算法

数据结构来实现链表

```
struct ListNode
{
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};
```

数据结构实现二叉树

```
struct TreeNode
{
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x): val(x),left(NULL),right(NULL){}    //构造函数
};
```

倒序打印链表

以下是用C++实现倒序打印链表的示例代码：

```
#include <iostream>

struct ListNode {
    int val;
    ListNode* next;

    ListNode(int value) : val(value), next(nullptr) {}
};

void reversePrint(ListNode* head) {
    if (head == nullptr) {
        return;
    }

    reversePrint(head->next);
    std::cout << head->val << " ";
}

int main() {
    // 创建链表
    ListNode* head = new ListNode(1);
    ListNode* node1 = new ListNode(2);
    ListNode* node2 = new ListNode(3);
    ListNode* node3 = new ListNode(4);
```

```
    head->next = node1;
    node1->next = node2;
    node2->next = node3;

    // 倒序打印链表
    std::cout << "Reversed List: ";
    reversePrint(head);

    // 释放链表内存
    while (head != nullptr) {
        ListNode* temp = head;
        head = head->next;
        delete temp;
    }

    return 0;
}
```

上述代码中，我们首先定义了一个`ListNode`结构体来表示链表节点，其中包括一个整数值`val`和指向下一个节点的指针`next`。然后，我们使用递归的方式进行倒序打印。在`reversePrint`函数中，我们先递归调用`reversePrint`函数来打印后续节点，然后再输出当前节点的值。最后，在`main`函数中创建了一个示例链表，并调用`reversePrint`函数来倒序打印链表。最后别忘了释放链表内存，以避免内存泄漏问题。

要将一行数字作为输入存储到vector中

```
#include <iostream>
#include <sstream>
#include <vector>
using namespace std;

int main() {
    string line;
    getline(cin, line);  // 读取一行输入

    istringstream iss(line);  // 创建字符串流
    vector<int> nums;
    int num;

    while (iss >> num) {
        nums.push_back(num);  // 将解析的数字添加到vector中
    }

    // 输出vector中的数字
    for (int i = 0; i < nums.size(); ++i) {
        cout << nums[i] << " ";
    }
    cout << endl;

    return 0;
}
```

在这个示例中，我们首先使用getline函数读取一行输入并存储在名为`line`的字符串中。然后，我们创建一个istringstream对象`iss`，并将`line`传给它。接下来，我们使用while循环从`iss`中逐个解析数字并将它们存储到`nums` vector中。最后，我们使用一个循环来输出存储在`nums` vector中的数字。

请注意，上述代码假设输入的数字之间用空格分隔。如果输入的数字之间使用其他分隔符（例如逗号），你可以在istringstream对象的构造函数中指定分隔符，例如`istringstream iss(line, ',')`。

输出字符串的全排列

一次遍历找到链表中间节点

不使用函数，实现对一个数开立方

求1+2!+3!+....+20!的值

墨菲定律。1,11,21,1211,111221

100个小球，两人拿每次不少一不多于5，你先拿怎么保证第100个是你拿的

100以内的素数

合并两个有序链表（递归与循环）

两个链表组成的数字求和

笔试

```
#include <iostream>
#include<stdio.h>
#include <vector>
#include <unordered_set>
using namespace std;

int main() {
    const char ch[][10] = {"rastar","game","world"};
    printf("%s", ch[0] + 10);
    return 0;
}
```

输出game

C++

哈希表和字典的区别

[哈希表和字典的异同*python哈希表和字典的区别*胖虎是只mao的博客-CSDN博客]

sizeof一个空的结构体，返回什么

因为在 C++ 中，每个对象都必须有一个独一无二的地址，所以即使结构体中没有任何成员，它也必须占用**至少 1 个字节**的空间，以便能够被区分开来。

sizeof("0~9")和strlen("0~9")分别等于多少

```cpp
#include <iostream>
#include<stdio.h>
#include <vector>
#include <unordered_set>
using namespace std;

int main() {
    int a = sizeof("0123456789");
    int b = strlen("0123456789");
    cout << a << " " << b << endl;
}

11 10
```

string写时复制

C++基础之string写时复制（代理模式）*string复制*菜鸟队长2012的博客-CSDN博客

1、Eager Copy(深拷贝)：无论什么情况，都采用拷贝字符串内容的方式解决；这种实现方式，在需要对字符串进行频繁复制而又并不改变字符串内容时，效率比较低下。所以需要对其实现进行优化，之后便出现了COW的实现方式。

2、COW(Copy-On-Write，写时复制）：当两个std::string发生复制或者赋值时，不会复制字符串内容，而是增加一个**引用计数**，然后字符串指针进行**浅拷贝**，其执行效率为O(1)。只有当修改其中一个字符串内容时，才执行真正的复制。即：浅拷贝 + 引用计数。Ubuntu14.04创建的string对象就是写时复制，大小为8。当执行复制或赋值时，引用计数加1，std::string对象共享字符串内容；当std::string对象销毁时，并不直接释放字符串所在的空间，而是先将引用计数减1，直到引用计数为0时，才真正释放字符串内容所在的空间。

内联函数的缺点

1. 代码膨胀：内联函数会在每个调用处插入函数的代码，这**可能导致程序的代码量增大。**如果内联函数的代码较大，或者被频繁调用，会导致可执行文件的大小增加，影响内存的使用效率。
2. **编译时间增加**：由于内联函数需要在每个调用处进行代码替换，编译器需要将函数的定义放置在每个调用点，这会增加编译时间。特别是当一个头文件中的内联函数被多个源文件引用时，编译时间可能会大幅增加。
3. 可维护性下降：内联函数通常是以函数定义的形式出现在头文件中，**函数定义的修改会导致所有引用该头文件的源文件都需要重新编译。**这可能增加代码维护的复杂性，并且容易导致代码一致性问题。
4. 限制性：内联函数要求函数体内部不能有递归调用、循环、异常处理等复杂结构。这是因为在内联函数中，编译器需要在调用点内部展开代码，而这些复杂结构可能导致代码无法正确展开或者引入难以解决的问题。

内存分配的方式

在C++中，有多种方式可以进行内存分配，包括静态内存分配、栈内存分配和动态内存分配。

1. 静态内存分配：
   静态内存分配是指在编译时为变量或对象分配固定大小的内存空间。在C++中，全局变量和静态变量都在程序开始执行时被分配，并在整个程序运行期间一直存在。静态内存分配由编译器完成。例如，以下代码中的变量 `x` 和数组 `arr` 都是静态分配的：

   ```cpp
   int x;
   static int arr[10];
   ```

2. 栈内存分配：
   栈内存分配是指在函数调用时为局部变量分配内存空间。栈是一种后进先出（LIFO）的数据结构，栈帧中的变量在函数调用结束后会被自动释放。栈内存分配由编译器自动管理。例如，以下代码中的变量 `a` 和 `b` 都是栈分配的：

   ```
   void foo() {
       int a;
       int b[5];
       // ...
   }
   ```

3. 动态内存分配：
   动态内存分配是指在程序运行时根据需要动态地分配和释放内存空间。C++中使用 `new` 运算符来进行动态内存分配，使用 `delete` 运算符来释放动态分配的内存空间。动态内存分配适用于需要根据运行时需求进行大小变化的数据结构或对象。例如，以下代码中使用动态内存分配创建一个整型数组：

   ```
   int* arr = new int[10];
   delete[] arr;
   ```

   在动态内存分配过程中，需要注意手动释放内存，否则会导致内存泄漏。

这些不同的内存分配方式在语义上有所区别，并且适用于不同的应用场景。正确选择合适的内存分配方式对于程序的性能和稳定性至关重要。在实际编程中，需要根据具体需求仔细选择合适的内存分配方式。

new可以重载吗，可以改写new函数吗

在C++中，`new` **不能**被直接重载或改写。

尽管 `new` 本身不能被重载或改写，但**可以通过重载类的 `operator new` 和 `operator delete` 成员函数来控制类对象的动态内存分配和释放过程。**这些重载的成员函数允许我们在对象的创建和销毁时进行一些自定义操作。

有了new是不是可以就不用malloc了

不可以，因为可能存在调用C的函数的情况，感觉还是有用的上的时候

new会先调用malloc分配内存，再调用构造函数，只用malloc不会调用构造函数的

如果new一个对象之后，之后不进行其它操作会有什么影响吗

内存泄漏

关于内存泄漏定位的办法

1. 内存检测工具：使用一些专门的内存检测工具，例如**Valgrind（Linux平台）**、Dr. Memory（Windows平台）等。这些工具能够跟踪程序中的内存分配和释放操作，并生成报告，指出内存泄漏的位置。
2. **重载 `new` 和 `delete`：在代码中重载自定义的 `operator new` 和 `operator delete` 运算符**，可以追踪对象的内存分配和释放情况，以及检测是否有未被释放的内存块。
3. 调试工具：**使用调试工具（如GDB、Visual Studio调试器等）**进行调试，在程序运行期间跟踪内存分配和释放的情况，以及查看堆栈信息，找出可能的内存泄漏点。

4. RAII（资源获取即初始化）原则：使用智能指针、容器等资源管理类，在其析构函数中进行资源的释放。通过正确使用这些资源管理类，可以减少手动管理内存的错误，提高代码的健壮性。

重载是在编译期还是在运行期确定

C++的函数重载是在**编译期**确定的。

编译器根据函数的名称、参数类型和数量来进行函数匹配，称为函数重载解析。在编译期间，编译器会根据函数调用的上下文信息来选择合适的重载函数。

多态和继承在什么情况下使用

1. 实现代码复用：通过继承，可以从现有的类派生出新的类，并继承父类的属性和方法。这样可以避免重复编写相似的代码，提高代码复用性。
2. 实现特定类型的抽象概念：**通过继承和多态，可以创建抽象基类，定义通用的接口和行为**，然后派生具体的子类来实现不同的功能。这样可以分离出共享的部分和可变的部分，并且可以通过基类指针或引用来实现多态调用，以增强程序的灵活性和可扩展性。
3. 实现运行时多态性：通过虚函数和动态绑定的机制，可以实现运行时多态性。当基类的指针或引用指向派生类的实例时，可以根据实际对象的类型来调用适当的虚函数，而不是根据指针或引用的静态类型来确定调用哪个函数。这样可以实现基于对象实际类型的灵活调用，为程序提供更好的可扩展性和可维护性。
4. 实现回调机制：通过将派生类对象传递给基类的函数或算法，可以实现回调机制。基类可以使用虚函数来定义通用的接口，派生类可以提供特定的实现，并通过回调机制参与到算法或框架中。

需要注意的是，尽管继承和多态能够提供很多优势，但过度使用或滥用它们可能会导致代码的复杂性增加。

除了多态和继承还有什么面向对象方法

除了多态和继承，面向对象编程还有以下几种常用的方法：

1. 封装（Encapsulation）：封装是将数据和操作封装在一个对象中，通过对外提供公共的接口来控制和访问对象内部的状态和行为。封装可以隐藏内部实现细节，提高代码的模块性和可维护性，同时也可以提供更好的安全性和防护机制。
2. 组合（Composition）：组合是通过将多个对象组合成一个更大的对象来实现复杂的功能。通过将一个对象作为另一个对象的成员变量或属性，可以实现不同对象之间的关联。这种方式可以提高系统的灵活性，因为组合关系可以动态地改变，而且可以将责任分配到不同的对象上。
3. 接口（Interface）：接口定义了一个对象或类提供给外部使用的方法和属性的规范。接口定义了对象的行为和功能，可以用作不同类之间的契约或协议，使得不同类可以以相同的方式与其他对象交互。通过接口，可以实现类之间的解耦和替换，提高代码的可扩展性和可维护性。
4. 抽象（Abstraction）：抽象是从一组具体的事物中提取出共同的特征和行为，形成一个抽象的概念或类别。通过抽象，可以将注意力集中在问题的本质上，忽略细节和实现的差异，提高代码的可读性和可理解性。
5. 反射（Reflection）：反射是一种在运行时获取对象的信息并动态调用其方法或访问其属性的能力。通过反射机制，可以实现动态加载类、创建对象、调用方法等功能，使得程序具有更大的灵活性和适应性。

这些面向对象的方法可以根据具体的情况和需求灵活地组合使用，以实现清晰、模块化和可扩展的代码结构。同时，还可以结合设计原则和模式来指导和优化面向对象的设计和实现。

虚函数可以私有化吗

可以，私有化就是父类指针无法在类外调用虚函数实现多态了

在C++中，虚函数可以声明为私有（private）成员函数，但是它仍然可以通过类的公共接口来调用。

当一个成员函数被声明为虚函数时，它将在类的虚函数表中注册一个入口地址。无论虚函数是私有的还是公共的，都会在虚函数表中有相应的地址。

虚函数的私有化可以用于实现某些特定的设计需求，例如防止外部代码直接调用虚函数，只能通过类的公共接口来访问。这样可以提高类的封装性和安全性，对外隐藏内部实现细节。

下面是一个示例代码，演示了如何将虚函数声明为私有成员，并通过公共接口间接调用该虚函数：

```
#include <iostream>

class Base {
private:
    virtual void foo() {
        std::cout << "Base::foo()" << std::endl;
    }

public:
    void callFoo() {
        foo();  // 通过公共接口调用私有虚函数
    }
};

class Derived : public Base {
private:
    void foo() override {
        std::cout << "Derived::foo()" << std::endl;
    }
};

int main() {
    Base* ptr = new Derived();
    ptr->callFoo();  // 输出：Derived::foo()

    delete ptr;
    return 0;
}
```

在上述示例中，`Base`类的`foo()`函数被声明为私有的虚函数。通过`Base`类中的公共接口`callFoo()`来间接调用私有虚函数`foo()`，最终输出了派生类`Derived`中的实现。

虚函数一定要在子类里面实现吗，可以不重写吗，纯虚函数呢

在C++中，虚函数是可以在子类中进行重写（override），也可以选择不重写。

如果子类不重写基类的虚函数，那么**子类对象**调用该虚函数时，会直接使用基类中定义的实现。这种情况下，子类并未提供自己的实现，而是继承了基类的实现。

纯虚函数是一种特殊的虚函数，它没有具体的实现，只是在基类中进行声明。**子类必须重写（override）纯虚函数，并为其提供具体的实现，否则子类将成为抽象类，无法实例化对象。**

使用纯虚函数可以定义一组接口规范，要求派生类必须提供相应的实现。这样可以达到多态的目的，通过基类指针或引用调用派生类特定的功能。

纯虚函数使用场景有哪些

在C++中，纯虚函数（pure virtual function）是一个没有具体实现的虚函数，它只有函数声明而没有函数体。**纯虚函数的主要作用是定义抽象基类（abstract base class），通过派生类强制实现该函数，以达到接口规范化、多态性的目的。**

静态成员函数可以是const类型吗

不可以，因为const限制访问的是成员变量，但是静态函数本来就不可以访问普通成员变量，因此没必要

一个对象=另一个对象会发生什么

拷贝构造函数

vector有哪些增加的方式?

push_back() emplace_back()

vector增加元素是值拷贝还是指针拷贝?

值拷贝

看代码

```
#include <iostream>
using namespace std;
class A
{
public:
    A() { cout << "A::A()called.\n"; }
    virtual ~A() { cout << "A:~A()called.\n"; }
};

class B : public A
{
public:
    B(int i)
    {
        cout << "B::B()called.\n";
        buf = new char[i];
    }
    virtual ~B()
    {
        delete[] buf;
        cout << "B:~B()called.\n";
    }

private:
    char *buf;
};

void fun(A *a)
{
    delete a;
}

int main()
```

```
{
    A *a = new B(15);
    fun(a);
    return 0;
}
//程序输出：
A::A()called.
B::B()called.
B:~B()called.
A:~A()called.
```

父类对象会在子类之前进行构造；销毁一个对象时，先调用子类的析构函数，再调用父类的析构函数。

使用智能指针了普通指针还能用吗，什么时候用智能指针

1. 动态分配的单个对象：当需要动态分配单个对象，并在使用完毕后自动释放内存时，可以使用 `std::unique_ptr`。它独占所指向的对象的所有权，一旦它离开其作用域或被显式地释放，它就会自动删除所指向的对象。
2. 动态分配的对象数组：如果要动态分配对象数组，并希望在使用完毕后自动释放内存，则可以使用 `std::unique_ptr` 与自定义删除器 `std::default_delete[]` 结合使用。
3. 共享拥有的对象：当需要多个指针共同拥有一个对象，并且在最后一个指针离开其作用域时才释放内存时，可以使用 `std::shared_ptr`。它使用引用计数的方式追踪所指向对象的所有权，当最后一个 `std::shared_ptr` 离开其作用域时，所指向的对象会被自动删除。
4. 避免内存泄漏和悬挂指针：由于智能指针具有自动释放内存的功能，可以帮助避免常见的内存泄漏和悬挂指针问题。使用智能指针可以减少手动管理内存的复杂性，提高代码的安全性和可靠性。

写代码实现一个shared_ptr

在 C++ 中，可以使用标准库中的 `std::shared_ptr` 来创建和管理共享拥有的对象。以下是一个示例代码，演示了如何使用 `std::shared_ptr`：

```
#include <iostream>
#include <memory>

class MyClass {
public:
    MyClass() {
        std::cout << "MyClass Constructor" << std::endl;
    }

    ~MyClass() {
        std::cout << "MyClass Destructor" << std::endl;
    }

    void doSomething() {
        std::cout << "Doing something..." << std::endl;
    }
};

int main() {
    // 创建一个 shared_ptr，指向一个 MyClass 对象
    std::shared_ptr<MyClass> sharedPtr(new MyClass());

    // 使用 shared_ptr 调用对象的成员函数
    sharedPtr->doSomething();
```

```
    // 复制 shared_ptr
    std::shared_ptr<MyClass> anotherSharedPtr = sharedPtr;

    // 输出引用计数
    std::cout << "Reference count: " << sharedPtr.use_count() << std::endl;

    // 当最后一个 shared_ptr 离开其作用域时，对象会被自动删除
    return 0;
}
```

