在语言层面上：面试 C++程序员的时候一般都是 3 板斧：

首先是基础问答

然后就是如虚函数、虚函数表、纯虚函数、抽象类、虚函数和析构函数、虚函数和构造函数，以及拷贝构造函数、操作符重载和 STL，内存管理，malloc/free 与 new/delete 等问题，

最后是 C++11 新特性比如智能指针，RAII，左值与右值等

++i 和 i++的主要区别有两个：

a) **i++ 返回原来的值，++i 返回加 1 后的值。**

b) **i++ 不能作为左值，i++最后返回的是一个临时变量。而++i 可以。**

++i 和 i++的效率哪个更高？

++i 是在原地操作，效率更高；i++选哟申请临时变量，++后再赋值回去，效率低(需要申请临时空间存储临时变量)

1. C++相关

1.1 左值和右值

1） 左值是有名字的变量(对象)，可以被赋值，可以在多条语句中使用。左值引用声明符号为 & 。

2） 右值是临时变量，没有名字，只在一条语句中出现，不能被赋值，右值声明符号为 &&。

(左值：int &a = 1，右值：int &&a=1)

凡是真正的存在内存当中，而不是寄存器当中的值就是左值，其余的都是右值。其实更通俗一点的说法就是：凡是取地址（&）操作可以成功的都是左值，

其余都是右值。

左值[lvalue : locator value]: 存储在内存，有明确地址的值

右值[rvalue : read value]：提供数据的值，不一定可寻址；比如寄存器中的数据。右值又可以分为纯右值和将亡值

a) 有名称的，可以获取存储地址的值即为左值，否则为右值

b) 左值引用 T&　右值引用 T&&

c) 同左值引用 T&一样，右值引用 T&& **创建时就必须初始化且只能使用右值进行初始化**

左右值最大的区别就是**是否可以取地址**，左值可以，右值不可以。

右值引用(T&&)主要是为了用于**移动语义**和**完美转发。**

1.2 右值引用的意义

1） 右值主要用于 移动语义 和 完美转发

2） 为临时变量延长生命周期。因为右值在表达式结束后就消亡了，如果想继续使用右值，那就会动用（昂贵的）拷贝构造函数。

3） 移动语义：将资源从一个对象转移到一个对象，避免不必要的拷贝，主要是将**所有权转出去，避免申请空间和数据复制来提高效率**。

a) 右值引用可以消除两个对象交互时的不必要的对象拷贝，节省运算存储资源，提高效率。

b) 右值引用是用来支持**转移语义**的。转移语义可以将资源（堆，系统对象等）从一个对象转移到另一个对象，这样能够减少不必要的临时对象的创建、拷贝和销毁，能够大幅度提高 C++应用程序的性能。

4） **完美转发**：函数模板可以将自己的参数 完美 的转发其内部调用的函数

a) 完美的意思是 不仅准确的转发参数的值，而且还能保证转发参数的左、右值属于不变。

b) 内部调用时使用模板函数 forward()，forward()函数用于转发，依赖于类型 T

5） std::forward() 有两种实现，分别对左值引用和右值引用使用不同

的模板函数，内部调用 static_cast。

6） std::move() 将传入的参数转为右值，std::move()实际上是 static_cast<T&&>()的简单封装。std::move()并不会 move，只是做了类型转化而已，左值的移动操作时在移动构造或移动赋值完成的。

```
template <typename T>
typename remove_reference<T>::type&& move(T&& t)
{
    return static_cast<typename remove_reference<T>::type&&>(t);
}
```

1.3 emplace_back()和 push_back()区别

a) emplace_back()和 push_back()都是在容器尾部添加元素是操作，但是它们的效率是不同的。

b) 在 c++11 引入右值引用之前，push_back() 是先调用构造函数创建一个临时对象，然后调用拷贝构造函数将临时对象拷贝到容器中，最后释放临时对象，有一个申请空间，拷贝，和释放空间的过程，导致临时变量申请资源的浪费。

c) 在引入右值引用之后，则是创建临时对象，然后使用移动构造函数，将里临时对象加入对象。从而避免了一次拷贝过程和开辟空间的消耗，在原来的基础上提高效率。

d) emplace_back()则是在这基础上，进一步改进，直接在容器中就地构造，没有赋值和移动操作，进一步提高效率。

1.4 C++中 struct 与 class 区别,C 与 C++中 struct 区别 C++中:

1） 在 c 中，struct 主要是正对数据结构，是面向对象的设计，struct 中不能包含函数。

2） c++中，struct 得到很大的扩充，作用几乎和 class 类似

a） struct 可以包含成员函数

b） struct 可以实现继承

c） struct 可以实现多态

3） 在 c++中 strcut 和 class 的区别：

a） 默认权限不同，struct 默认是 public，class 默认是 private。哪种继承取决于子类而不是基类，也就是说 struct 可以继承 class，class 也可以继承 struct

```
struct A{};
class B : A{}; //private 继承
struct C : B{};    //public 继承
```

b） class 关键字可用于定于模板参数，类似 typename。struct 不能。

1.5 C++和 C 相比的最大特点：

a） 面向对象，主要特点有**封装**，**继承**，**多态**

b） 引入了引用替代指针

c） const/inline/template 替代宏常量

d） namespace 解决重名问题

名字空间是用来划分冲突域的，把全局名字空间划分成几个小的名字空间。全局函数，全局变量，以及类的名字是在同一个全局名字空间中，有时为了防止命名冲突，会把这些名字放到不同的名字空间中去。

e） STL 提供高效的数据结构和算法

1.6 C++面向对象的三大特性：

1） 封装：将代码模块化，将其封装起来，对外隐藏，解决可扩展性。

2） 继承：子类继承父类，实现了代码的可重用。

3） 多态：一种接口，多种形态，虚函数是多态的重要实现方式。

作用： 一种接口，多种方法【不同对象对同一行为会有不同的状态,实现接口重用】

实现形式： 静态多态 和 动态多态

静态多态： 通过 函数重载 和 函数模板 来实现

同一函数，其参数的个数，顺序，类型不同来实现重载

在编译时就已经确定对象的行为，又称为静态绑定。

动态多态：是面向对象的一大特色，通过继承方式使得程序在运行时才确定相应的调用函数，又称为动态绑定。通过 虚函数 来实现

在运行时确定具体调用那个函数。

1.7 虚函数及纯虚函数的理解

1） 虚函数

虚函数作用： 实现多态

实现原理： 虚函数表（vtable）+ 虚表指针(vptr)

虚表指针指向一个由虚函数的地址构成的一个数组（指针数组），含有虚函数的类有一个虚函数表，每个类对象由一个虚表指针，指向虚函数表

虚函数表是存放着类中所有虚函数对象的函数指针的表，虚函数表类似一个数组，存储的是虚函数的地址，也即是虚函数的原始是指向类成员函数的指针。虚函数表的实现是编译器来实现的，编译器不同，可能实现方式不同

虚函数表是属于类的，一个类的所有对象共享这个虚函数表

2） 纯虚函数

形式：virtual bool function() = 0;

纯虚函数就是定义了一个虚函数但是没有实现，且在原型后面加了”=0”，包含纯虚函数的类是抽象类，抽象类不能实例化对象，但是可以右指针和引用对象。

3）构造函数可以是虚函数吗？

不可以，原因如下：

1.构造一个对象的时候，必须知道对象的实际类型，而虚函数是在运行期间确定实际类型的。但在构造一个对象时，由于对象还未构

造成功，编译器无法知道对象的实际类型，比如是该类本身还是派生类，因为无法确定。

2.虚函数的执行依赖于虚函数表，而虚函数表在构造函数中进行初始化，也即是初始化 虚函数指针 vptr,让它指向虚函数表。但在构造对象期间，虚函数还没有被初始化。

4) 构造函数/析构函数中可以调用虚函数吗？

1.从语法上来讲，这样调用是可以的。

2.但是从效果上来看，往往达不到需要的效果

(可以，但是没有意义，没有实现动态绑定的效果)

正如 effective C++ 条款 09 所讲： 绝对不要在构造函数或析构函数中调用虚函数。

在继承体系中，构造的顺序 ：

基类构造函数 ->对象成员构造函数->派生类的构造函数

析构的顺序刚好相反

父类先于子类被构造，当父类的构造函数被调用时，子类还没有形成，此时子类对于编译器而是是不存在，所以调用的仍然是父类本身的虚函数。

因为派生类对象构造期间进入基类的构造函数时，对象的类型变成了基类对象，而不是派生类对象，同样，进入基类析构函数时，对象也是基类类型。

5）虚继承和虚基类

虚继承是为了解决多重继承出现菱形继承的问题， 比如 A 是一个基类，B，C 分别继承了 A,D 类多重继承 B，C 类， 则 D 中会出现两份 A 类数据。在进程时，通过在继承关系前加上 virtual 关键字即可实现虚继承。

虚基类是用 virtual 关键字声明继承的父类，即该基类在多条链路上被一个子类继承，但是该子类中只含一份该虚基类的数据。主要用来解决继承中的二义性问题。在同一层次中同时包含虚基类和非虚基类，那么先调用虚基类的构造函数，在调用非虚基类的构造函数，最后调用派生类的构造函数，析构则相反；对于多个虚基类，则构造函数执行顺序从左到右；

