目录

C++

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。智能指针就是一个类，当超出了类的作用域时，类会自动调用析构函数，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，不需要手动释放空间。

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

- 运行时内存分配

内存分为五个区：

栈区：由编译器自动分配和释放，存放函数的参数值，局部变量的值。由高地址向低地址生长。

堆区：一般由程序员使用 new 进行分配，使用 delete 进行释放。由低地址向高地址生长。

全局区/静态区：存放全局变量和静态变量（编译器会自动赋值，不存在未初始化的）

常量区：存放不能修改的常量

自由存储区：由程序员使用 malloc/realloc/calloc 分配空间，由 free 释放，与 C 的堆对应

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。

● 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。

● http 请求中 get 和 post 的区别

PUT、DELETE、POST、GET 分别对应数据库操作中的增、删、改、查

1. Get 请求通常用于向服务器请求某个资源，Post 方法向服务器提交数据，用于修改和写入数据。

2. GET 产生一个 TCP 数据包，POST 产生两个 TCP 数据包（GET 会将 header 和 data 一并发送出去，然后服务器响应 200。POST 会先发送 header，服务器响应 100，再发送 data，服务器响应 200）

3. GET 的请求会作为 URL 的一部分，直接暴露出来，POST 会将请求信息写在 BODY 体里面。

4. GET 请求会被浏览器缓存下来，POST 不会（GET 的请求参数被完整保留在历史记录中，POST 不会）

5. GET 请求只能发送 ASCII 字符，POST 没有限制

6. 由于浏览器和 web 服务器的设置，GET 请求的 URL 长度是有限制的，而 POST 没有（HTTP 协议本身没有对长度做任何限制）

7. GET 比 POST 快（原因为第 2、3 条）

数据结构

● 堆、栈理论

stack

1. 由编译器自动分配和回收

2. 速度较快

3. 不会产生碎片

4. 线性结构

5. 由高地址向低地址生长，所以栈顶的地址和最大容量是已经规定好的

heap

1. 需要程序员手动分配和回收——C 使用 malloc 函数，C++使用 new 操作符

2. 速度较慢

3. 会产生碎片

4. 链式结构

5. 由低地址向高地址生长，大小受限于有效地虚拟内存大小

操作系统

● 进程

进程是一个程序及其数据在处理机上顺序执行时所发生的活动，是系统进行资源分配和管理的一个独立单位。每个进程都拥有自己独立的地址空间。

OS 中引入进程的目的是使多个程序能够并发执行，并且可以对并发执行的程序加以描述和控制，以此来提高资源利用率和系统吞吐量。

进程的两个基本属性：拥有一定的资源、是一个可以独立调度和分派的基本单位。

进程有三种基本状态：就绪状态、执行状态、阻塞状态。

● 线程

线程是 CPU 调度的基本单位，是为了将进程的两个属性分开，即不把作为调度和分派的基本单位也同时作为拥有资源的单位，因为进程之间进行切换会导致系统为此付出较大的时空开销，所以才形成了线程的概念。

● 进程和线程的比较

调度方面——在引入了线程的操作系统中，进程是拥有资源的基本单位，而线程是调度的基本单位，线程可以显著地提高系统的并发程度。在同一个进程中，线程的切换不会引起进程的切换。在由一个进程中的线程切换到另一个进程中的线程时才会引起进程间的切换。

并发性——在引入了线程的操作系统中，不仅进程之间可以并发执行，而且一个进程内的多个线程也可以并发执行，因此能够更有效地提升系统的利用率和吞吐量。

拥有资源——不管是传统的操作系统还是引入了线程的操作系统，进程都是拥有资源的基本单位。线程不拥有系统资源，但是它可以访问它隶属的进程的所有资源。

系统开销——由于在创建和销毁进程时，系统要为之分配或者回收资源，所以线程之间的切换开销要远远小于进程之间的切换

独立性——每个进程都拥有一个独立的地址空间，除了一些全局变量之外，不允许其他进程访问。但是一个进程内部的不同线程是共享进程的内存地址空间和资源的。

- 进程同步

进程同步是进程之间的直接制约关系，指为完成任务而建立的多个进程，这些进程需要协调它们之间的工作次序而等待、传递信息所产生的制约关系。例如写进程和读进程需要存在次序关系，写完了才可以读。

进程同步机制的主要任务是：对多个相关进程在执行次序上进行协调，使并发执行的多个进程之间能按照一定的规则共享系统资源，从而使程序的执行具有可再现性。如果对共享变量或数据结构等资源有不正确的访问次序，会造成进程每次执行结果的不一致。

- 进程互斥

进程互斥是进程之间的间接制约关系，当一个进程进入临界区使用临界资源时，另一个进程必须等待，当占用临界区的进程退出临界区之后，其他进程才可以进入临界区使用临界资源。

