鹅厂

面试职位：go后端开发工程师，接受从Java转语言

都知道鹅厂是cpp的主战场，而以cpp为背景的工程师大都对os，network这块要求特别高，不像是Java这种偏重业务层的语言，之前面试Java的公司侧重还是在数据结构、网络、框架、数据库和分布式。所以OS这块吃的亏比较大

一面基础技术面

电话面试，随便问了些技术问题，最后还问了个LeetCode里面medium级别的算法题，偏简单

1. redis有没有用过，常用的数据结构以及在业务中使用的场景，redis的hash怎么实现的，rehash过程讲一下和JavaHashMap的rehash有什么区别？redis cluster有没有了解过，怎么做到高可用的？redis的持久化机制，为啥不能用redis做专门的持久化数据库存储？
2. 了不了解tcp/udp，说下两者的定义，tcp为什么要三次握手和四次挥手？tcp怎么保证有序传输的，讲下tcp的快速重传和拥塞机制，知不知道time_wait状态，这个状态出现在什么地方，有什么用？（参考quic）
3. 知道udp是不可靠的传输，如果你来设计一个基于udp差不多可靠的算法，怎么设计？
4. http与https有啥区别？说下https解决了什么问题，怎么解决的？说下https的握手过程。
5. 看你项目里面用了etcd，讲解下etcd干什么用的，怎么保证高可用和一致性？
6. 既然你提到了raft算法，讲下raft算法的基本流程？raft算法里面如果出现脑裂怎么处理？有没有了解过paxos和zookeeper的zab算法，他们之前有啥区别？
7. 你们后端用什么数据库做持久化的？有没有用到分库分表，怎么做的？
8. 索引的常见实现方式有哪些，有哪些区别?MySQL的存储引擎有哪些，有哪些区别？InnoDB使用的是什么方式实现索引，怎么实现的？说下聚簇索引和非聚簇索引的区别？
9. 有没有了解过协程？说下协程和线程的区别？
10. 算法题一个，剑指offer第51题，数组中的重复数字？

自己的回答情况，redis这块没啥问题，具体rehash有印象是渐进式的，但是具体原理可能答的有点出入。tcp的time_wait这块答的不是很好，之前没有了解过quic机制的实现，所以问可靠性udp的时候，基本上脑子里就照着tcp的实现在说。https这块没啥说的，之前项目里面有用到类似的东西，研究的比较清楚了。raft算法这个因为刚好在刷6.824（才刷到lab2。。。），答的也凑合，不过paxos和zab算法确实不熟悉，直接说不会。MySQL这块很熟了，包括索引，锁，事务机制以及mvcc等等，没啥说的，都已经补齐了。协程和线程，主要说了go程和Java线程的区别以及go程的调度模型。面试官提示没有提到线程的有内核态的切换，go程只在用户态调度。最后一个算法题，首先说使用HashMap来做，说空间复杂度能不能降到O(1)，后面想了大概5min才想出来原地置换的思路。

二面项目技术面

1. 主要针对自己最熟悉的项目，画出项目的架构图，主要的数据表结构，项目中使用到的技术点，项目的总峰值qps，时延，以及有没有分析过时延出现的耗时分别出现在什么地方，项目有啥改进的地方没有？
2. 如果请求出现问题没有响应，如何定位问题，说下思路？
3. tcp 粘包问题怎么处理？
4. 问了下缓存更新的模式，以及会出现的问题和应对思路？
5. 除了公司项目之外，业务有没有研究过知名项目或做出过贡献？

基本都没有啥问题，除了面试官说项目经验稍弱之外，其余还不错。

三面综合技术面

这面面的是阵脚大乱，面试官采用刨根问底的方式提问，终究是面试经验不够，导致面试的节奏有点乱。 举个例子：

> 其中有个题是go程和线程有什么区别？ 答：1 起一个go程大概只需要4kb的内存，起一个Java线程需要1.5MB的内存；go程的调度在用户态非常轻量，Java线程的切换成本比较高。接着问为啥成本比较高？因为Java线程的调度需要在用户态和内核态切换所以成本高？为啥在用户态和内核态之间切换调度成本比较高？简单说了下内核态和用户态的定义。接着问，还是没有明白为啥成本高？心里瞬间崩溃，没完没了了呀，OS这块依旧是痛呀，支支吾吾半天放弃了。

后面所有的提问都是这种模式，结果回答的节奏全无，感觉被套路了。大多度都能回答个一二甚至是一二三，但是再往后或者再深入的OS层面就GG了。

后面问了下项目过程中遇到的最大的挑战，以及时怎么解决的？

后面还问了一个问题定位的问题，服务器CPU 100%怎么定位？可能是由于平时定位业务问题的思维定势，加之处于蒙蔽状态，随口就是：先查看监控面板看有无突发流量异常，接着查看业务日志是否有异常，针对CPU100%那个时间段，取一个典型业务流程的日志查看。最后才提到使用`top`命令来监控看是哪个进程占用到100%。果然阵脚大乱，张口就来，捂脸。。。 本来正确的思路应该是先用`top`定位出问题的进程，再用`top`定位到出问题的线程，再打印线程堆栈查看运行情况，这个流程换平时肯定能答出来，但是，但是没有但是。还是得好好总结。

最后问了一个系统设计题目（朋友圈的设计），白板上面画出系统的架构图，主要的表结构和讲解主要的业务流程，如果用户变多流量变大，架构将怎么扩展，怎样应对？ 这个答的也有点乱，直接上来自顾自的用了一个通用的架构，感觉毫无亮点。后面反思应该先定位业务的特点，这个业务明显是读多写少，然后和面试官沟通一期刚开始的方案的用户量，性能要求，单机目标qps是什么等等？在明确系统的特点和约束之后再来设计，而不是一开始就是用典型互联网的那种通用架构自顾自己搞自己的方案。

总结

1. tcp/udp，http和https还有网络这块（各种网络模型，已经select，poll和epoll）一定要非常熟悉
2. 一定要有拿的出手的项目经验，而且要能够讲清楚，讲清楚项目中取舍，设计模型和数据表
3. 分布式要非常熟悉
4. 常见问题定位一定要有思路
5. 操作系统，还是操作系统，重要的事情说三遍
6. 系统设计，思路，思路，思路，一定要思路清晰，一定要总结下系统设计的流程
7. 一点很重要的心得，平时blog和专栏看的再多，如果没有自己的思考不过是过眼云烟，根本不会成为自己的东西，就像内核态和用户态，平常也看过，但是没细想，突然要自己说，还真说不出来，这就很尴尬了。勿以浮沙筑高台，基础这种东西还是需要时间去慢慢打牢，多去思考和总结。

https://osjobs.net/topk/%E8%85%BE%E8%AE%AF/

C++

C++ 中智能指针和指针的区别是什么？

简单 参考1 参考2

1. 为什么要用智能指针?

①忘记释放导致内存泄漏

```
int main(){
    int *ptr = new int(0);
    return 0;
}
```

②产生野指针：程序虽然最后释放了申请的内存，但ptr会变成空悬指针（dangling pointer，也就是野指针）。空悬指针不同于空指针（nullptr），它会指向“垃圾”内存，给程序带去诸多隐患。

```
int main(){
     int *ptr = new int(0);
     delete ptr;
     return 0;
}
```

③由异常引起的内存泄漏：如下，当我们的程序运行到“if(hasException())”处且“hasException()”为真，那程序将会抛出一个异常，最终导致程序终止，而已申请的内存并没有释放掉。

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

int main()
{
    int *ptr = new(nothrow) int(0);
    if(!ptr)
    {
        cout << "new fails."
        return 0;
    }
    // 假定hasException函数原型是 bool hasException()
    if (hasException())
        throw exception();

