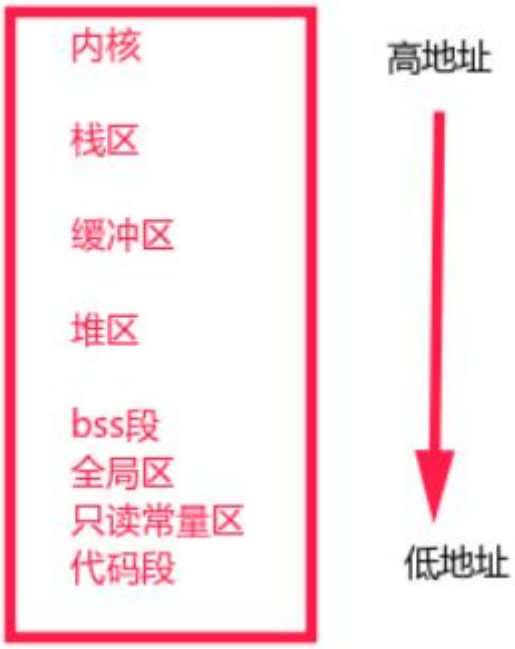
C/C++部分

C++内存分布



一个由C/C++编译的程序占用的内存分为以下几个部分：

1. 栈区（**Stack**）
   由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈
2. 堆区（**heap**）
   一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。不同于数据结构中的堆是两回事，分配方式倒是类似于链表
3. **.bss段（Block Started by Symbol）**
   未初始化的全局变量 / 未初始化的（局部/全局）静态变量(static修饰的变量).
4. 全局区（静态区，**static**）
   初始化的全局变量 / 初始化的（局部/全局）静态变量(static修饰的变量).
5. 常量区
   存放常量，程序结束后由系统释放
6. 代码区（**.text**）
   存放函数体的二进制代码

堆和栈的区别

1. 申请方式和回收方式不同
   栈空间申请是系统自动分配的；堆空间的申请由程序员自己完成
   栈分配的空间由系统自动回收，在函数调用结束时，堆分配的空间由程序员负责释放，若不主动释放，则会一直保留到程序结束由操作系统释放
2. 申请后系统的响应
   栈：只要栈的剩余空间大小大于所申请空间，系统就会为程序提供内存，否则将报异常提示
   堆：系统收到申请时，会遍历记录空闲内存地址的链表，寻找第一个空间大于申请空间的堆节点，然后将该节点从链表中删除，并将节点指向的地址空间分配给程序，在这块内存的首地址处还要记录分配的大小，方便正确释放。因为此节点指向的内存空间不一定等于申请的大小，所以系统还会将多余的部分重新加入空闲链表。
3. 申请大小的限制
   栈的大小是编译时确定的，固定，win下是2M，申请超过大小限制将出现overflow错误
   堆是不连续的内存区域，大小受限于系统中可用的虚拟内存，因此大小要比栈大很多
4. 堆和栈中的存储内容
   栈：
   - 函数返回后要执行的第一条语句地址
   - 函数的各参数，一般顺序为参数表从右往左
   - 函数中的各局部变量

   堆：头部用一字节存放堆的大小，内容自定
5. 存取效率
   栈一般>堆

什么是多态

在面向对象方法中，所谓多态性就是不同对象收到相同消息，产生不同的行为。

C++中，多态性是指用一个名字定义不同的函数，这些函数执行不同但又类似的操作，这样就可以用同一个函数名调用不同内容的函数。即，一个接口，多种方法。

C++中的多态有两种，一种是编译期的，一种是运行期的。

编译期的多态由静态编译负责，比如函数重载和函数模板

运行期的多态由动态编译负责，c++中为虚函数

虚函数

virtual

虚函数表

虚函数表即存放虚函数地址的表，一个含有虚函数的类中，至少有一个虚函数表指针

虚函数表本质是一个存虚函数指针的指针数组，这个数组最后面放了一个nullptr

派生类虚函数表生成的过程：

- 先将基类中的虚函数表内容拷贝一份到派生类中
- 如果派生类重写了基类中的某个虚函数，则用重写的虚函数地址覆盖被重写的虚函数地址
- 派生类自己新增的虚函数按申明次序添加到派生类虚函数表的末尾

多继承派生类中有多个虚函数表，排列方式和继承的顺序一致，派生类自定义的新虚函数将会在第一个类的虚函数表后面进行扩充

重载和重写的区别

重载overload

在同一个类中，函数名相同。参数列表不同，编译器会根据这些函数的不同参数列表，将同名函数名作修饰，从而生成一些不同名称的预处理函数，不体现多态

重写override

也称覆盖，子类重新定义父类中有相同名称相同参数的虚函数，即被重写的函数必须是虚函数，体现了多态

重定义redefining

也称隐藏，子类重新定义父类中有相同名称的非虚函数，参数列表可同可不同，会覆盖父类的同名函数，不体现多态

返回值是否能重载

因为C++重载函数的区分是根据函数名和参数列表相结合完成的，而且调用一个函数这个过程是可以忽略返回值的

虚函数和纯虚函数的区别

- 含有纯虚函数的类被称为抽象类，而只含有虚函数的类不能称为抽象类

- 虚函数可以直接调用，也可以被子类重载以后以多态形式调用，而纯虚函数必须在子类中实现该函数才可以使用
- 虚函数定义为virtual {}，纯虚函数定义为virtual {}=0
- 虚函数必须实现，纯虚函数只是声明
- 如果一个类中含有纯虚函数，那该类不能被实例化

