目录

C++智能指针

在 C++中，动态内存的管理是用一对运算符完成的：new 和 delete

new:在动态内存中为对象分配一块空间并返回一个指向该对象的指针

delete：指向一个动态独享的指针，销毁对象，并释放与之关联的内存。

动态内存管理经常会出现两种问题：一种是忘记释放内存，会造成内存泄漏；一种是尚有指针引用内存的情况下就释放了它，就会产生引用非法内存的指针。为了更加容易（更加安全）的使用动态内存，引入了智能指针的概念。智能指针的行为类似常规指针，重要的区别是它负责自动释放所指向的对象。

share_ptr 类允许多个指针指向同一个对象，使用 make_shared 标准库函数进行初始化。每个 share_ptr 都有一个关联计数器，称为引用计数器。每拷贝一个 share_ptr 都会使计数器加 1，当 share_ptr 被赋一个新值或者被销毁时都会使计数器减 1，一旦计数器为 0，则调用析构函数释放管理的对象。

```
1.  // 指针声明
2.  shared_ptr<string> p1;
3.
4.  // 指针初始化
5.  shared_ptr<int> p3 = make_shared<int> (42);
6.  shared_ptr<string> p4 = make_shared<string> (10,'9');
7.
8.  // 指针拷贝
9.  auto p = make_shared<int> (42);
10. auto q(p);
```

unique_ptr 类规定某个时刻只能有一个指针指向给定对象，所以不支持赋值和拷贝操作，但是可以通过调用 release 或者 reset 将指针的所有权从一个 unique_ptr 转移到另一个 unique_ptr。其中 release 返回保存的指针并将其置空，调用 release 会切断 unique_ptr 和原来管理的对象之间的联系。

```
1.  //将所有权从 p1（指向 string Stegosaurus）转移给 p2
```

```
2.  unique_ptr<string> p2(p1.release()); //release 将 p1 置为空
3.  unique_ptr<string>p3(new string("Trex"));
4.
5.  //将所有权从 p3 转移到 p2
6.  p2.reset(p3.release()); //reset 释放了 p2 原来指向的内存
```

weak_ptr 是为了配合 share_ptr 而引入的一种智能指针，因为其不具备普通指针的行为（即没有重载*和->符号），所以其最大的作用在于协助 share_ptr 工作，可以观测资源的使用情况。可以使用 use_count 来查看资源的引用计数，lock 方法从被观测的 share_ptr 中获得一个可用的 share_ptr 管理的对象，从而操作资源。

● static 关键字的用法

C 中的 static：

1. 静态局部变量，生命周期比函数长，只在第一次调用时进行初始化。
2. 静态全局变量，只在本文件内可见（即无法通过 extern 来获取）
3. 静态函数，只在本文件内可见

C++中 static 的额外功能：

1. 静态成员变量。非静态成员变量每个对象都有自己的拷贝，而静态数据成员每个对象要共享。
2. 静态成员函数。可以通过“类名::函数名”进行静态调用，静态方法只能访问静态变量或方法，非静态可以访问静态和非静态。

● const 关键字的用法

1. 定义常量。
2. 修饰常量和指针。（难点在于：区分指针常量和常量指针）
3. 修饰函数传入的参数。该种情况下，函数内部不能修改参数值。
4. 修饰函数的返回值。该种情况下，用户无法修改返回值。
5. 修饰成员函数。该种情况下，const 成员函数无法修改对象的数据。

● const 和#define 定义常量的区别

1. 数据类型

const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查，而后者只是字符替换，没有类型安全检查。

2. 调试

一些调试工具可以对 const 进行调试，但是不能对宏变量进行调试。

3. 存储方式

宏定义是直接进行字符替换的，所以不会分配内存。const 常量会进行内存分配。

4. 函数参数

宏定义无法作为参数传递给函数，const 常量可以做参数传递。

5. 取消定义

宏定义可以通过#undef 来使之前的宏定义失效，const 常量定义后在定义域内永久有效。

● 区分指针常量和常量指针

指针自身是一个对象，它的值是一个整数，表明指向对象的内存地址。因此指针的长度和所指向对象的类型无关，在 32 位系统下是 4 个字节，在 64 位系统下式 8 个字节。因此，指针本身是否为常量和所指向的对象是否为常量就是两个独立的问题。

常量指针，又叫做“顶层 const”，表示指针本身是一个常量。

指针常量，又叫做“底层 const”，表示指针指向的对象是一个常量。

```
1. int a = 1;
2. int b = 2;
3. const int* p1 = &a;
4. int* const p2 = &a;
```

上述代码中，变量的定义从右往左读，就可以区分指针常量和常量指针。

第 3 行从右往左读为：p1 是一个指针(*)，指向一个 int 型对象(int)，这个对象是个常量(const)，因此 p1 是一个指针常量。

第 4 行从右往左读为：p2 是一个常量(const)，p2 是一个指针(*)，指向一个 int 型对象(int)，因此 p2 是一个常量指针。

● 四种强制类型转换

static_cast<type-id> (expression)

