C++面试连环问-内存管理篇

阿Q正砖
公众号：阿Q正砖

关注

3 人赞同了该文章

内存篇

1、内存泄漏？怎么解决？

内存泄漏是指程序在动态分配内存后，未释放或者未能完全释放该内存空间的情况。这样会导致内存不断被占用，进而导致程序性能下降、甚至崩溃等问题。

解决内存泄漏问题需要先确定内存泄漏的原因，可以通过以下几个步骤来解决内存泄漏问题：

1. 排查代码：查看代码中是否有明显的内存泄漏的情况，例如忘记释放内存等。
2. 使用工具检查：可以使用一些内存泄漏检测工具，例如Valgrind、Purify、AddressSanitizer等，来检测程序中的内存泄漏情况。
3. 检查资源的使用情况：程序中除了内存泄漏还可能存在其他资源泄漏，例如文件句柄、网络连接等，需要逐一检查并进行相应的释放。
4. 使用智能指针：在C++中，可以使用智能指针（shared_ptr、unique_ptr、weak_ptr）等RAII技术来管理动态内存，自动释放资源，避免忘记释放内存的问题。
5. 重构代码：如果程序中的内存泄漏问题比较严重，无法通过以上方法解决，可以考虑对代码进行重构，优化内存使用情况，避免内存泄漏的问题。

2、说说常见的内存泄漏都有哪些？

- 对象被无意识地持续引用：在使用完对象后，程序没有将其引用置为NULL，导致这些对象一直占用内存。
- 内存分配未释放：程序中使用了动态分配内存的函数（如malloc、calloc、realloc等）分配内存，但没有调用free函数进行释放。
- 大对象未分配内存池：如果需要频繁地分配、释放大对象（如数组、矩阵等），直接调用系统函数分配内存可能会导致内存碎片化，进而导致系统内存泄漏。此时，可以使用内存池技术来解决这个问题。
- 循环引用：当两个或多个对象之间互相引用时，它们会互相持有对方的引用，当这些对象中有一个引用没有被释放时，将导致内存泄漏。
- 持续增长的缓存：当一个缓存区在使用后没有被清空或者不定期的清理，会导致缓存中的数据越来越多，最终导致内存泄漏。

为了解决内存泄漏问题，需要进行内存泄漏检测和内存泄漏排查。一些编程语言和开发工具可以提供内存泄漏检测的功能，可以通过这些工具来查找内存泄漏的代码位置，并及时修复。同时，在编写代码时，也应该遵循良好的编程习惯，及时释放已经不再使用的内存，以避免内存泄漏问题的出现。

3、如何避免内存泄漏？

- 确保在程序中每次使用完内存后及时释放，特别是对于动态分配的内存，要在不需要时及时释放。
- 确保内存释放的正确性，例如使用free函数时，需要确保传递给它的指针是指向动态分配的内存空间。
- 对于需要长时间占用内存的程序，可以考虑采用内存池技术，动态分配一定数量的内存空间，在使用完成后放回内存池中，避免频繁申请和释放内存造成的性能影响。
- 对于需要频繁申请和释放内存的程序，可以考虑采用内存缓存技术，将频繁使用的内存缓存起来，避免频繁申请和释放内存造成的性能影响。
- 使用内存泄漏检

4、你知道常见的内存错误吗？再说说解决的对策？

1. 内存泄漏：指已经不再需要使用的内存没有被释放，导致内存浪费。解决方案可以采用以下方法：
   1. 手动管理内存并调用 `free()` 释放不再使用的内存；
   2. 使用智能指针等自动内存管理机制；
   3. 使用内存泄漏检测工具定位和修复内存泄漏问题。

2. 内存溢出：指分配的内存空间不足以满足当前需要，导致程序崩溃。解决方案可以采用以下方法：
   1. 程序设计时充分考虑内存使用情况，合理地规划内存分配；
   2. 使用内存监控工具或者操作系统提供的性能工具，监测和分析程序内存使用情况；
   3. 优化算法或数据结构，减少内存占用。