- 进程同步和互斥的机制

信号量机制——信号量的数据结构是一个值和一个指针，该值与相应资源的使用情况有关，当它大于 0 时表示可用资源的数量，当它小于 0 时绝对值表示正在等待使用该资源的进程个数。利用信号量和 PV 操作是典型的同步机制。

管程机制——由于信号量使用起来操作分散、难以控制。所以提出了管程这个概念，利用共享数据结构抽象得表示系统中的共享资源，并将对共享数据结构实施的特定操作定义为一组过程，进程对共享资源的申请必须通过这组过程，因此就间接地对共享资源进行了管理。

- 进程通信机制

共享内存——操作系统内核维护一个共享的内存区域，其他进程把自己的逻辑地址映射到这块内存上，多个进程之间就可以共享这块内存了。优点是不需要信息复制，是最快的通信方式。缺点是会面临同步的问题。

消息队列——存放在内核中的一个消息队列，进程将通信的数据封装在消息中，通过操作系统进行进程间的消息传递。不同的进程可以访问这个队列。

无名管道——连接一个读进程和一个写进程以实现通信的一个共享文件。数据只能在单方向上流动，即具有固定的读端和写端。只能用于有亲缘关系的进程之间通信，如父子进程和兄弟进程。不属于文件系统，只存在于内存中。

命名管道——允许在没有亲缘关系的进程之间通信。存在于文件系统中。

套接字——不同机器之间的进程通信。

信号量

● 同步、异步、阻塞、非阻塞

同步和异步：与消息的通知机制有关，指的是需不需要等待返回结果，是相对于操作结果来说的。

同步——需要不断轮询数据是否已经准备好，或者一直等待数据准备好。

异步——发送一个请求后立即返回，去干别的事情，当数据准备好了之后会通知进行相关处理。

区别：请求发出后是否需要等待结果，才能执行其他操作。

同步的实时性比较好，异步的并发性比较好。

阻塞和非阻塞：是指需不需要阻塞线程，是相对于线程是否被阻塞来说的。

阻塞：当前线程不执行别的事情，一直等待。

非阻塞：当前线程可以执行别的事情，每隔一段时间之后检查数据是否准备好。

例子：

老王烧水，有两种水壶——普通水壶、响水壶（水开之后发出声响）

同步阻塞——老王用普通水壶烧水，并且站在旁边，每隔一段时间看看水是否烧开。

同步非阻塞——老王用普通水壶烧水，然后去客厅看书，每隔一段时间来看看水是否烧开。

异步阻塞——老王用响水壶烧水，并且站在旁边，不用每隔一段是件看水是否烧开，因为烧开后会有响声告诉他。

异步非阻塞——老王用响水壶烧水，然后去客厅看书，水烧开后发出响声告诉老王。

- 死锁的定义

如果一组进程中的每一个进程都在等待仅由该组进程中的其他进程才能引发的事件，那么该组进程是死锁的。

- 引起死锁的原因

1. 竞争不可抢占性资源引起死锁——P1 和 P2 都准备些两个文件 F1 和 F2，P1 打开 F1 的同时 P2 打开 F2，这两个进程都因为另一个文件已经被打开而阻塞，形成死锁。

2. 竞争可消耗资源引起死锁——

3. 进程推进顺序不当引起死锁——P1 保持资源 R1 申请 R2，P2 保持资源 R2 申请 R1，形成死锁。

- 产生死锁的必要条件

产生死锁必须同时具备以下四个条件，只要一个不成立就不会发生死锁。

互斥条件——进程对资源进行排它性使用，其他进程如果请求该资源只能等待。

请求和保持——进程已经保持了至少一个资源，然后请求其他新的资源，该资源被其他进程占有，于是该进程被阻塞，但是自己已经申请的资源保持不放。

不可抢占——进程已经获得的资源在使用完之前不能被抢占，只能用完后自己释放。

循环等待——发生死锁时必然存在一个进程--资源的循环链。

- 预防死锁

既然死锁的产生必须具备四个条件，那么打破除“互斥使用”外任意一个条件都可以防止死锁的发生。

破坏“请求和保持条件”——1）规定进程开始运行之前，必须一次性申请运行过程中需要的全部资源，如果此时系统有足够的资源可以分配，则分配给它运行。（设备利用率低、容易造成进程饥饿）2）允许进程只获得初期所需要的资源后就开始运行，运行过程中再释放用完的资源并申请新的资源。