在上面的代码中，首先定义了一个名为 `MyClass` 的简单类。然后在 `main` 函数中，通过 `std::shared_ptr` 创建了一个指向 `MyClass` 对象的智能指针 `sharedPtr`。可以使用箭头运算符 `->` 来调用 `sharedPtr` 所指向的对象的成员函数。接下来，将 `sharedPtr` 复制给另一个 `std::shared_ptr` 对象 `anotherSharedPtr`，并使用 `use_count` 函数获取引用计数。最后，由于 `sharedPtr` 和 `anotherSharedPtr` 离开其作用域，所指向的对象会被自动删除。

在实际的代码中，可以根据具体需求在构造和使用 `std::shared_ptr` 时进行适当的调整。请注意，`std::shared_ptr` 会通过引用计数来追踪所指向对象的所有权，并在最后一个 `std::shared_ptr` 离开其作用域时释放该对象。这样可以避免了手动管理内存的复杂性，并提供了更安全的内存管理机制。

shared_ptr是线程安全的吗

c++11总结15——shared_ptr在多线程下的安全性问题*shared_ptr多线程*却道天凉_好个秋的博客-CSDN博客

shared_ptr的引用计数是怎么实现的

见上一题

三个智能指针的应用场景

1. `std::unique_ptr`：
   `std::unique_ptr` 提供了独占所有权的智能指针，不能被复制或共享。它适用于以下场景：
   - 在函数返回时将对象所有权转移给调用者。
   - **管理堆上动态分配的单个对象。**
   - 在容器中存储对象，以及需要**在代码块结束时自动释放对象资源。**
2. `std::shared_ptr`：
   `std::shared_ptr` 提供了共享所有权的智能指针，可以多个指针共同拥有一个对象。它适用于以下场景：
   - **在多个对象之间共享所有权，避免内存泄漏。**
   - 构建循环引用的数据结构，如树状结构或图结构。
   - 实现观察者模式，其中有一个被观察的对象和多个观察者对象。
3. `std::weak_ptr`：
   `std::weak_ptr` 是一种弱引用智能指针，它允许访问由 `std::shared_ptr` 持有的对象，但不会引起对象引用计数的增加。它适用于以下场景：
   - **防止 `std::shared_ptr` 的循环引用问题，避免内存泄漏。**
   - **通过 `std::weak_ptr` 判断对象是否已经释放或销毁。**
   - 在需要临时访问 `std::shared_ptr` 所持有对象的场景，而不增加引用计数。

怎么debug，怎么看内存泄漏

工具Valgrind

检测内存泄漏的关键原理就是，检查malloc/new和free/delete是否匹配，一些工具也就是这个原理。要做到这点，就是利用宏或者钩子，在用户程序与运行库之间加了一层，用于记录内存分配情况。

如果new了之后出了问题直接return，会导致内存泄漏，怎么办

智能指针，RAII

移动构造函数，移动语义move()

C++11——移动构造函数及std::move() 的使用c++11移动构造函数的功能和用法ShenHang 的博客-CSDN博客

模板是怎么实现转化成不同类型的

显式实例化和隐式实例化

C++string字符串怎么拷贝到一个char的buf里面

将C++的`std::string`拷贝到一个`char`类型的缓冲区可以使用`strcpy()`或者`memcpy()`函数。这两个函数都需要传入目标缓冲区的指针和源字符串的指针。

下面是使用这两个函数进行拷贝的示例：

```
#include <iostream>
#include <cstring> // 包含字符串操作相关的头文件

int main() {
    std::string str = "Hello, World!";

    // 使用 strcpy()
    char buf1[20];
    strcpy(buf1, str.c_str());
    std::cout << "使用 strcpy() 拷贝的字符串：" << buf1 << std::endl;

    // 使用 memcpy()
    char buf2[20];
    memcpy(buf2, str.c_str(), str.length() + 1); // 需要拷贝 null 终止字符 '\0'
    std::cout << "使用 memcpy() 拷贝的字符串：" << buf2 << std::endl;

    return 0;
}
```

在上面的示例中，`str.c_str()`返回一个指向`std::string`中字符数组的指针。`strcpy()`和`memcpy()`函数分别将该字符数组拷贝到目标缓冲区`buf1`和`buf2`中。

请确保目标缓冲区足够大以容纳源字符串，同时也要考虑空字符('\0')的拷贝，因此目标缓冲区的大小通常要大于或等于源字符串的长度。
另外，如果你的 C++ 环境支持 C++11 或以上版本，你也可以使用`std::copy()`算法来进行拷贝：

```
#include <iostream>
#include <algorithm> // 包含算法操作相关的头文件

int main() {
    std::string str = "Hello, World!";

    char buf[20];
```

```
    std::copy(str.begin(), str.end(), buf);
    buf[str.size()] = '\0'; // 保证缓冲区以空字符('\0')结尾

    std::cout << "拷贝的字符串：" << buf << std::endl;

    return 0;
}
```

这种方式利用迭代器对源字符串进行迭代，并使用 `std::copy()` 将每个字符拷贝到目标缓冲区 `buf` 中。最后，为了确保缓冲区以空字符('\0')结尾，需要手动添加一个空字符。

在平时写C++代码的过程中，遇到过哪些坑，又是如何去解决的

在C++编程中，有一些常见的坑可能会导致程序出现问题。下面是一些常见情况以及如何解决它们的示例：

1. 野指针和空指针访问：使用未初始化的指针或访问空指针会导致不可预测的结果或崩溃。解决方法是始终将指针初始化为nullptr，并在使用指针之前进行有效性检查。

```
int* ptr = nullptr; // 初始化指针为nullptr

// 访问指针之前进行有效性检查
if (ptr != nullptr) {
    // do something with ptr
}
```

2. 内存泄漏：忘记释放动态分配的内存会导致内存泄漏。使用智能指针（例如std::unique_ptr、std::shared_ptr）可以自动管理内存生命周期，避免手动释放内存。

```
std::unique_ptr<int> ptr(new int(5)); // 使用 std::unique_ptr 管理动态内存

// 不需要手动释放内存，智能指针会在不再使用时自动释放
```

3. 数组越界访问：访问数组时，超出其边界范围会导致访问非法内存，可能引发崩溃或数据损坏。解决办法是确保在访问数组元素之前检查索引的有效性。

```
int arr[5] = {1, 2, 3, 4, 5};

// 确保索引有效性
int index = 6;
if (index >= 0 && index < 5) {
    int value = arr[index];
}
```

4. 悬空指针：在释放内存后，将指针指向已释放的内存块会导致悬空指针。解决方法是在释放内存后，将指针设置为nullptr。

```
int* ptr = new int(5);
delete ptr;
ptr = nullptr; // 避免成为悬空指针
```

5. 隐式类型转换问题：有时候隐式类型转换可能导致意外的结果。为了避免这种情况，应显式进行类型转换或使用更明确的代码。

```
double num = 3.14;
int result = static_cast<int>(num); // 显式类型转换

int num1 = 10;
double result1 = static_cast<double>(num1); // 显式类型转换
```

6. 字符串操作错误：对C风格字符串的操作需谨慎，如未正确处理空字符('\0')可能导致字符串处理错误。使用C++的`std::string`类来避免这个问题，它提供了更安全和方便的字符串操作接口。

```
std::string str = "Hello";
char c = str[5]; // 不会导致访问越界，而是返回'\0'

// 需要处理空字符('\0')的情况
char buffer[100];
strcpy(buffer, str.c_str());
```

这些只是一些常见的问题和解决方法示例，当然还有其他许多潜在的陷阱。为了避免这些问题，需要仔细阅读文档、理解C++的最佳实践，并进行适当的测试和调试。

自己写代码的时候，遇到bug如何调试

GDB

一个函数f(int a, int b)的b和a的地址关系

通常情况下，参数`a`和`b`在函数调用时被存储在栈上。栈是一种用于存储局部变量和函数参数的内存区域。在栈上，每个局部变量和函数参数都会分配一个地址。

一种典型的情况是，参数`a`的地址较低，参数`b`的地址较高。这是因为参数在栈上的分配通常是按照**从右到左**的顺序进行的。但是需要注意的是，这只是一种可能的情况，真正的地址关系还取决于编译器和编译选项。

以下是一个示例程序，可以输出`a`和`b`的地址来观察它们的关系：

```
#include <iostream>

void f(int a, int b) {
    std::cout << "Address of a: " << &a << std::endl;
    std::cout << "Address of b: " << &b << std::endl;
}

int main() {
    int x = 1;
    int y = 2;
    f(x, y);
    return 0;
}
```

运行以上代码的输出可能如下所示：

```
Address of a: 0x7ffee69b9a3c
Address of b: 0x7ffee69b9a38
```

lambda

C ++ Lambda表达式详解c++ lambda奥修的灵魂的博客-CSDN博客

lamda表达式捕获列表捕获的方式有哪些

值传递，引用传递

如果是一个引用捕获，要注意什么

生命周期

在C++中，当在lambda函数的捕获列表中使用引用捕获时，需要注意以下几点：

1. 引用的有效性：确保引用所绑定的对象在lambda函数执行期间仍然有效。如果引用指向的对象在lambda函数执行期间被销毁，那么操作这个引用将导致未定义行为。
2. 引用捕获的生命周期：引用捕获的生命周期取决于所捕获的变量的生命周期。确保在引用被使用之前，所捕获的变量仍然有效。
3. 引用捕获的可变性：默认情况下，引用捕获是不可变的，即无法通过引用来修改所捕获的变量。如果需要在lambda函数中修改所捕获的变量，可以使用`mutable`关键字来声明lambda函数可变，并通过引用来修改变量的值。

下面是一个示例代码，演示了引用捕获的注意事项：

```
#include <iostream>

int main() {
    int x = 10;
    int y = 20;

    auto lambda = [&]() mutable {
        // 使用引用来修改所捕获的变量
        x++;
        y++;

        std::cout << "Inside lambda: x = " << x << ", y = " << y << std::endl;
    };

    lambda(); // 调用lambda函数

    std::cout << "Outside lambda: x = " << x << ", y = " << y << std::endl;

    return 0;
}
```

输出结果将是：

```
Inside lambda: x = 11, y = 21
Outside lambda: x = 10, y = 20
```

需要注意的是，即使在lambda函数内部通过引用捕获对`x`和`y`进行了修改，但这些修改不会影响到外部的`x`和`y`的值。这是因为在示例中使用了`mutable`关键字，允许对所捕获的变量进行修改，并且lambda函数内部的修改只对lambda函数本身可见。

总结来说，当在捕获列表中使用引用捕获时，要确保引用指向的对象在lambda函数执行期间仍然有效，关注捕获的变量的生命周期并考虑是否需要使用`mutable`关键字来修改所捕获的变量。

操作系统

怎么代码实现一个死锁

一种是单线程对一个资源重复申请上锁；第二种是两个线程对两个资源申请上锁，形成环路。

```
#include <iostream>
#include <thread>
#include <mutex>
#include <unistd.h>

using namespace std;

int data = 1;
mutex mt1,mt2;

void a2() {
    data = data * data;
    mt1.lock();  //第二次申请对mt1上锁，但是上不上去
    cout<<data<<endl;
    mt1.unlock();
}
void a1() {
    mt1.lock();  //第一次对mt1上锁
    data = data+1;
    a2();
    cout<<data<<endl;
    mt1.unlock();
}

int main() {
    thread t1(a1);
    t1.join();
    cout<<"main here"<<endl;
    return 0;
}
```

```
#include <iostream>
#include <thread>
#include <mutex>
#include <unistd.h>

using namespace std;

int data = 1;
mutex mt1,mt2;

void a2() {
    mt2.lock();
    sleep(1);//可以理解为在sleep期间，执行了a1
    data = data * data;
    mt1.lock();  //此时a1已经对mt1上锁，所以要等待
    cout<<data<<endl;
    mt1.unlock();
    mt2.unlock();
}
```

```
void a1() {
    mt1.lock();
    sleep(1);
    data = data+1;
    mt2.lock();  //此时a2已经对mt2上锁，所以要等待
    cout<<data<<endl;
    mt2.unlock();
    mt1.unlock();
}

int main() {
    thread t2(a2);
    thread t1(a1);

    t1.join();
    t2.join();
    cout<<"main here"<<endl;  //要t1线程、t2线程都执行完毕后才会执行
    return 0;
}
```

上述代码执行后3处cout都不会打印信息，a1和a2互相死等，形成死锁。

三个线程A，B，C，顺序打印；三个线程，依次打印1-100

std::condition_variable是条件变量，当 std::condition_variable对象的某个wait 函数被调用的时候，它使用 std::unique_lock(通过 std::mutex) 来锁住当前线程。当前线程会一直被阻塞，直到另外一个线程在相同的 std::condition_variable 对象上调用了 notification 函数来唤醒当前线程。

```
#include <thread>
#include <iostream>
#include <mutex>
#include <condition_variable>

std::mutex data_mutex;
std::condition_variable data_var;
int flag=1;
int i=0;
#define MAXNUM 100

void printA(){
    while(i<MAXNUM){
//        std::this_thread::sleep_for(std::chrono::seconds(1)); //打印节奏变慢
        std::unique_lock<std::mutex> lck(data_mutex);
        //当flag为1时，wait不阻塞；否则wait阻塞当前线程。
        //允许其他锁住的线程继续，当其他线程调用notify函数后，并且flag为1时
        //此线程被唤醒
        data_var.wait(lck, []{return flag==1;});
        flag=2;
        if(i >= 100){
            data_var.notify_all();
            break;
        }
        i++;
        std::cout<<"threadA:"<<i<<std::endl;
        data_var.notify_all();
    }
    std::cout<<"finish A"<<std::endl;
```

```
}

void printB(){
    while(i<MAXNUM){
        std::unique_lock<std::mutex> lck(data_mutex);
        //当flag为2时，wait不阻塞；否则wait阻塞当前线程。
        //允许其他锁住的线程继续，当其他线程调用notify函数后，并且flag为2时
        //此线程被唤醒
        data_var.wait(lck, []{return flag==2;});
        flag = 3;
        if(i >= 100){
            data_var.notify_all();
            break;
        }
        i++;
        std::cout<<"threadB:"<<i<<std::endl;
        data_var.notify_all();
    }
    std::cout<<"finish B"<<std::endl;
}

void printC(){
    while(i<MAXNUM){
        std::unique_lock<std::mutex> lck(data_mutex);
        //当flag为3时，wait不阻塞；否则wait阻塞当前线程。
        //允许其他锁住的线程继续，当其他线程调用notify函数后，并且flag为3时
        //此线程被唤醒
        data_var.wait(lck, []{return flag==3;});
        flag = 1;
        if(i >= 100){
            data_var.notify_all();
            break;
        }
        i++;
        std::cout<<"threadC:"<<i<<std::endl;
        data_var.notify_all();
    }
    std::cout<<"finish C"<<std::endl;
}

int main(){
    std::thread tA(printA);
    std::thread tB(printB);
    std::thread tC(printC);
    tA.join();
    tB.join();
    tC.join();
    std::cout << i << std::endl;//最后验证i是否严格等于100
    system("pause");
    return 0;
}
```

```
...前面的93个打印行省略不写了
threadA:94
threadB:95
threadC:96
threadA:97
threadB:98
threadC:99
threadA:100
finish A
finish C
finish B
100
请按任意键继续. . .
```

进程上下文切换的过程

[进程的上下文切换*进程上下文切换的过程*刚好五岁的博客-CSDN博客]

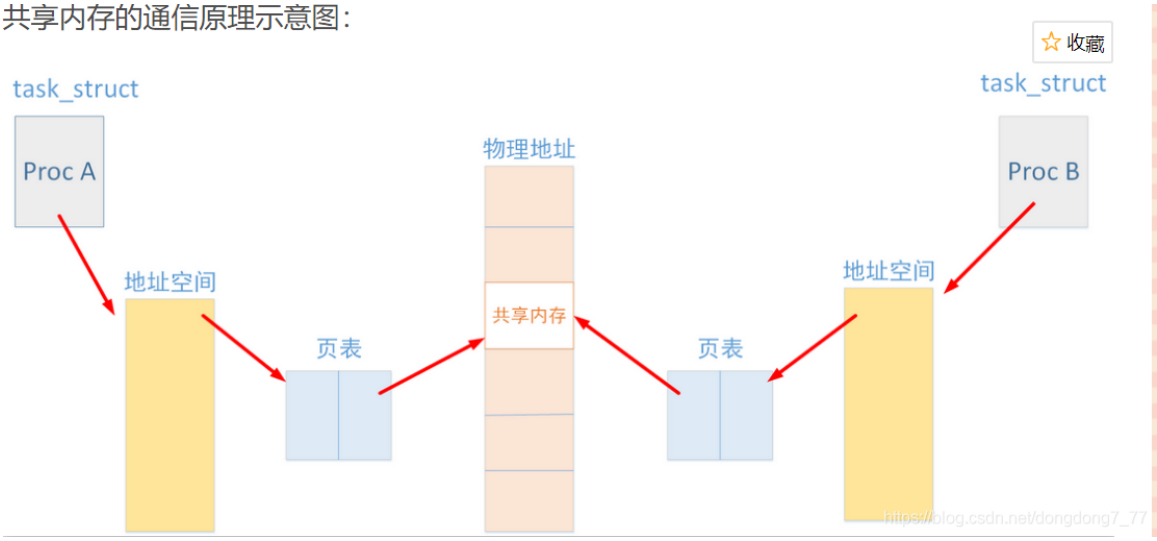
操作系统中的原子操作是怎么实现的

操作系统中的原子操作**通过硬件提供的原子性指令和中断屏蔽，以及操作系统提供的锁机制、原子操作指令和临界区等手段来实现。**这些机制保证了原子操作的不可中断性和执行的完整性。