1. 野指针：指指针指向了已经被释放的内存空间，或者指针未被初始化就被使用。解决方案可以采用以下方法：
   1. 对指针变量进行初始化；
   2. 在指针使用之前，检查其是否为空或者指向的内存是否被释放；
   3. 使用 `nullptr` 代替 `NULL` ，避免空指针问题；
   4. 使用智能指针等自动内存管理机制，避免手动释放内存的错误。

5、详细说说内存的分配方式？

内存的分配方式有两种：静态内存分配和动态内存分配。

1. 静态内存分配：在程序编译时就已经分配好内存，运行时不能改变分配的内存大小，程序执行速度快，但是空间利用率低，不能灵活分配内存空间。常见的静态内存分配有：
   1. 全局变量：在程序编译时分配内存，整个程序执行期间内存不释放。
   2. 静态局部变量：在函数执行时分配内存，但是该内存空间在函数执行完毕后不释放，可以用 static 修饰符来声明。
   3. 静态数组：在编译时就分配内存，执行期间内存不释放。
   4. 静态结构体：在编译时就分配内存，执行期间内存不释放。

2. 动态内存分配：在程序运行时才分配内存，可以根据需要灵活地分配和释放内存空间。常见的动态内存分配有：
   1. malloc()：在堆上分配指定大小的内存空间，返回一个指向这段内存的指针。
   2. calloc()：在堆上分配指定数量和大小的内存空间，返回一个指向这段内存的指针。
   3. realloc()：调整已分配内存的大小，返回一个指向这段内存的指针。
   4. free()：释放已经分配的内存。

动态内存分配的优点是空间利用率高、可以根据需要灵活地分配和释放内存空间，但是容易引起内存泄漏和内存碎片问题。因此在使用动态内存分配时要注意及时释放已经不再需要的内存空间，避免内存泄漏问题的发生。

6、堆和栈的区别？

栈（stack）是一种先进后出（Last In First Out，LIFO）的数据结构，由编译器自动管理，存放程序的局部变量、函数参数和返回地址等信息。栈的内存空间由操作系统自动分配和释放，其空间大小是固定的，一般为几MB至几十MB。

堆（heap）则是一种动态内存分配方式，程序员需要手动申请和释放堆内存。堆的内存空间由操作系统管理，其大小可以动态增加或减少，一般情况下堆的空间远远大于栈。

在程序运行过程中，栈的分配和释放速度较快，但栈的容量有限；堆的分配和释放速度较慢，但堆的容量较大。因此，对于较小的数据结构，优先使用栈来分配内存；对于较大的数据结构，需要动态管理内存时，可以使用堆来分配内存。

7、如何控制C++的内存分配？

在 C++ 中，可以通过重载 new 和 delete 运算符来控制内存分配。

重载 new 和 delete 运算符的方式如下：

```
void* operator new(size_t size);
void operator delete(void* p) noexcept;
```

其中， `operator new` 用于分配内存， `operator delete` 用于释放内存。

默认情况下， `operator new` 调用 `malloc` 函数分配内存， `operator delete` 调用 `free` 函数释放内存。但是，我们可以重载这些运算符，自定义内存分配和释放方式。

例如，下面是一个简单的例子，演示如何重载 `operator new` 和 `operator delete` 运算符：

```
#include <iostream>

void* operator new(size_t size)
{
    std::cout << "Allocating " << size << " bytes of memory" << std::endl;
    void* p = malloc(size);
    return p;
}

void operator delete(void* p) noexcept
{
    std::cout << "Deallocating memory" << std::endl;
    free(p);
}

int main()
{
    int* ptr = new int;
    delete ptr;
    return 0;
}
```

运行上面的代码，输出如下：

```
Allocating 4 bytes of memory
Deallocating memory
```

可以看到，重载后的 `operator new` 和 `operator delete` 运算符被调用，并输出了相关信息。

通过重载 `operator new` 和 `operator delete` 运算符，我们可以实现自定义的内存分配和释放方式，从而更好地控制 C++ 的内存分配。