1.8 const 与 #define 区别

1. 编译器处理阶段不同

#define 是宏定义，在预处理阶段就进行文本替换。

const 定义的是只读常量，在编译阶段处理。

2. 类型安全检查不同

#define 是宏定义，不会进行类型检查，只是简单的文本替换。

const 定义的是常量，在编译阶段会进行类型检查。

3. 存储方式不同

#define 不分配内存，而是在替换是对变量分配内存，因而在内存中可能存在多份拷贝。

const 常量在内存分配内存，存储在静态变量区，只存在一份拷贝。

4. 能否调试

#define 定义的常量不能调试。

const 定义的常量是可调试的。

1.9 C++11 的特性

1) nullptr ：

1. 新引入的空指针，NULL 本质是一个宏定义，本质是 0,可能会导致二义性

eg： void fa(char* p); void fb(int n);

2) 类型推导： auto 和 decltype

auto 一般用于变量类型， auto 使用时就必须初始化，在使用迭代器的时候比较方便

decltype 用于表达式类型推导 ： 编译器期间完成，只进行类型推导，不会实际执行表达式。

3) 右值引用 见 1.1

4) 移动语义

5) 基于范围的 for 循环

6) explicit ： 修饰构造函数， 防止由构造函数定义的隐式转换。

7) const 和 constexpr

两者都表示可读，const 表明 read-only constexptr 才是真正的常量，在编译期就计算出来，整个运行期间不可改变。

const 修饰的是类型，constexpr 修饰的是用来算出值的那段代码。

8) final 和 overrid 关键字：

1. final 用于修饰一个类 表示禁止该类禁止被派生和虚函数进一步重载

2. override 修饰成员函数，用于重写基类函数，若声明为 override，但是对父类没有该虚函数或重新，编译器报错

3. override 可以避免开发者在重写基类函数时无意产生错误。

9) Lambda

1.10 RAII（Resource Acquisition Is Initialization）资源获取即初始化

RAII 是一种利用对象生命周期来控制程序资源（如内存、文件句柄、网络连接、互斥量等等）的技术。

用法：在对象构造时获取资源，使用资源，在对象析构时释放资源，借此，把管理资源的认为交给了一个对象。应用比如智能指针。

好处：不需要显示的释放资源，在对象所需的资源在其生命周期内始终保持有效，

RAII 的本质是用对象代表资源，把管理资源的认为交给对象，把资源的申请和释放与对象的构造和析构对应起来，从而确保在对象的生命周期内资源始终有效，在对象销毁时资源被释放，避免内存泄漏等情况的发生。也即使对现在，资源在，对象消失，资源释放。

1.11 智能指针

a) 普通指针容易造成内存泄漏，二次释放等问题。程序发生异常时

内存泄漏等问题，使用智能指针可用很好的管理内存，避免内存泄漏。

b) 智能指针(smart pointer)是模板类，是行为类似指针的类对象，用于生存期控制，能够确保自动正确的销毁动态分配的对象，防止内存泄露（利用自动调用类的析构函数来释放内存）。它是 C++ RAII 的一种应用。它可以使用使用引用计数(shared_ptr)和所有权模式(unique_ptr)两种方式。

c) shared_ptr：采用引用计数的指针，多个 shared_ptr 可用指向同一给对象，并维护一个共享的引用计数器。基于引用计数的智能指针类将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。每次创建类的新对象时，初始化指针并将引用计数置为 1；当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数；对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至 0，则删除对象），并增加右操作数所指对象的引用计数；调用析构函数时，构造函数减少引用计数（如果引用计数减至 0，则删除基础对象）

d) weak_ptr：shared_ptr 有一个问题即是循环引用问题，若存在循环引用，会导致 shared_ptr 的引用计数不能变为 0，从而不能释放资源。为了解决该问题，就引入了 weak_ptr，weak_ptr 是一种弱引用，用于配合 shared_ptr 的使用，并不影响对象的生命周期，也即是 weak_ptr 存在与否不影响对引用计数。

e) unique_ptr ： 基于所有权模式的智能指针，任意时刻只能有一个拥有者，也不支持拷贝构造和赋值。

1.12 C++的 this 指针

a) this 指针是类指针

b) this 指针是一个隐士的指针，作为普通成员函数的参数，由编译器来完成

c） this 指针的存储位置可能在栈，寄存器中，具体与编译器实现有关

d） this 指针只能在普通成员函数中使用，在全局函数，静态函数中不能使用

e） this 指针只有在成员函数中才有定义，不能通过对象来使用 this 指针，无法直接知道一个 this 指针的位置，但是可以在成员函数中指定 this 指针的位置。

1.13 C++的类型转换有哪些

1） static_cast：

a） 静态类型转换[编译时转换]

b） 用于各种类型的转换，

c） 在编译时完成

2） dynamic_cast

a） 用于动态类型转换

b） 只能用于含有虚函数的类，用于类层次的向和向下转换，

c） 失败返回 NULL

3） const_cast：

用于将 const 变量转为非 const【去除常量属性】

4） reinterupt_cast：几乎什么都可以转，比如 int 转为指针，但可能会出问题

5） 为什么不用 C 的强制类型转换？

a） C 的强制类型转换看起来很强大，什么都可以转，但是转化不明确，不进行错误检查，容易出错。

1.14 new 和 malloc 区别以及 malloc 原理

1） 类型不同

a） new/delete 是 c++里面引入的运算符，

b） malloc 和 free 是标准库函数。

2) 申请内存的所在位置不同

a) new 操作符是从自由存储器(free store)上为对象动态分配内存空间，自由存储区释放可以是堆(或是 new 是否能够在队上动态分配内存)，这取决于 operater new 的实现细节。自由存储区不仅仅可以是堆，还可以是静态存储区。 new 甚至可以部位对象分配内存，而是使用已分配的内存，使用 定位 new 来做到。

b) malloc()则是从堆(heap)上分配内存。堆是操作系统中的术语，是操作系统所谓五的一块特殊内存，用于程序的动态内存分配。

3) 返回类型安全性

a) new 操作成功是返回对象类型的指针，类型严格与对象匹配，无须进行类型转换。因此 new 是类型安全性的操作符

b) malloc 内存分配成功返回 void*，则需要强制类型将 void* 指针转为需要的类型。

4) 内存分配失败时的返回值

a) new 内存分配失败时，会抛出 bac_alloc 异常，而非 NULL

b) malloc 分配内存失败时返回 NULL。

5) 是否需要指定内存大小

a) 使用 new 操作符申请分配内存时无须指定内存块大小，编译器会根据类型自行计算

b) malloc 则需要显示指定所需内存的大小。

6) 对数组的处理

a) C++提供了 new[]和 delete[]来专门处理数组类型，必须配合使用。

b) malloc 只是分配一块原始内存给你，不管该空间放置的东西，也不做初始化。

7) 是否可以重载

a) operator new/delete 可以被重载

b) malloc/free 不允许重载

有了 malloc，为什么还需要 new ？

a) 对于非内置的数据类型的对象而言，使用 malloc/free 是无法满足要求的，对象在创建时要自动执行构造函数，在对象消亡钱要自动执行析构函数，而 malloc/free 是库函数，不在编译器的控制权限之内，不能够把执行构造函数和析构函数的任务强加给 mallo/free。因此，C++需要一个能完成动态内存分配和初始化的运算符和清理与释放内存工作的运算符。

b) new 操作符分配内存时会经历三个过程：

1) 第一步是调用 operator new 函数，分配一块合适的，原始的，未命名的内存空间。

2) 第二步是编译器运行相应的构造函数以构造该对象。

3) 第三步是对象构造完成后，返回一个指向该对象的指针。

特征	new/delete	malloc/free
分配内存的位置	自由存储区	堆
内存分配成功的返回值	完整类型指针	void*
内存分配失败的返回值	默认抛出异常	返回NULL
分配内存的大小	由编译器根据类型计算得出	必须显式指定字节数
处理数组	有处理数组的new版本new[]	需要用户计算数组的大小后进行内存分配
已分配内存的扩充	无法直观地处理	使用realloc简单完成
是否相互调用	可以，看具体的operator new/delete实现	不可调用new
分配内存时内存不足	客户能够指定处理函数或重新制定分配器	无法通过用户代码进行处理
函数重载	允许	不允许
构造函数与析构函数	调用	不调用

malloc 内存分配原理：

1） malloc 在申请内存时，一般会通过 brk 或者 mmap 系统调用进行申请空间。malloc 申请内存小于 128k 时，使用 brk 分配内存，申请内存大于 128k 时，使用 mmap 分配内存

2） brk 和 mmap 这两种方式分配的都是虚拟内存，没有分配物理内存；当第一次访问已分配的虚拟地址空间时，会发生缺页中断，操作系统负责物理分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。brk 与 sbrk， mmap 与 munmap

   a) brk/sbrk/mmap 属于系统调用，如果每次申请内存，都调用其中一个，系统调用开销较大，若每次都用系统调用，则会影响性能。

   b) 这样申请内存容易产生内存碎片，因为堆是从低地址往高地址方向增长，若低地址没有释放，高地址内存就不能回收。

3） 因此，malloc 采用内存池的方法，来减少内存碎片的产生和减少系统调用的开销。先申请一个大内存，然后将其分为不同大小的内存块，并以块为管理的基本单位。用户申请内存时，就直接从内存池中选择一块相近的内存块即可。

