOSE_WAIT：（假设服务器）接收到客户机FIN包之后等待关闭的阶段。在接收到对方的FIN包之后，自然是需要立即回复ACK包的，表示已经知道断开请求。但是本方是否立即断开连接（发送FIN包）取决于是否还有数据需要发送给客户端，若有，则在发送FIN包之前均为此状态。
- FIN_WAIT_2：此时是半连接状态，即有一方要求关闭连接，等待另一方关闭。客户端接收到服务器的ACK包，但并没有立即接收到服务端的FIN包，进入FIN_WAIT_2状态。
- LAST_ACK：服务端发动最后的FIN包，等待最后的客户端ACK响应，进入此状态。
- TIME_WAIT：客户端收到服务端的FIN包，并立即发出ACK包做最后的确认，在此之后的2MSL时间称为TIME_WAIT状态。

7. 解释FIN_WAIT_2, CLOSE_WAIT状态和TIME_WAIT状态？

- FIN_WAIT_2：
 - 半关闭状态。
 - 发送断开请求一方还有接收数据能力，但已经没有发送数据能力。
- CLOSE_WAIT状态：
 - 被动关闭连接一方接收到FIN包会立即回应ACK包表示已接收到断开请求。
 - 被动关闭连接一方如果还有剩余数据要发送就会进入CLOSED_WAIT状态。
- TIME_WAIT状态：
 - 又叫2MSL等待状态。
 - 如果客户端直接进入CLOSED状态，如果服务端没有接收到最后一次ACK包会在超时之后重新再发FIN包，此时因为客户端已经CLOSED，所以服务端就不会收到ACK而是收到RST。所以TIME_WAIT状态目的是防止最后一次握手数据没有到达对方而触发重传FIN准备的。
 - 在2MSL时间内，同一个socket不能再被使用，否则有可能会和旧连接数据混淆（如果新连接和旧连接的socket相同的话）。

8. 解释RTO, RTT和超时重传？

- 超时重传：发送端发送报文后若长时间未收到确认的报文则需要重发该报文。可能有以下几种情况：

- 发送的数据没能到达接收端，所以对方没有响应。
- 接收端接收到数据，但是ACK报文在返回过程中丢失。
- 接收端拒绝或丢弃数据。
- RTO：从上一次发送数据，因为长期没有收到ACK响应，到下一次重发之间的时间。就是重传间隔。
 - 通常每次重传RTO是前一次重传间隔的两倍，计量单位通常是RTT。例：1RTT, 2RTT, 4RTT, 8RTT.....
 - 重传次数到达上限之后停止重传。
- RTT：数据从发送到接收到对方响应之间的时间间隔，即数据报在网络中一个往返用时。大小不稳定。
- 目的是接收方通过TCP头窗口字段告知发送方本方可接收的最大数据量，用以解决发送速率过快导致接收方不能接收的问题。所以流量控制是点对点控制。
- TCP是双工协议，双方可以同时通信，所以发送方接收方各自维护一个发送窗和接收窗。
 - 发送窗：用来限制发送方可以发送的数据大小，其中发送窗口的大小由接收端返回的TCP报文段中窗口字段来控制，接收方通过此字段告知发送方自己的缓冲（受系统、硬件等限制）大小。
 - 接收窗：用来标记可以接收的数据大小。
- TCP是流数据，发送出去的数据流可以被分为以下四部分：已发送且被确认部分 | 已发送未被确认部分 | 未发送但可发送部分 | 不可发送部分，其中发送窗 = 已发送未确认部分 + 未发但可发送部分。接收到的数据流可分为：已接收 | 未接收但准备接收 | 未接收不准备接收。接收窗 = 未接收但准备接收部分。
- 发送窗内数据只有当接收到接收端某段发送数据的ACK响应时才移动发送窗，左边缘紧贴刚被确认的数据。接收窗也只有接收到数据且最左侧连续时才移动接收窗口。

拥塞控制原理？

- 拥塞控制目的是防止数据被过多注网络中导致网络资源（路由器、交换机等）过载。因为拥塞控制涉及网络链路全局，所以属于全局控制。控制拥塞使用拥塞窗口。
- TCP拥塞控制算法：
 - 慢开始 & 拥塞避免：先试探网络拥塞程度再逐渐增大拥塞窗口。每次收到确认后拥塞窗口翻倍，直到达到阀值ssthresh，这部分是慢开始过程。达到阀值后每次以一个MSS为单位增长拥塞窗口大小，当发生拥塞（超时未收到确认），将阀值减为原先一半，继续执行线性增加，这个过程为拥塞避免。
 - 快速重传 & 快速恢复：略。

- 最终拥塞窗口会收敛于稳定值。

7、如何区分流量控制和拥塞控制?

- 流量控制属于通信双方协商；拥塞控制涉及通信链路全局。
- 流量控制需要通信双方各维护一个发送窗、一个接收窗，对任意一方，接收窗大小由自身决定，发送窗大小由接收方响应的TCP报文段中窗口值确定；拥塞控制的拥塞窗口大小变化由试探性发送一定数据量数据探查网络状况后而自适应调整。
- 实际最终发送窗口 = min{流控发送窗口, 拥塞窗口}。

8、TCP如何提供可靠数据传输的?

- 建立连接（标志位）：通信前确认通信实体存在。
- 序号机制（序号、确认号）：确保了数据是按序、完整到达。
- 数据校验（校验和）：CRC校验全部数据。
- 超时重传（定时器）：保证因链路故障未能到达数据能够被多次重发。
- 窗口机制（窗口）：提供流量控制，避免过量发送。
- 拥塞控制：同上。

9、TCP socket交互流程?

- 服务器：
 - 创建socket -> int socket(int domain, int type, int protocol);
 - domain: 协议域，决定了socket的地址类型，IPv4为AF_INET。
 - type: 指定socket类型，SOCK_STREAM为TCP连接。
 - protocol: 指定协议。IPPROTO_TCP表示TCP协议，为0时自动选择type默认协议。
 - 绑定socket和端口号 -> int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
 - sockfd: socket返回的套接字描述符，类似于文件描述符fd。
 - addr: 有个sockaddr类型数据的指针，指向的是被绑定结构变量。

```

// IPv4的sockaddr地址结构
struct sockaddr_in {
    sa_family_t sin_family;      // 协议类型, AF_INET
    in_port_t sin_port;         // 端口号
    struct in_addr sin_addr;    // IP地址
};

struct in_addr {
    uint32_t s_addr;
}

```

- `addrlen`: 地址长度。
- 监听端口号 -> `int listen(int sockfd, int backlog);`
 - `sockfd`: 要监听的socket描述字。
 - `backlog`: socket可以排队的最大连接数。
- 接收用户请求 -> `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
 - `sockfd`: 服务器socket描述字。
 - `addr`: 指向地址结构指针。
 - `addrlen`: 协议地址长度。
 - 注: 一旦accept某个客户机请求成功将返回一个全新的描述符用于标识具体客户的TCP连接。
- 从socket中读取字符 -> `ssize_t read(int fd, void *buf, size_t count);`
 - `fd`: 连接描述字。
 - `buf`: 缓冲区buf。
 - `count`: 缓冲区长度。
 - 注: 大于0表示读取的字节数, 返回0表示文件读取结束, 小于0表示发生错误。
- 关闭socket -> `int close(int fd);`
 - `fd`: accept返回的连接描述字, 每个连接有一个, 生命周期为连接周期。
 - 注: `sockfd`是监听描述字, 一个服务器只有一个, 用于监听是否有连接; `fd`是连接描述字, 用于每个连接的操作。
- 客户机:
 - 创建socket -> `int socket(int domain, int type, int protocol);`
 - 连接指定计算机 -> `int connect(int sockfd, struct sockaddr* addr, socklen_t addrlen);`

- sockfd客户端的sock描述字。
- addr: 服务器的地址。
- addrlen: socket地址长度。
- 向socket写入信息 -> ssize_t write(int fd, const void *buf, size_t count);
 - fd、buf、count: 同read中意义。
 - 大于0表示写了部分或全部数据，小于0表示出错。
- 关闭oscket -> int close(int fd);
 - fd: 同服务器端fd。

哈喽，我是小贺哥，就爱分享编程知识，如果觉得文章对你有帮助，别忘记关注我哦！



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复「大礼包」
白嫖编程学习资料

关注公众号回复「加群」拉
你进百人技术交流群

十、操作系统

1、操作系统特点

并发性、共享性、虚拟性、不确定性。

2、什么是进程

- 1) 进程是指在系统中正在运行的一个应用程序，程序一旦运行就是进程；
- 2) 进程可以认为是程序执行的一个实例，进程是系统进行资源分配的最小单位，且每个进程拥有独立的地址空间；
- 3) 一个进程无法直接访问另一个进程的变量和数据结构，如果希望一个进程去访问另一个进程的资源，需要使用进程间的通信，比如：管道、消息队列等
- 4) 线程是进程的一个实体，是进程的一条执行路径；比进程更小的独立运行的基本单位，线程也被称为轻量级进程，一个程序至少有一个进程，一个进程至少有一个线程；

3、进程

进程是程序的一次执行，该程序可以与其他程序并发执行；

进程有运行、阻塞、就绪三个基本状态；

进程调度算法：先来先服务调度算法、短作业优先调度算法、非抢占式优先级调度算法、抢占式优先级调度算法、高响应比优先调度算法、时间片轮转法调度算法；

4、进程与线程的区别

- 1) 同一进程的线程共享本进程的地址空间，而进程之间则是独立的地址空间；
- 2) 同一进程内的线程共享本进程的资源，但是进程之间的资源是独立的；
- 3) 一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程崩溃，所以多进程比多线程健壮；

- 4) 进程切换，消耗的资源大。所以涉及到频繁的切换，使用线程要好于进程；
- 5) 两者均可并发执行；
- 6) 每个独立的进程有一个程序的入口、程序出口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

5、进程状态转换图

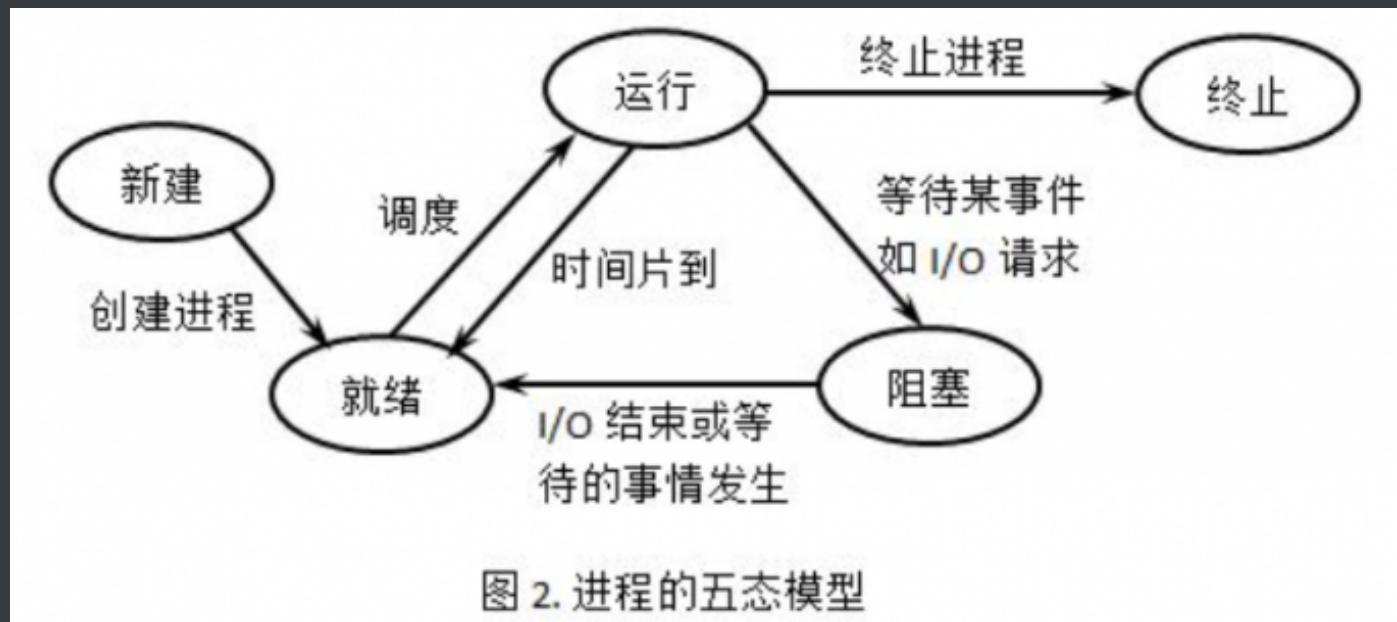


图 2. 进程的五态模型

- 1) 新状态：进程已经创建
- 2) 就绪态：进程做好了准备，准备执行，等待分配处理机
- 3) 执行态：该进程正在执行；
- 4) 阻塞态：等待某事件发生才能执行，如等待I/O完成；
- 5) 终止状态

6、进程的创建过程？需要哪些函数？需要哪些数据结构？

- 1) `fork` 函数创造的子进程是父进程的完整副本，复制了父亲进程的资源，包括内存的内容 `task_struct` 内容；

- 2) vfork创建的子进程与父进程共享数据段，而且由vfork创建的子进程将先于父进程运行；
- 3) linux上创建线程一般使用的是pthread库，实际上linux也给我们提供了创建线程的系统调用，就是clone；

7、进程创建子进程,fork详解

- 1) 函数原型

```
pid_t fork(void); //void代表没有任何形式参数
```

- 2) 除了0号进程（系统创建的）之外，linux系统中都是由其他进程创建的。创建新进程的进程，即调用fork函数的进程为父进程，新建的进程为子进程。

- 3) fork函数不需要任何参数，对于返回值有三种情况：

- ① 对于父进程，fork函数返回新建子进程的pid；

- ② 对于子进程，fork函数返回 0；

- ③ 如果出错，fork 函数返回 -1。

```
int pid=fork();
if(pid < 0){
    //失败，一般是该用户的进程数达到限制或者内存被用光了
    .....
}
else if(pid == 0){
    //子进程执行的代码
    .....
}
else{
    //父进程执行的代码
    .....
}
```

8、子进程和父进程怎么通信？

- 1) 在 Linux 系统中实现父子进程的通信可以采用 pipe() 和 fork() 函数进行实现；
- 2) 对于父子进程，在程序运行时首先进入的是父进程，其次是子进程，在此我个人认为，在创建父子进程的时候程序是先运行创建的程序，其次在复制父进程创建子进程。fork() 函数主要是以父进程为蓝本复制一个进程，其 ID 号和父进程的 ID 号不同。对于结果 fork 出来的子进程的父进程 ID 号是执行 fork() 函数的进程的 ID 号。
- 3) 管道：是指用于连接一个读进程和一个写进程，以实现它们之间通信的共享文件，又称 pipe 文件。
- 4) 写进程在管道的尾端写入数据，读进程在管道的首端读出数据。

9、进程和作业的区别？

- 1) 进程是程序的一次动态执行，属于动态概念；
- 2) 一个进程可以执行一个或几个程序，同一个程序可由几个进程执行；
- 3) 程序可以作为一种软件资源长期保留，而进程是程序的一次执行；
- 4) 进程具有并发性，能与其他进程并发执行；
- 5) 进程是一个独立的运行单位；

10、死锁是什么？必要条件？如何解决？

所谓死锁，是指多个进程循环等待它方占有的资源而无限期地僵持下去的局面。很显然，如果没有外力的作用，那么死锁涉及到的各个进程都将永远处于封锁状态。当两个或两个以上的进程同时对多个互斥资源提出使用要求时，有可能导致死锁。

（1）互斥条件。即某个资源在一段时间内只能由一个进程占有，不能同时被两个或两个以上的进程占有。这种独占资源如 CD-ROM 驱动器，打印机等等，必须在占有该资源的进程主动释放它之后，其它进程才能占有该资源。这是由资源本身的属性所决定的。如独木桥就是一种独占资源，两方的人不能同时过桥。

〈2〉 不可抢占条件。进程所获得的资源在未使用完毕之前，资源申请者不能强行地从资源占有者手中夺取资源，而只能由该资源的占有者进程自行释放。如过独木桥的人不能强迫对方后退，也不能非法地将对方推下桥，必须是桥上的人自己过桥后空出桥面（即主动释放占有资源），对方的人才能过桥。

〈3〉 占有且申请条件。进程至少已经占有一个资源，但又申请新的资源；由于该资源已被另外进程占有，此时该进程阻塞；但是，它在等待新资源之时，仍继续占用已占有的资源。还以过独木桥为例，甲乙两人在桥上相遇。甲走过一段桥面（即占有了一些资源），还需要走其余的桥面（申请新的资源），但那部分桥面被乙占有（乙走过一段桥面）。甲过不去，前进不能，又不后退；乙也处于同样的状况。

〈4〉 循环等待条件。存在一个进程等待序列{P₁, P₂, ..., P_n}，其中P₁等待P₂所占有的某一资源，P₂等待P₃所占有的某一源，……，而P_n等待P₁所占有的的某一资源，形成一个进程循环等待环。就像前面的过独木桥问题，甲等待乙占有的桥面，而乙又等待甲占有的桥面，从而彼此循环等待。

死锁的预防是保证系统不进入死锁状态的一种策略。它的基本思想是要求进程申请资源时遵循某种协议，从而打破产生死锁的四个必要条件中的一个或几个，保证系统不会进入死锁状态。

<1>打破互斥条件。即允许进程同时访问某些资源。但是，有的资源是不允许被同时访问的，像打印机等等，这是由资源本身的属性所决定的。所以，这种办法并无实用价值。

<2>打破不可抢占条件。即允许进程强行从占有者那里夺取某些资源。就是说，当一个进程已占有了某些资源，它又申请新的资源，但不能立即被满足时，它必须释放所占有的全部资源，以后再重新申请。它所释放的资源可以分配给其它进程。这就相当于该进程占有的资源被隐蔽地强占了。这种预防死锁的方法实现起来困难，会降低系统性能。

<3>打破占有且申请条件。可以实行资源预先分配策略。即进程在运行前一次性地向系统申请它所需要的全部资源。如果某个进程所需的全部资源得不到满足，则不分配任何资源，此进程暂不运行。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程。由于运行的进程已占有了它所需的全部资源，所以不会发生占有资源又申请资源的现象，因此不会发生死锁。

<4>打破循环等待条件，实行资源有序分配策略。采用这种策略，即把资源事先分类编号，按号分配，使进程在申请，占用资源时不会形成环路。所有进程对资源的请求必须严格按资源序号递增的顺序提出。进程占用了小号资源，才能申请大号资源，就不会产生环路，从而预防了死锁。

11、鸵鸟策略

假设的前提是，这样的问题出现的概率很低。比如，在操作系统中，为应对死锁问题，可以采用这样的一种办法。当系统发生[死锁]时不会对用户造成多大影响，或系统很少发生[死锁]的场合采用允许死锁发生的鸵鸟算法，这样一来可能开销比不允许发生死锁及检测和解除死锁的小。如果[死锁]很长时间才发生一次，而系统每周都会因硬件故障、[编译器]错误或操作系统错误而崩溃一次，那么大多数工程师不会以性能损失或者易用性损失的代价来设计较为复杂的死锁解决策略，来消除死锁。鸵鸟策略的实质：出现死锁的概率很小，并且出现之后处理死锁会花费很大的代价，还不如不做处理，OS中这种置之不理的策略称之为鸵鸟策略（也叫鸵鸟算法）。

12、银行家算法

在避免[死锁]的方法中，所施加的限制条件较弱，有可能获得令人满意的系统性能。在该方法中把系统的状态分为安全状态和不安全状态，只要能使系统始终都处于安全状态，便可以避免发生[死锁]。

银行家算法的基本思想是分配资源之前，判断系统是否是安全的；若是，才分配。它是最具有代表性的避免[死锁]的算法。

设进程 $cusneed$ 提出请求 $REQUEST[i]$ ，则银行家算法按如下规则进行判断。

(1)如果 $REQUEST[cusneed][i] \leq NEED[cusneed][i]$ ，则转 (2)；否则，出错。

(2)如果 $REQUEST[cusneed][i] \leq AVAILABLE[i]$ ，则转 (3)；否则，等待。

(3)系统试探分配资源，修改相关数据：

$AVAILABLE[i] -= REQUEST[cusneed][i];$

$ALLOCATION[cusneed][i] += REQUEST[cusneed][i];$

$NEED[cusneed][i] -= REQUEST[cusneed][i];$

(4) 系统执行安全性检查，如安全，则分配成立；否则试探性分配作废，系统恢复原状，进程等待。

13、进程间通信方式有几种，他们之间的区别是什么？

1) 管道

管道，通常指无名管道。

① 半双工的，具有固定的读端和写端；

② 只能用于具有亲属关系的进程之间的通信；

③ 可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write函数。但是它不是普通的文件，并不属于其他任何文件系统，只能用于内存中。

④ Int pipe(int fd[2]); 当一个管道建立时，会创建两个文件文件描述符，要关闭管道只需将这两个文件描述符关闭即可。

2) FiFO（有名管道）

① FIFO可以在无关的进程之间交换数据，与无名管道不同；

② FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中；

③ Int mkfifo(const char* pathname, mode_t mode);

3) 消息队列

① 消息队列，是消息的连接表，存放在内核中。一个消息队列由一个标识符来标识；

② 消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级；

③ 消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除；

④ 消息队列可以实现消息的随机查询

4) 信号量

- ① 信号量是一个计数器，信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据；
- ② 信号量用于进程间同步，若要在进程间传递数据需要结合共享内存；
- ③ 信号量基于操作系统的PV操作，程序对信号量的操作都是原子操作；

5) 共享内存

- ① 共享内存，指两个或多个进程共享一个给定的存储区；
- ② 共享内存是最快的一种进程通信方式，因为进程是直接对内存进行存取；
- ③ 因为多个进程可以同时操作，所以需要进行同步；
- ④ 信号量+共享内存通常结合在一起使用。

14、线程同步的方式？怎么用？

- 1) 线程同步是指多线程通过特定的设置来控制线程之间的执行顺序，也可以说在线程之间通过同步建立起执行顺序的关系；
- 2) 主要四种方式，临界区、互斥对象、信号量、事件对象；其中临界区和互斥对象主要用于互斥控制，信号量和事件对象主要用于同步控制；
- 3) 临界区：通过对多线程的串行化来访问公共资源或一段代码，速度快、适合控制数据访问。在任意一个时刻只允许一个线程对共享资源进行访问，如果有多个线程试图访问公共资源，那么在有一个线程进入后，其他试图访问公共资源的线程将被挂起，并一直等到进入临界区的线程离开，临界区在被释放后，其他线程才可以抢占。
- 4) 互斥对象：互斥对象和临界区很像，采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程同时访问。当前拥有互斥对象的线程处理完任务后必须将线程交出，以便其他线程访问该资源。

5) 信号量：它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。在用CreateSemaphore()创建信号量时即要同时指出允许的最大资源计数和当前可用资源计数。一般是将当前可用资源计数设置为最大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就会减1，只要当前可用资源计数是大于0的，就可以发出信号量信号。但是当当前可用计数减小到0时则说明当前占用资源的线程数已经达到了所允许的最大数目，不能在允许其他线程的进入，此时的信号量信号将无法发出。线程在处理完共享资源后，应在离开的同时通过ReleaseSemaphore()函数将当前可用资源计数加1。在任何时候当前可用资源计数决不可能大于最大资源计数。

6) 事件对象：通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程的优先级比较的操作。

15、页和段的区别？

1) 页是信息的物理单位，分页是由于系统管理的需要。段是信息的逻辑单位，分段是为了满足用户的要求。

2) 页的大小固定且由系统决定，段的长度不固定，决定于用户所编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。

3) 分页的作业的地址空间是一维的，程序员只需要利用一个记忆符，即可表示一个地址。分段的作业地址空间则是二维的，程序员在标识一个地址时，既需要给出段名，又需要给出段的地址值。

16、孤儿进程和僵尸进程的区别？怎么避免这两类进程？守护进程？

1、一般情况下，子进程是由父进程创建，而子进程和父进程的退出是无顺序的，两者之间都不知道谁先退出。正常情况下父进程先结束会调用 wait 或者 waitpid 函数等待子进程完成再退出，而一旦父进程不等待直接退出，则剩下的子进程会被init(pid=1)进程接收，成为孤儿进程。（进程树中除了init都会有父进程）。

2、如果子进程先退出了，父进程还未结束并且没有调用 wait 或者 waitpid 函数获取子进程的状态信息，则子进程残留的状态信息（task_struct 结构和少量资源信息）会变成僵尸进程。

子进程退出时向父进程发送SIGCHLD信号，父进程处理SIGCHLD信号。在信号处理函数中调用wait进行处理僵尸进程。

原理是将子进程成为孤儿进程，从而其的父进程变为init进程，通过init进程可以处理僵尸进程。

3、守护进程（daemon）是指在后台运行，没有控制终端与之相连的进程。它独立于控制终端，通常周期性地执行某种任务。守护进程脱离于终端是为了避免进程在执行过程中的信息在任何终端上显示并且进程也不会被任何终端所产生的终端信息所打断。

17、守护进程是什么？怎么实现？

1. 守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。守护进程是一种很有用的进程。
2. 守护进程特点

1) 守护进程最重要的特性是后台运行。

2) 守护进程必须与其运行前的环境隔离开来。这些环境包括未关闭的文件描述符，控制终端，会话和进程组，工作目录以及文件创建掩模等。这些环境通常是守护进程从执行它的父进程（特别是shell）中继承下来的。

3) 守护进程的启动方式有其特殊之处。它可以在Linux系统启动时从启动脚本/etc/rc.d中启动，可以由作业规划进程crond启动，还可以由用户终端（shell）执行。

3. 实现

1) 在父进程中执行fork并exit推出；

2) 在子进程中调用setsid函数创建新的会话；

3) 在子进程中调用chdir函数，让根目录 "/" 成为子进程的工作目录；

4) 在子进程中调用umask函数，设置进程的umask为0；

5) 在子进程中关闭任何不需要的文件描述符

18、线程和进程的区别？线程共享的资源是什么？

- 1) 一个程序至少有一个进程，一个进程至少有一个线程
- 2) 线程的划分尺度小于进程，使得多线程程序的并发性高
- 3) 进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率
- 4) 每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制
- 5) 多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配
- 6) 一个进程中的所有线程共享该进程的地址空间，但它们有各自独立的（私有的）栈（stack），Windows 线程的缺省堆栈大小为1M。堆(heap)的分配与栈有所不同，一般是一个进程有一个C运行时堆，这个堆为本进程中所有线程共享，windows 进程还有所谓进程默认堆，用户也可以创建自己的堆。

线程共享资源	线程独享资源
地址空间	程序计数器
全局变量	寄存器
打开的文件	栈
子进程	状态字
闹铃	
信号及信号服务程序	
记账信息	

线程私有：线程栈，寄存器，程序寄存器

共享：堆，地址空间，全局变量，静态变量

进程私有：地址空间，堆，全局变量，栈，寄存器

共享：代码段，公共数据，进程目录，进程ID

19、线程比进程具有哪些优势？

- 1) 线程在程序中是独立的，并发的执行流，但是，进程中的线程之间的隔离程度要小；
- 2) 线程比进程更具有更高的性能，这是由于同一个进程中的线程都有共性：多个线程将共享同一个进程虚拟空间；
- 3) 当操作系统创建一个进程时，必须为进程分配独立的内存空间，并分配大量相关资源；

20、什么时候用多进程？什么时候用多线程？

- 1) 需要频繁创建销毁的优先用线程；
- 2) 需要进行大量计算的优先使用线程；
- 3) 强相关的处理用线程，弱相关的处理用进程；
- 4) 可能要扩展到多机分布的用进程，多核分布的用线程；

21、协程是什么？

- 1) 是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程可以拥有多个协程；协程不是被操作系统内核管理，而完全是由程序所控制。
- 2) 协程的开销远远小于线程；
- 3) 协程拥有自己寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切换回来的时候，恢复先前保存的寄存器上下文和栈。
- 4) 每个协程表示一个执行单元，有自己的本地数据，与其他协程共享全局数据和其他资源。
- 5) 跨平台、跨体系架构、无需线程上下文切换的开销、方便切换控制流，简化编程模型；
- 6) 协程又称为微线程，协程的完成主要靠yield关键字，协程执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行；
- 7) 协程极高的执行效率，和多线程相比，线程数量越多，协程的性能优势就越明显；

8) 不需要多线程的锁机制；

22、递归锁？

1) 线程同步能够保证多个线程安全访问竞争资源，最简单的同步机制是引入互斥锁。互斥锁为资源引入一个状态：锁定/非锁定。某个线程要更改共享数据时，先将其锁定，此时资源的状态为“锁定”，其他线程不能更改；直到该线程释放资源，将资源的状态变成“非锁定”，其他的线程才能再次锁定该资源。互斥锁保证了每次只有一个线程进行写入操作，从而保证了多线程情况下数据的正确性。

2) 读写锁从广义的逻辑上讲，也可以认为是一种共享版的互斥锁。如果对一个临界区大部分是读操作而只有少量的写操作，读写锁在一定程度上能够降低线程互斥产生的代价。

3) Mutex可以分为递归锁(recursive mutex)和非递归锁(non-recursive mutex)。可递归锁也可称为可重入锁(reentrant mutex)，非递归锁又叫不可重入锁(non-reentrant mutex)。二者唯一的区别是，同一个线程可以多次获取同一个递归锁，不会产生死锁。而如果一个线程多次获取同一个非递归锁，则会产生死锁。

23、用户态到内核态的转化原理？

1) 系统调用

这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作，比如前例中fork()实际上就是执行了一个创建新进程的系统调用。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如Linux的int 80h中断。

2) 异常

当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。

3) 外围设备的中断

当外围设备完成用户请求的操作后，会向 CPU 发出相应的中断信号，这时 CPU 会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。

17. 中断的实现与作用，中断的实现过程？

- ① 关中断，进入不可再次响应中断的状态，由硬件实现。
- ② 保存断点，为了在[中断处理]结束后能正确返回到中断点。由硬件实现。
- ③ 将[中断服务程序]入口地址送PC，转向[中断服务程序]。可由硬件实现，也可由软件实现。
- ④ 保护现场、置屏蔽字、开中断，即保护CPU中某些寄存器的内容、设置[中断处理]次序、允许更高级的中断请求得到响应，实现中断嵌套由软件实现。
- ⑤ 设备服务，实际上有效的中断处理工作是在此程序段中实现的。由软件程序实现
- ⑥ 退出中断。在退出时，又应进入不可中断状态，即关中断、恢复屏蔽字、恢复现场、开中断、中断返回。由软件实现。

25、系统中断是什么，用户态和内核态的区别

1) 内核态与用户态是操作系统的两种运行级别,当程序运行在3级特权级上时，就可以称之为运行在用户态，因为这是最低特权级，是普通的用户进程运行的特权级，大部分用户直接面对的程序都是运行在用户态；反之，当程序运行在0级特权级上时，就可以称之为运行在内核态。运行在用户态下的程序不能直接访问操作系统内核数据结构和程序。当我们在系统中执行一个程序时，大部分时间是运行在用户态下的，在其需要操作系统帮助完成某些它没有权力和能力完成的工作时就会切换到内核态。

2) 这两种状态的主要差别是：处于用户态执行时，进程所能访问的内存空间和对象受到限制，其所处占有的处理机是可被抢占的；而处于核心态执行中的进程，则能访问所有的内存空间和对象，且所占有的处理机是不允许被抢占的。

26、CPU中断

1) CPU中断是什么

- ① 计算机处于执行期间；
- ② 系统内发生了非寻常或非预期的急需处理事件；
- ③ CPU暂时中断当前正在执行的程序而转去执行相应的事件处理程序；
- ④ 处理完毕后返回原来被中断处继续执行；

2) CPU中断的作用

- ① 可以使CPU和外设同时工作，使系统可以及时地响应外部事件；
- ② 可以允许多个外设同时工作，大大提高了CPU的利用率；
- ③ 可以使CPU及时处理各种软硬件故障。

27、执行一个系统调用时，OS发生的过程，越详细越好

1. 执行用户程序(如:fork)
2. 根据glibc中的函数实现，取得系统调用号并执行int \$0x80产生中断。
3. 进行地址空间的转换和堆栈的切换，执行SAVE_ALL。 (进行内核模式)
4. 进行中断处理，根据系统调用表调用内核函数。
5. 执行内核函数。
6. 执行 RESTORE_ALL 并返回用户模式

28、函数调用和系统调用的区别？

1) 系统调用

- ① 操作系统提供给用户程序调用的一组特殊的接口。用户程序可以通过这组特殊接口来获得操作系统内核提供的服务；
- ② 系统调用可以用来控制硬件；设置系统状态或读取内核数据；进程管理，系统调用接口用

来保证系统中进程能以多任务在虚拟环境下运行；

③ Linux中实现系统调用利用了0x86体系结构中的软件中断；

2) 函数调用

① 函数调用运行在用户空间；

② 它主要是通过压栈操作来进行函数调用；

3) 区别

函数库调用	系统调用
在所有的ANSI C编译器版本中，C库函数是相同的	各个操作系统的系统调用是不同的
它调用 函数库 中的一段程序（或函数）	它调用 系统内核 的服务
与 用户程序 相联系	是 操作系统 的一个入口点
在 用户地址空间 执行	在 内核地址空间 执行
它的运行时间属于“ 用户时间 ”	它的运行时间属于“ 系统时间 ”
属于 过程调用 ，调用开销较小	需要在 用户空间和内核上下文环境间切换 ，开销较大
在C函数库libc中有大约300个函数	在UNIX中大约有90个系统调用
典型的C函数库调用：system fprintf malloc	典型的系统调用：chdir fork write brk；

29、虚拟内存？使用虚拟内存的优点？什么是虚拟地址空间？

1) 虚拟内存，虚拟内存是一种内存管理技术，它会使程序自己认为自己拥有一块很大且连续的内存，然而，这个程序在内存中不是连续的，并且有些还会在磁盘上，在需要时进行数据交换；

2) 优点：可以弥补物理内存大小的不足；一定程度的提高反应速度；减少对物理内存的读取从而保护内存延长内存使用寿命；

3) 缺点：占用一定的物理硬盘空间；加大了对硬盘的读写；设置不得当会影响整机稳定性与速度。

4) 虚拟地址空间是对于一个单一进程的概念，这个进程看到的将是地址从0000开始的整个内存空间。虚拟存储器是一个抽象概念，它为每一个进程提供了一个假象，好像每一个进程都在独占的使用主存。每个进程看到的存储器都是一致的，称为虚拟地址空间。从最低的地址看起：程序代码和数据，堆，共享库，栈，内核虚拟存储器。大多数计算机的字长都是32位，这就限制了虚拟地址空间为4GB。

24. 线程安全？如何实现？

1) 如果你的代码所在的进程中多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和[单线程]运行的结果是一样的，而且其他的变量的值也和预期的一样的，就是线程安全的。

2) 线程安全问题都是由[全局变量及[静态变量]引起的。

3) 若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若多个线程同时执行写操作，一般都需要考虑[线程同步]，否则的话就可能影响线程安全。

4) 对于线程不安全的对象我们可以通过如下方法来实现线程安全：

① 加锁 利用Synchronized或者ReentrantLock来对不安全对象进行加锁，来实现线程执行的串行化，从而保证多线程同时操作对象的安全性，一个是语法层面的互斥锁，一个是API层面的互斥锁。

② 非阻塞同步来实现线程安全。原理就是：通俗点讲，就是先进性操作，如果没有其他线程争用共享数据，那操作就成功了；如果共享数据有争用，产生冲突，那就再采取其他措施(最常见的措施就是不断地重试，知道成功为止)。这种方法需要硬件的支持，因为我们需要操作和冲突检测这两个步骤具备原子性。通常这种指令包括CAS SC,FAI TAS等。

③ 线程本地化，一种无同步的方案，就是利用Threadlocal来为每一个线程创造一个共享变量的副本（副本之间是无关的）避免几个线程同时操作一个对象时发生线程安全问题。

31、常见的IO模型，五种？异步IO应用场景？有什么缺点？

1) 同步

就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。***也就是必须一件一件事做***，等前一件做完了才能做下一件事。就是我调用一个功能，该功能没有结束前，我死等结果。

2) 异步

当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。就是我调用一个功能，不需要知道该功能结果，该功能有结果后通知我（回调通知）

3) 阻塞

阻塞调用是指调用结果返回之前，当前线程会被挂起（线程进入非可执行状态，在这个状态下，cpu不会给线程分配时间片，即线程暂停运行）。函数只有在得到结果之后才会返回。对于同步调用来说，很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。就是调用我（函数），我（函数）没有接收完数据或者没有得到结果之前，我不会返回。

4) 非阻塞

指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。就是调用我（函数），我（函数）立即返回，通过select通知调用者。

1) 阻塞I/O

应用程序调用一个IO函数，导致应用程序阻塞，等待数据准备好。如果数据没有准备好，一直等待....数据准备好了，从内核拷贝到用户空间,IO函数返回成功指示。

2) 非阻塞I/O

我们把一个SOCKET接口设置为非阻塞就是告诉内核，当所请求的I/O操作无法完成时，不要将进程睡眠，而是返回一个错误。这样我们的I/O操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，会大量的占用CPU的时间。

3) I/O复用

I/O复用模型会用到select、poll、epoll函数，这几个函数也会使进程阻塞，但是和阻塞I/O所不同的，这三个函数可以同时阻塞多个I/O操作。而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时，才真正调用I/O操作函数。

4) 信号驱动I/O

首先我们允许套接口进行信号驱动I/O，并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个SIGIO信号，可以在信号处理函数中调用I/O操作函数处理数据。

5) 异步I/O

当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者的输入输出操作。

32、IO复用的原理？零拷贝？三个函数？ epoll 的 LT 和 ET 模式的理解。

1) IO复用是Linux中的IO模型之一，IO复用就是进程预先告诉内核需要监视的IO条件，使得内核一旦发现进程指定的一个或多个IO条件就绪，就通过进程进程处理，从而不会在单个IO上阻塞了。Linux中，提供了select、poll、epoll三种接口函数来实现IO复用。

2) Select

select的缺点：

- ① 单个进程能够监视的文件描述符的数量存在最大限制，通常是1024。由于select采用轮询的方式扫描文件描述符，文件描述符数量越多，性能越差；
- ② 内核/用户空间内存拷贝问题，select需要大量句柄数据结构，产生巨大开销；
- ③ Select返回的是含有整个句柄的数组，应用程序需要遍历整个数组才能发现哪些句柄发生事件；
- ④ Select的触发方式是水平触发，应用程序如果没有完成对一个已经就绪的文件描述符进行IO操作，那么每次select调用还会将这些文件描述符通知进程。

3) Poll

与select相比，poll使用链表保存文件描述符，一你才没有了监视文件数量的限制，但其他三个缺点依然存在

4) Epoll

上面所说的select缺点在epoll上不复存在，epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。Epoll是事件触发的，不是轮询查询的。没有最大的并发连接限制，内存拷贝，利用mmap（）文件映射内存加速与内核空间的消息传递。

区别总结：

1) 支持一个进程所能打开的最大连接数

① Select最大1024个连接，最大连接数有FD_SETSIZE宏定义，其大小是32位整数表示，可以改变宏定义进行修改，可以重新编译内核，性能可能会影响；

② Poll没有最大连接限制，原因是它是基于链表来存储的；

③ 连接数限数有上限，但是很大；

2) FD剧增后带来的IO效率问题

① 因为每次进行线性遍历，所以随着FD的增加会造成遍历速度下降，效率降低；

② Poll同上；

③ 因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的现象下降的性能问题。

3) 消息传递方式

① Select内核需要将消息传递到用户空间，都需要内核拷贝；

② Poll同上；

③ Epoll通过内核和用户空间共享来实现的。

epoll 的 LT 和 ET 模式的理解：

epoll对文件描述符的操作有两种模式：LT(level trigger)和ET(edge trigger)， LT是默认模式。

区别：

LT模式：当epoll_wait检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用epoll_wait时，会再次响应应用程序并通知此事件。

ET模式：当epoll_wait检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll_wait时，不会再次响应应用程序并通知此事件。

在 select/poll中，进程只有在调用一定方法后，内核才对所有监视的文件描述符进行扫描，而 epoll事先通过epoll_ctl()来注册一个文件描述符，一旦某个文件描述符就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，当进程调用epoll_wait时便得到通知（此处去掉了遍历文件描述符，而是通过监听回调的机制，这也是epoll的魅力所在）。

Epoll 的优点主要体现咋如下几个方面：

1. 监视的描述符不受限制，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，举个栗子，具体数目可以在cat/proc/sys/fs/file-max 查看，一般来说，这个数目和内存关系很大。
2. Select最大的缺点是进程打开的fd数目是有限制的，这对于连接数目较大的服务器来说根本不能满足，虽然也可以选择多进程的解决方案（Apache就是如此）；不过虽然linux上面创建进程的代价较小，但仍旧不可忽视，加上进程间数据同步远比不上线程间同步高效，所以并不是一种完美的解决方案。
3. IO的效率不会随着监视fd的数量的增长而下降，epoll不同于select和poll的轮询方式，而是通过每个fd定义的回调函数来实现，只有就绪的fd才会执行回调函数。
4. 如果没有大量的idle -connection或者dead-connection，epoll的效率并不会比select/poll高很多，但是当遇到大量的idle- connection，就会发现epoll的效率大大高于select/poll。

哈喽，我是小贺哥，就爱分享编程知识，如果觉得文章对你有帮助，别忘记关注我哦！



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复 「大礼包」
白嫖编程学习资料

关注公众号回复 「加群」 拉
你进百人技术交流群

十一、数据库

1、一二三范式

1) 第一范式，数据库表中的字段都是单一属性的，不可再分；每一个属性都是原子项，不可分割；如果实体中的某个属性有多个值时，必须拆分为不同的属性 通俗解释。1NF是关系模式应具备的最起码的条件，如果数据库设计不能满足第一范式，就不称为关系型数据库。也就是说，只要是关系型数据库，就一定满足第一范式。

2) 第二范式，数据库表中不存在非关键字段对任一候选关键字段的部分函数依赖，即符合第二范式；如果一个表中某一个字段A的值是由另外一个字段或一组字段B的值来确定的，就称为A函数依赖于B；当某张表中的非主键信息不是由整个主键函数来决定时，即存在依赖于该表中不是主键的部分或者依赖于主键一部分的部分时，通常会违反2NF。

3) 第三范式，在第二范式的基础上，数据表中如果不存在非关键字段对任一候选关键字段的传递函数依赖则符合3NF；第三范式规则查找以消除没有直接依赖于第一范式和第二范式形成的表的主键的属性。我们为没有与表的主键关联的所有信息建立了一张新表。每张新表保存了来自源表的信息和它们所依赖的主键；如果某一属性依赖于其他非主键属性，而其他非主键属性又依赖于主键，那么这个属性就是间接依赖于主键，这被称作传递依赖于主属性。通俗理解：一张表最多只存2层同类型信息 *。*

2、数据库的索引类型，数据库索引的作用

1) 数据库索引好比是一本书前面的目录，能加快数据库的查询速度。索引是对数据库表中一个或多个列（例如，employee 表的姓氏 (lname) 列）的值进行排序的结构。如果想按特定职员的姓来查找他或她，则与在表中搜索所有的行相比，索引有助于更快地获取信息。

2) 优点

大大加快数据的检索速度；创建唯一性索引，保证数据库表中每一行数据的唯一性；加速表和表之间的连接；在使用分组和排序子句进行数据检索时，可以显著减少查询中分组和排序的时间。

3) 缺点

索引需要占用数据表以外的物理存储空间；创建索引和维护索引要花费一定的时间；当对表进行更新操作时，索引需要被重建，这样降低了数据的维护速度。

4) 类型

唯一索引——UNIQUE，例如：create unique index stusno on student (sno)；表明此索引的每一个索引值只对应唯一的数据记录，对于单列唯一性索引，这保证单列不包含重复的值。对于多列唯一性索引，保证多个值的组合不重复。

主键索引——primary key，数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。

聚集索引（也叫聚簇索引）——cluster，在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引，如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。

5) 实现方式

B+树、散列索引、位图索引

3、聚集索引和非聚集索引的区别

1) 聚集索引表示表中存储的数据按照索引的顺序存储，检索效率比非聚集索引高，但对数据更新影响较大。非聚集索引表示数据存储在一个地方，索引存储在另一个地方，索引带有指针指向数据的存储位置，非聚集索引检索效率比聚集索引低，但对数据更新影响较小。

2) 聚集索引一个表只能有一个，而非聚集索引一个表可以存在多个。聚集索引存储记录是物理上连续存在，而非聚集索引是逻辑上的连续，物理存储并不连续。

4、唯一性索引和主键索引的区别

对于主键索引，oracle/sql server/mysql 等都会自动建立唯一索引；

- 主键不一定只包含一个字段，所以如果你在主键的其中一个字段建唯一索引还是必要的；
- 主键可作外键，唯一索引不可；
- 主键不可为空，唯一索引可以；
- 主键也可是多个字段的组合；
- 主键与唯一索引不同的是
- 主键索引有 not null 属性；
- 主键索引每个表只能有一个。