共享内存是如何确定物理地址的

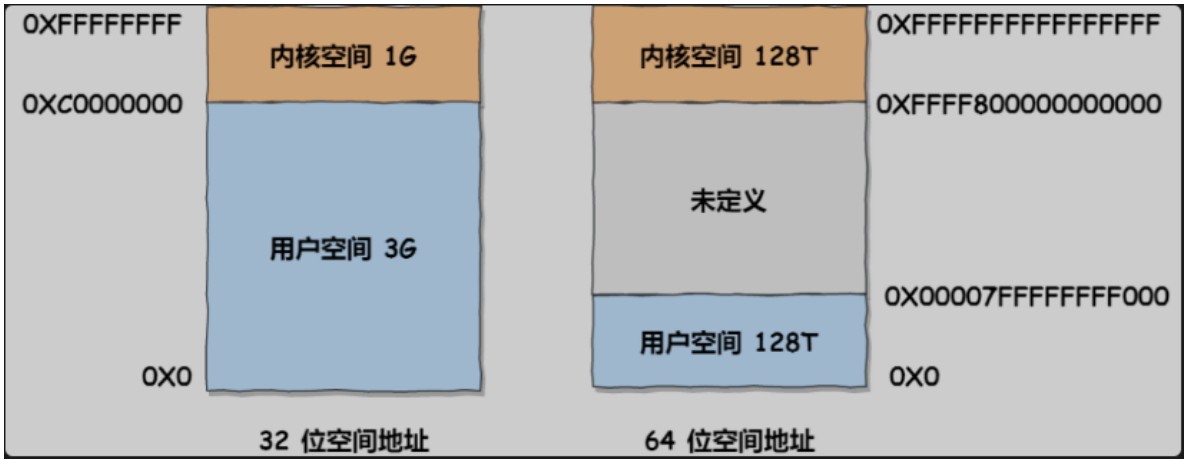
在Linux中，每个进程都有属于自己的进程控制块（PCB）和地址空间（Addr Space），并且都有一个与之对应的页表，负责将进程的虚拟地址与物理地址进行映射，通过内存管理单元（MMU）进行管理。**两个不同的虚拟地址通过页表映射到物理空间的同一区域，它们所指向的这块区域即共享内存。**

共享内存的通信原理示意图：



2G的内存加载4G的文件是否能加载成功

32位操作系统和64位操作系统的虚拟地址空间大小是不同的，在Linux操作系统中，虚拟地址空间的内部又被分为内核空间和用户空间两部分：



32位系统的内核空间占用1G，位于最高处，剩下的3G是用户空间；
64位系统的内核空间和用户空间都是128T,分别占据整个内存空间的最高和最低处，剩下的中间部分是未定义的。
在32位操作系统上，进程最多申请3GB大小的虚拟内存空间，所以进程申请4GB内存的话，在申请虚拟内存阶段就会失败(可能错误原因是OOM)

在64位操作系统，进程可以使用128T大小的虚拟内存空间，所以进程申请4GB内存是没问题的，因为进程申请内存是申请虚拟内存，只要不读写这个虚拟内存，操作系统就不会分配物理内存。

CPU32位支持最大内存

32位的CPU最大支持的内存容量为4GB（2^32 bytes）。这是因为在32位系统中，每个内存地址用32个比特表示，最多可以表示2^32个不同的地址。由于每个地址对应一个字节，所以最大寻址能力为2^32字节，即4GB。

然而，实际上可用于进程的内存可能会小于4GB。这是因为在32位操作系统中，内存空间要被分配给各种系统资源，例如显卡显存、BIOS、固件等。因此，通常只有一部分地址空间被留给进程使用，实际可用内存可能在2GB到3GB之间。

如果需要使用更大容量的内存，需要使用64位的操作系统和CPU。64位系统具有更宽的地址空间，因此可以支持更大的内存容量，可以达到几TB甚至更多。

进程状态是怎么转变的，进程会不会睡眠，哪个阶段

会睡眠，送到总线到显示器打印输出的这个阶段内会睡眠

线程同步中的互斥锁，在抢锁失败的情况下会主动放弃CPU进入睡眠状态，等到锁的状态发生改变时再将其唤醒。

在操作系统中，进程可以主动选择进入睡眠状态（Sleep）。当一个进程需要等待一段时间或者特定条件满足时，可以主动调用睡眠函数，使自己暂停执行，直到等待的条件满足后再继续执行。

在上面提到的情况中，当进程需要将输出发送到显示器打印时，它可能会通过总线与显示器进行通信。在这个阶段内，当进程向显示器发送数据时，它可能需要等待一段时间，以确保显示器已准备好接收数据并完成处理。在这种情况下，进程可能会选择主动进入睡眠状态，等待显示器准备好接收数据。

在睡眠状态下，操作系统会将CPU资源分配给其他就绪状态的进程来执行，从而提高CPU利用率。一旦显示器准备好接收数据，操作系统会唤醒进程，使其恢复执行，并将数据发送到显示器进行打印输出。

需要注意的是，进程的睡眠与阻塞状态是不同的。睡眠是由进程自己主动选择进入的状态，而阻塞是由于等待某个事件的发生而被动进入的状态。在阻塞状态下，进程无法继续执行，直到等待的事件发生。而睡眠状态则是进程根据自身需要选择主动暂停执行的状态。

总线在其中发挥的作用

总线（Bus）是计算机系统中连接各个组件和设备之间传输数据和信号的物理通道。它扮演着信息传递和协调各个部件之间通信的重要角色。

总线有多种类型，包括数据总线、地址总线和控制总线。

1. 数据总线（Data Bus）：数据总线用于在计算机系统中传输数据。它可以是单向的或双向的，负责在计算机的不同部件之间传递二进制数据。
2. 地址总线（Address Bus）：地址总线用于指定或传输内存或IO设备的地址。它确定了计算机可以访问的内存范围。通过地址总线，CPU可以确定读取或写入数据的内存位置。
3. 控制总线（Control Bus）：控制总线用于传输控制信号，例如读写控制、时钟信号、中断请求等。它负责向系统中的各个部件发送控制信息，以实现数据的传输和处理。

总线的作用在于提供了一种标准化的通信接口，使得计算机系统中的各个部件能够相互连接和交流。它简化了计算机的设计，实现了各个组件之间的数据传输和控制，促进了系统的协同工作和整体性能的提升。

在上述问题中，提到了进程将数据发送到显示器进行打印输出，这涉及到进程与显示设备之间的通信。总线在这里起到了承载数据传输的作用，通过总线，进程可以将数据发送给显示设备，实现打印输出的功能。

swap分区是做什么的

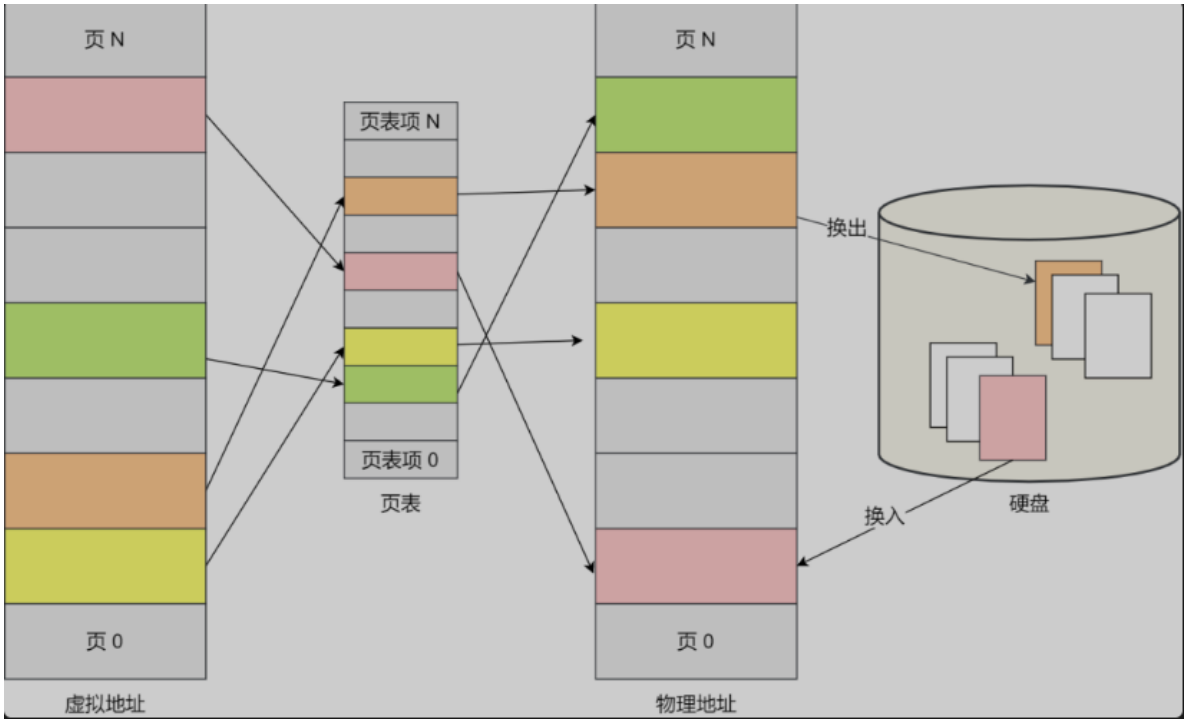
当系统的物理内存不够用的时候，就需要将物理内存中的一部分空间释放出来，以供当前运行程序使用。那些被释放的空间可能来自一些很长时间没有什么操作的程序，这些被释放的空间会被临时保存到磁盘，等到那些程序要运行时，再从磁盘中恢复数据到内存中。

当内存使用存在压力的时候，会开始触发内存回收行为，会把这些不常访问的内存先写到磁盘中，然后释放这些内存，给其他更需要的进程使用。再次访问这些内存时，重新从磁盘中读入内存就可以了。

这种，将内存数据换出磁盘，又将磁盘中恢复数据到内存的过程，就是Swap机制负责的。

Swap就是把一块磁盘空间或者本地文件，当成内存来使用，包含换出和换入两个过程：

- 换出(Swap Out),是把进程暂时不用的内存数据存储到磁盘中，并释放这些数据占用的内存
- 换入(Swap In),是在进程再次访问这些内存的时候，把它们从磁盘读到内存中来



使用 Swap 机制优点是，应用程序实际可以使用的内存空间将远远超过系统的物理内存。由于硬盘空间的价格远比内存要低，因此这种方式无疑是经济实惠的。当然，频繁地读写硬盘，会显著降低操作系统的运行速率，这也是 Swap 的弊端。

Linux 提供了两种不同的方法启用 Swap，分别是 Swap 分区（Swap Partition）和 Swap 文件（Swapfile）：

- Swap 分区是硬盘上的独立区域，该区域只会用于交换分区，其他的文件不能存储在该区域上，我们可以使用 Swapon -s 命令查看当前系统上的交换分区；
- Swap 文件是文件系统中的特殊文件，它与文件系统中的其他文件也没有太多的区别。

网络

TCP三次握手，第一次、第二次、第三次丢失都怎么办

TCP 才不傻：三次握手和四次挥手的异常处理 - 知乎 (zhihu.com)

注意：第三次握手丢失是由服务器重发连接请求的

TCP超时，断开连接RST是由谁发出的

当TCP连接中出现超时或者意外错误时，断开连接的RST（Reset）报文通常由**发起连接的一方**发送。这意味着在建立连接的过程中，如果发起连接的一方（通常是客户端）在某个阶段等待太久或者发生了错误，它可以发送一个RST报文来终止连接。

如何用 UDP 实现可靠传输?

字节一面：如何用 UDP 实现可靠传输？ - 知乎 (zhihu.com)

UDP为什么没有粘包

UDP基于报文，一整个直接发的

TCP在什么情况下会出现大量time_wait，哪个阶段出现

1.1 出现的原因

- **高并发短连接的服务器上会出现这样的情况，导致创建大量的tcp连接然后close，出现大量的连接出现time_wait的状态。**

1.2.大量time_wait的危害

- 在socket的TIME_WAIT状态结束之前，该socket所占用的本地端口号将一直无法释放
- 在高并发（每秒几万qps）并且采用短连接方式进行交互的系统中运行一段时间后，系统中就会存在大量的time_wait状态，如果time_wait状态把系统所有可用端口都占完了且尚未被系统回收时，就会出现无法向服务端创建新的socket连接的情况。此时系统几乎停转，任何链接都不能建立。
- 大量的time_wait状态也会系统一定的fd，内存和cpu资源，当然这个量一般比较小，并不是主要危害

1.3.大量time_wait解决方案

方式一：调整系统内核参数

```
net.ipv4.tcp_tw_reuse = 1 表示开启重用。允许将TIME-WAIT sockets重新用于新的TCP连接，默认为0，表示关闭；
net.ipv4.tcp_tw_recycle = 1 表示开启TCP连接中TIME-WAIT sockets的快速回收，默认为0，表示关闭。
net.ipv4.tcp_tw_recycle = 1表示开启TCP连接中TIME-WAIT sockets的快速回收，默认为0，表示关闭。
123
```

允许允许将TIME-WAIT sockets重新用于新的TCP连接，同时TIME-WAIT sockets的加快回收

方式二：改短连接为长连接

- 短连接和长连接工作方式的区别：

短连接：

- 连接->传输数据->关闭连接
- HTTP是无状态的，浏览器和服务器每进行一次HTTP操作，就建立一次连接，但任务结束就中断连接。
- 也可以这样说：短连接是指SOCKET连接后发送后接收完数据后马上断开连接。

长连接：

- 连接->传输数据->保持连接 -> 传输数据-> 。。。->关闭连接。
- 长连接指建立SOCKET连接后不管是否使用都保持连接，但安全性较差。

从区别上可以看出，长连接比短连接从根本上减少了关闭连接的次数，减少了TIME_WAIT状态的产生数量，在高并发的系统中，这种方式的改动非常有效果，可以明显减少系统TIME_WAIT的数量。

解决上述 time_wait 状态大量存在，导致新连接创建失败的问题，一般解决办法：

1.客户端，HTTP 请求的头部，connection 设置为 keep-alive，保持存活一段时间：现在的浏览器，一般都这么进行了

2.服务器端 允许 time_wait 状态的 socket 被重用 缩减 time_wait 时间，设置为 1 MSL（即，2 mins）

大量close_wait的出现

- close_wait是被动关闭连接是形成的，根据TCP状态机，服务器端收到客户端发送的FIN，TCP协议栈会自动发送ACK，链接进入close_wait状态。但**如果服务器端不执行socket的close()操作（即不向客户端发送FIN），状态就不能由close_wait迁移到last_ack，则系统中会存在很多close_wait状态的连接**

我觉得第一时间应该去判断是客户端的问题还是服务端的问题，有可能是客户端一直在向服务端发送FIN，也有可能是服务端一直没有发送自己的FIN。

可能的原因如下：

- 关闭socket不及时：例如I/O线程被意外阻塞，或者I/O线程执行的用户自定义Task比例过高，导致I/O操作处理不及时，链路不能被及时释放。

通常，CLOSE_WAIT 状态在服务器停留时间很短，如果你发现大量的 CLOSE_WAIT 状态，那么就意味着被动关闭的一方没有及时发出 FIN 包，一般有如下几种可能：

- 程序问题：如果代码层面忘记了 close 相应的 socket 连接，那么自然不会发出 FIN 包，从而导致 CLOSE_WAIT 累积；或者代码不严谨，出现死循环之类的问题，导致即便后面写了 close 也永远执行不到。
- 响应太慢或者超时设置过小：如果连接双方不和谐，一方不耐烦直接 timeout，另一方却还在忙于耗时逻辑，就会导致 close 被延后。响应太慢是首要问题，不过换个角度看，也可能是 timeout 设置过小。

在上层域名服务器逐层往下访问时，它怎么知道该去找下层的哪个域名服务器呢

查看网络状况

在Linux系统中，可以使用以下命令来查看网络状况：

1. ifconfig：显示当前的网络接口配置信息，包括IP地址、子网掩码、MAC地址等。
2. ping：用于测试与目标主机之间的连通性。可以通过向目标主机发送ICMP回显请求（ping请求）并等待回复来判断网络是否正常工作。

   示例：ping www.example.com
3. traceroute或traceroute6：用于跟踪数据包从本地主机到目标主机的路径。它显示经过的每个路由器的IP地址和延迟时间。

   示例：traceroute www.example.com
4. netstat：用于查看网络连接信息和网络统计数据，包括正在建立的连接、已经建立的连接以及端口的监听状态等。

   示例：netstat -ano

ping属于什么协议，在哪一层?

ICMP，网络层

输入 ping 某个ip，发生了什么

当你在命令行输入 "ping" 命令，后跟一个IP地址时，发生了以下过程：

1. 发送ICMP请求：计算机将发送一个ICMP（Internet Control Message Protocol）回显请求，也称为ping请求，到目标IP地址。这个请求是一个特殊的数据包，用于测试与目标主机的连通性。
2. 等待回复：目标主机接收到ICMP请求后，会解析该请求，并向源主机发送一个ICMP回显应答。该应答包含与原始请求相同的数据，以确认目标主机的连通性。

3. 计算往返时间（Round-Trip Time, RTT）：源主机接收到目标主机的ICMP回显应答后，会计算往返时间（RTT），即从源主机发送请求到接收到应答的时间延迟。
4. 显示结果：源主机接收到ICMP回显应答后，会显示结果给用户。通常显示的信息包括目标主机的IP地址、每个数据包的往返时间、数据包丢失率等。

通过使用ping命令并提供目标主机的IP地址，你可以测试你的计算机与目标主机之间的网络连通性和响应时间。这对于诊断网络问题、检查主机是否可达以及评估网络性能非常有用。

建立连接后，客户端拔掉网线后会怎么样?