4） malloc 使用链表来管理内存块：

   a) 隐式空闲链表：将堆区分为地址连续，大小不一的块，包含已分配和未分配的内存块。

   b) 显示空闲链表：用一个链表将可用的内存块连接起来，组成一个双向的空闲链表，链表节点含有一个前向指针和后继指针，每个节点记录一个地址连续，未被使用的内存块，节点同时记录内存首地址和大小。

malloc 之 ptmlloc/tcmaclloc/jemalloc：

a) gcc 默认使用的是 ptmalloc 分配器对内存碎片进行优化，但是这种优化不理想，胆汁看似内存泄漏，实际上时内存碎片的问题。

b) 为了更好解决该问题，google 开发了 tcmalloc 和 jemalloc 内

存管理工具,比如在 Redis 中就使用的时 tcmalloc 和 jemalloc。

c) tcmalloc/jemlloc 原理很类似，都是在链接时期替换标准库的 maloc 和 free，在不改变 diamond 情况下，解决内存碎片的问题。

d) tcmalloc 就是一个内存分配器，管理堆内存，主要影响 malloc 和 free，用于降低频繁分配、释放内存造成的性能影响，且有效地控制内存碎片。

1) tcmalloc 比 ptmalloc 快，一次 malloc 和 free 操作，ptmalloc 需要大约 300ns，而 tcmalloc 只需要 50ns。

2) tcmalloc 优化了存储对象，需要的空间更少，

tcmalloc 对多线程做了优化，对于小对象的分配基本不存在锁竞争。

1.15 new、operator new 和 placement new 的区别：

1) new operator:

不能被重载，执行过程如下：

a) 调用 operator new 分配内存

b) 调用并执行构造函数

c) 返回相应类型的指针

delete 过程 ：

a) 调用析构函数

b) 调用 operator delete,释放空间

2) operator new

实现内存分配，重载时 new 时，实际上重载的是 operator new,

要实现不同的内存分配行为，应该重载 operator new

3) placement new:

operator new 只是 operator new 的一个重载版本

a) 并不分配内存，只是返回指向已分配的某段内存的一个指针

b) 因此，不能删除，但需要调用对象的析构函数。

c) placement new 允许在一个已分配的内存中构造一个新对象。

d) 引入 placement new 的理由：

用 placement new 解决 buffer 问题：

若是用 new 分配数组缓冲时，先要申请分配内存，然后调用构造函数，效率不急啊；使用 placement new 可以直接在已预分配的内存上构造函数。

增大时空效率问题：

使用 new 申请分配空间时，需要在堆中查找足够大的空间，然后分配，这个操作比较慢，同时可能出现内存不足，内存分配异常；而 placement new 是在一个预先已分配的内存缓冲区将那些，不需要查找内存，这样内存分配的时间就就是常数，且不会出现内存不足的异常。因此，placement new 非常适合对时间要求较高，长时间运行不希望被打断的应用程序。

1.16 引用和指针有什么区别

1）指针有自己的一块空间，而引用只是一个别名。

2）指针可以被初始化为 NULL，而引用必须被初始化且必须是一个已有对象的引用。

3）使用 sizeof 看一个指针的大小是 4，而引用则是被引用对象的大小。

4）指针在使用中可以指向其他对象，但是引用只能是一个对象的引用，不可被改变。

1.17 sizeof 和 strlen()的区别：

1) sizeof 是一个操作符， strlen()是一个库函数

2) sizeof 的参数可以是数据类，也可以是变量， 而 strlen()只能以结尾’\0’的字符串作为参数

3) sizeof 在编译时期就确定了结果，strlen()需要在运行时才能计算出结果

4) sizeof 计算的是分配的内存大小，strlen()则是字符串实际的长度

5) 数组作为 sizeof 参数时，不退化， 作为 strlen()参数时，则退化为指针。

1.18 产生 coredump 文件的原因

1) 访问内存越界（如使用下标访问字符串未进行结束符判断，strcpy，strcmp,strcat 等越界）

2) 非法指针(使用空指针或随意转使用指针转换)

3) 堆栈溢出

4) 多线程使用了线程不安全的函数

5) 多线程读写的数据没有加锁保护

定位 coredump 文件：

1) 先编译生成含有调试信息的可执行文件 gcc -g test.cpp -o test

2) 在执行 ./test,从而产生 coredump 文件

3) 在执行 gdb ./test ./core

4) 在 gdb 下输入 where

1.19 gdb 调试

```
g++ -g test.cpp -o test
./test
```

l (list)	罗列出当前代码信息
b (break point)	设置断点
r (run)	运行直到断点或结束
n (next)	单步调试，不会进入函数 类似 vs 的 F10
s (step)	单步调试 会进入函数体 类似 vs 的 F11
f (frame)	切换函数的栈帧
p (print)	打印
t (thread)	切换线程

c (continue)	程序继续执行直到下一个断点或程序结束
bt (backtrace)	显示堆栈信息
i (info)	查看信息
d (delete)	删除断点
fin (finish)	退出函数

1.20 缓存淘汰算法

LRU（least recently used，最近最少使用），根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

1） 新数据插入到链表头部；

2） 每当缓存命中（即缓存数据被访问），则将数据移到链表头部；

3） 当链表满的时候，将链表尾部的数据丢弃。

1.21 Valgrind

a) 内存分析工具，由 **Memcheck**，Cachegrind，Helgrind 等组成。

b) **Memcheck** 是一款 内存泄漏 检测工具

c) **Memcheck** 能够检测出内存问题的原理：关键在于建立两个全局表：

1) **Valid-Value 表：**对于进程的整个地址空间的每一个字节（Byte），都与之对应的 8 bit，这些 bit 负责记录该字节或寄存器的值是否有效，是否被初始化等。（CPU 的每个继承寄存器，也有与之对的 bit 向量）

2) **Valid-Address 表：** 对于进程整个地址空间中每一个字节（Byte），还有与之对应的 1bit，负责记录该地是否能够被读写。

d) MemCheck 原理：

1) 当要读写某个字节时，首先检测该字节的 A bit，若 A bit 显示无效位置，memcheck 则报告错误。

2) 当字节加载到 cpu 时，它对应的 v bit 也被加载到 cpu

环境中，首先检测对应的 v bit，若为初始化，则 memcheck 报错。

1.22 进程通讯的几种方法

1） 管道。管道的机制类似于缓存，是单向传输的，效率较为低下，比如：a 进程给 b 进程传输数据，只能等待 b 进程取了数据之后 a 进程才能返回。

2） 消息队列。优点：把进程的数据放在某内存以后，无需等待其他进程来取就返回。缺点：

如果 a 进程发送的数据占内存比较大，并且两个进程之间通信较为频繁，消息队列模型就不合适，因为 a 发送的数据很大的话，发送消息（拷贝）这个过程需要花很多时间来读内存。

3） 共享内存。系统加载一个进程的时候，分配给进程的内存并不是实际物理内存，二是虚拟内存空间。将两个进程的虚拟内存空间映射到同一个实际物理内存中，就实现了内存共享的机制。

4） 信号量。为了解决共享内存的线程安全问题，引入信号量的通信方式。

5） Socket。

1.23 linux 进程间共享内存如何同步

1） **利用 POSIX 有名信号灯实现共享内存的同步。**两个进程，对同一个共享内容读写。利用两个有名信号量 semr，smew。Semr 信号量控制能否读，初始化为 0。Smew 信号量控制能否写，初始为 1。

- 读共享内存的程序示例代码：

```
Semr = sem_open(“mysem_r”,O_CREAT | O_RDWR, 0666,0);
```

2） **利用 POSIX 无名信号灯实现。**POSIX 无名信号量是基于内存的信号量，可以用于线程间同步也可以用于进程间同步，需要在共享内存中创建无名信号量。

3） **利用信号实现共享内存的同步。**信号是软件层次上对中断机制的一种模拟，是一种异步通信方式。利用信号也可以实现共享内存的同步。

1.24 文件读写基本流程

读文件

1） 进程调用库函数向内核发起读文件请求；

2） 内核通过检查进程的文件描述符定位到虚拟文件系统的已打开文件列表表项；

3） 调用该文件可用的系统调用函数 read()；

4） Read()函数通过文件表链接到目录项模块，根据传入的文件路径，在目录项模块中检索，找到该文件的 inode；

5） 在 inode 中，通过文件内容偏移量计算出要读取的页；

6） 通过 inode 找到文件对应的 address_space

7） 在 address_space 中访问该文件的页缓存树，查找对应的页缓存节点：

（1） 如果页缓存命中，那么直接返回文件内容；

（2） 如果页缓存缺失，那么产生一个页缺失异常，创建一个页缓存页，同时通过 inode 找到文件该页的磁盘地址，读取相应的页填充该缓存页；重新进行第 7 步查找页缓存；