8、你能讲讲C++内存对齐的使用场景吗？

C++内存对齐是指按照一定的规则将数据结构中的数据成员排列在内存中的过程，其目的是为了优化内存访问速度。常见的使用场景包括：

- 减少内存碎片：对齐可以保证结构体或类中的数据成员按照规则排列，避免因为数据成员的大小不一致而导致的内存碎片。
- 提高数据访问速度：由于现代计算机的内存访问是按照一定的块大小进行的，对齐可以保证数据成员按照块的大小进行存储，从而提高内存访问速度。
- 保证跨平台兼容性：不同的平台可能对内存对齐有不同的要求，使用内存对齐可以保证程序在不同平台上的运行效果一致。

需要注意的是，使用内存对齐可能会增加数据结构的大小，从而增加内存的占用。在一些对内存占用要求较高的场景下，需要仔细权衡内存占用和内存访问速度等因素，选择合适的内存对齐方式。

9、内存对齐应用于哪几种数据类型及其对齐原则是什么?

内存对齐通常应用于结构体、联合体和类中的数据成员，以保证数据在内存中的存储效率。

对于结构体、联合体和类中的数据成员，编译器会按照某种规则将它们存放在内存中，以保证各个数据成员之间的距离是整齐的，并且数据成员的地址是一致的。

在进行内存对齐时，通常需要遵守以下三个原则：

- 数据成员的偏移量必须是对齐数的整数倍。对齐数指的是编译器为了满足对齐要求而添加的字节大小。例如，对于4字节对齐的结构体，其对齐数为4，数据成员的偏移量必须是4的整数倍。
- 结构体、联合体和类的大小必须是对齐数的整数倍。即结构体、联合体和类的大小必须是它所包含的最大的数据成员大小的整数倍。例如，如果结构体中最大的数据成员的大小是8字节，对齐数是4，那么结构体的大小必须是8的整数倍，即16字节。
- 结构体中嵌套的结构体或联合体的起始地址必须符合其内部最严格数据成员的对齐要求。

10、你能说说什么是内存对齐吗?

内存对齐是指将数据结构中的每个成员按照一定的规则进行排列，使得每个成员的起始地址相对于该结构的起始地址偏移量为该成员大小的整数倍。这样做的目的是为了让处理器在读取数据时更加高效，因为处理器可以一次性读取多个连续地址上的数据，如果数据不对齐，处理器就需要多次读取，降低了读取速度。

11、那为什么要内存对齐呢

内存对齐是为了提高内存读取效率和数据存储安全而进行的一种处理方式。

当CPU从内存中读取数据时，如果数据没有按照规定的对齐方式进行存储，那么CPU需要分两次或更多次读取内存，这会增加CPU访问内存的时间和系统的开销。因此，内存对齐可以减少CPU访问内存的时间和系统开销，提高系统的效率。

此外，对于结构体等复合类型数据的内存存储，内存对齐还可以保证数据存储的安全性。如果数据没有按照规定的对齐方式存储，可能会导致数据被拆分存储在两个内存块中，这样会增加访问内存的复杂度，并且在多线程环境下可能会发生数据竞争的问题，导致数据的不一致性。而通过内存对齐，可以避免这些问题的发生，提高数据存储的安全性。

12、能否举一个内存对齐的例子呢?

当某个结构体成员变量的类型与起始地址不是它大小的整数倍时，就需要字节对齐。以下是一个字节对齐的例子：

```
struct Example {
  char a;     // 占用 1 个字节
  int b;      // 占用 4 个字节
  double c;   // 占用 8 个字节
  char d[3];  // 占用 3 个字节
};
```

在这个例子中， `a` 变量占用了 1 个字节， `b` 变量占用了 4 个字节， `c` 变量占用了 8 个字节， `d` 数组占用了 3 个字节。如果这个结构体按照自然对齐（默认情况下的对齐方式）来分配内存，那么变量的内存布局如下所示：

```
| 1字节 |   3字节   | 4字节 | 8字节 |
|-------|-----------|-------|-------|
|   a   |   [pad]   |   b   |   c   |
|       |   [pad]   |       |       |
|       |   [pad]   |       |       |
|       |     d     |       |       |
```

在这个布局中， `a` 变量的内存占用了 1 个字节，与其大小相等。但是， `b` 变量的内存占用了 4 个字节，虽然它只需要占用 4 字节，但是却占用了 8 字节的内存，多出了 4 个字节。这是因为在默认情况下，编译器会为 `b` 变量分配 4 个字节的内存，并在它后面填充 3 个字节的 padding，以保证变量的地址是 8 的倍数，从而提高内存访问的效率。

同理， `c` 变量的内存占用了 8 个字节，与其大小相等， `d` 数组的内存占用了 3 个字节，与其大小相等，但是为了保证结构体占用的内存是 8 的倍数，它后面填充了 5 个字节的 padding。

13、你知道C++内存分配可能会出现哪些问题吗?