介绍智能指针**

智能指针是一个类，这个类的构造函数中传入一个普通指针，析构函数中释放传入的指针。智能指针的类都是栈上的对象，所以当函数（或程序）结束时会自动被释放

auto_ptr

智能指针可以像类的原始指针一样访问类的public成员，成员函数get()返回一个原始的指针，成员函数reset()重新绑定指向的对象，而原来的对象则会被释放。

不支持复制构造和赋值，只有所有权的转移，因此不能放入容器中，也不能作为函数参数使用，C++11已弃用

unique_ptr

unique_ptr 是一个独享所有权的智能指针，它提供了严格意义上的所有权，包括：

- 拥有它指向的对象
- 无法进行复制构造，无法进行复制赋值操作，但可以进行移动构造和移动赋值操作
- 保存指向某个对象的指针，当它本身被删除释放的时候，会使用给定的删除器释放它指向的对象

unique_ptr相较于auto_ptr，能将动态申请的内存所有权传递给某函数，从某函数返回动态申请内存所有权，在容器中保存指针

总的来说，unique_ptr比auto_ptr更安全

share_ptr

基于引用计数的智能指针，允许多个指针指向同一对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。可以通过成员函数use_count()来查看资源的所有者个数。

当两个对象相互使用一个shared_ptr成员变量指向对方，会造成循环引用，使引用计数失效，从而导致内存泄漏。

weak_ptr

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记数的增加或减少。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。

数组和链表的区别

数组在分配内存的时候是一块连续的空间，并且每个元素的内存大小是一样的，因此可以用下标快速访问；但正因为如此，在其中插入或者删除的操作就比较麻烦，要移动别的元素的位置，因此需要快速访问存取并且不频繁增删就用数组；

链表的每个元素使用指针相互链接，分配的空间比较自由，每个元素可以不同类型不同大小，但是访问就必须链式线扫且没有下标，插入删除比较方便，只用替换和删除指针即可，适合频繁增删的操作需求。

介绍单例模式

单例 Singleton 是设计模式的一种，其特点是只提供唯一一个类的实例,具有全局变量的特点，在任何位置都可以通过接口获取到那个唯一实例;
具体运用场景如：

1. 设备管理器，系统中可能有多个设备，但是只有一个设备管理器，用于管理设备驱动;
2. 数据池，用来缓存数据的数据结构，需要在一处写，多处读取或者多处写，多处读取;

基础要点

- 全局只有一个实例：static特性，同时禁止用户自己声明并定义实例（把构造函数设为 private）
- 线程安全
- 禁止赋值和拷贝
- 用户通过接口获取实例：使用 static 类成员函数

有缺陷的懒汉式

```
class Singleton
{
private:
   Singleton()
   {
      std::cout<<"constructor called!"<<std::endl;
   }

   Singleton(Singleton&)=delete;
   Singleton& operator=(const Singleton&)=delete;
   static Singleton* m_instance_ptr;
public:
   ~Singleton()
   {
     std::cout<<"destructor called!"<<std::endl;
   }

   static Singleton* get_instance()
   {
      if(m_instance_ptr==nullptr){
        m_instance_ptr = new Singleton;
      }
      return m_instance_ptr;
   }

   void use() const {
      std::cout << "in use" << std::endl;
   }
};

Singleton* Singleton::m_instance_ptr = nullptr;

int main()
{
   Singleton* instance = Singleton::get_instance();
   Singleton* instance_2 = Singleton::get_instance();
   return 0;
}
```

存在以下问题：

1. 线程安全问题，多线程获取单例时可能引发竞争条件，需要加锁
2. 内存泄露，没有释放对象，需要使用共享指针

线程安全，内存安全的懒汉式

```
class Singleton
{
public:
   typedef std::shared_ptr<Singleton> Ptr;

   ~Singleton()
   {
      std::cout<<"destructor called!"<<std::endl;
   }

   Singleton(Singleton&)=delete;
   Singleton& operator=(const Singleton&)=delete;

   static Ptr get_instance()
   {
      // "double checked lock"
      if(m_instance_ptr==nullptr){
         std::lock_guard<std::mutex> lk(m_mutex);
         if(m_instance_ptr == nullptr){
           m_instance_ptr = std::shared_ptr<Singleton>(new Singleton);
         }
      }
      return m_instance_ptr;
   }


private:
   Singleton()
   {
      std::cout<<"constructor called!"<<std::endl;
   }

   static Ptr m_instance_ptr;
   static std::mutex m_mutex;
};

// initialization static variables out of class
Singleton::Ptr Singleton::m_instance_ptr = nullptr;
std::mutex Singleton::m_mutex;

int main()
{
   Singleton::Ptr instance = Singleton::get_instance();
   Singleton::Ptr instance2 = Singleton::get_instance();
   return 0;
}
```

1. 使用了共享指针防止内存泄漏，但是约束用户必须使用共享指针
2. 加锁使用互斥量达到线程安全，但是锁具有使用开销

最推荐懒汉式

```
class Singleton
{
public:
   ~Singleton()
   {
      std::cout<<"destructor called!"<<std::endl;
   }

   Singleton(const Singleton&)=delete;
   Singleton& operator=(const Singleton&)=delete;

   static Singleton& get_instance()
   {
      static Singleton instance;
      return instance;

   }
private:
   Singleton()
   {
      std::cout<<"constructor called!"<<std::endl;
   }
};

int main(int argc, char *argv[])
{
   Singleton& instance_1 = Singleton::get_instance();
   Singleton& instance_2 = Singleton::get_instance();
   return 0;
}
```