    delete ptr;
    ptr = nullptr;
    return 0;
}
```

2. 什么是智能指针?

- 智能指针(smart pointer)是存储指向动态分配（堆）对象指针的类，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。
- 它的一种通用实现技术是使用引用计数(reference count)。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，减少引用计数（如果引用计数减至0，则删除基础对象）。

- **智能指针就是模拟指针动作的类。**所有的智能指针都会重载 -> 和 * 操作符。智能指针还有许多其他功能，比较有用的是自动销毁。这主要是利用栈对象的有限作用域以及临时对象（有限作用域实现）析构函数释放内存。当然，智能指针还不止这些，还包括复制时可以修改源对象等。智能指针根据需求不同，设计也不同（写时复制，赋值即释放对象拥有权限、引用计数等，控制权转移等）。

C++ 提供了四个智能指针模板类, 分别是: `auto_ptr`, `unique_ptr`, `shared_ptr`和`weak_ptr`. (`auto_ptr`是 C++98 提供的解决方案, C++11 已经将其摒弃, 并提供了另外三种解决方案). 这三个智能指针模板都定义了类似指针的对象, 可以将`new`获得(直接或间接)的地址赋给这种对象. **当智能指针过期时, 其析构函数将使用`delete`来释放内存.** (要创建智能指针对象, 需要包含头文件`<memory>`)

3、三种智能指针的区别?

- auto_ptr: 当进行赋值时, 会将旧指针的所有权转让, 使得 **对于特定的对象, 只能有一个智能指针可以拥有它**.
- unique_ptr: 当进行赋值时, 会将旧指针的所有权转让, 使得 **对于特定的对象, 只能有一个智能指针可以拥有它**. unique_ptr 相比于 auto_ptr 会执行更加严格的所有权转让策略
- shared_ptr: 通过引用计数(reference counting), 跟踪引用特定特定对象的智能指针数. 当发生赋值操作时, 计数增1, 当指针过期时, 计数减1. 仅当最后一个指针过期时, 才调用 delete.
- weak_ptr: 不控制对象声明周期的智能指针, 它指向一个 shared_ptr 管理的对象, 而进行内存管理的只有 shared_ptr. weak_ptr 主要用来帮助解决循环引用问题, 它的构造和析构不会引起引用计数的增加或者减少. weak_ptr 一般都是配合 shared_ptr 使用, 通常不会单独使用.

unique_ptr 和 auto_ptr 的区别?

- 所有权转让机制不同: auto_ptr 允许通过直接赋值进行转让, 但是这样会留下危险的 **悬挂指针**, 容易使得程序在运行阶段崩溃. unique_ptr 仅仅允许将临时右值进行赋值, 否则会在编译阶段发生错误, 这样更加安全(编译阶段错误比潜在的程序崩溃更安全).
- 相比于 auto_ptr 和 share_ptr, unique_ptr 可以使用`new[]`分配的内存作为参数: `std::unique_ptr<double[]> pda(new double(5));`

如何选择合适的只能指针?

如果程序要使用多个指向同一个对象的指针, 应选择`shared_ptr`, 这样的情况包括:

- 对于智能指针数组, 用辅助指针来标识最大值或最小值的情况
- 很多 STL 算法都支持复制和赋值操作, 这些操作可用于`shared_ptr`, 但不能用于`unique_ptr`和`auto_ptr`.

如果程序不需要多个指向同一个对象的指针, 则可以使用`unique_ptr`, 如果函数使用`new`分配内存, 并返回指向该内存的指针, 将其返回类型声明为`unique_ptr`是不错的选择, 这样, 所有权将转让给接受返回值的`unique_ptr`.

4. 野指针

1. 什么是野指针?

野指针和空指针不一样，是一个指向垃圾内存的指针。

2. 为什么会产生野指针?

①指针变量没有被初始化:

任何指针变量被刚创建时不会被自动初始化为NULL指针，它的缺省值是随机的。所以，指针变量在创建的同时应当被初始化，要么将指针设置为NULL，要么让它指向合法的内存。

②指针被free或者delete之后，没有设置为NULL，让人误以为这是一个合法指针： free和delete只是把指针所指向的内存给释放掉，但并没有把指针本身给清理掉。这时候的指针依然指向原来的位置，只不过这个位置的内存数据已经被毁尸灭迹，此时的这个指针指向的内存就是一个垃圾内存。但是此时的指针由于并不是一个NULL指针（在没有置为NULL的前提下）。

③指针操作超越了变量的作用范围： 由于C/C++中指针有++操作，因而在执行该操作的时候，稍有不慎，就容易指针访问越界，访问了一个不该访问的内存，结果程序崩溃。 另一种情况是指针指向一个临时变量的引用，当该变量被释放时，此时的指针就变成了一个野指针。

简述 C++ 右值引用与转移语义

简单 参考1 参考2

C++11中的右值引用是非常重要的特性，带来了语言层面的性能优化，其使用方法简洁方便，利于掌握。

1.右值引用特性的目的

右值引用 (Rvalue Referene) 是 C++ 新标准 (C++11, 11 代表 2011 年) 中引入的新特性 , 它实现了转移语义 (Move Sementics) 和精确传递 (Perfect Forwarding)。它的主要目的有两个方面:

- 消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。
- 能够更简洁明确地定义泛型函数。

2.左值和右值

2.1 定义

网上一堆关于左值和右值定义的博客，我觉得有的根本就是错误的，建议直接看primer的定义：**一个左值表达式表示的是对象的身份，而右值表达式表示的是对象的值**，我个人的理解很简单：**左值对应变量的存储位置，而右值对应变量的值本身。**因此右值引用同样也是实际内存对象的一个名字，而左值要求一个明确的对象（不然哪来的身份），右值则为常见的立即数、临时对象，这些只要在内存中有明确的值存在的对象上。

C++(包括 C) 中所有的表达式和变量要么是左值，要么是右值。通俗的左值的定义就是非临时对象，那些可以在多条语句中使用的对象。所有的变量都满足这个定义，在多条代码中都可以使用，都是左值。右值是指临时的对象，它们只在当前的语句中有效。请看下列示例 :

简单的赋值语句

```
1 int i = 0;
```

在这条语句中，i 是左值，0 是临时值，就是右值。在下面的代码中，i 可以被引用，0 就不可以了。立即数都是右值。

右值也可以出现在赋值表达式的左边，但是不能作为赋值的对象，因为右值只在当前语句有效，赋值没有意义。

```
((i>0) ? i : j) = 1;
```

在这个例子中，0 作为右值出现在了 = 的左边。但是赋值对象是 i 或者 j，都是左值。

在 C++11 之前，右值是不能被引用的，最大限度就是用常量引用绑定一个右值，如：

```
const int &a = 1;
```

在这种情况下，右值不能被修改的。但是实际上右值是可以被修改的，如：

```
T().set().get();
```

T 是一个类，set 是一个函数为 T 中的一个变量赋值，get 用来取出这个变量的值。在这句中，T() 生成一个临时对象，就是右值，set() 修改了变量的值，也就修改了这个右值。

既然右值可以被修改，那么就可以实现右值引用。右值引用能够方便地解决实际工程中的问题，实现非常有吸引力的解决方案。

2.2 语法符号

左值的声明符号为`&`， 为了和左值区分，右值的声明符号为`&&`。

示例1：

```
void process_value(int& i) {
 std::cout << "LValue processed: " << i << std::endl;
}

void process_value(int&& i) {
 std::cout << "RValue processed: " << i << std::endl;
}

int main() {
 int a = 0;
 process_value(a);
 process_value(1);
}
```

运行结果：

```
LValue processed: 0
RValue processed: 1
```

Process_value 函数被重载，分别接受左值和右值。由输出结果可以看出，临时对象是作为右值处理的。

但是如果临时对象通过一个接受右值的函数传递给另一个函数时，就会变成左值，因为这个临时对象在传递过程中，变成了命名对象。

示例2：

```
void process_value(int& i) {
```

```
 std::cout << "LValue processed: " << i << std::endl;
}

void process_value(int&& i) {
 std::cout << "RValue processed: " << i << std::endl;
}

void forward_value(int&& i) {
 process_value(i);
}

int main() {
 int a = 0;
 process_value(a);
 process_value(1);
 forward_value(2);
}
```

运行结果：

```
LValue processed: 0
RValue processed: 1
LValue processed: 2
```

虽然 2 这个立即数在函数 forward_value 接收时是右值，但到了 process_value 接收时，变成了左值。

3.转移语义

3.1 定义

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、拷贝以及销毁，能够大幅度提高 C++ 应用程序的性能。临时对象的维护 (创建和销毁) 对性能有严重影响。

转移语义是和**拷贝语义**相对的，可以类比文件的剪切与拷贝，当我们将文件从一个目录拷贝到另一个目录时，速度比剪切慢很多。

通过转移语义，临时对象中的资源能够转移其它的对象里。

在现有的 C++ 机制中，我们可以定义拷贝构造函数和赋值函数。要实现转移语义，需要定义转移构造函数，还可以定义转移赋值操作符。对于右值的拷贝和赋值会调用转移构造函数和转移赋值操作符。如果转移构造函数和转移拷贝操作符没有定义，那么就遵循现有的机制，拷贝构造函数和赋值操作符会被调用。

普通的函数和操作符也可以利用右值引用操作符实现转移语义。

3.2 转移构造函数和转移赋值函数

对于拷贝语义，C++中一些类必须显示定义拷贝构造函数和拷贝赋值函数，同样的，对于转移语义，一个类同样需要定义转移构造函数和转移赋值函数。

以一个简单的 string 类为示例，实现拷贝构造函数和拷贝赋值操作符。

示例类：

```
class MyString {

private:
    char* _data;
    size_t _len;
    void _init_data(const char *s) {
        _data = new char[_len+1];
        memcpy(_data, s, _len);
        _data[_len] = '\0';
    }

public:
    // 默认构造函数
    MyString() {
        _data = nullptr;
        _len = 0;
    }

    // 构造函数
    MyString(const char* p) {
        _len = strlen(p);
        _init_data(p);
    }

    // 拷贝构造函数
    MyString(const MyString& str) {
        _len = str._len;
        _init_data = (str._data);
        std::cout << "Copy Constructor is called! source: " << str._data <<
std::endl;
    }