保活机制；客户端既不能发也不能收，如果有任务正在进行，则任务会中断；recv一直阻塞，线程无法退出

如果在服务器和客户端建立连接后，客户端拔掉了网线（即网络连接被中断），将会发生以下情况：

1. 网络连接中断：客户端拔掉网线后，与服务器之间的物理网络连接将中断。这意味着服务器无法通过该网络连接与客户端进行通信。
2. 客户端无法发送请求：由于客户端的网络连接已经中断，它将无法向服务器发送任何新的请求。无法建立新的连接或发送数据。
3. 服务器可能继续等待：如果服务器正在等待从客户端接收数据或等待客户端发送进一步的请求，它可能会继续等待。服务器无法感知到客户端连接的中断，除非服务器尝试向客户端发送数据。
4. 连接超时或中止：服务器可能会在一段时间后发现连接超时，因为无法接收到来自客户端的任何响应。根据具体的配置和应用程序，服务器可能会主动关闭连接或释放相关资源。

总的来说，客户端拔掉网线后，服务器将无法与该客户端进行进一步的通信。服务器可能会继续等待一段时间，但最终可能会超时或中止该连接。一旦网络连接恢复，客户端需要重新建立连接才能与服务器进行通信。

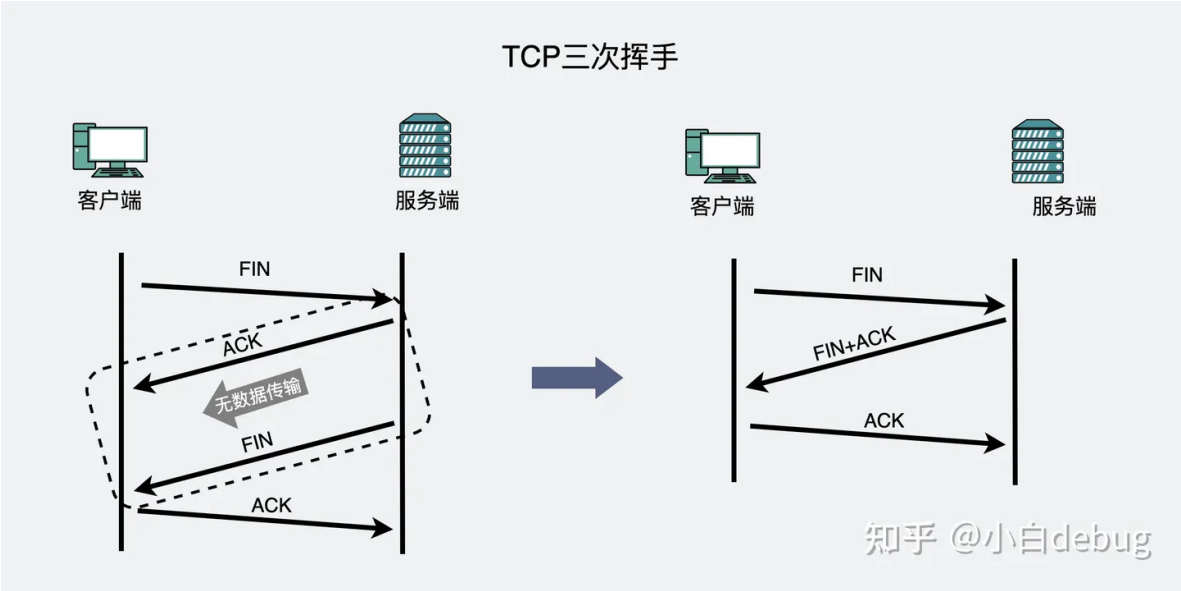
接上问，检测的这个时间有点长，有什么办法优化吗

应用层心跳机制

tcp四次挥手第二次和第三次不能合并吗

我们知道，TCP四次挥手里，第二次和第三次挥手之间，是有可能有数据传输的。第三次挥手的目的是为了告诉主动方，被动方没有数据要发了"。

所以，在第一次挥手之后，如果被动方没有数据要发给主动方。第二和第三次挥手是**有可能**合并传输的。这样就出现了三次挥手。



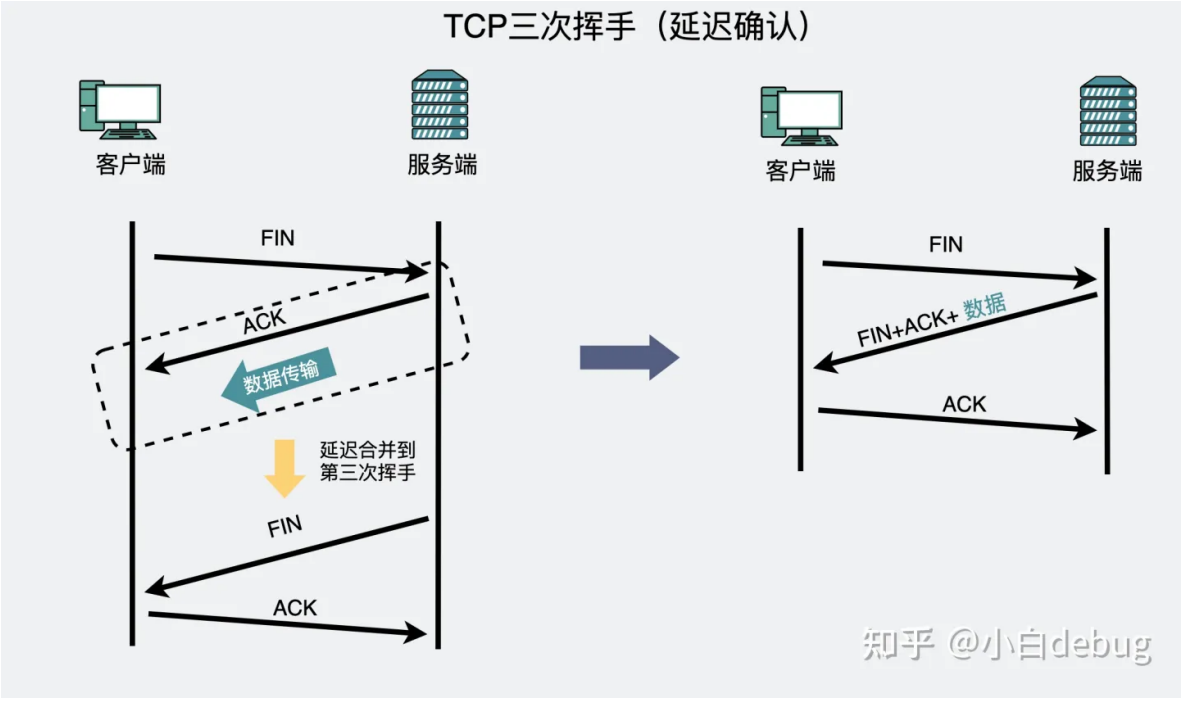
如果有数据要发，就不能是三次挥手了吗

上面提到的是**没有数据要发**的情况，如果第二、第三次挥手之间**有数据**要发，就不可能变成三次挥手了吗?

并不是。TCP中还有个特性叫**延迟确认。**可以简单理解为：**接收方收到数据以后不需要立刻马上回复ACK确认包。**

在此基础上，**不是每一次发送数据包都能对应收到一个** `ACK` **确认包，因为接收方可以合并确认。**

而这个合并确认，放在四次挥手里，可以把第二次挥手、第三次挥手，以及他们之间的数据传输都合并在一起发送。因此也就出现了三次挥手。



dns服务器用的是什么协议

需要注意的是，DNS 使用了 UDP 协议来获取域名对应的 IP 地址，这个没错，但有些片面，准确的来说，DNS **查询**在刚设计时主要使用 **UDP** 协议进行通信，而 TCP 协议也是在 DNS 的演进和发展中被加入到规范的：

1. DNS 在设计之初就在区域 **传输中引入了 TCP 协议**， **在查询中使用 UDP 协议**，它同时占用了 UDP 和 TCP 的 **53** 端口
2. 当 DNS 超过了 **512** 字节的限制，我们第一次在 DNS 协议中明确了 **『当 DNS 查询被截断时，应该使用 TCP 协议进行重试』** 这一规范；
3. 随后引入的 EDNS 机制允许我们使用 UDP 最多传输 4096 字节的数据，但是由于 MTU 的限制导致的数据分片以及丢失，使得这一特性不够可靠；
4. 在最近的几年，我们重新规定了 DNS 应该同时支持 UDP 和 TCP 协议，TCP 协议也不再只是重试时的选择；

如果解析http请求的时候，用户一次性没传完数据（如果头部都没传完，请求报文长度字段都没传完，怎么办）

如果在解析HTTP请求时，用户一次性没有完全传输完数据（包括请求头和请求报文长度字段），服务器会根据当前接收到的数据来处理。以下是可能的处理方式：

1. 等待接收完整的请求：服务器可以暂时等待，继续接收数据直到接收到完整的请求头和请求报文长度字段为止，然后再进行解析和处理。**服务器可以使用超时机制，在一定时间内等待客户端继续发送数据，若超过设定的时间还未接收到完整的请求，则可以中断连接或返回错误信息。**
2. 部分处理请求：如果服务器已经接收到部分请求数据，但还未接收到完整的请求头或请求报文长度字段，可以根据已有的数据进行初步处理。例如，可以解析已接收到的请求头部分并进行一些基本的验证，如判断请求的方法、路径等是否正确。但是，对于需要请求报文长度字段才能确定的操作（如获取请求体的长度），服务器将无法完成。
3. 错误处理：如果服务器无法接收到完整的请求头和请求报文长度字段，或者在一定时间内没有接收到期望的数据，**服务器可以返回错误响应，如400 Bad Request等。**

当服务器收到部分数据时，要格外小心处理，以避免在处理不完整的请求时引发安全问题或错误解析导致的异常行为。为了确保数据的完整性和正确性，通常建议客户端按照HTTP协议规范发送完整的请求。

http1.0，2.0版本的区别

一、HTTP1.0

`HTTP`协议的第二个版本，第一个在通讯中指定版本号的HTTP协议版本

`HTTP 1.0` 浏览器与服务器只保持短暂的连接，每次请求都需要与服务器建立一个`TCP`连接

服务器完成请求处理后立即断开`TCP`连接，服务器不跟踪每个客户也不记录过去的请求

简单来讲，**每次与服务器交互，都需要新开一个连接**

例如，解析`html`文件，当发现文件中存在资源文件的时候，这时候又创建单独的链接

最终导致，一个`html`文件的访问包含了多次的请求和响应，每次请求都需要创建连接、关系连接

这种形式明显造成了性能上的缺陷

如果需要建立长连接，需要设置一个非标准的Connection字段 `Connection: keep-alive`

二、HTTP1.1

在`HTTP1.1`中，默认支持**长连接**（`Connection: keep-alive`），即在一个TCP连接上可以传送多个`HTTP`请求和响应，减少了建立和关闭连接的消耗和延迟

建立一次连接，多次请求均由这个连接完成

这样，在加载`html`文件的时候，文件中多个请求和响应就可以在一个连接中传输

同时，`HTTP 1.1`还允许客户端不用等待上一次请求结果返回，就可以发出下一次请求，但服务器端必须按照接收到客户端请求的先后顺序依次回送响应结果，以保证客户端能够区分出每次请求的响应内容，这样也显著地减少了整个下载过程所需要的时间

同时，`HTTP1.1`在`HTTP1.0`的基础上，增加更多的请求头和响应头来完善的功能，如下：

- 引入了更多的缓存控制策略，如If-Unmodified-Since, If-Match, If-None-Match等缓存头来控制缓存策略
- 引入range，允许值请求资源某个部分
- 引入host，实现了在一台WEB服务器上可以在同一个IP地址和端口号上使用不同的主机名来创建多个虚拟WEB站点

并且还添加了其他的请求方法：`put`、`delete`、`options`...

三、HTTP2.0

而`HTTP2.0`在相比之前版本，性能上有很大的提升，如添加了一个特性：

- 多路复用
- 二进制分帧
- 首部压缩
- 服务器推送

多路复用

`HTTP/2`复用`TCP`连接，在一个连接里，客户端和浏览器都可以**同时**发送多个请求或回应，而且不用按照顺序一一对应，这样就避免了"队头堵塞"

上图中，可以看到第四步中`css`、`js`资源是同时发送到服务端

二进制分帧

帧是`HTTP2`通信中最小单位信息

`HTTP/2`采用二进制格式传输数据，而非`HTTP 1.x`的文本格式，解析起来更高效

将请求和响应数据分割为更小的帧，并且它们采用二进制编码

`HTTP2`中，同域名下所有通信都在单个连接上完成，该连接可以承载任意数量的双向数据流

每个数据流都以消息的形式发送，而消息又由一个或多个帧组成。多个帧之间可以乱序发送，根据帧首部的流标识可以重新组装，这也是多路复用同时发送数据的实现条件

首部压缩

`HTTP/2`在客户端和服务器端使用"首部表"来跟踪和存储之前发送的键值对，对于相同的数据，不再通过每次请求和响应发送

首部表在`HTTP/2`的连接存续期内始终存在，由客户端和服务器共同渐进地更新

例如：下图中的两个请求， 请求一发送了所有的头部字段，第二个请求则只需要发送差异数据，这样可以减少冗余数据，降低开销

服务器推送

`HTTP2` 引入服务器推送，允许服务端推送资源给客户端

服务器会顺便把一些客户端需要的资源一起推送到客户端，如在响应一个页面请求中，就可以随同页面的其它资源

免得客户端再次创建连接发送请求到服务器端获取

这种方式非常合适加载静态资源

四、总结

HTTP1.0：

- 浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个TCP连接

HTTP1.1：

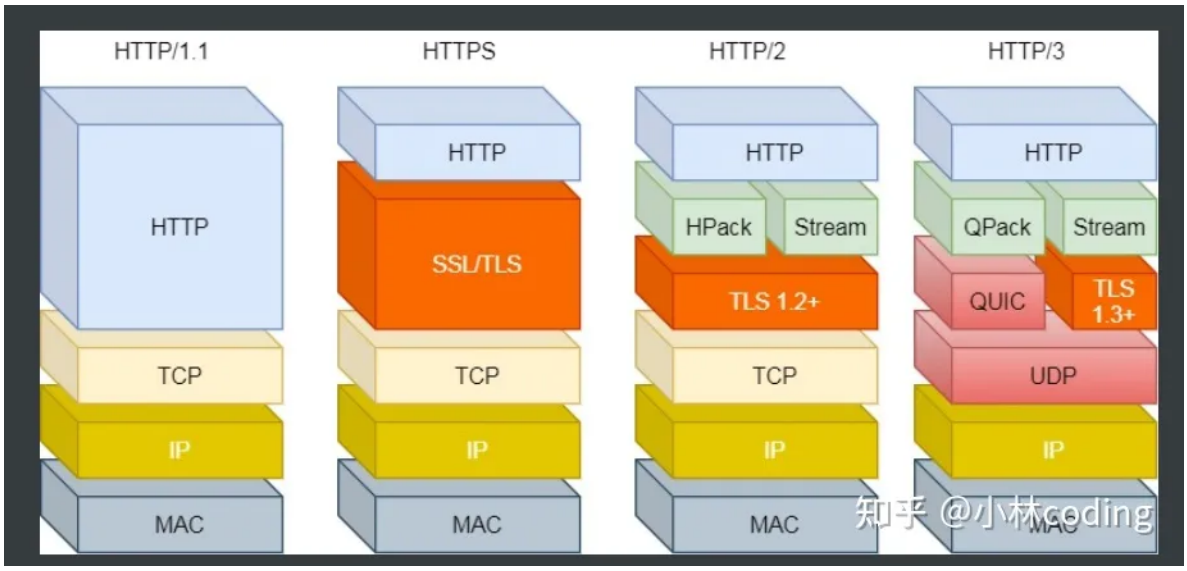
- 引入了持久连接，即TCP连接默认不关闭，可以被多个请求复用
- 在同一个TCP连接里面，客户端可以同时发送多个请求
- 虽然允许复用TCP连接，但是同一个TCP连接里面，所有的数据通信是按次序进行的，服务器只有处理完一个请求，才会接着处理下一个请求。如果前面的处理特别慢，后面就会有许多请求排队等着
- 新增了一些请求方法
- 新增了一些请求头和响应头

HTTP2.0：

- 采用二进制格式而非文本格式
- 完全多路复用，而非有序并阻塞的、只需一个连接即可实现并行
- 使用报头压缩，降低开销
- 服务器推送

http一定是tcp吗

HTTP 版本，HTTP/1.1 和 HTTP/2 都是基于 TCP传输协议的，而 HTTP/3 是基于 UDP 传输协议的。



服务器多个进程可以都绑定80这个端口监听吗，原理是什么，操作系统怎么转发的

对于Web而已，80端口和443端口是十分重要的，原则上需要输入http://domain.com:80才可以浏览网页的，但由于默认端口是80，所以':80'可以忽略。同理对于https的443端口也一样。

随着服务器性能的提升和业务的需求，一台服务器上往往会同时有多个服务，这些服务都希望监听80端口，比如有vue.msg.com和react.msg.com。这时候我们可以使用**nginx的代理转发功能**帮我们实现共用80端口的需求。

首先我们先在两个空闲的端口上分别部署项目（非80，假设是8080和8081），*nginx.conf*配置如下：

```
$ vim /ect/nginx/nginx.conf
```

```
// nginx.conf
# vue项目配置
server {
    listen       8080;
    root         /web/vue-base-demo/dist/;
    index        index.html;
    location / {
        try_files $uri $uri/ /index.html; # 路由模式history的修改
    }
}

# react项目配置
server {
    listen       8081;
    root         /web/react-base-demo/build;
    index        index.html;
    location / {}
}
```

上面就是常规的配置，紧接着如果已经做好域名解析，希望vue.msg.com打开vue项目，react.msg.com打开react项目。我们需要再做两个代理，如下:

```
// nginx.conf
# nginx 80端口配置 （监听vue二级域名）
server {
    listen  80;
    server_name     vue.msg.com;
    location / {
        proxy_pass      http://localhost:8080; # 转发
    }
}

# nginx 80端口配置 （监听react二级域名）
server {
    listen  80;
    server_name     react.msg.com;
    location / {
        proxy_pass      http://localhost:8081; # 转发
    }
}
```

nginx如果检测到vue.msg.com的请求，将原样转发请求到本机的8080端口，如果检测到的是react.msg.com请求，也会将请求转发到8081端口。