1. 通过局部静态变量的特性保证了线程安全
2. 不需要使用共享指针
3. 使用时需要声明单例的引用singles&才能获取对象

饿汉式自身便是线程安全的

介绍lambda表达式

C++ 11 中的 Lambda 表达式用于定义并创建匿名的函数对象，以简化编程工作。

Lambda 的语法形式如下：

```
[函数对象参数] (操作符重载函数参数) mutable 或 exception 声明 -> 返回值类型 {函数体}
[ capture list ] ( parameter list) -> return type { function body }
```

[函数对象参数（capture list）]

标识一个 Lambda 表达式的开始，这部分必须存在，不能省略。函数对象参数是传递给编译器自动生成的函数对象类的构造函数的。函数对象参数只能使用那些到定义 Lambda 为止时 Lambda 所在作用范围内可见的局部变量(包括 Lambda 所在类的this)。

函数对象参数有以下形式：

- 空
- = 函数体可以使用Lambda 所在范围内所有可见的局部变量（包括 Lambda 所在类的 this），并且是值传递方式（相当于编译器自动为我们按值传递了所有局部变量）。
- & 函数体内可以使用 Lambda 所在范围内所有可见的局部变量（包括 Lambda 所在类的 this），并且是引用传递方式（相当于是编译器自动为我们按引用传递了所有局部变量）。
- this 函数体内可以使用 Lambda 所在类中的成员变量。
- a 将 a 按值进行传递。按值进行传递时，函数体内不能修改传递进来的 a 的拷贝，因为默认情况下函数是 const 的，要修改传递进来的拷贝，可以添加 mutable 修饰符。
- &a 将 a 按引用进行传递。

(操作符重载函数参数（parameter list）)

标识重载的 () 操作符的参数，没有参数时，这部分可以省略。参数可以通过按值（如:(a, b)）和按引用 (如: (&a, &b)) 两种方式进行传递。

mutable 或 exception 声明

这部分可以省略。按值传递函数对象参数时，加上 mutable 修饰符后，可以修改传递进来的拷贝（注意是能修改拷贝，而不是值本身）。exception 声明用于指定函数抛出的异常，如抛出整数类型的异常，可以使用 throw(int)。

-> 返回值类型（-> return type）

标识函数返回值的类型，当返回值为 void，或者函数体中只有一处 return 的地方（此时编译器可以自动推断出返回值类型）时，这部分可以省略。

{函数体（{ function body }）}

标识函数的实现，这部分不能省略，但函数体可以为空。

析构函数为什么是虚函数

在实现多态时，当用基类操作派生类，在析构时防止只析构基类而不析构派生类的状况发生，

在公有继承中，基类对派生类及其对象的操作，只能影响到那些从基类继承下来的成员。如果想要用基类对非继承成员进行操作，则要把基类的这个函数定义为虚函数。

容器介绍

在数据存储上，有一种对象类型，它可以持有其它对象或指向其它对像的指针，这种对象类型就叫做容器。容器就是保存其它对象的对象。

顺序容器（vector，deque，list）

一种各元素之间有顺序关系的线性表，是一种线性结构的可序群集。顺序性容器不会根据元素的特点排序而是直接保存了元素操作时的逻辑顺序，比如我们一次性对一个顺序性容器追加三 个元素，这三个元素在容器中的相对位置和追加时的逻辑次序是一致的。

关联容器（set，multiset，map，multimap）

关联式容器是非线性的树结构，更准确的说是二叉树结构。各元素之间没有严格的物理上的顺序关系，也就是说元素在容器中并没有保存元素置入容器时的逻辑顺序。但是关联式容器提供了另一种根据元素特点排序的功能，这样迭代器就能根据元素的特点“顺序地”获取元素。关联式容器另一个显著的特点是它是以键值的方式来保存数据，就是说它能把关键字和值关联起来保存。

容器适配器（stack，queue，priority_queue）

容器适配器本质上还是容器，只不过此容器模板类的实现，利用了大量其它基础容器模板类中已经写好的成员函数。

容器迭代器，插入删除时迭代器的状态（顺序容器、无序容器）

顺序容器：vector，list等数组型的结构，插入或删除，迭代器失效；list等链表型结构，插入不会使任何迭代器失效，删除会使指向删除点的迭代器失效

无序容器：树形结构和哈希型结构，插入不会使任何迭代器失效，删除会使指向删除点的迭代器失效

容器适配器不支持迭代器

一个程序突然崩溃该怎么处理

捕捉异常，尽可能定位bug

调用纯虚函数会发生什么

程序终止，输出相关信息

子类重写纯虚函数实在什么时候进行覆盖的

编译期

HashMap和数组的区别

map的底层实现

红黑树

set的特点

set的特性是，所有元素都会根据元素的键值自动被排序，set的元素不像map那样可以同时拥有实值（value）和键值（key），set元素的键值就是实值，实值就是键值。set不允许两个元素有相同的键值。

介绍栈溢出

一般出现于数组开太大，导致栈空间被分配完了

线程有什么执行方式

malloc和new的区别/free和delete的区别

属性

new/delete是C++关键字，需要编译器支持。malloc/free是库函数，需要头文件支持c。

参数

使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算。而malloc则需要显式地指出所需内存的尺寸。

返回类型

new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故new是符合类型安全性的操作符。而malloc内存分配成功则是返回void * ，需要通过强制类型转换将void*指针转换成我们需要的类型。

分配失败

new内存分配失败时，会抛出bac_alloc异常。malloc分配内存失败时返回NULL。

自定义类型

new会先调用operator new函数，申请足够的内存（通常底层使用malloc实现）。然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete先调用析构函数，然后调用operator delete函数释放内存（通常底层使用free实现）。

malloc/free是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。

重载

C++允许重载new/delete操作符，特别的，布局new的就不需要为对象分配内存，而是指定了一个地址作为内存起始区域，new在这段内存上为对象调用构造函数完成初始化工作，并返回此地址。而malloc不允许重载。