8） 文件内容读取成功。

写文件

前 6 步和读文件一致，在 address_space 中查询对应页的页缓存是否存在：

7）如果页缓存命中，直接将文件内容修改更新在此页中。写文件就结束了。这时候文件修改位于页缓存，并没有写回到磁盘文件中去。

8）如果页缓存缺失，那么产生一个缺页异常，创建一个页缓存页，同时通过 inode 找到文件该页的磁盘地址，读取相应的页填充该缓存页。此时缓存页命中，进行第 7 步。

9）一个页缓存中的页如果被修改，那么会被标记为脏页。脏页需要写回到磁盘中的文件块。有两种方式：

（1）手动调用 sync()或者 fsync()系统调用能够把脏页写回。

（2）pdflush 进程会定时把脏页写回到磁盘

同时注意，脏页不能被置换出内存，如果脏页正在被写回，那么会被设置写

回标记，这时候该页就被上锁，其他写请求被阻塞直到锁释放。

1.25 编译的过程

编译的分别四步，也即是：预编译处理(.c) —> 编译、优化程序（.s）—> 汇编程序(.obj、.o、.a、.ko) —> 链接程序（.exe、.elf、.axf 等），具体如下：

如下：

1) 预处理 ：预处理器指令处理 # 号开头的，比如#include, #define; 一般来讲因为引入了头文件，预处理编译的文件比源文件大，将 .cpp 文件转为 .i 文件。

a) 命令 g++ -E test.cpp -o test.i

b) 主要处理以 # 开始的预编译指令， 如 #include、 #define 和注释

c) #pragma 则保留，编译器需要它们

2) 编译 ： 通过预处理后的 .i 文件中，只有常量，数字，字符串及关键字等，编译主要是通过语法分析和词法分析，来确定所有指令是否符号规则，并将其转换成汇编代码， 将 .i 文件转为 .s 文件。

a) 命令 ： g++ -S test.i -o test.s

b) 主要进行词法分析，语法分析，语义分析等

3) 汇编：将汇编语言翻译成目标机器指令的过程，将 .s 文件转为 .o 文件。

a) 命令 ：g++ -c test.s -o test.o

4) 链接：汇编生成的目标并不能立即执行，还需要通过链接，因为 某个文件可能调用另外一个源文件中的函数或常量，或是函数中调用了库函数; 链接的主要工作就是将有关目标文件连接起来， 将 .o 文件转为可执行文件。

a) 静态链接 ： 在编译期间就确定了

i. 对库函数的链接是在编译期间完成的。 Libxxx.a

b) 动态链接 ： 在运行时再载入

i. 在程序运行时期才载入

1.26 g++与 gcc 的区别:

1) 它们都可以编译.c 和.cpp 文件；后缀为 .c 的文件， gcc 把它当作 C 程序， g++ 则将它当作 C++ 程序，对于 .cpp 文件，gcc 和 g++ 都将其视为 c++程序。

2) 编译阶段，g++会调用 gcc，对于 cpp 文件，两者等价，但 gcc 命令

不能自动和C++程序使用的库链接。因此，对于cpp文件，一般使用g++。

3） 对于 extern ”C” ，gcc/g++都将以C的方式进行编译。

1.27 epoll 怎么实现的？

1. linux epoll机制是通过红黑树和双向链表实现的。
   a) 首先通过 epoll_create()系统调用在内核中创建一个eventpoll类型的句柄，其中包括红黑树的根节点和双向链表的头结点。
   b) 然后通过epoll_ctl()系统调用，向epoll对象的红黑树结构中添加、删除和修改感兴趣的事件，返回0表示成功，返回-1表示失败。
   c) 最后通过 epoll_wait()系统调用判断双向链表是否为空，如果为空则阻塞。当文件描述符状态改变，fd 上的回调函数被调用，该函数将fd加入到双向链表中，此时epoll_wait()函数被唤醒，返回就绪好的事件。
2. 什么是epoll
   d) epoll 是为处理大批量句柄而做了改进的 poll，是性能最好的多路I/O就绪通知方法；
   e) 只有三个系统调用：epoll_create, epoll_ctl, epoll_wait;
   f) epoll_ctl ，epoll的事件注册函数，它不同于select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里注册要监听的事件类型；
3. epoll工作原理

1） epoll 只告知那些就绪的文件描述符，而且当我们调用epoll_wait()获得就绪文件描述符时，返回的不是实际的描述符，而是一个代表就绪描述符数量的值。

2） 只需要去 epoll 指定的一个数组中依次取得相应数量的文件描述符即可，这里也使用了内存映射技术，这样便彻底省掉了这些文件描述符在系统调用时复制的开销。

3） 在select/poll中，进程只有在调用一定的方法后，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，一旦基于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；

4. .epoll 总结：

一颗红黑树，一张准备就绪句柄链表，少量的内核 cache，就帮我们解决了大并发下的 socket 处理问题。

1） 执行 epoll_create()时，创建了红黑树和就绪链表；

2） 执行 epoll_ctl()时，如果增加 socket 句柄，则检查在红黑树中是否存在，存在立即返回，不存在则添加到树干上，然后向**内核注册回调函数**，用于事件发生时向准备就绪链表中插入数据；

3） 执行 epoll_wait()时立刻返回准备就绪链表里的数据即可

表 9-2 select、poll 和 epoll 的区别

系统调用	select	poll	epoll
事件集合	用户通过 3 个参数分别传入感兴趣的可读、可写及异常等事件，内核通过对这些参数的在线修改来反馈其中的就绪事件。这使得用户每次调用 select 都要重置这 3 个参数	统一处理所有事件类型，因此只需一个事件集参数。用户通过 pollfd.events 传入感兴趣的事件，内核通过修改 pollfd.revents 反馈其中就绪的事件	内核通过一个事件表直接管理用户感兴趣的所有事件。因此每次调用 epoll_wait 时，无须反复传入用户感兴趣的事件。epoll_wait 系统调用的参数 events 仅用来反馈就绪的事件
应用程序索引就绪文件描述符的时间复杂度	O(n)	O(n)	O(1)
最大支持文件描述符数	一般有最大值限制	65 535	65 535
工作模式	LT	LT	支持 ET 高效模式
内核实现和工作效率	采用轮询方式来检测就绪事件，算法时间复杂度为 O(n)	采用轮询方式来检测就绪事件，算法时间复杂度为 O(n)	采用回调方式来检测就绪事件，算法时间复杂度为 O(1)

1.25 为什么析构函数必须是虚函数？为什么 C++默认的析构函数不是虚函数？

答：

1） 将可能被继承的父类的析构函数设置为虚函数，可以保证当我们 new 一个子类，然后使用基类指针指向该子类对象时，释放基类指针时可以释放掉子类的空间。

2） 默认的析构函数不是虚函数是因为虚函数需要额外的虚函数表和虚函数指针，占用额外的内存。而对于不会被继承的类来说，其析构函数如果是虚函数，就会浪费内存。因此 C++默认的析构函数不是虚函数，而是只有当需要当做父类时，才设置为虚函数。

3） 构造函数为什么不能为虚函数？

a）在构造对象时必须知道确切的类型

b）在构造函数之前，对象时不存在的，从而就无法使用指向此对象的指针来调用构造函数。

1.26 容器相关

容器类型	容器特性
vector	1.序列容器，内存是连续，可以随机访问元素，任意元素的读取、修改具有常数时间复杂度，在序列尾部进行插入、删除是常数时间复杂度，但在序列的头部插入、删除的时间复杂度是O(n)，插入和删除操作会使得其之后的迭代器失效，因为vector是连续的地址，插入和删除时，需要移动元素。 2. 在尾部添加元素的push_back()平均复杂度为O(1) ,推导如下： 若插入N个元素，采用2那里增长的方式，则会引发 lgN次的内存扩充，每次内存扩充，都会有一个过程：申请内存 -> 拷贝元素 -> 释放原来的内存，则拷贝的次数为 2^0 ,2^1, 2^2,…, 2^lgN;则所有拷贝次数相加可以得到的总拷贝次数：2^0 + 2^1 + 2^2 + … + 2^lgN = 2*2^lgN - 1 ≈ 2N ,若在push_back()，则又会扩容，每个元素拷贝一次，共拷贝了N次，所有总的操作次数为3N,平均下来就是3次 因此每个push_back()平均复杂度为常数事件复杂度O(1). 3是动态可增长数组。增长时，一般为现有capacity的1.5或2倍方式，1.5倍的方式可以使用前面重复的内存,2倍增长则不可以。 4.适合于大量的随机访问，若在任意位置插入删除较多，则不适应。
deque	1. deque是双端队列，是一种序列容器，和vector类似，但又有不同。 2. deque是由一段一段的定量连续的空间构成， deque采用map作为主控，map是一小块连续空间，每个元素都是指针，指向另外一段线性连续空间；deque提供了两级数组结构，第一季完全类似vector，代表实际容器，另外一级维护容器的首地址。此外，deque还支持高效首/尾插入和删除操作。push_front()和push_back()。deque在元素添加时扩充空间，因此没有capacity。 3.缺点就是占用内存过多。
list	1.list是双向链表，非连续的地址空间。每个元素都维护一个前向和后续指针，因而支持前向和后向遍历。 2.支持高效的随机插入和删除操作，但不能随机访问。 3.C++11新增了forward_list是单项链表。
set	1.set是关联容器，元素不允许有重复，底层实现是采用的是红黑树，查找的速度非常快，时间复杂度是O(logN)
multiset	1. 关联容器，和set一样，允许有重复的元素，具备时间复杂度O(logN)查找功能。