5、数据库引擎，innodb和myisam的特点与区别

1) Innodb引擎提供了对数据库ACID事务的支持，并且实现了SQL标准的四种隔离级别，关于数据库事务与其隔离级别的内容请见数据库事务与其隔离级别这篇文章。该引擎还提供了行级锁和外键约束，它的设计目标是处理大容量数据库系统，它本身其实就是基于MySQL后台的完整数据库系统，MySQL运行时Innodb会在内存中建立缓冲池，用于缓冲数据和索引。但是该引擎不支持FULLTEXT类型的索引，而且它没有保存表的行数，当SELECT COUNT(*) FROM TABLE时需要扫描全表。当需要使用数据库事务时，该引擎当然是首选。由于锁的粒

度更小，写操作不会锁定全表，所以在并发较高时，使用Innodb引擎会提升效率。但是使用行级锁也不是绝对的，如果在执行一个SQL语句时MySQL不能确定要扫描的范围，InnoDB表同样会锁全表。

2) MyISAM是MySQL默认的引擎，但是它没有提供对数据库事务的支持，也不支持行级锁和外键，因此当INSERT(插入)或UPDATE(更新)数据时即写操作需要锁定整个表，效率便会低一些。不过和Innodb不同，MyISAM中存储了表的行数，于是SELECT COUNT(*) FROM TABLE时只需要直接读取已经保存好的值而不需要进行全表扫描。如果表的读操作远远多于写操作且不需要数据库事务的支持，那么MyISAM也是很好的选择。

3) 大尺寸的数据集倾向于选择InnoDB引擎，因为它支持事务处理和故障恢复。数据库的大小决定了故障恢复的时间长短，InnoDB可以利用事务日志进行数据恢复，这会比较快。主键查询在InnoDB引擎下也会相当快，不过需要注意的是如果主键太长也会导致性能问题，关于这个问题我会在下文中讲到。大批的INSERT语句(在每个INSERT语句中写入多行，批量插入)在MyISAM下会快一些，但是UPDATE语句在InnoDB下则会更快一些，尤其是在并发量大的时候。

6、关系型和非关系型数据库的区别

数据库类型	特性	优点	缺点
关系型数据库 SQLite、 Oracle、 mysql	1、关系型数据库，是指采用了关系模型来组织数据的数据库； 2、关系型数据库的最大特点就是事务的一致性； 3、简单来说，关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。	1、容易理解：二维表结构是非常贴近逻辑世界一个概念，关系模型相对网状、层次等其他模型来说更容易理解； 2、使用方便：通用的SQL语言使得操作关系型数据库非常方便； 3、易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率； 4、支持SQL，可用于复杂的查询。	1、为了维护一致性所付出的巨大代价就是其读写性能比较差； 2、固定的表结构； 3、高并发读写需求； 4、海量数据的高效率读写；
非关系型数据库 MongoDB、 redis、 HBase	1、使用键值对存储数据； 2、分布式； 3、一般不支持ACID特性； 4、非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。	1、无需经过sql层的解析，读写性能很高； 2、基于键值对，数据没有耦合性，容易扩展； 3、存储数据的格式：nosql的存储格式是key,value形式、文档形式、图片形式等等，文档形式、图片形式等等，而关系型数据库则只支持基础类型。	1、不提供sql支持，学习和使用成本较高； 2、无事务处理，附加功能bi和报表等支持也不好；

7、数据库的隔离级别

1) 隔离级别高的数据库的可靠性高，但并发量低，而隔离级别低的数据库可靠性低，但并发量高，系统开销小。

2) READ UNCOMMITTED（未提交读），事务中的修改，即使没有提交，其他事务也可以看得到，比如说上面的两步这种现象就叫做脏读，这种隔离级别会引起很多问题，如无必要，不要随便使用；这就是事务还没提交，而别的事务可以看到他其中修改的数据的后果，也就是脏读；

3) READ COMMITTED（提交读），大多数数据库系统的默认隔离级别是READ COMMITTED，这种隔离级别就是一个事务的开始，只能看到已经完成的事务的结果，正在执行的，是无法被其他事务看到的。这种级别会出现读取旧数据的现象

4) REPEATABLE READ（可重复读），REPEATABLE READ解决了脏读的问题，该级别保证了每行的记录的结果是一致的，也就是上面说的读了旧数据的问题，但是却无法解决另一个问题，幻行，顾名思义就是突然蹦出来的行数据。指的就是某个事务在读取某个范围的数据，但是另一个事务又向这个范围的数据去插入数据，导致多次读取的时候，数据的行数不一致。虽然读取同一条数据可以保证一致性，但是却不能保证没有插入新的数据。

5) SERIALIZABLE（可串行化），SERIALIZABLE是最高的隔离级别，它通过强制事务串行执行（注意是串行），避免了前面的幻读情况，由于他大量加上锁，导致大量的请求超时，因此性能会比较底下，再特别需要数据一致性且并发量不需要那么大的时候才可能考虑这个隔离级别。

8、数据库连接池的作用

1) 在内部对象池中，维护一定数量的数据库连接，并对外暴露数据库连接的获取和返回方法，如外部使用者可通过getConnection方法获取数据库连接，使用完毕后再通过releaseConnection方法将连接返回，注意此时的连接并没有关闭，而是由连接池管理器回收，并为下一次使用做好准备。

2) 资源重用，由于数据库连接得到重用，避免了频繁创建、释放连接引起的大量性能开销。在减少系统消耗的基础上，增进了系统环境的平稳性（减少内存碎片以及数据库临时进程、线程的数量）

3) 更快的系统响应速度，数据库连接池在初始化过程中，往往已经创建了若干数据库连接置于池内备用。此时连接池的初始化操作均已完成。对于业务请求处理而言，直接利用现有可用连接，避免了数据库连接初始化和释放过程的时间开销，从而缩减了系统整体响应时间。

4) 新的资源分配手段，对于多应用共享同一数据库的系统而言，可在应用层通过数据库连接的配置，实现数据库连接技术。

5) 统一的连接管理，避免数据库连接泄露，较较为完备的数据库连接池实现中，可根据预先的连接占用超时设定，强制收回被占用的连接，从而避免了常规数据库连接操作中可能出现的资源泄露。

9、数据的锁的种类，加锁的方式

1) 锁是网络数据库中的一个非常重要的概念，当多个用户同时对数据库并发操作时，会带来数据不一致的问题，所以，锁主要用于多用户环境下保证数据库完整性和一致性。

2) 数据库锁出现的目的：处理并发问题；

3) 并发控制的主要采用的技术手段：乐观锁、悲观锁和时间戳。

4) 从数据库系统角度分为三种：排他锁、共享锁、更新锁。从程序员角度分为两种：一种是悲观锁，一种乐观锁。

10、数据库union join的区别

1) join 是两张表做交连后里面条件相同的部分记录产生一个记录集，union是产生的两个记录集(字段要一样的)并在一起，成为一个新的记录集。

2) union在数据库运算中会过滤掉重复数据，并且合并之后的是根据行合并的，即：如果a表和b表中的数据各有五行，且有两行是重复数据，合并之后为8行。运用场景：适合于需要进行统计的运算

3) union all是进行全部合并运算的，即：如果a表和b表中的数据各有五行，且有两行是重复数据，合并之后为10行。

4) join是进行表关联运算的，两个表要有一定的关系。即：如果a表和b表中的数据各有五行，且有两行是重复数据，根据某一列值进行笛卡尔运算和条件过滤，假如a表有2列，b表有2列，join之后是4列。

11、面试前必知的 MySQL 常用命令

启动与退出

指定 IP 地址和端口号登录 MySQL 数据库

命令格式为：

```
mysql -h ip -u root -p -P 3306
```

例如：

```
mysql -h 127.0.0.1 -u root -p -P 3306
```

退出 MySQL

使用 quit 或 exit 退出 MySQL

查看数据库

```
SHOW DATABASES ;
```

创建数据库

```
CREATE DATABASE IF NOT EXISTS dbname ;
```

选择数据库

```
USE 数据库名 ;
```

查看数据库中的数据表

```
SHOW TABLES ;
```

删除数据库

```
DROP DATABASE IF EXISTS dbname;
```

创建一个简单的数据库表

字段 类型(长度) 属性 索引

```
CREATE TABLE IF NOT EXISTS 表名(
id INT UNSTGND AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(255) NOT NULL
)ENGINE = InnoDB DEFAULT CHARSET=utf8;
```

添加数据

```
INSERT INTO table_name ( field1, field2,...fieldN )VALUES ( value1,
value2,...valueN );
```

查询数据

```
SELECT * FROM table ;
```

修改数据

```
UPDATE table SET 字段1 = '值1', 字段1='值2' WHERE 条件 ;
```

删除数据

```
DELETE FROM table WHERE 条件 ;
```

创建新普通用户

```
GRANT 权限 ON 库名.表名 TO '用户名'@'主机名' IDENTIFIED BY '密码'
```

查询所有用户

```
SELECT user,host FROM mysql.user;
```

删除普通用户

```
DROP USER '用户名'@'主机名' ;
```

修改 root 用户密码

```
SET PASSWORD = PASSWORD('新密码');
```

root 用户修改普通用户密码

```
SET PASSWORD FOR '用户名'@'主机名' =PASSWORD('新密码');
```

授权

```
GRANT 权限 ON 库名.表名 TO '用户名'@'主机名' IDENTIFIED BY '密码';  
GRANT SELECT,INSERT,UPDATE,DELETE ON cendxia.user TO '用户名'@'主机名'  
IDENTIFIED BY '密码';
```

查看权限

```
SHOW GRANTS FOR '用户名'@'主机名';
```

收回权限

```
REVOKE 权限 ON 库名.表名 FROM '用户名'@'主机名';
```

备份

```
mysqldump -u root -p 数据库名 > 要保存的位置
```

还原数据

```
mysql -u yser -p dbname < filename.sql;
```

建表引擎

MyISAM -- 读取速度快，不支持事务

InnoDB -- 读取速度稍慢 支持事务 事务回滚

一些常用属性

UNSTGND 无符号属性

AUTO_INCREMENT 自增属性(一般用在id字段上)

ZEROFILL 零填充

字符串类型

CHAR 定长的字符串类型 (0-255)个字符

VARCHAR 变长的字符串类型，5.0以前(0-255)个字符，5.0版本以后(0-65535)个字符

查看表结构

DESC 表名; (缩写版)

DESCRIBE 表名 ;

查看建表语句

SHOW CREATE TABLE 表名;

修改表名

ALTER TABLE 原表名 RENAME TO 新表名;

修改字段的数据类型

ALTER TABLE 表名 MODIFY 字段名 数据类型 属性 索引;

ALTER TABLE testalter_tbl MODIFY c CHAR(10);

修改字段名

ALTER TABLE 表名 CHANGE 原字段名 新字段名 数据类型 属性 索引;

增加字段

ALTER TABLE 表名 ADD 字段名 数据类型 属性 索引;

-- [FIRST|AFTER 字段名]

-- (FIRST 在最前面添加字段。AFTER 字段名 在某字段后面添加)

删除字段

```
ALTER TABLE 表名 DROP 字段名;
```

修改字段的排列位置

```
ALTER TABLE 表名 MODIFY 字段名 数据类型 属性 索引 AFTER 字段名;
```

修改表引擎

```
ALTER TABLE 表名 ENGINE=引擎名; --MyISAM 或 InnoDB
```

高级用法

```
explain sql;
```

explain 命令我们可以学习到该条 SQL 是如何执行的，随后解析 explain 的结果可以帮助我们使用更好的索引，最终来优化它！

通过 explain 命令我们可以知道以下信息：

表的读取顺序，数据读取操作的类型，哪些索引可以使用，哪些索引实际使用了，表之间的引用，每张表有多少行被优化器查询等信息。

```
mysql> explain select count(*) from app_summary where source = 'mi' and status = 1 and download_url1 != 'https://app.mi.comjavascipt:void(0)';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | type    | possible_keys | key     | key_len | ref      | rows   | Extra      |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | app_summary | ref    | source       | source  | 123    | const    | 488947 | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.18 sec)
```

格式化输出

```
sql \G
```

在命令最后面加上 \G 即可。

查看帮助

在 MySQL 提示符中输入 help;或者 \h 获取使用帮助。

链接: <https://mp.weixin.qq.com/s/QyGtqBhT4YehX7LYwg8WvQ>

最后, 数据库这块的知识点还有很多, 具体可参考 GitHub 上总结: https://github.com/rongweihe/CS_Offer



Hello GitHub

Chapter 1	Chapter 2	Chapter 3	Chapter 4	Chapter 5	Chapter 6	Chapter 7
错题+真题精解	编程语言	数据结构与算法	计算机网络	操作系统	数据库	Linux网络编程

后台开发基础知识 (持续更新中)

哈喽, 我是小贺哥, 就爱分享编程知识, 如果觉得文章对你有帮助, 别忘记关注我哦!



一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复 「大礼包」
白嫖编程学习资料

关注公众号回复 「加群」 拉
你进百人技术交流群

扫一扫, 关注「herongwei」公众号

十二、设计模式（设计和代码实现）

《大话设计模式》一书中提到 24 种设计模式，这 24 种设计模式没必要面面俱到，但一定要深入了解其中的几种，最好结合自己在实际开发过程中的例子进行深入的了解。

设计模式有 6 大设计原则：

单一职责原则：就一个类而言，应该仅有一个引起它变化的原因。

开放封闭原则：软件实体可以扩展，但是不可修改。即面对需求，对程序的改动可以通过增加代码来完成，但是不能改动现有的代码。

里氏代换原则：一个软件实体如果使用的是一个基类，那么一定适用于其派生类。即在软件中，把基类替换成派生类，程序的行为没有变化。

依赖倒转原则：抽象不应该依赖细节，细节应该依赖抽象。即针对接口编程，不要对实现编程。

迪米特原则：如果两个类不直接通信，那么这两个类就不应当发生直接的相互作用。如果一个类需要调用另一个类的某个方法的话，可以通过第三个类转发这个调用。

接口隔离原则：每个接口中不存在派生类用不到却必须实现的方法，如果不这样，就要将接口拆分，使用多个隔离的接口。

设计模式分为三类：

创造型模式：单例模式、工厂模式、建造者模式、原型模式

结构型模式：适配器模式、桥接模式、外观模式、组合模式、装饰模式、享元模式、代理模式

行为型模式：责任链模式、命令模式、解释器模式、迭代器模式、中介者模式、备忘录模式、观察者模式、状态模式、策略模式、模板方法模式、访问者模式。

介绍常见的几种设计模式：

单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

工厂模式：包括简单工厂模式、抽象工厂模式、工厂方法模式

简单工厂模式：主要用于创建对象。用一个工厂来根据输入的条件产生不同的类，然后根据不同类的虚函数得到不同的结果。

抽象工厂模式：定义了一个创建一系列相关或相互依赖的接口，而无需指定他们的具体类。

观察者模式：定义了一种一对多的关系，让多个观察对象同时监听一个主题对象，主题对象发

生变化时，会通知所有的观察者，使他们能够更新自己。

装饰模式：动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成派生类更为灵活。

1、说说面对对象中的设计原则

SRP (Single Responsibility Principle) : **单一职责原则**，就是说一个类只提供一种功能和仅有一个引起它变化的因素。

OCP (Open Close Principle) : **开放封闭原则**，就是对一个类来说，对它的内部修改是封闭的，对它的扩展是开放的。

DIP (Dependence Inversion Principle) : **依赖倒置原则**，就是程序依赖于抽象，而不依赖于实现，它的主要目的是为了降低耦合性，它一般通过反射和配置文件来实现的。

LSP (Liskov Substitution Principle) : **里氏替换原则**，就是基类出现的地方，通过它的子类也完全可以实现这个功能

ISP (Interface Segregation Principle) : **接口隔离原则**，建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。

CRP (Composite Reuse Principle) : **合成复用原则**，多用组合设计类，少用继承。

2、单一职责原则和接口隔离原则的区别

- 单一职责原则注重的是职责；而接口隔离原则注重对接口依赖的隔离。
- 单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口，主要针对抽象，针对程序整体框架的构建。

3、单例模式

有两种懒汉和饿汉：

饿汉：饿了就饥不择食了，所以在单例类定义的时候就进行实例化。

懒汉：顾名思义，不到万不得已就不会去实例化类，也就是在第一次用到的类实例的时候才会去实例化。

饿汉模式（线程安全）：

在最开始的时候静态对象就已经创建完成，设计方法是类中包含一个静态成员指针，该指针指向该类的一个对象，提供一个公有的静态成员方法，返回该对象指针，为了使得对象唯一，构造函数设为私有。

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

class SingleInstance {
public:
    static SingleInstance* GetInstance() {
        static SingleInstance ins;
        return &ins;
    }
    ~SingleInstance() {};
private:
    //涉及到创建对象的函数都设置为private
    SingleInstance() { std::cout<<"SingleInstance() 饿汉"<<std::endl;
}
    SingleInstance(const SingleInstance& other) {};
    SingleInstance& operator=(const SingleInstance& other) {return
*this;};
}

int main(){
    //因为不能创建对象所以通过静态成员函数的方法返回静态成员变量
    SingleInstance* ins = SingleInstance::GetInstance();
    return 0;
}
//输出 SingleInstance() 饿汉
```

懒汉模式（线程安全需要加锁）：

尽可能的晚的创建这个对象的实例，即在单例类第一次被引用的时候就将自己初始化，C++很多地方都有类型的思想，比如写时拷贝，晚绑定等。

```
#include <pthread.h>
#include <iostream>
#include <algorithm>
using namespace std;

class SingleInstance {
public:
    static SingleInstance* GetInstance() {
        if (ins == nullptr) {
            pthread_mutex_lock(&mutex);
            if (ins == nullptr) {
                ins = new SingleInstance();
            }
            pthread_mutex_unlock(&mutex);
        }
        return ins;
    }
    ~SingleInstance() {};
    //互斥锁
    static pthread_mutex_t mutex;
private:
    //涉及到创建对象的函数都设置为private
    SingleInstance() { std::cout<<"SingleInstance() 懒汉"<<std::endl;
    }
    SingleInstance(const SingleInstance& other) {};
    SingleInstance& operator=(const SingleInstance& other) { return
    *this; }
    //静态成员
    static SingleInstance* ins;
};
```

```
//懒汉式 静态变量需要定义
SingleInstance* SingleInstance::ins = nullptr;
pthread_mutex_t SingleInstance::mutex;

int main(){
    //因为不能创建对象所以通过静态成员函数的方法返回静态成员变量
    SingleInstance* ins = SingleInstance::GetInstance();
    delete ins;
    return 0;
}
//输出 SingleInstance() 懒汉
```

单例模式的适用场景

- (1) 系统只需要一个实例对象，或者考虑到资源消耗的太大而只允许创建一个对象。
- (2) 客户调用类的单个实例只允许使用一个公共访问点，除了该访问点之外不允许通过其它方式访问该实例（就是共有的静态方法）。

4、工厂模式

简单工厂模式：

就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建。简单工厂模式的实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类（这些产品类继承自一个父类或接口）的实例。

```
#include <iostream>
#include <pthread.h>
using namespace std;

//产品类（抽象类，不能实例化）
class Product{
public:
    Product(){}
    virtual void show()=0; //纯虚函数
```

```
};

class productA : public Product{
public:
    productA(){}
    void show(){ std::cout << "product A create!" << std::endl; };
    ~productA(){}
};

class productB : public Product{
public:
    productB(){}
    void show(){ std::cout << "product B create!" << std::endl; };
    ~productB(){}
};

class simpleFactory{ // 工厂类
public:
    simpleFactory(){}
    Product* product(const string str){
        if (str == "productA")
            return new productA();
        if (str == "productB")
            return new productB();
        return NULL;
    }
};

int main(){
    simpleFactory obj; // 创建工厂
    Product* pro; // 创建产品
    pro = obj.product("productA");
    pro->show(); // product A create!
    delete pro;

    pro = obj.product("productB");
```

```
    pro->show(); // product B create!
    delete pro;
    return 0;
}
```

工厂模式目的就是代码解耦，如果我们不采用工厂模式，如果要创建产品 A、B，通常做法采用用 switch...case 语句，那么想一想后期添加更多的产品进来，我们不是要添加更多的 switch...case 吗？这样就很麻烦，而且也不符合设计模式中的**开放封闭原则**。

为了进一步解耦，在简单工厂的基础上发展出了抽象工厂模式，即连工厂都抽象出来，实现了进一步代码解耦。

代码如下：

```
#include <iostream>
#include <pthread.h>
using namespace std;

//产品类（抽象类，不能实例化）
class Product{
public:
    Product(){}
    virtual void show()=0; //纯虚函数
};

class Factory{//抽象类
public:
    virtual Product* CreateProduct()=0;//纯虚函数
};

//产品A
class ProductA:public Product{
public:
    ProductA(){}
    void show(){ std::cout<<"product A create!"<<std::endl; };
};


```

```
//产品B
class ProductB:public Product{
public:
    ProductB(){}
    void show(){ std::cout<<"product B create!"<<std::endl; };
};

//工厂类A，只生产A产品
class FactorA: public Factory{
public:
    Product* CreateProduct(){
        Product* product_ = nullptr;
        product_ = new ProductA();
        return product_;
    }
};

//工厂类B，只生产B产品
class FactorB: public Factory{
public:
    Product* CreateProduct(){
        Product* product_ = nullptr;
        product_ = new ProductB();
        return product_;
    }
};

int main(){
    Product* product_ = nullptr;
    auto MyFactoryA = new FactorA();
    product_ = MyFactoryA->CreateProduct();// 调用产品A的工厂来生产A产品
    product_->show();
    delete product_;

    auto MyFactoryB=new FactorB();
    product_ = MyFactoryB->CreateProduct();// 调用产品B的工厂来生产B产品
    product_->show();
}
```

```
    delete product_;

    return 0;
}

//输出
//product A create! product B create!
```

5、观察者模式

观察者模式：定义一种一（被观察类）对多（观察类）的关系，让多个观察对象同时监听一个被观察对象，被观察对象状态发生变化时，会通知所有的观察对象，使他们能够更新自己的状态。

观察者模式中存在两种角色：

观察者：内部包含被观察者对象，当被观察者对象的状态发生变化时，更新自己的状态。（接收通知更新状态）

被观察者：内部包含了所有观察者对象，当状态发生变化时通知所有的观察者更新自己的状态。（发送通知）

应用场景：

当一个对象的改变需要同时改变其他对象，且不知道具体有多少对象有待改变时，应该考虑使用观察者模式；

一个抽象模型有两个方面，其中一方面依赖于另一方面，这时可以用观察者模式将这两者封装在独立的对象中使它们各自独立地改变和复用。

实现方式：

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

class Subject;
//观察者 基类 （内部实例化了被观察者的对象sub）
class Observer {
protected:
```

```
    string name;
    Subject *sub;

public:
    Observer(string name, Subject *sub) {
        this->name = name;
        this->sub = sub;
    }
    virtual void update() = 0;
};

class StockObserver : public Observer {
public:
    StockObserver(string name, Subject *sub) : Observer(name, sub){}
    void update();
};

class NBAObserver : public Observer {
public:
    NBAObserver(string name, Subject *sub) : Observer(name, sub){}
    void update();
};

//被观察者 基类 (内部存放了所有的观察者对象，以便状态发生变化时，给观察者发通知)
class Subject {
protected:
    std::list<Observer *> observers;
public:
    string action; //被观察者对象的状态
    virtual void attach(Observer *) = 0;
    virtual void detach(Observer *) = 0;
    virtual void notify() = 0;
};

class Secretary : public Subject {
    void attach(Observer *observer) {
        observers.push_back(observer);
    }
};
```

```
}

void detach(Observer *observer) {
    list<Observer *>::iterator iter = observers.begin();
    while (iter != observers.end()) {
        if ((*iter) == observer) {
            observers.erase(iter);
            return;
        }
        ++iter;
    }
}

void notify() {
    list<Observer *>::iterator iter = observers.begin();
    while (iter != observers.end()) {
        (*iter)->update();
        ++iter;
    }
}

};

void StockObserver::update() {
    cout << name << " 收到消息: " << sub->action << endl;
    if (sub->action == "老板来了!") {
        cout << "我马上关闭股票，装做很认真工作的样子！" << endl;
    }
}

void NBAObserver::update() {
    cout << name << " 收到消息: " << sub->action << endl;
    if (sub->action == "老板来了!") {
        cout << "我马上关闭 NBA，装做很认真工作的样子！" << endl;
    }
}

int main()
{
```

```
Subject *BOSS = new Secretary();
Observer *xa = new NBAObserver("xa", BOSS);
Observer *xb = new NBAObserver("xb", BOSS);
Observer *xc = new StockObserver("xc", BOSS);

BOSS->attach(xz);
BOSS->attach(xb);
BOSS->attach(xc);

BOSS->action = "去吃饭了!";
BOSS->notify();
cout << endl;
BOSS->action = "老板来了!";
BOSS->notify();
return 0;

}

//输出
//product A create! product B create!
```

6、装饰器模式

装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，**同时又不改变其结构**。

这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

代码没有改变 Car 类的内部结构，还为其增加了新的功能，这就是装饰器模式的作用。

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

//抽象构件类 Transform (变形金刚)
class Transform{
public:
```

```
virtual void move() = 0;
};

//具体构件类Car
class Car : public Transform{
public:
    Car(){
        std::cout << "变形金刚是一辆车！" << endl;
    }
    void move(){
        std::cout << "在陆地上移动。" << endl;
    }
};

//抽象装饰类
class Changer : public Transform{
public:
    Changer(shared_ptr<Transform> transform){
        this->transform = transform;
    }
    void move(){
        transform->move();
    }
private:
    shared_ptr<Transform> transform;
};

//具体装饰类Robot
class Robot : public Changer{
public:
    Robot(shared_ptr<Transform> transform) : Changer(transform){
        std::cout << "变成机器人！" << std::endl;
    }

    void say(){
        std::cout << "说话！" << std::endl;
    }
}
```

```
    }

};

//具体装饰类AirPlane
class Airplane : public Changer{
public:
    Airplane(shared_ptr<Transform> transform) : Changer(transform){
        std::cout << "变成飞机!" << std::endl;
    }

    void say(){
        std::cout << "在天空飞翔!" << std::endl;
    }
};

int main(void){
    shared_ptr<Transform> camaro = make_shared<Car>();
    camaro->move();
    std::cout << "-----" << endl;
    shared_ptr<Robot> bumblebee = make_shared<Robot>(camaro);
    bumblebee->move();
    bumblebee->say();
    return 0;
}

/*
输出
变形金刚是一辆车!
在陆地上移动。
-----
变成机器人!
在陆地上移动。
说话!
-----
变成飞机!
在陆地上移动。
在天空飞翔!
```

*/

参考：

[https://www.nowcoder.com/discuss/632757,](https://www.nowcoder.com/discuss/632757)

<https://leetcode-cn.com/leetbook/read/cpp-interview-highlights/o52f77/>

以上部分知识整理来源于网络，侵权必删。

哈喽，我是小贺哥，就爱分享编程知识，如果觉得文章对你有帮助，别忘记关注我哦！



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复「大礼包」
白嫖编程学习资料

关注公众号回复「加群」拉
你进百人技术交流群

加餐-图解 STL 源码

十三、千字长文 30 图解陪你手撕 STL 空间配置器源码

大家好，我是小贺。

文章每周持续更新，可以微信搜索公众号「herongwei」第一时间阅读和催更。

本文 GitHub : <https://github.com/rongweihe/CPPNotes> 已经收录，有一线大厂面试点思维导图，也整理了很多我的文档，欢迎点个小和完善。一起加油，变得更好！

13.1 前言

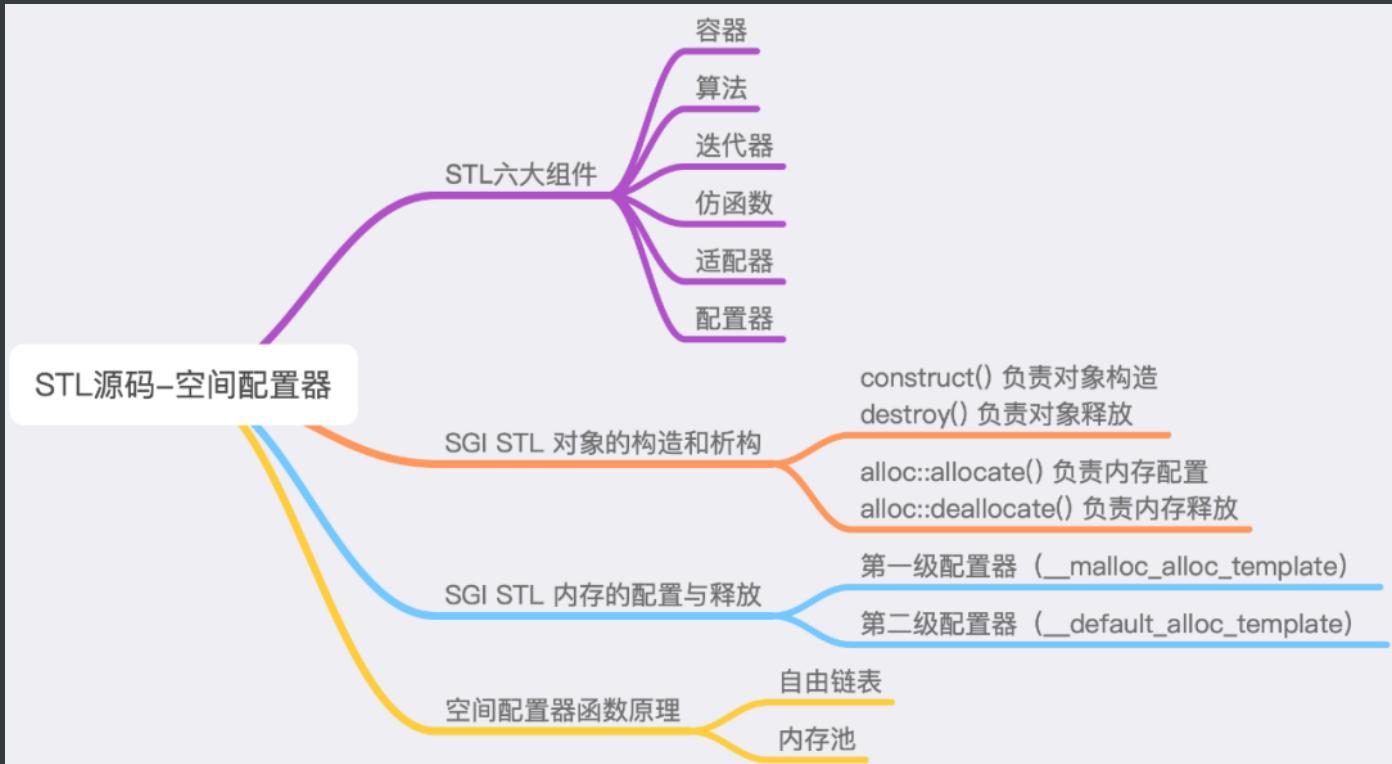
天下大事，必作于细。

源码之前，了无秘密。

你清楚下面这几个问题吗？

- 调用 new 和 delete 时编译器底层到底做了哪些工作？
- STL 器底层空间配置原理是怎样的？
- STL 空间配置器到底要考虑什么？
- 什么是内存的配置和释放？

这篇，我们就来回答这些问题。



13.2 STL 六大组件

在深入配置器之前，我们有必要了解下 STL 的背景知识：

标准模板库（英文：Standard Template Library，缩写：STL），是一个 C++ 软件库。

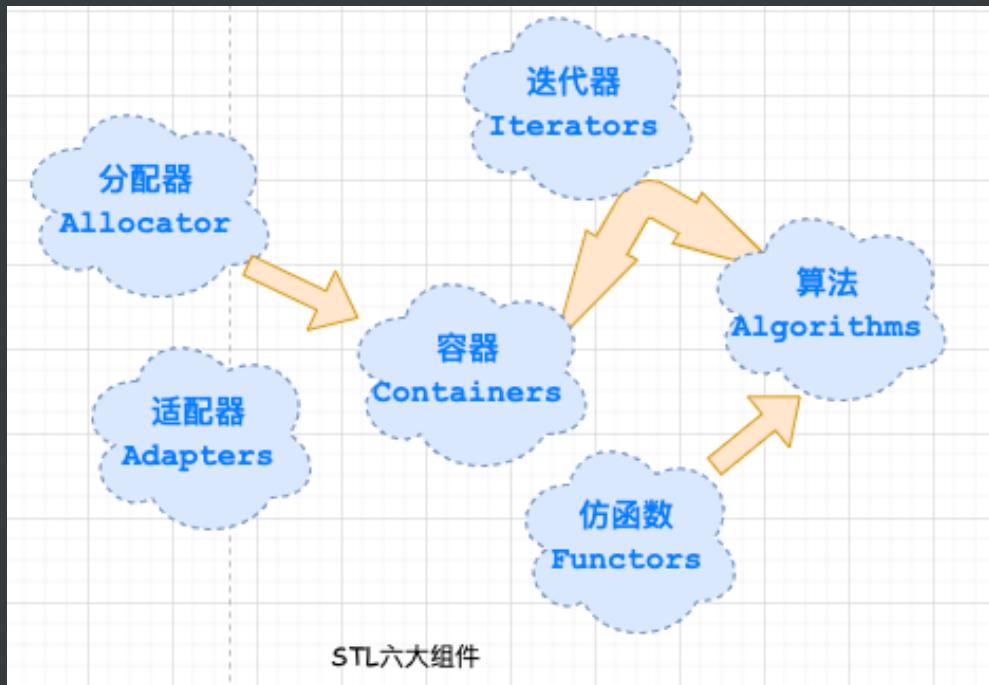
STL 的价值在于两个方面，就底层而言，STL 带给我们一套极具实用价值的零部件以及一个整合的组织；除此之外，STL 还带给我们一个高层次的、以泛型思维 (Generic Paradigm) 为基础的、系统化的“软件组件分类学”。

STL 提供六大组件，了解这些为接下来的阅读打下基础。

- 容器 (containers) : 各种数据结构，如 `vector`, `list`, `deque`, `set`, `map` 用来存放数据。从实现的角度来看，STL 容器是一种 class template。
- 算法 (algorithms) : 各种常用的算法如 `sort`, `search`, `copy`, `erase`...从实现角度来看，STL 算法是一种 function template。
- 迭代器 (iterators) : 扮演容器与算法之间的胶合剂，是所谓的“泛型指针”。从实现角度来看，迭代器是一种将 `operator *`, `operator ->`, `operator++`, `operator-` 等指针相关操作予以重载的class template。
- 仿函数 (functors) : 行为类似函数，可以作为算法的某种策略。从实现角度来看，仿函

数是一种重载了 operator() 的 class 或 class template。

- 适配器 (adapters)：一种用来修饰容器或仿函数或迭代器接口的东西。例如 STL 提供的 queue 和 stack，虽然看似容器，其实只能算是一种容器适配器，因为它们的底部完全借助 deque，所有操作都由底层的 deque 供应。
- 配置器 (allocator)：负责空间配置与管理，从实现角度来看，配置器是一个实现了动态空间配置、空间管理、空间释放的 class template。



13.3 何为空间配置器

3.1 为何需要先了解空间配置器？

从使用 STL 层面而言，空间配置器并不需要介绍，因为容器底层都给你包装好了，但若是从 STL 实现角度出发，空间配置器是首要理解的。

作为 STL 设计背后的支撑，空间配置器总是在默默地付出着。为什么你可以使用算法来高效地处理数据，为什么你可以对容器进行各种操作，为什么你用迭代器可以遍历空间，这一切的一切，都有“空间配置器”的功劳。

3.2 SGI STL 专属空间配置器

SGI STL 的空间配置器与众不同，且与 STL 标准规范不同。

其名为 alloc，而非 allocator。

虽然 SGI 也配置了 allocatalor，但是它自己并不使用，也不建议我们使用，原因是效率比较感人，因为它只是在基层进行配置/释放空间而已，而且不接受任何参数。

SGI STL 的每一个容器都已经指定缺省的空间配置器是 alloc。



```
template <class T, class Alloc = alloc> //预设使用alloc配置器
class vector{...};
```

在 C++ 里，当我们调用 new 和 delete 进行对象的创建和销毁的时候，也同时会有内存配置操作和释放操作：

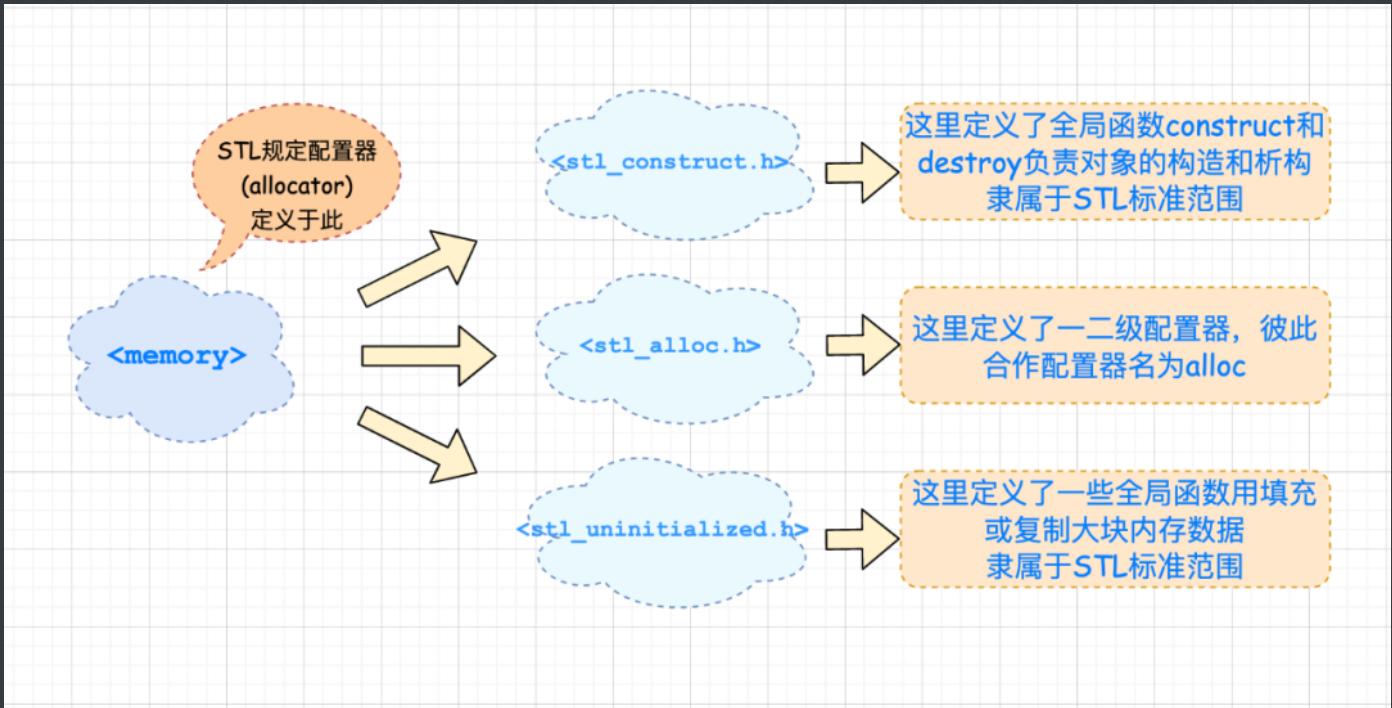
这其中的 new 和 delete 都包含两阶段操作：

- 对于 new 来说，编译器会先调用 ::operator new 分配内存；然后调用 Obj::Obj() 构造对象内容。
- 对于 delete 来说，编译器会先调用 Obj::~Obj() 析构对象；然后调用 ::operator delete 释放空间。

为了精密分工，STL allocator 决定将这两个阶段操作区分开来。

- 对象构造由 ::construct() 负责；对象释放由 ::destroy() 负责。
- 内存配置由 alloc::allocate() 负责；内存释放由 alloc::deallocate() 负责；

STL配置器定义在 `<new>` 中，下图直观的描述了这一框架结构



13.4 构造和析构源码

我们知道，程序内存的申请和释放离不开基本的构造和析构基本工具：construct() 和 destroy()。

在 STL 里面，construct() 函数接受一个指针 P 和一个初始值 value，该函数的用途就是将初值设定到指针所指的空间上。

destroy() 函数有两个版本，第一个版本接受一个指针，准备将该指针所指之物析构掉。直接调用析构函数即可。

第二个版本接受 first 和 last 两个迭代器，将[first,last)范围内的所有对象析构掉。



```
#include <new.h>      //欲使用placement new, 需要包含此文件

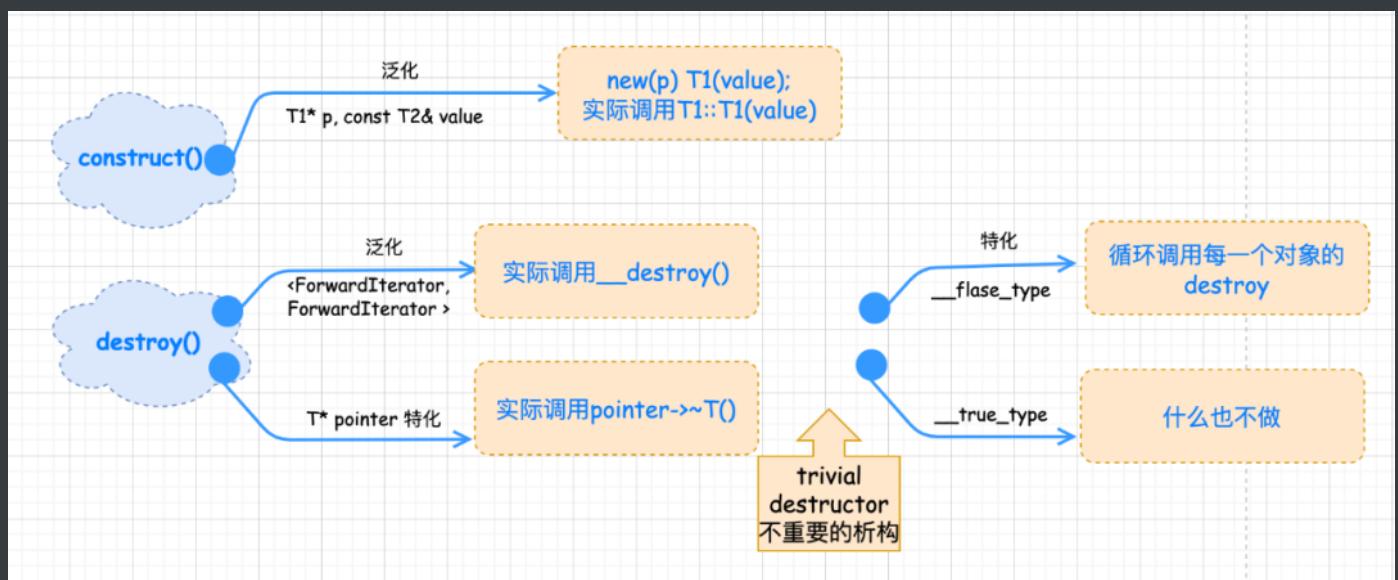
template <class T1, class T2>
inline void construct(T1*p, const T2& value) {
    new (p) T1(value); //placement new;调用T1::T1(value)
}

//以下是destroy()第一版本, 接受一个指针
template <class T>
inline void destroy(T* pointer) {
    pointer->~T(); //调用dtor ~T()
}

//以下是destroy()第二版本, 接受两个迭代器, 此函数设法找出元素的数值类型
//进而利用__type_traits<>求取最适当措施
template <class ForwardIterator>
inline void destroy(ForwardIterator first, ForwardIterator last) {
    __destroy(first, last, value_type(first));
}
```

其中 `destroy()` 只截取了部分源码，全部实现还考虑到特化版本，比如判断元素的数值类型 (`value type`) 是否有 `trivial destructor` 等限于篇幅，完整代码请参阅《STL 源码剖析》。

再来张图吧，献丑了。



13.5 内存的配置与释放

前面所讲都是对象的构造和析构，接下来要讲的是对象构造和析构背后的故事—（内存的分配与释放），这块是才真正的硬核，不要搞混了哦。

5.1 真·`alloc` 设计奥义

对象构造和析构之后的内存管理诸项事宜，由 `<stl_alloc.h>` 一律负责。SGI 对此的设计原则如下：

- 向 system heap 要求空间
- 考虑多线程 (multi-threads) 状态
- 考虑内存不足时的应变措施
- 考虑过多“小型区块”可能造成的内存碎片 (fragment) 问题

考虑到小型区块可能造成的内存破碎问题，SGI 为此设计了双层级配置器。当配置区块超过 128bytes 时，称为足够大，使用第一级配置器，直接使用 `malloc()` 和 `free()`。

当配置区块不大于 128bytes 时，为了降低额外负担，直接使用第二级配置器，采用复杂的 memory pool 处理方式。

无论使用第一级配接器（**malloc_alloc_template**）或是第二级配接器（**default_alloc_template**），alloc 都为其包装了接口，使其能够符合 STL 标准。