内存区域

> 堆是操作系统维护的一块内存，而自由存储是C++中通过new与delete动态分配和释放对象的抽象概念。堆与自由存储区并不等价。

new操作符从自由存储区（free store）上为对象动态分配内存空间，而malloc函数从堆上动态分配内存。自由存储区是C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的对应内存。自由存储区不等于堆，如上所述，布局new就可以不位于堆中。

select,poll,epoll

select

- 监听的文件描述符数量受限
- 轮询扫描，效率较低
- 用户空间和内核空间的复制非常消耗资源

poll

- 与select相似，但连接数不受限
- 水平触发

epoll

- 水平触发 若就绪的事件一次没有处理完要做的事件，就会一直去处理。即就会将没有处理完的事件继续放回到就绪队列之中（即那个内核中的链表），一直进行处理。
- 边缘触发 就绪的事件只能处理一次，若没有处理完会在下次的其它事件就绪时再进行处理。而若以后再也没有就绪的事件，那么剩余的那部分数据也会随之而丢失。
- 不需要主动轮询，被动出发

STL库用过哪些数据结构

vecotr

底层数据结构为数组 ，支持快速随机访问。

常用成员函数：

- size() 返回实际元素个数
- empty() 判断容器中是否有元素，若无元素，则返回 true；反之，返回 false
- front() 返回第一个元素的引用
- back() 返回最后一个元素的引用
- insert() 在指定的位置插入一个或多个元素
- clear() 移出所有的元素，容器大小变为 0

queue

底层一般用list或deque实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时。

常用成员函数：

- empty() 如果 queue 中没有元素的话，返回 true
- size() 返回 queue 中元素的个数

- front() 返回 queue 中第一个元素的引用。如果 queue 是常量，就返回一个常引用；如果 queue 为空，返回值是未定义的。
- back() 返回 queue 中最后一个元素的引用。如果 queue 是常量，就返回一个常引用；如果 queue 为空，返回值是未定义的。
- push(const T& obj) 在 queue 的尾部添加一个元素的副本。这是通过调用底层容器的成员函数 push_back() 来完成的。
- push(T&& obj) 以移动的方式在 queue 的尾部添加元素。这是通过调用底层容器的具有右值引用参数的成员函数 push_back() 来完成的。
- pop() 删除 queue 中的第一个元素。

deque

底层数据结构为一个中央控制器和多个缓冲区，支持首尾（中间不能）快速增删，也支持随机访问

常用成员函数：

- size() 返回实际元素个数。
- empty() 判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。
- front() 返回第一个元素的引用
- back() 返回最后一个元素的引用
- push_back() 在序列的尾部添加一个元素。
- push_front() 在序列的头部添加一个元素。
- pop_back() 移除容器尾部的元素。
- pop_front() 移除容器头部的元素。
- insert() 在指定的位置插入一个或多个元素。
- clear() 移出所有的元素，容器大小变为 0。

stack

底层一般用list或deque实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时。

常用成员函数：

- empty() 当 stack 栈中没有元素时，该成员函数返回 true；反之，返回 false。
- size() 返回 stack 栈中存储元素的个数。
- top() 返回一个栈顶元素的引用，类型为 T&。如果栈为空，程序会报错。
- push(const T& val) 先复制 val，再将 val 副本压入栈顶。这是通过调用底层容器的 push_back() 函数完成的。
- push(T&& obj) 以移动元素的方式将其压入栈顶。这是通过调用底层容器的有右值引用参数的 push_back() 函数完成的。
- pop() 弹出栈顶元素。

map

底层数据结构为红黑树，有序，不重复.

常用成员函数：

- find(key) 在 map 容器中查找键为 key 的键值对，如果成功找到，则返回指向该键值对的双向迭代器；反之，则返回和 end() 方法一样的迭代器。另外，如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
- empty() 若容器为空，则返回 true；否则 false。
- size() 返回当前 map 容器中存有键值对的个数。
- insert() 向 map 容器中插入键值对。
- clear() 清空 map 容器中所有的键值对，即使 map 容器的 size() 为 0。
- 在当前 map 容器中，查找键为 key 的键值对的个数并返回。注意，由于 map 容器中各键值对的键的值是唯一的，因此该函数的返回值最大为 1。

unordered_map

底层数据结构为hash表，无序，不重复

常用成员函数：

- empty() 若容器为空，则返回 true；否则 false。
- size() 返回当前容器中存有键值对的个数。
- find(key) 查找以 key 为键的键值对，如果找到，则返回一个指向该键值对的正向迭代器；反之，则返回一个指向容器中最后一个键值对之后位置的迭代器（如果 end() 方法返回的迭代器）。
- count(key) 在容器中查找以 key 键的键值对的个数。
- insert() 向容器中添加新键值对。
- clear() 清空容器，即删除容器中存储的所有键值对。

set

底层数据结构为红黑树，有序，不重复.