容器类型	容器特性
map	1.关联容器，按照{键，值}方式组成集合，底层实现是采用的是红黑树，查找的时间复杂度 O(logN)，其中键不允许重复
multimap	和 map 一样，区别是键可以重复

2. 网络相关

2.1 网路模型：

七层	五层	四层
应用层	应用层	应用层
表示层		
会话层		
传输层	传输层	传输层
网络层	网路层	网络层
数据链路层	数据链路层	
物理层	物理层	网际接口层

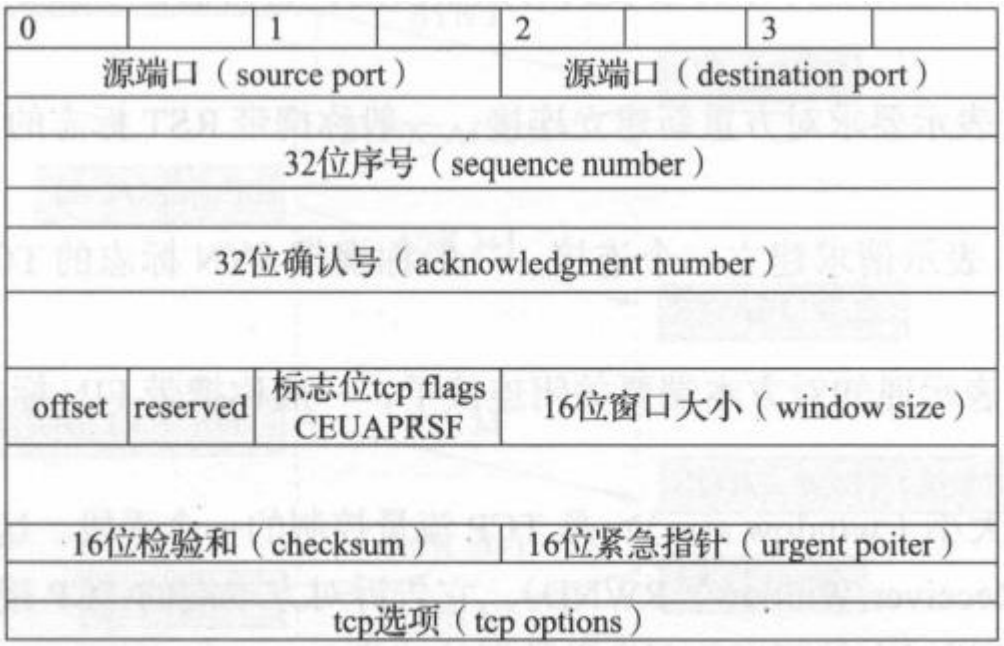


图 6-2 TCP 头部

2.2 TCP 如何保证传输可靠性

1） 校验和。发送的数据包的二进制相加然后取反，目的是检测数据在传输过程中是否变化。如果检验和有差错，TCP 将丢弃这个报文段和不确认收

到此报文段。

2） 确认应答和序列号。TCP 给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。

3） 超时重传。当 TCP 发出一个段后，启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

4） 流量控制。TCP 连接的每一方都有固定大小的缓冲空间，TCP 的接受端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送速率，防止包丢失。TCP 使用的流量控制协议是可变大小的滑动窗口协议。接受方有即时窗口（滑动窗口），随 ACK 报文发送。

5） 拥塞控制。当网络拥塞时，减少数据的发送。（慢启动、拥塞避免、拥塞发送、快速恢复）

2.3 http1.1、http1.0 和 http2.0 区别

http1.1 和 http1.0 区别：

a) 1.1 默认支持长连接。

b) 1.1 带宽优化，并支持断点续传。这是因为 1.1 在请求消息里加入 range 头域，它支持只请求资源的某个部分。在响应消息中 Content-Range 头域声明了返回的这部分对象的偏移值和长度。

c) 新增例如 ETag， If-None-Match 等更多的缓存控制策略

d) Host 头域

e) 新增了 24 个错误状态响应码。

http2.0 与 http1.1 相比有以下不同：

a) 多路复用，可以做到在一个连接并行的处理多个请求
b) Header 压缩
c) 服务端推送；
d) 解析格式不同，1.0 和 1.1 的解析是基于文本，2.0 协议解析采用二进制，实现方便且健壮。

2.4 https 和 http

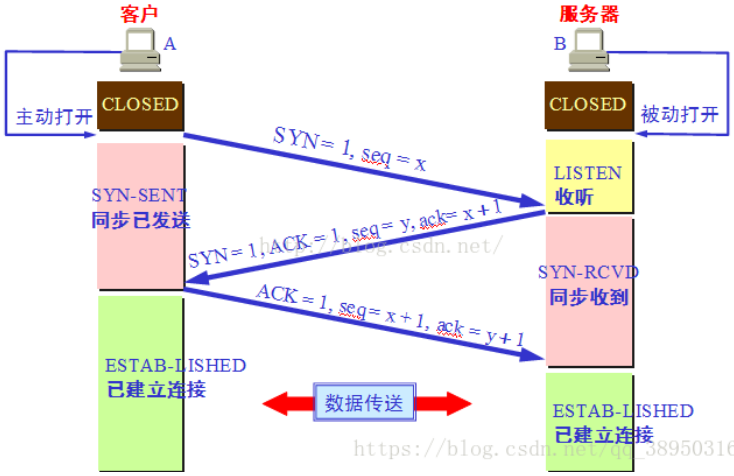
https 特点：

1） 内容加密：采用混合加密技术，中间者无法直接查看明文内容

2） 验证身份：通过证书认证客户端访问的是自己的服务器

3） 保护数据完整性：防止传输的内容被中间人冒充或者篡改

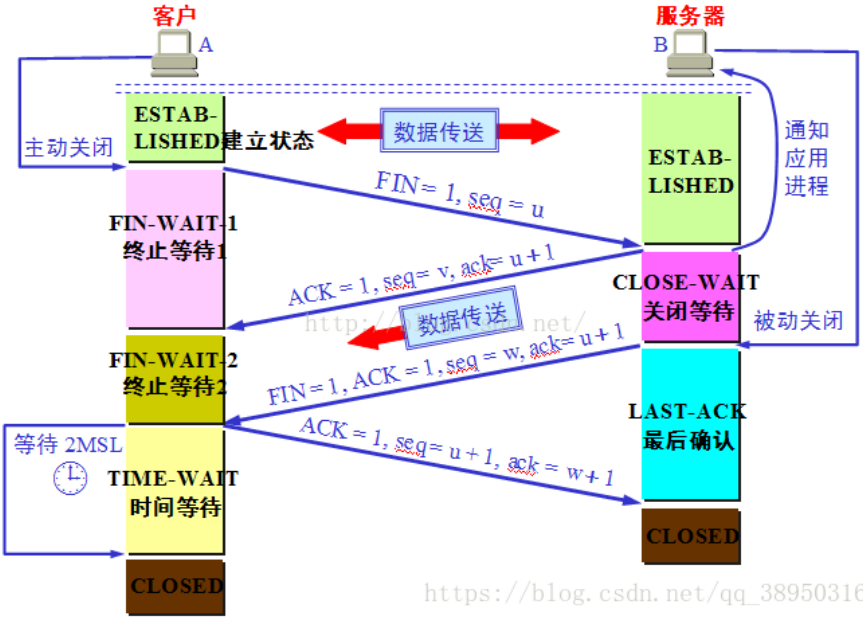
2.5 TCP 三次握手和四次挥手过程理解



第一次握手：建立连接时，客户端发送 syn 包（syn=x）到服务器，并进入 SYN_SENT 状态，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）。

第二次握手：服务器收到 syn 包，必须确认客户的 SYN（ack=x+1），同时自己也发送一个 SYN 包（syn=y），即 SYN+ACK 包，此时服务器进入 SYN_RECV 状态；

第三次握手：客户端收到服务器的 SYN+ACK 包，向服务器发送确认包 ACK(ack=y+1)，此包发送完毕，客户端和服务器进入 ESTABLISHED（TCP 连接成功）状态，完成三次握手。



1）客户端进程发出连接释放报文，并且停止发送数据。释放数据报文首部，FIN=1，其序列号为 seq=u（等于前面已经传送过来的数据的最后一个字节的序号加 1），此时，客户端进入 FIN-WAIT-1（终止等待 1）状态。 TCP 规定，FIN 报文段即使不携带数据，也要消耗一个序号。

2）服务器收到连接释放报文，发出确认报文，ACK=1，ack=u+1，并且带上自己的序列号 seq=v，此时，服务端就进入了 CLOSE-WAIT（关闭等待）状态。TCP 服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个 CLOSE-WAIT 状态持续的时间。

3）客户端收到服务器的确认请求后，此时，客户端就进入 FIN-WAIT-2（终止等待 2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。

4)服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，FIN=1，ack=u+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为 seq=w，此时，服务器就进入了 LAST-ACK（最后确认）状态，等待客户端的确认。