```
#ifdef __USE_MALLOC
...
typedef __malloc_alloc_template<0> malloc_alloc;
typedef malloc_alloc alloc; // 令 alloc 为第一级配置器
#else
...
typedef __default_alloc_template<__NODE_ALLOCATOR_THREADS, 0>alloc; // 令 alloc 为第二级配置器
#endif /* !__USE_MALLOC */

template <class T, class Alloc>
class simple_alloc {
public:
    static T *allocate(size_t n){
        return 0==n ? 0 : (T*)Alloc::allocate(n*sizeof(T));
    }
    static T *allocate(void){
        return (T*) Alloc::allocate(sizeof(T));
    }
    static void deallocate(T *p, size_t n){
        if (0!=n) Alloc::deallocate(p, n*sizeof(T));
    }
    static void deallocate (T *p){
        Alloc::deallocate(p, sizeof(T));
    }
};
```

其中，**__malloc_alloc_template** 就是第一级配置器；

__default_alloc_template 就是第二级配置器。

这么一大堆源码看懵了吧，别着急，请看下图。

SGI STL 第一级配置器

```
template<int inst>
class __malloc_alloc_template {...}
```

其中：

- 1、allocate()直接使用malloc(),
deallocate()直接使用free().
- 2、模拟C++的set_new_handler()以处理内存不足的情况。

SGI STL 第二级配置器

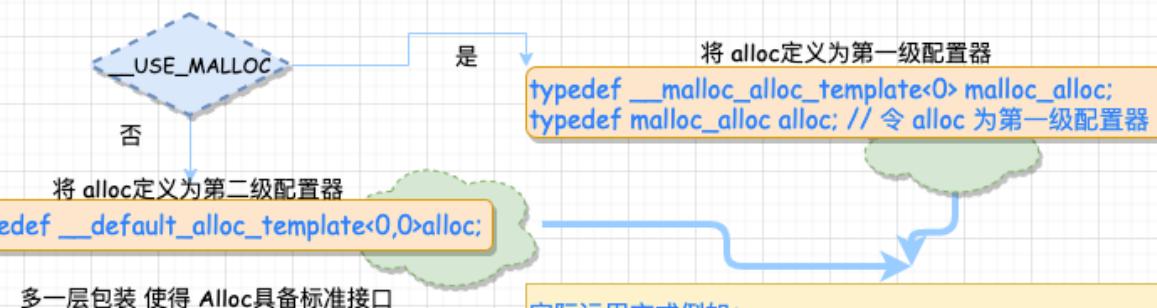
```
template<bool threads, int inst>
class __default_alloc_template {...}
```

其中：

- 1、维护 16 个自由链表(free list), 负责16种小型区块的次配置能力。
内存池以malloc配置而得，如果内存不足转调用第一级配置器。
- 2、如果需求区块小于 128bytes 转调用第一级配置器。

SGI STL 第一级配置器和第二级配置器

其中 SGI STL 将配置器多了一层包装使得 Alloc 具备标准接口。



实际运用方式例如：

```
template <class T, class Alloc=alloc>
class simple_alloc {
public:
    static T *allocate(size_t);
    static T *allocate(void);
    static void deallocate(T *p, size_t n);
    static void deallocate(T *p);
};
```

实际运用方式例如：

```
template <class T, class Alloc=alloc>
class vector {
    typedef simple_alloc<T, Alloc> data_allocator;
    data_allocator::allocate(n); // 配置n个元素
    // 配置完成之后接下来必须设定初值
};

// 又例
template <class T, class Alloc=alloc, size_t BufSize=0>
class deque {
    typedef simple_alloc<T, Alloc> data_allocator;
    typedef simple_alloc<T*, Alloc> map_allocator;
    data_allocator::allocate(n); // 配置n个元素
    map_allocator::allocate(n); // 配置n个节点
    // 配置完成之后接下来必须设定初值
};
```

第一级配置器和第二级配置器的包装接口和运用方式

13.6 alloc 一级配置器源码解读

这里截取部分（精华）解读

(1) 第一级配置器以 malloc(), free(), realloc() 等 C 函数执行实际的内存配置、释放和重配置操作，并实现类似 C++ new-handler 的机制（因为它并非使用 ::operator new 来配置内存，所以不能直接使用C++ new-handler 机制）。

(2) SGI 第一级配置器的 allocate() 和 reallocate() 都是在调用malloc() 和 realloc() 不成功后，改调用 oom_malloc() 和oom_realloc()。



```
//malloc-based allocator.通常比后面介绍的default alloc速度慢
//一般而言是thread-safe，并且对于空间的运用比较高效(efficient)
//以下是第一级配置器
//注意，无“template类型参数”，至于“非类型参数”inst，则完全没派上用场
template <int inst>
class __malloc_alloc_template {

private:
//以下函数将用来处理内存不足的情况
//oom:out of memory
    static void *oom_malloc(size_t);
    static void *oom_realloc(void *, size_t);
    static void (* __malloc_alloc_oom_handler) ();

public:
    static void * allocate(size_t n){
        void *result = malloc(n);    //第一级配置器直接使用malloc()
        //以下无法满足需求时，改用oom_malloc
        if (0==result) result = oom_malloc(n);
        return result;
    }

    static void * deallocate(void *p, size_t /*n*/ ) {
        free(p);      //第一级配置器直接使用free()
    }

    static void *realloc(void *p, size_t /*old_sz*/, size_t new_sz) {
        void *result = realloc(p, new_sz);    //第一级配置器直接使用realloc()
        //以下无法满足需求时，改用oom_realloc()
        if (0==result) result = oom_realloc(p, new_sz);
        return result;
    }

    //以下仿真C++的set_new_handler()，换句话说，你可以通过它指定你自己的out-of-memory handler
    static void (* set_malloc_handler(void (*f)())()) {
        void (*old) () = __malloc_alloc_oom_handler;
        __malloc_alloc_oom_handler = f;
        return (old);
    }
};
```

(3) oom_malloc() 和 oom_realloc() 都有内循环，不断调用“内存不足处理例程”，期望某次调用后，获得足够的内存而圆满完成任务，哪怕有一丝希望也要全力以赴申请啊，如果用户并没有指定“内存不足处理程序”，这个时候便无力乏天，真的是没内存了，STL 便抛出异常。或调用exit(1) 终止程序。



```
//malloc_alloc out-of-memory handling
//初值为0，有待客户端设定
template <int inst>
void (* __malloc_alloc_template<inst>::__malloc_alloc_oom_handler)() = 0;

template <int inst>
void *__malloc_alloc_template<inst>::oom_malloc(size_t n) {
    void (* my_malloc_handler)();
    void *result;

    for (;;){ //不断尝试释放、配置、再释放、在配置...
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler) {__THROW_BAD_ALLOC;}
        (*my_malloc_handler)(); //调用处理例程，企图释放内存
        result = malloc(n); //再次尝试配置内存
        if (result) return (result);
    }
}

template <int inst>
void *__malloc_alloc_template<inst>::oom_realloc(void *p, size_t n) {
    void (*my_malloc_handler)();
    void *result;

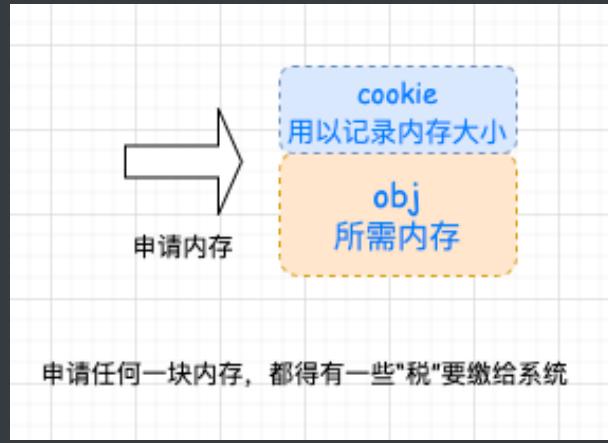
    for(;;) { //不断尝试释放、配置、再释放、在配置...
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler) {__THROW_BAD_ALLOC;}
        (*my_malloc_handler)(); //调用处理例程，企图释放内存
        result = realloc(p, n); //再次尝试配置内存
        if (result) return (result);
    }
}

//注意，以下直接将参数inst指定为0
type __malloc_alloc_template<0> malloc_alloc;
```

13.7 alloc 二级配置器源码解读

照例，还是截取部分（精华）源码解读。看累了嘛，远眺歇会，回来继续看，接下来的这部分，将会更加的让我们为大师的智慧折服！

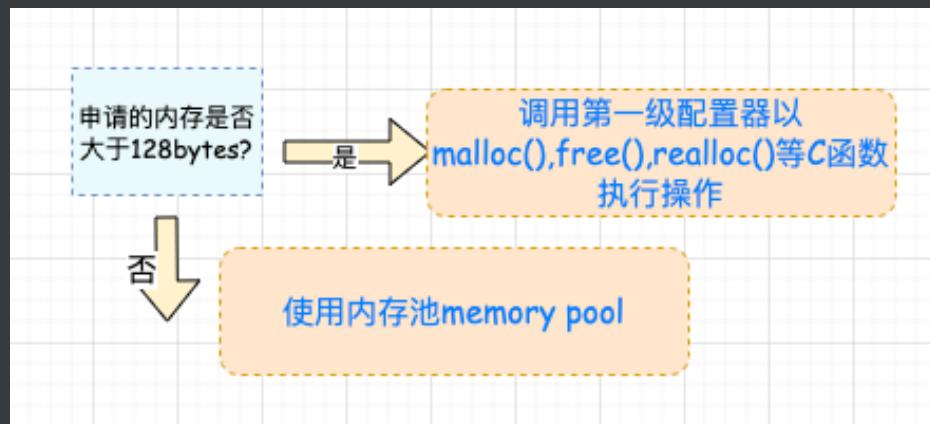
第二级配置器多了一些机制，专门针对内存碎片。内存碎片化带来的不仅仅是回收时的困难，配置也是一个负担，额外负担永远无法避免，毕竟系统要划出这么多的资源来管理另外的资源，但是区块越小，额外负担率就越高。



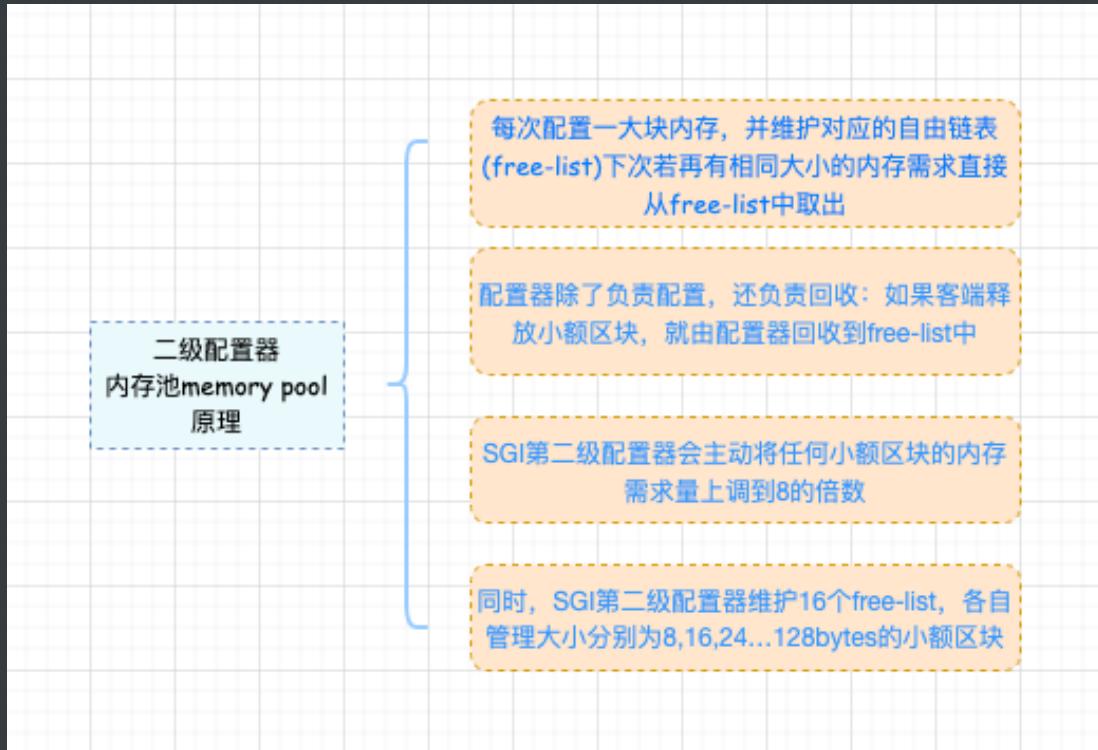
7.1 SGI 第二级配置器到底解决了多少问题呢？

简单来说 SGI第二级配置器的做法是：sub-allocation （层次架构）：

前面也说过了， SGI STL 的第一级配置器是直接使用 `malloc()`, `free()`, `realloc()` 并配合类似 C++ new-handler 机制实现的。第二级配置器的工作机制要根据区块的大小是否大于 128bytes 来采取不同的策略：



继续跟上节奏，上面提到了 `memory pool`，相信程序员朋友们很熟悉这个名词了，没错，这就是二级配置器的精髓所在，如何理解？请看下图：



有了内存池，是不是就可以了，当然没有这么简单。上图中还提到了自由链表，这个又是何方神圣？

我们来一探究竟！

7.2 自由链表自由在哪？又该如何维护呢？

我们知道，一方面，自由链表中有些区块已经分配给了客户端使用，所以 free_list 不需要再指向它们；另一方面，为了维护 free-list，每个节点还需要额外的指针指向下一个节点。

那么问题来了，如果每个节点有两个指针？这不就造成了额外负担吗？本来我们设计 STL 容器就是用来保存对象的，这倒好，对象还没保存之前，已经占据了额外的内存空间了。那么，有方法解决吗？当然有！再来感受一次大师的智慧！

(1) 在这之前我们先来了解另一个概念——union（联合体/共用体），对 union 已经熟悉的读者可以跳过这一部分的内容；如果忘记了也没关系，趁此来回顾一下：

- (a) 共用体是一种特殊的类，也是一种构造类型的数据结构。
- (b) 共用体表示几个变量共用一个内存位置，在不同的时间保存不同的数据类型和不同长度的变量。

(c) 所有的共用体成员共用一个空间，并且同一时间只能储存其中一个成员变量的值。例如如下：

```
union ChannelManager {  
    char ch;  
    int num;  
    char *str;  
    double exp;  
}
```

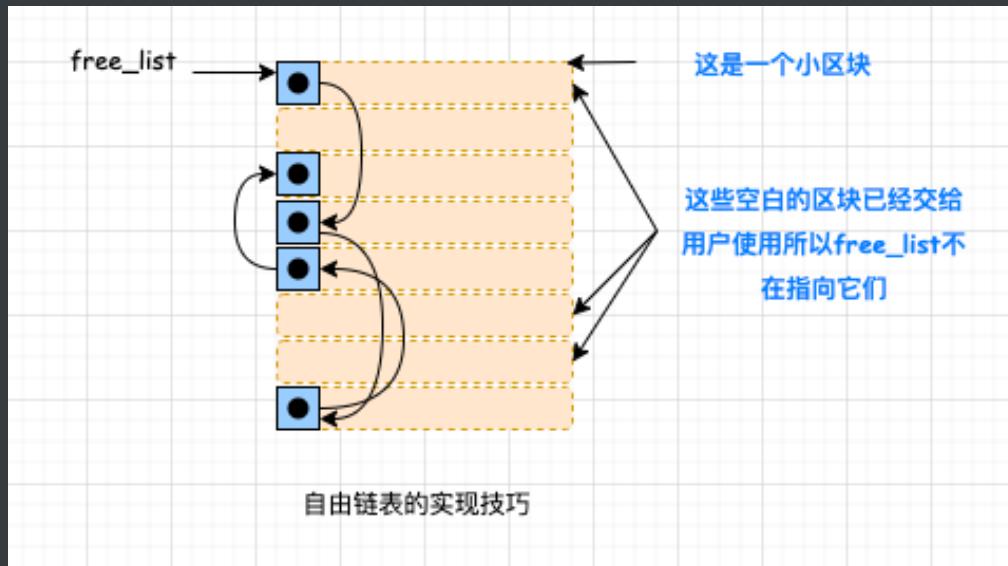
一个union 只配置一个足够大的空间以来容纳最大长度的数据成员，以上例而言，最大长度是double 类型，

所以 ChannelManager 的空间大小就是 double 数据类型的大小。在 C++ 里，union 的成员默认属性页为 public。union 主要用来压缩空间，如果一些数据不可能在同一时间同时被用到，则可以使用 union。

(2) 了解了 union 之后，我们可以借助 union 的帮助，先来观察一下 free-list 节点的结构

```
union obj {  
    union obj* free_list_link;  
    char client_data[1];//the client sees this.  
}
```

来深入了解 free_list 的实现技巧，请看下图。



在 union obj 中，定义了两个字段，再结合上图来分析：

从第一个字段看，obj 可以看做一个指针，指向链表中的下一个节点；

从第二个字段看，obj 可以也看做一个指针，不过此时是指向实际的内存区。

一物二用的好处就是不会为了维护链表所必须的指针而造成内存的另一种浪费，或许这就是所谓的自由奥义所在！大师的智慧跃然纸上。

7.3 第二级配置器的部分实现内容

到这里，我们已经基本了解了第二级配置器中 free_list 的工作原理了。附上第二级配置器的部分实现内容源码：

```
enum {__ALIGN = 8}; //小区块的上调边界
enum {__MAX_BYTES = 128}; //小型区块的上限
enum {__NFREELISTS = __MAX_BYTES/__ALIGN}; //free-list个数

//以下是第二级配置器
//注意，无“template类型参数”，且第二参数完全没派上用场
//第一参数用于多线程环境下。这里不讨论多线程环境
template <bool threads, int inst>
class __default_alloc_template {

private:
```

```

private:
    //ROUND_UP( )将bytes上调至8的倍数
    static size_t ROUND_UP(size_t bytes) {
        return (((bytes) + __ALIGN-1) & ~(__ALIGN-1));
    }
private:
    union obj {
        union obj * free_list_link;
        char client_data[1];
    };
private:
    //16个free-lists
    static obj *volatile free_list [__NFREELISTS];
    //以下函数根据区块大小，决定使用第n号free-list.n从0起算
    static size_t FREELIST_INDEX(size_t bytes) {
        return (((bytes) + __ALIGN-1)/__ALIGN-1);
    }

    //返回一个大小为n的对象，并可能加入大小为n的其他区块到free list
    static void *refill(size_t n);
    //配置一大块空间，可容纳nobjs个大小为“size”的区块
    //如果配置nobjs个区块有所不便，nobjs可能会降低
    static char *chunk_alloc(size_t size, int &nobjs);

    //Chunk allocation state
    static char *start_free;      //内存池起始位置。只在chunk_alloc()中变化
    static char *end_free;        //内存池结束位置。只在chunk_alloc()中变化
    static size_t heap_size;

public:
    static void *allocate(size_t n) { /*详述于后*/}
    static void deallocate(void *p, size_t n) { /*详述于后*/}
    static void *realloc(void *p, size_t old_sz, size_t new_sz);
};

//以下是static data member的定义与初始值设定
template <bool threads, int inst>
char *__default_alloc_template<threads, inst>::start_free = 0;

template <bool threads, int inst>
char *__default_alloc_template<threads, inst>::end_free = 0;

template <bool threads, int inst>
size_t __default_alloc_template<threads, inst>::heap_size = 0;

template <bool threads, int inst>
__default_alloc_template<threads, inst>::obj *volatile
__default_alloc_template<threads, inst>::free_list[__NFREELISTS] =
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

```

太长了吧，看懵逼了，没关系，请耐心接着往下看。

13.8 空间配置器函数allocate源码解读

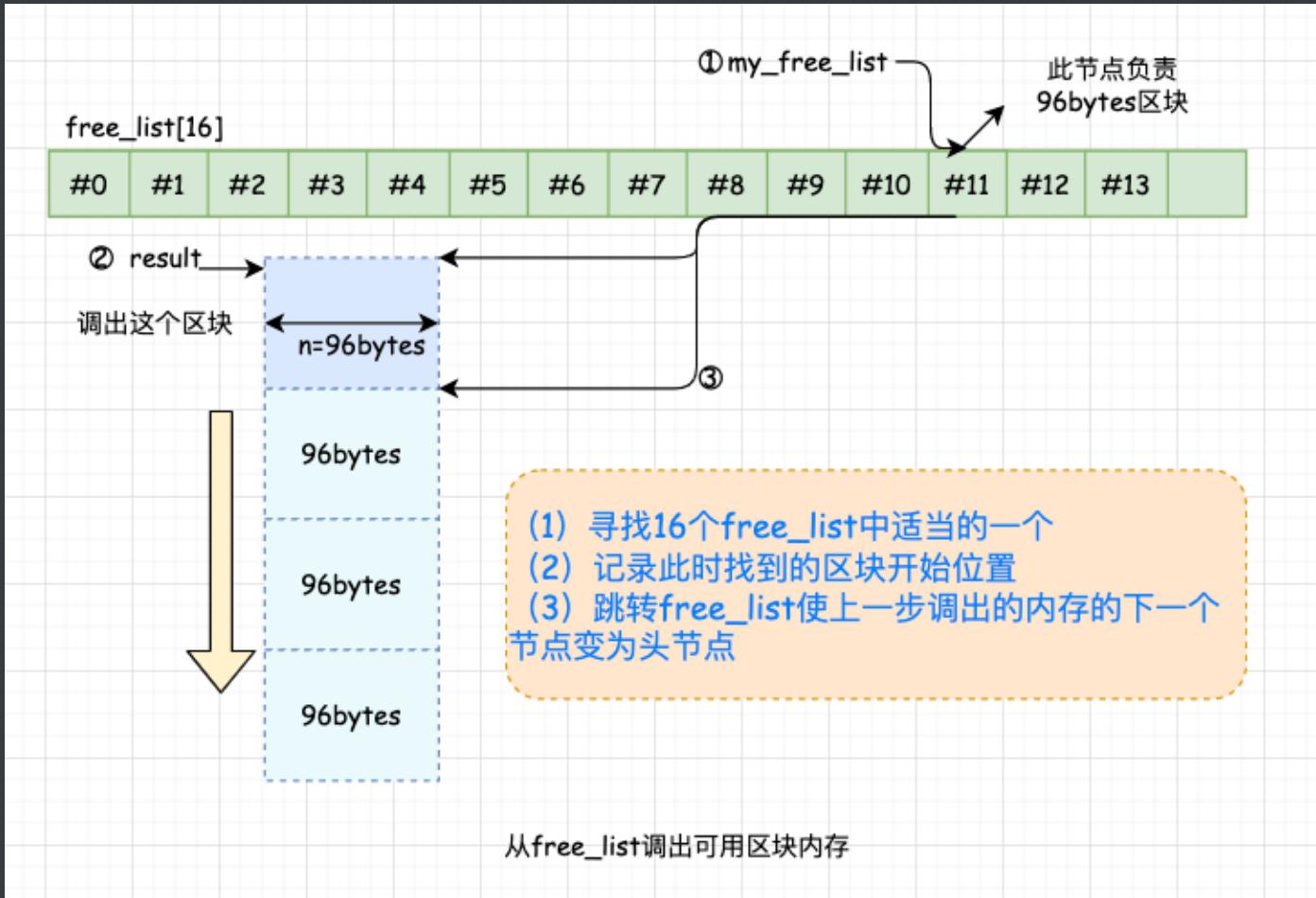
我们知道第二级配置器拥有配置器的标准接口函数 `allocate()`。此函数首先判断区块的大小，如果大于 128bytes -> 调用第一级配置器；小于128bytes-> 就检查对应的 `free_list`（如果没有可用区块，就将区块上调至 8 倍数的边界，然后调用 `refill()`, 为 `free list` 重新填充空间。

8.1 空间申请

调用标准接口函数 `allocate()`:



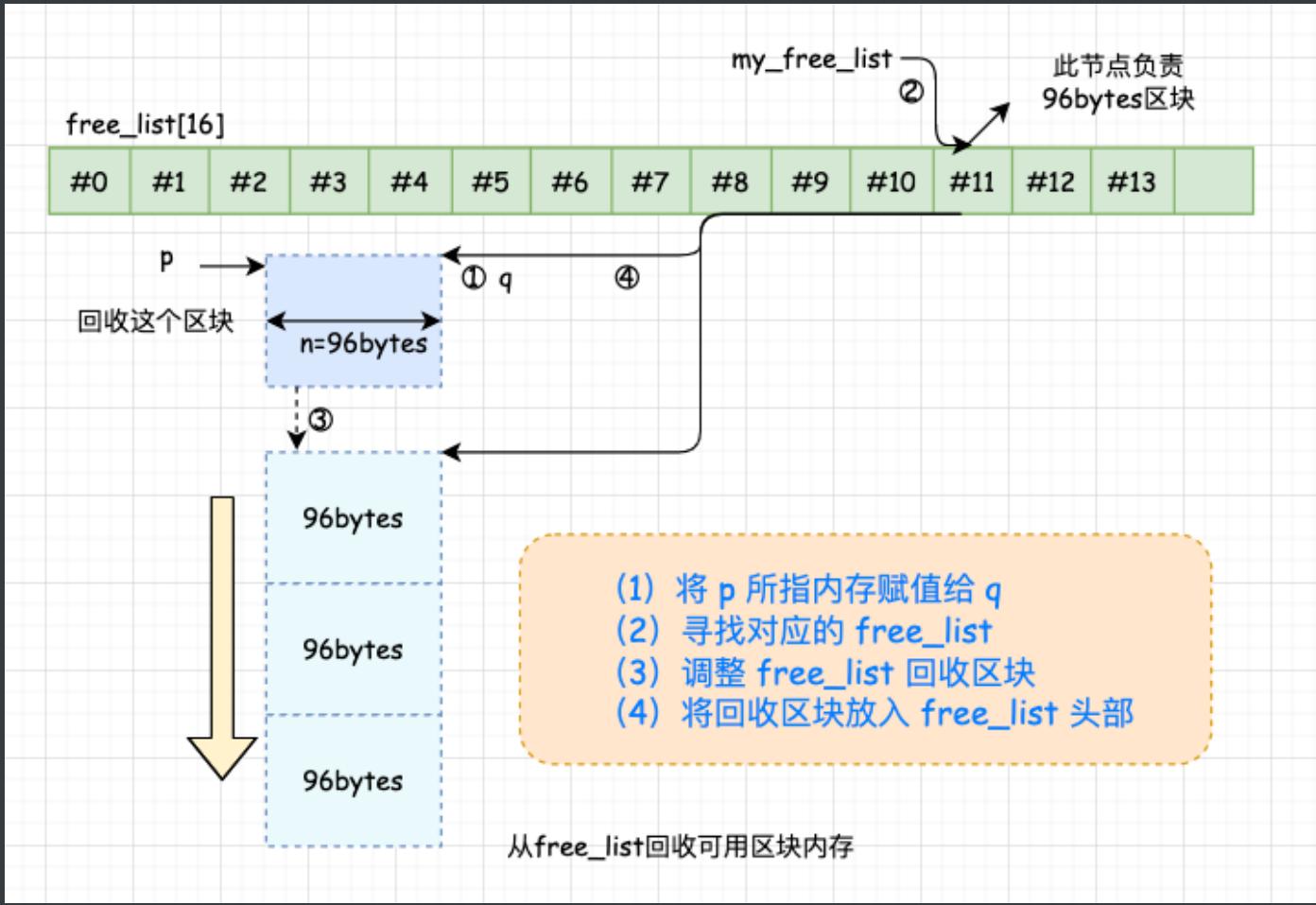
```
//n must be > 0
static void *allocate(size_t n)
{
    obj *volatile *my_free_list;
    obj *result;
    //大于128就调用第一级配置器
    if (n >(size_t)_MAX_BYTES) {
        return (malloc_alloc::allocate(n));
    }
    //寻找16个free lists中适当的一个
    my_free_list = free_list + FREELIST_INDEX(n);
    result = *my_free_list;
    if (result ==0 ) {
        //没有找到可用的free list, 准备重新填充free list
        void *r = refill(ROUND_UP(n)); //下节详述
        return r;
    }
    //调整free list
    *my_free_list = result ->free_list_link;
    return (result);
};
```



NOTE: 每次都是从对应的 free_list 的头部取出可用的内存块。然后对 free_list 进行调整，使上一步拨出的内存的下一个节点变为头结点。

8.2 空间释放

同样，作为第二级配置器拥有配置器标准接口函数 deallocate()。该函数首先判断区块大小，大于 128bytes 就调用第一级配置器。小于 128 bytes 就找出对应的 free_list，将区块回收。



NOTE: 通过调整 free_list 链表将区块放入 free_list 的头部。

区块回收纳入 free_list 的操作，如图所示：

8.3 重新填充 free_lists

- (1) 当发现 free_list 中没有可用区块时，就会调用 refill() 为 free_list 重新填充空间；
- (2) 新的空间将取自内存池（经由 chunk_alloc() 完成）；
- (3) 缺省取得 20 个新节点（区块），但万一内存池空间不足，获得的节点数可能小于 20。



```
//返回一个大小为n的对象，并且有时候会为适当的free list增加节点
//假设n已经适当上调至8的倍数
template <bool threads, int inst>
void* __default_alloc_template<threads, inst>::refill(size_t n)
{
    int nobjs = 20;
    //调用chunk_alloc(),尝试取得nobjs个区块作为free list的新节点
    //注意参数nobjs是pass by reference
    char * chunk = chunk_alloc(n, nobjs);    //稍后详述
    obj * volatile *my_free_list;
    obj *result;
    obj *current_obj, *next_obj;
    int i;

    //如果只获得一个区块，这个区块就分配给调用者用，free list无新节点
    if (1 == nobjs) return(chunk);
    //否则准备调用free list, 纳入新节点
    my_free_list = free_list + FREELIST_INDEX(n);

    //以下在chunk空间内建立free list
    result = (obj*)chunk;    //这一块准备返回客户端
    //以下导引free list指向新配置的空间(取自内存池)
    *my_free_list = next_obj = (obj*)(chunk + n);
    //以下将free list的各节点串接起来
    for (i = 1; ; i++) { //从1开始，因为第0个将返回给客户端
        current_obj = next_obj;
        next_obj = (obj*) ((char*)next_obj + n);
        if (nobjs - 1 == i) {
            current_obj->free_list_link = 0;
            break;
        } else {
            current_obj->free_list_link = next_obj;
        }
    }
    return(result);
}
```

8.4 内存池 (memory pool)

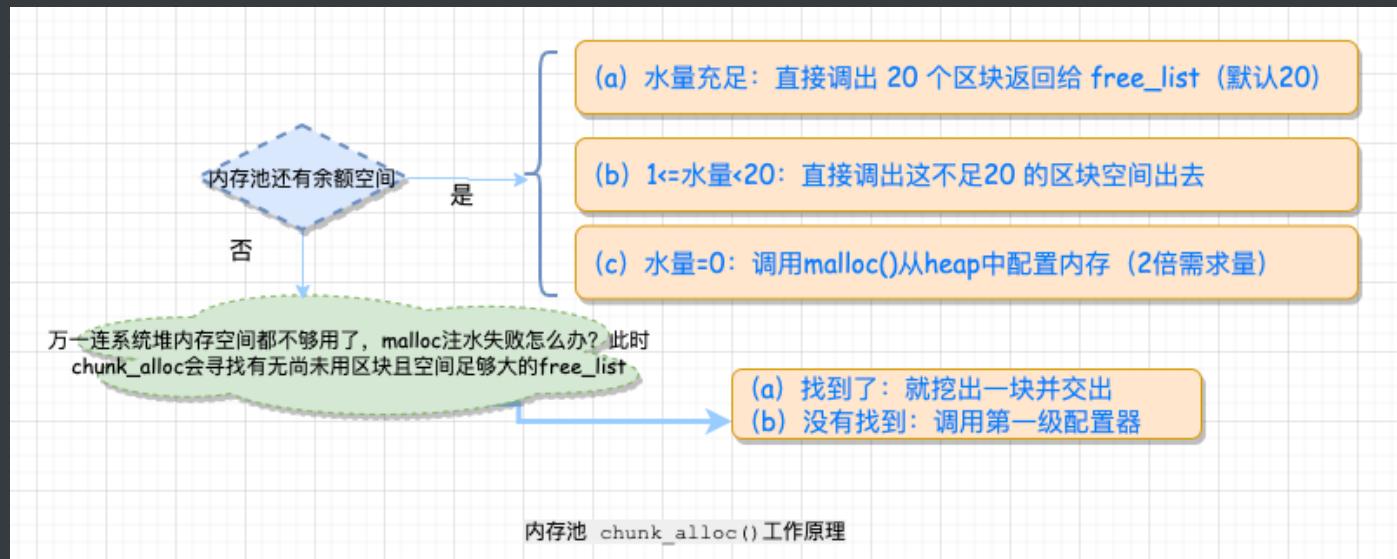
唔...在前面提到了 memory pool, 现在终于轮到这个大 boss 上场。

首先，我们要知道从内存池中取空间给 free_list 使用，是 chunk_alloc() 在工作，它是怎么工作的呢？

我们先来分析 chunk_alloc() 的工作机制：

chunk_alloc() 函数以 end_free – start_free 来判断内存池的“水量”（哈哈，很形象的比喻）。

具体逻辑都在下面的图里了，很形象吧。



如果第一级配置器的 malloc() 也失败了，就发出 bad_alloc 异常。

说了这么多来看一下 STL 的源码吧。



```
//假设size已经适当上调至8的倍数
//注意参数nobjs是pass by reference
template <bool threads, int inst>
char*
__default_alloc_template<threads, inst>::
chunk_alloc(size_t size, int& nobjs)
{
    char *result;
    size_t total_bytes = size * nobjs;
    size_t bytes_left = end_free - start_free; //内存池剩余空间

    if (bytes_left >= total_bytes) {
        //内存池剩余空间完全满足需求量
        result = start_free;
        start_free += total_bytes;
        return (result);
    }
    else if (bytes_left >= size) {
        //内存池剩余空间不能完全满足需求量但足够供应一个（含）以上的区块
        nobjs = bytes_left / size;
        total_bytes = size * nobjs;
        result = start_free;
        start_free += total_bytes;
        return (result);
    }
    else {
        //内存池剩余空间连一个区块的大小都无法提供
        size_t bytes_to_get = 2 * total_bytes + ROUND_UP(heap_size >> 4);
        //以下试着让内存池中的残余零头还有利用价值
        if (bytes_left > 0) {
            //内存池还有一些零头，先配给适当的free list
            //首先寻找适当的free list
            obj *volatile *my_free_list = free_list + FREELIST_INDEX(bytes_left);
            //调整free list，将内存池中的残余空间编入
            ((obj*)start_free)-> free_list_link = *my_free_list;
            *my_free_list = (obj*)start_free;
        }

        //配置heap空间，用来补充内存池
        start_free = (char*)malloc(bytes_to_get);
        if (0 == start_free) {
            //heap 空间不足，malloc( )失败
            int i;
            obj* volatile * my_free_list, *p;
            //试着检视我们手上拥有的东西。这不会造成伤害，我们不打算尝试配置
            //较小的区块，因为那再多进程机器上容易导致灾难
            //以下搜寻适当的free list
            //所谓适当是指“尚有未用区块，且区块够大”的free list
            for (i = size; i <= __MAX_BYTES; i += __ALIGN) {
                my_free_list = free_list + FREELIST_INDEX(i);
                p = *my_free_list;
                if (0 != p) { //free list内尚有未用区块
                    //调整free list以释出未用区块
                    *my_free_list = p->free_list_link;
                }
            }
        }
    }
}
```

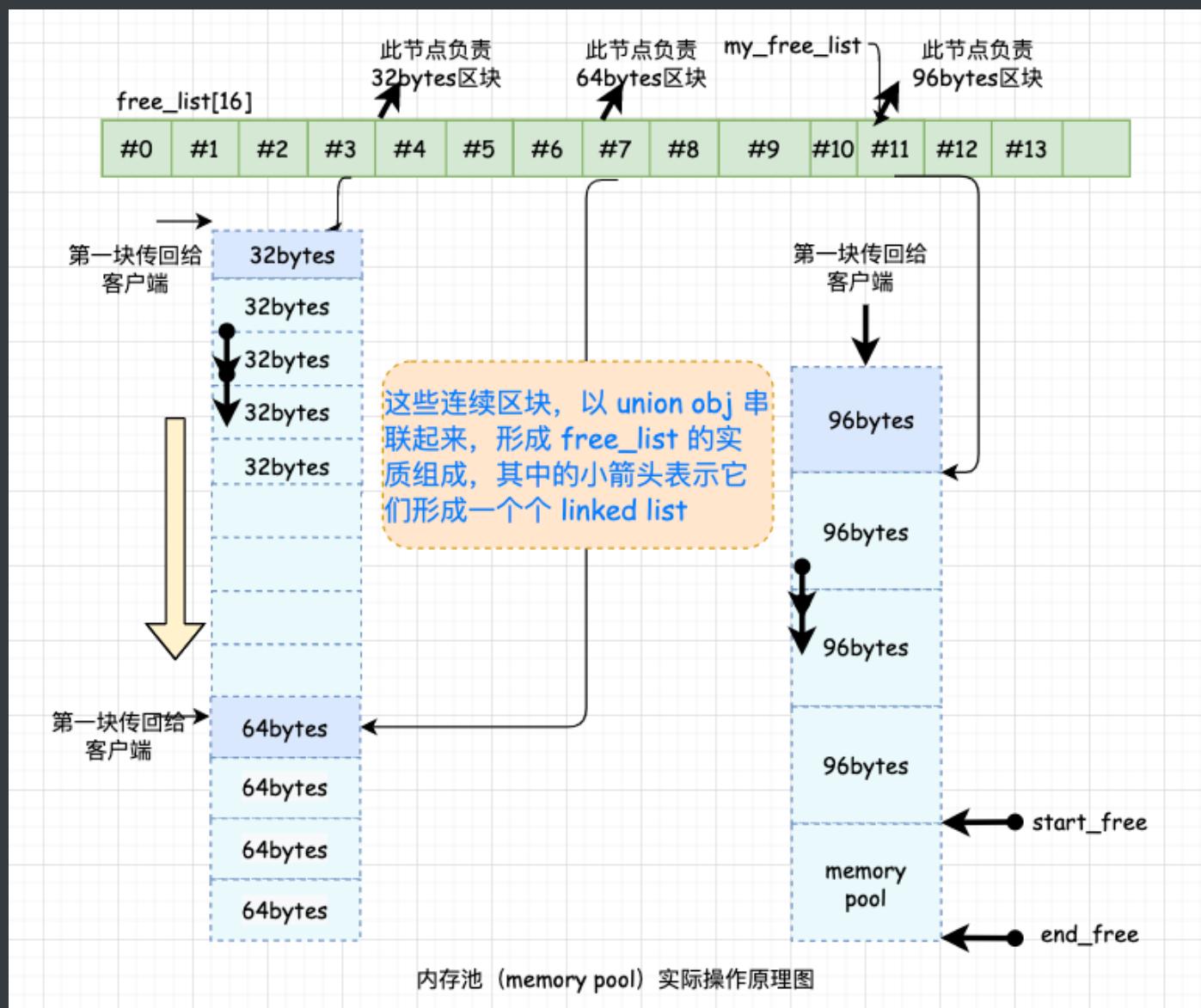
```

        start_tree = (char*)p;
        end_free = start_free + i;
        //递归调用自己, 为了修正nobjs
        return (chunk_alloc(size, nobjs));
        //注意, 任何残余零头终将被编入释放的free list中备用
    }
}