常用成员函数：

- find(val) 在 set 容器中查找值为 val 的元素，如果成功找到，则返回指向该元素的双向迭代器；反之，则返回和 end() 方法一样的迭代器。另外，如果 set 容器用 const 限定，则该方法返回的是 cons t 类型的双向迭代器。
- empty() 若容器为空，则返回 true；否则 false。
- size() 返回当前 set 容器中存有元素的个数。
- insert() 向 set 容器中插入元素。

- clear() 清空 set 容器中所有的元素，即令 set 容器的 size() 为 0。
- count(val) 在当前 set 容器中，查找值为 val 的元素的个数，并返回。注意，由于 set 容器中各元素的值是唯一的，因此该函数的返回值最大为 1。

list

底层数据结构为双向链表，支持快速增删

常用成员函数：

- empty() 判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。
- size() 返回当前容器实际包含的元素个数。
- front() 返回第一个元素的引用。
- back() 返回最后一个元素的引用。
- push_front() 在容器头部插入一个元素。
- pop_front() 删除容器头部的一个元素。
- push_back() 在容器尾部插入一个元素。
- pop_back() 删除容器尾部的一个元素。
- insert() 在容器中的指定位置插入元素。
- clear() 删除容器存储的所有元素。
- remove(val) 删除容器中所有等于 val 的元素。
- remove_if() 删除容器中满足条件的元素。
- unique() 删除容器中相邻的重复元素，只保留一个。
- merge() 合并两个事先已排好序的 list 容器，并且合并之后的 list 容器依然是有序的。
- sort() 通过更改容器中元素的位置，将它们进行排序。

面向对象语言的特点

抽象、封装、继承、多态

操作系统相关

进程、线程、协程之间的区别

对操作系统来说，线程是最小的执行单元，进程是最小的资源管理单元。

进程

进程是正在运行的程序的实例，是一个具有一定独立功能的程序关于某个数据集合的一次运行活动

线程

线程是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。

线程拥有自己的栈空间，是程序的执行者。

协程

协程是一种用户级的轻量级线程。

协程不被操作系统内核所管理，由程序完全控制（工作在用户态）

区别

多进程开发比单进程多线程开发稳定性要强，但是多进程开发比多线程开发资源开销要大

多进程中某一进程出错不会影响其他进程，而多线程中某一线程出错会影响其他线程

进程之间不共享全局变量，线程之间共享全局变量

多线程开发线程之间执行是无序的，协程之间执行按照一定顺序交替执行

协程的资源开销要小于线程

协程的本质是单线程，无法利用多核CPU的优势

进程调度算法

进程间通信的方式

1. 无名管道
   管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；
   管道只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）
   一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。管道大小约64k
2. 有名管道

有名管道不同于无名管道之处在于它提供一个路径名与之关联，以FIFO的文件形式存在于文件系统中，因此参与的进程不必具有亲缘关系，只要可以访问该路径，就能通信，有名管道仍然是半双工，管道大小约64k

3. 信号方式
   通过信号发出和捕捉信息通信
4. 消息队列方式
   单一消息队列依旧无法双向传输，且消息队列随内核持续，是有限资源，单个大小约为16k
5. 共享内存方式
   最快的方式，映射一段能被其他进程所访问的内存
   为避免同时写共享内存发生混乱，可设置主人标志位进行指导操作
6. **Unix套接字方式**
   可实现双向通信，与TCP套接字不同为协议模式不同，一个是网络传输，一个是本地传输，Unix域可以设置阻塞和非阻塞方式，使用select控制读写
7. 文件锁方式
8. 信号量
   信号量是一个计数器，可以用来控制多个进程对共享资源的访问。

生产者消费者模型，多个生产者，一个消费者，消费者无法访问队列时该怎么办

解释物理内存

物理内存指通过物理内存条而获得的内存空间

解释虚拟内存

虚拟内存的目的是为了让物理内存扩充成更大的逻辑内存，从而让程序获得更多的可用内存。为了更好的管理内存，操作系统将内存抽象成地址空间。

虚拟内存允许程序不用将地址空间中的每一页都映射到物理内存，这使得有限的内存运行大程序成为可能

什么是死锁，以及产生条件

死锁产生的四个必要条件：

1. 互斥条件：进程要求对所分配的资源进行排它性控制，即在一段时间内某资源仅为一进程所占用。

2. 请求和保持条件：当进程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不可剥夺条件：进程已获得的资源在未使用完之前，不能剥夺，只能在使用完时由自己释放。
4. 环路等待条件：若干进程之间形成一种头尾相接的循环等待资源的关系

如何预防死锁

内核态和用户态

LRU实现方式

并行并发区别

逻辑地址如何转为物理地址

分页管理和分段管理

数据库相关

介绍redis

数据库范式

好的关系模式具备以下性质：

- 不会发生插入异常、删除异常、更新异常，
- 数据冗余应尽可能少

1NF

如果一个关系模式R的所有属性都是不可分的基本数据项，则R∈1NF

2NF

若R∈1NF，且每一个非主属性完全函数依赖于码，则R∈2NF（消除每一个非主属性对码的部分函数依赖）

3NF

若R∈3NF，则每一个非主属性既不部分依赖于码也不传递依赖于码（）

介绍MySQL数据库索引

索引用于快速找出在某个列中有一特定值的行

优点：

1. 所有的MySql列类型(字段类型)都可以被索引，也就是可以给任意字段设置索引
2. 大大加快数据的查询速度

缺点：

1. 创建索引和维护索引要耗费时间，并且随着数据量的增加所耗费的时间也会增加
2. 索引也需要占空间
3. 当对表中的数据进行增加、删除、修改时，索引也需要动态的维护，降低了数据的维护速度

使用原则：

1. 对经常更新的表就避免对其进行过多的索引，对经常用于查询的字段应该创建索引
2. 数据量小的表最好不要使用索引，因为由于数据较少，可能查询全部数据花费的时间比遍历索引的时间还要短，索引就可能不会产生优化效果
3. 在不同值少的列上(字段上)不要建立索引，比如在学生表的"性别"字段上只有男，女两个不同值。相反的，在一个字段上不同值较多可以建立索引