- 内存泄漏：在使用完堆上的内存后没有及时释放，导致程序运行过程中不断地占用内存。
- 内存溢出：在申请内存时超出了操作系统或程序所能提供的内存上限，导致程序崩溃。
- 悬垂指针：指向已经被释放的内存区域，导致程序访问非法内存而崩溃。
- 双重释放：在释放内存时出现重复释放同一内存区域的情况，导致程序崩溃。
- 内存访问越界：程序访问了已经超出了申请内存空间的范围，导致程序崩溃。

为了避免这些问题的发生，我们在编写C++程序时需要遵循一些规则，如正确使用new/delete、malloc/free等内存管理函数，合理地设计数据结构等。此外，还可以使用一些工具来辅助检测内存相关的问题，例如Valgrind、GDB等。

14、说一说指针参数是如何传递内存?

指针参数在函数调用时传递的是地址，也就是指向变量内存地址的指针。因此，在函数中通过指针参数修改变量的值，其实就是通过地址间接修改了变量的值。指针参数的传递是按值传递的，也就是传递的是指针变量的值，也就是地址。函数中的指针参数是函数调用者的一个变量地址的副本，也就是指针变量的值的副本，因此修改指针的值不会影响原始的指针变量。但是，修改指针所指向的内存地址中的内容，会直接影响原始变量的值。

举个例子，假设有以下代码：

```
void func(int* p) {
    *p = 10;
    p = NULL;
}

int main() {
    int a = 0;
    int* p = &a;
    func(p);
    printf("%d\n", a);
    printf("%p\n", p);
    return 0;
}
```

在调用 `func` 函数时，将指向 `a` 的指针 `p` 传递给函数。在函数中，将 `p` 指向的内存地址中的内容修改为 `10` 。但是，对 `p` 赋值为 `NULL` 只会影响函数内部的 `p` 指针副本，不会影响函数外部的 `p` 指针变量。因此，函数结束后， `p` 仍然指向 `a` 的地址。最后输出 `a` 的值为 `10` ， `p` 的值为 `a` 的地址。

15、什么是野指针？如何预防呢?

野指针是指指向已经释放的内存空间的指针，或者指向未被分配的内存空间的指针。当程序试图使用野指针时，就可能会导致程序崩溃或者出现意想不到的结果。

为了预防野指针问题，可以采取以下措施：

- 初始化指针：在定义指针时，尽量立即进行初始化，可以将指针赋值为NULL或nullptr。这样即使在后续使用过程中出现了未被分配的指针，也不会成为野指针。
- 及时释放指针：
  野指针的产生。

- 置空指针：在释放指针的内存之后，及时将指针赋值为NULL或nullptr，以防止指针继续被使用。
- 避免悬挂指针：当一个指针被释放之后，如果仍然指向原来的内存区域，那么在其他代码中可能会误认为该内存区域仍然可用，从而出现悬挂指针问题。为了避免这种情况，可以在释放指针时将其指向的内存区域清零或者赋值为特定的值，这样就可以避免出现悬挂指针的问题。

16、内存耗尽怎么办?

- 使用内存池：内存池是一种管理内存分配和释放的技术，它可以预分配一定数量的内存，并将其缓存起来，当程序需要分配内存时，就直接从缓存中取出一块内存使用。
- 优化算法：尽可能地避免不必要的内存分配，可以考虑使用一些高效的算法和数据结构，如缓存、哈希表等。
- 调整系统参数：可以通过修改操作系统的一些参数来增加可用内存，如增加虚拟内存、减少进程数量等。
- 释放不必要的内存：在程序运行过程中，及时释放不再使用的内存，避免内存浪费。

17、什么是内存碎片，怎么避免内存碎片?