end_free = 0; //如果出现意外 (山穷水尽, 到处都没内存可用了)
//调用第一级配置器, 看看out-of-memory机制能够尽点力
start_free = (char*)malloc_alloc::allocate(bytes_to_get);
//这会导致抛出异常, 或内存不足的情况获得改善
}
heap_size += bytes_to_get;
end_free = start_free + bytes_to_get;
//递归调用自己, 为了修正nobjs
return (chunk_alloc(size, nobjs));
}
}

```

太长了，又看懵逼了吧，没关系，请看下图。



NOTE：上述就是 STL 源码当中实际内存池的操作原理，我们可以看到其实以共用体串联起来共享内存形成了 free_list 的实质组成。

13.9 本文小结

STL 源码本身博大精深，还有很多精妙的设计等着大家去探索。

小贺本人才疏学浅，在这里也只是在自己掌握的程度下写出自己的理解，不足之处希望大家多多指出，互相讨论学习。肝了一个礼拜的文章，文中所有的图都是自己一个个亲手画的，不画不知道，画完之后真心感觉不容易啊。

参考文章：

《STL源码剖析-侯捷》

<https://cloud.tencent.com/developer/article/1686585>

<https://dulishu.top/allocator-of-stl/>

十四、万字长文手撕 STL 迭代器源码与 traits 编程技法

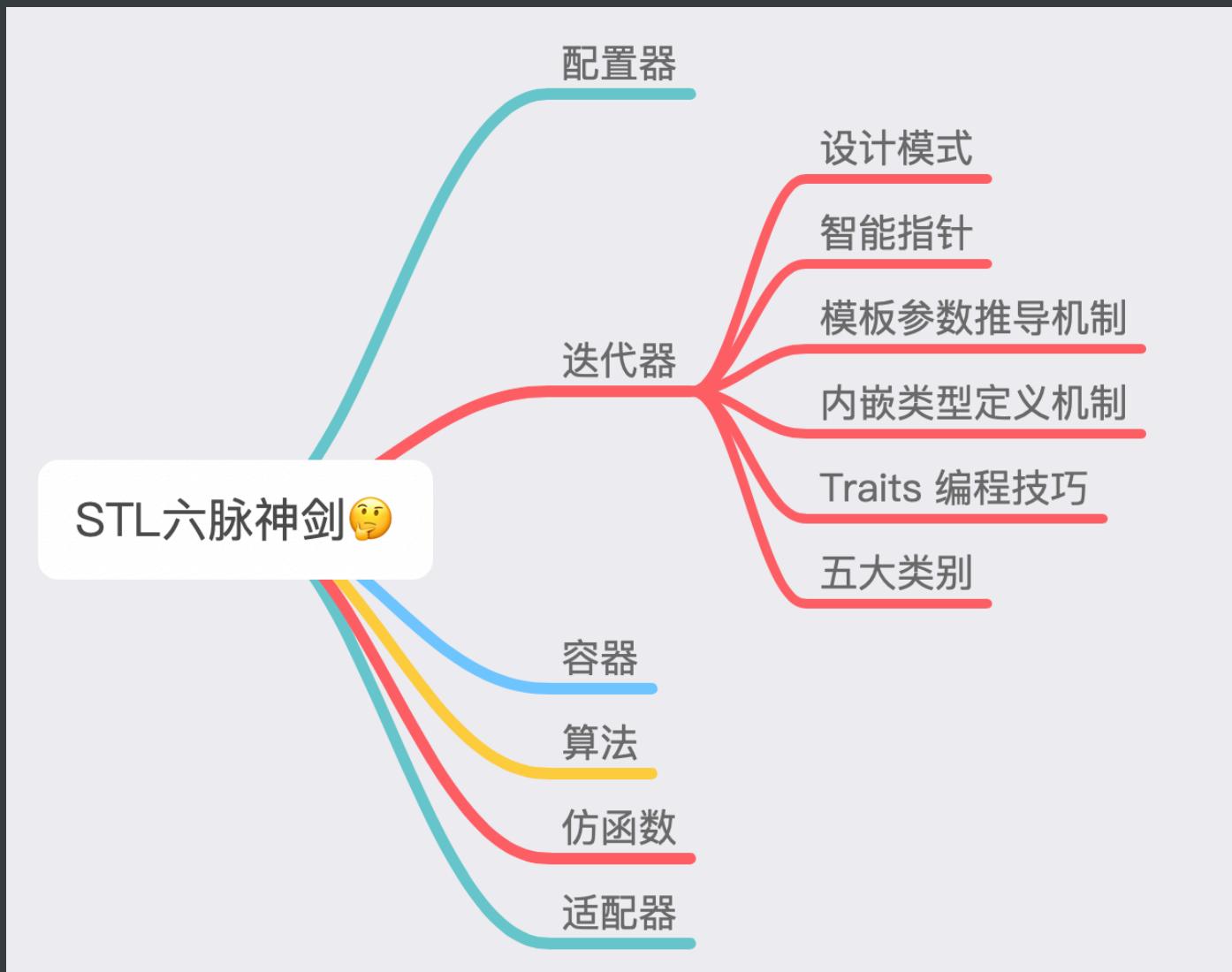
大家好，我是小贺。

文章每周持续更新，可以微信搜索公众号「herongwei」第一时间阅读和催更。

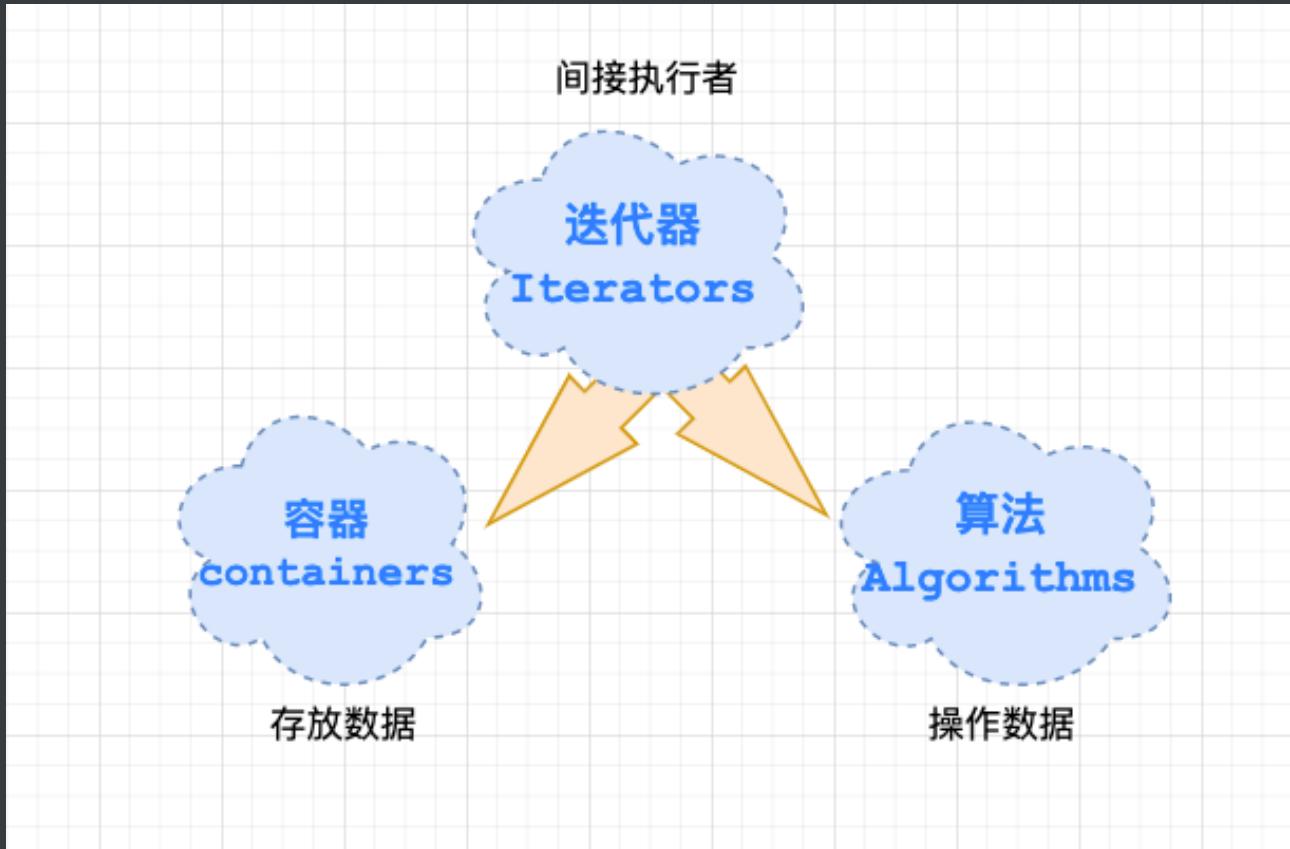
本文 GitHub：<https://github.com/rongweihe/CPPNotes> 已经收录，有一线大厂面试点思维导图，也整理了很多我的文档，欢迎点个小★和完善。一起加油，变得更好！

14.1 前言

上一篇，我们剖析了 STL 空间配置器，这一篇文章，我们来学习下 STL 迭代器以及背后的 traits 编程技法。



在 STL 编程中，容器和算法是独立设计的，容器里面存的是数据，而算法则是提供了对数据的操作，在算法操作数据的过程中，要用到迭代器，迭代器可以看做是容器和算法中间的桥梁。



14.2 迭代器设计模式

为何说迭代器的时候，还谈到了设计模式？这个迭代器和设计模式又有什么关系呢？

其实，在《设计模式：可复用面向对象软件的基础》（GOF）这本经典书中，谈到了 23 种设计模式，其中就有 iterator 迭代模式，且篇幅颇大。

碰巧，笔者在研究 STL 源码的时候，同样的发现有 iterator 迭代器，而且还占据了一章的篇幅。

在设计模式中，关于 iterator 的描述如下：一种能够顺序访问容器中每个元素的方法，使用该方法不能暴露容器内部的表达方式。而类型萃取技术就是为了要解决和 iterator 有关的问题的。

有了上面这个基础，我们就知道了迭代器本身也是一种设计模式，其设计思想值得我们仔细体会。

那么 C++ STL 实现 iterator 和 GOF 介绍的迭代器实现方法什么区别呢？那首先我们需要了解 C++ 中的两个编程范式的概念，OOP（面向对象编程）和 GP（泛型编程）。

在 C++ 语言里面，我们可用以下方式来简单区分一下 OOP 和 GP：



OOP：将 `methods` 和 `datas` 关联到一起（通俗点就是方法和成员变量放到一个类中实现），通过继承的方式，利用虚函数表（`virtual`）来实现运行时类型的判定，也叫“动态多态”，由于运行过程中需根据类型去检索虚函数表，因此效率相对较低。

GP：泛型编程，也被称为“静态多态”，多种数据类型在同一种算法或者结构上皆可操作，其效率与针对某特定数据类型而设计的算法或者结构相同，具体数据类型在编译期确定，编译器承担更多，代码执行效率高。在 STL 中利用 GP 将 `methods` 和 `datas` 实现了分而治之。

而 C++ STL 库的整个实现采用的就是 GP (Generic Programming)，而不是 OOP (Object Oriented Programming)。而 GOF 设计模式采用的就是继承关系实现的，因此，相对来讲，C++ STL 的实现效率会相对较高，而且也更有利于维护。

在 STL 编程结构里面，迭代器其实也是一种模板 `class`，迭代器在 STL 中得到了广泛的应用，通过迭代器，容器和算法可以有机的绑定在一起，只要对算法给予不同的迭代器，比如 `vector::iterator`、`list::iterator`，`std::find()` 就能对不同的容器进行查找，而无需针对某个容器来设计多个版本。

这样看来，迭代器似乎依附在容器之下，那么，有没有独立而适用于所有容器的泛化的迭代器呢？这个问题先留着，在后面我们会看到，在 STL 编程结构里面，它是如何把迭代器运用的炉火纯青。

14.3 智能指针

STL 是泛型编程思想的产物，是以泛型编程为指导而产生的。具体来说，STL 中的迭代器将范型算法 (`find`, `count`, `find_if`) 等应用于某个容器中，给算法提供一个访问容器元素的工具，`iterator` 就扮演着这个重要的角色。

稍微看过 STL 迭代器源码的，就明白迭代器其实也是一种智能指针，因此，它也就拥有了一般指针的所有特点——能够对其进行 `*` 和 `->` 操作。

```
template<typename T>
class ListIterator { // myList 迭代器
public:
    ListIterator(T *p = 0) : m_ptr(p){} // 构造函数
    T& operator*() const { return *m_ptr;} // 取值，即 dereference
    T* operator->() const { return m_ptr;} // 成员访问，即 member access
    // ...
};
```

但是在遍历容器的时候，不可避免的要对遍历的容器内部有所了解，所以，干脆把迭代器的开发工作交给容器的设计者，如此以来，所有实现细节反而得以封装起来不被使用者看到，这也正是为什么每一种 STL 容器都提供有专属迭代器的缘故。

比如笔者自己实现的 `list` 迭代器在这里使用的好处主要有：

- (1) 不用担心内存泄漏（类似智能指针，析构函数释放内存）；
- (2) 对于 `list`，取下一个元素不是通过自增而是通过 `next` 指针来取，使用智能指针可以对自增进行重载，从而提供统一接口。

14.4 template 参数推导

参数推导能帮我们解决什么问题呢？

在算法中，你可能会定义一个简单的中间变量或者设定算法的返回变量类型，这时候，你可能会遇到这样的问题，假如你需要知道迭代器所指元素的类型是什么，进而获取这个迭代器操作的算法的返回类型，但是问题是 C++ 没有 `typeof` 这类判断类型的函数，也无法直接获取，那该如何是好？

注意是类型，不是迭代器的值，虽然 C++ 提供了一个 `typeid()` 操作符，这个操作符只能获得型别的名称，但不能用来声明变量。要想获得迭代器型别，这个时候又该如何是好呢？

`function template` 的参数推导机制是一个不错的方法。

例如：

如果 `I` 是某个指向特定对象的指针，那么在 `func` 中需要指针所指向对象的型别的时候，怎么办呢？这个还比较容易，模板的参数推导机制可以完成任务，

```
template <class I>
inline void func(I iter) {
    func_imp(iter, *iter); // 传入 iter 和 iter 所指的值，class 自动推导
}
```

通过模板的推导机制，就能轻而易举的获得指针所指向的对象的类型。

```
template <class I, class T>
void func_imp(I iter, T t) {
    T tmp; // 这里就是迭代器所指物的类别
    // ... 功能实现
}

int main() {
    int i;
    func(&i); // 这里传入的是一个迭代器（原生指针也是一种迭代器）
}
```

使用函数
模板参数
推导

```
template<class I>
inline void func(I iter) {
    //func的工作全部在func_impl实现
    func_impl(iter, *iter);
}
```

```
template<class I, class T>
void func_impl(I iter, T t) {
    T tmp; //T就是迭代器所指之物的类型
    //这里做原本func()应该做的事情
}
```

上面的做法呢，通过多层的迭代，很巧妙地导出了 `T`，但是却很有局限性，比如，我希望 `func()` 返回迭代器的 `value type` 类型返回值，函数的 "template 参数推导机制" 推导的只是参数，无法推导函数的返回值类型。万一需要推导函数的返回值，好像就不行了，那么又该如何是好？

这就引出了下面的内嵌型别。

14.5 声声明内嵌型别

上述所说的 迭代器所指对象的型别，称之为迭代器的 `value type`。

尽管在 `func_impl` 中我们可以把 `T` 作为函数的返回值，但是问题是用户需要调用的是 `func`。

如果在参数推导机制上加上内嵌型别 (`typedef`) 呢？为指定的对象类型定义一个别名，然后直接获取，这样来看一下实现：

```
template<typename T>
class MyIter {
public:
```

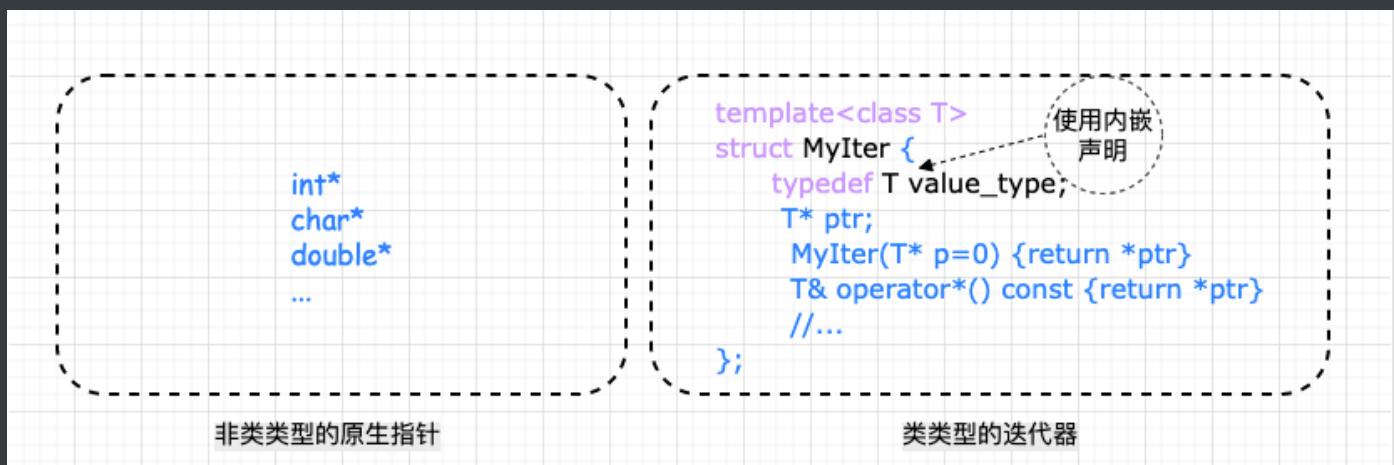
```

typedef T value_type; //内嵌类型声明
MyIter(T *p = 0) : m_ptr(p) {}
T& operator*() const { return *m_ptr;}
private:
T *m_ptr;
};

//以迭代器所指对象的类型作为返回类型
//注意typename是必须的，它告诉编译器这是一个类型
template<typename MyIter>
typename MyIter::value_type Func(MyIter iter) {
    return *iter;
}

int main(int argc, const char *argv[]) {
    MyIter<int> iter(new int(666));
    std::cout<<Func(iter)<<std::endl; //print=> 666
}

```



上面的解决方案看着可行，但其实呢，实际上还是有问题，这里有一个隐晦的陷阱：实际上并不是所有的迭代器都是 `class type`，原生指针也是一种迭代器，由于原生指针不是 `class type`，所以没法为它定义内嵌型别。

算法函数体重对迭代器类型的需求

```
template<class I>
inline void func(I iter) {
    //func的工作全部在func_impl实现...
    func_impl(iter, *iter); //使用函数模板参数推导
}

template<class I, class T>
void func_impl(I iter, T t) {
    T tmp; //T就是迭代器所指之物的类型
    //这里做原本func()应该做的事情
}
```

算法返回类型对迭代器类型的需求

```
template<class I>
typename I::value_type func(I iter) {
    return *iter; //func的返回类型
}

返回类型必须加上关键字 typename 因为
T 是一个 template 参数, 在它被编译器具体实现之前
编译器对T一无所知, 因此不知道 MyIter<T>::value
代表的是一个类型或者一个 member function 或是一
个 data member 所以需要关键字 typename 告诉编译
器这是一个类型
```

因为 `func` 如果是一个泛型算法，那么它也绝对要接受一个原生指针作为迭代器，下面的代码编译没法通过：

```
int *p = new int(5);
cout<<Func(p)<<endl; // error
```

要解决这个问题，`Partial specialization`（模板偏特化）就出场了。

14.6 Partial specialization (模板偏特化)

所谓偏特化是指如果一个 `class template` 拥有一个以上的 `template` 参数，我们可以针对其中某个（或多个，但不是全部）`template` 参数进行特化，比如下面这个例子：

```
template <typename T>
class C {...}; //此泛化版本的 T 可以是任何类型
template <typename T*>
class C<T*> {...}; //特化版本, 仅仅适用于 T 为“原生指针”的情况, 是泛化版本的限制版
```

所谓特化，**就是特殊情况特殊处理**，第一个类为泛化版本，`T` 可以是任意类型，第二个类为特化版本，是第一个类的特殊情况，只针对原生指针。

14.6.1、原生指针怎么办？——特性“萃取” traits

还记得前面说过的**参数推导机制+内嵌型别机制获取型别**有什么问题吗？问题就在于原生指针虽然是迭代器但不是 class，无法定义内嵌型别，而偏特化似乎可以解决这个问题。

有了上面的认识，我们再看看 STL 是如何应用的。STL 定义了下面的类模板，它专门用来“萃取”迭代器的特性，而 value_type 正是迭代器的特性之一：

traits 在 bits/stl_iterator_base_types.h 这个文件中：

```
template<class _Tp>
struct iterator_traits<_Tp*> {
    typedef ptrdiff_t difference_type;
    typedef typename _Tp::value_type value_type;
    typedef typename _Tp::pointer pointer;
    typedef typename _Tp::reference reference;
    typedef typename _Tp::iterator_category iterator_category;
};
```

```
template<typename Iterator>
struct iterator_traits { //类型萃取机
    typedef typename Iterator::value_type value_type; //value_type 就是
Iterator 的类型型别
}
```

加入萃取机前后的变化：

```
template<typename Iterator> //萃取前
typename Iterator::value_type func(Iterator iter) {
    return *iter;
}

//通过 iterator_traits 作用后的版本
template<typename Iterator> //萃取后
typename iterator_traits<Iterator>::value_type func(Iterator iter) {
    return *iter;
}
```

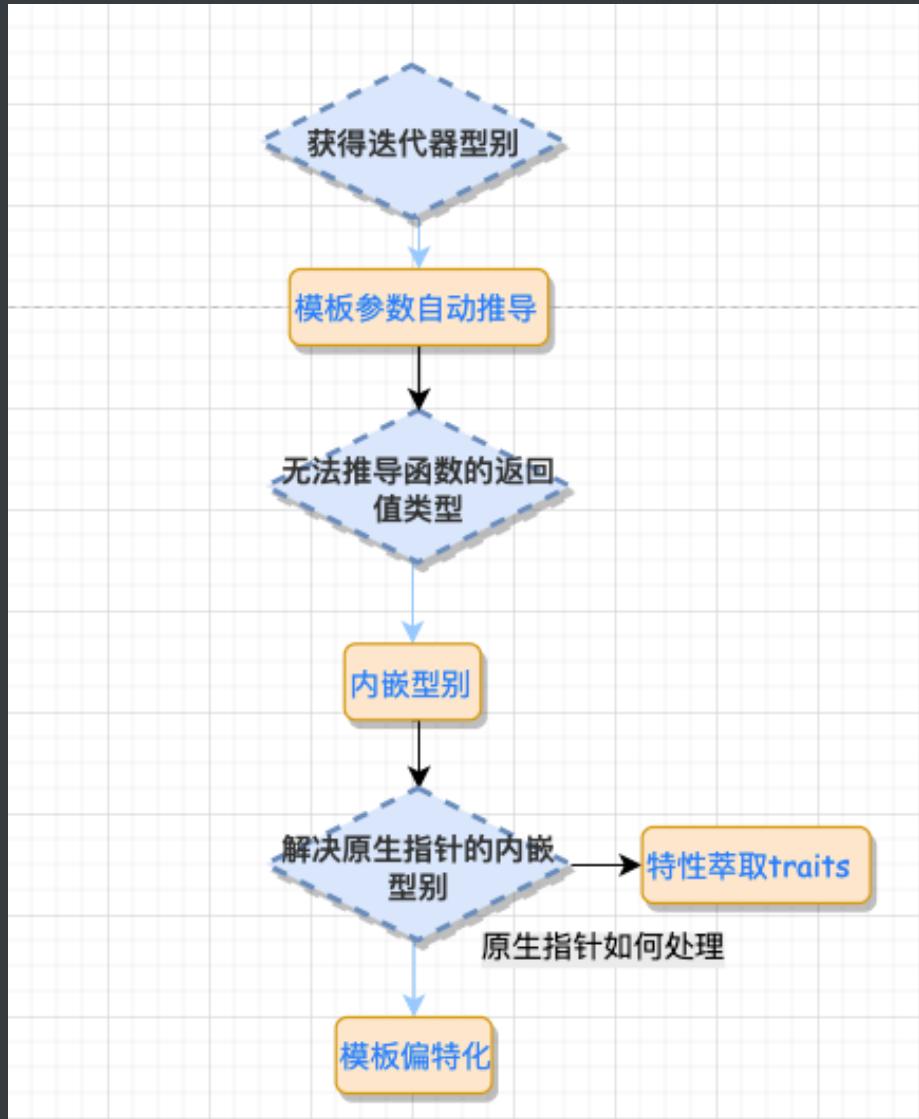
看到这里也许你会问了，这个萃取前和萃取后的 `typename`：

`iterator_traits::value_type` 跟 `Iterator::value_type` 看起来一样啊，为什么还要增加 `iterator_traits` 这一层封装，岂不是多此一举？

回想萃取之前的版本有什么缺陷：不支持原生指针。而通过萃取机的封装，**我们可以通过类模板的特化来支持原生指针的版本！**如此一来，无论是智能指针，还是原生指针，`iterator_traits::value_type` 都能起作用，这就解决了前面的问题。

```
//iterator_traits的偏特化版本，针对迭代器是原生指针的情况
template<typename T>
struct iterator_traits<T*> {
    typedef T value_type;
};
```

看到这里，我们不得不佩服的 STL 的设计者们，真·秒啊！我们用下面这张图来总结一下前面的流程：



14.6.2、const 偏特化

通过偏特化添加一层中间转换的 traits 模板 class，能实现对原生指针和迭代器的支持，有的读者可能会继续追问：对于指向常数对象的指针又该怎么处理呢？比如下面的例子：

```
iterator_traits<const int*>::value_type // 获得的 value_type 是 const int, 而不是 int
```

const 变量只能初始化，而不能赋值（这两个概念必须区分清楚）。这将带来下面的问题：

```

template<typename Iterator>
typename iterator_traits<Iterator>::value_type func(Iterator iter) {
    typename iterator_traits<Iterator>::value_type tmp;
    tmp = *iter; // 编译 error
}

int val = 666 ;
const int *p = &val;
func(p); // 这时函数里对 tmp 的赋值都将是不允许的

```

那该如何是好呢？答案还是**偏特化**，来看实现：

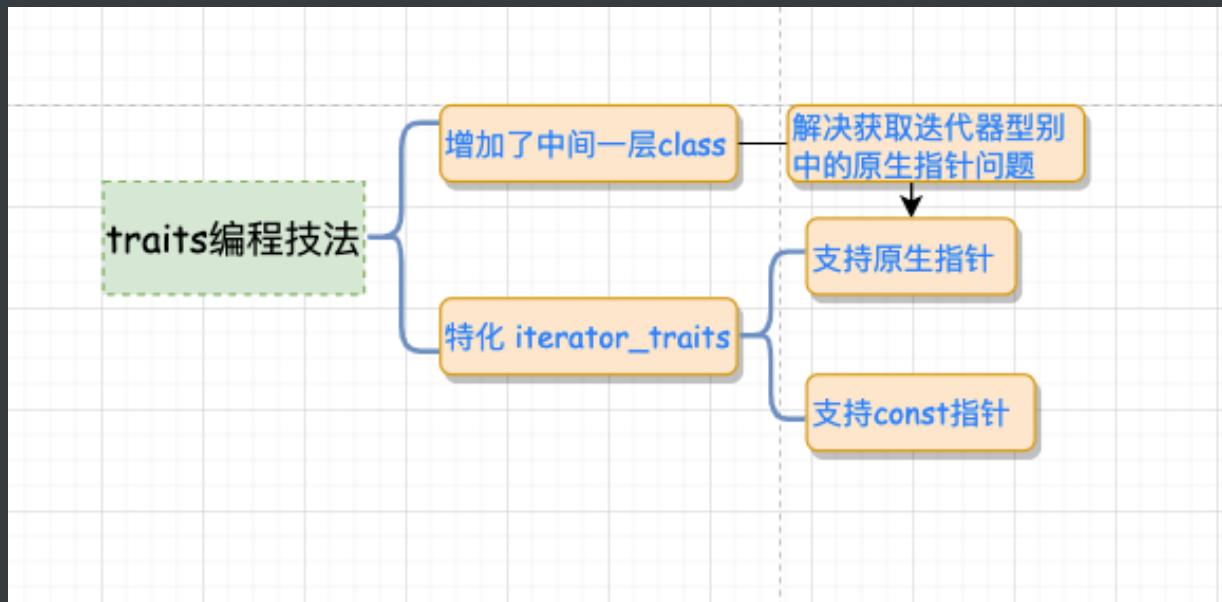
```

template<typename T>
struct iterator_traits<const T*> { //特化const指针
    typedef T value_type; //得到T而不是const T
}

```

14.7 traits编程技法总结

通过上面几节的介绍，我们知道，所谓的 traits 编程技法无非 就是增加一层中间的模板 **class**，以解决获取迭代器的型别中的原生指针问题。利用一个中间层 **iterator_traits** 固定了 **func** 的形式，使得重复的代码大量减少，唯一要做的就是稍稍特化一下 **iterator_traits** 使其支持 **pointer** 和 **const pointer**。



```
#include <iostream>

template <class T>
struct MyIter {
    typedef T value_type; // 内嵌型别声明
    T* ptr;
    MyIter(T* p = 0) : ptr(p) {}
    T& operator*() const { return *ptr; }
};

// class type
template <class T>
struct my_iterator_traits {
    typedef typename T::value_type value_type;
};

// 偏特化 1
template <class T>
struct my_iterator_traits<T*> {
    typedef T value_type;
};

// 偏特化 2
template <class T>
struct my_iterator_traits<const T*> {
    typedef T value_type;
};

// 首先询问 iterator_traits<I>::value_type, 如果传递的 I 为指针, 则进入特化版
// 本, iterator_traits 直接回答; 如果传递进来的 I 为 class type, 就去询问
// T::value_type.
template <class I>
typename my_iterator_traits<I>::value_type Func(I ite) {
    std::cout << "normal version" << std::endl;
    return *ite;
}

int main(int argc, const char *argv[]) {
    MyIter<int> ite(new int(6));
    std::cout << Func(ite)<<std::endl;//print=> 6
```

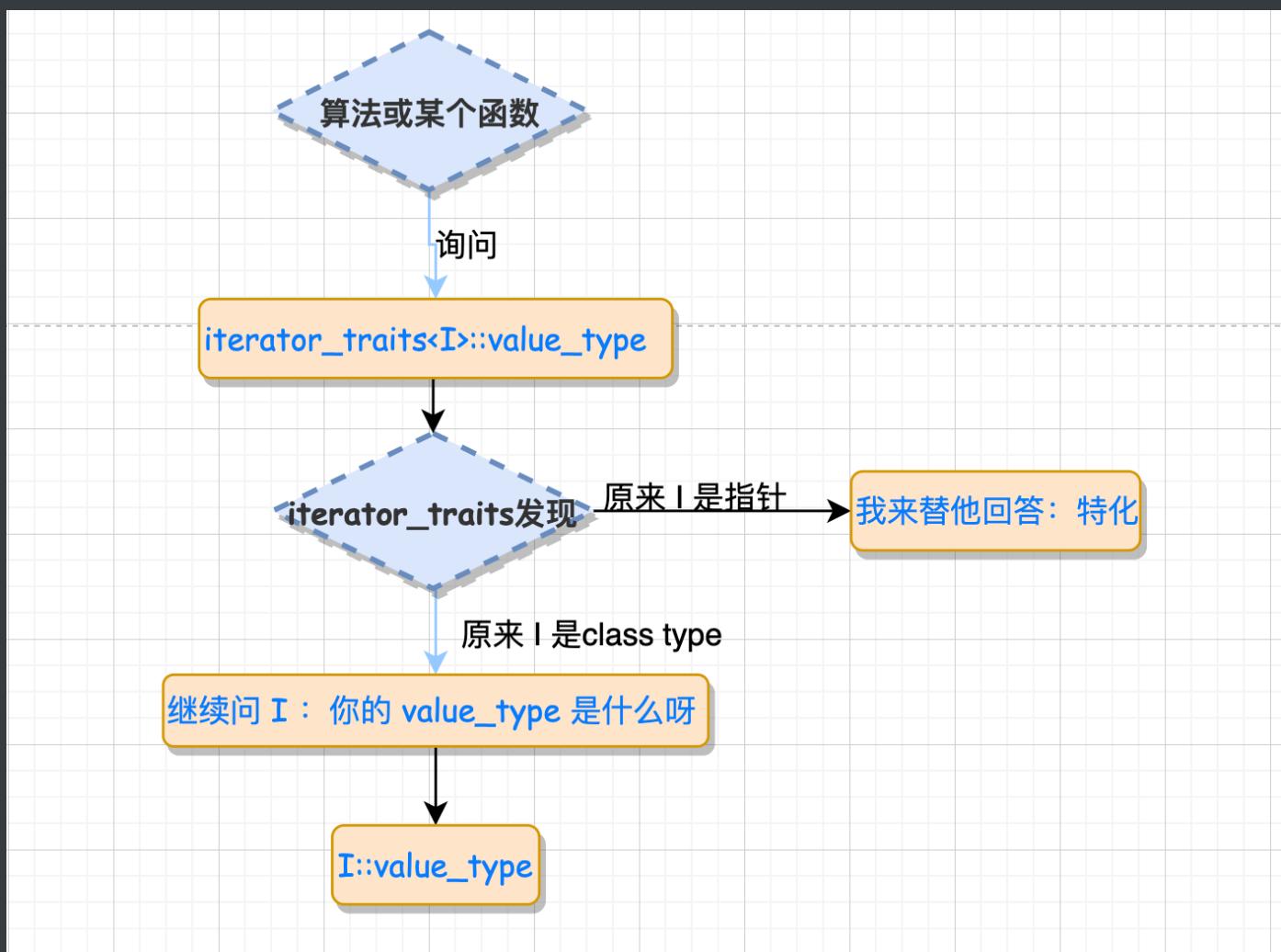
```

int *p = new int(7);
std::cout<<Func(p)<<std::endl;//print=> 7
const int k = 8;
std::cout<<Func(&k)<<std::endl;//print=> 8
}

```

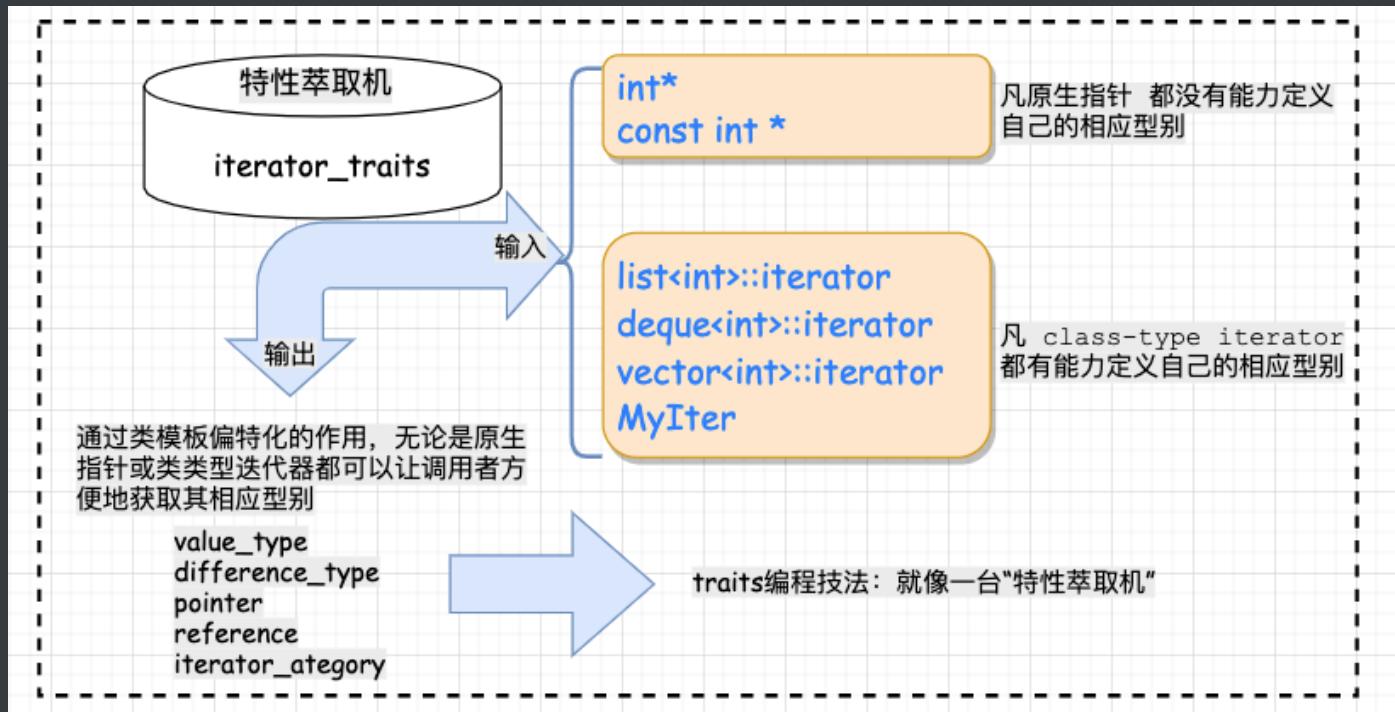
上述的过程是首先询问 `iterator_traits::value_type`，如果传递的 I 为指针，则进入特化版本，`iterator_traits` 直接回答 T；如果传递进来的 I 为 class type，就去询问 `T::value_type`。

通俗的解释可以参照下图：



总结：核心知识点在于 **模板参数推导机制+内嵌类型定义机制**，为了能处理原生指针这种特殊的迭代器，引入了**偏特化机制**。`traits` 就像一台“特性萃取机”，把迭代器放进去，就能榨取出迭代器的特性。

这种偏特化是针对可调用函数 func 的偏特化，想象一种极端情况，假如 func 有几百万行代码，那么如果不这样做的话，就会造成非常大的代码污染。同时增加了代码冗余。



14.8 迭代器的型别和种类

14.8.1 迭代器的型别

我们再来看看迭代器的型别，常见迭代器相应型别有 5 种：

- `value_type`：迭代器所指对象的类型，原生指针也是一种迭代器，对于原生指针 `int*`，`int` 即为指针所指对象的类型，也就是所谓的 `value_type`。
- `difference_type`：用来表示两个迭代器之间的距离，对于原生指针，STL 以 C++ 内建的 `ptrdiff_t` 作为原生指针的 `difference_type`。
- `reference_type`：是指迭代器所指对象的类型的引用，`reference_type` 一般用在迭代器的 * 运算符重载上，如果 `value_type` 是 `T`，那么对应的 `reference_type` 就是 `T&`；如果 `value_type` 是 `const T`，那么对应的 `reference_type` 就是 `const T&`。
- `pointer_type`：就是相应的指针类型，对于指针来说，最常用的功能就是 `operator*` 和 `operator->` 两个运算符。

- `iterator_category`： 的作用是标识迭代器的移动特性和可以对迭代器执行的操作，从 `iterator_category` 上，可将迭代器分为 Input Iterator、Output Iterator、Forward Iterator、Bidirectional Iterator、Random Access Iterator 五类，这样分可以尽可能地提高效率。

```
template<typename Category,
         typename T,
         typename Distance = ptrdiff_t,
         typename Pointer = T*,
         typename Reference = T&>
struct iterator //迭代器的定义
{
    typedef Category iterator_category;
    typedef T value_type;
    typedef Distance difference_type;
    typedef Pointer pointer;
    typedef Reference reference;
};
```

`iterator class` 不包含任何成员变量，只有类型的定义，因此不会增加额外的负担。由于后面三个类型都有默认值，在继承它的时候，只需要提供前两个参数就可以了。这个类主要是用来继承的，在实现具体的迭代器时，可以继承上面的类，这样子就不会漏掉上面的 5 个型别了。

对应的迭代器萃取机设计如下：

```
template<typename I>
struct iterator_traits { //特性萃取机，萃取迭代器特性
    typedef typename I::iterator_category iterator_category;
    typedef typename I::value_type value_type;
    typedef typename I::difference_type difference_type;
    typedef typename I::pointer pointer;
    typedef typename I::reference reference;
};

//需要对型别为指针和 const 指针设计特化版本看
```

14.8.2、迭代器的分类

最后，我们来看看，迭代器型别 `iterator_category` 对应的迭代器类别，这个类别会限制迭代器的操作和移动特性。

除了原生指针以外，迭代器被分为五类：

- `Input Iterator`：此迭代器不允许修改所指的对象，是只读的。支持 `==`、`!=`、`++`、`*`、`->` 等操作。
- `Output Iterator`：允许算法在这种迭代器所形成的区间上进行只写操作。支持 `++`、`*` 等操作。
- `Forward Iterator`：允许算法在这种迭代器所形成的区间上进行读写操作，但只能单向移动，每次只能移动一步。支持 `Input Iterator` 和 `Output Iterator` 的所有操作。
- `Bidirectional Iterator`：允许算法在这种迭代器所形成的区间上进行读写操作，可双向移动，每次只能移动一步。支持 `Forward Iterator` 的所有操作，并另外支持 `-` 操作。
- `Random Access Iterator`：包含指针的所有操作，可进行随机访问，随意移动指定的步数。支持前面四种 `Iterator` 的所有操作，并另外支持 `[n]` 操作符等操作。

category		characteristic	valid expressions			
all categories		Can be copied and copy-constructed	X b(a); b = a;			
		Can be incremented	++a a++ *a++			
Random Access 随机迭代器	Bidirectional 双向迭代器	Forward 前向	Input 输入	Accepts equality/inequality comparisons	a == b a != b	
				Can be dereferenced as an <i>rvalue</i>	*a a->m	
		Output 输出		Can be dereferenced to be the left side of an assignment operation	*a = t *a++ = t	
				Can be default-constructed	X a; X()	
				Can be decremented	--a a-- *a--	
				Supports arithmetic operators + and -	a + n n + a a - n a - b	
				Supports inequality comparisons (<, >, <= and >=) between iterators	a < b a > b a <= b a >= b	
				Supports compound assignment operations += and -=	a += n a -= n	
				Supports offset dereference operator ([])	a[n]	

Where X is an iterator type, a and b are objects of this iterator type, t is an object of the type pointed by the iterator type, and n is an integer value.

那么，这里，小贺想问大家，为什么我们要对迭代器进行分类呢？迭代器在具体的容器里是到底如何运用的呢？这个问题就放到下一节在讲。

最最后，我们再来看看一下六大组件的关系：

container (容器) 通过 **allocator** (配置器) 取得数据储存空间

algorithm (算法) 通过 **iterator** (迭代器) 存取 **container** (容器) 内容

functor (仿函数) 可以协助 **algorithm** (算法) 完成不同的策略变化

adapter (配接器) 可以修饰或套接 **functor** (仿函数) 。

参考文章：

- 《STL源码剖析-侯捷》
 - <https://zhuanlan.zhihu.com/p/85809752>
 - <https://wendeng.github.io/>
-

十五、2万字 20图带你手撕 STL 序列式容器源码

大家好，我是小贺。

文章每周持续更新，可以微信搜索公众号「herongwei」第一时间阅读和催更。

本文 GitHub : <https://github.com/rongweihe/CPPNotes> 已经收录，有一线大厂面试点思维导图，也整理了很多我的文档，欢迎点个小和完善。一起加油，变得更好！

15.1 前言

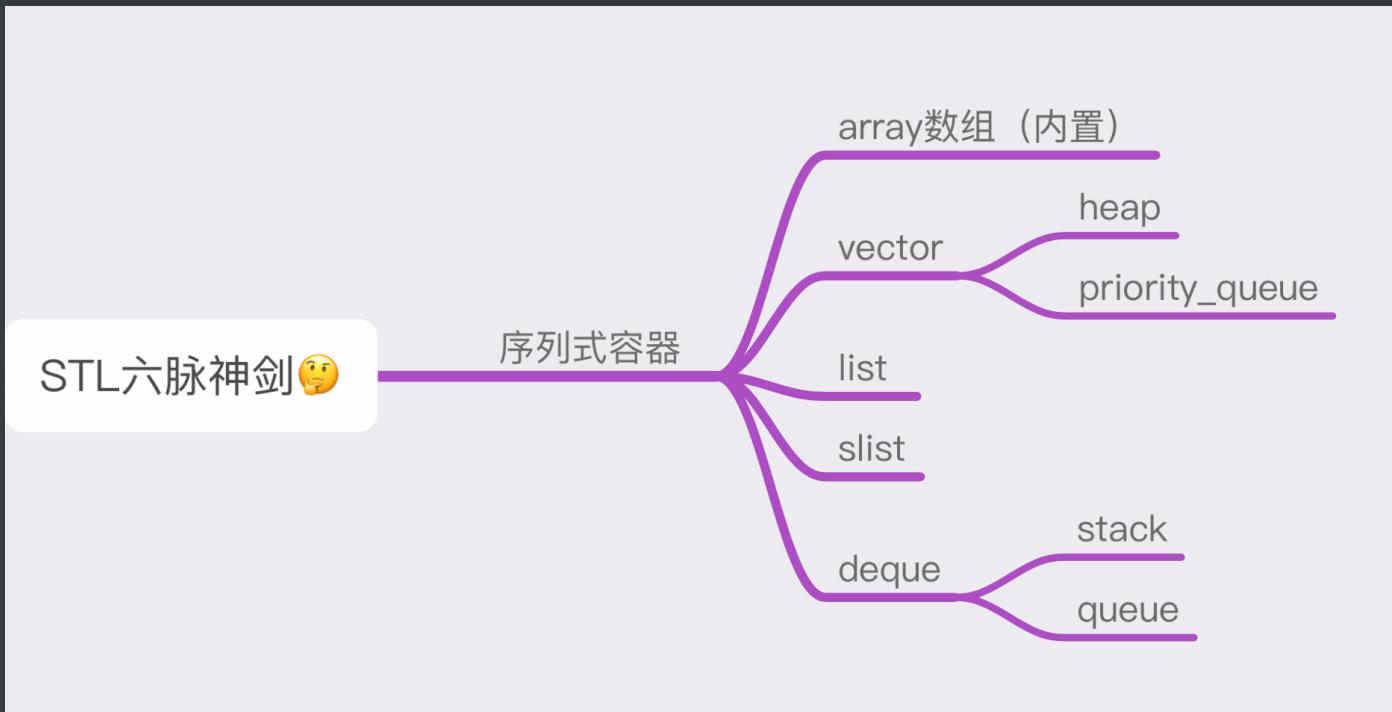
源码之前，了无秘密。

上一篇，我们剖析了 `STL` 迭代器源码与 `traits` 编程技法，这一篇我们来学习下容器。

在 `STL` 编程中，容器是我们经常会用到的一种数据结构，容器分为序列式容器和关联式容器。

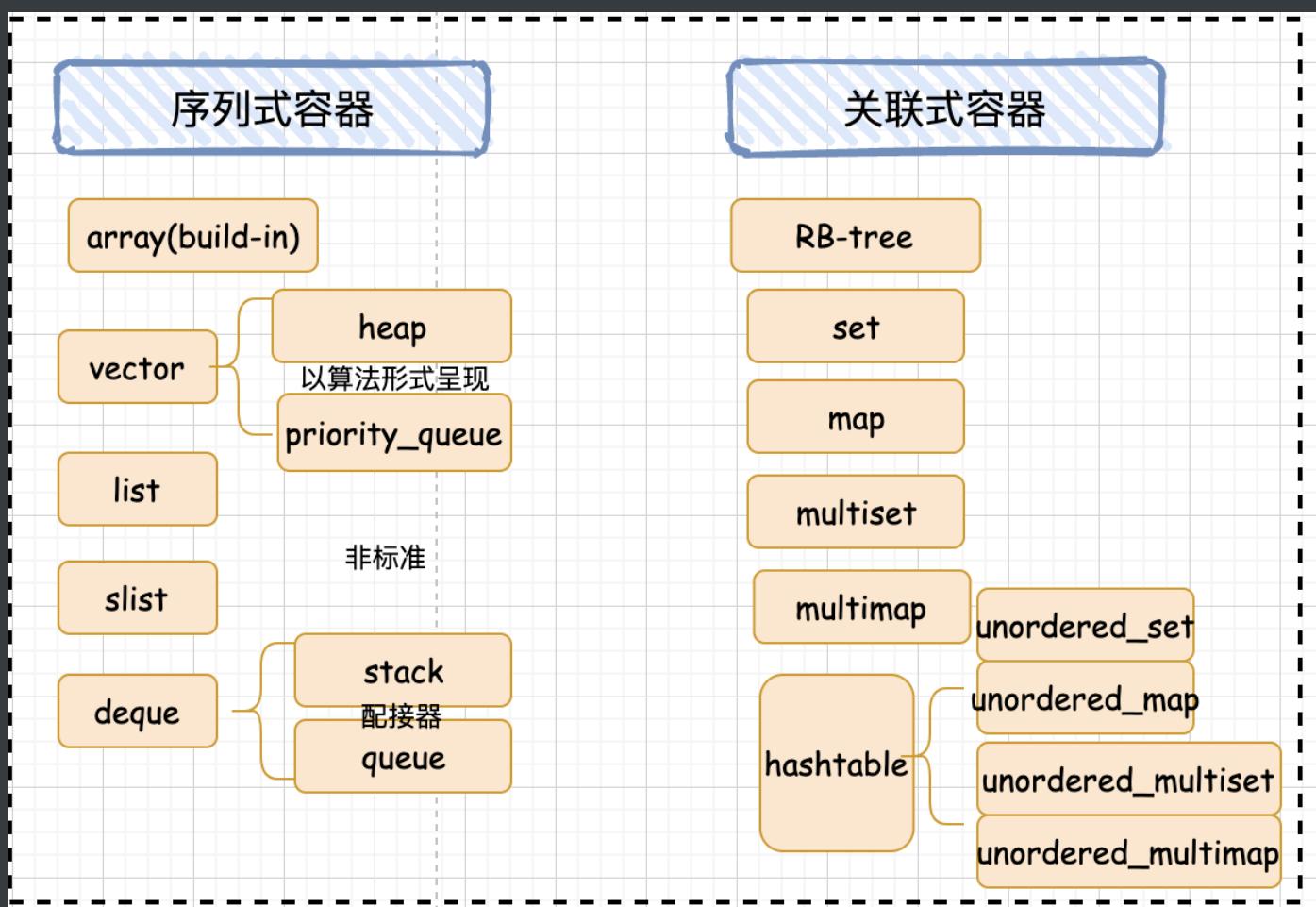
两者的本质区别在于：序列式容器是通过元素在容器中的位置顺序存储和访问元素，而关联容器则是通过键 (`key`) 存储和读取元素。

本篇着重剖析序列式容器相关背后的的知识点。



15.2 容器分类

前面提到了，根据元素存储方式的不同，容器可分为序列式和关联式，那具体的又有哪些分类呢，这里我画了一张图来看一下。



限于篇幅，这篇文章小贺会来重点讲解一下经常使用到的那些容器，比如 vector, list, deque，以及衍生的栈和队列其背后核心的设计和奥秘，不多 BB，马上就来分析。

15.3 vector

写 C++ 的小伙伴们，应该对 vector 都非常熟悉了，vector 基本能够支持任何类型的对象，同时它也是一个可以动态增长的数组，使用起来非常的方便。

但如果我问你，知道它是如何做到动态扩容的吗？哎，是不是一时半会答不上来了，哈哈，没事，我们一起来看看。

vector 基本数据结构

基本上，STL 里面所有的容器的源码都包含至少三个部分：

- 迭代器，遍历容器的元素，控制容器空间的边界和元素的移动；
- 构造函数，满足容器的多种初始化；
- 属性的获取，比如 begin(), end() 等；

vector 也不例外，其实看了源码之后就发现，vector 相反是所有容器里面最简单的一种。