单列索引

一个索引只包含单个列，但一个表中可以有多个单列索引

普通索引（INDEX）

MySQL中基本索引类型，没有什么限制，允许在定义索引的列中插入重复值和空值，纯粹为了查询数据更快一点

唯一索引（UNIQUE INDEX）

索引列中的值必须是唯一的，但是允许为空值

主键索引（PRIMARY KEY）

是一种特殊的唯一索引，不允许有空值

组合索引（INDEX）

在表中的多个字段组合上创建的索引，只有在查询条件中使用了这些字段的左边字段时，索引才会被使用，使用组合索引时遵循最左前缀集合。

全文索引（FULLTEXT INDEX）

就是在一堆文字中，通过其中的某个关键字等，就能找到该字段所属的记录行

MySQL内部数据的组织方式

计算机网络相关

输入网址访问一个网站的全过程

1. DNS解析
   将域名解析为目的ip地址
2. 建立TCP连接
3. 浏览器向服务器发送http请求
4. 服务器处理http请求，并回复
5. 浏览器解析渲染服务器http回复

什么是http

http叫做超文本传输协议，是一个属于应用层的面向对象的协议，它规定了浏览器和web服务器之间相互通信的规则。

http每次连接只处理一个请求和响应，对每一个页面的访问，浏览器与web服务器都要建立一次单独的连接

http1.1相对于http1.0，允许浏览器在拿到当前请求对应的全部资源后再断开连接，提高了效率

一个完整的http请求包括：

- 请求行
- 首部（若干消息头）
- 实体内容（可无）

例如：

```
//请求行
POST /books/java.html HTTP/1.1
//首部
Accept:*/*
Accept-Language:en-us
Connection:Keep-Alive
Host:localhost
Referer:http://localhost/links.jsp
User-Agent:Mozilla/4.0
Accept-Encoding:gzip,deflate

//实体内容，与消息头之间有一个空行
name=tom&password=123
```

请求行用于描述客户端的请求方式，请求资源名称，以及使用的HTTP协议版本号

首部用于描述客户端请求哪台主机，以及客户端的一些环境信息

一个http响应包括

- 状态行
- 首部（若干消息头）
- 实体内容

状态行用于描述服务器对请求的处理结果

首部用于描述服务器的基本信息，以及数据的描述，服务器通过这些数据的描述信息，可以通知客户端如何处理回送的数据

试题内容代表服务器向客户端回送的数据

常用**通用首部**

代码	说明
Cache-Control	控制缓存的行为
Connection	控制不再转发给代理的首部字段、管理持久连接，即完成后连接是否保持
Date	创建报文的日期时间

代码	说明
Pragma	报文指令
Trailer	报文末端的首部一览
Transfer-Encoding	指定报文主体的传输编码方式
Upgrade	升级为其他协议
Via	代理服务器的相关信息
Warning	错误通知

常用**请求首部**

代码	说明
Accept:text/html,image/*	支持的数据类型
Accept-Charset:utf-8	支持的数据的编码字符集
Accpet-Encoding:gzip	支持的压缩方式
Accept-Language:en-us,zh-cn	支持的语言
Host:localhost:8888	请求的主机名
Referer:http://www.it315.org/index.jsp	发送请求的界面对应的url
User-Agent: Mozilla/4.0	http客户端程序相关信息
Authorization	Web 认证信息
Expect	期待服务器的特定行为
From	用户的电子邮箱地址
If-Match	比较实体标记（ETag）
If-Modified-Since	比较资源的更新时间
If-None-Match	比较实体标记（与 If-Match 相反）
If-Range	资源未更新时发送实体 Byte 的范围请求
If-Unmodified-Since	比较资源的更新时间（与 If-Modified-Since 相反）
Max-Forwards	最大传输逐跳数

代码	说明
Proxy-Authorization	代理服务器要求客户端的认证信息
Range	实体的字节范围请求
TE	传输编码的优先级

常用**响应首部**

代码	说明
Accept-Ranges	是否接受字节范围请求
Age	推算资源创建经过时间
ETag	资源的匹配信息
Location	令客户端重定向至指定 URL
Proxy-Authenticate	代理服务器对客户端的认证信息
Retry-After	对再次发起请求的时机要求
Server	HTTP 服务器的安装信息
Vary	代理服务器缓存的管理信息
WWW-Authenticate	服务器对客户端的认证信息

常用**实体首部**

代码	说明
Allow	资源可支持的 HTTP 方法
Content-Encoding	实体主体适用的编码方式
Content-Language	实体主体的自然语言
Content-Length	实体主体的大小
Content-Location	替代对应资源的 URL
Content-MD5	实体主体的报文摘要
Content-Range	实体主体的位置范围

代码	说明
Content-Type	实体主体的媒体类型
Expires	实体主体过期的日期时间
Last-Modified	资源的最后修改日期时间

http的安全性问题

http协议属于明文传输协议，交互过程以及数据传输都没有进行加密，通信双方也没有进行任何认证，通信过程非常容易遭遇劫持、监听、篡改，严重情况下，会造成恶意的流量劫持等问题，甚至造成个人隐私泄露（比如银行卡卡号和密码泄露）等严重的安全问题

http状态码

1XX

信息性状态码，接受的请求正在处理

- **100 Continue：**表明到目前为止都很正常，客户端可以继续发送请求或者忽略这个响应

2XX

成功状态码，请求正常处理完毕

- **200 OK：** 成功处理了请求
- **204 No Content：**请求已经成功处理，但是返回的响应报文不包含实体的主体部分。一般在只需要从客户端往服务器发送信息，而不需要返回数据时使用
- **206 Partial Content：**表示客户端进行了范围请求，响应报文包含由 Content-Range 指定范围的实体内容