这样nginx对外就有四个服务，我们只需要公布80端口的就可以了，这样就实现了多个服务共用80端口。

https可以绑定80端口吗，绑定了还能加密吗，为什么

如果默认的443端口被占用了，那么https端口也是可以修改的。只不过使用了自定义的端口，**在访问网站时需要在网址后面加上端口号**，这样就比较麻烦。当然，自定义的端口是要存在于Web服务器的，如果不存在，就会出现超时报错。

在一般情况下，HTTPS协议默认使用443端口进行通信，而不是80端口。这是因为HTTPS协议需要加密传输数据，而加密的过程需要使用SSL/TLS协议进行握手、证书验证和加密操作。这些额外的安全机制需要占用一定的资源和处理时间，因此在设计时将HTTPS默认指定为443端口，以避免与常规的HTTP流量混淆。

尽管如此，理论上是可以将HTTPS协议绑定到80端口的。事实上，某些特殊情况下可能会出现这样的配置，但这并不是推荐的做法。如果将HTTPS协议绑定到80端口，会导致与标准HTTP流量发生冲突，并可能引起一系列的问题。

至于是否能在绑定80端口的同时保持数据的加密传输，答案是否定的。必须注意的是，加密功能是由SSL/TLS协议提供的，而SSL/TLS协议需要建立在TCP连接之上。只有通过默认的端口443或其他指定的端口才能正常执行SSL/TLS握手，确保客户端和服务器之间的安全通信。

如果将HTTPS协议绑定到80端口，那么传输层的协议就是普通的HTTP（非加密状态），而不是SSL/TLS加密的HTTPS。这意味着无法对数据进行加密保护，所有的传输内容都是明文传输，在网络中容易被恶意攻击者窃听和篡改。

总结起来，虽然可以将HTTPS协议绑定到80端口，但这并不是推荐的做法。HTTPS通常应该使用443端口，以确保数据加密传输和安全性。

绑定的端口可以建立多个连接吗

多个tcp也能绑定一个端口，开SO_REUSEPORT就行，这种就相当于有多个acceptor监听一个端口

该参数允许多个socket绑定到同一本地地址，即使socket是处于listen状态的。

当多个listen状态的socket绑定到同一地址时，各个socket的accept操作都能接受到新的tcp连接。

数据库

mysql的ACLS（事务）

查询学生姓名和挂科门数

要查询学生姓名和挂科门数，需要使用MySQL的SELECT语句结合聚合函数。假设学生信息存储在名为"students"的表中，包含字段"姓名"（name）和"科目成绩"（subject_score）。

以下是查询学生姓名和挂科门数的MySQL命令：

```
SELECT name, COUNT(*) AS 挂科门数
FROM students
WHERE subject_score < 60
GROUP BY name;
```

这条命令会筛选出科目成绩低于60分的记录，并按姓名进行分组，然后统计每个分组中的记录数量作为挂科门数。查询结果将会显示学生的姓名和对应的挂科门数。

请注意，上述命令是一个示例，具体的表名、字段名及条件应根据实际情况进行调整。

找到一张表其中一个字段中所有重复的

要找到MySQL表中某个字段中的所有重复值，可以使用SELECT语句结合GROUP BY和HAVING子句来实现。

假设你需要找到名为"table_name"的表中字段"field_name"中的所有重复值，以下是相应的MySQL命令：

```
SELECT field_name, COUNT(*) AS 重复次数
FROM table_name
GROUP BY field_name
HAVING COUNT(*) > 1;
```

这条命令会对表中的所有记录按照字段"field_name"进行分组，并统计每个分组中的记录数量。然后通过HAVING子句筛选出重复次数大于1的分组，即找到所有重复值。查询结果将会显示重复的字段值以及对应的重复次数。

请注意，上述命令中的"table_name"需要替换为实际的表名，"field_name"需要替换为实际的字段名。

主键索引和唯一索引区别

- 主键索引：主键索引是用于唯一标识表中每一行数据的索引。主键索引要求索引列的值唯一且不为空，一个表只能有一个主键索引。
- 唯一索引：唯一索引也要求索引列的值唯一，但可以包含空值（NULL）。一个表可以有多个唯一索引。

总体而言，主键索引是一种特殊的唯一索引，用于唯一标识表中每一行数据并确定数据的物理存储顺序。唯一索引则更通用，用于确保索引列的值唯一，可以包含空值，并且一个表可以有多个唯一索引。选择使用主键索引还是唯一索引，应根据具体的业务需求和数据库设计来决定。

一个联合索引（a，b，c），查询（a，c），能用到索引吗

不能，不符合最左前缀匹配原则；只有a列用到了索引

项目

多线程编程要注意的问题

在进行多线程编程时，C++开发者需要注意以下问题：

1. 线程安全性：确保多个线程可以安全地访问共享的数据结构或资源，避免数据竞争和产生未定义行为。可以使用互斥锁、条件变量等同步机制来保证线程安全性。
2. 死锁（Deadlock）：当多个线程相互等待资源并无法释放时，会导致死锁。要避免死锁，需要合理安排资源申请和释放的顺序，以及使用超时机制或避免循环资源依赖关系。
3. 线程同步和通信：多线程之间通常需要进行同步和通信，以确保正确的执行顺序和数据传递。可以使用互斥锁、条件变量、信号量等机制来实现线程间的同步和通信。
4. 内存管理：多线程环境下，需要合理管理内存分配和释放，避免内存泄漏和悬空指针。可以使用智能指针、锁定内存分配器等技术来辅助内存管理。
5. 性能优化：多线程编程可能涉及到并发和并行执行，可通过合理的任务划分、线程池、使用锁的粒度控制等手段来优化程序性能，并减少线程之间的竞争。

6. 异常处理：在多线程环境下，异常处理需要格外注意。确保捕获和处理异常，以避免线程异常终止导致整个程序异常终止。
7. 资源管理：使用多线程时，要合理管理线程的生命周期、资源的分配和释放。确保适当地创建和销毁线程，并在不需要时及时释放资源。
8. 调试和测试：多线程程序的调试和测试比单线程要困难得多。使用合适的工具和技术，如断点调试、日志记录、线程安全检查工具等，来帮助定位和解决多线程问题。

以上是多线程编程中需要特别注意的问题，为了确保多线程程序的正确性和稳定性，建议在编写代码时注重线程安全性和合理的并发控制，并进行充分的测试和调试。

为什么要用多线程，多进程可以吗

在 C++ 中实现 Web 服务器时，可以使用多线程或多进程的方式来处理并发请求。这两种方法都有其优势和适用场景。

1. 多线程：
    - 优势：多线程适合于处理 **I/O 密集型**任务，例如接收请求、读写文件、访问数据库等。使用多线程**可以更好地利用 CPU 时间片，提高并发处理能力。**线程之间共享内存空间，可以方便地共享数据，减少数据传输和通信开销。
    - 注意事项：多线程程序需要正确处理线程同步和互斥，以避免竞态条件（Race Condition）和其他线程安全问题。还需要注意线程间的资源管理和内存管理，避免内存泄漏或悬挂指针等问题。
2. 多进程：
    - 优势：多进程适合于处理 **CPU 密集型**任务，例如密集计算、图像处理、加密解密等。**每个进程都有独立的内存空间，可以保证各个进程之间的隔离性和稳定性。**多进程模型更容易实现并发的水平扩展，因为进程可以在不同的计算机上运行。
    - 注意事项：多进程需要更多的系统资源，如内存和 CPU 时间。进程间通信较为复杂，可能需要使用进程间通信机制（如管道、共享内存、消息队列等）来实现进程间的数据传输和通信。

综合考虑，**对于大多数 Web 服务器场景，使用多线程模型更为常见和合适。多线程能够更好地利用 CPU 和内存资源，并发处理请求。另外，C++ 在多线程编程方面有较好的支持，例如标准库中的线程、互斥锁和条件变量等工具。**

然而，对于某些特定的场景和需求，**例如密集计算或需要保持隔离性的情况，多进程模型可能更为合适。**

需要根据具体的项目需求、硬件资源和性能要求来选择多线程或多进程模型。同时，编写并发程序需要特别注意线程/进程安全性、同步机制和资源管理，以确保代码的正确性和稳定性。

多进程相比多线程的优点

不同进程完全隔离，如果一个销毁，另一个不受影响

多进程fork后不同进程会共享哪些资源

文件描述符、内存地址空间（堆、栈、数据段）、执行的代码

需要注意的是，尽管子进程与父进程会共享某些资源，**但它们拥有各自独立的进程标识符（PID）以及不同的运行环境。**所以在编写并发程序时，需要**注意对共享资源的访问控制和同步机制**，以避免竞态条件和其他并发问题的发生。

这个服务器功能上与nginx服务器有什么不同之处吗

如果有两个服务器，一个服务器坏了，另一个服务器怎么判断并接手坏的服务器的用户数据

共用一个堆；

当一个服务器发生故障无法正常工作时，另一个服务器可以通过以下几种方式来判断并接手坏的服务器的用户数据：

1. **心跳检测**：
    - 另一个服务器可以定期向坏的服务器发送心跳检测请求，检查坏服务器是否响应。如果坏服务器没有响应，则可以判断其为故障状态。
    - 一旦检测到坏的服务器，另一个服务器可以立即接管该服务器的用户数据。
2. 负载均衡器（Load Balancer）：
    - 如果使用了负载均衡器来分发用户请求到多个服务器，当坏的服务器发生故障时，负载均衡器会自动将用户请求重新路由到其他正常工作的服务器上。
    - 这样，另一个服务器只需要监控负载均衡器的状态，并且确保自己正常工作即可接手坏的服务器的用户数据。
3. 数据同步和复制：
    - 如果两个服务器之间有进行数据同步和复制，坏的服务器的用户数据已经在另一个服务器上有备份。
    - 当坏的服务器发生故障时，另一个服务器可以直接使用已备份的用户数据来接手处理用户请求。

4.**keep-alive**

需要注意的是，在以上方法中，要确保系统能够及时发现坏的服务器，并采取相应的措施来保证数据的完整性和可用性。此外，还可以采用其他高可用性架构和技术，如**主从复制**、冗余设备等来提高系统的容错能力和可靠性。

epoll 底层为什么用红黑树不用hash

在 Linux 中，epoll 是一种高性能的事件驱动 I/O 模型，用于处理大量并发连接的情况。在 epoll 的内部实现中，使用红黑树来管理待处理的事件。

红黑树是一种自平衡的二叉查找树，具有较好的查找效率，时间复杂度为 O(log n)。它在内核中被用作 epoll 的底层数据结构，主要基于以下几点原因：

1. **查找效率高**：红黑树通过自平衡策略确保树的高度始终保持在较小的范围内，从而保证了在大量节点时的快速查找效率。
2. **有序性**：红黑树具有有序性，**可以按照事件发生的顺序进行排序和处理。**这对于事件驱动模型非常重要，可以确保事件被按照正确的顺序处理。
3. 插入和删除操作效率较高：红黑树的插入和删除操作都是比较高效的，特别是对于大量节点的情况，它们的时间复杂度为 O(log n)。

相比之下，使用哈希表作为底层数据结构可能会存在以下问题：

1. 哈希表需要**冲突**解决机制：在哈希表中，多个元素可能会映射到同一个哈希桶，这就需要解决冲突的问题。而 epoll 中的事件管理是根据文件描述符（File Descriptor）来进行查找和操作的，使用哈希表会引入额外的冲突解决机制，增加了复杂性。
2. **无序性**：哈希表是无序的，无法直接按照事件发生的顺序处理。**在事件驱动模型中，有序性是非常重要的，可以保证事件的正确处理顺序。**

总的来说，红黑树作为底层数据结构能够提供较好的平衡性、有序性和高效的插入、删除、查找操作，在 epoll 这样需要高并发处理的场景下更为适合。

线程池里面任务队列是无限的还是有限的，如果客户请求的任务太多了数组爆了怎么办，就把请求给丢了吗

线程池的任务队列**可以是有限的，也可以是无限的**，具体取决于实现的方式和设计需求。

如果线程池的任务队列是有限的，并且达到了队列的最大容量，那么当客户请求的任务过多导致队列已满时，可以根据具体的策略来处理这种情况：

1. **阻塞等待**：在队列已满的情况下，新到来的请求可以选择阻塞等待，直到有空闲位置将任务插入队列中。这样**可以保证所有的请求最终都能被处理**，但可能会导致一些延迟。
2. **丢弃任务**：另一种策略是丢弃当前无法插入队列的任务。这意味着一部分请求会被直接丢弃而不被处理。通常，**可以选择丢弃最早的请求**或者丢弃最新的请求。
3. **扩展任务队列**：当任务队列满时，还可以考虑动态扩展任务队列的容量，以容纳更多的请求。这需要根据实际情况评估系统负载和资源消耗。

java线程池ThreadPoolExecutor类详解（一）——常见的任务队列*java线程池任务队列*众星揽月的博客-CSDN博客

消息队列

线程池中主线程什么时候退出

在线程池中，主线程指的是创建和管理线程池的线程，也可以称为调用线程。主线程在创建完线程池后，会继续执行其余的任务，不会退出。

主线程会一直存在，直到显式地关闭线程池或应用程序退出。当主线程执行完所有任务后，如果不关闭线程池，它会阻塞在那里等待新的任务到来。

通常情况下，在使用线程池的应用程序中，在所有任务执行完毕后，我们需要手动调用线程池的关闭方法（如`ThreadPoolExecutor.shutdown()`）来关闭线程池，这时主线程才会退出。

线程池的关闭方法将会中断空闲线程并停止接受新任务，然后等待线程池中的所有任务完成。一旦所有任务完成，主线程和其他非守护线程退出，整个应用程序就会结束。

所以，线程池中的主线程在调用线程池的关闭方法后，等待所有任务完成后退出。

EAGAIN：非阻塞+循环读；判断读完了，其实不算一种错误

是的，EAGAIN 错误码通常在非阻塞模式下进行循环读取操作时出现。它表示当前没有可用的数据，需要稍后再试。

在非阻塞模式下，当没有数据可读取时，读取操作不会等待数据到达，而是立即返回 EAGAIN 错误码。这时，程序可以根据需要进行其他处理，然后再次尝试读取。

通常情况下，EAGAIN **不被视为一种错误**，而是一种正常的返回值。它表明当前没有可用的数据，但并不表示读取操作失败或出现错误。因此，在使用循环读取非阻塞 I/O 时，要对 EAGAIN 进行特殊处理，以便在有数据可读时再次尝试进行读取操作。

注意：需要根据具体的编程语言和操作系统进行相应的处理，以正确处理 EAGAIN 错误码。

发送一个HTTP请求，服务器生成响应，怎么判断这个响应是否完整然后浏览器进行渲染

要判断一个HTTP响应是否完整并进行渲染，需要依据HTTP协议规范和浏览器的工作机制。下面是一般的工作流程：

1. 解析HTTP头部：收到响应后，先解析HTTP头部，该头部包含了响应的元数据，如响应状态码、内容长度等信息。

2. 根据头部信息处理响应体：根据HTTP头部中的内容长度或者分块传输编码（chunked）等机制，确定响应体的长度。如果使用内容长度机制，可以通过读取总长度来判断是否完整。如果使用分块传输编码，则需要在读取每个分块时进行判断，直到读取到最后一个分块。
3. 合并响应数据：如果响应体被分割成多个分块，需要将它们按顺序合并起来。一旦所有分块都被接收并合并，就可以认为响应完整。
4. 渲染响应：一旦确定响应完整，浏览器会根据接收到的响应数据进行页面渲染和展示。

需要注意的是，浏览器一般会处理各种情况，如重定向、压缩等，这些操作可能会影响到完整响应的判断和渲染过程。

此外，在实际开发中，可以使用现成的HTTP库、框架或浏览器引擎来处理HTTP请求和响应，它们会自动处理这些细节，简化开发流程。

项目里如果一个HTTP请求不完整怎么判断

在项目中，如果一个HTTP请求不完整，可以通过以下几种方式来进行判断：

1. 检查响应头部：HTTP协议规定了响应头部需要包含`Content-Length`或`Transfer-Encoding`字段来指示响应的长度或分块传输编码。你可以通过检查这些字段是否存在，并根据其值来判断响应是否完整。
2. 检查数据流结束符：在HTTP协议中，数据流结束可以使用空行（`\r\n\r\n`）或者`Content-Length`字段指定的长度来判断。你可以检查接收到的数据流中是否存在这些结束符或者长度是否与`Content-Length`字段一致。
3. 设置超时时间：可以设置一个适当的超时时间，在规定时间内没有接收到完整的HTTP响应，则可以判断该请求不完整。
4. 使用**状态机**或解析器：可以使用状态机或HTTP解析器来处理HTTP协议。它们可以帮助你按照HTTP协议规范进行逐步解析，从而判断请求是否完整。

以上方法只是一些常见的判断方式，具体的实现取决于项目的需求和开发环境。在实际应用中，可以结合使用多种判断方式来提高判断的准确性和可靠性。

内存映射是有名还是匿名

内存映射可以是有名（named）的，也可以是匿名（anonymous）的。

有名内存映射是指**通过文件进行内存映射**，可以将文件中的内容映射到内存中的某个地址空间。这样，就可以通过内存的方式对文件进行读写操作，而不需要使用传统的文件读写接口。有名内存映射通常用于进程间共享数据，多个进程可以将同一个文件映射到它们各自的地址空间中，从而实现数据的共享和通信。

匿名内存映射是指**在没有特定文件的情况下，直接将一段物理内存映射到进程的虚拟地址空间中。**匿名内存映射通常用于为进程分配一块连续的内存区域，用于存储临时数据或动态分配的数据结构。

需要注意的是，有名和匿名内存映射并不是互斥的概念，它们可以同时存在于一个系统中，根据具体场景和需求选择合适的方式进行内存映射。

用epoll实现一个单线程的FTP文件传输器，可以吗？epoll不能实现文件读写

使用 epoll 单独实现一个单线程的 FTP 文件传输器是有可能的，但是需要注意的是，**epoll 本身并不负责文件的读写操作，它主要用于实现事件驱动的网络编程。**