```
template <class T, class Alloc = allocator>
class vector {
public:
    // 定义 vector 自身的嵌套型别
    typedef T value_type;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    // 定义迭代器，这里就只是一个普通的指针
    typedef value_type* iterator;
    typedef const value_type* const_iterator;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    ...
}
```

```

protected:
    typedef simple_alloc<value_type, Alloc> data_allocator; // 设置其空间
配置器
    iterator start;          // 当前使用空间的头
    iterator finish;         // 当前使用空间的尾
    iterator end_of_storage; // 当前可用空间的尾
    ...
};


```

因为 vector 需要表示用户操作的当前数据的起始地址，结束地址，还需要其真正的最大地址。所以总共需要 3 个迭代器分别指向：数据的头(start)，数据的尾(finish)，数组的尾(end_of_storage)。

构造函数

vector 有多个构造函数，为了满足多种初始化。

```

vector() : start(0), finish(0), end_of_storage(0) {}           // 默认构造函数
explicit vector(size_type n) { fill_initialize(n, T()); }      // 必须显示的调用这个构造函数，接受一个值
vector(size_type n, const T& value) { fill_initialize(n, value); } // 接受一个大小和初始化值。int 和 long 都执行相同的函数初始化
vector(int n, const T& value) { fill_initialize(n, value); }
vector(long n, const T& value) { fill_initialize(n, value); }
vector(const vector<T, Alloc>& x); // 接受一个vector参数的构造函数

```

其中 `fill_initialize` 内部调用 `allocate_and_fill` 函数初始化，并调整迭代器的位置

```

void fill_initialize(size_type n, const T& value) {
    start = allocate_and_fill(n, value);           // 初始化并初始化值
    finish = start + n;
    end_of_storage = finish;
}

// 调用默认的第二配置器分配内存，分配失败就释放所分配的内存
iterator allocate_and_fill(size_type n, const T& x) {
    iterator result = data_allocator::allocate(n);   // 申请n个元素的线性空间。
    __STL_TRY // 对整个线性空间进行初始化，如果有一个失败则删除全部空间并抛出异常。
    {
        uninitialized_fill_n(result, n, x);
        return result;
    }
    __STL_UNWIND(data_allocator::deallocate(result, n));
}

```

析构函数就是直接调用 `deallocate` 空间配置器，从头释放到数据尾部，最后将内存还给空间配置器。



我们看到，这里面，初始化满足要么都初始化成功，要么一个都不初始化并释放掉抛出异常，异常机制这块拿捏的死死的呀。

因为 vector 是一种 class template，所以呢，我们并不需要手动的释放内存，生命周期结束后就自动调用析构从而释放调用空间，当然我们也可以直接调用析构函数释放内存。

```

void deallocate() {
    if (start)
        data_allocator::deallocate(start, end_of_storage - start);
}

// 调用析构函数并释放内存
~vector() {
    destroy(start, finish);
    deallocate();
}

```

属性获取

下面的部分就涉及到了位置参数的获取，比如返回 `vector` 的开始和结尾，返回最后一个元素，返回当前元素个数，元素容量，是否为空等。

这里需要注意的是因为 `end()` 返回的是 `finish`，而 `finish` 是指向最后一个元素的后一个位置的指针，所以使用 `end()` 的时候要注意。

```

public:
    // 获取数据的开始以及结束位置的指针。记住这里返回的是迭代器，也就是 vector 迭代器
    // 就是该类型的指针。
    iterator begin() { return start; }
    iterator end() { return finish; }
    reference front() { return *begin(); } // 获取值
    reference back() { return *(end() - 1); }
    const_iterator begin() const { return start; } // 获取右值
    const_iterator end() const { return finish; }

    const_reference front() const { return *begin(); }
    const_reference back() const { return *(end() - 1); }

    size_type size() const { return size_type(end() - begin()); } // 数
    // 组元素的个数
    size_type max_size() const { return size_type(-1) / sizeof(T); } // 最大能存储的元素个数

```

```
size_type capacity() const { return size_type(end_of_storage - begin()); } // 数组的实际大小
bool empty() const { return begin() == end(); }
//判断 vector 是否为空， 并不是比较元素为 0， 是直接比较头尾指针。
```

push 和 pop 操作

vector 的 push 和 pop 操作都只是对尾进行操作， 这里说的尾部是指数据的尾部。

当调用 push_back 插入新元素的时候，首先会检查是否有备用空间，如果有就直接在备用空间上构造元素，并调整迭代器 finish。

```
void push_back(const T& x) {
    if (finish != end_of_storage) { //尚有可用空间
        construct(finish, x);      //全局函数
        ++finish;                  //调整高度
    } else {
        insert_aux(end(), x);    //数组被填充满，调用insert_aux必须重新寻找新的更大的连续空间，再进行插入
    }
}
```



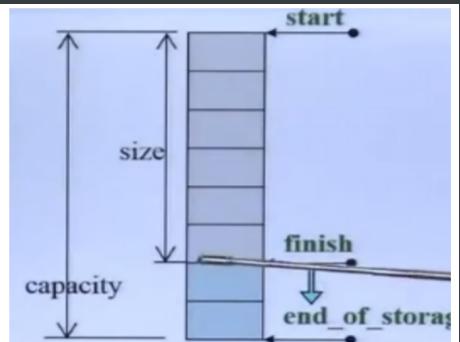
```
//<memory> -> <stl_construct.h>
#include <new.h>
template <class T1, class T2>
inline void construct(T1 *p, const T2& value) {
    new (p) T1(value);
}
```

当如果没有备用空间，就扩充空间(重新配置-移动数据-释放原空间)，这里则是调用了 insert_aux 函数。

```

void push_back(const T& x) {
    if (finish != end_of_storage) { //尚有可用空间
        construct(finish, x); //全局构造函数
        ++finish; //调整迭代器
    }
    else //已无可用空间
        insert_aux(end(), x);
}

```



```

template<class T, class Alloc>
void vector<T, Alloc>::insert_aux(iterator position, const T& x) {
    if (finish != end_of_storage) { //为什么又重新判断一遍呢?
        construct(finish, *(finish-1)); //在备用空间起始处构造一个元素并以vector最后一个元素值为初始值
        ++finish;
        T x_copy = x;
        copy_backward(position, finish - 2, finish - 1); //TODO
        *position = x_copy;
    } else { //没有备用空间 下一页 }
}

```

全局函数: 将 [first, last) 内的元素从 pos 从后往前拷贝

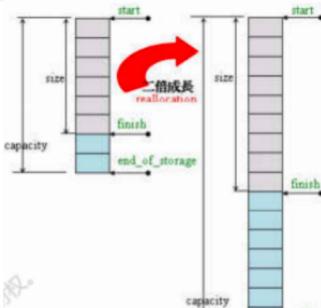
在上面这张图里，可以看到，push_back 这个函数里面又判断了一次 finish != end_of_storage 这是因为啥呢？原来这是因为 insert_aux 函数可能还被其他函数调用哦。

在下面的 else 分支里面，我们看到了 vector 的动态扩容机制：如果原空间大小为 0 则分配 1 个元素，如果大于 0 则分配原空间两倍的新空间，然后把数据拷贝过去。

```

else { //没有备用空间
    const size_type old_size = size();
    const size_type len = old_size != 0 ? 2 * old_size : 1;
    //以上分配原则:如果原大小为0则分配1个元素 因为 0 的任意倍数仍然是0
    //如果原大小不为0则分配原大小的两倍
    //前半段用来放置原数据后半段用来放置新数据
    iterator new_start = data_allocator::allocate(len);
    iterator new_finish = new_start;
    try {
        ...
    }
    catch(...) {
        ...
    }
    //构造并释放原 vector
    destroy(begin(), end());
    deallocate();
    //调整迭代器指向新 vector
    start = new_start;
    finish = new_finish;
    end_of_storage = new_start + len;
}

```



```

try {
    //将原来的vector的内容拷贝到新vector//为新元素赋值
    //拷贝完插点后的内容 因为也可能被insert(p,x)调用
    new_finish = uninitialized_copy(start, position,
                                     new_start);
    construct(new_finish, x);
    ++new_finish;
    new_finish = uninitialized_copy(position, finish,
                                     new_finish);
}
catch(const std::exception& e) {
    //commit or rollback"
    destroy(new_start, new_finish);
    data_allocator::deallocate(new_start, len);
    throw;
}

```

pop 元素

```
public:  
    //将尾端元素拿掉 并调整大小  
    void pop_back() {  
        --finish; //将尾端标记往前移动一个位置 放弃尾端元素  
        destroy(finish);  
    }
```

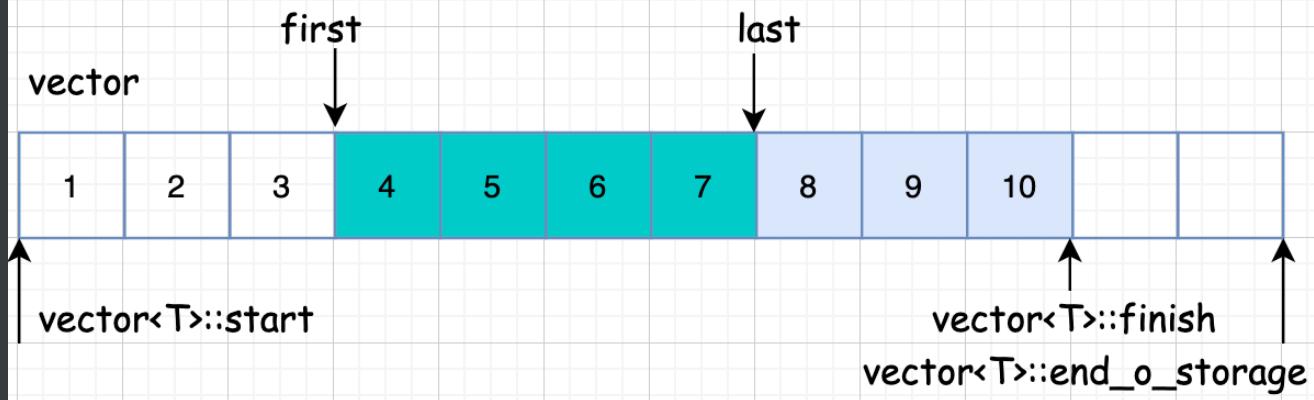
erase 删除元素

erase 函数清除指定位置的元素， 其重载函数用于清除一个范围内的所有元素。实际实现就是将删除元素后面所有元素往前移动，对于 vector 来说删除元素的操作开销还是很大的，所以说 vector 它不适合频繁的删除操作，毕竟它是一个数组。

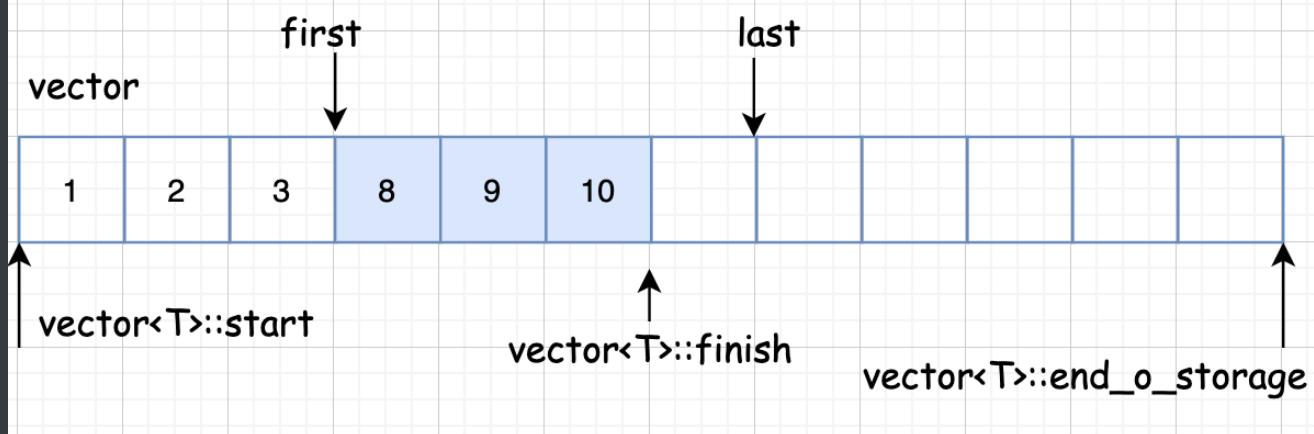
```
//清楚[first, last)中的所有元素  
iterator erase(iterator first, iterator last) {  
    iterator i = copy(last, finish, first);  
    destroy(i, finish);  
    finish = finish - (last - first);  
    return first;  
}  
  
//清除指定位置的元素  
iterator erase(iterator position) {  
    if (position + 1 != end())  
        copy(position + 1, finish, position); //copy 全局函数  
    --finish;  
    destroy(finish);  
    return position;  
}  
  
void clear() {  
    erase(begin(), end());  
}
```

我们结合图解来看一下：

erase(first, last)之前



erase(first, last)之后：删除了4个元素



清楚范围内的元素，第一步要将 `finish` 迭代器后面的元素拷贝回去，然后返回拷贝完成的尾部迭代器，最后在删除之前的。

删除指定位置的元素就是实际就是将指定位置后面的所有元素向前移动，最后析构掉最后一个元素。

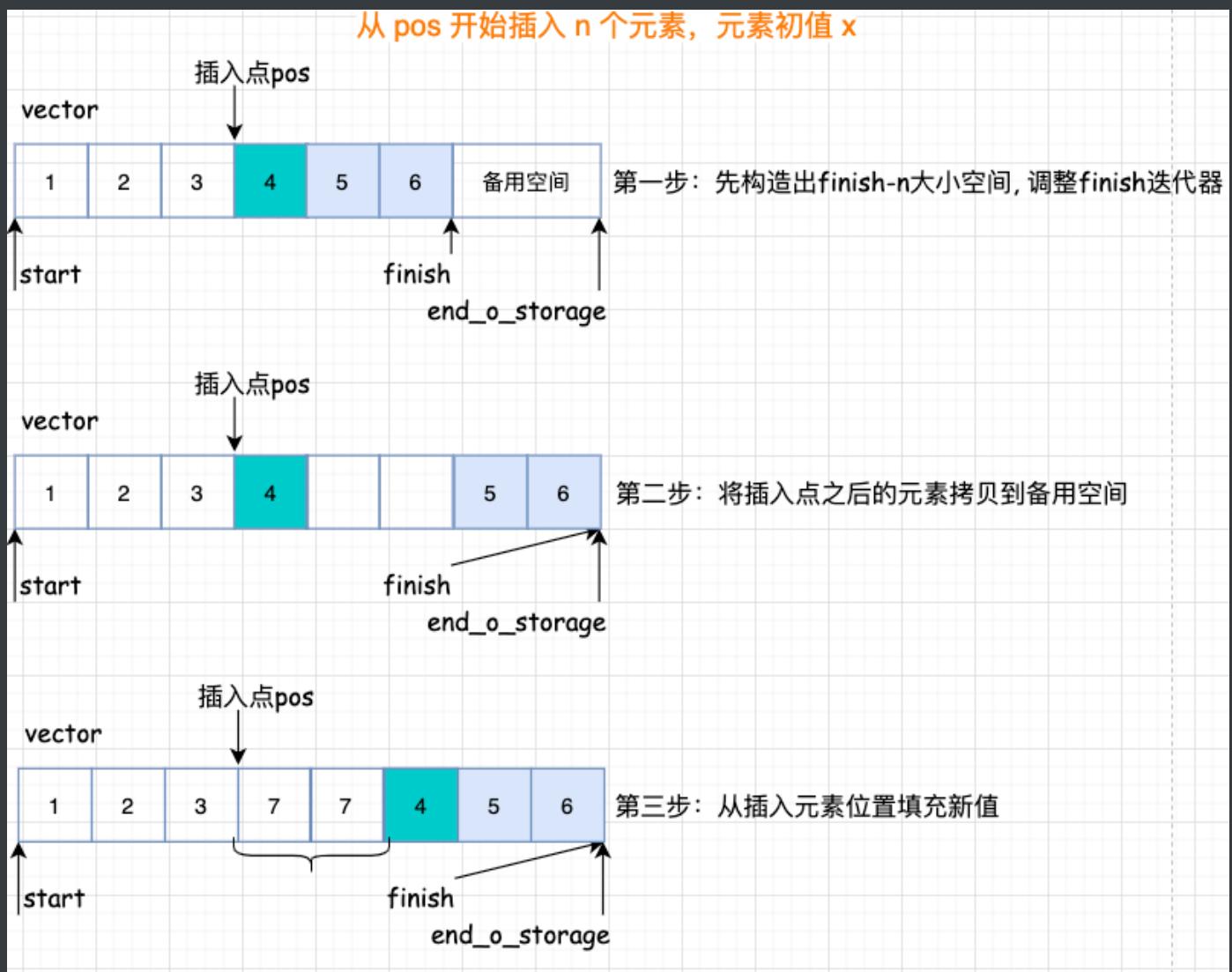
insert 插入元素

vector 的插入元素具体来说呢，又分三种情况：

- 1、如果备用空间足够且插入点的现有元素多于新增元素；
- 2、如果备用空间足够且插入点的现有元素小于新增元素；
- 3、如果备用空间不够；

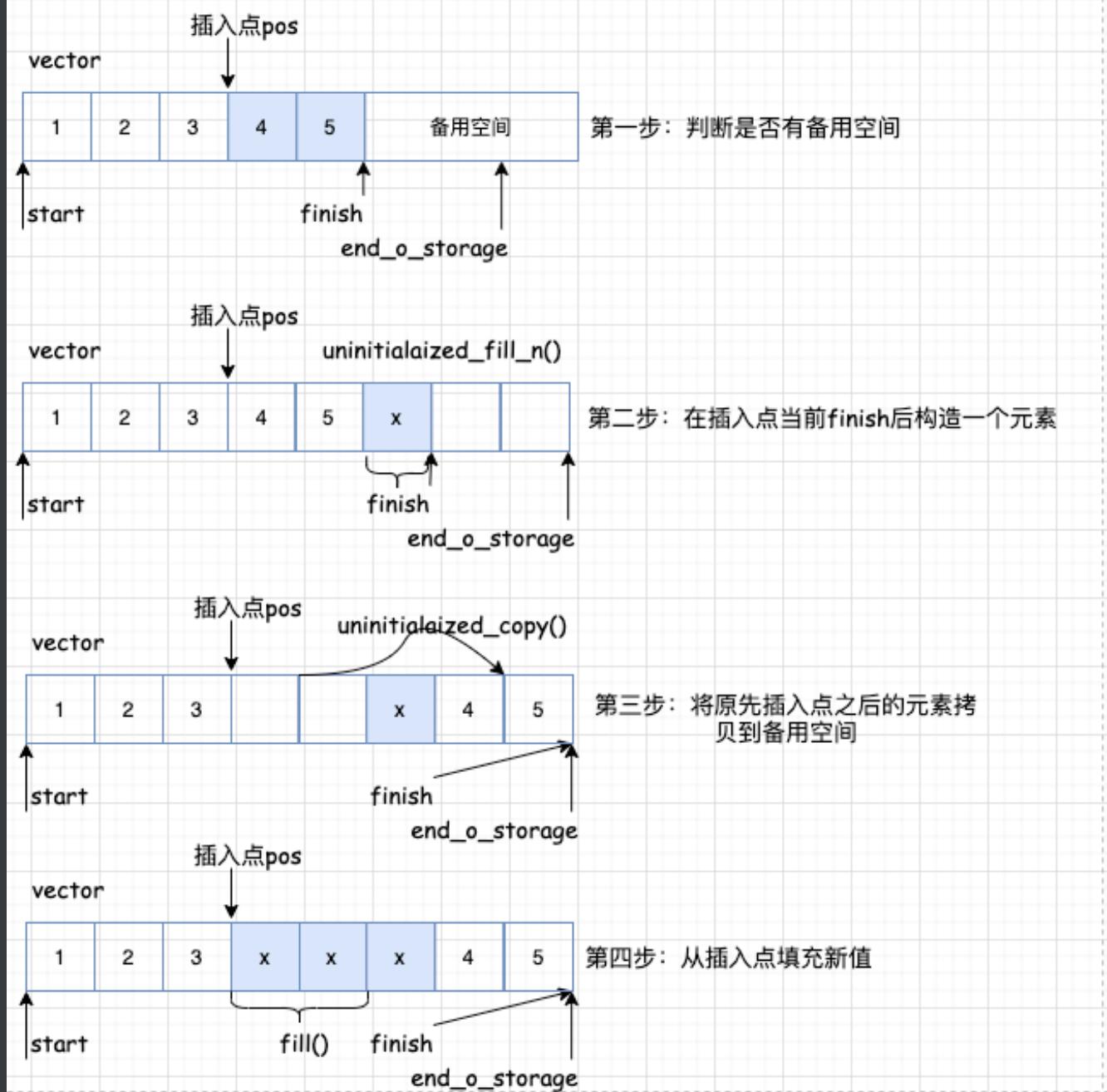
我们一个一个来分析。

- 插入点之后的现有元素个数 > 新增元素个数

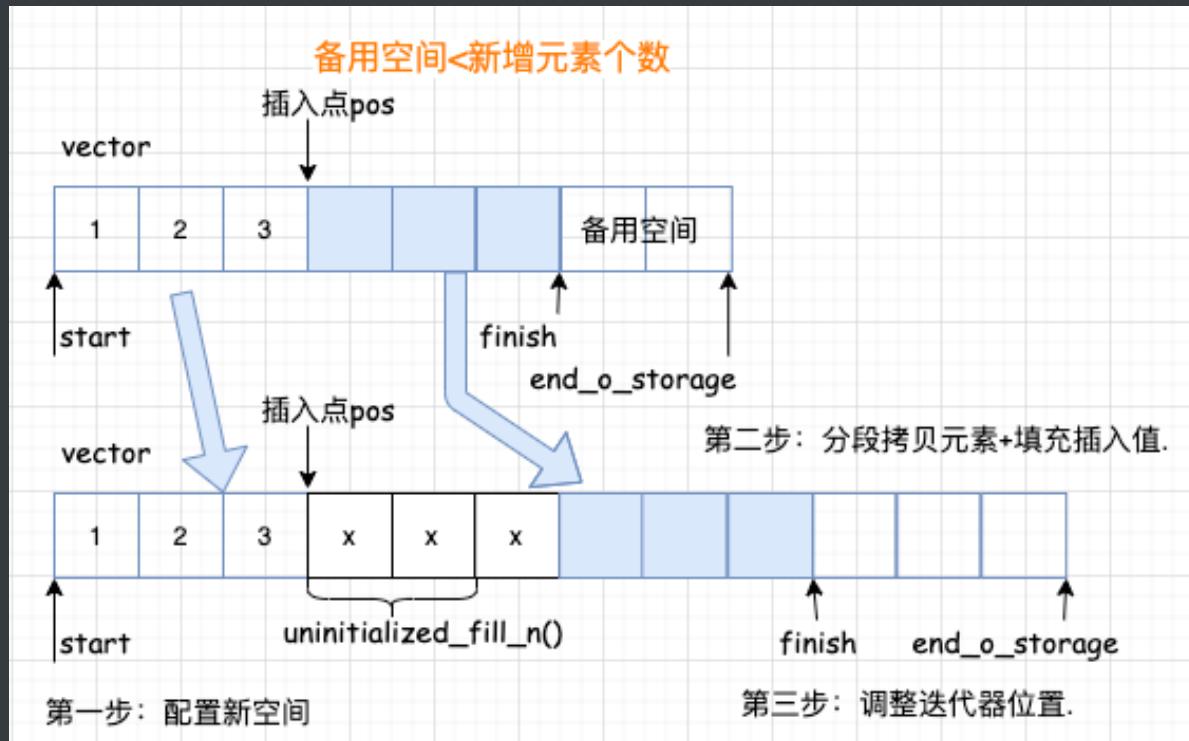


- 插入点之后的现有元素个数 <= 新增元素个数

从 pos 开始插入 n 个元素，元素初值 x



如果备用空间不足



这里呢，要注意一个坑，就是所谓的**迭代器失效问题**。

通过图解我们就明白了，所谓的迭代器失效问题是由于元素空间重新配置导致之前的迭代器访问的元素不在了，总结来说有两种：

- 由于插入元素，使得容器元素整体迁移导致存放原容器元素的空间不再有效，从而使得指向原空间的迭代器失效；
- 由于删除元素，使得某些元素次序发生变化导致原本指向某元素的迭代器不再指向期望指向的元素。

前面提到的一些全局函数，这里总结一下：

- `copy(a,b,c)`: 将(a,b)之间的元素拷贝到(c,c-(b-a))位置
- `uninitialized_copy(first, last, result)`: 具体作用是将 [first,last)内的元素拷贝到 result 从前往后拷贝
- `copy_backward(first, last, result)`: 将 [first,last)内的元素拷贝到 result 从后往前拷贝

vector 总结

到这里呢，vector 分析的就差不多了，最后提醒需要注意的是：vector 的成员函数都不做边界检查 (at方法会抛异常)，使用者要自己确保迭代器和索引值的合法性。

我们来总结一下 vector 的优缺点。

优点

- 在内存中分配一块连续的内存空间进行存，可以像数组一样操作，动态扩容。
- 随机访问方便，支持下标访问和vector.at()操作。
- 节省空间。

缺点

- 由于其顺序存储的特性，vector 插入删除操作的时间复杂度是 $O(n)$ 。
- 只能在末端进行pop和push。
- 当动态长度超过默认分配大小后，要整体重新分配、拷贝和释放空间。

vector的缺点也很明显，在频率较高的插入和删除时效率就太低了

15.4 list

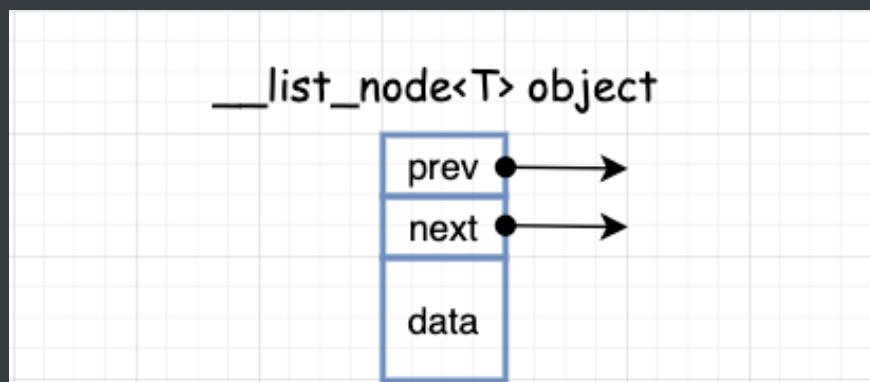
好了，下面我们来看一下 list，list 是一种双向链表。

list 的设计更加复杂一点，好处是每次插入或删除一个元素，就配置或释放一个元素，list 对于空间的运用有绝对的精准，一点也不浪费。而且对于任何位置的元素插入或删除，list 永远是常数空间。

注意：list 源码里其实分了两个部分，一个部分是 list 结构，另一部分是 list 节点的结构。

那这里不妨思考一下，为什么 list 节点分为了两个部分，而不是在一个结构体里面呢？也就是说为什么指针变量和数据变量分开定义呢？

如果看了后面的源码就晓得了，这里是为了给迭代器做铺垫，因为迭代器遍历的时候不需要数据成员的，只需要前后指针就可以遍历该 list。

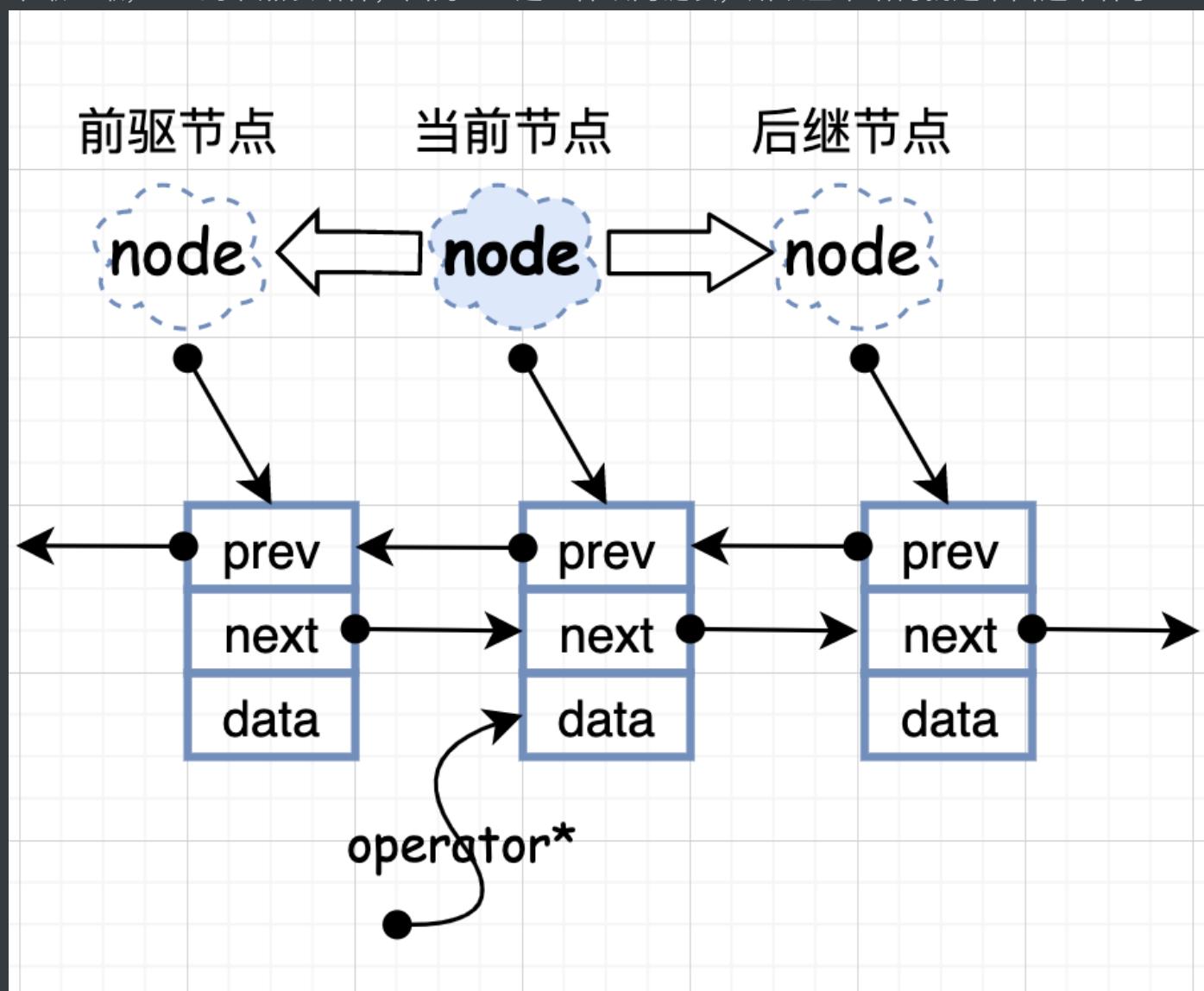


list 数据结构-节点

__list_node 用来实现节点，数据结构中就储存前后指针和属性。

```
template <class T> struct __list_node {  
    // 前后指针  
    typedef void* void_pointer;  
    void_pointer next;  
    void_pointer prev;  
    // 属性  
    T data;  
};
```

来瞅一瞅，list 的节点长啥样，因为 list 是一种双向链表，所以基本结构就是下面这个样子：



基本类型

```
template<class T, class Ref, class Ptr> struct __list_iterator {  
    typedef __list_iterator<T, T&, T*> iterator; // 迭代器  
    typedef __list_iterator<T, const T&, const T*> const_iterator;  
    typedef __list_iterator<T, Ref, Ptr> self;  
  
    // 迭代器是bidirectional_iterator_tag类型  
    typedef bidirectional_iterator_tag iterator_category;  
    typedef T value_type;  
    typedef Ptr pointer;  
    typedef Ref reference;  
    typedef size_t size_type;  
    typedef ptrdiff_t difference_type;  
    ...  
};
```

构造函数

```
template<class T, class Ref, class Ptr> struct __list_iterator {  
    ...  
    // 定义节点指针  
    typedef __list_node<T>* link_type;  
    link_type node;  
    // 构造函数  
    __list_iterator(link_type x) : node(x) {}  
    __list_iterator() {}  
    __list_iterator(const iterator& x) : node(x.node) {}  
    ...  
};
```

重载

```
template<class T, class Ref, class Ptr> struct __list_iterator {  
    ...
```

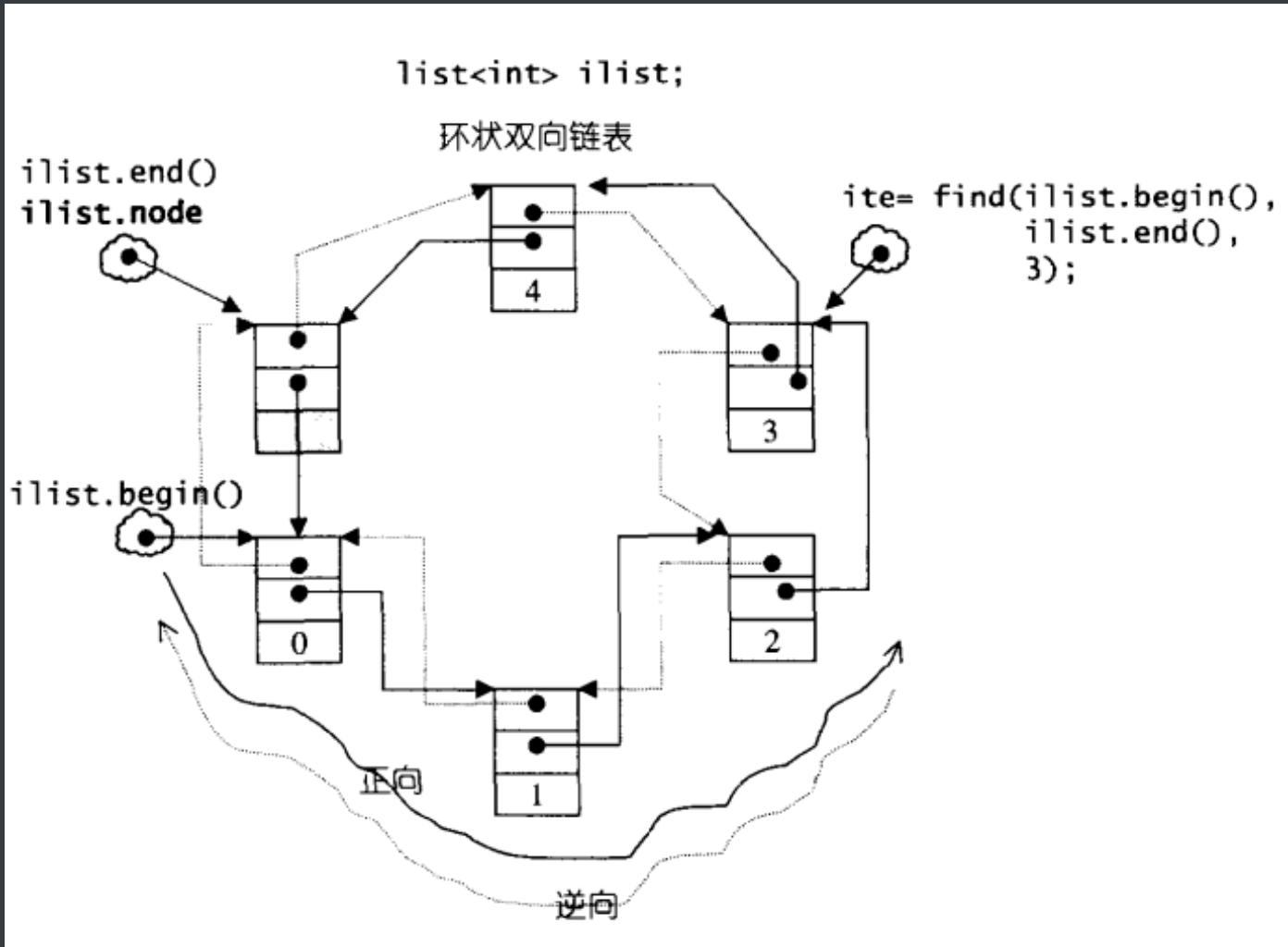
```
// 重载
bool operator==(const self& x) const { return node == x.node; }
bool operator!=(const self& x) const { return node != x.node; }
...

// ++和--是直接操作的指针指向next还是prev，因为list是一个双向链表
self& operator++() {
    node = (link_type)((*node).next);
    return *this;
}
self operator++(int) {
    self tmp = *this;
    ++*this;
    return tmp;
}
self& operator--() {
    node = (link_type)((*node).prev);
    return *this;
}
self operator--(int) {
    self tmp = *this;
    --*this;
    return tmp;
}
};

};
```

list 结构

list 自己定义了嵌套类型满足 traits 编程， list 迭代器是 bidirectional_iterator_tag 类型，并不是一个普通指针。



list在定义 node 节点时， 定义的不是一个指针。这里要注意。

```

template <class T, class Alloc = alloc>
class list {
protected:
    typedef void* void_pointer;
    typedef __list_node<T> list_node; // 节点
    typedef simple_alloc<list_node, Alloc> list_node_allocator; // 空间配
置器
public:
    // 定义嵌套类型
    typedef T value_type;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef value_type& reference;
    typedef const value_type& const_reference;

```

```
typedef list_node* link_type;
typedef size_t size_type;
typedef ptrdiff_t difference_type;

protected:
    // 定义一个节点，这里节点并不是一个指针.
    link_type node;

public:
    // 定义迭代器
    typedef __list_iterator<T, T&, T*> iterator;
    typedef __list_iterator<T, const T&, const T*> const_iterator;
    ...
};
```

list 构造和析构函数实现

构造函数前期准备：

每个构造函数都会创造一个空的 node 节点，为了保证我们在执行任何操作都不会修改迭代器。

list 默认使用 alloc 作为空间配置器，并根据这个另外定义了一个 list_node_allocator，目的是更加方便以节点大小来配置单元。

```
template <class T, class Alloc = alloc>
class list {
protected:
    typedef void* void_pointer;
    typedef __list_node<T> list_node; // 节点
    typedef simple_alloc<list_node, Alloc> list_node_allocator; // 空间配置器
```

其中，list_node_allocator(n)表示配置 n 个节点空间。以下四个函数，分别用来配置，释放，构造，销毁一个节点。

```

class list {
protected:
    // 配置一个节点并返回
    link_type get_node() { return list_node_allocator::allocate(); }
    // 释放一个节点
    void put_node(link_type p) { list_node_allocator::deallocate(p); }
    // 产生(配置并构造)一个节点带有元素初始值
    link_type create_node(const T& x) {
        link_type p = get_node();
        __STL_TRY {
            construct(&p->data, x);
        }
        __STL_UNWIND(put_node(p));
        return p;
    }
    // 销毁(析构并释放)一个节点
    void destroy_node(link_type p) {
        destroy(&p->data);
        put_node(p);
    }
    // 对节点初始化
    void empty_initialize() {
        node = get_node();
        node->next = node;
        node->prev = node;
    }
};

```

基本属性获取

```

template <class T, class Alloc = alloc>
class list {
    ...
public:
    iterator begin() { return (link_type)((*node).next); } // 返回指向头的
指针

```

```
const_iterator begin() const { return (link_type)((*node).next); }
iterator end() { return node; } // 返回最后一个元素的后一个的地址
const_iterator end() const { return node; }

// 这里是为旋转做准备， rbegin返回最后一个地址， rend返回第一个地址. 我们放在配
接器里面分析
reverse_iterator rbegin() { return reverse_iterator(end()); }
const_reverse_iterator rbegin() const {
    return const_reverse_iterator(end());
}
reverse_iterator rend() { return reverse_iterator(begin()); }
const_reverse_iterator rend() const {
    return const_reverse_iterator(begin());
}

// 判断是否为空链表， 这是判断只有一个空node来表示链表为空.
bool empty() const { return node->next == node; }
// 因为这个链表， 地址并不连续， 所以要自己迭代计算链表的长度.
size_type size() const {
    size_type result = 0;
    distance(begin(), end(), result);
    return result;
}
size_type max_size() const { return size_type(-1); }
// 返回第一个元素的值
reference front() { return *begin(); }
const_reference front() const { return *begin(); }
// 返回最后一个元素的值
reference back() { return *(--end()); }
const_reference back() const { return *(--end()); }

// 交换
void swap(list<T, Alloc>& x) { __STD::swap(node, x.node); }
...
};

template <class T, class Alloc>
```

```
inline void swap(list<T, Alloc>& x, list<T, Alloc>& y) {
    x.swap(y);
}
```

list 的头插和尾插

因为 list 是一个循环的双链表, 所以 push 和 pop 就必须实现是在头插入, 删除还是在尾插入和删除。

在 list 中, push 操作都调用 insert 函数, pop 操作都调用 erase 函数。

```
template <class T, class Alloc = alloc>
class list {
    ...
    // 直接在头部或尾部插入
    void push_front(const T& x) { insert(begin(), x); }
    void push_back(const T& x) { insert(end(), x); }
    // 直接在头部或尾部删除
    void pop_front() { erase(begin()); }
    void pop_back() {
        iterator tmp = end();
        erase(--tmp);
    }
    ...
};
```

上面的两个插入函数内部调用的 insert 函数。

```
class list {  
    ...  
public:  
    // 最基本的insert操作，之插入一个元素  
    iterator insert(iterator position, const T& x) {  
        // 将元素插入指定位置的前一个地址  
        link_type tmp = create_node(x);  
        tmp->next = position.node;  
        tmp->prev = position.node->prev;  
        (link_type(position.node->prev))->next = tmp;  
        position.node->prev = tmp;  
        return tmp;  
    }  
}
```

这里需要注意的是

- 节点实际是以 node 空节点开始的。
- 插入操作是将元素插入到指定位置的前一个地址进行插入的。

删除操作

删除元素的操作大都是由 `erase` 函数来实现的, 其他的所有函数都是直接或间接调用 `erase`。
`list` 是链表, 所以链表怎么实现删除, `list` 就在怎么操作: 很简单, 先保留前驱和后继节点, 再调整指针位置即可。
由于它是双向环状链表, 只要把边界条件处理好, 那么在头部或者尾部插入元素操作几乎是一样的, 同样的道理, 在头部或者尾部删除元素也是一样的。

```
template <class T, class Alloc = alloc>  
class list {  
    ...  
    iterator erase(iterator first, iterator last);  
    void clear();  
    // 参数是一个迭代器 修改该元素的前后指针指向再单独释放节点就行了  
    iterator erase(iterator position) {  
        link_type next_node = link_type(position.node->next);  
        link_type prev_node = link_type(position.node->prev);  
    }  
}
```

```
    prev_node->next = next_node;
    next_node->prev = prev_node;
    destroy_node(position.node);
    return iterator(next_node);
}
...
};

};