    // 拷贝赋值函数
    MyString& operator=(const MyString& str) {
        if (this != &str) {
            _len = str._len;
            _init_data(str._data);
        }
        std::cout << "Copy Assignment is called! source: " << str._data <<
std::endl;
        return *this;
    }

    // 析构函数
    virtual ~MyString() {
        if (_data)
            free(_data)
    }
};


int main() {
 MyString a;
 a = MyString("Hello");
 std::vector<MyString> vec;
 vec.push_back(MyString("World"));
}
```

运行结果：

```
Copy Assignment is called! source: Hello
Copy Constructor is called! source: World
```

这个 string 类已经基本满足我们演示的需要。在 main 函数中，实现了调用拷贝构造函数的操作和拷贝赋值操作符的操作。MyString(“Hello”) 和 MyString(“World”) 都是临时对象，也就是右值。虽然它们是临时的，但程序仍然调用了拷贝构造和拷贝赋值，造成了没有意义的资源申请和释放的操作。如果能够直接使用临时对象已经申请的资源，既能节省资源，有能节省资源申请和释放的时间。这正是定义转移语义的目的。

我们先定义转移构造函数：

```
MyString(MyString&& str) {
   std::cout << "Move Constructor is called! source: " << str._data <<
std::endl;
   _len = str._len;
   _data = str._data;

   // 修改资源链接和标记
   str._len = 0;
   str._data = NULL;
}
```

转移构造函数定义和拷贝构造函数类似，有几点需要注意：

- **参数（右值）的符号必须是右值引用符号，即 `&&`。**
- **参数（右值）不可以是常量，因为我们需要修改右值。**
- **参数（右值）的资源链接和标记必须修改。否则，右值的析构函数就会释放资源。转移到新对象的资源也就无效了。**

现在我们定义转移赋值操作符：

```
MyString& operator=(MyString&& str) {
   std::cout << "Move Assignment is called! source: " << str._data <<
std::endl;
   if (this != &str) {
     _len = str._len;
     _data = str._data;
     str._len = 0;
     str._data = NULL;
   }
   return *this;
}
```

这里需要注意的问题和转移构造函数是一样的。

增加了转移构造函数和转移复制操作符后，我们的程序运行结果为：

```
Move Assignment is called! source: Hello
Move Constructor is called! source: World
```

由此看出，编译器区分了左值和右值，**对右值调用了转移构造函数和转移赋值操作符。节省了资源，提高了程序运行的效率。**

有了右值引用和转移语义，我们在设计和实现类时，对于需要动态申请大量资源的类，应该设计转移构造函数和转移赋值函数，以提高应用程序的效率。**这个非常重要，对于一些类对象的函数传值，返回等等操作，都会因为右值引用而大大提高效率，更重要的是，这种操作是编译器自动优化的，我们只需要定义好转移语义的函数，编译器会自动调用转移语义的函数来提高效率。**

3.3 std::move

既然**编译器只对右值引用才能调用转移构造函数和转移赋值函数**，而**所有命名对象都只能是左值引用**，如果已知一个命名对象不再被使用而想对它调用转移构造函数和转移赋值函数，也就是**把一个左值引用当做右值引用来使用**，怎么做呢？标准库提供了函数 std::move，这个函数以非常简单的方式将左值引用转换为右值引用。

示例程序：

```
void ProcessValue(int& i) {
 std::cout << "LValue processed: " << i << std::endl;
}

void ProcessValue(int&& i) {
 std::cout << "RValue processed: " << i << std::endl;
}

int main() {
 int a = 0;
 ProcessValue(a);
 ProcessValue(std::move(a));
}
```

运行结果：

```
LValue processed: 0
RValue processed: 0
```

std::move在提高 swap 函数的的性能上非常有帮助，一般来说，swap函数的通用定义如下：

```
template <class T> swap(T& a, T& b)  {
       T tmp(a);   // copy a to tmp
       a = b;      // copy b to a
       b = tmp;    // copy tmp to b
}
```

有了 std::move，swap 函数的定义变为 :

```
template <class T> swap(T& a, T& b) {
       T tmp(std::move(a)); // move a to tmp
       a = std::move(b);    // move b to a
       b = std::move(tmp);  // move tmp to b
}
```

通过 std::move，一个简单的 swap 函数就避免了 3 次不必要的拷贝操作。

4 完美转发(Perfect Forwarding)

回忆一下，前文一个例子里，右值引用传递到函数内部就变成左值了，这就是**引用塌缩**，因为一个引用本身是命名对象，所以是左值。在传统 C++ 中，我们不能够对一个引用类型继续进行引用，但 C++ 由于右值引用的出现而放宽了这一做法，从而产生了引用坍缩规则，允许我们对引用进行引用，既能左引用，又能右引用。但是却遵循如下规则：

函数形参类型	传入实参类型	推导后的实际函数参数类型
T&	左引用	T&
T&	右引用	T&
T&&	左引用	T&
T&&	右引用	T&&

完美转发就是基于上述规律产生的。所谓完美转发，就是为了让我们在传递参数的时候，保持原来的参数类型（左引用保持左引用，右引用保持右引用）。为了解决这个问题，我们应该使用 std::forward 来进行参数的转发（传递）：

```
#include <iostream>
#include <utility>
void reference(int& v) {
    std::cout << "左值引用" << std::endl;
}
void reference(int&& v) {
    std::cout << "右值引用" << std::endl;
}
template <typename T>
void pass(T&& v) {
    std::cout << "普通传参:";
    reference(v);
    std::cout << "std::move 传参:";
    reference(std::move(v));
    std::cout << "std::forward 传参:";
    reference(std::forward<T>(v));

}
int main() {
    std::cout << "传递右值:" << std::endl;
    pass(1);

    std::cout << "传递左值:" << std::endl;
    int v = 1;
    pass(v);

    return 0;
}
传递右值:
普通传参:左值引用
std::move 传参:右值引用
std::forward 传参:右值引用
传递左值:
普通传参:左值引用
std::move 传参:右值引用
```

```
36 std::forward 传参:左值引用
```

无论传递参数为左值还是右值，普通传参都会将参数作为左值进行转发，所以 std::move 总会接受到一个左值，从而转发调用了reference(int&&) 输出右值引用。

唯独 std::forward 即没有造成任何多余的拷贝，同时完美转发(传递)了函数的实参给了内部调用的其他函数。

实际上， **std::forward 和 std::move 一样，没有做任何事情，std::move 单纯的将左值转化为右值，std::forward 也只是单纯的将参数做了一个类型的转换，从是实现来看，std::forward(v) 和 static_cast(v) 是完全一样的。**

5.总结

右值引用带来的一系列特性是C++11最重要的特性之一，熟悉这些特性的性能优化和原理实现，可以很方便的帮助我们更好的控制C++的内存资源和对象。

C++ 中多态是怎么实现的

简单 参考1 参考2

多态是面向对象程序设计语言中数据抽象和继承之外的第三个基本特征。

多态性(polymorphism)提供接口与具体实现之间的另一层隔离，从而将"what"和"how"分离开来。多态性改善了代码的可读性和组织性，同时也使创建的程序具有可扩展性，项目不仅在最初创建时期可以扩展，而且当项目在需要有新的功能时也能扩展。

c++支持编译时多态(静态多态)和运行时多态(动态多态)，运算符重载和函数重载就是编译时多态，而派生类和虚函数实现运行时多态。

静态多态和动态多态的区别就是函数地址是早绑定(静态联编)还是晚绑定(动态联编)。如果函数的调用，在编译阶段就可以确定函数的调用地址，并产生代码，就是静态多态(编译时多态)，就是说地址是早绑定的。而如果函数的调用地址不能编译不能在编译期间确定，而需要在运行时才能决定，这这就属于晚绑定(动态多态,运行时多态)。

```
1  //计算器
2  class Caculator{
3  public:
4      void setA(int a){
5          this->mA = a;
6      }
7      void setB(int b){
8          this->mB = b;
9      }
10     void setOperator(string oper){
11         this->mOperator = oper;
12     }
13     int getResult(){
14
15         if (this->mOperator == "+"){
16             return mA + mB;
17         }
18         else if (this->mOperator == "-"){
19             return mA - mB;
20         }
21         else if (this->mOperator == "*"){
22             return mA * mB;
```

```
        }
        else if (this->mOperator == "/"){
            return mA / mB;
        }
    }
private:
    int mA;
    int mB;
    string mOperator;
};

//这种程序不利于扩展，维护困难，如果修改功能或者扩展功能需要在源代码基础上修改
//面向对象程序设计一个基本原则:开闭原则(对修改关闭，对扩展开放)

//抽象基类
class AbstractCaculator{
public:
    void setA(int a){
        this->mA = a;
    }
    virtual void setB(int b){
        this->mB = b;
    }
    virtual int getResult() = 0;
protected:
    int mA;
    int mB;
};

//加法计算器
class PlusCaculator : public AbstractCaculator{
public:
    virtual int getResult(){
        return mA + mB;
    }
};

//减法计算器
class MinusCaculator : public AbstractCaculator{
public:
    virtual int getResult(){
        return mA - mB;
    }
};