5）客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，ack=w+1，而自己的序列号是 seq=u+1，此时，客户端就进入了 TIME-WAIT（时间等待）状态。注意此时 TCP 连接还没有释放，必须经过 2**MSL（最长报文段寿命）的时间

后，当客户端撤销相应的 TCB 后，才进入 CLOSED 状态。

6）服务器只要收到了客户端发出的确认，立即进入 CLOSED 状态。同样，撤销 TCB 后，就结束了这次的 TCP 连接。可以看到，服务器结束 TCP 连接的时间要比客户端早一些。

2.5 从输入网址到获得页面的过程

1）浏览器查询 DNS，获取域名对应的 IP 地址：具体过程包括浏览器搜索自身的 DNS 缓存、搜索操作系统的 DNS 缓存、读取本地的 Host 文件和向本地 DNS 服务器进行查询等。对于向本地 DNS 服务器进行查询，如果要查询的域名包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析；如果要查询的域名不由本地 DNS 服务器区域解析，但该服务器已经缓存了此网址映射关系，则调用这个 IP 地址映射，完成域名解析。如果本地域名服务器并未缓存该网址映射关系，那么将根据其设置发起递归查询或迭代查询。

2）浏览器获得域名对应的 IP 地址后，浏览器向服务器请求建立连接，发起三次握手

3）TCP/IP 连接建立起来后，浏览器向服务器发送 HTTP 请求

4）服务器接收到这个请求，根据路径参数映射到特定的请求处理器进行处理，并将处理结果返回给浏览器

5）浏览器根据请求到的资源，最终向用户呈现一个完整的页面

3．算法

3.1 堆排序

a. 将无序序列构建成一个堆，根据升序降序需求选择大顶堆或小顶堆；

b. 将堆顶元素与末尾元素交换，将最大元素“沉”到数组末端；

c. 重新调整结构，使其满足堆定义，然后继续交换堆顶元素与当前末尾元素，反复执行调整和交换步骤，直到整个序列有序。

4．设计模式

4.1 单例模式

```
class Singleton
{
```

```
public:
    ~Singleton() {
    }
    Singleton(const Singleton& other) = delete;
    static Singleton& getInstance()
    {
        static Singleton instance;
        return instance;
    }
private:
    Singleton() {
    }
};
```

新建一个实例：

```
Singleton &a = Singleton::getInstance();
```

4.2 工厂模式

工厂模式主要解决接口选择的问题。该模式下定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，使其创建过程延迟到子类进行。

5. 操作系统

5.1 进程与线程的区别

1）一个进程可以有多个线程，而一个线程只能属于一个进程程。

2）进程是资源分配的最小单位，线程是系统调度的最小单位。

3）进程用于自己的独立地址空间。系统创建一个进行时，会为其分配地址空间，建立数据段，代码段和堆栈段；线程没有自己的地址空间，只用于一些必不可少的资源，如线程栈，程序状态寄存器 wpc， 程序计数器 pc 等。同一个进程的线程共享进行的地址空间。

4）进程的切换开销比线程的切换开销大。进程创建和销毁时，系统都要为

之分配或回收资源，而线程所拥有的资源很少，因此，操作系统所付出的开销显著大于创建或撤销线程时的开销。在进行进程切换时，涉及到整个当前进程 CPU 环境的保存以及新被调度运行的进程的 CPU 环境的设置。而线程切换只须保存和设置少量寄存器内容。

5）多进程程序更安全。一个进程崩溃不会对另外一个进程造成影响，而同一进程的多个线程，若某线程崩溃，可能导致整个进程崩溃。

6）线程之间通信更加方便，同一个进程间的多个线程，可以通过全局变量等方式实现通信，而进程之间需要通过 IPC 方式进行。

5.2 有了进程为什么还需要线程？

线程产生的原因：　进程可以使得多个程序能并发的执行以提高资源的利用率和系统吞吐量，但是同一进程在同一时刻只能干一件事。

若进程阻塞，整个进程都会挂起，即使进程中某些认为不需要等待的资源，因此，操作系统引入了比进程更小的线程作为并发执行的基本单位，从而减少程序在并发执行时所需要付出的时空开销，进一步提高并发性。

5.3 进程间通信 IPC

1）管道：

a) 包括无名管道和命名管道。无名管道用于具有亲缘关系的父子进程间通信。命名管道则没限制。

b) 管道是一种两个进程间单向通信的机制，是半双工通信。

c) 管道的本质是固定大小的内核缓冲区，因而大小是由限制的。可以通过 unlimited －a 查看，大小一般为 4k Byte。

2）消息队列：

a) 用于同一台机器上的进程间通信，在内核中用来保存消息的队列。

b) 在内核中以消息链表的形式出现。

c) 相比于管道，消息队列可以独立发送和结束，可以通过消息类型来选择性的结束数据。

3）信号量(system V)：

a) 信号量用于实现进程间互斥和同步。

4）共享内存：

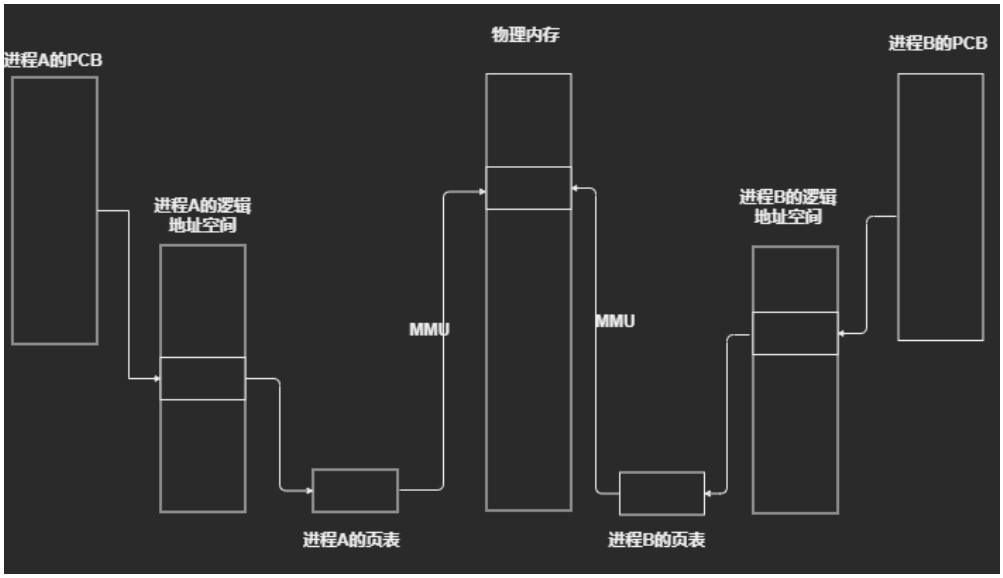
a) 多个进程将通过 MMU 将逻辑地址映射到同一物理地址上，从而进程间共享内存。允许两个或多个进程访问同一内存，当某一个进程改变了内存中的内容时，其他进程都会察觉到这个变化。

b) 共享内存没有提供同步机制，需要使用同步机制来保证对内存访问，避免产生竞态条件，通常使用信号量机制来实现进程间同步与互斥。

c) 相关 API

1. shmget():创建或获取一段共享内存
2. shmat()：关联到进程的地址空间
3. shmdt()：从进程地址空间分离
4. shmctl()：更改共享内存属性

d) 原理： 每个进程有自己的 PCB 和地址空间及页表。页表通过 MMU 管理，将逻辑地址地址和物理地址进程映射。可以将两个进程映射都同一块物理内存，实现共享内存。



AB 两个进程通过共享内存来通信示意

e) 共享内存优点：

进程间通信效率高，通过直接访问内存实现通信，避免数据拷贝和系统调用。

但是需要解决进程间同步比如信号量等方式，来避免竞态。

5）socket：注意用于网络间的通信

a) socket(), bind(), listen, accept()

5.4 进程间通信相关命令(ipcs)

ipcs 是一个 Linux 命令,用于报告系统的消息队列,信号量共享内存等。
ipcs -a 列出本用户所有相关 ipcs 参数。
ipcs -q 列出进程中的消息队列
ipcs -s 列出所有信号量
ipcs -m 列出所有的共享内存信息
ipcs -l 列出系统的限额
ipcs -t 列出最后的访问时间
ipcs -u 列出当前的使用情况

5.5 线程间同步通信

1）临界区

2）锁机制

某些资源的访问需要互斥访问，怎么保证互斥访问呢？就引入了锁的概念，只有获取锁的线程才能对资源进程访问，再同一时刻只能有一个线程获获取到锁，那么没有获取到锁的线程怎么办？

一般有两种处理方式：

1. 没有获取到锁的线程一直循环等待并判断该资源释放已经被释放，这就是自旋锁。自旋锁不会阻塞，而是忙等
2. 另外一种就是把自己阻塞起来，等待重新调度，这就是互斥锁。

a) 互斥锁(mutex)：

为了确保同一时间只有一个线程访问数据，在访问共享资源前需要对互斥量上锁。一旦对互斥 量上锁后，任何其他试图再次对互斥量上锁的线程都会被阻塞，即进入等待队列，当其他线程释放互斥 量后，操作系统会激活那个被挂起的线程，让其投入运行。[加锁失败，被阻塞进入等待队列]

b) 自旋锁(reader-writer lock)：

自旋锁与互斥量最不同的是：非阻塞锁，它不会被挂起，而是在获取锁之前一直处于忙等，即 不停在消耗 cpu，执行循环。适用于多核处理器、临界区无阻塞情况，其中一个 CPU 上的线程进入临界区，另一个 CPU 上的线程尝试获取锁会自旋，因为它不会阻塞，所以只要稍稍等一下下就能进入临界区（预计线程等待锁的时间很短，短到比线程两次上下文切换时间要少的情况下），对于多核 CPU 来说，会提高并发率。[加锁失败，不被阻塞，而