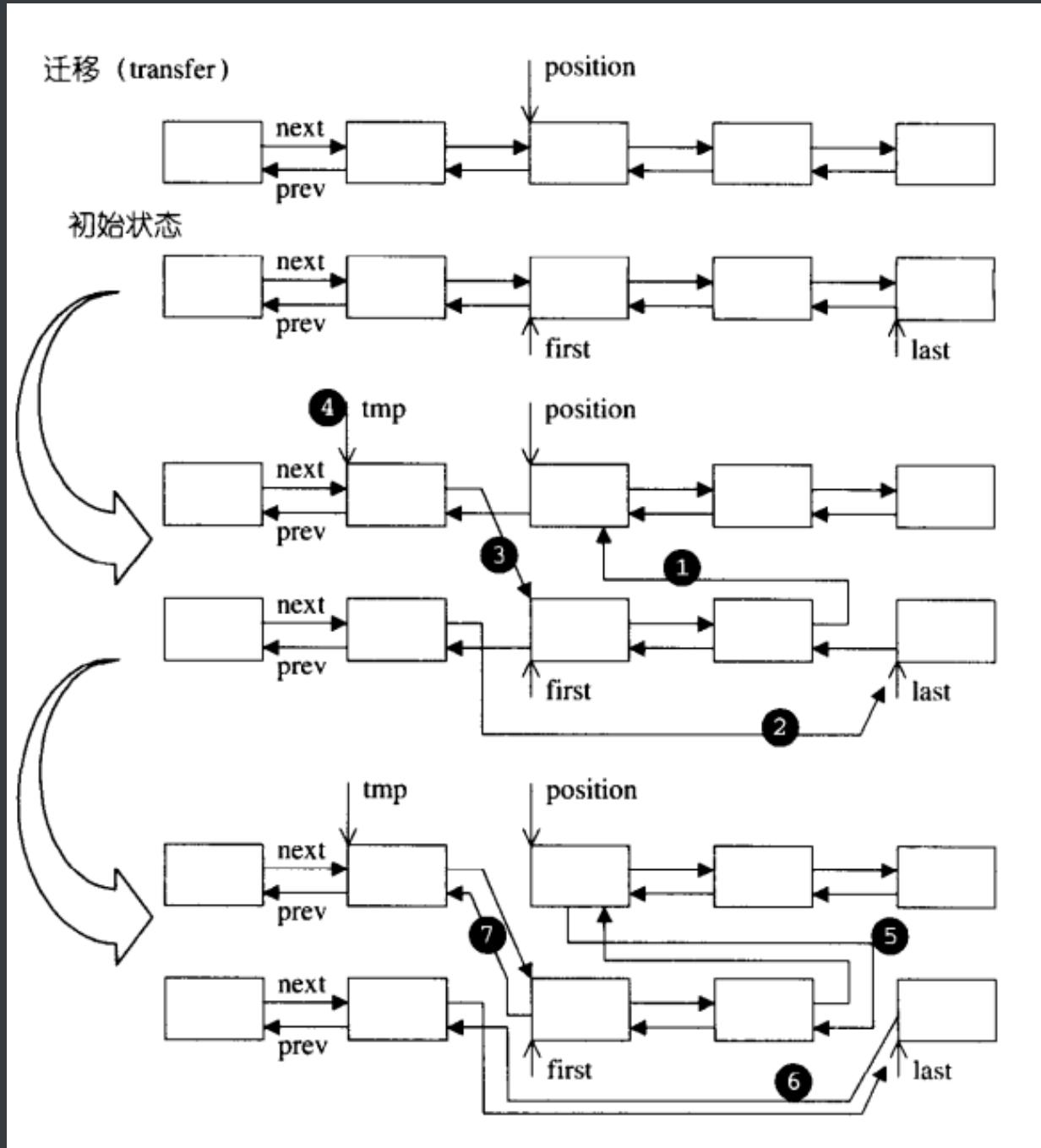
...
}
```

list 内部提供一种所谓的迁移操作(transfer): 将某连续范围的元素迁移到某个特定位置之前, 技术上实现其实不难, 就是节点之间的指针移动, 只要明白了这个函数的原理, 后面的 splice, sort, merge 函数也就一一知晓了, 我们来看一下 transfer 的源码:

```
template <class T, class Alloc = alloc>
class list {
    ...
protected:
    void transfer(iterator position, iterator first, iterator last) {
        if (position != last) {
            (*link_type((*last.node).prev)).next = position.node;
            (*link_type((*first.node).prev)).next = last.node;
            (*link_type((*position.node).prev)).next = first.node;
            link_type tmp = link_type((*position.node).prev);
            (*position.node).prev = (*last.node).prev;
            (*last.node).prev = (*first.node).prev;
            (*first.node).prev = tmp;
        }
    }
    ...
};

};
```

上面代码的七行分别对应下图的七个步骤, 看明白应该不难吧。



另外 list 的其它的一些成员函数这里限于篇幅，就不贴出源码了，简单说一些注意点。

splice 函数：将两个链表进行合并：内部就是调用的 transfer 函数。

merge 函数：将传入的 list 链表 x 与原链表按从小到大合并到原链表中(前提是两个链表都是已经从小到大排序了). 这里 merge 的核心就是 transfer 函数。

reverse 函数：实现将链表翻转的功能：主要是 list 的迭代器基本不会改变的特点，将每一个元素一个个插入到 begin 之前。

sort 函数：list 这个容器居然还自己实现一个排序，看一眼源码就发现其实内部调用的 merge 函数，用了一个数组链表用来存储 2^i 个元素，当上一个元素存储满了之后继续往下一个链表存储，最后将所有的链表进行 merge 归并(合并)，从而实现了链表的排序。

赋值操作：需要考虑两个链表的实际大小不一样时的操作

- 原链表大：复制完后要删除掉原链表多余的元素
- 原链表小：复制完后要还要将x链表的剩余元素以插入的方式插入到原链表中

resize 操作：重新修改 list 的大小。

传入一个 new_size，如果链表旧长度大于 new_size 的大小，那就删除后面多余的节点

clear 操作：清除所有节点

遍历每一个节点，销毁(析构并释放)一个节点

remove 操作：清除指定值的元素

遍历每一个节点，找到就移除

unique 操作：清除数值相同的连续元素，注意只有“连续而相同的元素”，才会被移除剩一个。

遍历每一个节点，如果在此区间段有相同的元素就移除之

感兴趣的读者可以自行去阅读体会。

好啦，list 的内容到这里就结束了。

list 总结

我们来总结一下。

list 是一种双向链表。每个结点都包含一个数据域、一个前驱指针 `prev` 和一个后驱指针 `next`。

由于其链表特性，实现同样的操作，相对于 STL 中的通用算法，list 的成员函数通常有更高的效率，内部仅需做一些指针的操作，因此尽可能选择 list 成员函数。

优点

- 不适用连续内存完成动态操作
- 在内部方便进行插入删除操作。
- 可在两端进行push和pop操作。

缺点

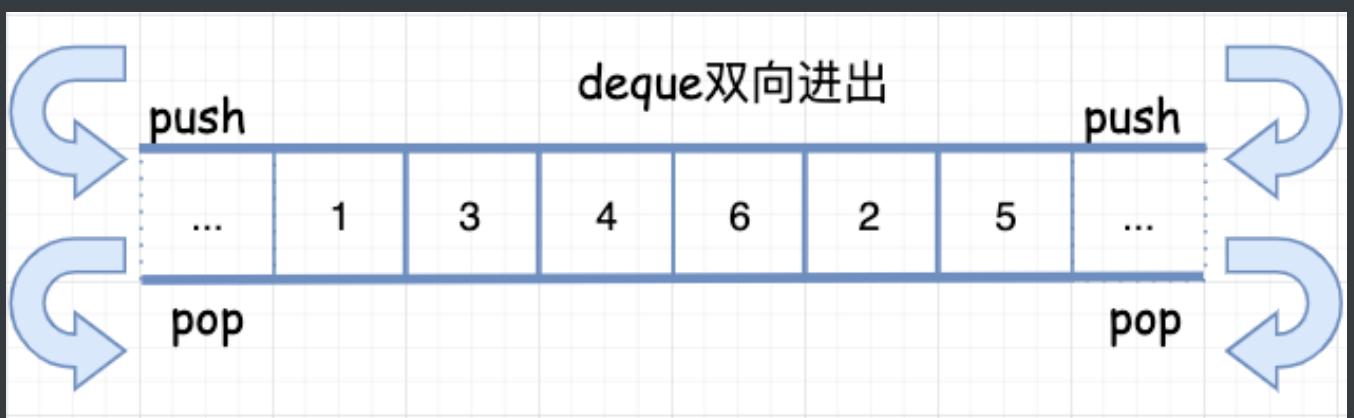
- 不支持随机访问，即下标操作和`.at()`。
- 相对于vector占用内存多。

15.5 deque

下面到了最硬核的内容了，接下来我们学习一下双端队列 `deque`。

`deque` 的功能很强大。

首先来一张图吧。



上面就是 `deque` 的示例图，`deque` 和 `vector` 的最大差异一在于 `deque` 允许常数时间内对头端或尾端进行元素的插入或移除操作。

二在于 `deque` 没有所谓的容量概念，因为它是动态地以分段连续空间组合而成随时可以增加一块新的空间并拼接起来。

虽然 `deque` 也提供 随机访问的迭代器，但它的迭代器和前面两种容器的都不一样，其设计相当复杂度和精妙，因此，会对各种运算产生一定影响，除非必要，尽可能的选择使用 `vector` 而非 `deque`。一一来探究下吧。

deque 的中控器

`deque` 在逻辑上看起来是连续空间，内部是由一段一段的定量连续空间构成。

一旦有必要在 `deque` 的前端或尾端增加新空间，便配置一段定量的连续空间，串接在整个 `deque` 的头部或尾部。

设计 `deque` 的大师们，想必是让 `deque` 的最大挑战就是在这些分段的定量连续空间上，维护其整体连续的假象，并提供其随机存取的接口，从而避开了像 `vector` 那样的“重新配置-复制-释放”开销三部曲。这样一来，虽然开销降低，却提高了复杂的迭代器架构。

因此数据结构的设计和迭代器前进或后退等操作都非常复杂。

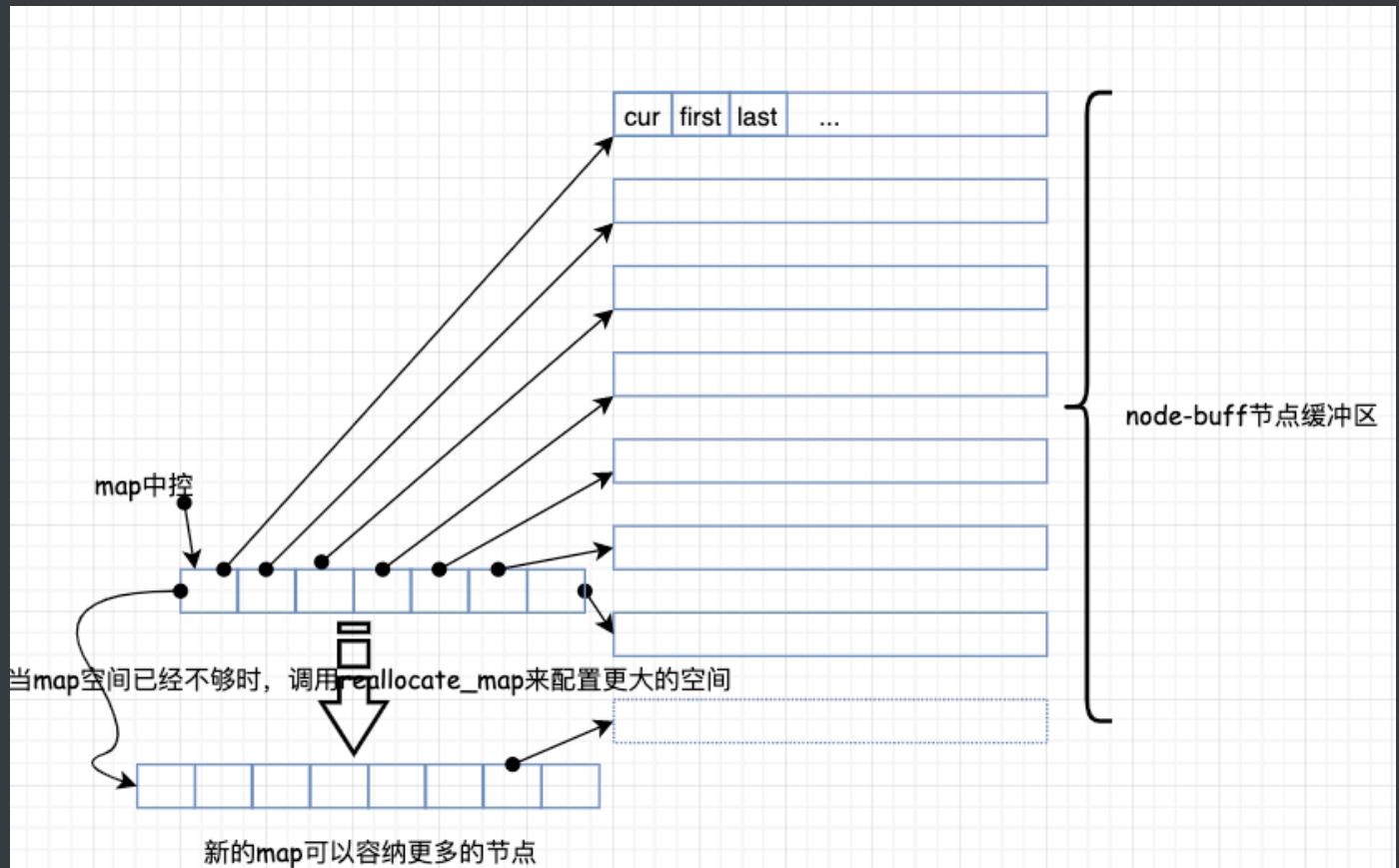
`deque` 采用一块所谓的 map（注意不是STL里面的map容器）作为中控器，其实就是一小块连续空间，其中的每个元素都是指针，指向另外一段较大的连续线性空间，称之为缓冲区。,在后面我们看到，缓冲区才是 `deque` 的储存空间主体。

```

#ifndef __STL_NON_TYPE_TMPL_PARAM_BUG
template <class T, class Ref, class Ptr, size_t BufSiz>
class deque {
public:
    typedef T value_type;
    typedef value_type* pointer;
    ...
protected:
    typedef pointer** map_pointer;
    map_pointer map; //指向 map, map 是连续空间, 其内的每个元素都是一个指针。
    size_type map_size;
    ...
};

```

其示例图如下：deque 的结构设计中，map 和 node-buffer 的关系如下：



deque 的迭代器

deque 是分段连续空间，维持其“整体连续”假象的任务，就靠它的迭代器来实现，也就是 operator++ 和 operator-- 两个运算子上面。

在看源码之前，我们可以思考一下，如果让你来设计，你觉得 deque 的迭代器应该具备什么样的结构和功能呢？

首先第一点，我们能想到的是，既然是分段连续，迭代器应该能指出当前的连续空间在哪里；

其次，第二点因为缓冲区有边界，迭代器还应该要能判断，当前是否处于所在缓冲区的边缘，如果是，一旦前进或后退，就必须跳转到下一个或上一个缓冲区；

第三点，也就是实现前面两种情况的前提，迭代器必须能随时控制中控器。

有了这样的思想准备之后，我们再来看源码，就显得容易理解一些了。

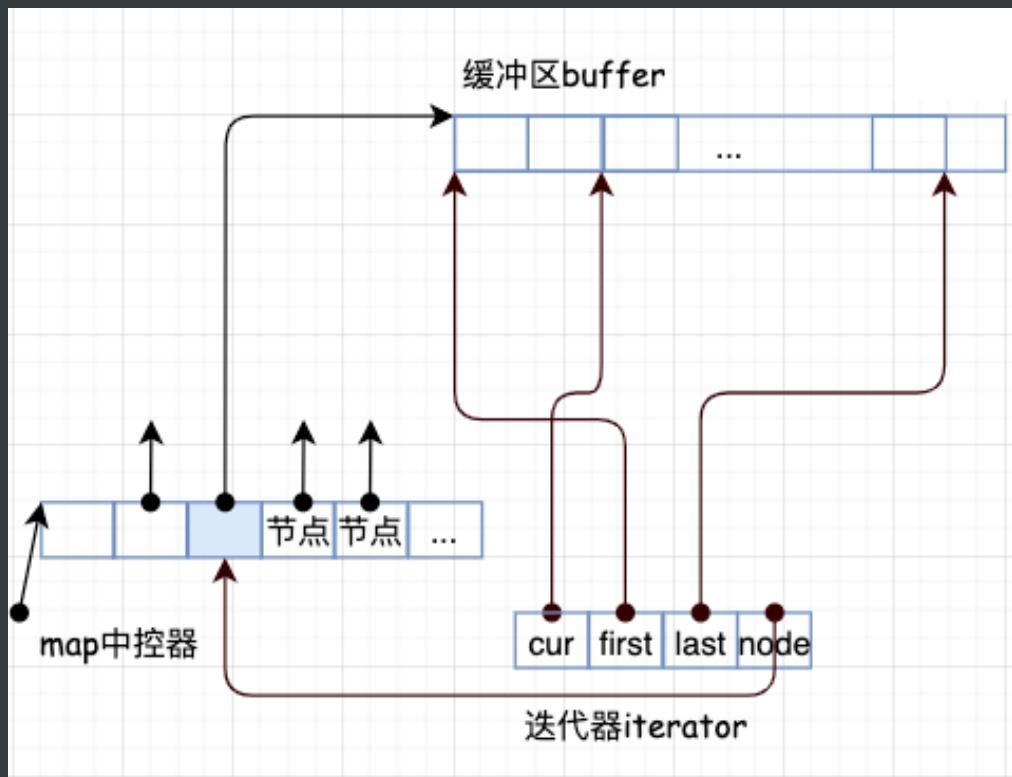
```
template <class T, class Ref, class Ptr, size_t BufSiz>
struct __deque_iterator {
    // 迭代器定义
    typedef __deque_iterator<T, T&, T*, BufSiz> iterator;
    typedef __deque_iterator<T, const T&, const T*, BufSiz>
    const_iterator;
    static size_t buffer_size() {return __deque_buf_size(BufSiz,
        sizeof(T)); }
    // deque是random_access_iterator_tag类型
    typedef random_access_iterator_tag iterator_category;
    // 基本类型的定义，满足traits编程
    typedef T value_type;
    typedef Ptr pointer;
    typedef Ref reference;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    // node
    typedef T** map_pointer;
    map_pointer node;
```

```
typedef __deque_iterator self;  
...  
};
```

deque 的每一个缓冲区由设计了三个迭代器 (为什么这样设计?)

```
struct __deque_iterator {  
    ...  
    typedef T value_type;  
    T* cur;  
    T* first;  
    T* last;  
    typedef T** map_pointer;  
    map_pointer node;  
    ...  
};
```

那，为什么要这样设计呢？回到前面我们刚才说的，因为它是分段连续的空间，下图描绘了 deque 的中控器、缓冲区、迭代器之间的相互关系：



看明白了吗，每一段都指向一个缓冲区 buffer，而缓冲区是需要知道每个元素的位置的，所以需要这些迭代器去访问。

其中 cur 表示当前所指的位置；

first 表示当前数组中头的位置；

last 表示当前数组中尾的位置。

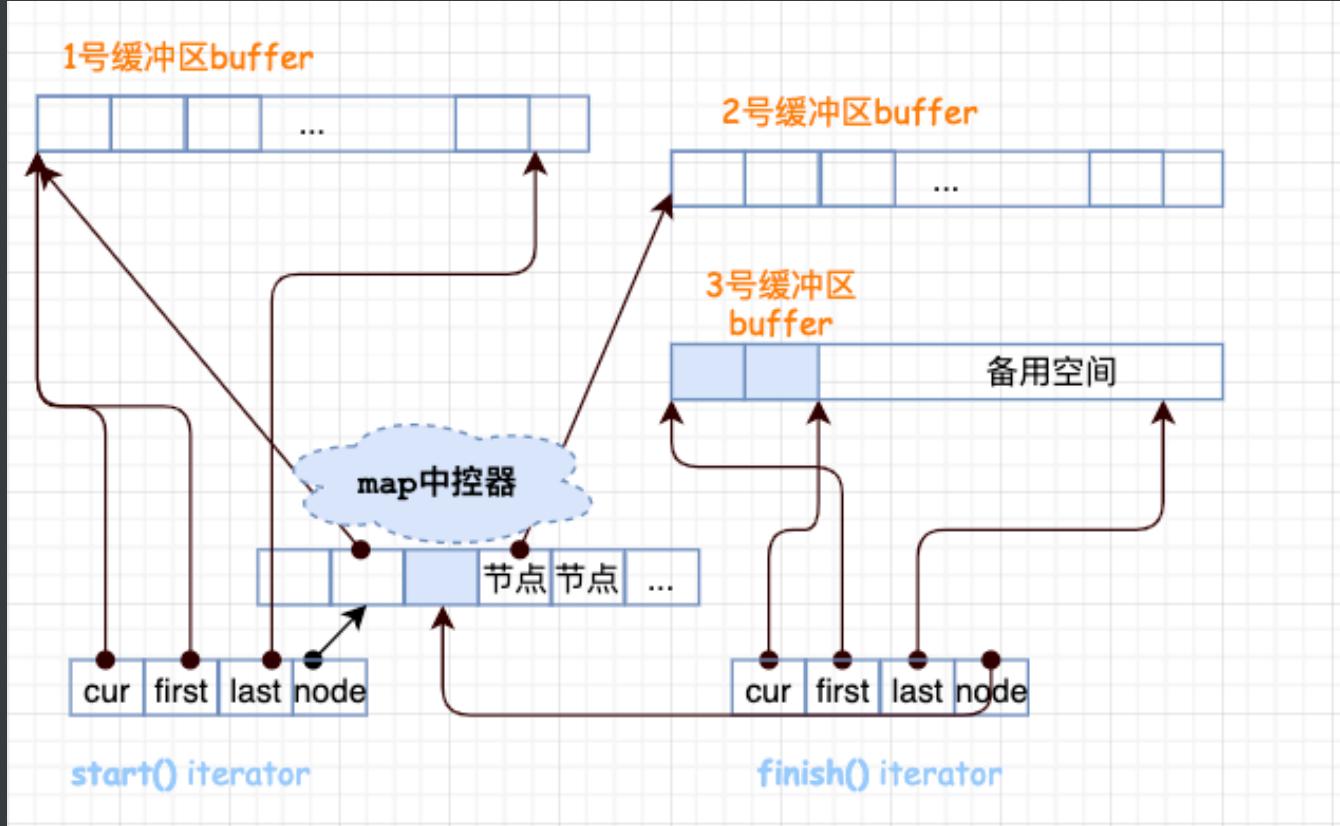
这样就方便管理，需要注意的是 deque 的空间是由 map 管理的，它是一个指向指针的指针，所以三个参数都是指向当前的数组，但这样的数组可能有多个，只是每个数组都管理这3个变量。

那么，缓冲区大小是谁来决定的呢？这里呢，用来决定缓冲区大小的是一个全局函数：

```
inline size_t __deque_buf_size(size_t n, size_t sz) {
    return n != 0 ? n : (sz < 512 ? size_t(512 / sz) : size_t(1));
}

//如果 n 不为0，则返回 n，表示缓冲区大小由用户自定义
//如果 n == 0，表示 缓冲区大小默认值
//如果 sz = (元素大小 sizeof(value_type)) 小于 512 则返回 521/sz
//如果 sz 不小于 512 则返回 1
```

假设我们现在构造了一个 int 类型的 deque，设置缓冲区大小等于 32，这样一来，每个缓冲区可以容纳 $32/\text{sizeof}(\text{int}) = 8$ (64位系统) 个元素。经过一番操作之后，deque 现在有 20 个元素了，那么成员函数 begin() 和 end() 返回的两个迭代器应该是怎样的呢？如下图所示：



20 个元素需要 $20/(\text{sizeof}(\text{int})) = 5$ (图中只展示3个) 个缓冲区。所以 map 运用了三个节点。迭代器 start 内的 cur 指针指向缓冲区的第一个元素，迭代器 finish 内的 cur 指针指向缓冲区的最后一个元素(的下一个位置)。

注意，最后一个缓冲区尚有备用空间，如果之后还有新元素插入，则直接插入到备用空间。

deque 迭代器的操作

前进和后退

operator++ 操作代表是需要切换到下一个元素，这里需要先切换再判断是否已经到达缓冲区的末尾。

```
self& operator++() {
    ++cur;           //切换至下一个元素
    if (cur == last) { //如果已经到达所在缓冲区的末尾
        set_node(node+1); //切换下一个节点
        cur = first;
    }
    return *this;
}
```

operator-- 操作代表切换到上一个元素所在的位置，需要先判断是否到达缓冲区的头部，再后退。

```
self& operator--() {
    if (cur == first) { //如果已经到达所在缓冲区的头部
        set_node(node - 1); //切换前一个节点的最后一个元素
        cur = last;
    }
    --cur;           //切换前一个元素
    return *this;
}
```

deque 的构造和析构函数

构造函数. 有多个重载函数, 接受大部分不同的参数类型. 基本上每一个构造函数都会调用 create_map_and_nodes, 这就是构造函数的核心, 待会就来分析这个函数实现.

```
template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
    ...
public:           // Basic types
    deque() : start(), finish(), map(0), map_size(0){
        create_map_and_nodes(0);
    } // 默认构造函数
    deque(const deque& x) : start(), finish(), map(0), map_size(0) {
        create_map_and_nodes(x.size());
```

```
__STL_TRY {
    uninitialized_copy(x.begin(), x.end(), start);
}
__STL_UNWIND(destroy_map_and_nodes());
}

// 接受 n:初始化大小, value:初始化的值
deque(size_type n, const value_type& value) : start(), finish(),
map(0), map_size(0) {
    fill_initialize(n, value);
}
deque(int n, const value_type& value) : start(), finish(), map(0),
map_size(0) {
    fill_initialize(n, value);
}
deque(long n, const value_type& value) : start(), finish(), map(0),
map_size(0){
    fill_initialize(n, value);
}
...
```

下面我们来学习一下 deque 的中控器是如何配置的

```
void
deque<T,Alloc,BufSize>::create_map_and_nodes(size_type num_elements) {
    //需要节点数= (每个元素/每个缓冲区可容纳的元素个数+1)
    //如果刚好整除, 多配一个节点
    size_type num_nodes = num_elements / buffer_size() + 1;
    //一个 map 要管理几个节点, 最少 8 个, 最多是需要节点数+2
    map_size = max(initial_map_size(), num_nodes + 2);
    map = map_allocator::allocate(map_size);
    // 计算出数组的头前面留出来的位置保存并在nstart.
    map_pointer nstart = map + (map_size - num_nodes) / 2;
    map_pointer nfinish = nstart + num_nodes - 1;
    map_pointer cur;//指向所拥有的节点的最中央位置
    ...
}
```

注意了，看了源码之后才知道：**deque** 的 **begin** 和 **end** 不是一开始就是指向 **map** 中控器里开头和结尾的，而是指向所拥有的节点的最中央位置。

这样带来的好处是可以使得头尾两边扩充的可能性和一样大，换句话来说，因为 **deque** 是头尾插入都是 O(1)，所以 **deque** 在头和尾都留有空间方便头尾插入。

那么，什么时候 **map** 中控器本身需要调整大小呢？触发条件在于 **reserve_map_at_back** 和 **reserve_map_at_front** 这两个函数来判断，实际操作由 **reallocate_map** 来执行。

那 **reallocate_map** 又是如何操作的呢？这里先留个悬念。

```

// 如果 map 尾端的节点备用空间不足, 符合条件就配置一个新的map(配置更大的, 拷贝原来的,
// 的, 释放原来的)
void reserve_map_at_back (size_type nodes_to_add = 1) {
    if (nodes_to_add + 1 > map_size - (finish.node - map))
        reallocate_map(nodes_to_add, false);
}

// 如果 map 前端的节点备用空间不足, 符合条件就配置一个新的map(配置更大的, 拷贝原来的,
// 的, 释放原来的)
void reserve_map_at_front (size_type nodes_to_add = 1) {
    if (nodes_to_add > start.node - map)
        reallocate_map(nodes_to_add, true);
}

```

deque 的插入元素和删除元素

因为 deque 的是能够双向操作, 所以其 push 和 pop 操作都类似于 list 都可以直接有对应的操作, 需要注意的是 list 是链表, 并不会涉及到界线的判断, 而deque 是由数组来存储的, 就需要随时对界线进行判断。

push 实现

```

template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {

    ...

public:                                // push_* and pop_*
    // 对尾进行插入
    // 判断函数是否达到了数组尾部. 没有达到就直接进行插入
    void push_back(const value_type& t) {
        if (finish.cur != finish.last - 1) {
            construct(finish.cur, t);
            ++finish.cur;
        }
        else
            push_back_aux(t);
    }

    // 对头进行插入

```

```
// 判断函数是否达到了数组头部. 没有达到就直接进行插入
void push_front(const value_type& t) {
    if (start.cur != start.first) {
        construct(start.cur - 1, t);
        --start.cur;
    }
    else
        push_front_aux(t);
}
...
};

};
```

pop 实现

```
template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {

    ...
public:
    // 对尾部进行操作
    // 判断是否达到数组的头部. 没有到达就直接释放
    void pop_back() {
        if (finish.cur != finish.first) {
            --finish.cur;
            destroy(finish.cur);
        }
        else
            pop_back_aux();
    }

    // 对头部进行操作
    // 判断是否达到数组的尾部. 没有到达就直接释放
    void pop_front() {
        if (start.cur != start.last - 1) {
            destroy(start.cur);
            ++start.cur;
        }
        else
    }
```

```
    pop_front_aux();
}
...
};
```

reserve_map_at一类函数. pop和push都先调用了reserve_map_at_XX函数, 这些函数主要是为了判断前后空间是否足够.

删除操作

不知道还记得, 最开始构造函数调用 create_map_and_nodes 函数, 考虑到 deque 实现前后插入时间复杂度为O(1), 保证了在前后留出了空间, 所以 push 和 pop 都可以在前面的数组进行操作。

现在就来看 erase, 因为 deque 是由数组构成, 所以地址空间是连续的, 删除也就像 vector 一样, 要移动所有的元素。

deque 为了保证效率尽可能的高, 就判断删除的位置是中间偏后还是中间偏前来进行移动。

```
template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
    ...
public:           // Erase
    iterator erase(iterator pos)
    {
        iterator next = pos;
        ++next;
        difference_type index = pos - start;
        // 删除的地方是中间偏前, 移动前面的元素
        if (index < (size() >> 1))
        {
            copy_backward(start, pos, next);
            pop_front();
        }
        // 删除的地方是中间偏后, 移动后面的元素
    else {
```

```
        copy(next, finish, pos);
        pop_back();
    }
    return start + index;
}

// 范围删除，实际也是调用上面的erase函数。
iterator erase(iterator first, iterator last);
void clear();
...
};
```

最后讲一下 insert 函数

deque 源码的基本每一个insert 重载函数都会调用了 insert_auto 判断插入的位置离头还是尾比较近。

如果离头进：则先将头往前移动，调整将要移动的距离，用 copy 进行调整。

如果离尾近：则将尾往前移动，调整将要移动的距离，用 copy 进行调整。

注意：push_back 是先执行构造在移动 node, 而 push_front 是先移动 node 在进行构造. 实现的差异主要是 finish 是指向最后一个元素的后一个地址而 first 指向的就只第一个元素的地址. 下面 pop 也是一样的。

源码里还有一些其它的成员函数，限于篇幅，这里就不贴源码，简单的过一遍 还有一些函数：

realloc_map: 判断中控器的容量是否够用，如果不够用，申请更大的空间，拷贝元素过去，修改 map 和 start, finish 的指向。

fill_initialize 函数: 申请空间，对每个空间进行初始化，最后一个数组单独处理. 毕竟最后一个数组一般不是会全部填充满。

clear函数. 删除所有元素. 分两步执行:

首先从第二个数组开始到倒数第二个数组一次性全部删除，这样做是考虑到中间的数组肯定都是满的，前后两个数组就不一定是填充满的，最后删除前后两个数组的元素。

deque的swap操作也只是交换了 start, finish, map, 并没有交换所有的元素.

resize函数. 重新将deque进行调整, 实现与list一样的.

析构函数: 分步释放内存.

deque 总结

deque 其实是在功能上合并了 vector 和 list。

优点:

- 1、随机访问方便, 即支持 [] 操作符和 vector.at();
- 2、在内部方便的进行插入和删除操作;
- 3、可在两端进行 push、pop

缺点: 因为涉及比较复杂, 采用分段连续空间, 所以占用内存相对多。

使用区别:

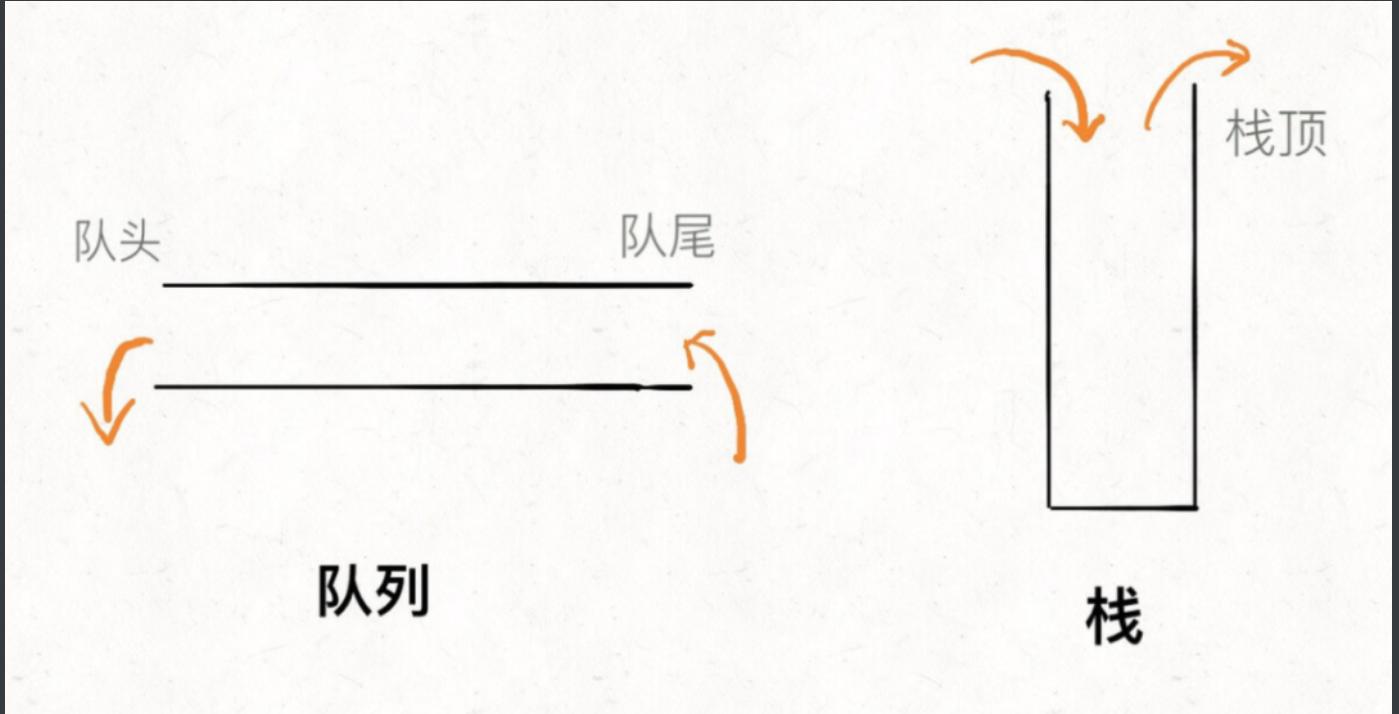
- 1、如果你需要高效的随即存取, 而不在乎插入和删除的效率, 使用 vector。
- 2、如果你需要大量的插入和删除, 而不关心随机存取, 则应使用 list。
- 3、如果你需要随机存取, 而且关心两端数据的插入和删除, 则应使用 deque。

15.6 以 deque 为底层容器的适配器

最后要介绍的三种常用的数据结构, 准确来说其实是一种适配器, 底层都是已其它容器为基准。

栈-stack: 先入后出, 只允许在栈顶添加和删除元素, 称为出栈和入栈。

队列-queue: 先入先出, 在队首取元素, 在队尾添加元素, 称为出队和入队。



优先队列-priority_queue：带权值的队列。

常见栈的应用场景包括括号问题的求解，表达式的转换和求值，函数调用和递归实现，深度优先遍历DFS等；

常见的队列的应用场景包括计算机系统中各种资源的管理，消息缓冲队列的管理和广度优先遍历BFS等。

源码之前，了无秘密，翻一下源码，就知道 stack 和 queue 的底层其实就是使用 deque，用 deque 为底层容器封装。

stack 的源码：

```
#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class T, class Sequence = deque<T> >
#else
template <class T, class Sequence>
#endif
class stack {
public:
    typedef typename Sequence::value_type value_type;
    typedef typename Sequence::size_type size_type;
    typedef typename Sequence::reference reference;
    typedef typename Sequence::const_reference const_reference;
protected:
    Sequence c;
```

queue 的源码：

```
#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class T, class Sequence = deque<T> >
#else
template <class T, class Sequence>
#endif
class queue {
public:
    typedef typename Sequence::value_type value_type;
    typedef typename Sequence::size_type size_type;
    typedef typename Sequence::reference reference;
    typedef typename Sequence::const_reference const_reference;
protected:
    Sequence c;
```

heap

最后我们来看一下， heap ， heap 并不是一个容器，所以他没有实现自己的迭代器，也就没有遍历操作，它只是一种算法。

push_heap 插入元素

插入函数是push_heap. heap只接受RandomAccessIterator类型的迭代器.

```
template <class RandomAccessIterator>
inline void push_heap(RandomAccessIterator first, RandomAccessIterator last) {
    __push_heap_aux(first, last, distance_type(first), value_type(first));
}

template <class RandomAccessIterator, class Distance, class T>
inline void __push_heap_aux(RandomAccessIterator first,
RandomAccessIterator last, Distance*, T*) {
    // 这里传入的是两个迭代器的长度, 0, 还有最后一个数据
    __push_heap(first, Distance((last - first) - 1), Distance(0), T(*(last - 1)));
}
```

pop_heap 删除元素

pop操作其实并没有真正意义去删除数据, 而是将数据放在最后, 只是没有指向最后的元素而已, 这里array也可以使用, 毕竟没有对数组的大小进行调整. pop的实现有两种, 这里都罗列了出来, 另一个传入的是 cmp 伪函数.

```
template <class RandomAccessIterator, class Compare>
inline void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                     Compare comp) {
    __pop_heap_aux(first, last, value_type(first), comp);
}

template <class RandomAccessIterator, class T, class Compare>
inline void __pop_heap_aux(RandomAccessIterator first,
                           RandomAccessIterator last, T*, Compare comp)
{
    __pop_heap(first, last - 1, last - 1, T(*(last - 1)), comp,
```

```

        distance_type(first));
}

template <class RandomAccessIterator, class T, class Compare, class
Distance>
inline void __pop_heap(RandomAccessIterator first, RandomAccessIterator
last,
                        RandomAccessIterator result, T value, Compare
comp,
                        Distance*) {
    *result = *first;
    __adjust_heap(first, Distance(0), Distance(last - first), value,
comp);
}

template <class RandomAccessIterator, class T, class Distance>
inline void __pop_heap(RandomAccessIterator first, RandomAccessIterator
last,
                        RandomAccessIterator result, T value, Distance*)
{
    *result = *first; // 因为这里是大根堆，所以first的值就是最大值，先将最大值保
存.
    __adjust_heap(first, Distance(0), Distance(last - first), value);
}

```

make_heap 将数组变成堆存放

```

template <class RandomAccessIterator>
inline void make_heap(RandomAccessIterator first, RandomAccessIterator
last) {
    __make_heap(first, last, value_type(first), distance_type(first));
}

template <class RandomAccessIterator, class T, class Distance>
void __make_heap(RandomAccessIterator first, RandomAccessIterator last,
T*,
                Distance*) {
    if (last - first < 2) return;
    // 计算长度，并找出中间的根值

```

```

        Distance len = last - first;
        Distance parent = (len - 2)/2;

        while (true) {
            // 一个个进行调整，放到后面
            __adjust_heap(first, parent, len, T(*(first + parent)));
            if (parent == 0) return;
            parent--;
        }
    }
}

```

sort_heap 实现堆排序

其实就是每次将第一位数据弹出从而实现排序功能.