//乘法计算器
class MultipliesCaculator : public AbstractCaculator{
public:
    virtual int getResult(){
        return mA * mB;
    }
};

void DoBussiness(AbstractCaculator* caculator){
    int a = 10;
    int b = 20;
    caculator->setA(a);
    caculator->setB(b);
```

```
81      cout << "计算结果：" << caculator->getResult() << endl;
82      delete caculator;
83  }
```

C++ 11 有什么新特性

简单 参考1 参考2

简述 C++ 中智能指针的特点，简述 new 与 malloc 的区别

中等 参考1 参考2

- auto_ptr: 当进行赋值时, 会将旧指针的所有权转让, 使得 **对于特定的对象, 只能有一个智能指针可以拥有它**.
- unique_ptr: 当进行赋值时, 会将旧指针的所有权转让, 使得 **对于特定的对象, 只能有一个智能指针可以拥有它**. unique_ptr 相比于 auto_ptr 会执行更加严格的所有权转让策略
- shared_ptr: 通过引用计数(reference counting), 跟踪引用特定特定对象的智能指针数. 当发生赋值操作时, 计数增1, 当指针过期时, 计数减1. 仅当最后一个指针过期时, 才调用 delete.
- weak_ptr: 不控制对象声明周期的智能指针, 它指向一个 shared_ptr 管理的对象, 而进行内存管理的只有 shared_ptr. weak_ptr 主要用来帮助解决循环引用问题, 它的构造和析构不会引起引用计数的增加或者减少. weak_ptr 一般都是配合 shared_ptr 使用, 通常不会单独使用.

1. 申请的内存所在位置

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。

那么自由存储区是否能够是堆（问题等价于new是否能在堆上动态分配内存），这取决于operator new的实现细节。自由存储区不仅可以是堆，还可以是静态存储区，这都看operator new在哪里为对象分配内存。

特别的，new甚至可以不为对象分配内存！**定位new**的功能可以办到这一点：

```
new` `(place_address) type
```

place_address为一个指针，代表一块内存的地址。当使用上面这种仅以一个地址调用new操作符时，new操作符调用特殊的operator new，也就是下面这个版本：

```
void` `* operator ``new` `(``size_t``,``void` `*) ``//不允许重定义这个版本的
operator new
```

这个operator new**不分配任何的内存**，它只是简单地返回指针实参，然后右new表达式负责在place_address指定的地址进行对象的初始化工作。

2.返回类型安全性

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。 类型安全很大程度上可以等价于内存安全，类型安全的代码不会试图方法自己没被授权的内存区域。关于C++的类型安全性可说的又有很多了。

3.内存分配失败时的返回值

new内存分配失败时，会抛出bac_alloc异常，它**不会返回NULL**；malloc分配内存失败时返回NULL。在使用C语言时，我们习惯在malloc分配内存后判断分配是否成功：

4.是否需要指定内存大小

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。

5.是否调用构造函数/析构函数

使用new操作符来分配对象内存时会经历三个步骤：

- 第一步：调用operator new 函数（对于数组是operator new[]）分配一块足够大的，**原始**的，未命名的内存空间以便存储特定类型的对象。
- 第二步：编译器运行相应的**构造函数**以构造对象，并为其传入初值。
- 第三部：对象构造完成后，返回一个指向该对象的指针。

使用delete操作符来释放对象内存时会经历两个步骤：

- 第一步：调用对象的析构函数。
- 第二步：编译器调用operator delete(或operator delete[])函数释放内存空间。

总之来说，new/delete会调用对象的构造函数/析构函数以完成对象的构造/析构。而malloc则不会。如果你不嫌啰嗦可以看下我的例子：

6.对数组的处理

C++提供了new[]与delete[]来专门处理数组类型:

```
A * ptr = ``new` `A[10];``//分配10个A对象
```

使用new[]分配的内存必须使用delete[]进行释放：

```
delete` `[] ptr;
```

new对数组的支持体现在它会分别调用构造函数函数初始化每一个数组元素，释放对象时为每个对象调用析构函数。注意delete[]要与new[]配套使用，不然会找出数组对象部分释放的现象，造成内存泄漏。

至于malloc，它并知道你在这块内存上要放的数组还是啥别的东西，反正它就给你一块原始的内存，在给你个内存的地址就完事。所以如果要动态分配一个数组的内存，还需要我们手动自定数组的大小：

```
int` `* ptr = (``int` `*) ``malloc``( ``sizeof``(``int``) );``//分配一个10个int元素的数组
```

7.new与malloc是否可以相互调用

operator new /operator delete的实现可以基于malloc，而malloc的实现不可以去调用new。

8.是否可以被重载

opeartor new /operator delete可以被重载。标准库是定义了operator new函数和operator delete函数的8个重载版本：

9. 能够直观地重新分配内存

使用malloc分配的内存后，如果在使用过程中发现内存不足，可以使用realloc函数进行内存重新分配实现内存的扩充。realloc先判断当前的指针所指内存是否有足够的连续空间，如果有，原地扩大可分配的内存地址，并且返回原来的地址指针；如果空间不够，先按照新指定的大小分配空间，将原有数据从头到尾拷贝到新分配的内存区域，而后释放原来的内存区域。

new没有这样直观的配套设施来扩充内存。

10. 客户处理内存分配不足

在operator new抛出异常以反映一个未获得满足的需求之前，它会先调用一个用户指定的错误处理函数，这就是**new-handler**。new_handler是一个指针类型：

特征	**new/delete**	**malloc/free**
分配内存的位置	自由存储区	堆
内存分配失败返回值	完整类型指针	void*
内存分配失败返回值	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给你的就好像一块原始的土地，你要种什么需要自己在土地上来播种

而new帮你划好了田地的分块（数组），帮你播了种（构造函数），还提供其他的设施给你使用:

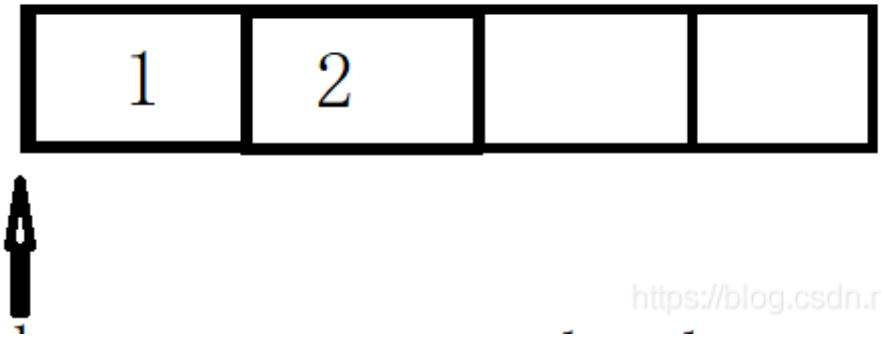
当然，malloc并不是说比不上new，它们各自有适用的地方。在C++这种偏重OOP的语言，使用new/delete自然是更合适的。

STL 中 vector 与 list 具体是怎么实现的？常见操作的时间复杂度是多少？

参考1 参考2

一、vector

是动态数组，在堆中分配内存，元素连续存放，有保留内存，如果减少大小后，内存也不会释放；如果新值大于当前大小时才会重新分配内存。



扩容方式： 1、倍数开辟二倍的内存 2、旧的数据开辟到新内存 3、释放旧的内存 4、指向新内存

时间复杂度：

插入	删除	访问
push_back O(1)	pop_back O(1)	O(1)
insert O(n)	erase O(n)	

特点：

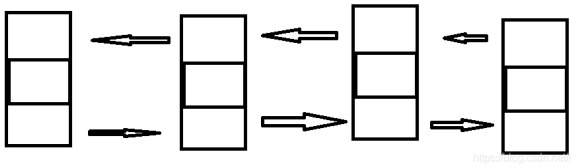
拥有一段连续的内存空间，并且起始地址不变，因此能够非常好的支持随机存取，即[]操作符，但是由于它的内存空间是连续的，所以在头部和中间进行插入和删除操作会造成内存块的拷贝，另外，当该数组的内存空间不够时，需要重新申请一块足够大得内存并且进行内存的拷贝，这些都大大的影响了vector的效率。 对头部和中间进行添加删除元素操作需要移动内存，如果你得元素是结构或类，那么移动的同时还会进行构造和析构操作，所以性能不高 对任何元素的访问时间都是O(1)，所以常用来保存需要经常进行随机访问的内容，并且不需要经常对中间元素进行添加删除操作 属性与string差不多，同样可以使用capacity看当前保留的内存，使用swap来减少它使用的内存，如push_back 1000个元素，capacity返回值为16384 对最后元素操作最快（在后面添加删除元素最快），此时一般不需要移动内存，只有保留内存不够时才需要

二、list（双向循环链表）

元素存放在堆中，每个元素都是放在一块内存中，他的内存空间可以是不连续的，通过指针来进行数据的访问，这个特点使得它的随机存取变得非常没有效率，因此它没有提供[]操作符的重载。但是由于链表的特点，它可以很有效率的支持任意地方的删除和插入操作。

优点：可以在任意位置进行插入删除

缺点：访问效率低



时间复杂度：

插入	删除	访问
push_front O(1)	pop_front O(1)	O(n)
push_back O(1)	pop_back O(1)	
insert O(1)	erase O(1)	

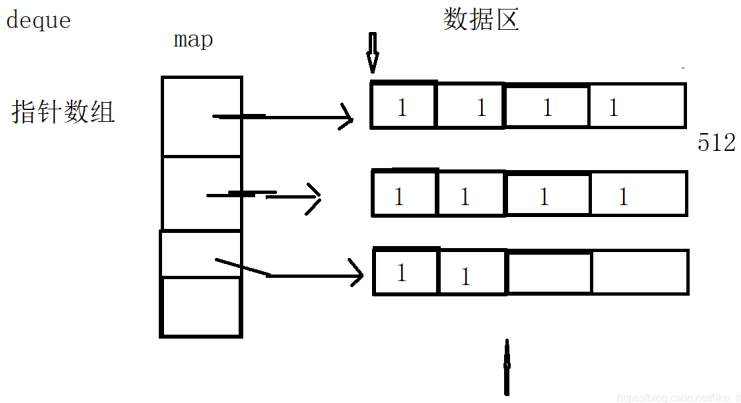
特点：

没有空间预留习惯，所以每分配一个元素都会从内存中分配，每删除一个元素都会释放它占用的内存；在哪里添加删除元素性能都很高，不需要移动内存，当然也不需要对每个元素都进行构造与析构了，所以常用来做随机插入和删除操作容器； 访问开始和最后两个元素最快，其他元素的访问时间都是O(n)；如果经常进行添加和删除操作并且不经常随机访问的话，使用list。

三、deque(双端队列)

是一种优化了的、对序列两端元素进行添加和删除操作的基本序列容器。它允许较为快速地随机访问，但它不像vector 把所有的对象保存在一块连续的内存块，而是采用多个连续的存储块，并且在一个映射结构中保存对这些块及其顺序的跟踪。向deque 两端添加或删除元素的开销很小。它不需要重新分配空间，所以向末端增加元素比vector 更有效。

deque 是对vector 和list 优缺点的结合，它是处于两者之间的一种容器。



时间复杂度：

插入	删除	访问
push_front O(1)	pop_front O(1)	O(1)
push_back O(1)	pop_back O(1)	
insert O(n)	erase O(n)	

特点：

随机访问方便； 可以在内部进行插入和删除操作； 可以在两端进行push和pop。

简述 vector 的实现原理

C++ 中虚函数与纯虚函数的区别

编译时链接有几种方式？静态链接和动态链接的区别是什么？

const、static 关键字有什么区别

类默认的构造函数是什么？

内存中堆与栈的区别是什么？

简述 STL 中的 map 的实现原理

什么是字节对齐，为什么要采用这种机制？

简述 C++ 中内存对齐的使用场景

只定义析构函数，会自动生成哪些构造函数?

变量的声明和定义有什么区别?

C/C++内存存储区有哪几种类型?

C++ 的重载和重写是如何实现的?

C++的重载（overload）与重写（override）

成员函数被重载的特征： （1）相同的范围（在同一个类中）； （2）函数名字相同； （3）参数不同； （4）virtual关键字可有可无。

重写是指派生类函数重写基类函数，是C++的多态的表现，特征是： （1）不同的范围（分别位于派生类与基类）； （2）函数名字相同； （3）参数相同； （4）基类函数必须有virtual关键字。

示例中，函数Base::f(int)与Base::f(float)相互重载，而Base::g(void)被Derived::g(void)重写。

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

class Base
{
public:
    void f(int x){ cout << "Base::f(int) " << x << endl; }
    void f(float x){ cout << "Base::f(float) " << x << endl; }
    virtual void g(void){ cout << "Base::g(void)" << endl;}
};

class Derived : public Base
{
public:
    virtual void g(void){ cout << "Derived::g(void)" << endl;}
};

int main()
{
    Derived  d;
    Base *pb = &d;
    pb->f(42);         // Base::f(int) 42
    pb->f(3.14f);      // Base::f(float) 3.14
    pb->g();           // Derived::g(void)

    return 0;
}
```