1. 用于类层次结构中基类与派生类之间指针或引用的转换，进行上行转换时是安全的，进行下行转换时由于没有动态类型检查，所以是不安全的

2. 进行基本数据类型之间的转换，如把 int 转换为 char，这种转换的安全性需要开发人员来保证

3. 把空指针转换成目标类型的空指针

4. 把任何类型的表达式转换为 void 型

const_cast<type-id> (expression)

1. 用于强制去掉不能被修改的常数特性，即去掉指向常数对象的指针或者引用的常量性。

reinterpret_cast<type-id> (expression)

1. 改变指针或引用的类型

2. 将指针或引用转换成一个足够长度的整形

3. 将整形转换为指针或引用类型

dynamic_cast<type-id> (expression)

1. 其他三种都是编译时完成的，dynamic_cast 是在运行时完成的，要进行类型检查

2. 不能用于内置的基本数据类型之间的转换

3. dynamic_cast 要求<>中的目标类型必须是指针或者引用，转换成功后返回指向类的指针或者引用，失败返回 nullptr

4. 在类进行上行转换时效果和 static_cast 一致。在进行下行转换时具有类型检查的功能，即要转换的指针指向的对象的实际类型与转换以后的对象类型一定要相同，否则会转换失败。

5. 使用该强制转换时，基类中一定要有虚函数（有虚函数说明想要让基类指针或者引用指向派生对象，这样的转换才有意义），这是由于运行时类型检查需要运行时类型信息，而这个信息存储在虚表中。

- new/delete 和 malloc/free 的区别

1. new/delete 是 C++中的关键字/操作符，而 malloc/free 是 C/C++中的标准库函数

2. 使用 new 来申请内存时会自动计算所需要的内存大小，而 malloc 需要显式地指定内存大小

3. new 分配内存成功时，返回的是对象类型的指针，而 malloc 分配内存成功时，返回的是 void*，需要转换成所需要的类型

4. new 在分配失败的时候会抛出 bac_alloc 异常，而 malloc 分配失败时会返回 NULL

5. new 从自由存储区上进行内存空间的分配，而 malloc 从堆上进行内存空间的分配

6. new 能够触发构造函数的调用，malloc 仅分配空间；delete 能够触发析构函数的调用，free 仅归还申请的空间

7. new/delete 因为是操作符，所以可以进行重载，而 malloc/free 不能进行重载

数据库

- DB 加快查询的方法

数据库设计方面：

1. 避免全表扫描，首先考虑在 where 及 order by 涉及的列上建立索引，但是也不要建立过多的索引，因为会降低 insert 和 update 的效率，这个过程中可能会重建索引。

2. 尽可能避免更新索引列，因为索引列的顺序就是表记录的物理存储顺序，一旦改变会导致整个表记录顺序的调整。

3. 只含有数值信息的字段尽量使用数值型而不是字符型，因为字符型需要逐个字符进行对比，而数值型只比较一次。

4. 尽可能使用 varchar 代替 char，因为变长存储可以节省空间。

SQL 语句方面：

1. 尽量避免在 where 子句中使用!=或<>操作符、使用 or 来连接条件、对字段进行 null 值判断，否则引擎会放弃使用索引而进行全表扫描。

2. select 中选择需要使用的字段而不是*。

● DB 事务隔离级别

数据库中的三种 BUG：

1. 脏读。事务 A 读取到事务 B 修改了但未提交的数据。

2. 不可重复读。事务 A 中两次相同的查询之间，B 事务修改了某个值并提交，导致 A 的两次查询结果不同。

3. 幻读。事务 A 在读取某个范围的数据，但事务 B 向这个范围中插入数据，导致 A 的多次读取数据行数不一致。

四种隔离级别：

1. 读未提交

指一个事务读取到另一个事务还未提交的内容。这种隔离级别下查询不加锁，是最差的级别，可能会产生“脏读”问题。

2. 读已提交

事务 A 只能读取到由其他事务修改并提交之后的数据，可以避免“脏读”的产生。可能会产生“不可重复读”问题。

3. 可重复读（MySQL 的默认隔离级别）

在一个事务使用某行的过程中，不允许别的事务在对这行数据进行操作，即加上了行锁。可能会产生“幻读”问题。

4. 串行化

事务完全串行执行，失去并发的效率。

● MySQL 中的锁

1. 行级锁

对当前操作的行进行加锁，其加锁粒度最小，加锁开销最大，发生锁冲突的概率最小，并发度最高，会出现死锁。分为共享锁和排它锁。

共享锁又叫读锁，获取共享锁的事务只能读取数据，不能修改数据。如果一个事务对数据加上了共享锁，其他事务只能再加共享锁，不能加排它锁。

排它锁又叫写锁，获取排它锁的事务既能读取数据也能修改数据，如果一个事务对数据加上了排它锁，其他事务不能再加任何锁。

2. 表级锁

对当前操作的表进行加锁，其加锁粒度最大，加锁开销最小，发生锁冲突的概率最大，并发度最小，不会出现死锁。

3. 页级锁

对行级锁和表级锁的一种折中方法，一次锁定一组记录。

面向对象设计

● 类设计的基本原则

1. 单一职责原则

一个类，应当只有一个引起他变化的原因。一个类应该只有一个职责，其职责越少，对象之间的依赖关系就越少，耦合度就越低。

2. 开闭原则

在面向对象的设计中，应该遵循“对扩展开放，对修改关闭”的原则，“对扩展开放”是指应该在不修改原有模块的基础上扩展其功能。

3. 里氏替换原则

所有引用父类的地方必须能透明地使用其子类对象，即只要父类出现的地方子类就可以出现，而且替换为子类不会发生任何异常。即子类可以扩展父类的功能，但是子类不能改变父类原有的功能。