3XX

重定向状态码，需要进行附加操作以完成请求

- **301 Moved Permanently：**永久性重定向
- **302 Found：**临时性重定向
- **303 See Other：**和 302 有着相同的功能，但是 303 明确要求客户端应该采用 GET 方法获取资源

- **304 Not Modified：**如果请求报文首部包含一些条件，例如：If-Match，If-Modified-Since，If-None-Match，If-Range，If-Unmodified-Since，如果不满足条件，则服务器会返回 304 状态码
- **307 Temporary Redirect：**临时重定向，与 302 的含义类似，但是 307 要求浏览器不会把重定向请求的 POST 方法改成 GET 方法

4XX

客户端错误状态码，服务器无法处理请求

- **400 Bad Request：**请求报文中存在语法错误
- **401 Unauthorized：**该状态码表示发送的请求需要有认证信息（BASIC 认证、DIGEST 认证）。如果之前已进行过一次请求，则表示用户认证失败。
- **403 Forbidden：**请求被拒绝
- **404 Not Found：**找不到请求的资源

5XX

- **500 Internal Server Error：**服务器正在执行请求时发生错误
- **503 Service Unavailable：**服务器暂时处于超负载或正在进行停机维护，现在无法处理请求

什么是https

HTTPS 并不是新协议，而是让 HTTP 先和 SSL（Secure Sockets Layer）通信，再由 SSL 和 TCP 通信，也就是说 HTTPS 使用了隧道进行通信。

通过使用 SSL，HTTPS 具有了加密（防窃听）、认证（防伪装）和完整性保护（防篡改）。

https如何实现安全性

HTTPS 采用**混合的加密机制**，使用非对称密钥加密方式，传输对称密钥加密方式所需要的 Secret Key，从而保证安全性;获取到 Secret Key 后，再使用对称密钥加密方式进行通信，从而保证效率。

通过使用**证书**来对通信方进行认证。服务器首先生成公私钥，将公钥提供给相关机构（CA），CA将公钥放入数字证书并将数字证书颁布给服务器，此时服务器就不是简单的把公钥给客户端，而是给客户端一个数字证书，数字证书中加入了一些数字签名的机制，保证了数字证书一定是服务器给客户端的。中间人发送的伪造证书，不能够获得CA的认证，此时，客户端和服务器就知道通信被劫持了。

即 **非对称加密算法（公钥和私钥）交换对称密钥+数字证书验证身份（验证公钥是否是伪造的）+利用对称密钥加解密后续传输的数据**

SSL 提供报文摘要功能来进行完整性保护。HTTPS 的报文摘要功能之所以安全，是因为它结合了加密和认证这两个操作。

介绍http指令

请求方式包括：

- **GET**
  用于向服务器获取信息，可以携带参数，所携带信息通常不能超过4K，不适用于提交大量表单数据
- **POST**
  用于向服务器发送数据，将请求参数放在请求体中，并非URL之后，发送的数据大小无限制
- **HEAD**
  获取报文首部，主要用于确认URL的有效性以及资源更新的日期时间等
- **PUT**
  用于上传文件，自身不带验证机制，存在安全性问题
- **PATCH**
  对资源进行部分修改
- **DELETE**
  用于删除文件，自身不带验证机制
- **OPTIONS**
  查询指定的URL能够支持的方法
- **CONNECT**
  要求在与代理服务器通信时建立隧道，使用SSL和TLS协议把通信内容加密后经网络隧道传输
- **TRACE**
  追踪路径，服务器会将通信路径返回给客户端

GET和POST的区别

GET 用于获取资源，而 POST 用于传输实体主体

GET 和 POST 的请求都能使用额外的参数，但是 GET 的参数是以查询字符串出现在 URL 中，而 POST 的参数存储在实体主体中。

GET 方法是安全的，而 POST 却不是，因为 POST 的目的是传送实体主体内容，这个内容可能是用户上传的表单数据，上传成功之后，服务器可能把这个数据存储到数据库中，因此状态也就发生了改变。

GET是幂等的，而POST不是。

socket绑定0.0.0.0，127.0.1.1的意义

绑定0.0.0.0表示绑定主机上任一地址

绑定127.0.1.1表示绑定本机环回地址，其它计算机无法与本机建立连接

UDP协议

UDP协议全称是用户数据报协议

UDP特点：

面向无连接

首先 UDP 是不需要和 TCP一样在发送数据前进行三次握手建立连接的，想发数据就可以开始发送了。并且也只是数据报文的搬运工，不会对数据报文进行任何拆分和拼接操作。

有单播，多播，广播的功能

UDP 不止支持一对一的传输方式，同样支持一对多，多对多，多对一的方式，也就是说 UDP 提供了单播，多播，广播的功能。

UDP是面向报文的

发送方的UDP对应用程序交下来的报文，在添加首部后就向下交付IP层。UDP对应用层交下来的报文，既不合并，也不拆分，而是保留这些报文的边界。因此，应用程序必须选择合适大小的报文

不可靠性

首先不可靠性体现在无连接上，通信都不需要建立连接，想发就发，这样的情况肯定不可靠。

并且收到什么数据就传递什么数据，并且也不会备份数据，发送数据也不会关心对方是否已经正确接收到数据了。

再者网络环境时好时坏，但是 UDP 因为没有拥塞控制，一直会以恒定的速度发送数据。即使网络条件不好，也不会对发送速率进行调整。这样实现的弊端就是在网络条件不好的情况下可能会导致丢包，但是优点也很明显，在某些实时性要求高的场景（比如电话会议）就需要使用 UDP 而不是 TCP。