是忙等]

如果持有锁的线程能在短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞状态，它们只需要等一等(自旋)，等到持有锁的线程释放锁之后即可获取，这样就避免了用户进程和内核切换的消耗。

因为自旋锁避免了操作系统进程调度和线程切换，所以自旋锁通常适用在时间比较短的情况下。由于这个原因，**操作系统的内核经常使用自旋锁(若自旋的消耗会小于线程阻塞挂起再唤醒的操作的消耗)**

c) 读写锁(spin lock)：

可以多个线程同时读，但是不能多个线程同时写，锁处于读模式时可以线程共享，而锁处于写 模式时只能独占，所以读写锁又叫做共享-独占锁，读写锁比互斥锁更加具有适用性和并行性，最适用于对数据结构的读操作读操作次数多余写操作次数的场合。(适合读多写少的场景)

3）信号量(sem)：

信号量的实现中一般具有两种操作，分别是 P 操作和 V 操作。对一个信号量进行 P 操作时，首先检查其值是否大于 0，如果大于 0，则将其值减一之后返回进行后续操作，如果值小于等于 0，则该进程将进行阻塞。(P，V 都是原子操作)

信号量（sem）和互斥锁的区别：互斥锁只允许一个线程进入临界区，而信号量允许多个线程 进入临界区。

4）条件变量(cond)

当线程在等待满足某些条件时使线程进入睡眠状态，一旦条件满足，就唤醒线程

在条件变量的内部，有一次解锁加锁过程，条件不满足，将本线程加入等待队列，同时将传入的 mutex变量解锁，一旦等待队列中的线程被唤醒，会再次对传入的 mutex变量加锁！[条件不满足时，将线程放入队列中，对mutex变量进行解锁（这两步都是原子操作）一旦线程被唤醒，就对mutex变量进行加

锁]

条件变量产生的原因：

互斥量不是万能的，比如某个线程正在等待共享数据内某个条件出现，可能需要重复对数 据对象加锁和解锁（轮询），但是这样轮询非常耗费时间和资源，而且效率非常低，所以互斥锁不太适

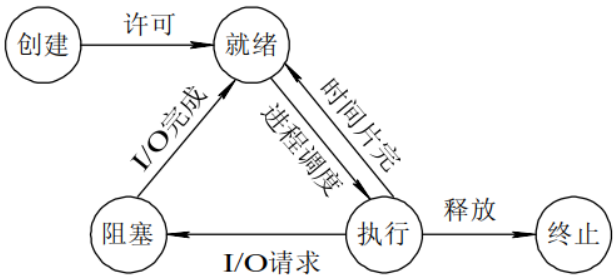


图 2-7　进程的五种基本状态及转换

5.6 进程相关概念

1) 进程一般由 3 部分组成：数据段，堆栈段和堆栈段，其中进程控制块 PCB 存放在堆栈段。
2) PCB 进程控制块，PCB 是进程存在的唯一标识，系统通过感知 PCB 的存在而感知进程的存在。
3) 系统通过对 PCB 来对进程管理和调度，比如进程的创建，调度，优先级等都是通过 PCB 来实现处理的。
4) 由两种创建进程的方式:
    a) 系统创建:比如 Linux,开机时内核执行 0 号进程，它是所有进程的主线，0 号进程创建 1 号进程(inti 1 内核进程)再继续执行下去。大致过程为：
    0 号进程->1 号内核进程->1 号用户进程。[0 号进程通过系统自举时由系统创建]
    b) 由父进程创建(fork,vfork,clone)
        a) fork 返回两次的原因：
            由于复制时复制了父进程的堆栈段，因此两个进程都停留在 fork 函数中，fork 返回两次，一次是在父进程中返回(返回的子进程的 id)，另一次是在子进程中返回(0)，返回值不同;出错返回-1。

b) fork 与 vfork 及 clone

fork

创建一个和父进程一样的进程，完全复制父进程的资源，现在的 Linux 系统一般都采用 copy-on-write(COW)技术，降低开销，也即是在一开始并不会复制父进程的资源，当要对数据写入时，才进行拷贝，从而降低开销。

fork()会返回两次，父子进程执行顺序不定。

vfork ：

vfork 也是创建子进程，但是 vfork 创建的子进程与父进程共享地址空间，也即是子进程完全允许在父进程地址空间上。

vfork()创建的子进程要显示调用 exit()来结束。

vfork 创建的子进程的执行顺序是固定的，先执行子进程，然后执行父进程。

vfork 返回值和 fork 一样。

clone:

fork 和 vfork 都是无参的，clone 带有参数，fork 是全部复制，vfork 是共享，clone 则是将父进程的资源有选择性的复制给子进程，而为复制的数据结构则通过复制指针的方式让子进程共享，具体复制哪些资源有参数列表指定。

clone()返回的是子进程的 pid。

5.7 返回值

正常退出返回的方式：

1) return ： 是函数执行完后返回，return 把控制权交给调用函数
2) exit() ： 是一个函数，带有参数，exit 执行后把控制权交给系统，0 时，代表进程正常终止，若为其他值，表示程序执行过程中有错误发生。
3) _exit()
4) exit 与 _exit()的区别：
   a) 两个都是函数，用来终止进程的。
   b) 头文件不同，exit()在 stdlib.h, _exit()在 unistd.h
   c) _exit()不会刷新流(stdin,stdout,stderr)， exit()会刷新数据流。
   d) exit()是在_exit()基础之上一个封装。
   e) 两个最大的区别：exit()会把文件缓冲区的内容写回文件，而_exit()不会写回，导致缓冲区的数据丢失。

异常退出的返回方式：

1) 调用 abort(),
2) 进程收到信号，使得程序终止

5.8 孤儿进程与僵尸进程

子进程是通过父进程创建的。子进程的结束和父进程的运行是一个异步过程，即父进程永远无法预测子进程到底什么时候结束。于是就产生了孤儿进程和僵尸进程

孤儿进程：父进程退出，而其子进程还在运行，则这些子进程将成为孤儿进程，这些孤儿进程将被init进程(进程id为 1)接收，并由init进程处理这些子进程的善后工作。

僵尸进程：子进程退出，而父进程并没有调用wait() 或waitpid()获取子进程的状态信息，对子进程进行善后工作，导致资源浪费，如子进程的进程描述符未回收。

孤儿进程和僵尸进程的区别：孤儿进程是父进程已退出，而子进程未退出；僵尸进程是父进程未退出，而子进程已退出。

守护进程是脱离于终端并且在后台运行的进程

5.9 虚拟内存 与 内存管理机制：

虚拟内存

5.10 局部性原理

1) 时间局部性：如果程序中的某条指令一旦执行，则不久以后该指令可能再次执行；如果某数据被访问过，则不久以后该数据可能再次被访问。产生时间局部性的典型原因，是由于在程序中存在着大量的循环操作。
2) 空间局部性：一旦程序访问量某个存储单元，在不久之后，其附近的存储单元也将被访问，即程序在一段时间内所访问的地址，可能集中在一定的范围之内，其典型情况便是程序的顺序执行。
3) 时间局部性是通过将进来使用的指令和数据保存到高速缓存存储器中，并使用高速缓存的层次结构实现。空间局部性通常是使用较大的高速缓存，并将预取机制集成到高速缓存控制逻辑中实现。虚拟内存技术实际上就是建立了“内存-外存”的两级存储器的结构，利用局部性原理实现高速缓存。

5.11 缺页中断 ：

malloc 及 mmap 等内存分配函数，在申请内存时，分配的只是逻辑地址空间，并没有分配实际物理地址，当访问这些地址时，会触发 缺页异常

缺页中断： 在 请求分页系统中， 可以通过查询 页表 中的状态来确定所访问的页面是否存在于内存中；当要访问的页面不在内存中时，会产生 缺页中断， 此时会按照一定的调度算法 从外存中将所却的页面调入内存。

缺页中断也是一种中断，过程如下：

1. 保护 CPU 现场(保存程序计数器，程序状态寄存器和栈数据等信息)
2. 分析中断原因
3. 转入 缺页中断处理程序进程处理
4. 恢复 CPU 现场，继续执行

缺页异常因为要访问的页面不在内存中，是硬件所产生的一种特殊中断，与一般中断有如下区别：

1. 是在指令执行期间产生和处理缺页中断
2. 一条指令期间，可能产生多次缺页中断

5.11 操作系统内存管理

操作系统对内存的划分和动态分配即为内存管路，总的来说，操作系统内存管理分为物理内存管理和虚拟内存管理。

内存管理的功能：

a) 内存空间分配和回收，包括内存的分配和共享
b) 地址转换，把逻辑地址转为相应的物理地址
c) 内存空间扩充，利用虚拟技术，从逻辑上扩充内存
d) 存储保护，保证各道作业在各自存储空间运行，互不干扰。