内存碎片是指内存中存在大量不连续的、小块的未使用内存空间，这些空间不能被分配给大块的内存请求，从而导致系统无法满足内存请求的情况。内存碎片可能会导致程序性能下降，甚至系统崩溃。

为了避免内存碎片，可以采取以下措施：

- 尽量避免频繁的内存分配和释放，可以采用对象池等技术来管理内存。
- 使用内存池技术，对一定大小范围内的内存进行预分配，避免频繁的内存分配和释放。
- 使用动态分配内存的时候，尽量分配固定大小的块，而不是小块，避免出现大量的内存碎片。
- 使用内存对齐技术，可以减少内存碎片的发生。
- 定期进行内存整理，将多个小的内存块合并成一个大的内存块。
- 对于长时间运行的应用程序，可以考虑使用内存映射文件等技术，将数据保存在文件中，而不是内存中，避免内存碎片的发生。

18、简单介绍一下C++五大存储区

- 代码区（Code Segment）：存储程序执行的代码。
- 全局区（Global Segment/Data Segment）：存储全局变量和静态变量，包括未初始化和已初始化的变量。
- 堆区（Heap Segment）：由程序员手动申请和释放的内存空间。
- 栈区（Stack Segment）：存储函数的参数值、局部变量等。
- 常量区（Constant Segment）：存储常量数据，如字符串常量。

这五个存储区都有其特定的作用和生命周期，在 C++ 编程中需要了解清楚它们的特点，合理地利用它们，才能编写出高效可靠的程序。

19、内存池的作用及其实现方法

内存池是一种常见的内存管理技术，它的作用是提高内存的利用率，减少内存碎片，以及提高内存分配和释放的效率。

内存池的实现方法一般有两种：

1. 预分配固定大小的内存块，当需要分配内存时，从内存池中取出一个已经分配好的内存块，使用完之后再将其归还到内存池中。
2. 动态分配内存，但是将内存分为大小相等的块，当需要分配内存时，从内存池中取出一个大小合适的内存块，使用完之后再将其归还到内存池中。

这两种方法的优缺点如下：

1. 预分配固定大小

优点：

- 分配和释放内存非常快，因为内存块的大小是固定的。
- 可以避免内存碎片的问题，因为内存块的大小是固定的，不会出现大小不一的内存块。

缺点：

- 浪费空间，因为预分配的内存块可能并不全部被使用，这些未使用的内存块就浪费了。
- 不够灵活，因为内存块的大小是固定的，如果某些对象需要更大或更小的内存块，就需要重新设计内存池的大小和结构。

1. 动态分配内存：

优点：

- 更灵活，因为内存块的大小可以根据需要动态调整。
- 更节省空间，因为只分配需要的内存块。

缺点：

- 分配和释放内存较慢，因为需要动态分配和回收内存。
- 可能会出现内存碎片的问题，因为内存块的大小不固定，容易出现大小不一的内存块，造成内存碎片。

20、如何构造一个类，使得只能在堆上或者在栈上分配内存?

构造一个类，使得只能在堆上或者在栈上分配内存，可以通过重载 `new` 和 `delete` 运算符来实现。

对于栈上分配内存，可以重载 `new` 和 `delete` 运算符，并将 `new` 运算符重载为返回地址。

对于堆上分配内存，可以使用 `placement new` 运算符手动调用构造函数，并将返回的指针作为类的指针。在堆上分配内存时，需要重载 `new` 和 `delete` 运算符来调用 `malloc` 和 `free` 进行内存分配和释放。同时，需要使用类的 `placement new` 运算符来调用构造函数，以确保对象被正确初始化，并在析构时调用类的析构函数。

下面是一个示例代码，演示如何将类的内存分配限制为堆上或者栈上：

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

class MyClass {
public:
    // 重载 new 运算符，只允许在堆上分配内存
    void* operator new(std::size_t size) {
        void* ptr = std::malloc(size);
        if (!ptr) {
            throw std::bad_alloc();
        }
        return ptr;
    }

    // 重载 delete 运算符，释放在堆上分配的内存
    void operator delete(void* ptr) {
        std::free(ptr);
    }