令人迷惑的隐藏规则

本来仅仅区别重载与重写并不算困难，但是C++的隐藏规则（遮蔽现象）使问题复杂性陡然增加。这里“隐藏”是指派生类的函数屏蔽了与其同名的基类函数，规则如下： （1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无virtual关键字，基类的函数将被隐藏。 （2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual关键字。此时，基类的函数被隐藏。 这种隐藏规则，不仅仅是表现在对成员函数上，对同名的data member也是如此。

示例程序中： （1）函数Derived::f(float)重写了Base::f(float)。 （2）函数Derived::g(int)隐藏了Base::g(float)。 （3）函数Derived::h(float)隐藏了Base::h(float)。

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

class Base
{
public:
    virtual void f(float x){ cout << "Base::f(float) " << x << endl; }
    virtual void g(float x){ cout << "Base::g(float) " << x << endl; }
    void h(float x){ cout << "Base::h(float) " << x << endl; }
};

class Derived : public Base
{
public:
    virtual void f(float x){ cout << "Derived::f(float) " << x << endl; }
    virtual void g(int x){ cout << "Derived::g(int) " << x << endl; }
    void h(float x){ cout << "Derived::h(float) " << x << endl; }
};

int main()
{
    Derived  d;
    Base *pb = &d;
    Derived *pd = &d;

    // Good : behavior depends solely on type of the object
    pb->f(3.14f); // Derived::f(float) 3.14
    pd->f(3.14f); // Derived::f(float) 3.14

    // Bad : behavior depends on type of the pointer
    pb->g(3.14f); // Base::g(float) 3.14 (surprise!)
    pd->g(3.14f); // Derived::g(int) 3

    // Bad : behavior depends on type of the pointer
    pb->h(3.14f); // Base::h(float) 3.14  (surprise!)
    pd->h(3.14f); // Derived::h(float) 3.14

    return 0;
}
```

另一个关于虚函数很微妙的错误情况：参数相同，但是基类的函数是const的，派生类的函数却不是。

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

class Base
{
public:
    virtual void f(float x){ cout << "Base::f(float) " << x << endl; }
};

class Derived : public Base
{
public:
    virtual void f(float x) const { cout << "Derived::f(float) " << x <<
endl; }
};

int main()
{
    Derived  d;
    Base *pb = &d;
    Derived *pd = &d;

    // Bad : behavior depends solely on type of the object
    pb->f(3.14f); // Base::f(float) 3.14
    pd->f(3.14f); // Derived::f(float) 3.14

    return 0;
}
```

（1）一个函数在基类申明一个virtual，那么在所有的派生类都是是virtual的。 （2）一个函数在基类为普通函数，在派生类定义为virtual的函数称为越位，函数行为依赖于指针/引用的类型，而不是实际对象的类型。

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

class Base
{
public:
    void f(){ cout << "Base::f() " << endl; }
    virtual void g(){ cout << "Base::g() " << endl; }
};

class Derived : public Base
{
public:
    virtual void f(){ cout << "Derived::f() " << endl; }
    void g(){ cout << "Derived::g() " << endl; }
};
```

```
17
18 class VirtualDerived : virtual public Base
19 {
20 public:
21     void f(){ cout << "VirtualDerived::f() " << endl; }
22     void g(){ cout << "VirtualDerived::g() " << endl; }
23 };
24
25 int main()
26 {
27     Base *d = new Derived;
28     Base *vd = new VirtualDerived;
29
30     d->f(); // Base::f() Bad behavior
31     d->g(); // Derived::g()
32
33     vd->f(); // Base::f() Bad behavior
34     vd->g(); // VirtualDerived::g()
35
36     delete d;
37     delete vd;
38
39     return 0;
40 }
```

《Effective C++》条款: 决不要重新定义继承而来的非虚函数。说明了不能重新定义继承而来的非虚函数的理论依据是什么。 以下摘自《Effective C++》： 公有继承的含义是 "是一个"，"在一个类中声明一个非虚函数实际上为这个类建立了一种特殊性上的不变性"。如果将这些分析套用到类B、类D和非虚成员函数B::mf，那么:

（1）适用于B对象的一切也适用于D对象，因为每个D的对象 "是一个" B的对象。 （2）B的子类必须同时继承mf的接口和实现，因为mf在B中是非虚函数。

那么，如果D重新定义了mf，设计中就会产生矛盾。如果D真的需要实现和B不同的mf，而且每个B的对象（无论怎么特殊）也真的要使用B实现的mf，那么每个D将不 "是一个" B。这种情况下，D不能从B公有继承。相反，如果D真的必须从B公有继承，而且D真的需要和B不同的mf的实现，那么，mf就没有为B反映出特殊性上的不变性。这种情况下，mf应该是虚函数。最后，如果每个D真的 "是一个" B，并且如果mf真的为B建立了特殊性上的不变性，那么，D实际上就不需要重新定义mf，也就决不能这样做。

不管采用上面的哪一种论据都可以得出这样的结论：任何条件下都要禁止重新定义继承而来的非虚函数。

一个类里面： 重载（函数名相同，函数参数的个数和类型不同）

父类和子类之间：重写（函数原型完全相同）→ 有virtual关键字声明：多态

父类和子类之间：重写（函数原型完全相同）→ 无virtual关键字声明：重定义

、

简述 C++ 从代码到可执行二进制文件的过程

预处理-->编译-->汇编-->链接

1.一步完成从代码到可执行程序：

对**c**程序来说使用

gcc name.c -o name.exe

执行命令后会生成可执行文件 name.exe。

对**c++**程序来使用

g++ name.cpp -o name.exe

执行命令后生成可执行文件name.exe。

gcc和g++的区别：

对C程序来说，gcc使用c代码的方式编译 ， 而g++则使用C++代码的方式编译。注意：使用C++的方式编译C代码可能会出错。

对C++程序来说，两者没有差别。

2.四个阶段完成从代码到可执行程序：

源文件b.cpp代码如下：

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

/*
    这是一个注释!
*/
int main()
{
    cout<<"c++ program!"<<endl;
    system("pause");
    return 0;
}
```