5.11.1 物理内存管理包括程序装入等概念，交换技术，连续分配管理方式和非连续分配管理方式(分页，分段，段页式)。

1） 连续分配管理方式:连续分配内存是指为一个用户程序分配一个连续的内存空间，简单，没有外部碎片，但有内部碎片且只能用于单用户，单任务的操作系统。主要方法有：
   a) 单一连续分配
   b) 固定分区分配
   c) 动态分区分配

2） 非连续分配管理方式： 运行一个程序分散的装入不相邻的内存分区中，根据根据分区大小是否固定可分为分页存储管理和分段存储管理以及段页式存储。
   a) 分页存储管理：根据运行作业时是否把所有页面都装入内存分为基本分页存储管理和请求分页存储管理。
      A. 分页的思想：把内存空间划分为大小相等且固定的块，块的大小不宜过多或过小，一般为 4KB,作为主存的基本单位。每个进程以块为单位进行划分，运行时，以块为单位逐个申请主从中的块空间。
      B. 进程中块称为页(虚拟)，内存中块称为页框(物理页)。
      C. 页表是记录逻辑页在内存中对应的物理页号的地址映射，页表一般常驻内存，页表的作用是实现页号到物理块号

的地址映射。

D. 快表(TLB):若利用页表，存取一个数据至少需要两次访问内存，一次是访问页表获取物理地址，第二次是根据物理地址获取数据。为了加快速度，在地址变换时设置了高速缓冲存储器也即是快表(联想寄存器 TLB)。TLB 在寄存器中，首先先去 TLB 查找，若没有，再去快表，若在没有，则缺页中断等操作。快表利用的局部性原理。

b) 分段存储管理：按照进程的自然段进行逻辑空间，段内地址空间要求连续，而段间可以不连续。它的逻辑地址由两部分组成：段号和段内偏移量。

1. 在分页管理中，逻辑页号和页内偏移量对用户透明，但在段式管理中，段号和段内偏移量由用户显示提供。
2. 段表：每个进程都有一张逻辑空间与主存空间映射的段表，每一段表项对应进程的一个段，段表现记录内存中的起始地址和段的长度。段表和页表作用类似，都是用于实现从逻辑段到物理内存区的映射。
3. 地址变换机构：为了实现进程从逻辑地址到物理地址的变换功能，在系统中设置了段表寄存器

c) 段页式存储管理：

1. 页式存储管理能有效地提高内存利用率，而分段存储管理能反应程序的逻辑结构并利用段的共享。因此，段页式存储管理将两则结合起来。
2. 将作业的地址空间分为若干逻辑段，每个段都有自己的段号，然后将每一个段分成若干页。对内存空间的管理和分页存储管理一样，
3. 段页式存储系统中，作业的逻辑地址分为三部分：段号，页号和页面偏移量，每个进程有一张段表，每个段有一张页表。系统中还有段表寄存器，指出段表的起始地址和段表长度。

5.11.2 虚拟内存管理包括虚拟内存概念，请求分页管理，页面置换算法，页面分配策略，工作集和抖动等。

1) 虚拟内存(virtual memory):

2) 请求分页管理

a) 请求分页系统中，将当前需要的一部分页装入内存，便于进程运行，在进程运行期间，去查找相应的页，当访问的页面不在内存时，将页表调入内存，若分配空间已满，则通过相应的页面置换算法将换出暂不用的页面到外存，腾出内存空

间，便于将所需要的页面调入。

3） 页面置换算法

a） 最佳页面置换算法(OPT 理想置换算法)：从主存中移出永远不再需要的页面；如无这样的页面存在，则选择最长时间不需要访问的页面。于所选择的被淘汰页面将是以后永不使用的，或者是在最长时间内不再被访问的页面，这样可以保证获得最低的缺页率。这一种理想的 页面置换 算法，因为在置换之前，是不知道需要后面需要的页面的。

b） 先进先出置换算法(FIFO):先进入主存的页面先淘汰。其理由是：最早调入主存的页面不再被使用的可能性最大。
FIFO 算法实现简单，但是性能较差。

c） 最长最久未访问算法(LRU Least Recently Used)：利用局部性原理，根据一个作业在执行过程中过去的页面访问历史来推测未来的行为。本质就是 当需要淘汰一个页面时，总是选择一个最近一段时间内最久未被使用的页面予以淘汰。
LRU 性能接近 OPT,但是实现比较困难且开销大。

d） 时钟置换算法(clock)：在 LRU 基础上的改进，增加一个使用位，对于页替换算法，用于替换的候选帧集合看做一个循环缓冲区，并且有一个指针与之相关联。当某一页被替换时，该指针被设置成指向缓冲区中的下一帧。当需要替换一页时，操作系统扫描缓冲区，以查找使用位被置为 0 的一帧。每当遇到一个使用位为 1 的帧时，操作系统就将该位重新置为 0；如果在这个过程开始时，缓冲区中所有帧的使用位均为 0，则选择遇到的第一个帧替换；如果所有帧的使用位均为 1,则指针在缓冲区中完整地循环一周，把所有使用位都置为 0，并且停留在最初的位置上，替换该帧中的页。由于该算法循环地检查各页面的情况，故称为 CLOCK 算法，又称为最近未用(Not Recently Used, NRU)算法。

1. 最近未被访问，也未被修改 u = 0， m = 0
2. 最近被访问， 但未被修改 u = 1， m = 0
3. 最近未被访问，但被修改 u = 0， m = 1
4. 最近被访问， 同时被修改 u = 1， m = 1

算法执行如下操作步骤：

1） 从指针的当前位置开始，扫描帧缓冲区。在这次扫描过程中，对使用位不做任何修改。选择遇到的第一个帧(u=0, m=0)用于替换。

2） 如果第 1)步失败，则重新扫描，查找(u=0, m=1)的帧。选择遇到的第一个这样的帧用于替换。在这个扫描过程中，对每个跳过的帧，把它的使用位设置成 0。

3） 如果第 2)步失败，指针将回到它的最初位置，并且集合中所有帧的使用位均为 0。重复第 1 步，并且如果有必要，重复第 2 步。这样将可以找到供替换的帧。

4） 页面分配策略

a） **首次适应算法**：空闲分区以 地址递增 的顺序连接，分配内存时，顺序查找，找到大小满住要求的第一个空闲分区。
b） **最佳适应算法**：空闲分区以 容量递增 的顺序连接，分配内存时，找到大小能满足要求的第一个空闲分区。
c） **最坏/大适应算法**： 空闲分区以 容量递减 的顺序连接，分配内存时，找到大小满足要求的低一个空闲分区。(也即是每次选取最大的空闲分区)
d） **临近适应算法(循环首次适应算法)**：与首次适应算法类似，但是每次分配内存时，从前一次结束的位置开始查找。

5） 工作集：工作集是指在某段时间间隔内，进程实际要访问的页面的集合。
6） 抖动： 刚被淘汰出内存的页面，不久后再次被调入，不久后又再次被淘汰出内存，然后又要访问它，如此反复，使得系统把大部分时间耗费在页面的调入调出上的现象。

a） 原因：产生 抖动 的根本原因是 系统中运行的进程太多，导致分配给每一个内存的五龙窟太少，不能满足正常运行的基本要求，导致进程运行时频繁的出现缺页和调入。
b） 解决：采用局部置换策略，缺页时，只能在分配给自己的内存里进行置换，不允许从其他进程获取新物理块，避免影响其他进程。或是暂停进程。或者将工作集融入调度算法中。

5.9 mysql 引擎

1）InnoDB

InnoDB 是一个事务型的存储引擎，有行级锁定和外键约束。mysql 运行时 InnoDB 会在内存中建立缓冲池，用于缓冲数据和索引。但是它没有保存表的行数。

适用场景：经常更新的表，适合处理多重并发的更新请求。

1） MyISAM

是 mysql 默认的引擎，但是它没有提供对数据库事务的支持，也不支持行级锁定和外键，因此当 insert 或者 update 会锁定整个表。

6 Linux 相关

6.1 Linux 系统卡顿，请问怎么排查：

a） CPU 的使用情况： 使用 top 命令产看，CPU 使用后情况。

b) 内存使用 ： 可以用 free -g 来查看内存使用清理，若 free 栏为 0 或接近 0，表示内存基本被吃完了，因此就需要释放内存。

c) 磁盘使用：可以用 df -h 来查看 磁盘使用情况，若发现磁盘使用率很高，就需要释放磁盘空间，删除一些不必要的文件。 可以使用 du 相关命令查看目录和文件的磁盘占用情况 比如 dh -sh * 查看党情目录和文件使用情况

d) 磁盘 I/O 使用 :可以使用 iostax-x 1 查看磁盘 I/O 情况。

6.2 Linux 开机启动过程？

1) 主机加电自检，加载 BIOS 硬件信息。
2) 读取 MBR 的引导文件(GRUB、LILO)。
3) 引导 Linux 内核。
4) 运行第一个进程 init (进程号永远为 1)。
5) 进入相应的运行级别。
6) 运行终端，输入用户名和密码