```

template <class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last) {
    while (last - first > 1) pop_heap(first, last--);
}

template <class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp) {
    while (last - first > 1) pop_heap(first, last--, comp);
}

```

priority_queue

最后我们来看一下 priority_queue

上一节分析 heap 其实就是为 priority_queue 做准备. priority_queue 是一个优先级队列, 是带权值的. 支持插入和删除操作, 其只能从尾部插入, 头部删除, 并且其顺序也并非是根据加入的顺序排列的。

priority_queue 因为也是队列的一种体现, 所以也就跟队列一样不能直接的遍历数组, 也就没有迭代器. priority_queue 本身也不算是一个容器, 它是以 vector 为容器以 heap 为数据操作的配置器。

类型定义

```
#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class T, class Sequence = vector<T>,
          class Compare = less<typename Sequence::value_type> >
#else
template <class T, class Sequence, class Compare>
#endif
class priority_queue {
public:
    // 符合traits编程规范
    typedef typename Sequence::value_type value_type;
    typedef typename Sequence::size_type size_type;
    typedef typename Sequence::reference reference;
    typedef typename Sequence::const_reference const_reference;
protected:
    Sequence c; // 定义vector容器的对象
    Compare comp; // 定义比较函数(伪函数)
    ...
};

};
```

属性获取

priority_queue 只有简单的 3 个属性获取的函数, 其本身的操作也很简单, 只是实现依赖了 vector 和 heap 就变得比较复杂。

```
class priority_queue {
    ...
public:
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    const_reference top() const { return c.front(); }
    ...
};
```

push 和 pop 实现

push 和 pop 具体都是采用的 heap 算法。

priority_queue 本身实现是很复杂的，但是当我们已经了解过 vector, heap 之后再来看，它其实就简单了。

就是将 vector 作为容器， heap 作为算法来操作的配置器，这也体现了 STL 的灵活性：通过各个容器与算法的结合就能实现另一种功能。

最后，来自实践生产环境的一个体会：上面所列的所有容器的一个原则：为了避免拷贝开销，不要直接把大的对象直接往里塞，而是使用指针。

好了，本期的内容就到这里了，我们下期再见。

PS：看有多少人点赞，下期不定期更新关联式容器哦，先买个关子，下期有个硬核的内容带大家手撕红黑树源码，红黑树的应用可以说很广了，像 Java 集合中的 TreeSet 和 TreeMap、STL 中的 set 和 map、Linux 虚拟内存的管理都用到了哦。

参考

- 1、《STL 源码剖析》
- 2、<https://github.com/FunctionDou/STL>

十六、2 万字 10 图带你手撕 STL 关联式容器源码

大家好，我是小贺。

文章每周持续更新，可以微信搜索公众号「herongwei」第一时间阅读和催更。

本文 GitHub : <https://github.com/rongweihe/CPPNotes> 已经收录，有一线大厂面试点思维导图，也整理了很多我的文档，欢迎点个小★和完善。一起加油，变得更好！

鸽了好久的 STL 源码系列，这周开始更新，还剩最后两篇，分别是关联式容器和 STL 基本算法。

距离上篇源码剖析的文章好像在几个月前？

咕咕咕，连我自己都看不下去了，怎么能这么懒呢？正好趁着这几天休假，一鼓作气的把该写的文章补上吧。



16.1 前言

STL 源码剖析系列已经出了三篇：

[5千字长文+30张图解 | 陪你手撕 STL 空间配置器源码](#)

[万字长文炸裂！手撕 STL 迭代器源码与 traits 编程技法](#)

[超硬核 | 2万字+20图带你手撕 STL 序列式容器源码](#)

上一篇，我们剖析了序列式容器，这一篇我们来学习下关联式容器。

在 STL 编程中，容器是我们经常会用到的一种数据结构，容器分为序列式容器和关联式容器。

两者的本质区别在于：序列式容器是通过元素在容器中的位置顺序存储和访问元素，而关联容器则是通过键 (key) 存储和读取元素。

本篇着重剖析关联式容器相关背后的知识点，来一张思维导图。



16.2 容器分类

前面提到了，根据元素存储方式的不同，容器可分为序列式和关联式，那具体的又有哪些分类呢，这里我画了一张图来看一下。

序列式容器

关联式容器

array(build-in)

vector

list

slist

deque

heap

以算法形式呈现

priority_queue

非标准

stack

配接器

queue

RB-tree

set

map

multiset

multimap

hashtable

unordered_set

unordered_map

unordered_multiset

unordered_multimap

关联式容器比序列式容器更好理解，从底层实现来分的话，可以分为 RB_tree 还是 hash_table，所有暴露给用户使用的关联式容器都绕不过底层这两种实现。

不多 BB。我们先来分析其底层的两种实现，后面在逐个一一剖析其外在形式，这样对于新手还是老手，对于其背后核心的设计和奥秘，理解起来都会丝滑顺畅。



16.3 RB-tree 介绍与应用

首先来介绍红黑树，RB Tree 全称是 Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红 (Red) 或黑 (Black)。

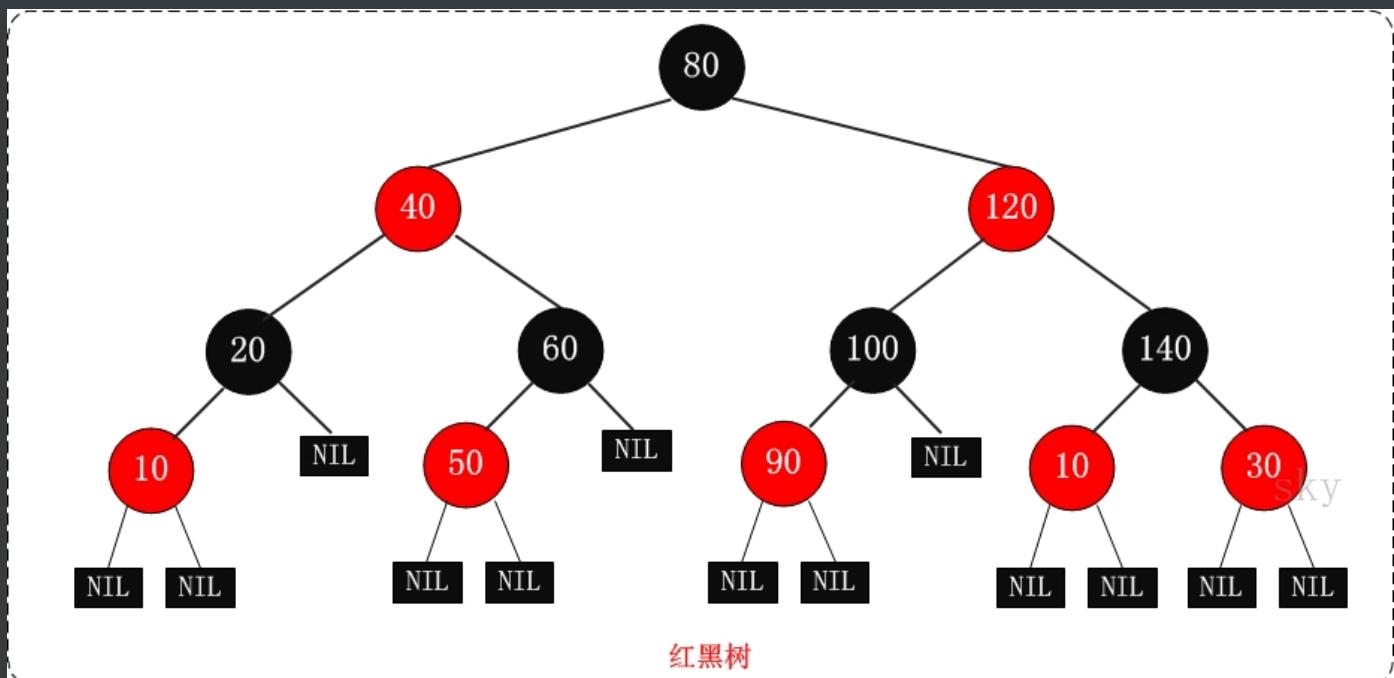
红黑树的特性：

- 每个节点或者是黑色，或者是红色。
- 根节点是黑色。
- 每个叶子节点 (NIL) 是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]
- 如果一个节点是红色的，则它的子节点必须是黑色的。
- 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

注意：

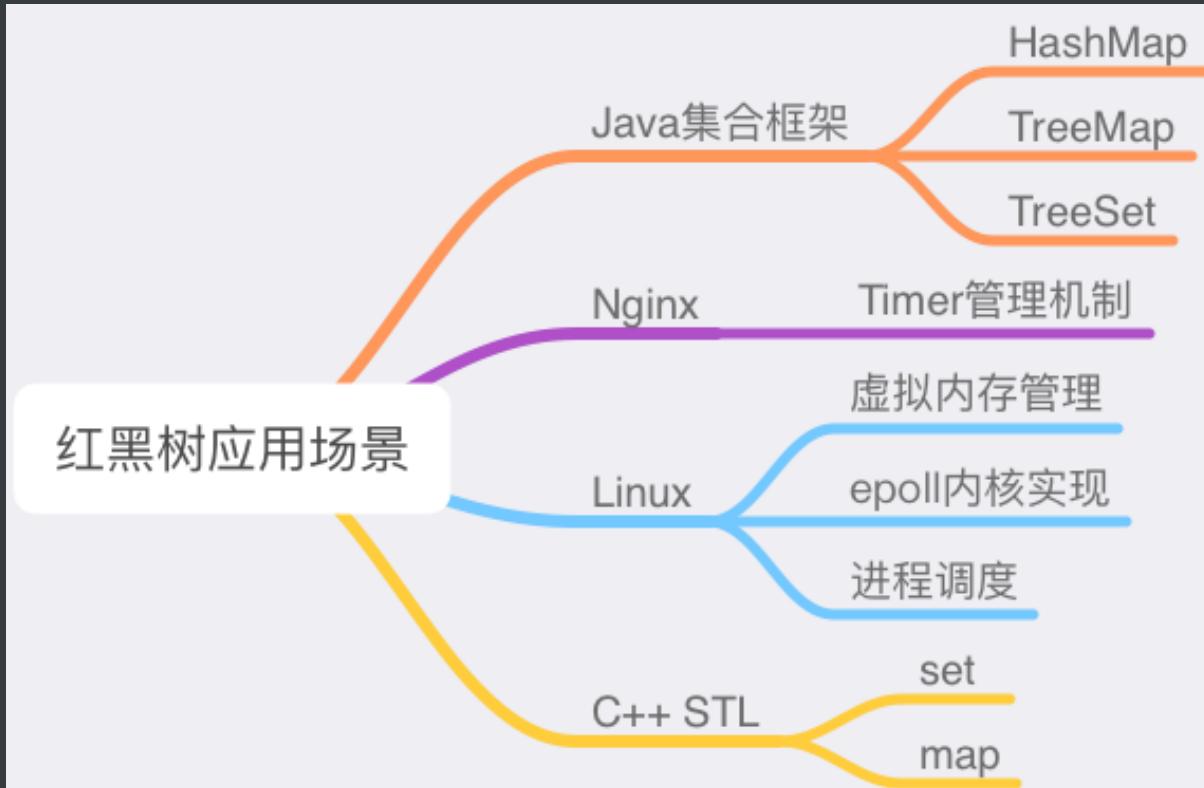
- 特性 (3)中的叶子节点，是只为空(NIL或null)的节点。
- 特性 (5)确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近衡的二叉树。

红黑树示意图如下：



红黑树保证了最坏情形下在 $O(\log n)$ 时间复杂度内完成查找、插入及删除操作；效率非常高。

因此红黑树可用于很多场景，比如下图。



好了，红黑树介绍到这里差不多了，关于红黑树的分析在深入又是另一篇文章了，下面我们在简单介绍一下红黑树的两种数据操作方式。

16.4 RB-tree 的基本操作

红黑树的基本操作包括 **添加、删除**。

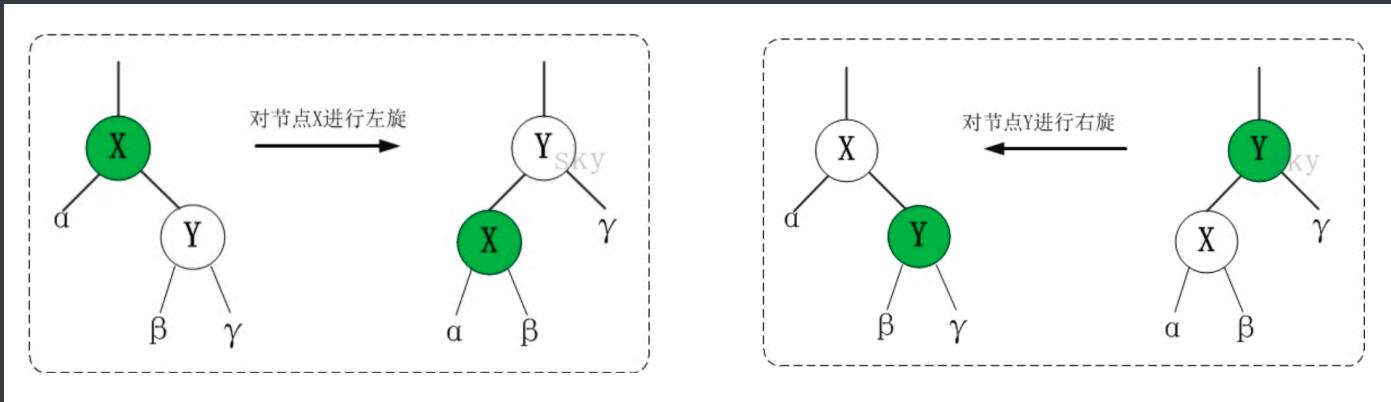
在对红黑树进行添加或删除之后，都会用到旋转方法。原因在于添加或删除红黑树中的节点之后，红黑树就发生了变化，可能不满足红黑树的 5 条性质，也就说不再是一颗红黑树了，而是一颗普通的树。

而通过旋转，可以使这颗树重新成为红黑树。简单点说，旋转的目的是让树保持红黑树的特性。

在红黑树里的旋转包括两种：左旋和右旋。

左旋： 对节点 X 进行左旋，也就说让节点 X 成为左节点。

右旋： 对节点 X 进行右旋，也就说让节点 X 成为右节点。



说完了旋转，我们再来看一下它的插入，有两种插入方式：

```
//不允许键值重复插入
pair<iterator, bool> insert_unique(const value_type& x);

//允许键值重复插入
iterator insert_equal(const value_type& x);
```

RB-tree 里面分两种插入方式，一种是允许键值重复插入，一种不允许。可以简单的理解，如果调用 `insert_unique` 插入重复的元素，在 RB-tree 里面其实是无效的。

其实在 RB-tree 源码里面，上面两个函数走到最底层，调用的是同一个 `_insert()` 函数。

知道了数据的操作方式，我们再来看 RB-tree 的构造方式：内部调用 `rb_tree_node_allocator`，每次恰恰配置一个节点，会调用 `simple_alloc` 空间配置器来配置节点。

并且分别调用四个节点函数来进行初始化和构造化。

```
get_node(), put_node(), create_node(), clone_node(), destroy_node();
```

RB-tree 的构造方式也有两种：一种是以现有的 RB-tree 复制一个新的 RB-tree，另一种是产生一棵空的树。

16.5 哈希表 (hashtable) 介绍和应用

红黑树的介绍就到哪里了，下面我们来看一下哈希表。

我们知道数组的特点是：寻址容易，插入和删除困难；而链表的特点是：寻址困难，插入和删除容易。

那么我们能不能综合两者的特性，做出一种寻址容易，插入删除也容易的数据结构？

答案是肯定的，这就是哈希表。

哈希表，也被称为散列表，是一种常用的数据结构，这种结构在插入、删除、查找等操作上也具有“常数平均时间”的表现。

也可以视为一种字典结构。

在讲具体的 hashtable 源码之前，我们先来认识两个概念：

- 散列函数：使用某种映射函数，将大数映射为小数。负责将某一个元素映射为一个“大小可接受内的索引”，这样的函数称为 hash function（散列函数）。
- 使用散列函数可能会带来问题：可能会有不同的元素被映射到相同的位置，这无法避免，因为元素个数有可能大于分配的 array 容量，这就是所谓的碰撞问题，解决碰撞问题一般有：线性探测、二次探测、开链等。

不同的方法有不同的效率差别，本文以 SGI STL 源码里采用的开链法来进行 hashtable 的学习。

拉链法，可以理解为“链表的数组”，其思路是：如果多个关键字映射到了哈希表的同一个位置处，则将这些关键字记录在同一个线性链表中，如果有重复的，就顺序拉在这条链表的后面。

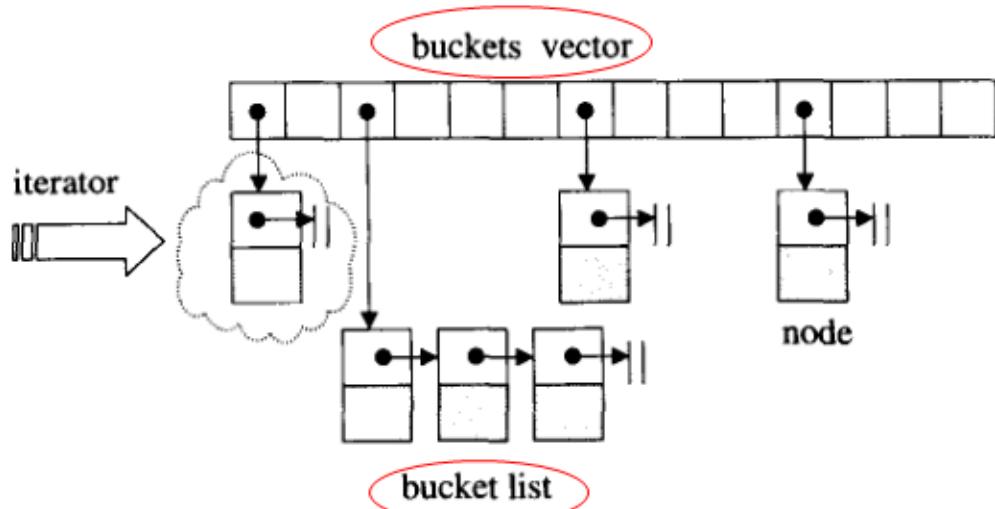
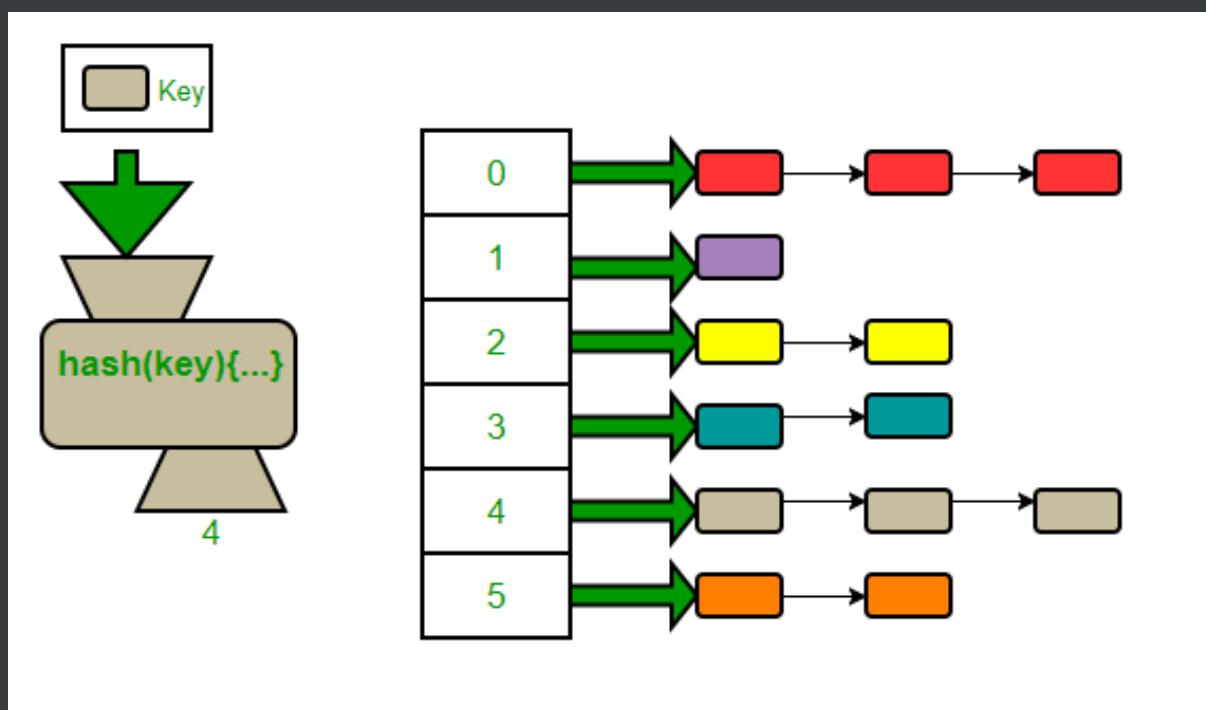


图 5-24 以开链 (separate chaining) 法完成的 hash table。SGI 即采此法。



注意，bucket 维护的链表，并不采用 STL 的 list，而是自己维护的 hash table node，至于 buckets 表格，则是以 vector 构造完成，以便具有动态扩充能力。

hash table 的定义：

```
//模板参数定义
/*
Value: 节点的实值类型
Key:    节点的键值类型
```

```

HashFcn: hash function的类型
ExtractKey: 从节点中取出键值的方法 (函数或仿函数)
EqualKey: 判断键值是否相同的方法 (函数或仿函数)
Alloc: 空间配置器
*/
//hash table的线性表是用 vector 容器维护
template <class _Val, class _Key, class _HashFcn,
           class _ExtractKey, class _EqualKey, class _Alloc>
class hashtable {
public:
    typedef _Key key_type;
    typedef _Val value_type;
    typedef _HashFcn hasher;
    typedef _EqualKey key_equal;

    typedef size_t           size_type;
    typedef ptrdiff_t        difference_type;
    typedef value_type*      pointer;
    typedef const value_type* const_pointer;
    typedef value_type&      reference;
    typedef const value_type& const_reference;

    hasher hash_funct() const { return _M_hash; }
    key_equal key_eq() const { return _M_equals; }

private:
    typedef _Hashtable_node<_Val> _Node;

```

这里需要注意的是， hashtable 的迭代器是正向迭代器，且必须维持这整个 buckets vector 的关系，并记录目前所指的节点。其前进操作是目前所指的节点，前进一个位置。

```

//以下是hash table的成员变量
private:
    hasher           _M_hash;
    key_equal        _M_equals;
    _ExtractKey      _M_get_key;

```

```

vector<_Node*,_Alloc> _M_buckets;//用vector维护buckets
size_type           _M_num_elements;//hashtable中list节点个数

public:
    typedef
    _Hashtable_iterator<_Val,_Key,_HashFcn,_ExtractKey,_EqualKey,_Alloc>
    iterator;
    typedef
    _Hashtable_const_iterator<_Val,_Key,_HashFcn,_ExtractKey,_EqualKey,
                           _Alloc>
    const_iterator;

public:
    //构造函数
    hashtable(size_type __n,
              const _HashFcn&      __hf,
              const _EqualKey&     __eql,
              const _ExtractKey&   __ext,
              const allocator_type& __a = allocator_type())
        : __HASH_ALLOC_INIT(__a)
        _M_hash(__hf),
        _M_equals(__eql),
        _M_get_key(__ext),
        _M_buckets(__a),
        _M_num_elements(0)
    {
        _M_initialize_buckets(__n);//预留空间，并将其初始化为空
        //预留空间大小为大于n的最小素数
    }

```

提供两种插入元素的方法：insert_equal允许重复插入；insert_unique不允许重复插入。

```

//插入元素节点，不允许存在重复元素
pair<iterator, bool> insert_unique(const value_type& __obj) {
    //判断容量是否够用，否则就重新配置
    resize(_M_num_elements + 1);

```

```

//插入元素,不允许存在重复元素
    return insert_unique_noresize(__obj);
}

//插入元素节点,允许存在重复元素
iterator insert_equal(const value_type& __obj)
{//判断容量是否够用,否则就重新配置
    resize(_M_num_elements + 1);
//插入元素,允许存在重复元素
    return insert_equal_noresize(__obj);
}

```

16.6 hashtable 的基本操作

后面马上要介绍的关联容器 set、multiset、map 和 multimap 的底层机制都是基于 RB-Tree 红黑树，虽然能够实现在插入、删除和搜索操作能够达到对数平均时间，可是要求输入数据有足够的随机性。

而 hash table 不需要要求输入数据具有随机性，在插入、删除和搜索操作都能达到常数平均时间。

SGI 中实现 hash table 的方式，是在每个 buckets 表格元素中维护一个链表，然后在链表上执行元素的插入、搜寻、删除等操作，该表格中的每个元素被称为桶 (bucket)。

虽然开链法并不要求表格大小为质数，但 SGI STL 仍然已质数来设计表格大小，并且将 28 个质数计算好，以备随时访问。

```

// Note: assumes long is at least 32 bits.
// 注意：假设long至少为32-bits，可以根据自己需要修改
//定义28个素数用作hashtable的大小
enum { __stl_num_primes = 28 };

static const unsigned long __stl_prime_list[__stl_num_primes] = {
    53ul,           97ul,           193ul,          389ul,          769ul,
    1543ul,         3079ul,         6151ul,        12289ul,        24593ul,
    49157ul,        98317ul,        196613ul,      393241ul,      786433ul,
    1572869ul,      3145739ul,      6291469ul,     12582917ul,    25165843ul,

```

```

50331653ul, 100663319ul, 201326611ul, 402653189ul, 805306457ul,
1610612741ul, 3221225473ul, 4294967291ul
};

```

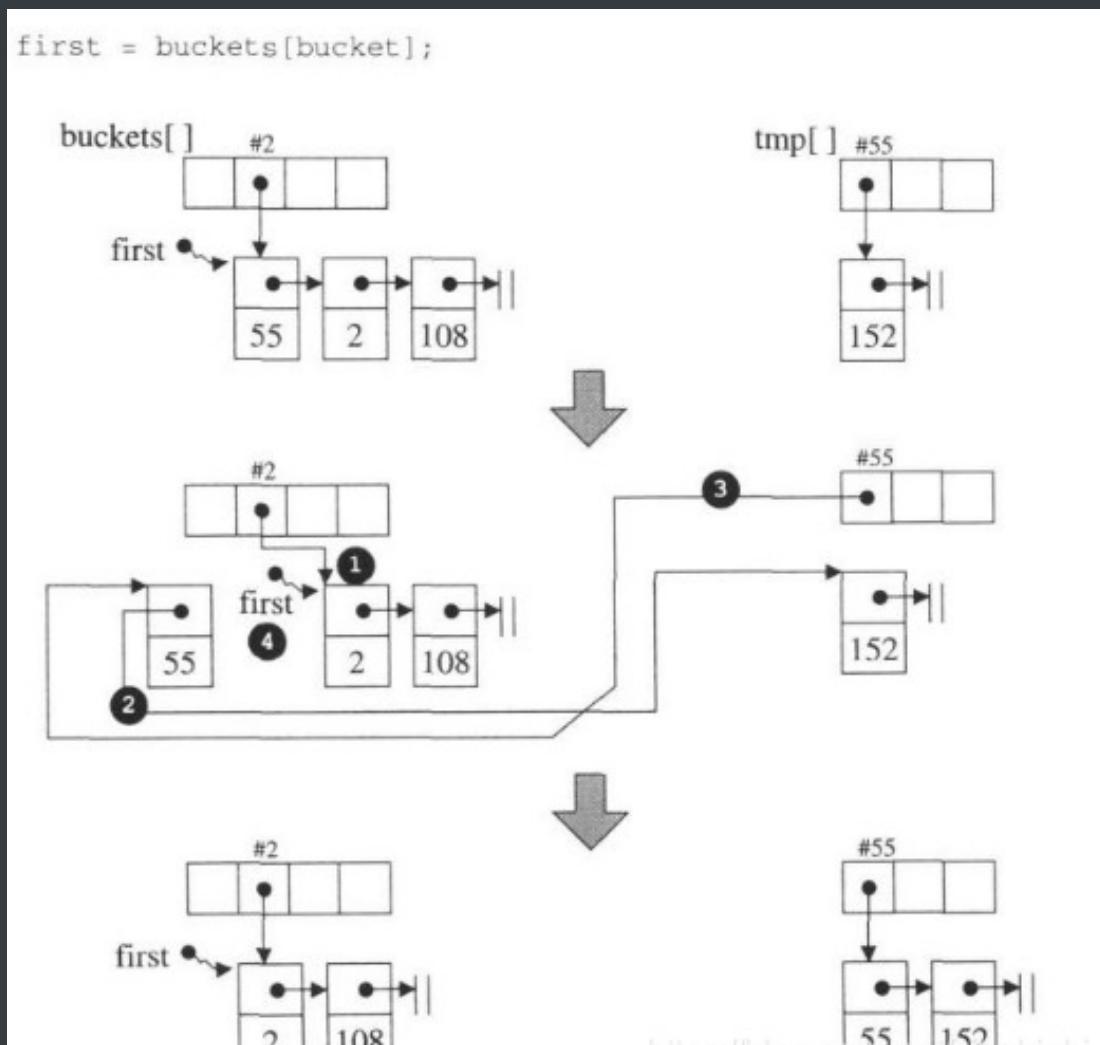
//返回大于n的最小素数

```

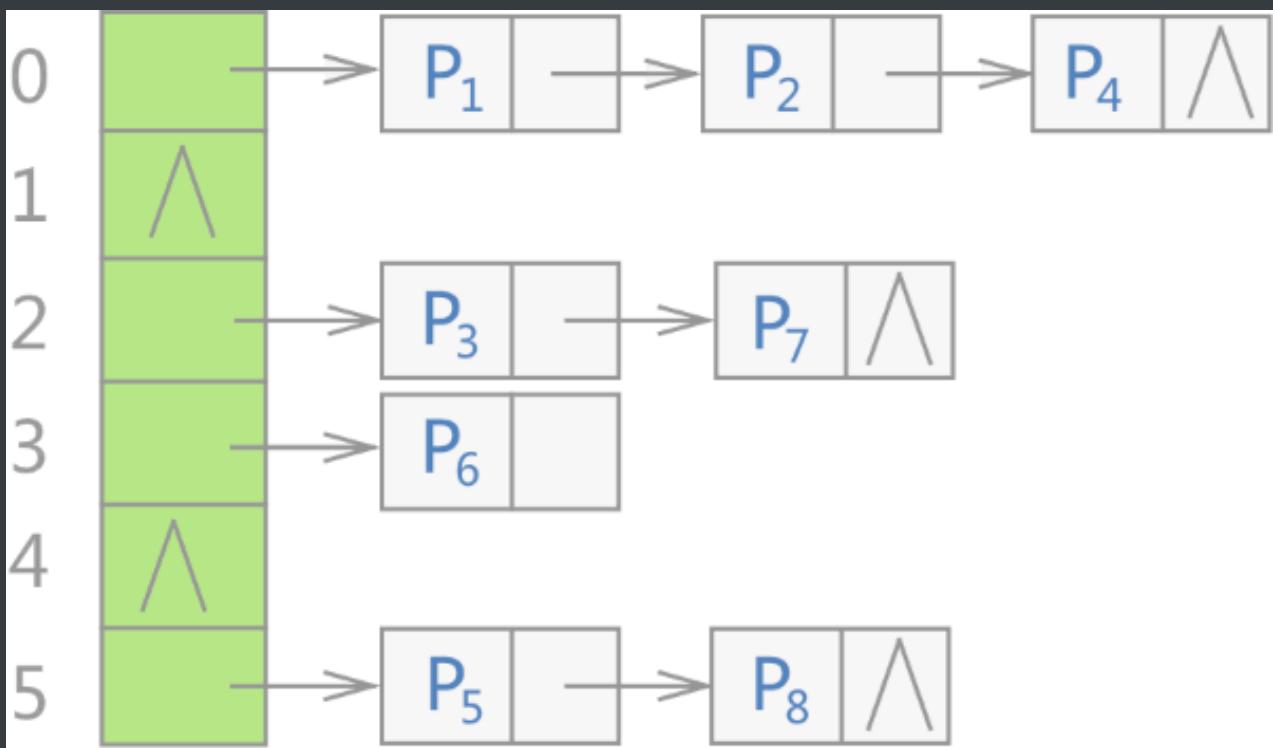
inline unsigned long __stl_next_prime(unsigned long __n) {
    const unsigned long* __first = __stl_prime_list;
    const unsigned long* __last = __stl_prime_list +
(int)__stl_num_primes;
    const unsigned long* pos = lower_bound(__first, __last, __n);

```

hashtable的节点配置和释放分别由 new_node 和 delete_node 来完成，并且插入操作和表格重整分别由 insert_unique 和 insert_equal ,resize 三个函数来完成。限于篇幅，这里用三张图来展示：



C++ STL 标准库中，不仅是 `unordered_xxx` 容器，所有无序容器的底层实现都采用的是哈希表存储结构。更准确地说，是用“链地址法”（又称“开链法”）解决数据存储位置发生冲突的哈希表，整个存储结构如图所示。



其中， P_i 表示存储的各个键值对。

最左边的绿色称之为 `bucket` 桶，可以看到，当使用无序容器存储键值对时，会先申请一整块连续的存储空间，但此空间并不用来直接存储键值对，而是存储各个链表的头指针，各键值对真正的存储位置是各个链表的节点。

在 C++ STL 标准库中，将图 1 中的各个链表称为桶（`bucket`），每个桶都有自己的编号（从 0 开始）。当有新键值对存储到无序容器中时，整个存储过程分为如下几步：

- 将该键值对中键的值带入设计好的哈希函数，会得到一个哈希值（一个整数，用 H 表示）；
- 将 H 和无序容器拥有桶的数量 n 做整除运算（即 $H \% n$ ），该结果即表示应将此键值对存储到的桶的编号；
- 建立一个新节点存储此键值对，同时将该节点链接到相应编号的桶上。

另外值得一提的是，哈希表存储结构还有一个重要的属性，称为负载因子（**load factor**）。

该属性同样适用于无序容器，用于衡量容器存储键值对的空/满程度，即负载因子越大，意味着容器越满，即各链表中挂载着越多的键值对，

这无疑会降低容器查找目标键值对的效率；反之，负载因子越小，容器肯定越空，但并不一定各个链表中挂载的键值对就越少。

举个例子，如果设计的哈希函数不合理，使得各个键值对的键带入该函数得到的哈希值始终相同（所有键值对始终存储在同一链表上）。这种情况下，即便增加桶数是的负载因子减小，该容器的查找效率依旧很差。

无序容器中，负载因子的计算方法为：

负载因子 = 容器存储的总键值对 / 桶数

默认情况下，无序容器的最大负载因子为 1.0。如果操作无序容器过程中，使得最大复杂因子超过了默认值，则容器会自动增加桶数，并重新进行哈希，以此来减小负载因子的值。

需要注意的是，此过程会导致容器迭代器失效，但指向单个键值对的引用或者指针仍然有效。

这也就解释了，为什么我们在操作无序容器过程中，键值对的存储顺序有时会“莫名”的发生变动。

C++ STL 标准库为了方便用户更好地管控无序容器底层使用的哈希表存储结构，各个无序容器的模板类中都提供表 所示的成员方法。

成员方法	功能
bucket_count()	返回当前容器底层存储键值对时，使用桶的数量
max_bucket_count()	返回当前系统中，unordered_xxx 容器底层最多可以使用多少个桶
bucket_size(n)	返回第 n 个桶中存储键值对的数量
bucket(key)	返回以 key 为键的键值对所在桶的编号
load_factor()	返回 unordered_map 容器中当前的负载因子
max_load_factor()	返回或者设置当前 unordered_map 容器的最大负载因子
rehash(n)	尝试重新调整桶的数量为等于或大于 n 的值。如果 n 大于当前容器使用的桶数，则该方法会是容器重新哈希，该容器新的桶数将等于或大于 n。反之，如果 n 的值小于当前容器使用的桶数，则调用此方法可能没有任何作用。
reserve(n)	将容器使用的桶数 (bucket_count() 方法的返回值) 设置为最适合存储 n 个元素的桶
hash_function()	返回当前容器使用的哈希函数对象

介绍到这里， hashtable 的源码的大观也差不多了，想深入研究源码等细节大家可以访问开头的 GitHub 链接。

下面开始讲解具体的关联式容器，这里的分类比较多，有的读者可能会有点分不清。

那么小贺也给大家总结了一句话：只要是前缀带了 **unordered** 的就是无序，后缀带了 **multi** 的就是允许键重复，插入采用 **insert_equal** 而不是 **insert_unique**。

16.7 set、multiset、unordered_set、unordered_multiset

有了前面的 RB_tree 做铺垫，下面来学习 set/multiset 和 map/multimap 就容易多了。

先来看一下 set 的性质

- set 以 RB-tree 作为其底层机制，所有元素都会根据元素的键值自动被排序。
- set 的元素就是键值，set 不允许两个元素有相同的键值。

- 不允许通过 set 的迭代器来改变 set 的元素值，因为 set 的元素值就是键值，更改了元素值就会影响其排列规则，如果任意更改元素值，会严重破坏 set 组织，因此在定义 set 的迭代器时被定义成了 RB-tree 的 const_iterator。
- 由于 set 不允许有两个相同的键值，所以插入时采用的是 RB-tree 的 insert_unique 方式
- 这里的类型的定义要注意一点，都是 const 类型，因为 set 的主键定义后就不能被修改了，所以这里都是以const类型。

下面来看一下 set 的源码

set 的主要实现大都是调用 RB-tree 的接口，这里的类型的定义要注意一点，都是 const 类型，因为 set 的主键定义后就不能被修改了，所以这里都是以 const 类型。

```
#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class Key, class Compare = less<Key>, class Alloc = alloc>
#else
template <class Key, class Compare, class Alloc = alloc>
#endif
class set {
public:
    // typedefs:
    typedef Key key_type;
    typedef Key value_type;
    typedef Compare key_compare;
    typedef Compare value_compare;
private:
    // —RB-tree为接口封装
    typedef rb_tree<key_type, value_type, identity<value_type>,
key_compare, Alloc> rep_type;
    rep_type t; // red-black tree representing set
public:
    // 定义的类型都是const类型，不能修改
    typedef typename rep_type::const_pointer pointer;
    typedef typename rep_type::const_pointer const_pointer;
    typedef typename rep_type::const_reference reference;
    typedef typename rep_type::const_reference const_reference;
    typedef typename rep_type::const_iterator iterator;
```

```
typedef typename rep_type::const_iterator const_iterator;
typedef typename rep_type::const_reverse_iterator reverse_iterator;
typedef typename rep_type::const_reverse_iterator
const_reverse_iterator;
typedef typename rep_type::size_type size_type;
typedef typename rep_type::difference_type difference_type;
...
};
```

构造函数构造成员的时候调用的是 RB-tree 的 insert_unique。

```
class set {
public:
    ...
set() : t(Compare{}) {}
explicit set(const Compare& comp) : t(comp) {} // 不能隐式转换

        // 接受两个迭代器
        // 构造函数构造成员的时候调用的是RB-tree的insert_unique
template <class InputIterator>
set(InputIterator first, InputIterator last)
    : t(Compare{}) { t.insert_unique(first, last); }
template <class InputIterator>
set(InputIterator first, InputIterator last, const Compare& comp)
    : t(comp) { t.insert_unique(first, last); }

set(const value_type* first, const value_type* last)
    : t(Compare{}) { t.insert_unique(first, last); }
set(const value_type* first, const value_type* last, const Compare&
comp)
    : t(comp) { t.insert_unique(first, last); }

set(const_iterator first, const_iterator last)
    : t(Compare{}) { t.insert_unique(first, last); }
set(const_iterator first, const_iterator last, const Compare& comp)
    : t(comp) { t.insert_unique(first, last); }
```

```
    ...
};
```

成员属性获取

```
class set {
public:
    ...
// 所有的操作都是通过调用RB-tree获取的
key_compare key_comp() const { return t.key_comp(); }
value_compare value_comp() const { return t.value_comp(); }
iterator begin() const { return t.begin(); }
iterator end() const { return t.end(); }
reverse_iterator rbegin() const { return t.rbegin(); }
reverse_iterator rend() const { return t.rend(); }
bool empty() const { return t.empty(); }
size_type size() const { return t.size(); }
size_type max_size() const { return t.max_size(); }
// 交换
void swap(set<Key, Compare, Alloc>& x) { t.swap(x.t); }
// 其他的find, count等都是直接调用的RB-tree的接口
iterator find(const key_type& x) const { return t.find(x); }
size_type count(const key_type& x) const { return t.count(x); }
iterator lower_bound(const key_type& x) const {
    return t.lower_bound(x);
}
iterator upper_bound(const key_type& x) const {
    return t.upper_bound(x);
}
pair<iterator,iterator> equal_range(const key_type& x) const {
    return t.equal_range(x);
}
...
};
```

insert 操作源码

```
class set {
public:
    ...
    // pair类型我们准备下一节分析，这里是直接调用insert_unique，返回插入成功就是
    // pair( , true)，插入失败则是( , false)
    typedef pair<iterator, bool> pair_iterator_bool;
    pair<iterator,bool> insert(const value_type& x) {
        pair<typename rep_type::iterator, bool> p = t.insert_unique(x);
        return pair<iterator, bool>(p.first, p.second);
    }
    // 指定位置的插入
    iterator insert(iterator position, const value_type& x) {
        typedef typename rep_type::iterator rep_iterator;
        return t.insert_unique((rep_iterator&)position, x);
    }
    // 可接受范围插入
    template <class InputIterator>
    void insert(InputIterator first, InputIterator last) {
        t.insert_unique(first, last);
    }
    ...
};
```

erase 的实现是通过调用 RB-tree 实现的 erase。

```
class set {
public:
    ...
    // erase的实现是通过调用RB-tree实现的erase
    void erase(iterator position) {
        typedef typename rep_type::iterator rep_iterator;
        t.erase((rep_iterator&)position);
    }
};
```

```
size_type erase(const key_type& x) {
    return t.erase(x);
}
void erase(iterator first, iterator last) {
    typedef typename rep_type::iterator rep_iterator;
    t.erase((rep_iterator&)first, (rep_iterator&)last);
}
void clear() { t.clear(); }
...
};
```

最后剩下最后一个重载运算符，也是以 RB-tree 为接口调用。

到这里，set 大部分的源码都已经过了一遍。

multiset 与 set 特性完全相同，唯一差别在于它允许键值重复，因此插入操作采用的是底层机制 RB-tree 的 insert_equal() 而非 insert_unique()。

接下来我们来了解一下两个新的数据结构：hash_set 与 unordered_set。

它们都属于基于哈希表(hash table)构建的数据结构，并且是关键字与键值相等的关联容器。

那 hash_set 与 unordered_set 哪个更好呢？实际上 **unordered_set** 在C++11的时候被引入标准库了，而 **hash_set** 并没有，所以建议还是使用 unordered_set 比较好，这就好比一个是官方认证的，一个是民间流传的。

在 SGI STL 源码剖析里，是以 hash_set 剖析的。

hash_set 将哈希表的接口在进行了一次封装，实现与 set 类似功能。

```
#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class Value, class HashFcn = hash<Value>,
          class EqualKey = equal_to<Value>,
          class Alloc = alloc>
#else
template <class Value, class HashFcn, class EqualKey, class Alloc =
          alloc>
```

```
#endif

class hash_set {
private:
    // 定义 hashtable
    typedef hashtable<Value, Value, HashFcn, identity<Value>, EqualKey,
Alloc> ht;
    ht rep;

public:
    typedef typename ht::key_type key_type;
    typedef typename ht::value_type value_type;
    typedef typename ht::hasher hasher;
    typedef typename ht::key_equal key_equal;

    // 定义为const类型，键值不允许修改
    typedef typename ht::size_type size_type;
    typedef typename ht::difference_type difference_type;
    typedef typename ht::const_pointer pointer;
    typedef typename ht::const_pointer const_pointer;
    typedef typename ht::const_reference reference;
    typedef typename ht::const_reference const_reference;

    // 定义迭代器
    typedef typename ht::const_iterator iterator;
    typedef typename ht::const_iterator const_iterator;
    // 仿函数
    hasher hash_funct() const { return rep.hash_funct(); }
    key_equal key_eq() const { return rep.key_eq(); }
    ...
};

};
```

构造函数

```
class hash_set
{
```

```
    ...
public:
    hash_set() : rep(100, hasher(), key_equal()) {} // 默认构造函数，表大小默认认为100最近的素数
    explicit hash_set(size_type n) : rep(n, hasher(), key_equal()) {}
    hash_set(size_type n, const hasher& hf) : rep(n, hf, key_equal()) {}
    hash_set(size_type n, const hasher& hf, const key_equal& eql)
        : rep(n, hf, eql) {}

#ifndef __STL_MEMBER_TEMPLATES
    template <class InputIterator>
    hash_set(InputIterator f, InputIterator l)
        : rep(100, hasher(), key_equal()) { rep.insert_unique(f, l); }
    template <class InputIterator>
    hash_set(InputIterator f, InputIterator l, size_type n)
        : rep(n, hasher(), key_equal()) { rep.insert_unique(f, l); }
    template <class InputIterator>
    hash_set(InputIterator f, InputIterator l, size_type n,
             const hasher& hf)
        : rep(n, hf, key_equal()) { rep.insert_unique(f, l); }
    template <class InputIterator>
    hash_set(InputIterator f, InputIterator l, size_type n,
             const hasher& hf, const key_equal& eql)
        : rep(n, hf, eql) { rep.insert_unique(f, l); }
    ...
};
```

插入删除等操作

insert调用的是insert_unqiue函数

```

class hash_set
{
    ...
public:
    // 都是调用hashtable的接口，这里insert_unique函数
    pair<iterator, bool> insert(const value_type& obj)
    {
        pair<typename ht::iterator, bool> p = rep.insert_unique(obj);
        return pair<iterator, bool>(p.first, p.second);
    }
}

```

set、multiset、unordered_set、unordered_multiset 总结

性质	set	multiset	unordered_set	unordered_multiset
底层实现	红黑树	红黑树	哈希表	哈希表
键值重复	不允许	允许	不允许	允许
插入元素	insert_unique	insert_equal	insert_unique	insert_equal
元素有序	有序	有序	无序	无序
是否支持[]运算符	不支持	不支持	不支持	不支持
迭代器性质	const_iterator	const_iterator	const_iterator	const_iterator

16.8 map、multimap、unordered_map、unordered_multimap

在分析 map 之前，我们来分析一下 pair 这种结构。

pair 是一个有两个变量的结构体，即谁都可以直接调用它的变量，毕竟 struct 默认权限都是 public，将两个变量用 pair 绑定在一起，这就为 map<T1, T2> 提供的存储的基础。

```

template <class T1, class T2> // 两个参数类型
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
}

```

```

// 定义的两个变量
T1 first;
T2 second;

// 构造函数
pair() : first(T1()), second(T2()) {}
pair(const T1& a, const T2& b) : first(a), second(b) {}

#ifndef __STL_MEMBER_TEMPLATES
template <class U1, class U2>
pair(const pair<U1, U2>& p) : first(p.first), second(p.second) {}
#endif
};


```

重载实现：

```

template <class T1, class T2>
inline bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y) {
    return x.first == y.first && x.second == y.second;
}

template <class T1, class T2>
inline bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y) {
    return x.first < y.first || (!(y.first < x.first) && x.second <
y.second);
}

```

整体 pair 的功能与实现都是很简单的，这都是为 map 的实现做准备的，接下来我们就来分析 map 的实现。

map 基本结构定义

```

#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class Key, class T, class Compare = less<Key>, class Alloc =
alloc>
#else

```

```

template <class Key, class T, class Compare, class Alloc = allocator>
#endif
class map {
public:
    typedef Key key_type; // 定义键值
    typedef T data_type; // 定义数据
    typedef T mapped_type;
    typedef pair<const Key, T> value_type; // 这里定义了map的数据类型为pair,
    // 且键值为const类型, 不能修改
    typedef Compare key_compare;

private:
    typedef rb_tree<key_type, value_type,
                    select1st<value_type>, key_compare, Alloc> rep_type;
    // 定义红黑树, map是以rb-tree结构为基础的
    rep_type t; // red-black tree representing map
public:
    ...

```

构造函数：map 所有插入操作都是调用的 RB-tree 的 insert_unique，不允许出现重复的键。

```

class map {
public:
    ...
public:
    // allocation/deallocation
    map() : t(Compare{}) {} // 默认构造函数
    explicit map(const Compare& comp) : t(comp) {}
#ifndef __STL_MEMBER_TEMPLATES
    // 接受两个迭代器
    template <class InputIterator>
    map(InputIterator first, InputIterator last)
        : t(Compare{}) { t.insert_unique(first, last); }
    template <class InputIterator>
    map(InputIterator first, InputIterator last, const Compare& comp)
        : t(comp) { t.insert_unique(first, last); }

```

...

基本属性的获取

```
class map {
public:
    ...
public:
    // 实际调用的是RB-tree的key_comp函数
    key_compare key_comp() const { return t.key_comp(); }
    // value_comp实际返回的是一个仿函数value_compare
    value_compare value_comp() const { return value_compare(t.key_comp()); }
}

// 以下的begin, end等操作都是调用的是RB-tree的接口
iterator begin() { return t.begin(); }
const_iterator begin() const { return t.begin(); }
iterator end() { return t.end(); }
const_iterator end() const { return t.end(); }
reverse_iterator rbegin() { return t.rbegin(); }
const_reverse_iterator rbegin() const { return t.rbegin(); }
reverse_iterator rend() { return t.rend(); }
const_reverse_iterator rend() const { return t.rend(); }
bool empty() const { return t.empty(); }
size_type size() const { return t.size(); }
size_type max_size() const { return t.max_size(); }