1、预处理

g++ -E(大写) name.cpp

-E 选项用来对原代码做预处理操作。使用 g++ -E b.cpp命令后，只会将预处理指操作的结果输出到屏幕，若想保存操作结果则要搭配-o使用。

预处理主要处理源文件和头文件中以#开头的命令（#等），并删除程序中的注释信息等。

运行 ***g++ -E b.cpp -o b.i*** 会生成一个b.i文件

打开b.i文件后可以看到一些信息，发现其中注释已经被删掉了，头文件被替换为好多行代码。

```
# 1 "b.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "b.cpp"
# 1 "C:/mingw64/lib/gcc/x86_64-w64-mingw32/8.1.0/include/c++/iostream" 1 3
# 36 "C:/mingw64/lib/gcc/x86_64-w64-mingw32/8.1.0/include/c++/iostream" 3

# 37 "C:/mingw64/lib/gcc/x86_64-w64-mingw32/8.1.0/include/c++/iostream" 3

# 1 "C:/mingw64/lib/gcc/x86_64-w64-mingw32/8.1.0/include/c++/x86_64-w64-mingw32/bits/c++config.h" 1 3
# 236 "C:/mingw64/lib/gcc/x86_64-w64-mingw32/8.1.0/include/c++/x86_64-w64-mingw32/bits/c++config.h" 3

# 236 "C:/mingw64/lib/gcc/x86_64-w64-mingw32/8.1.0/include/c++/x86_64-w64-mingw32/bits/c++config.h" 3
namespace std
{
  typedef long long unsigned int size_t;
  typedef long long int ptrdiff_t;
  typedef decltype(nullptr) nullptr_t;
            .....
            .....
            .....
            .....
            .....
  extern istream cin;
  extern ostream cout;
  extern ostream cerr;
  extern ostream clog;
  extern wistream wcin;
  extern wostream wcout;
  extern wostream wcerr;
  extern wostream wclog;
  static ios_base::Init __ioinit;

}
# 2 "b.cpp" 2
# 3 "b.cpp"
using namespace std;

int main()
{
 cout<<"c++ program!"<<endl;
 system("pause");
 return 0;
}
```

2、编译

g++ -S(大写) name.cpp / name.i

使用该g++ -S b.cpp 后会自动生成b.s汇编代码文件，**将c++代码翻译为汇编代码。**

g++ -S指令并非必须经过预处理后的.i文件，-S选项的功能是令GCC编译器将指定文件预处理至编译阶段结束。

也就是说g++ -S可以处理.i文件，也可以处理原代码文件。

```
    .file    "b.cpp"
    .text
    .section .rdata,"dr"
_ZStL19piecewise_construct:
    .space 1
.lcomm _ZStL8__ioinit,1,1
    .def    __main;    .scl    2;    .type    32;    .endef
.LC0:
    .ascii "c++ program!\0"
.LC1:
    .ascii "pause\0"
    .text
    .globl    main
    .def    main;    .scl    2;    .type    32;    .endef
    .seh_proc    main
main:
.LFB1573:
    pushq    %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe    %rbp, 0
    subq    $32, %rsp
    .seh_stackalloc    32
    .seh_endprologue
    call    __main
    leaq    .LC0(%rip), %rdx
    movq    .refptr._ZSt4cout(%rip), %rcx
    call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
    movq
.refptr._ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_(%rip),
%rdx
    movq    %rax, %rcx
    call    _ZNSolsEPFRSoS_E
    leaq    .LC1(%rip), %rcx
    call    system
    movl    $0, %eax
    addq    $32, %rsp
    popq    %rbp
    ret
    .seh_endproc
    .def    __tcf_0;    .scl    3;    .type    32;    .endef
    .seh_proc    __tcf_0
__tcf_0:
.LFB2063:
    pushq    %rbp
```

```
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe    %rbp, 0
    subq    $32, %rsp
    .seh_stackalloc    32
    .seh_endprologue
    leaq    _ZStL8__ioinit(%rip), %rcx
    call    _ZNSt8ios_base4InitD1Ev
    nop
    addq    $32, %rsp
    popq    %rbp
    ret
    .seh_endproc
    .def    _Z41__static_initialization_and_destruction_0ii;    .scl    3;
   .type    32;    .endef
    .seh_proc    _Z41__static_initialization_and_destruction_0ii
_Z41__static_initialization_and_destruction_0ii:
.LFB2062:
    pushq    %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe    %rbp, 0
    subq    $32, %rsp
    .seh_stackalloc    32
    .seh_endprologue
    movl    %ecx, 16(%rbp)
    movl    %edx, 24(%rbp)
    cmpl    $1, 16(%rbp)
    jne    .L6
    cmpl    $65535, 24(%rbp)
    jne    .L6
    leaq    _ZStL8__ioinit(%rip), %rcx
    call    _ZNSt8ios_base4InitC1Ev
    leaq    __tcf_0(%rip), %rcx
    call    atexit
.L6:
    nop
    addq    $32, %rsp
    popq    %rbp
    ret
    .seh_endproc
    .def    _GLOBAL__sub_I_main;    .scl    3;    .type    32;    .endef
    .seh_proc    _GLOBAL__sub_I_main
_GLOBAL__sub_I_main:
.LFB2064:
    pushq    %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe    %rbp, 0
    subq    $32, %rsp
    .seh_stackalloc    32
    .seh_endprologue
    movl    $65535, %edx
    movl    $1, %ecx
    call    _Z41__static_initialization_and_destruction_0ii
    nop
    addq    $32, %rsp
    popq    %rbp
```

```
    ret
    .seh_endproc
    .section    .ctors,"w"
    .align 8
    .quad    _GLOBAL__sub_I_main
    .ident    "GCC: (x86_64-posix-seh-rev0, Built by MinGW-W64 project) 8.1.0"
    .def    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc;    .scl    2;    .type    32;    .endef
    .def    _ZNSolsEPFRSoS_E;    .scl    2;    .type    32;    .endef
    .def    system;    .scl    2;    .type    32;    .endef
    .def    _ZNSt8ios_base4InitD1Ev;    .scl    2;    .type    32;    .endef
    .def    _ZNSt8ios_base4InitC1Ev;    .scl    2;    .type    32;    .endef
    .def    atexit;    .scl    2;    .type    32;    .endef
    .section    .rdata$.refptr._ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_, "dr"
    .globl    .refptr._ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
    .linkonce    discard
.refptr._ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_:
    .quad    _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
    .section    .rdata$.refptr._ZSt4cout, "dr"
    .globl    .refptr._ZSt4cout
    .linkonce    discard
.refptr._ZSt4cout:
    .quad    _ZSt4cout
```

3、汇编

g++ -c name.cpp / name.s

使用g++ -c b.s对**汇编代码进行汇编**会生成相应的目标文件（b.o文件）

目标文件为二进制文件，由于尚未经过链接操作，所以无法直接运行。

4、链接

g++ b.o -o b.exe

将得到的b.o文件经过链接（一些函数和全局变量等）后便得到了可执行程序。

并不需要在g++添加任何选项，g++会在b.o的基础上完成链接操作。

Golang

简述 Goroutine 的调度流程

简述 Golang 垃圾回收的机制

Golang 是如何实现 Maps 的?

简述 defer 的执行顺序

有缓存的管道和没有缓存的管道区别是什么?

简述 slice 的底层原理，slice 和数组的区别是什么?

Channels 怎么保证线程安全?

Maps 是线程安全的吗？怎么解决它的并发安全问题?

协程与进程，线程的区别是什么？协程有什么优势?

Golang 的一个协程能保证绑定在一个内核线程上吗?

Golang 的协程可以自己主动让出 CPU 吗?

网络协议

简述 TCP 三次握手以及四次挥手的流程。为什么需要三次握手以及四次挥手?