在 FTP 文件传输中，涉及到的操作包括**与客户端建立连接、接收和解析客户端发送的命令、处理文件的读写等。对于文件的读写操作，可以使用文件描述符（file descriptor）相关的系统调用（如`open()`、`read()`、`write()`、`close()`等）来实现。**

epoll 在这个场景下的作用是监听套接字上的事件，当有新连接到来或者已有连接有数据可读或可写时，触发相应的回调函数进行处理。你可以将 FTP 的控制连接作为一个套接字注册到 epoll 中，然后根据不同的事件进行相应的操作，例如接受新连接、接收命令、处理文件传输等。

需要注意的是，FTP 的文件传输通常使用的是阻塞 I/O 模型，而 epoll 是基于非阻塞 I/O 模型的事件驱动机制。在单线程实现中，你可能需要将阻塞 I/O 转换为非阻塞 I/O，并结合 epoll 进行事件的处理。

总之，单线程的 FTP 文件传输器可以使用 epoll 来实现事件的监听和分发，但仍然需要利用文件描述符相关的系统调用来实现文件的读写操作。

怎么判断网络拥塞

判断网络拥塞可以采取以下几种方式：

1. **丢包率**（Packet Loss）：当网络拥塞时，网络中的路由器或链路可能无法及时处理所有的数据包，导致部分数据包丢失。通过监测发送和接收数据包的丢失率，可以判断网络是否存在拥塞。高丢包率通常是拥塞的一个明显指标。
2. 延迟（Latency）：网络拥塞也会导致数据传输的延迟增加。通过测量数据传输的往返时间（Round-Trip Time, RTT）或者其他延迟指标（如平均延迟、最大延迟等），可以获取网络的延迟信息。如果延迟明显增加，可能是网络出现了拥塞。
3. **带宽利用率**（Bandwidth Utilization）：网络拥塞时，网络链路的带宽可能被过多的流量占用，导致带宽利用率饱和。通过监测网络链路的带宽利用率，可以得到一定程度上的拥塞信息。如果带宽利用率接近或达到了链路容量的上限，可能存在网络拥塞。
4. 队列长度（Queue Length）：拥塞时，网络设备（如路由器）中的队列可能会积累大量的待处理数据包。通过监测队列的长度，可以得到网络设备是否过载的信息。如果队列长度高于设备的缓冲区容量，可能发生了网络拥塞。

需要注意的是，单独依靠以上几种指标并不能完全确定网络是否拥塞，通常需要结合多个指标和综合分析。另外，还可以使用一些网络性能监控工具或者流量分析工具来帮助判断网络的拥塞情况。

超时重传什么时候关闭

超时重传（Timeout Retransmission）的关闭时间点通常取决于具体的实现和网络条件。下面是一些常见的情况：

1. **成功传输**：当数据成功传输后，即接收方正确接收到发送方发送的数据，并且发送方也收到了确认信息（ACK），可以关闭超时重传机制。因为没有出现丢包或错误，所以不需要再进行重传。
2. **连接关闭**：当发送方和接收方确定要关闭连接时，超时重传机制也可以关闭。此时已经不需要继续进行数据传输，因此也不需要再进行超时重传。
3. 超时时间过长：在某些情况下，如果超时时间设置得过长，可能会导致传输过程中的延迟过高，影响整体性能。在这种情况下，可以考虑适当缩短超时时间，从而提高传输效率。

需要注意的是，关闭超时重传机制并不意味着不再处理丢包或错误。即使关闭了超时重传，仍然需要采取其他机制来应对网络中的丢包、错误或拥塞情况。例如，可以使用前向纠错技术（Forward Error Correction, FEC）或者使用其它可靠性协议（如TCP）来处理丢包和错误。

总之，关闭超时重传机制的时间点应该是根据具体的应用需求和网络条件来确定，需要综合考虑传输的可靠性、性能和延迟等因素。

零拷贝

笔记P75

[原来 8 张图，就可以搞懂「零拷贝」了 - 小林coding - 博客园 (cnblogs.com)]

浏览器怎么知道服务器返回的是图片还是js或者css

浏览器通过**HTTP响应头部的Content-Type字段**来确定服务器返回的数据类型是图片、JavaScript文件还是CSS文件等。

在你的Web服务器代码中，当服务器处理请求并准备发送响应时，你需要设置正确的Content-Type字段。

epoll怎么知道文件描述符上发生了事件

感觉是想问ET模式和LT模式，当时没反应过来面试官想问什么

内存为什么设计栈和堆，不能都在堆上分配吗

内存既可以在栈上分配，也可以在堆上分配。这是因为栈和堆都有各自的特点和用途。

栈（Stack）和堆（Heap）是在程序运行时动态分配内存的两种主要方式：

1. 栈：栈内存由编译器自动管理，用于存储局部变量、函数参数和函数调用信息等。栈内存的分配和释放是自动的，遵循先进后出（FILO）的原则。栈的内存分配速度快，在函数调用结束时会自动释放，不需要手动管理内存。但栈的容量有限，通常比较小。
2. 堆：堆内存由程序员手动管理，用于存储动态分配的对象和数据结构。堆的内存分配和释放需要手动进行，如果不及时释放可能导致内存泄漏。堆的容量相对较大，可以动态增长和缩减。堆内存的分配和释放开销较大，可能导致碎片化问题。

为什么不能将所有内存都分配在堆上呢?

1. 分配效率：栈上的内存分配速度快，仅仅是栈指针的移动，而堆上的内存分配需要维护数据结构、查找合适的内存块等，因此分配效率要比栈低。
2. 内存管理：栈内存由编译器自动管理，无需手动释放，简化了内存管理。而堆内存需要手动分配和释放，容易出现内存泄漏或悬空指针等问题，增加了程序员的负担。
3. 容量限制：栈的容量通常比较小，受限于操作系统和编译器的限制。将所有内存都分配在堆上可以解决容量限制的问题。

总而言之，栈和堆都有自己的优势和适用场景。在牛客网C++项目的Web服务器中，可以根据变量的生命周期和容量需求来选择是使用栈上内存还是堆上内存。局部变量和较小的对象可以分配在栈上，而大型对象和动态分配的数据结构可以放在堆上。合理利用栈和堆，可以提高内存管理的效率和灵活性。

浏览器访问服务器会经过哪些协议

当浏览器访问服务器时，会经过以下一系列协议：

1. DNS（Domain Name System）：浏览器首先会将服务器的域名解析为服务器的 IP 地址。这个过程使用 DNS 协议完成，通过查询 DNS 服务器来获取对应的 IP 地址。
2. TCP（Transmission Control Protocol）：一旦浏览器获得了服务器的 IP 地址，它与服务器之间建立 TCP 连接。TCP 是一种可靠的传输协议，确保数据的完整性和有序性。它提供了双向通信的能力。
3. HTTP（Hypertext Transfer Protocol）：在建立了 TCP 连接后，浏览器和服务器之间开始使用 HTTP 协议进行通信。HTTP 是一种应用层协议，用于在 Web 上传输超文本资源。它定义了客户端和服务器之间的请求和响应格式，包括请求方法、状态码、头部信息等。
4. SSL/TLS（Secure Socket Layer/Transport Layer Security）：如果使用 HTTPS 协议进行安全通信，浏览器和服务器之间会进行 SSL/TLS 握手。SSL/TLS 协议提供了数据加密和身份验证的功能，保护数据的安全性。
5. HTTP 请求和响应：浏览器发送 HTTP 请求到服务器，请求中包含了所需的资源信息，例如 URL、请求方法（GET、POST 等）、请求头部等。服务器接收到请求后，根据请求的处理逻辑生成 HTTP 响应，响应中包含了状态码、响应头部和响应体等信息。
6. 数据传输：一旦服务器生成了 HTTP 响应，它会通过 TCP 连接将响应发送回浏览器。TCP 协议确保数据的可靠传输，将数据分割为小的数据包，并在接收端重新组装。

7. 渲染和展示：浏览器接收到服务器的响应后，根据返回的数据进行页面渲染和展示，最终呈现给用户。

以上是浏览器访问服务器时常见的协议流程，其中 DNS、TCP、HTTP 和 SSL/TLS 是其中关键的协议。这些协议协同工作，使得浏览器能够与服务器进行通信，获取所需的资源并在页面上展示。

怎么做到多服务器的负载均衡

要实现多服务器的负载均衡，可以采用以下几种常见的方法：

1. 硬件负载均衡器：使用专门的硬件设备（如负载均衡器）来分发请求到多个服务器。负载均衡器位于客户端和服务器之间，根据预设的策略将请求转发到不同的服务器上，以平衡服务器的负载。
2. 软件负载均衡器：使用软件实现负载均衡的功能。常见的软件负载均衡器有 Nginx、HAProxy 等。这些负载均衡器通过配置策略（如轮询、最少连接、IP 哈希等）将请求分发到后端的多个服务器上。
3. DNS 负载均衡：通过在 DNS 服务器中配置多个服务器 IP 地址，使得每个请求根据一定策略获得不同的 IP 地址，从而将请求分发到多个服务器上。常见的策略有轮询、加权轮询、随机等。
4. 反向代理：将负载均衡功能集成到反向代理服务器中。反向代理服务器接收客户端的请求，然后根据预设的负载均衡策略将请求转发给后端的多个服务器。常见的反向代理服务器有 Nginx、Apache 等。
5. 会话保持：为了确保用户在多个请求中的会话状态不丢失，需要采用一定的会话保持机制。可以使用 cookie、URL 重写等方式将用户会话与特定的服务器绑定，使得用户在整个会话过程中访问同一个服务器。
6. 动态扩展：当负载增加时，可以动态地向服务器集群中添加新的服务器来分担负载。根据实际需要，可以自动或手动地启动新的服务器，并将其加入到负载均衡器的服务池中。

以上是常见的多服务器负载均衡方法，可以根据具体需求选择合适的方式进行实施。负载均衡可以提高系统的可用性、性能和扩展性，确保多个服务器能够平衡处理请求，提供稳定可靠的服务。

其它

linux中替换某个文件中的字符串

linux查找某个字符串

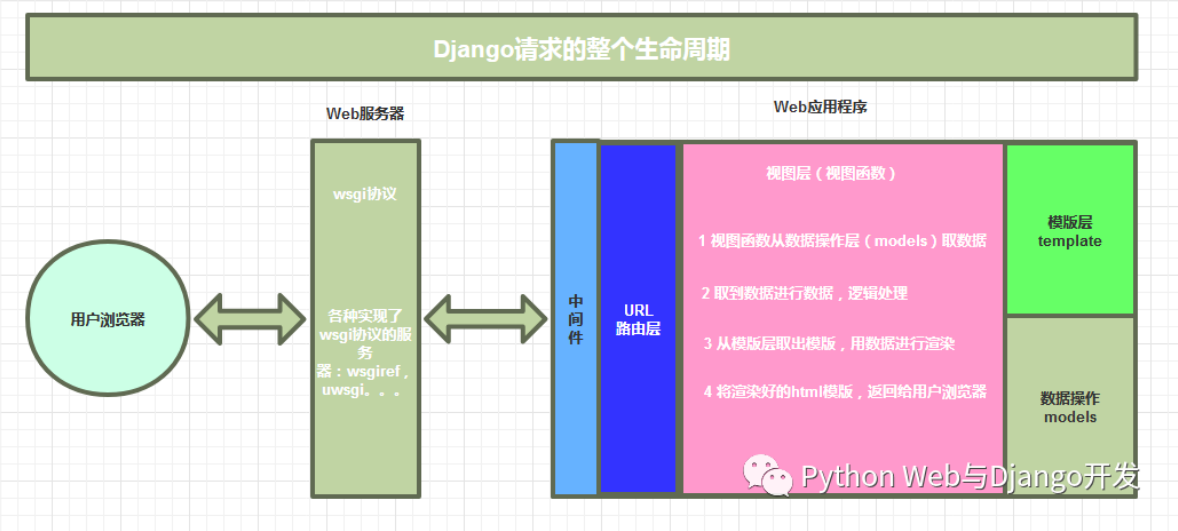
git问题：merge时候出现conflict怎么办

Python为什么跨平台

实习

Django

Django请求的生命周期



当用户在浏览器输入url时，浏览器会生成请求头和请求体发送给服务器，url经过Django中的wsgi的时候请求对象创建完成，再经过Django的中间件，然后到路由系统匹配路由，匹配成功后走到对应的views函数，视图函数执行相关的逻辑代码并返回执行结果，Django把客户端想要的数据作为一个字符串返回给客户端，客户端接收数据，渲染到页面展现给用户。

Restframework

简介

`Restframework`是一个能快速为我们提供`API`接口，方便我们编程的框架。`API`**是后端的人写的，为了让前端拿数据的一个接口，通常是以**`url`**的形式存在。**

`Django Restframework`可以帮助实现前后端分离开发，后台人员通过`Restframework`来开发`API`接口，前端人员写前端的页面（不论是PC还是移动端），**到时候整合直接使用后端人员的接口就可以完成整个项目的开发。**

Django REST Framework（简称DRF）是一个用于构建Web API的强大且灵活的框架。它是在Django基础上构建的，为开发人员提供了一系列工具和库，帮助快速构建高性能、可扩展的RESTful API。

以下是 Django REST Framework 提供的主要功能和特点：

1. 序列化：DRF 提供了强大的序列化机制，能够简化数据的序列化和反序列化过程。通过定义序列化器（Serializer），可以将模型数据转换为JSON、XML等格式（在服务器），并将请求数据反序列化为验证过的Python对象（在客户端）。
2. 视图：DRF 提供了视图类和装饰器来定义API视图，包括基于函数的视图（Function-Based Views）和基于类的视图（Class-Based Views）。这些视图类支持多种HTTP方法（GET、POST、PUT、DELETE等），并提供了通用视图类（Generic Views）以便快速实现常见的接口逻辑。
3. 路由：DRF 提供了简单易用的路由系统，可以映射 URL 到对应的视图函数或类。可以通过默认的简单路由器（SimpleRouter）或自定义路由器（Custom Router）来管理API的URL。
4. 身份验证与权限控制：DRF 内置了身份验证（Authentication）和权限控制（Permission）的机制，可以轻松实现用户认证和授权管理。支持常见的认证方式，如基于令牌的身份验证、会话身份验证等。
5. 响应处理：DRF 提供了多种响应渲染器（Renderer）和内容协商（Content Negotiation）选项，可以通过请求头部信息来返回适合客户端的响应格式，如JSON、XML等。
6. 分页和过滤：DRF 内建了分页（Pagination）和过滤器（Filtering）的功能，可轻松处理大量数据集的分页展示和筛选操作。
7. 认证与授权：DRF 支持多种身份验证和权限控制方式，包括基于标头、令牌、OAuth2等。

8. **文档生成：DRF 集成了强大的文档生成工具，可以基于定义的API视图类和序列化器自动生成交互式的Web API文档。这样可以减少文档编写的重复工作，并帮助团队更好地理解和使用API。**

Django REST Framework 是一个功能强大且广泛使用的框架，它使得构建高质量的Web API变得更加容易和高效。无论是小型项目还是大型企业级应用程序，DRF都能提供丰富的工具和特性来简化API开发过程。

Restframework在Django中的作用

Django REST framework（简称DRF）是一个强大的、灵活的用于构建Web API的库，它是基于Django框架的扩展。DRF提供了一系列工具和功能，使得在Django应用程序中创建和管理Web API变得更加容易。

下面是DRF在Django中的主要作用：

1. Web API开发：DRF提供了一套用于构建Web API的工具和框架。它可以方便地定义和序列化数据模型、处理请求和响应、进行身份验证和权限管理等。DRF使得开发者可以更快速和高效地构建出功能完善的Web API。
2. 序列化和反序列化：DRF的核心功能之一是提供了强大的序列化和反序列化功能，用于将复杂的数据结构转换为可传输的格式（如JSON）或从可传输格式转换为对象。这可以帮助开发者更轻松地在API中处理和传递数据，并支持自动验证和转换输入数据。
3. 身份验证和权限管理：DRF提供了各种身份验证机制，包括基于令牌(Token)、会话(Session)、OAuth等。它还支持灵活的权限管理，可以根据用户角色和权限对API进行保护和访问控制。
4. 视图和路由：DRF提供了用于创建API视图的类和函数，可以方便地定义API的行为和逻辑。DRF还提供了简单的路由功能，通过简单的配置即可将URL映射到相应的API视图，并支持自动生成API文档。
5. 认证和限流：DRF提供了多种认证选项（如JWT、OAuth等）和限流机制，可以帮助开发者保护API免受未经授权的访问和滥用。
6. API文档生成：DRF具有内置的API文档生成功能，可以根据代码自动生成交互式的API文档，包括接口列表、请求参数、响应数据等。这对于API的使用者和开发者来说非常有帮助，可以更好地理解和使用API。

综上所述，Django REST framework作为一个强大的Web API开发框架，为开发者提供了众多工具和功能，使得在Django应用程序中开发、管理和文档化Web API变得更加简单高效。它在简化开发流程、提供安全性和灵活性方面起到了重要作用。

序列化与反序列化

在 Django REST Framework（DRF）中，序列化和反序列化是一种将数据模型（如Django模型）与请求数据进行相互转换的过程。序列化将模型实例转换为可传输的数据格式，如JSON或XML，而反序列化将请求数据转换为模型实例。

DRF提供了强大的序列化和反序列化机制，其中主要使用的是序列化器（Serializer）类。

1. 序列化：
    - 创建一个继承自`serializers.Serializer`的序列化器类。
    - 在序列化器类中定义字段，这些字段对应于模型中的属性或关联字段。
    - 使用各种字段类型（如`CharField`、`IntegerField`、`DateTimeField`等）来声明和定制每个字段的行为和验证规则。
    - 根据需要，可以添加自定义方法和验证逻辑。
    - 调用序列化器的`to_representation`方法将模型实例转换为可传输格式的数据。
    - 示例：

```python
from rest_framework import serializers

class MyModelSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    name = serializers.CharField(max_length=100)
    created_at = serializers.DateTimeField()

    def to_representation(self, instance):
        # 自定义转换逻辑
        ...
```