    // 重载 placement new 运算符，只允许在栈上分配内存
    void* operator new(std::size_t size, void* ptr) {
        return ptr;
    }

    // 构造函数
```

```
    MyClass() {
        std::cout << "MyClass constructor\n";
    }

    // 析构函数
    ~MyClass() {
        std::cout << "MyClass destructor\n";
    }
};

int main() {
    // 在堆上分配内存
    MyClass* p1 = new MyClass();
    delete p1;

    // 在栈上分配内存
    alignas(MyClass) char buffer[sizeof(MyClass)];
    MyClass* p2 = new(buffer) MyClass();
    p2->~MyClass();

    return 0;
}
```

在上面的示例代码中， `operator new` 和 `operator delete` 运算符被重载，以限制内存分配在堆上。同时，使用了 `placement new` 运算符，手动调用构造函数，以便在栈上分配内存。

21、物理内存和虚拟内存的原理和区别分别是什么?

物理内存是指计算机中实际存在的内存，它由硬件组成，是直接可见的。而虚拟内存是操作系统提供的一种机制，它将计算机的硬盘空间作为内存的一部分来使用，使得程序可以访问比物理内存更大的内存空间。

物理内存的原理是通过内存条等硬件设备将数据存储在RAM中，它的访问速度非常快。当物理内存不足时，操作系统会将一部分内存中的数据转移到硬盘空间中，这就是虚拟内存的原理。虚拟内存将硬盘空间中的一部分作为内存空间来使用，通过虚拟内存地址与物理内存地址之间的映射关系，使得程序可以访问比物理内存更大的内存空间。

物理内存和虚拟内存的区别主要有以下几点：

- 大小不同：物理内存的大小受限于计算机硬件的配置，而虚拟内存的大小受限于硬盘的空间大小。
- 访问速度不同：物理内存的访问速度非常快，而虚拟内存的访问速度相对较慢。
- 内存管理方式不同：物理内存由操作系统直接管理，而虚拟内存则是由操作系统和硬件一起管理的。
- 分配方式不同：物理内存的分配是静态的，一般在启动时就已经分配好了，而虚拟内存的分配是动态的，操作系统会根据需要动态地分配虚拟内存。

22、C++中变量的存储位置？程序的内存分配?

在C++中，变量的存储位置可以分为以下几种：

- 栈（stack）：用于存储函数的局部变量和参数等。当函数被调用时，局部变量和参数等被分配在栈上，当函数返回时，这些变量就会被自动销毁。
- 堆（heap）：用于动态分配内存，比如new、malloc等函数分配的内存就位于堆上。需要手动管理内存的生命周期，使用完后需要调用delete或free等函数来释放内存，否则就会发生内存泄漏。
- 全局区（data segment）：用于存储全局变量、静态变量和常量等。这些变量的生命周期从程序开始到程序结束，它们位于程序的数据段中，内存由系统自动管理。
- 代码区（code segment）：用于存储程序的代码。

程序的内存分配是
数据区和堆栈区。

存。虚拟内存是一种将主存看作磁盘存储器扩展的技术，它可以将硬盘空间当作主存来使用。操作系统会将一部分主存空间作为虚拟内存，当程序需要分配内存时，操作系统会将一部分虚拟内存映射到主存中，程序就可以使用这些虚拟内存了。如果程序需要更多的内存，操作系统会将其余的虚拟内存映射到硬盘上，这样程序就可以继续使用虚拟内存了，这就是虚拟内存的原理。

物理内存是计算机中实际存在的内存，它是由硬件提供的，而虚拟内存则是由操作系统提供的一种扩展内存的技术，它利用硬盘空间来扩展主存空间，从而使得计算机可以运行更多的程序和更大的程序。在操作系统看来，虚拟内存和物理内存是两个不同的概念，它们之间的区别在于虚拟内存是一种抽象的概念，而物理内存是实际存在的硬件。