中等 参考1 参考2

TCP 的特性

- TCP 提供一种**面向连接的、可靠的**字节流服务
- 在一个 TCP 连接中，仅有两方进行彼此通信。广播和多播不能用于 TCP
- TCP 使用校验和，确认和重传机制来保证可靠传输
- TCP 给数据分节进行排序，并使用累积确认保证数据的顺序不变和非重复
- TCP 使用滑动窗口机制来实现流量控制，通过动态改变窗口的大小进行拥塞控制

注意：TCP 并不能保证数据一定会被对方接收到，因为这是不可能的。TCP 能够做到的是，如果有可能，就把数据递送到接收方，否则就（通过放弃重传并且中断连接这一手段）通知用户。因此准确说 TCP 也不是 100% 可靠的协议，它所能提供的是数据的可靠递送或故障的可靠通知。

三次握手与四次挥手

所谓三次握手(Three-way Handshake)，是指建立一个 TCP 连接时，需要客户端和服务器总共发送3个包。

三次握手的目的是连接服务器指定端口，建立 TCP 连接，并同步连接双方的序列号和确认号，交换 TCP 窗口大小信息。在 socket 编程中，客户端执行 `connect()` 时。将触发三次握手。

- 第一次握手(SYN=1, seq=x):

  客户端发送一个 TCP 的 SYN 标志位置1的包，指明客户端打算连接的服务器的端口，以及初始序号X,保存在包头的序列号(Sequence Number)字段里。

  发送完毕后，客户端进入 `SYN_SEND` 状态。

- 第二次握手(SYN=1, ACK=1, seq=y, ACKnum=x+1):

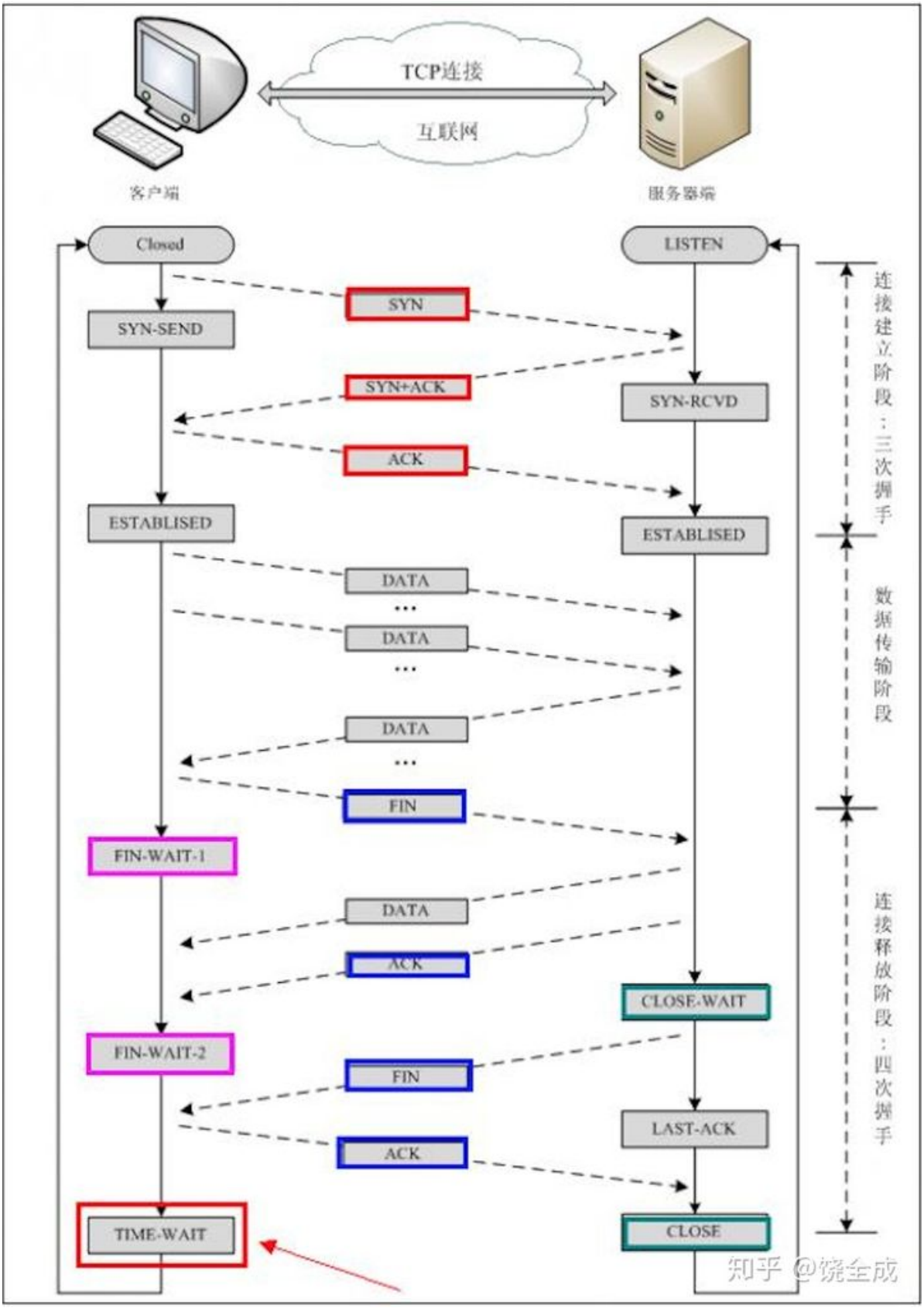
  服务器发回确认包(ACK)应答。即 SYN 标志位和 ACK 标志位均为1。服务器端选择自己 ISN 序列号，放到 Seq 域里，同时将确认序号(Acknowledgement Number)设置为客户的 ISN 加1，即 X+1。 发送完毕后，服务器端进入 `SYN_RCVD` 状态。

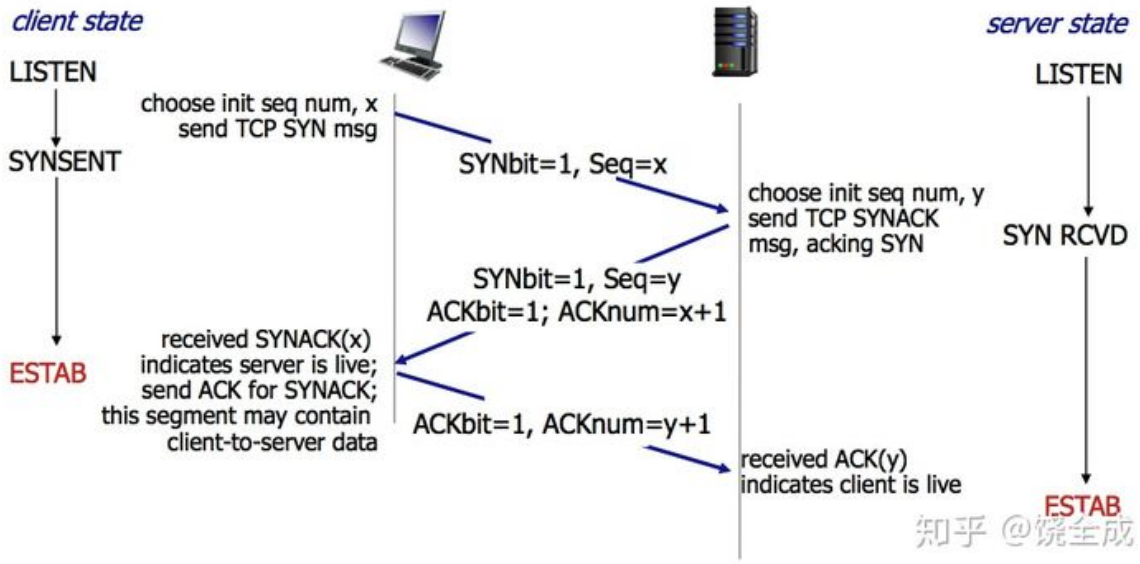
- 第三次握手(ACK=1，ACKnum=y+1)

  客户端再次发送确认包(ACK)，SYN 标志位为0，ACK 标志位为1，并且把服务器发来 ACK 的序号字段+1，放在确定字段中发送给对方，并且在数据段放写ISN的+1

  发送完毕后，客户端进入 `ESTABLISHED` 状态，当服务器端接收到这个包时，也进入 `ESTABLISHED` 状态，TCP 握手结束。

三次握手的过程的示意图如下：


TCP连接
互联网
客户端
服务器端
Closed
LISTEN
SYN
SYN-SEND
SYN+ACK
SYN-RCVD
ACK
ESTABLISED
ESTABLISED
DATA
DATA
DATA
FIN
FIN-WAIT-1
DATA
ACK
CLOSE-WAIT
FIN-WAIT-2
FIN
LAST-ACK
ACK
TIME-WAIT
CLOSE
连接建立阶段：三次握手
数据传输阶段
连接释放阶段：四次握手
知乎 @饶全成



TCP 的连接的拆除需要发送四个包，因此称为四次挥手(Four-way handshake)，也叫做改进的三次握手。客户端或服务器均可主动发起挥手动作，在 socket 编程中，任何一方执行 `close()` 操作即可产生挥手操作。