// 交换, 调用RB-tree的swap, 实际只交换head和count
void swap(map<Key, T, Compare, Alloc>& x) { t.swap(x.t); }

...
};

template <class Key, class T, class Compare, class Alloc>
inline void swap(map<Key, T, Compare, Alloc>& x,
                 map<Key, T, Compare, Alloc>& y) {
    x.swap(y);
}
```

重载的分析

```
class map {  
public:  
    ...  
public:  
    T& operator[](const key_type& k) {  
        return (*((insert(value_type(k, T()))).first)).second;  
    }  
    ...  
};
```

- insert(value_type(k, T0)) : 查找是否存在该键值, 如果存在则返回该pair, 不存在这重新构造一该键值并且值为空
- *((insert(value_type(k, T0))).first) : pair的第一个元素表示指向该元素的迭代器, 第二个元素指的是(false与true)是否存在, first 便是取出该迭代器而 * 取出pair.
- (*((insert(value_type(k, T0))).first)).second : 取出pair结构中的second保存的数据

这里有坑，初学者容易掉进去，请注意：

重载 operator[], 这一步返回是实值 value(即pair.second)的引用，假如原先没有定义 map 对象，即你访问的键值 key 不存在，则会自动新建一个 map 对象，键值 key 为你访问的键值 key，实值 value 为空，看下面的例子就明白了。

我在自己的开发机上测试，int 类型默认 value 为 0，bool 类型默认 value 为 false，string 类型默认是空。

```

_Tp& operator[](const key_type& __k) {
    iterator __i = lower_bound(__k);
    // __i->first is greater than or equivalent to __k.
    if (__i == end() || key_comp()(__k, (*__i).first))
        __i = insert(__i, value_type(__k, _Tp()));
    return (*__i).second;
    //其实简单的方式是直接返回
    //return (*((insert(value_type(k, T()))).first)).second;
}

```

map 的其他 insert, erase, find 都是直接调用 RB-tree 的接口函数实现的，这里就不直接做分析了。

16.9 map、multimap、unordered_map、unordered_multimap 总结

map 和 multimap 的共同点：

- 两者底层实现均为红黑树，不可以通过迭代器修改元素的键，但是可以修改元素的值；
- 拥有和 list 某些相同的特性，进行元素的新增和删除后，操作前的迭代器依然可用；

不同点：

- map 键不能重复，支持 [] 运算符；
- multimap 支持重复的键，不支持 [] 运算符；

map 并不像 set 一样将 iterator 设为 RB-tree 的 const_iterator，因为它允许用户通过其迭代器修改元素的实值。

map 和 unordered_map 共同点：

- 两者均不能有重复的键，均支持[]运算符

不同点：

- map 底层实现为红黑树

- unordered_map 底层实现为哈希表

unordered_map 是不允许存在相同的键存在，底层调用的 insert_unique() 插入元素
 unordered_multimap 可以允许存在多个相同的键，底层调用的 insert_equal() 插入元素

map 并不像 set 一样将 iterator 设为 RB-tree 的 const_iterator，因为它允许用户通过其迭代器修改元素的实值。

性质	map	multimap	unordered_map	unordered_multimap
底层实现	红黑树	红黑树	哈希表	哈希表
键值重复	不允许	允许	不允许	允许
插入元素	insert_unique	insert_equal	insert_unique	insert_equal
元素有序	有序	有序	无序	无序
是否支持[]运算符	支持	不支持	支持	不支持
迭代器性质	非 const_iterator	非 const_iterator	非 const_iterator	非 const_iterator
是否能修改元素值	不能修改key，可以修改value	不能修改key，可以修改value	不能修改key，可以修改value	不能修改key，可以修改value

16.10 思考

为什么 std::set 不支持[]运算符？

对于 std::map 而言，我们看一个例子：

```
std::map<std::string,int> m = { {"a",1}, {"b", 2} };
```

m["a"] 返回的是1所在单元的引用。

而如果对于 std::set std::string s = { "a", "b" }; 而言 s["a"] 应该是个什么类型呢？

我们用索引取一个容器的元素 $a[key] = value$ 的前提是既有 key 又有 value。
set 只有 key 没有 value，加了[]会导致歧义。

参考

- 1、《STL 源码剖析》
 - 2、<https://github.com/FunctionDou/STL>
-

十七、万字长文+ STL 算法总结

大家好，我是小贺。

文章每周持续更新，可以微信搜索公众号「herongwei」第一时间阅读和催更。

本文 GitHub：<https://github.com/rongweihe/CPPNotes> 已经收录，有一线大厂面试点思维导图，也整理了很多我的文档，欢迎点个小★和完善。一起加油，变得更好！

17.1 前言

上一篇更新了 STL 关联式容器源码，今天我们来学习下 STL 算法。

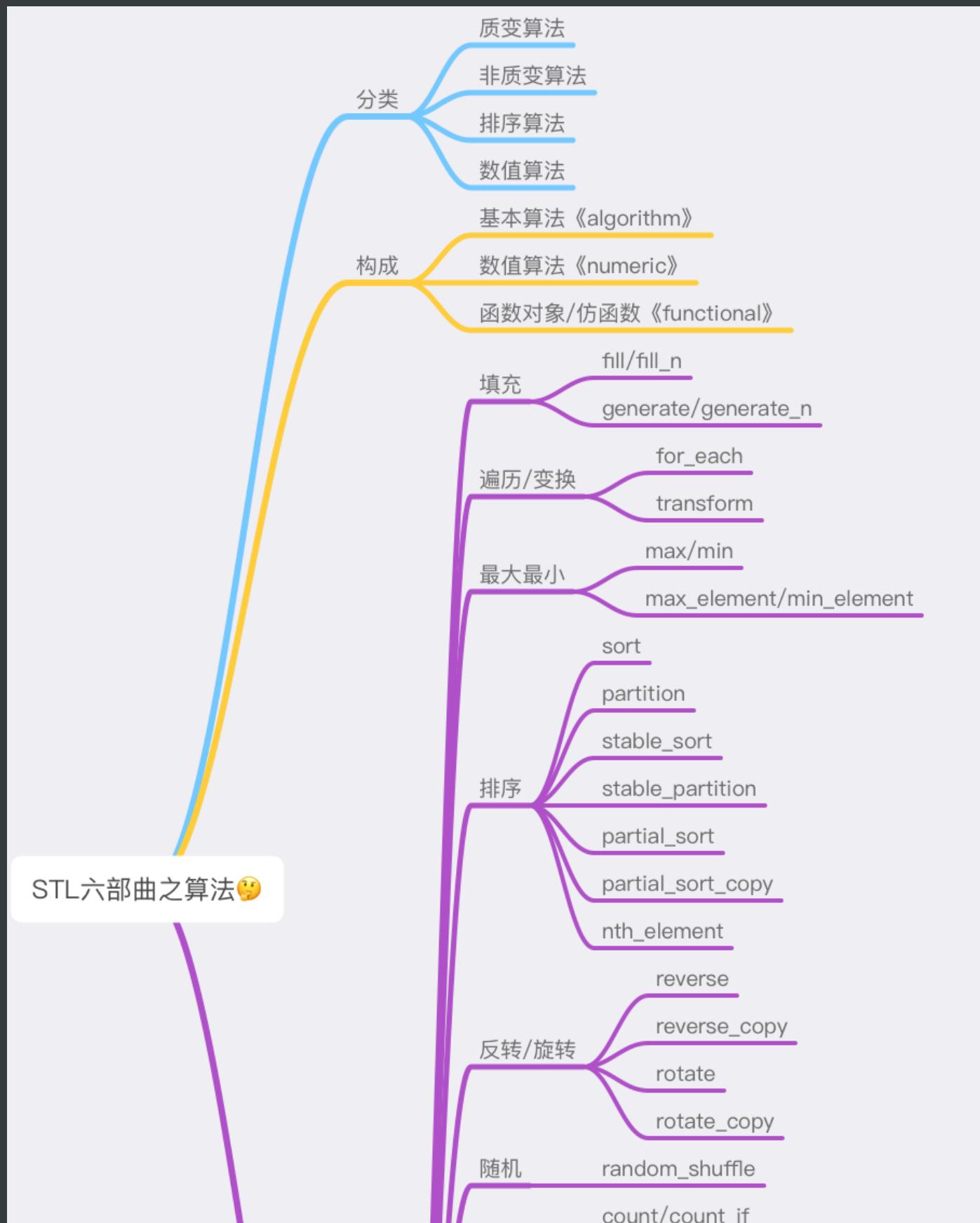
STL 算法博大精深，涵盖范围之广，其算法之大观，细节之深入，泛型思维之于字里行间，每每阅读都会有不同的收获。

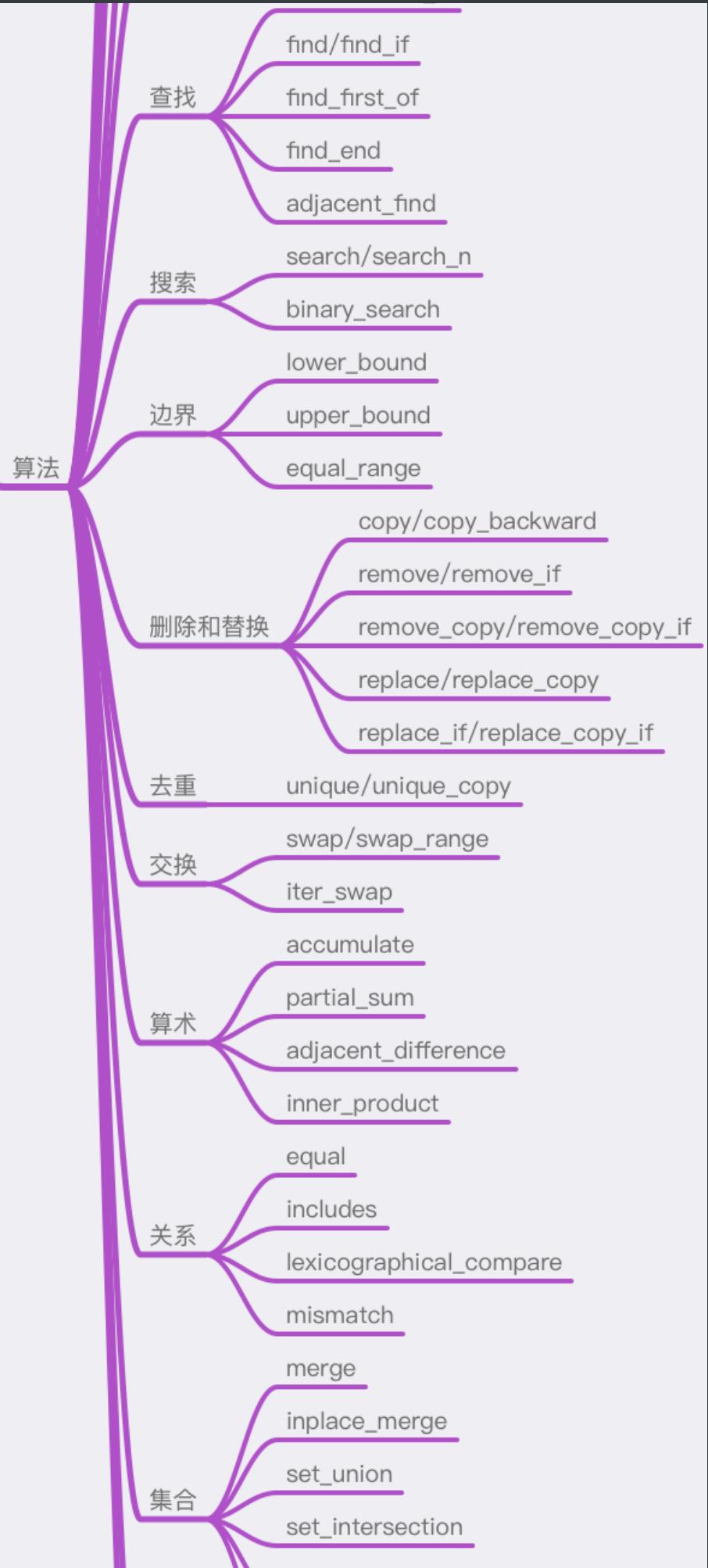
STL 将很多常见的逻辑都封装为现成的算法，熟悉这些算法的使用和实现很多时候可以大大简化编程。

并且在需要的时候能够对 STL 进行扩展，将自定义的容器和算法融入到 STL 中。

侯捷大师在书中说到：深入源码之前，先观察每一个算法的表现和大观，是一个比较好的学习方式。

不多 BB，先上思维导图：







17.2 回顾

STL 源码剖析系列：

[5千字长文+30张图解 | 带你手撕 STL 空间配置器源码](#)

[万字长文炸裂！手撕 STL 迭代器源码与 traits 编程技法](#)

[超硬核 | 2万字+20图带你手撕 STL 序列式容器源码](#)

[硬核来袭 | 2万字 + 10图带你手撕 STL 关联式容器源码](#)

17.3 基本算法

在 STL 标准规格中，并没有区分基本算法或复杂算法，然而 SGI 却把常用的一些算法定义于 `<stl_algobase.h>` 之中，其它算法定义于 `<stl_algo.h>` 中。

常见的基本算法有 `equal`、`fill`、`fill_n`、`iter_swap`、`lexicographical_compare`、`max`、`min`、`mismatch`、`swap`、`copy`、`copy_backward` 等。

17.4 质变算法和非质变算法

所有的 STL 算法归根到底，都可以分为两类。

所谓“质变算法”是指作用在由迭代器[first, last]所标示出来的区间，上运算过程中会更改区间内的元素内容：

比如拷贝(copy)、互换(swap)、替换(replace)、填写(fill)、删除(remove)、排列组合(permuation)、分割(partition)。随机重排(random shuffling)、排序(sort)等算法，都属于这一类。

而非质变算法是指在运算过程中不会更改区间内的元素内容。比如查找(find)，匹配(search)、计数(count)、遍历(for_each)、比较(equal_mismatch)、寻找极值(max,min)等算法。

17.5 输入参数

所有泛型算法的前两个参数都是一对迭代器，通过称为 first, last。用来标示算法的操作区间。

每一个 STL 算法的声明，都表现出它所需要的最低程度的迭代器类型。比如 find() 需要一个 inputiterator，这是它的最低要求，但同时也可接受更高类型的迭代器。

如 Forwarditerator、Bidirectionaliterator 或 RandomAccessIterator，因为，前者都可以看做是一个 inputiterator，而如果你给 find() 传入一个 Outputiterator，会导致错误。

将无效的迭代器传给某个算法，虽然是一种错误，但不能保证能够在编译器期间就被捕捉出来。因为所谓“迭代器类型”并不是真实的型别，它们只是 function template 的一种型别参数。

许多 STL 算法不仅支持一个版本，往往第一个版本算法会采用默认的行为，另一个版本会提供额外的参数，接受一个仿函数，以便采取其它的策略。

例如 unique() 默认情况下会使用 equality 操作符来比较两个相邻元素，但如果这些元素的型别并没有提供，那么便可以传递一个自定义的函数（或者叫仿函数）。

17.6 算法的泛型化

将一个表述完整的算法转化为程序代码，是一个合格程序员的基本功。

如何将算法独立于其所处理的数据结构之外，不受数据的牵绊，使得设计的算法在即将处理的未知的数据结构上（也许是 array，也许是 vector，也许是 list，也许是 deque）上，正确地实现所有操作呢？

这就需要进一步思考，关键在于只要把操作对象的型别加以抽象化，把操作对象的标示法和区间目标的移动行为抽象化，整个算法也就在一个抽象层面上工作了。

这个过程就叫做算法的泛型化，简称泛化。比如在 STL 源码剖析这本书里举了一个 `find` 的例子，如果一步步改成 `template + 迭代器` 的形式，来说明了泛化的含义。

下面我们就来看看 STL 那些牛批的算法，限于篇幅，算法的代码没有贴出。

具体源码细节可以去开头的 GitHub 仓库里研究，还有注释哦。

17.7 构成

头文件	功能
<code><algorithm></code>	算法函数
<code><numeric></code>	数值算法
<code><functional></code>	函数对象/仿函数

17.8 分类

No.	分类	说明	
1	非质变算法	Non-modifying sequence operations	不直接修改容器内容的算法
2	质变算法	Modifying sequence operations	可以修改容器内容的算法
3	排序算法	Sorting/Partitions/Binary search/	对序列排序、合并、搜索算法操作
4	数值算法	Merge/Heap/Min/max	对容器内容进行数值计算

17.9 填充

函数	作用
<code>fill(beg, end, val)</code>	将值 <code>val</code> 赋给 <code>[beg , end]</code> 范围内的所有元素
<code>fill_n(beg, n, val)</code>	将值 <code>val</code> 赋给 <code>[beg , beg+n]</code> 范围内的所有元素
<code>generate(beg, end, func)</code>	连续调用函数 <code>func</code> 填充 <code>[beg , end]</code> 范围内的所有元素
<code>generate_n(beg, n, func)</code>	连续调用函数 <code>func</code> 填充 <code>[beg , beg+n]</code> 范围内的所有元素

- `fill()` / `fill_n()` 用于填充相同值, `generate()` / `generate_n()` 用于填充不同值。

17.10 遍历/变换

函数	作用
<code>for_each(beg, end, func)</code>	将 <code>[beg , end]</code> 范围内所有元素依次调用函数 <code>func</code> , 返回 <code>func</code> 。不修改序列中的元素
<code>transform(beg, end, res, func)</code>	将 <code>[beg , end]</code> 范围内所有元素依次调用函数 <code>func</code> , 结果放入 <code>res</code> 中
<code>transform(beg2, end1, beg2, res, binary)</code>	将 <code>[beg , end]</code> 范围内所有元素与 <code>[beg2 , beg2+end-end]</code> 中所有元素依次调用函数 <code>binary</code> , 结果放入 <code>res</code> 中

17.11 最大最小

函数	作用
<code>max(a,b)</code>	返回两个元素中较大一个
<code>max(a,b,cmp)</code>	使用自定义比较操作 <code>cmp</code> , 返回两个元素中较大一个
<code>max_element(beg,end)</code>	返回一个 <code>ForwardIterator</code> , 指出[<code>beg</code> , <code>end</code>)中最大的元素
<code>max_element(beg,end,cmp)</code>	使用自定义比较操作 <code>cmp</code> , 返回一个 <code>ForwardIterator</code> , 指出[<code>beg</code> , <code>end</code>)中最大的元素
<code>min(a,b)</code>	返回两个元素中较小一个
<code>min(a,b,cmp)</code>	使用自定义比较操作 <code>cmp</code> , 返回两个元素中较小一个
<code>min_element(beg,end)</code>	返回一个 <code>ForwardIterator</code> , 指出[<code>beg</code> , <code>end</code>)中最小的元素
<code>min_element(beg,end,cmp)</code>	使用自定义比较操作 <code>cmp</code> , 返回一个 <code>ForwardIterator</code> , 指出[<code>beg</code> , <code>end</code>)中最小的元素

17.12 排序算法(12个): 提供元素排序策略

函数	作用
sort(beg, end)	默认升序重新排列元素
sort(beg, end, comp)	使用函数 comp 代替比较操作符执行 sort()
partition(beg, end, pred)	元素重新排序，使用 pred 函数，把结果为 true 的元素放在结果为 false 的元素之前
stable_sort(beg, end)	与 sort() 类似，保留相等元素之间的顺序关系
stable_sort(beg, end, pred)	使用函数 pred 代替比较操作符执行 stable_sort()
stable_partition(beg, end)	与 partition() 类似，保留容器中的相对顺序
stable_partition(beg, end, pred)	使用函数 pred 代替比较操作符执行 stable_partition()
partial_sort(beg, mid, end)	部分排序，被排序元素个数放到[beg,end)内
partial_sort(beg, mid, end, comp)	使用函数 comp 代替比较操作符执行 partial_sort()
partial_sort_copy(beg1, end1, beg2, end2)	与 partial_sort() 类似，只是将[beg1,end1)排序的序列复制到[beg2,end2)
partial_sort_copy(beg1, end1, beg2, end2, comp)	使用函数 comp 代替比较操作符执行 partial_sort_copy()
nth_element(beg, nth, end)	单个元素序列重新排序，使所有小于第 n 个元素的元素都出现在它前面，而大于它的都出现在后面
nth_element(beg, nth, end, comp)	使用函数 comp 代替比较操作符执行 nth_element()

17.13 反转/旋转

函数	作用
reverse(beg, end)	元素重新反序排序
reverse_copy(beg, end, res)	与 reverse() 类似, 结果写入 res
rotate(beg, mid, end)	元素移到容器末尾, 由 mid 成为容器第一个元素
rotate_copy(beg, mid, end, res)	与 rotate() 类似, 结果写入 res

17.14 随机

函数	作用
random_shuffle(beg, end)	元素随机调整次序
random_shuffle(beg, end, gen)	使用函数 gen 代替随机生成函数执行 random_shuffle()

17.15 查找算法(13个): 判断容器中是否包含某个值

统计

函数	作用
count(beg, end, val)	利用 == 操作符, 对[beg , end)的元素与 val 进行比较, 返回相等元素个数
count_if(beg, end, pred)	使用函数 pred 代替 == 操作符执行 count()

查找

函数	作用
<code>find(beg, end, val)</code>	利用 == 操作符，对[beg , end)的元素与 val 进行比较。当匹配时结束搜索，返回该元素的 InputIterator
<code>find_if(beg, end, pred)</code>	使用函数 pred 代替 == 操作符执行 find()
<code>find_first_of(beg1, end1, beg2, end2)</code>	在[beg1 , end1)范围内查找[beg2 , end2)中任意一个元素的第一次出现。返回该元素的 Iterator
<code>find_first_of(beg1, end1, beg2, end2, pred)</code>	使用函数 pred 代替 == 操作符执行 find_first_of() 。返回该元素的 Iterator
<code>find_end(beg1, end1, beg2, end2)</code>	在[beg1 , end1)范围内查找[beg2 , end2)最后一次出现。找到则返回最后一对的第一个 ForwardIterator ，否则返回 end1
<code>find_end(beg1, end1, beg2, end2, pred)</code>	使用函数 pred 代替 == 操作符执行 find_end() 。返回该元素的 Iterator
<code>adjacent_find(beg, end)</code>	对[beg , end)的元素，查找一对相邻重复元素，找到则返回指向这对元素的第一个元素的 ForwardIterator 。否则返回 end
<code>adjacent_find(beg, end, pred)</code>	使用函数 pred 代替 == 操作符执行 adjacent_find()

搜索

函数	作用
<code>search(beg1, end1, beg2, end2)</code>	在[<code>beg1</code> , <code>end1</code>)范围内查找[<code>beg2</code> , <code>end2</code>)首次出现, 返回一个 <code>ForwardIterator</code> , 查找成功, 返回[<code>beg1</code> , <code>end1</code>)内第一次出现[<code>beg2</code> , <code>end2</code>)的位置, 查找失败指向 <code>end1</code>
<code>search(beg1, end1, beg2, end2, pred)</code>	使用函数 <code>pred</code> 代替 <code>==</code> 操作符执行 <code>search()</code>
<code>search_n(beg, end, n, val)</code>	在[<code>beg</code> , <code>end</code>)范围内查找 <code>val</code> 出现 <code>n</code> 次的子序列
<code>search_n(beg, end, n, val, pred)</code>	使用函数 <code>pred</code> 代替 <code>==</code> 操作符执行 <code>search_n()</code>
<code>binary_search(beg, end, val)</code>	二分查找, 在[<code>beg</code> , <code>end</code>)中查找 <code>val</code> , 找到返回 <code>true</code>
<code>binary_search(beg, end, val, comp)</code>	使用函数 <code>comp</code> 代替比较操作符执行 <code>binary_search()</code>

边界

函数	作用
lower_bound(beg, end, val)	在[beg , end)范围内的可以插入 val 而不破坏容器顺序的第一个位置, 返回一个 ForwardIterator (返回范围内第一个大于等于值val的位置)
lower_bound(beg, end, val, comp)	使用函数 comp 代替比较操作符执行 lower_bound()
upper_bound(beg, end, val)	在[beg , end)范围内插入 val 而不破坏容器顺序的最后一个位置, 该位置标志一个大于 val 的值, 返回一个 ForwardIterator (返回范围内第一个大于 val的位置)
upper_bound(beg, end, val, comp)	使用函数 comp 代替比较操作符执行 upper_bound()
equal_range(beg, end, val)	返回一对 iterator , 第一个表示 lower_bound , 第二个表示 upper_bound
equal_range(beg, end, val, comp)	使用函数 comp 代替比较操作符执行 lower_bound()

17.16 删 除 和 替 换 算 法 (15个)

复 制

函数	作用
copy(beg, end, res)	复制[beg , end)到 res
copy_backward(beg, end, res)	与 copy() 相同, 不过元素是以相反顺序被拷贝

移 除

函数	作用
<code>remove(beg, end, val)</code>	移除 [first, last) 区间内所有与 val 值相等的元素，并不是真正的从容器中删除这些元素(原容器的内容不会改变)而是将结果复制到一个以 result 为起始位置的容器中。新容器可以与原容器重叠
<code>remove_if(beg, end, pred)</code>	删除 [beg , end) 内 pred 结果为 true 的元素
<code>remove_copy(beg, end, res, val)</code>	将所有不等于 val 元素复制到 res ，返回 OutputIterator 指向被拷贝的末元素的下一个位置
<code>remove_copy_if(beg, end, res, pred)</code>	将所有使 pred 结果为 true 的元素拷贝到 res

替换

函数	作用
<code>replace(beg, end, oval, nval)</code>	将[beg , end)内所有等于 oval 的元素都用 nval 代替
<code>replace_copy(beg, end, res, oval, nval)</code>	与 replace() 类似，不过将结果写入 res
<code>replace_if(beg, end, pred, nval)</code>	将[beg , end)内所有 pred 为 true 的元素用 nval 代替
<code>replace_copy_if(beg, end, res, pred, nval)</code>	与 replace_if() ，不过将结果写入 res

去重

函数	作用
unique(beg, end)	清除序列中相邻重复元素，不真正删除元素。重载版本使用自定义比较操作
unique(beg, end, pred)	将所有使 pred 结果为 true 的相邻重复元素去重
unique_copy(beg, end, res)	与 unique 类似，不过把结果输出到 res
unique_copy(beg, end, res, pred)	与 unique 类似，不过把结果输出到 res

交换

函数	作用
swap(a, b)	交换存储在 a 与 b 中的值
swap_range(beg1, end1, beg2)	将[beg1 , end1)内的元素[beg2 , beg2+beg1-end1)元素值进行交换
iter_swap(it_a, it_b)	交换两个 ForwardIterator 的值

17.17 算术算法(4个) <numeric>

函数	作用
accumulate(beg, end, val)	对[beg , end)内元素之和, 加到初始值 val 上
accumulate(beg, end, val, binary)	将函数 binary 代替加法运算, 执行 accumulate()
partial_sum(beg, end, res)	将[beg , end)内该位置前所有元素之和放进 res 中
partial_sum(beg, end, res, binary)	将函数 binary 代替加法运算, 执行 partial_sum()
adjacent_difference(beg1, end1, res)	将[beg , end)内每个新值代表当前元素与上一个元素的差放进 res 中
adjacent_difference(beg1, end1, res, binary)	将函数 binary 代替减法运算, 执行 adjacent_difference()
inner_product(beg1, end1, beg2, val)	对两个序列做内积(对应元素相乘, 再求和)并将内积加到初始值 val 上
inner_product(beg1, end1, beg2, val, binary1, binary2)	将函数 binary1 代替加法运算, 将 binary2 代替乘法运算, 执行 inner_product()

17.18 关系算法(4个) <stl_algobase.h>

函数	作用
equal(beg1, end1, beg2)	判断[beg1 , end1)与[beg2 , end2)内元素都相等
equal(beg1, end1, beg2, pred)	使用 pred 函数代替默认的 == 操作符
includes(beg1, end1, beg2, end2)	判断[beg1 , end1)是否包含[beg2 , end2), 使用底层元素的 < 操作符, 成功返回true。重载版本使用用户输入的函数
includes(beg1, end1, beg2, end2, comp)	将函数 comp 代替 < 操作符, 执行 includes()
lexicographical_compare(beg1, end1, beg2, end2)	按字典序判断[beg1 , end1)是否小于[beg2 , end2)
lexicographical_compare(beg1, end1, beg2, end2, comp)	将函数 comp 代替 < 操作符, 执行 lexicographical_compare()
mismatch(beg1, end1, beg2)	并行比较[beg1 , end1)与[beg2 , end2), 指出第一个不匹配的位置, 返回一对 iterator , 标志第一个不匹配元素位置。如果都匹配, 返回每个容器的 end
mismatch(beg1, end1, beg2, pred)	使用 pred 函数代替默认的 == 操作符

17.19 集合算法(6个)

函数	作用
merge(beg1,end1,beg2,end2,res)	合并[beg1 , end1)与[beg2 , end2)存放到 res
merge(beg1,end1,beg2,end2,res,comp)	将函数 comp 代替 < 操作符, 执行 merge()
inplace_merge(beg,mid,end)	合并[beg , mid)与[mid , end), 结果覆盖 [beg , end)
inplace_merge(beg,mid,end,cmp)	将函数 comp 代替 < 操作符, 执行 inplace_merge()
set_union(beg1,end1,beg2,end2,res)	取[beg1 , end1)与[beg2 , end2)元素并集存放 到 res
set_union(beg1,end1,beg2,end2,res,comp)	将函数 comp 代替 < 操作符, 执行 set_union()
set_intersection(beg1,end1,beg2,end2,res)	取[beg1 , end1)与[beg2 , end2)元素交集存放 到 res
set_intersection(beg1,end1,beg2,end2,res,comp)	将函数 comp 代替 < 操作符, 执行 set_intersection()
set_difference(beg1,end1,beg2,end2,res)	取[beg1 , end1)与[beg2 , end2)元素内差集存放 到 res
set_difference(beg1,end1,beg2,end2,res,comp)	将函数 comp 代替 < 操作符, 执行 set_difference()
set_symmetric_difference(beg1,end1,beg2,end2,res)	取[beg1 , end1)与[beg2 , end2)元素外差集存放 到 res

17.20 排列组合算法：提供计算给定集合按一定顺序的所有可能排列组合

函数	作用
next_permutation(beg, end)	取出[beg , end)内的下移一个排列
next_permutation(beg, end, comp)	将函数 comp 代替 < 操作符, 执行 next_permutation()
prev_permutation(beg, end)	取出[beg , end)内的上移一个排列
prev_permutation(beg, end, comp)	将函数 comp 代替 < 操作符, 执行 prev_permutation()

17.21 堆算法(4个)

函数	作用
make_heap(beg, end)	把[beg , end)内的元素生成一个堆
make_heap(beg, end, comp)	将函数 comp 代替 < 操作符, 执行 make_heap()
pop_heap(beg, end)	重新排序堆。它把first和last-1交换, 然后重新生成一个堆。可使用容器的back来访问被"弹出"的元素或者使用 pop_back进行真正的删除。并不真正把最大元素从堆中弹出
pop_heap(beg, end, comp)	将函数 comp 代替 < 操作符, 执行 pop_heap()
push_heap(beg, end)	假设first到last-1是一个有效堆, 要被加入到堆的元素存放在位置 last-1 , 重新生成堆。在指向该函数前, 必须先把元素插入容器后
push_heap(beg, end, comp)	将函数 comp 代替 < 操作符, 执行 push_heap()
sort_heap(beg, end)	对[beg , end)内的序列重新排序
sort_heap(beg, end, comp)	将函数 comp 代替 < 操作符, 执行 push_heap()

参考:

《STL源码剖析》 -侯捷

<https://www.jianshu.com/p/eb554b0943ab>

17.22 结尾

哈喽，我是小贺，就爱分享知识，如果觉得文章对你有帮助，别忘记关注我哦！



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复「大礼包」
白嫖编程学习资料

关注公众号回复「加群」拉
你进百人技术交流群

十八、如何更好的学习一门知识

1、学会学习再学习

《刻意练习》里面提到顶级高手都是练出来的，如果没有刻意的训练，一切都是天真的练习。

学习任何知识和技能都需要有目的的练习。

什么叫有目的的练习呢？有目的的练习包含四个方面的特点：

- 第一个有目的的练习是具有定义明确的特定目标

- 第二个有目的的练习是专注的
- 第三个有目的的练习包含反馈
- 第四个有目的的练习需要走出舒适区

学会怎么学习，比学习本身，更重要，我们必须有线性而具象化的场景，去确定自己是否掌握了这份内容，简单来说，

比如，你想要做一个事儿，你必须告诉自己，几月几号，我将通过怎样的行为，必须拿到如此这般的结果，

目标，是必须完成的，而不是想要完成的，而我常用的三部曲是，

A, 归类整理，（比如，想学某个 C++ 领域的技能，先把圈内同领域相关内容笔记都看一遍）

B, 横向迁移，（然后，求同存异的整理出属于自己咀嚼实战后的一篇笔记流程和注意事项）

C, 学以致用，（最后，亲手实操，遇到问题，解决问题，不懂就问，然后拿到第一个正反馈）

2、三步走战略

看视频/书 + 敲代码 + 记笔记（视频为主、书为辅） 算法 + 基础 + 技术栈。

计算机科学是一门注重实践的学科，多写代码，多看优秀的源码，水滴石穿，日拱一卒，技术水平就会潜移默化的提高。

3、五大能力提升

快速的学习能力、应用业务的能力、解决问题的能力、信息检索的能力、有效资料判断力是一个任何行业从业者最基本的也是最根本的能力。

1、快速的学习能力：指的是面对新的技术领域，自己从未接触过的技术，能不能在有效的时间内，把它学会、弄懂，这就是学习能力；

2、应用业务的能力：指的是技术到位了，能不能用于项目中，用于实际问题中，在工作中尤为重要，学好跟用起来差距还是挺大的，听懂了不一定会做题，这就是学以致用；

3、解决问题的能力：指的是根据现有的技术，现有的资源、举一反三、触类旁通、灵活运用，能否解决一个技术难题、攻克困难的能力；

4、信息检索的能力：指的是利用互联网上面的一切资源，利用各种搜索技术、各种网站、各种论坛、快速的找到你所需要的资源（视频、电子书、图片等等等）的能力；

5、有效资料判断力：指的是现在互联网上面，优质资源有，但是劣质的资源更多，随手一搜，一大片，你是否具有一定的判断力呢？比如 C++ 视频一大堆，哪个是比较优质的呢？有效资料判断力可以帮助你少走弯路，节省大把的时间，这就是学会筛选的能力。

五大能力的提升，是通往高手的必经之路！

哈喽，我是小贺哥，就爱分享编程知识，如果觉得文章对你有帮助，别忘记关注我哦！



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复 「**大礼包**」
白嫖编程学习资料

关注公众号回复 「**加群**」 拉
你进百人技术交流群

十九、Google 面试准备完全指南 && C++代码规范：

面试准备完全指南参考链接

<https://wizardforcel.gitbooks.io/gainlo-interview-guide/content/cgkip.html>

Google C++ 风格

The screenshot displays two side-by-side snippets of C++ code from the Google C++ Style Guide, each annotated with numerous callouts explaining specific coding practices. The left snippet shows general style rules like header file organization, namespaces, and loop constructs. The right snippet focuses on class and interface definitions, including inheritance, access control, and function signatures. Annotations cover topics such as header file naming, class names, and the use of various C++ features like templates and enums.

Annotations include:

- Header file organization and naming conventions.
- Namespace usage and scope.
- Loop constructs and iteration patterns.
- Class and interface definitions, including inheritance and access control.
- Function signatures and parameter handling.
- Variable and member variable naming conventions.
- String literals and character sets.
- Commenting and documentation practices.

A prominent callout at the bottom right points to the original source: <https://blog.csdn.net/YoungHong1992>.

哈喽，我是小贺哥，就爱分享编程知识，如果觉得文章对你有帮助，别忘记关注我哦！



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复 「大礼包」
白嫖编程学习资料

关注公众号回复 「加群」 拉
你进百人技术交流群

二十、关于面试（面试经验和资料）

看到这里，相信这位读者也是有耐心的，那么，我就多 BB 几句吧，可能有人觉得背下八股文，面试就没问题了，这里，我很坦诚的告诉你，也给大家提个醒，**只背八股文，是不行的，不太好进大公司。**

大家要知道，面试官也是人，也知道候选人都在背八股文，而且面试官面过很多人，身经百战，你是背诵的，还是自己深刻理解过的，面试官一面你，就能知道你几斤几两。

也就是说，如果只会照本宣科的背八股文，面试会比较难受，碰到稍微严格的面试官，挖你细节，问到你不会为止，你会扛不住，无法根据情景说出自己的理解，这会给面试官很不好的印象，觉得你只会照猫画虎。

在我看来，无论多么浅显的八股文，都要经过自己的**实战经验，深度思考，再用自己的理解说出来**，就算你的回答不是最好的答案，我觉得都没关系，你要让面试官看到你的潜力，看到你严谨的思维，清晰的表达。

按照应试的方式来学这些八股文，也确实管用，但大家一定要注意，**要理解，要捋清思路，然后用自己的话说出来。**

那么，BB 了这么多，对于面试，小贺结合自己当年找工作，还有最近和同事参与了部门的一些招聘说几点经验吧：

- 不要一直在做准备，准备的差不多了，就直接去面试，往往进步是最快的，要给自己 deadline。
- 面试中没有回答上来的问题，后面一定要复盘并做准备，尤其是高频问题。
- 在面试表达中，要时刻提醒自己，反思自己的表达是否合理。
- 回答问题的时候先直接抛出答案，先让面试官给一个不刷你的理由，再逐步展开讲解。
- 不要一个问题回答的绕来绕去，答非所问，会让面试官反感，毕竟大家的时间都有限。
- 面试挂了不代表你的水平就不行，多面几家，公司招的是一个合适的人选，你面的也是一个适合你的公司。
- 算法题很重要，要好好准备，现在的互联网面试第一关算法手撕几乎是必备的。
- 小贺朋友圈一个大佬朋友一个月面试 50 多场的感悟：**如果你还不是到即将面试的时间点，多花时间系统学习底层知识，同时扩充知识面，才是长久之计。这个时间你需要的是加速度，而不是速度。**
- 如何系统的学习一门知识，知乎上搜索「如何系统学习xxx」很多答案。

关于面试经验，我之前在公众号也分享了研究生的两位师弟的面经，如果没有看的同学，可以参考：

- [研二师弟拿下微信 offer](#)
- [研二师弟斩获阿里，美团，华为 offer！](#)

关于面试资料，我也整理了一些牛客网的 C++ 资料放到百度网盘上，大家按需自取：

- 链接: https://pan.baidu.com/s/1En4ATBTxL29aNvZQs_hBhw 提取码: kg74
- 如果网盘失效访问 https://github.com/rongweihe/CS_Offer/tree/master/cs_books

哈喽，我是小贺哥，就爱分享编程知识，如果觉得文章对你有帮助，别忘记关注我哦！



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复 「大礼包」
白嫖编程学习资料

关注公众号回复 「加群」 拉
你进百人技术交流群

二十一、画图经验

正所谓，工欲善其事必先利其器。

小贺写了这么多篇技术文章，最大的体会就是，熟练掌握使用一些好用的工具，会让你做事效率提高很多，那之前也有很多读者朋友来问我是使用什么画图工具，那我就干脆写一篇介绍下我是怎么画图的。

如果一篇技术文章缺少了自己画的图片，相当于一部电影缺失了主角一样失去了灵魂，技术文章本身就很枯燥，如果文章中没有几张图片，读者被劝退的概率就会大大滴，剩下没被劝退的估计看着看着就睡着了。

所以，精美的图片可以说是必不可少的一部分，不仅在阅读时能带来视觉的冲击，而且图片相比文字能涵盖更多的信息，不然怎会有一图胜千言的说法呢？

图画的好，有的时候也会让外行人看起来有种高大上的感觉，哈哈。

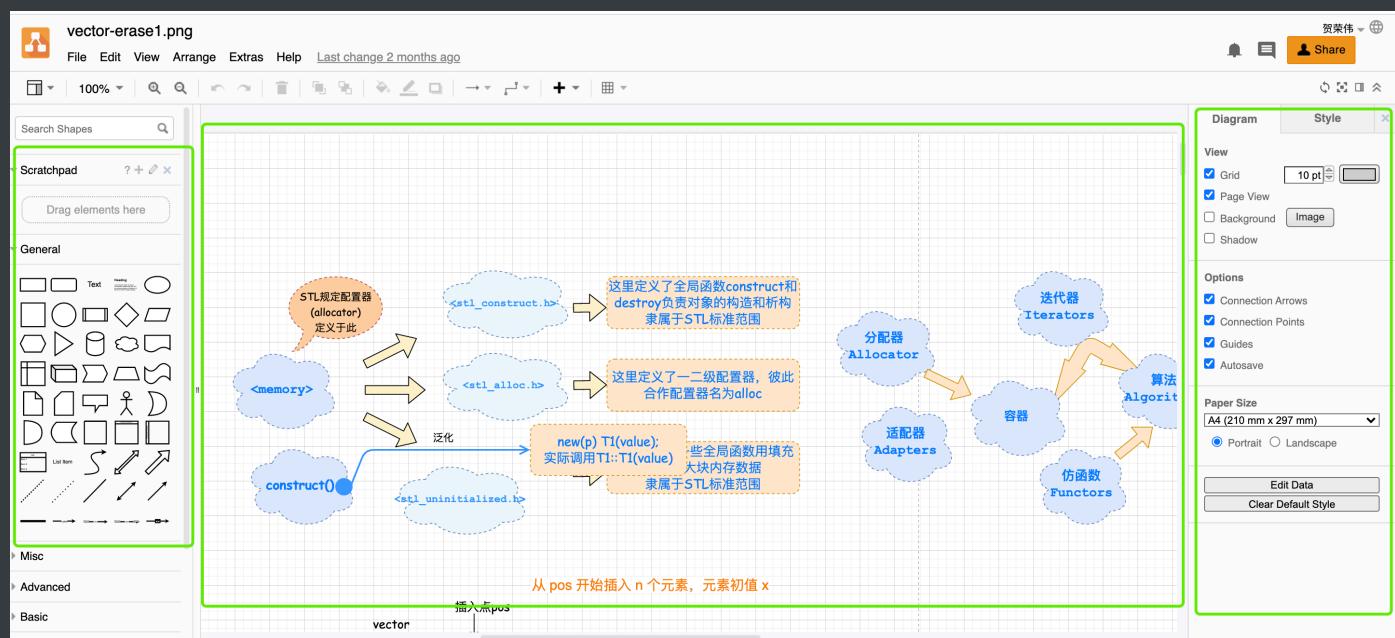
那么小贺也不绕弯子了，直接坦白讲，我用的是一个在线的画图网址，地址是：

<https://www.draw.io/> (会重定向到 <https://app.diagrams.net/>)

用它的原因是使用方便和简单，当然最重要的是它完全免费，没有什么限制，甚至还能直接把图片保存到 GoogleDrive 、 OneDrive 和 Github，我就是保存到 Github，然后用 Github 作为我的图床。

打开上面的网址，主要分为三个区域，从左往右的顺序是「图形选择区域、绘图区域、属性设置区域」。

可以按照不同的文章风格去画图，简而言之，这个网址强烈推荐！👍👍



另外，在给大家推荐一个代码截图网址：直接把代码粘贴进去，一键生成图片形式，有的时候，代码通常都是比较长，在手机看文章用图片的呈现的方式会更舒服清晰。

<https://carbon.now.sh/>

网站页面如下图，怎么样，代码显示的效果是不是很美观？

carbon

Create and share beautiful images of your source code.
Start typing or drop a file into the text area to get started.

```
const pluckDeep = key => obj => key.split('.').reduce((accum, key) => accum[key], obj)

const compose = (...fns) => res => fns.reduce((accum, next) => next(accum), res)

const unfold = (f, seed) => {
  const go = (f, seed, acc) => {
    const res = f(seed)
    return res ? go(f, res[1], acc.concat([res[0]])) : acc
  }
  return go(f, seed, [])
}
```

哈喽，我是小贺哥，就爱分享编程知识，如果觉得文章对你有帮助，别忘记关注我哦！



扫一扫，关注「herongwei」公众号

一个逗比的五道口打工仔
分享有趣的编程知识和生活

关注公众号回复「大礼包」
白嫖编程学习资料

关注公众号回复「加群」拉
你进百人技术交流群

二十二、赞赏支持

本系列「C++八股文」的所有图片都是小贺纯手打的，全文共 20 字 + 100 张图，这么做的原因很简单，就是为了大家突破面试，阅读源码的痛点。

如果对你有帮助，可以给小贺一个小小的赞赏，金额多少不重要，要的是你们的小小心意，会助力小贺输出更多优质的文章，先提前跟大家说声谢谢。

微信赞赏码



支付宝赞赏码



推荐使用支付宝



加贝木苇 (**伟)

打开支付宝[扫一扫]

申请官方收钱码：拨打 95188-6