23、静态内存分配和动态内存分配的区别?

- 静态内存分配是指在程序编译期间，由编译器在编译期间为变量分配内存，这些内存空间在程序运行期间一直存在，直到程序结束才会被释放。静态内存分配适用于一些固定大小、生命周期长、不需要频繁创建和释放的变量，如全局变量和静态局部变量等。静态内存分配的内存大小在编译时确定，因此不能动态调整内存大小。
- 动态内存分配是指在程序运行期间，根据需要动态地为变量分配内存。动态内存分配由程序员手动管理，需要使用 `new` 操作符申请内存，使用 `delete` 操作符释放内存。动态内存分配适用于生命周期不确定、大小不固定、需要频繁创建和释放的变量。动态内存分配的优势是可以动态调整内存大小，但需要程序员自行管理内存分配和释放，如果不当使用可能会造成内存泄漏和内存溢出等问题。

总之，静态内存分配和动态内存分配在不同的场景下有各自的优势和劣势，程序员需要根据实际情况选择合适的内存分配方式。

24、什么是段错误？什么时候发生段错误?

段错误（Segmentation fault）是指程序试图访问非法的内存地址，或试图对没有写权限的内存地址进行写操作时产生的错误。它是一种常见的运行时错误，通常由于指针操作不当或者动态内存分配不当等原因引起。

具体来说，当程序访问一个未映射的地址、非法地址、只读地址或已释放的地址，或者当程序试图使用空指针访问内存时，就会触发段错误。

除此之外，还有一些其他的原因也会导致段错误，比如堆栈溢出、缓冲区溢出等。

在出现段错误时，操作系统会发送一个信号（SIGSEGV）给进程，导致程序崩溃或者被操作系统杀死。为了避免段错误的发生，开发人员需要注意程序中所有指针和内存操作的合法性，确保程序不会访问非法地址或已释放的地址。另外，对于动态内存的分配和释放，也需要谨慎处理，防止出现内存泄漏或者重复释放等问题。

25、内存块太小导致malloc和new返回空指针，该怎么处理?

当我们调用 `malloc` 或 `new` 分配内存时，如果请求的内存块大小过大，超过了系统可用的内存空间，则会返回一个空指针。同样地，如果请求的内存块大小过小，系统也无法为其分配足够的内存空间，也会导致返回空指针。这个空指针表示系统无法满足我们的内存请求。因此，我们需要在代码中对此进行处理，以确保程序的健壮性和稳定性。

针对内存块太小的情况，我们可以考虑减小内存块的分配单位或者增加可用内存大小。比如，可以将分配单位改为字节级别，或者增加系统可用的物理内存或虚拟内存空间。

当然，如果我们确定程序需要的内存大小是有限的，可以考虑预先分配一定的内存池或缓存池，以避免内存块太小的问题。此外，如果程序只需要在某些特定的场景下使用内存，可以通过惰性初始化等方式来避免在程序启动时分配大量的内存空间。

26、你知道程序可执行文件的结构吗?

- 头部信息：包含
- 代码段：存放程

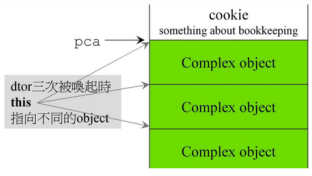
- 数据段：存放程序的静态变量和全局变量，包括可读写数据和只读数据，通常是程序中定义的变量和常量。
- 栈：存放函数的局部变量和函数调用的上下文信息，以及函数参数等信息。栈的大小在程序运行时动态变化，通常由操作系统或者运行时库进行管理。
- 堆：存放动态分配的内存，由程序通过malloc或new等操作进行申请和释放。

在不同的操作系统和编译器下，程序可执行文件的结构可能会有所不同，但通常包含以上几个部分。

发布于 2023-12-18 11:07 · IP 属地北京

内存管理　C++　程序员面试

推荐阅读



C++动态内存管理与源码剖析

linux　发表于linux...

C++内存管理效率问题

内存管理可以分为三个层次自底向上分别是：第一层：操作系统内核的内存管理-虚拟内存管理第二层：glibc层维护的内存管理算法第三层：应用程序从glibc动态分配内存后，根据应用程序本身的程...

沃德锅　发表于c/c++...

浅谈C++内存管理

吉良吉影

【笔记】C++的内栈

内存管理各区块介绍存分为5个区：堆、区、全局/静态存储区。 栈：是由编译器分配，不需要时自动储区。通常存放局部

诸葛灬孔暗