该原则为良好的继承定义了一个规范：子类必须完全实现父类的方法、子类可以有自己的个性、覆盖或者实现父类方法时输入参数可以被放大、覆盖或者实现父类方法时输入参数可以被缩小。

4. 依赖倒置原则

高层模块不应该依赖于低层模块，两者都应该依赖于抽象；抽象不依赖于细节，细节应该依赖抽象；面向接口编程，而不是面向实现编程。

在高层模块和低层模块之间增加一个抽象接口层，高层模块直接依赖于抽象接口层，抽象接口层不依赖于低层模块的实现细节，而是低层模块依赖抽象接口层。一般来说，类与类之间都通过抽象接口层来建立关系。这样一来，低层的实现细节即便发生变化，高层模块调用的接口也不会发生其他什么变化。

5. 接口分隔原则

不能强迫用户去依赖那些他们不使用的接口。

接口的设计应该遵循最小接口原则，不要把用户不使用的方法都塞进一个接口里面。如果一个接口的方法没有被使用到，说明该接口应该被分割为多个功能专一的接口。该原则指导我们：类与类的依赖应该建立在最小的接口上；建立单一的接口，而不是庞大臃肿的接口；尽量细化接口，接口中的方法尽可能少。

6. 迪米特法则

又叫“最少知道”原则，只与你的朋友通信，不和陌生人说话。

可以解释为：一个对象应该尽可能少得了解其他对象，不关心对象内部如何处理。该原则的主要目的在于降低类之间的耦合，容易使系统的功能模块相互独立。

计算机网络

● http 请求的三次握手过程

第一次握手：客户端向服务端发送连接请求报文，进入“同步已发送”状态。

第二次握手：服务端收到后，向客户端发送确认报文，进入“同步收到”状态。

第三次握手：客户端收到后，向服务端发送确认报文，进入“已建立连接”状态。

通俗地讲：

第一步：client——喂，你听得到吗

第二步：server——我听得到你，你听得到我吗

第三步：client——我听得到你

● 为什么不在两次握手后就开始发送数据

考虑一种情况：client 向 server 发送了连接请求报文，但是由于网络原因在中途某个网络节点滞留了，过了很久才到达 server，这本来应该是一个失效的报文，但是 server 收到这个报文后还是会向 client 发送确认报文，表示同意连接。如果没有第三次握手，只要 server 发出确认就表示连接建立，但是这已经是一个失效的连接了，client 不会发送任何信息，而 server 一直在等待 client 发送数据，于是就浪费了很多资源。第三次握手就是为了防止这种情况的发生。

● http 请求的四次挥手过程

第一次挥手：client 发送完数据后，向 server 发送连接释放报文，关闭从 client 到 server 的数据传送。

第二次挥手：server 收到后，向 client 发送确认报文，表示知道 client 数据发送完毕了，但是自己还没有发送完毕。

第三次挥手：server 发送完数据后，向 client 发送连接释放报文，关闭从 server 到 client 的数据传送。

第四次挥手：client 收到后，发出确认报文。（server 收到此确认报文后可以直接进入关闭状态）此时 TCP 连接没有释放，client 经过 2MSL（最长报文段寿命）时间后，撤销相应的 TCB 后，进入关闭状态。

通俗地讲：

第一步：client——我数据发送完了，我要关闭连接了

第二步：server——我知道了，等我发完再关

第三步：server——我发完了，我也要关闭连接了

第四步：client——好的，我知道了

- 为什么建立连接需要 3 次，释放连接需要 4 次

因为当建立连接时，server 收到 client 的建立连接请求时，可以直接发送 syn 和 ack 报文给 client 表示同意建立连接。而到了释放连接时，server 收到 client 的释放连接请求时，仅仅代表 server 知道 client 的数据发送完了，但是 server 的数据还没有发送完，所以先发送一个 ack 表示知道 client 数据发送完毕。等到 server 数据发送完毕之后，再发送连接释放请求。所以两个报文一般会分开发送。

- 为什么 client 发送确认报文后要再等待 2MSL 时间后再关闭连接

第一个原因：为了保证 client 的确认报文能到达 server。在 server 的角度来看，已经发送了断开连接请求，但是 client 没有给任何回复，可能这个报文 client 没收到，于是 server 会再发一次断开连接请求。client 会在 2MSL 时间内收到这个重传的报文，接着给出回应报文，重启 2MSL 时间。

第二个原因：为了避免类似于“三次握手”中已经失效的请求情况出现，在 2MSL 时间内可以使产生的所有报文段都从网络中消失，就不会在新连接中出现旧请求。