破坏“不可抢占条件”——规定当某个进程在保持了某些不可抢占的资源后，申请新的资源不能得到满足时，必须释放已经保持的所有资源。

破坏“循环等待条件”——对所有资源类型进行线性排序，规定每个进程必须按照序号递增的顺序来请求资源。如果一个进程需要使用多个资源，必须先申请序号小的，再申请序号大

的。如果一个进程已经申请到了一些高序号的资源又要请求一个低序号的资源，必须先释放高序号资源后才可以申请低序号资源。这种策略不可能出现环路，因为总有一个进程占据较高序号的资源，此后他申请的所有资源都是空闲的。

- 为什么需要虚拟内存

各种存储器的管理方式有一个共同的特点：将一个作业全部装入内存后才能运行，于是出现了两种情况：

1. 有的作业很大，所要求的的内存空间超过了内存总容量，无法全部装入内存运行

2. 有大量作业要求运行，内存容量不足以容纳所有这些作业，只能将少数放入内存先运行，将其他大量的留在外存上等待。

解决这两个问题的方法有两种：一种是从物理上增加内存容量；另一种是从逻辑上扩充内存容量，这正是虚拟存储要解决的问题。

- 局部性原理

时间局限性——如果某条指令被执行，那么不久以后该指令可能被再次执行；如果某数据被访问过，那么不久以后该数据可能被再次访问。

空间局限性——一旦程序访问了某个存储单元，在不久之后其附近的存储单元也将被访问。

- 虚拟存储器

虚拟存储器是指具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储器系统。

应用程序在运行之前没有必要将其全部装入内存，仅需要将当前运行的少数页面或段先装入内存便可运行。程序要在运行时如果他所需要的页或者段已经调入内存，便可继续执行下去；如果尚未调入内存，便发出缺页中断，OS 利用请求调页功能将他们调入内存。如果此时内存已满，则 OS 利用页的置换功能，将内存中暂时不用的页调至盘上，腾出空间后再调入所需要的页。这样可以使一个大的用户程序在较小的内存空间中运行，内存中也就可以装入更多的进程，使其并发执行。

Hadoop 和 Spark 的区别和联系

Hadoop 实质上是一个分布式数据基础设施，将巨大的数据集存储在集群的多个节点上，Spark 是一个专门的对那些分布式存储的大数据进行处理的工具，并不会进行分布式存储。

Hadoop 除了提供分布式存储之外，还提供了叫做 MapReduce 的数据处理功能，可以直接进行数据处理。Spark 只是一个数据处理工具，必须依托于某个分布式文件系统才能运作，默认是被用在 Hadoop 上的。

Spark 的处理速度比 Hadoop 快。Hadoop 的中间结果都会写入磁盘，然后再从磁盘读取进行下一次处理，而 Spark 的中间结果都在内存中，做完所有处理后才会将结果写入磁盘（实时性强）。

为什么要有 Shuffle 阶段

在 Hadoop 这样的集群环境里，map 和 reduce 通常工作在不同的节点上，所以 reduce 经常要跨节点去 map 上拷贝结果数据。这个过程对集群内部网络资源的消耗很严重。所以如果能在 map 的输出结果上做一些工作，就可以在数据拉取过程中减少不必要的消耗。

Shuffle 阶段的详细过程

MapReduce 过程中，map 阶段的数据如何传递给 reduce，是一个很关键的流程，这个流程叫做 shuffle。主要的工作任务是：对数据进行分区、排序、缓存。

map 端的 shuffle——partition、spill(sort & combiner)、merge

每个 map 任务都有一个环形缓冲区来存储任务输出，默认为 100MB。当达到某个阈值时（如 0.8）就会触发溢写（spill）：一个后台线程会将缓冲区内容写到磁盘。写磁盘之前线程会根据数据最终要传到的 reduce 将数据划分成不同的分区（partition），然后在每个分区内按键进行内排序，如果有 combiner 过程则可以使结果更紧凑（如 wordCount 中将同一个词对应的所有次数累加起来作为一条数据），减少数据量的大小。每一次内存缓冲区到达溢出阈值都会新建一个溢出文件，当 map 任务完成之后会有多个溢出文件，这些文件将会被合并为一个已分区且已排序的文件（merge），此过程如果有 combiner 则会合并来自不同溢出文件的 Key，减少数据的大小。

reduce 端的 shuffle——copy、merge

当有一个 map 任务结束之后，reduce 就可以对该任务的输出结果进行拷贝（copy）。如果 map 的输出结果很小，则会被复制到 reduce 的内存缓冲区中，当缓冲区大小到达阈值时合并后溢写到磁盘上。随着磁盘上文件越来越多，后台线程会将其合并为更大的、排好序的文件，如果有 combiner 过程则可以合并不同文件的 key，减小数据的大小。复制完所有的 map 结果后，对已排序输入中的每个键调用 reduce 函数，输出直接写入文件系统。