头部开销小，传输数据报文时是很高效的。

UDP 头部包含了以下几个数据：

- 两个十六位的端口号，分别为源端口（可选字段）和目标端口
- 整个数据报文的长度
- 整个数据报文的检验和（IPv4 可选 字段），该字段用于发现头部信息和数据中的错误

因此 UDP 的头部开销小，只有八字节，相比 TCP 的至少二十字节要少得多，在传输数据报文时是很高效的

TCP

TCP协议全称是传输控制协议是一种面向连接的、可靠的、基于字节流的传输层通信协议

TCP连接有三次握手，四次挥手

面向连接

面向连接，是指发送数据之前必须在两端建立连接。建立连接的方法是“三次握手”，这样能建立可靠的连接。建立连接，是为数据的可靠传输打下了基础。

仅支持单播传输

每条TCP传输连接只能有两个端点，只能进行点对点的数据传输，不支持多播和广播传输方式。

面向字节流

TCP不像UDP一样那样一个个报文独立地传输，而是在不保留报文边界的情况下以字节流方式进行传输。

可靠传输

对于可靠传输，判断丢包，误码靠的是TCP的段编号以及确认号。TCP为了保证报文传输的可靠，就给每个包一个序号，同时序号也保证了传送到接收端实体的包的按序接收。然后接收端实体对已成功收到的字节发回一个相应的确认(ACK)；如果发送端实体在合理的往返时延(RTT)内未收到确认，那么对应的数据（假设丢失了）将会被重传。

提供拥塞控制

当网络出现拥塞的时候，TCP能够减小向网络注入数据的速率和数量，缓解拥塞

TCP提供全双工通信

TCP允许通信双方的应用程序在任何时候都能发送数据，因为TCP连接的两端都设有缓存，用来临时存放双向通信的数据。当然，TCP可以立即发送一个数据段，也可以缓存一段时间以便一次发送更多的数据段（最大的数据段大小取决于MSS）

使用UDP如何实现可靠传输

为UDP自定义一些类似于TCP的可靠传输的协议

ip包如何辨别tcp和udp

看ip头的8bit协议字段，TCP为6，UDP为17

tcp为什么是三次握手、四次挥手

三次握手，是为了防止已失效的连接请求报文段突然又传送到了服务端从而导致错误。

四次挥手，是因为TCP是全双工的，当c发出FIN报文表示无数据待发送时，s依旧可以向c发送数据，只有s向c发送ack确认报文时，才表示s知道了c没有数据要发。而s没有数据要发送时也要向c发送FIN报文，同理C也要回复ack报文表示知晓，这样才能断开连接

第一、二次握手后，服务端并不知道客户端的接收能力以及自己的发送能力是否正常。

而在第三次握手时，服务端收到了客户端对第二次握手作的回应。从服务端的角度，我在第二次握手时的响应数据发送出去了，客户端接收到了。所以，我的发送能力是正常的。而客户端的接收能力也是正常的。

TCP如何进行流量控制

流量控制：数据的传送与接收过程当中很可能出现收方来不及接收的情况,这时就需要对发方进行控制,以免数据丢失。流量控制用于防止在端口阻塞的情况下丢帧。对发送方发送速率的控制，我们称之为流量控制。

接收方每次收到数据包，可以在发送确定报文的时候，同时告诉发送方自己的缓存区还剩余多少是空闲的，我们也把缓存区的剩余大小称之为接收窗口大小，用变量win来表示接收窗口的大小。

发送方收到之后，便会调整自己的发送速率，也就是调整自己发送窗口的大小，当发送方收到接收窗口的大小为0时，发送方就会停止发送数据，防止出现大量丢包情况的发生。

当发送方收到接受窗口 win = 0 时，这时发送方停止发送报文，并且同时开启一个定时器，每隔一段时间就发个测试报文去询问接收方，打听是否可以继续发送数据了，如果可以，接收方就告诉他此时接受窗口的大小；如果接受窗口大小还是为0，则发送方再次刷新启动定时器。

OSI七层模型

tcp报文头有什么

- 16位源端口号和16位目的端口号

- 32位序号
- 32位确认序号（SYN报文无）
- 4位首部长度+6位保留长度+6位标志位
- 16位窗口大小
- 16位校验和
- 16位紧急指针

总长度为20字节

tcp如何分配报文序号

按照字节流的序号来分配报文序号

比如现在服务端有一个500字节的数据包需要发送到客户端，分两段发送，第一个报文序号初始为1，数据长度为200，那么下一个报文的序号就是201

客户端收到数据包后，向服务端发送确认序号为201的确认包

服务端收到确认包，发送序号为201的数据包，以此类推

四层网络协议

应用层有什么协议

传输层有什么协议

为什么要使用不安全的UDP

追求实时效率而不追求准确性的时候，比如视频聊天，电话会议等

内容来源于：

> https://blog.csdn.net/weixin_43870646/article/details/86575142
> https://blog.csdn.net/weixin_42157608/article/details/80362542
> https://github.com/CyC2018/CS-Notes/blob/master/notes/HTTP.md#缓存
> https://blog.csdn.net/xiaoming100001/article/details/81109617
> https://www.cnblogs.com/jiahuafu/p/8575044.html
> https://blog.csdn.net/yingms/article/details/53188974

> https://blog.csdn.net/studyhardi/article/details/90815766
> https://blog.csdn.net/komtao520/article/details/82424468
> https://www.cnblogs.com/sunchaothu/p/10389842.html
> https://www.cnblogs.com/jimodetiantang/p/9016826.html
> https://blog.csdn.net/crusierLiu/article/details/82626090