2. 反序列化：
    - 创建一个继承自`serializers.Serializer`的序列化器类。
    - 在序列化器类中定义与客户端请求数据对应的字段。
    - 使用各种字段类型来声明每个字段的验证规则。
    - 调用序列化器的`is_valid`方法验证请求数据的有效性。
    - 如果数据有效，可以通过调用序列化器的`save`方法将数据保存到数据库中。
    - 示例：

```python
from rest_framework import serializers

class MyModelSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=100)
    email = serializers.EmailField()

    def save(self):
        # 自定义保存逻辑
        ...
```

通过序列化器，我们可以轻松地将模型实例转换为可传输的数据格式，并在反序列化时验证和保存来自客户端的请求数据。序列化器还提供了许多其他功能，如嵌套关系、字段级别的验证等，以帮助开发人员更好地管理和控制数据的转换和验证过程。

RESTful API

RESTful API（Representational State Transfer）是一种设计和构建 Web API 的软件架构风格。它基于HTTP协议，通过使用统一的资源标识符（URI）来表示资源，并通过不同的HTTP方法（GET、POST、PUT、DELETE等）对资源进行操作。

以下是 RESTful API 的主要特点和设计原则：

1. 资源（Resources）：API 的核心是资源，每个资源都有唯一的URI来标识。资源可以是实体（如用户、文章）或集合（如用户列表、文章列表）。资源通过URI进行访问和操作。
2. HTTP方法（HTTP Methods）：RESTful API 使用不同的HTTP方法来表示不同的操作。常用的方法包括：
    - GET：获取指定资源或资源列表。
    - POST：创建新资源。
    - PUT：更新指定资源。
    - DELETE：删除指定资源。
3. 表述性状态（Representational State）：资源的表述形式可以是多种多样的，如JSON、XML等。客户端和服务器之间通过资源的表述进行交互，服务器根据客户端请求的表述来响应。

4. 无状态（Stateless）：RESTful API 是无状态的，每个请求都独立处理，服务器不会保存客户端的状态信息。客户端在每个请求中提供必要的身份验证和完整的数据。
5. 统一接口（Uniform Interface）：RESTful API 使用统一的接口设计原则，包括以下几个方面：
    - 资源标识符（URI）：每个资源都有唯一的URI进行标识。
    - HTTP方法：使用不同的HTTP方法对资源进行操作。
    - 自描述消息（Self-descriptive Messages）：使用资源的表述来描述请求和响应。
    - 超媒体驱动（Hypermedia-Driven）：通过响应中提供相关链接（如HATEOAS）来引导客户端进行下一步操作。
6. 缓存（Caching）：RESTful API 支持缓存，服务器可以在响应中添加缓存控制头，以便客户端在后续请求时可以使用缓存数据，减少网络请求。
7. 安全性（Security）：RESTful API 可以使用各种安全机制来保护资源和用户数据，如基于令牌的身份验证、HTTPS等。

通过遵循这些设计原则，RESTful API 提供了一种简洁、可扩展和易于理解的方式来设计和构建 Web API。它具有高度的可见性和可测试性，能够满足不同平台和客户端的需求，并提供了良好的灵活性和互操作性。

python多线程

```
import RPi.GPIO as GPIO
import time
import threading
#指定第一个电机的GPIO口
IN1 = 40    # pin40
IN2 = 38
IN3 = 36
IN4 = 35
#指定第二个电机的GPIO口
IN5 = 18
IN6 = 16
IN7 = 13
IN8 = 15
#设置两台电机的频率和步数
motor1_delay = 0.0002
motor1_steps = 4000
motor2_delay = 0.02
motor2_steps = 400
#定义电机的类
class Motor:
    def __init__(self,I1,I2,I3,I4):
    #初始化类的参数
        self.I1 = I1
        self.I2 = I2
        self.I3 = I3
        self.I4 = I4
        GPIO.setwarnings(False)
        GPIO.setmode(GPIO.BOARD)       # 设置寻址模式，按照pin引脚寻址
        GPIO.setup(self.I1, GPIO.OUT)      # 指定IN1输出
        GPIO.setup(self.I2, GPIO.OUT)
        GPIO.setup(self.I3, GPIO.OUT)
        GPIO.setup(self.I4, GPIO.OUT)
def loop(self,delay_loop,steps_loop):
    print ("backward...")
    Motor.backward(self,delay_loop,steps_loop)
```

```
    print ("stop...")
    Motor.stop(self)                    # stop
    time.sleep(3)           # sleep 3s

    print ("forward...")
    Motor.forward(self,delay_loop,steps_loop)

    print ("stop...")
    Motor.stop(self)
    time.sleep(3)

def setStep(self, w1, w2, w3, w4):
    #GPIO.output（）---指定脚位输出high还是low
    GPIO.output(self.I1, w1)#.output（in1, 1）代表指定IN1引脚输出的是高电平，0就是低电
平
    GPIO.output(self.I2, w2)
    GPIO.output(self.I3, w3)
    GPIO.output(self.I4, w4)

def backward(self, delay, steps):
    for i in range(0, steps):
        Motor.setStep(self,1, 0, 0, 1)
        time.sleep(delay)
        Motor.setStep(self,0, 1, 0, 1)
        time.sleep(delay)
        Motor.setStep(self,0, 1, 1, 0)
        time.sleep(delay)
        Motor.setStep(self,1, 0, 1, 0)
        time.sleep(delay)

def forward(self,delay, steps):
    for i in range(0, steps):
        Motor.setStep(self,1, 0, 1, 0)
        time.sleep(delay)
        Motor.setStep(self,0, 1, 1, 0)
        time.sleep(delay)
        Motor.setStep(self,0, 1, 0, 1)
        time.sleep(delay)
        Motor.setStep(self,1, 0, 0, 1)
        time.sleep(delay)

def stop(self):
    Motor.setStep(self,0, 0, 0, 0)
def destroy():
    GPIO.cleanup()

if __name__ == '__main__':  #主函数测试
    motor1 = Motor(IN1,IN2,IN3,IN4)#实例化类
    #motor1.setup(IN1,IN2,IN3,IN4)
    motor2 = Motor(IN5,IN6,IN7,IN8)#实例化类
    #motor2.setup(IN5,IN6,IN7,IN8)
try:
    while True:
        shuru = int(input('input:'))
        if shuru ==1:
            thread1 = threading.Thread(target=motor1.loop,
```

```
                                        args=(motor1_delay,motor1_steps))#创建线程
1
            thread2 = threading.Thread(target=motor2.loop,
                                        args=(motor2_delay,motor2_steps))#创建线程
2
            thread1.start()#开始线程1
            thread2.start()#开始线程2

except KeyboardInterrupt:  # When 'Ctrl+C' is pressed, the child function
destroy() will be  executed.
    destroy()
```

python线程池

```
if __name__ == '__main__':  # Program start from here
    motor1=Motora(IN1,IN2,IN3,IN4)# null
    Motora.setup(motor1)
    motor2=Motorb(IN5,IN6,IN7,IN8)# camera
    Motorb.setup(motor2)
    motor3=Motorb(IN9,INa,INb,INc)# dianxiang
    Motorb.setup(motor3)

    try:
        # 创建线程池
        executor = concurrent.futures.ThreadPoolExecutor()

        while True:
            futures = []

            # 执行任务motor2.loop3（camera）
            future2 = executor.submit(motor2.loop3, motor2_delay, motor2_steps)
            future2.result()

            for i in range(2):
                # 提交任务motor1.loop4（null）到线程池中，但不立即执行
                future1 = executor.submit(motor1.loop4, motor1_delay,
motor1_steps)
                futures.append(future1)

                # 提交任务motor3.loop4（dianxiang）到线程池中，但不立即执行
                future3 = executor.submit(motor3.loop4, motor3_delay,
motor3_steps)
                futures.append(future3)

                # 等待所有任务完成
                for future in concurrent.futures.as_completed(futures):
                    future.result()

                # 执行任务motor2.loop3（camera）
                future2 = executor.submit(motor2.loop3, motor2_delay,
motor2_steps)
                future2.result()

            # 创建新的线程池用于并行执行motor1.loop5和motor3.loop5
            parallel_executor = concurrent.futures.ThreadPoolExecutor()
```

```
            # 并行执行任务motor1.loop5（null）和任务motor3.loop5（dianxiang），并等待
完成
            futures_parallel = [parallel_executor.submit(motor1.loop5, 2 *
motor1_delay, 2 * motor1_steps),
                                parallel_executor.submit(motor3.loop5, 2 *
motor3_delay, 2 * motor3_steps)]

            # 等待并行执行的任务完成
            concurrent.futures.wait(futures_parallel)

            # 关闭并行执行的线程池
            parallel_executor.shutdown()

    except KeyboardInterrupt:  # When 'Ctrl+C' is pressed, the child function
destroy() will be  executed.
        destroy()
```

实习中遇到的问题

多线程、线程池

深度图像与RGB图像对齐

分布式Linux性能分析监控

设计模式

单例模式

深入理解设计模式（一）：单例模式 - 一指流砂~ - 博客园 (cnblogs.com)

设计模式-单例模式（五种实现方法详解）-腾讯云开发者社区-腾讯云 (tencent.com)

单例模式是如何析构释放对象的

在单例模式中，对象的析构和释放通常由程序的终止负责，而不是由单例对象本身负责。当程序结束时，操作系统会回收所有分配的资源，包括单例对象占用的内存空间。

因为单例对象只有一个实例存在，所以它的析构和释放过程往往比较简单。一般情况下，单例对象不需要显式地释放资源，因为在程序终止时，操作系统会自动清理所有未释放的资源。

当然，有些编程语言提供了析构函数（Destructor）机制，允许在对象被销毁之前执行一些清理工作。对于单例对象，如果确实需要在它被销毁之前执行一些特定的操作，可以在析构函数中进行处理。例如，在C++中，可以通过定义类的析构函数来实现：

```
class Singleton {
public:
    static Singleton& getInstance() {
        static Singleton instance;
        return instance;
    }

    ~Singleton() {
        // 在析构函数中执行清理工作
```

```
        // ...
    }

private:
    // 私有构造函数和拷贝构造函数，确保只能通过getInstance方法获取唯一实例
    Singleton() {}
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
};
```

当程序终止时，单例对象的析构函数会被自动调用，从而实现对象的清理和资源的释放。需要注意的是，单例对象的析构函数应该是私有的或受保护的，以确保只能通过getInstance方法获取到唯一实例，并阻止通过其他方式创建和销毁对象。

综上所述，单例模式中的对象析构和释放通常由程序的终止负责，而不是由单例对象本身负责。在对象被销毁之前，可以使用析构函数执行一些清理工作。

工厂模式

工厂模式-简单工厂、工厂方法、抽象工厂解析 - 知乎 (zhihu.com)

设计模式中的原则

设计模式中涉及的面向对象编程（OOP）原则主要有以下六大原则：

1. 单一职责原则（Single Responsibility Principle，SRP）：一个类应该只有一个引起变化的原因。换句话说，每个类应该只负责完成一个单一的任务或功能。
2. 开闭原则（Open-Closed Principle，OCP）：软件实体（类、模块、函数等）应该对扩展开放，对修改关闭。也就是说，在变化需求下，我们应该尽量避免修改已有的代码，而是通过扩展来实现新的功能或适应新需求。
3. 里氏替换原则（Liskov Substitution Principle，LSP）：子类必须能够替换其父类并且程序行为不会产生变化。也就是说，任何基类可以被派生类替换，并且程序的表现行为不变。
4. 依赖倒置原则（Dependency Inversion Principle，DIP）：高层模块不应该依赖于低层模块，二者都应该依赖于抽象。抽象不应该依赖于细节，细节应该依赖于抽象。简单来说，依赖关系应该建立在抽象上，而不是具体的实现上。
5. 接口隔离原则（Interface Segregation Principle，ISP）：客户端不应该依赖它不使用的接口。也就是说，一个类对另一个类的依赖应该建立在最小的接口上。
6. 迪米特法则（Law of Demeter，LoD）：一个对象应该对其他对象有尽可能少的了解。也就是说，一个对象应该尽量减少与其他对象之间的直接交互，而通过尽量少的方法来操作其他对象。

这些面向对象编程原则提供了指导设计模式开发的基本原则，帮助我们编写出可维护、可扩展和高效的代码。设计模式是基于这些原则的实际应用，可以帮助我们解决一些常见的软件设计问题。

为什么要使用工厂模式，是为了满足OOP的六大原则的哪一点

工厂模式是一种常用的设计模式，用于封装对象的创建过程。它通过定义一个公共接口来创建对象，而不是直接调用构造函数，从而实现了创建对象的解耦和灵活性。

工厂模式可以满足面向对象编程（OOP）中的"**开闭原则**"（Open-Closed Principle）。开闭原则要求软件实体（类、模块、函数等）应该对扩展开放，对修改关闭。也就是说，在变化需求下，我们应该尽量避免修改已有的代码，而是通过扩展来实现新的功能或适应新需求。

具体来说，使用工厂模式可以将实例化对象的过程与客户端代码分离，将对象的创建过程封装在工厂类中。当需要增加新的产品类型时，只需扩展工厂类而不需要修改客户端代码，从而符合了开闭原则。

举个例子，假设我们有一个简单的图形绘制程序，可以绘制圆形和矩形。如果直接在客户端代码中通过构造函数创建图形对象，那么当需要新增三角形时，就需要修改客户端代码来支持三角形的创建。这违反了开闭原则。

而如果使用工厂模式，我们可以定义一个抽象的图形工厂接口，并派生出具体的圆形工厂和矩形工厂。客户端代码只需要通过图形工厂接口来创建图形对象，而不需要关心具体的实现类。当需要新增三角形时，只需实现一个三角形工厂，并在客户端代码中使用该工厂来创建三角形对象，而不需要修改已有的客户端代码。

因此，工厂模式可以满足OOP的开闭原则，使得代码更加灵活、可扩展，减少了代码的修改和维护成本。

docker

docker是什么

什么是docker，它与虚拟机有什么区别？_docker_itmkyuan-华为云开发者联盟 (csdn.net)

docker的工作原理是什么，讲一下

docker是一个Client-Server结构的系统，docker守护进程运行在宿主机上，守护进程从客户端接受命令并管理运行在主机上的容器，容器是一个运行时环境，这就是我们说的集装箱。

docker的组成包含哪几大部分

一个完整的docker有以下几个部分组成：
1、docker client，客户端，为用户提供一系列可执行命令，用户用这些命令实现跟 docker daemon 交互；
2、docker daemon，守护进程，一般在宿主主机后台运行，等待接收来自客户端的请求消息；
3、docker image，镜像，镜像run之后就生成为docker容器；
4、docker container，容器，一个系统级别的服务，拥有自己的ip和系统目录结构；运行容器前需要本地存在对应的镜像，如果本地不存在该镜像则就去镜像仓库下载。

docker 使用客户端-服务器 (C/S) 架构模式，使用远程api来管理和创建docker容器。docker 容器通过 docker 镜像来创建。容器与镜像的关系类似于面向对象编程中的对象与类。

docker与传统虚拟机的区别什么?

1、传统虚拟机是需要安装整个操作系统的，然后再在上面安装业务应用，启动应用，通常需要几分钟去启动应用，而docker是直接使用镜像来运行业务容器的，其容器启动属于秒级别；
2、Docker需要的资源更少，Docker在操作系统级别进行虚拟化，Docker容器和内核交互，几乎没有性能损耗，而虚拟机运行着整个操作系统，占用物理机的资源就比较多；
3、Docker更轻量，Docker的架构可以共用一个内核与共享应用程序库，所占内存极小;同样的硬件环境，Docker运行的镜像数远多于虚拟机数量，对系统的利用率非常高；
4、与虚拟机相比，Docker隔离性更弱，Docker属于进程之间的隔离，虚拟机可实现系统级别隔离；
5、Docker的安全性也更弱，Docker的租户root和宿主机root相同，一旦容器内的用户从普通用户权限提升为root权限，它就直接具备了宿主机的root权限，进而可进行无限制的操作。虚拟机租户root权限和宿主机的root虚拟机权限是分离的，并且虚拟机利用如Intel的VT-d和VT-x的ring-1硬件隔离技术，这种技术可以防止虚拟机突破和彼此交互，而容器至今还没有任何形式的硬件隔离；
6、Docker的集中化管理工具还不算成熟，各种虚拟化技术都有成熟的管理工具，比如：VMware vCenter提供完备的虚拟机管理能力；
7、Docker对业务的高可用支持是通过快速重新部署实现的，虚拟化具备负载均衡，高可用、容错、迁移和数据保护等经过生产实践检验的成熟保障机制，Vmware可承诺虚拟机99.999%高可用，保证业务连续性；

8、虚拟化创建是分钟级别的，Docker容器创建是秒级别的，Docker的快速迭代性，决定了无论是开发、测试、部署都可以节省大量时间;
9、虚拟机可以通过镜像实现环境交付的一致性，但镜像分发无法体系化，Docker在Dockerfile中记录了容器构建过程，可在集群中实现快速分发和快速部署。

docker技术的三大核心概念是什么?

镜像：镜像是一种轻量级、可执行的独立软件包，它包含运行某个软件所需的所有内容，我们把应用程序和配置依赖打包好形成一个可交付的运行环境(包括代码、运行时需要的库、环境变量和配置文件等)，这个打包好的运行环境就是image镜像文件。
容器：容器是基于镜像创建的，是镜像运行起来之后的一个实例，容器才是真正运行业务程序的地方。如果把镜像比作程序里面的类，那么容器就是对象。
镜像仓库：存放镜像的地方，研发工程师打包好镜像之后需要把镜像上传到镜像仓库中去，然后就可以运行有仓库权限的人拉取镜像来运行容器了。

centos镜像几个G，但是docker centos镜像才几百兆，这是为什么?

一个完整的Linux操作系统包含Linux内核和rootfs根文件系统，即我们熟悉的/dev、/proc/、/bin等目录。我们平时看到的centOS除了rootfs，还会选装很多软件，服务，图形桌面等，所以centOS镜像有好几个G也不足为奇。
而对于容器镜像而言，所有容器都是共享宿主机的Linux 内核的，而对于docker镜像而言，docker镜像只需要提供一个很小的rootfs即可，只需要包含最基本的命令，工具，程序库即可，所有docker镜像才会这么小。

讲一下镜像的分层结构以及为什么要使用镜像的分层结构?

一个新的镜像其实是从 base 镜像一层一层叠加生成的。每安装一个软件，dockerfile中使用RUM命令，就会在现有镜像的基础上增加一层，这样一层一层的叠加最后构成整个镜像。所以我们docker pull拉取一个镜像的时候会看到docker是一层层拉去的。

分层机构最大的一个好处就是： 共享资源。比如：有多个镜像都从相同的 base 镜像构建而来，那么 Docker Host 只需在磁盘上保存一份 base 镜像；同时内存中也只需加载一份 base 镜像，就可以为所有容器服务了。而且镜像的每一层都可以被共享。

讲一下容器的copy-on-write特性，修改容器里面的内容会修改镜像吗?

我们知道，镜像是分层的，镜像的每一层都可以被共享，同时，镜像是只读的。当一个容器启动时，一个新的可写层被加载到镜像的顶部，这一层通常被称作“容器层”，“容器层”之下的都叫“镜像层”。

所有对容器的改动 - 无论添加、删除、还是修改文件，都只会发生在容器层中，因为只有容器层是可写的，容器层下面的所有镜像层都是只读的。镜像层数量可能会很多，所有镜像层会联合在一起组成一个统一的文件系统。如果不同层中有一个相同路径的文件，比如 /a，上层的 /a 会覆盖下层的 /a，也就是说用户只能访问到上层中的文件 /a。在容器层中，用户看到的是一个叠加之后的文件系统。

添加文件时：在容器中创建文件时，新文件被添加到容器层中。
读取文件：在容器中读取某个文件时，Docker 会从上往下依次在各镜像层中查找此文件。一旦找到，立即将其复制到容器层，然后打开并读入内存。
修改文件：在容器中修改已存在的文件时，Docker 会从上往下依次在各镜像层中查找此文件。一旦找到，立即将其复制到容器层，然后修改之。
删除文件：在容器中删除文件时，Docker 也是从上往下依次在镜像层中查找此文件。找到后，会在容器层中记录下此删除操作。

只有当需要修改时才复制一份数据，这种特性被称作 Copy-on-Write。可见，容器层保存的是镜像变化的部分，不会对镜像本身进行任何修改。

简单描述一下Dockerfile的整个构建镜像过程

好的。
1、首先，创建一个目录用于存放应用程序以及构建过程中使用到的各个文件等；
2、然后，在这个目录下创建一个Dockerfile文件，一般建议Dockerfile的文件名就是Dockerfile；
3、编写Dockerfile文件，编写指令，如，使用FROM 指令指定基础镜像，COPY指令复制文件，RUN指令指定要运行的命令，ENV设置环境变量，EXPOSE指定容器要暴露的端口，WORKDIR设置当前工作目录，CMD容器启动时运行命令，等等指令构建镜像；
4、Dockerfile编写完成就可以构建镜像了，使用`docker build -t 镜像名:tag<span> </span>.`命令来构建镜像，最后一个点是表示当前目录，docker会默认寻找当前目录下的Dockerfile文件来构建镜像，如果不使用默认，可以使用-f参数来指定dockerfile文件，如：`docker build -t 镜像名:tag<span> </span>-f<span> </span>/xx/xxx/Dockerfile`；
5、使用docker build命令构建之后，docker就会将当前目录下所有的文件发送给docker daemon，顺序执行Dockerfile文件里的指令，在这过程中会生成临时容器，在临时容器里面安装RUN指定的命令，安装成功后，docker底层会使用类似于docker commit命令来将容器保存为镜像，然后删除临时容器，以此类推，一层层的构建镜像，运行临时容器安装软件，直到最后的镜像构建成功。

Dockerfile构建镜像出现异常，如何排查?

首先，Dockerfile是一层一层的构建镜像，期间会产生一个或多个临时容器，构建过程中其实就是在临时容器里面安装应用，如果因为临时容器安装应用出现异常导致镜像构建失败，这时容器虽然被清理掉了，但是期间构建的中间镜像还在，那么我们可以根据异常时上一层已经构建好的临时镜像，将临时镜像运行为容器，然后在容器里面运行安装命令来定位具体的异常。

Dockerfile的基本指令有哪些?

FROM 指定基础镜像（必须为第一个指令，因为需要指定使用哪个基础镜像来构建镜像）；
MAINTAINER 设置镜像作者相关信息，如作者名字，日期，邮件，联系方式等；
COPY 复制文件到镜像；
ADD 复制文件到镜像（ADD与COPY的区别在于，ADD会自动解压tar、zip、tgz、xz等归档文件，而COPY不会，同时ADD指令还可以接一个url下载文件地址，一般建议使用COPY复制文件即可，文件在宿主机上是什么样子复制到镜像里面就是什么样子这样比较好）；
ENV 设置环境变量；
EXPOSE 暴露容器进程的端口，仅仅是提示别人容器使用的哪个端口，没有过多作用；
VOLUME 数据卷持久化，挂载一个目录；
WORKDIR 设置工作目录，如果目录不在，则会自动创建目录；
RUN 在容器中运行命令，RUN指令会创建新的镜像层，RUN指令经常被用于安装软件包；
CMD 指定容器启动时默认运行哪些命令，如果有多个CMD，则只有最后一个生效，另外，CMD指令可以被docker run之后的参数替换；
ENTRYOINT 指定容器启动时运行哪些命令，如果有多个ENTRYOINT，则只有最后一个生效，另外，如果Dockerfile中同时存在CMD和ENTRYOINT，那么CMD或docker run之后的参数将被当做参数传递给ENTRYOINT；

如何进入容器？使用哪个命令

进入容器有两种方法：docker attach、docker exec；
docker attach命令是attach到容器启动命令的终端，docker exec 是另外在容器里面启动一个TTY终端。

```
docker run -d centos /bin/bash -c "while true;do sleep 2;echo
I_am_a_container;done"
3274412d88ca4f1d1292f6d28d46f39c14c733da5a4085c11c6a854d30d1cde0
docker attach 3274412d88ca4f                          #attach进入容器
Ctrl + c  退出，Ctrl + c会直接关闭容器终端，这样容器没有进程一直在前台运行就会死掉了
Ctrl + pq 退出（不会关闭容器终端停止容器，仅退出）

docker exec -it 3274412d88ca /bin/bash                #exec进入容器
[root@3274412d88ca /]# ps -ef                         #进入到容器了开启了一个bash进程
UID         PID   PPID  C STIME TTY          TIME CMD
root          1      0  0 05:31 ?        00:00:01 /bin/bash -c while true;do
sleep 2;echo I_am_a_container;done
root        306      0  1 05:41 pts/0    00:00:00 /bin/bash
root        322      1  0 05:41 ?        00:00:00 /usr/bin/coreutils --
coreutils-prog-shebang=sleep /usr/bin/sleep 2
root        323    306  0 05:41 pts/0    00:00:00 ps -ef
[root@3274412d88ca /]#exit                            #退出容器，仅退出我们自己的bash窗
口
```

小结：attach是直接进入容器启动命令的终端，不会启动新的进程；exec则是在容器里面打开新的终端，会启动新的进程；一般建议已经exec进入容器。

protobuf

反射

ProtoBuf-反射原理与使用*protobuf 反射*末日在做什么呢的博客-CSDN博客

protobuf序列化后的数据为什么相比xml更小

浅析 Protobuf 整形编码方式：Varint 与 Zigzag - 掘金 (juejin.cn)

protobuf序列化后的数据相比XML更小，其中使用了Varint编码和ZigZag编码这两种技术，有助于进一步减小数据大小。

1. Varint编码：Varint是一种可变长度编码方式，用于对整数类型进行序列化。Varint通过动态地选择字节长度来表示整数，较小的整数使用较少的字节数。具体实现方式是将整数按7位为一个单位进行分组，并使用最高位的第8位表示是否继续读取下一个字节。这样，在protobuf序列化中，较小的整数只需要使用较少的字节来表示，从而减小了数据的大小。
2. ZigZag编码：ZigZag编码是Varint编码的一种扩展，用于对有符号整数进行编码。由于Varint编码只针对无符号整数，但在实际应用中，我们通常需要对有符号整数进行序列化。ZigZag编码通过将有符号整数映射到无符号整数，并利用Varint编码进行序列化。具体的映射规则是将正整数左移一位后加1，负整数左移一位后取反再加1。这种编码方式可以有效地压缩有符号整数的表示，减小数据的大小。

通过使用Varint编码和ZigZag编码，protobuf可以更紧凑地表示整数类型的字段，从而减小了序列化后数据的大小。这两种编码方式结合在一起，能够有效地减少不必要的字节数，并提升网络传输和存储的效率。

monitor

容器里CPU信息，系统负载和虚拟机的负载相同吗

相同

在容器里运行的话，获取的CPU信息和中断是容器的还是虚拟机的，两个不一样

虚拟机

grpc

rpc原理

RPC框架：从原理到选型，一文带你搞懂RPC-腾讯云开发者社区-腾讯云 (tencent.com)

什么是RPC

RPC（Remote Procedure Call Protocol）远程过程调用协议。一个通俗的描述是：客户端在不知道调用细节的情况下，调用存在于远程计算机上的某个对象，就像调用本地应用程序中的对象一样。

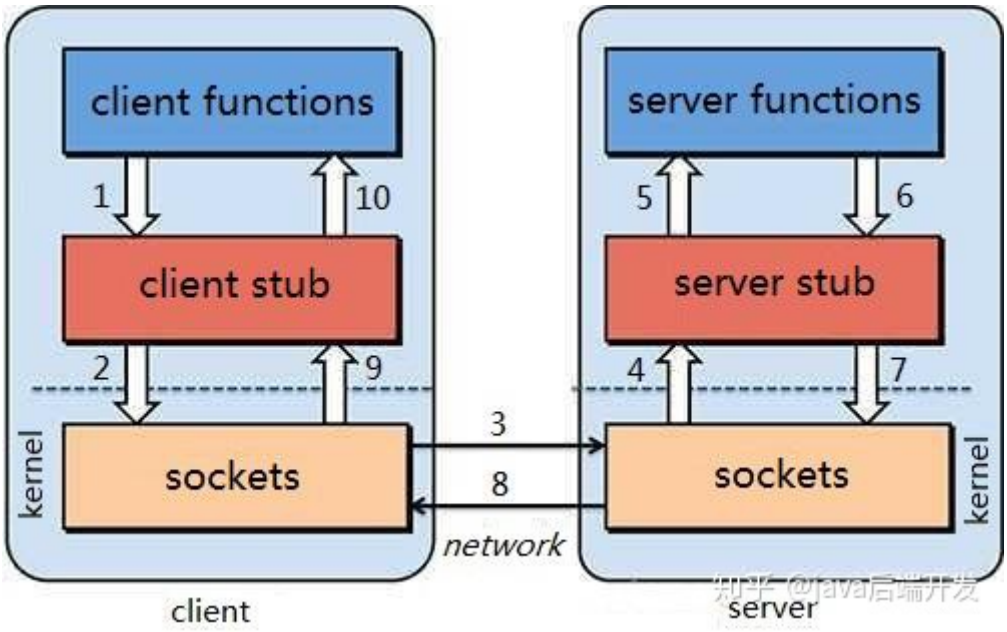
比较正式的描述是：一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。

RPC架构

一个完整的RPC架构里面包含了四个核心的组件，分别是Client，Client Stub，Server以及Server Stub，这个Stub可以理解为存根。

- 客户端(Client)，服务的调用方。
- 客户端存根(Client Stub)，存放服务端的地址消息，再将客户端的请求参数打包成网络消息，然后通过网络远程发送给服务端。
- 服务端(Server)，真正的服务提供者。
- 服务端存根(Server Stub)，接收客户端发送过来的消息，把消息解包，并调用本地的方法。

RPC调用过程



(1) 客户端（client）以本地调用方式（即以接口的方式）调用服务；

(2) 客户端存根（client stub）接收到调用后，负责将方法、参数等组装成能够进行网络传输的消息体（将消息体对象序列化为二进制）；

(3) 客户端通过sockets将消息发送到服务端；

(4) 服务端存根(server stub）收到消息后进行解码（将消息对象反序列化）；

(5) 服务端存根(server stub）根据解码结果调用本地的服务；

(6) 本地服务执行并将结果返回给服务端存根(server stub）；

(7) 服务端存根(server stub）将返回结果打包成消息（将结果消息对象序列化）；

(8) 服务端（server）通过sockets将消息发送到客户端；

(9) 客户端存根（client stub）接收到结果消息，并进行解码（将结果消息发序列化）；

(10) 客户端（client）得到最终结果。

RPC的目标是要把2、3、4、7、8、9这些步骤都封装起来。
注意：无论是何种类型的数据，最终都需要转换成二进制流在网络上进行传输，数据的发送方需要将对象转换为二进制流，而数据的接收方则需要把二进制流再恢复为对象。

对分布式的理解

分布式 - 谈谈你对分布式的理解，为什么引入分布式？ Q.E.D.的博客-CSDN博客

qt

信号槽

Qt信号与槽使用方法最完整总结 - AI观星台 - 博客园 (cnblogs.com)

Qt一篇全面的信号和槽函数机制总结 - 知乎 (zhihu.com)

Qt教程(3) : 信号与槽*信号与槽函数同名*QtHalcon的博客-CSDN博客

消息推送

Web端即时通讯、消息推送的实现 Quiet-Night的博客-CSDN博客

其它

项目遇到的困难

monitor模块中的join与qt模块中的detach

monitor模块中cpu_stat，每循环一次，就要清空一次cpu_stat_list.clear()

如果不按功能进行模块的划分，还有什么划分的方法

模块划分的原则及方法*模块划分的原则是什么*忆常的博客-CSDN博客

是否碰到过，一个模块强依赖于另一个模块，有什么方法可以降低代码的耦合性

在开发程序时，常常会遇到模块之间存在强依赖关系的情况。为了降低代码的耦合性，可以采取以下方法：

1. 使用接口和抽象类：将依赖关系抽象成接口或者抽象类，而不是直接依赖具体的实现类。这样可以将依赖的具体实现延迟到运行时动态决定，减少模块之间的直接依赖关系，提高代码的灵活性和可维护性。
2. 依赖注入（Dependency Injection）：通过依赖注入的方式，将一个模块所依赖的其他模块通过接口或构造函数等方式注入进来，而不是在模块内部直接实例化和依赖其他模块。这样可以降低模块之间的直接耦合，提高代码的可测试性和可扩展性。

3. 事件驱动架构：使用事件驱动的架构模式可以降低模块之间的耦合度。每个模块通过监听和触发事件的方式进行通信，而不是直接依赖其他模块的具体实现。这样每个模块只需要关注自身需要处理的事件，而不需要关心其他模块的细节，降低了模块之间的耦合性。
4. 中介者模式：中介者模式将多个模块之间的复杂交互逻辑集中到一个中介者对象中，其他模块只需要和中介者进行通信，而不需要直接依赖其他模块。这样可以减少模块之间的直接依赖关系，提高代码的可维护性和可扩展性。
5. 模块化设计：将程序划分成多个独立的模块，每个模块只关注自身的功能和责任，并定义清晰的接口进行通信。遵循单一职责原则和开闭原则等设计原则，将模块之间的依赖关系降低到最小，提高代码的可重用性和可测试性。

通过使用上述方法，可以有效降低模块之间的耦合性，提高代码的可维护性、扩展性和可测试性。选择合适的方法取决于具体的项目需求和架构设计。

多个服务端的数据到达客户端是并发的吗

这个项目不是一个客户端对多个客户端的模型，是一个客户端对一个服务端。一台主机监控多个服务器性能是开了多个客户端的，是一对一模型。
那么怎么实现一对多？

考虑多线程，qt的qthread

怎么测试能连接多少个

grpc有ghz工具，可以压测，常用的压测工具也可以

开放性问题

优缺点

优点：做事严谨认真，会详细检查

缺点：有点强迫症

团队合作的利弊

利益：

1. 分工合作：团队合作可以将工作任务分解成更小的工作单元，并由不同的团队成员负责完成。这样可以充分发挥每个人的专长和优势，提高工作效率和质量。
2. 促进创新：团队合作可以汇聚不同的思想、经验和观点，通过成员之间的交流和讨论，促进创新和灵感的产生。团队中的各种想法和见解可以相互激发，帮助团队思考出更好的解决方案。
3. 提高工作质量：通过团队合作，成员之间可以相互审查和检查彼此的工作，减少错误和疏漏的几率。团队成员可以互相监督和提醒，确保工作质量达到最佳水平。
4. 共享资源和知识：团队合作可以促进成员之间的信息共享和知识传递。成员可以通过交流和学习，不断增加自己的知识和技能，并将其应用到团队的工作中，提升整个团队的能力。

限制：

1. 沟通成本高：团队合作需要成员之间进行大量的沟通和协调，包括讨论、会议、文件共享等。如果沟通不畅或耗费过多时间，可能导致效率降低和进度延误。
2. 不一致和冲突：团队合作中，不同成员可能有不同的意见、目标或工作方式，容易出现意见不合和冲突。如果不能有效解决冲突，可能会对团队的效果和氛围产生负面影响。
3. 责任分散：在团队合作中，责任的分配和追踪可能更加复杂。如果没有明确的角色和责任分工，可能导致成员之间互相推诿责任，最终影响项目的进展和结果。

4. 依赖性和缺乏个人自由：团队合作意味着成员需要相互依赖和配合工作，有时可能会减少个人的自主性和自由度。某些成员可能会感到束缚，无法按照自己的方式和节奏进行工作。

团队合作的利与弊是相互关联的，并且受到团队成员素质、团队氛围、项目需求等多个因素的影响。对于团队合作，关键是在合作过程中积极解决问题、加强沟通和协调，以提高团队效率和工作质量。