- 第一次挥手(FIN=1，seq=x)

  假设客户端想要关闭连接，客户端发送一个 FIN 标志位置为1的包，表示自己已经没有数据可以发送了，但是仍然可以接受数据。

  发送完毕后，客户端进入 `FIN_WAIT_1` 状态。

- 第二次挥手(ACK=1，ACKnum=x+1)

  服务器端确认客户端的 FIN 包，发送一个确认包，表明自己接受到了客户端关闭连接的请求，但还没有准备好关闭连接。

  发送完毕后，服务器端进入 `CLOSE_WAIT` 状态，客户端接收到这个确认包之后，进入 `FIN_WAIT_2` 状态，等待服务器端关闭连接。

- 第三次挥手(FIN=1，seq=y)

  服务器端准备好关闭连接时，向客户端发送结束连接请求，FIN 置为1。

  发送完毕后，服务器端进入 `LAST_ACK` 状态，等待来自客户端的最后一个ACK。

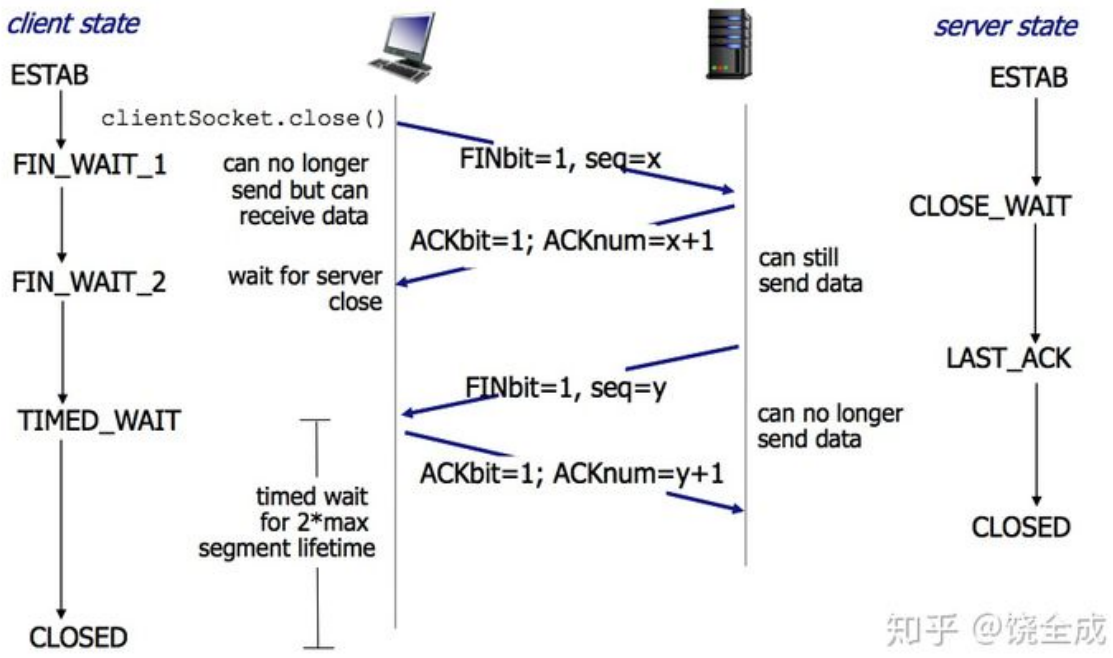
- 第四次挥手(ACK=1，ACKnum=y+1)

  客户端接收到来自服务器端的关闭请求，发送一个确认包，并进入 `TIME_WAIT` 状态，等待可能出现的要求重传的 ACK 包。

  服务器端接收到这个确认包之后，关闭连接，进入 `CLOSED` 状态。

  客户端等待了某个固定时间（两个最大段生命周期，2MSL，2 Maximum Segment Lifetime）之后，没有收到服务器端的 ACK ，认为服务器端已经正常关闭连接，于是自己也关闭连接，进入 `CLOSED` 状态。

四次挥手的示意图如下：



TCP KeepAlive

TCP 的连接，实际上是一种纯软件层面的概念，在物理层面并没有“连接”这种概念。TCP 通信双方建立交互的连接，但是并不是一直存在数据交互，有些连接会在数据交互完毕后，主动释放连接，而有些不会。在长时间无数据交互的时间段内，交互双方都有可能出现掉电、死机、异常重启等各种意外，当这些意外发生之后，这些 TCP 连接并未来得及正常释放，在软件层面上，连接的另一方并不知道对端的情况，它会一直维护这个连接，长时间的积累会导致非常多的半打开连接，造成端系统资源的消耗和浪费，为了解决这个问题，在传输层可以利用 TCP 的 KeepAlive 机制实现来实现。主流的操作系统基本都在内核里支持了这个特性。

TCP KeepAlive 的基本原理是，隔一段时间给连接对端发送一个探测包，如果收到对方回应的 ACK，则认为连接还是存活的，在超过一定重试次数之后还是没有收到对方的回应，则丢弃该 TCP 连接。

TCP-Keepalive-HOWTO 有对 TCP KeepAlive 特性的详细介绍，有兴趣的同学可以参考。这里主要说一下，TCP KeepAlive 的局限。首先 TCP KeepAlive 监测的方式是发送一个 probe 包，会给网络带来额外的流量，另外 TCP KeepAlive 只能在内核层级监测连接的存活与否，而连接的存活不一定代表服务的可用。例如当一个服务器 CPU 进程服务器占用达到 100%，已经卡死不能响应请求了，此时 TCP KeepAlive 依然会认为连接是存活的。因此 TCP KeepAlive 对于应用层程序的价值是相对较小的。需要做连接保活的应用层程序，例如 QQ，往往会在应用层实现自己的心跳功能。

RestFul 与 RPC 的区别是什么？RestFul 的优点在哪里？ 中等 参考1 参考2

HTTP 与 HTTPS 有哪些区别？ 中等 参考1 参考2

从输入 URL 到展现页面的全过程 困难 参考1

RestFul 是什么？RestFul 请求的 URL 有什么特点？ 简单 参考1 参考2 参考3

TCP 怎么保证可靠传输？ 中等 参考1 参考2

TCP 与 UDP 在网络协议中的哪一层，他们之间有什么区别？ 简单 参考1

TCP 中常见的拥塞控制算法有哪些？ 中等 参考1 参考2

从系统层面上，UDP如何保证尽量可靠？ 简单 参考1

简述 TCP 的 TIME_WAIT 和 CLOSE_WAIT 简单 参考1 参考2

简述 HTTP 1.0，1.1，2.0 的主要区别 简单 参考1 参考2

TCP 挥手时出现大量 CLOSE_WAIT 或 TIME_WAIT 怎么解决？ 简单 参考1 参考2

TCP 的 keepalive 了解吗？说一说它和 HTTP 的 keepalive 的区别？ 简单 参考1

简述 TCP 滑动窗口以及重传机制 简单 参考1

HTTP 中 GET 和 POST 区别 简单 参考1

HTTP 的方法有哪些？ 简单 参考1

什么是 TCP 粘包和拆包？ 简单

简述 HTTPS 的加密与认证过程 中等 参考1

简述常见的 HTTP 状态码的含义（301，304，401，403） 简单 参考1

简述 TCP 协议的延迟 ACK 和累计应答 简单

简述对称与非对称加密的概念 简单

简述 TCP 半连接发生场景 简单

什么是 SYN flood，如何防止这类攻击？ 简单

SYN攻击

- 什么是 SYN 攻击（SYN Flood）？

  在三次握手过程中，服务器发送 SYN-ACK 之后，收到客户端的 ACK 之前的 TCP 连接称为半连接(half-open connect)。此时服务器处于 SYN_RCVD 状态。当收到 ACK 后，服务器才能转入ESTABLISHED 状态.

  SYN 攻击指的是，攻击客户端在短时间内伪造大量不存在的IP地址，向服务器不断地发送SYN包，服务器回复确认包，并等待客户的确认。由于源地址是不存在的，服务器需要不断的重发直至超时，这些伪造的SYN包将长时间占用未连接队列，正常的SYN请求被丢弃，导致目标系统运行缓慢，严重者会引起网络堵塞甚至系统瘫痪。

  SYN 攻击是一种典型的 DoS/DDoS 攻击。

- 如何检测 SYN 攻击?

  检测 SYN 攻击非常的方便，当你在服务器上看到大量的半连接状态时，特别是源IP地址是随机的，基本上可以断定这是一次SYN攻击。在 Linux/Unix 上可以使用系统自带的 netstats 命令来检测 SYN 攻击。

- 如何防御 SYN 攻击?

  SYN攻击不能完全被阻止，除非将TCP协议重新设计。我们所做的是尽可能的减轻SYN攻击的危害，常见的防御 SYN 攻击的方法有如下几种：

  - 缩短超时（SYN Timeout）时间
  - 增加最大半连接数
  - 过滤网关防护
  - SYN cookies技术

什么是中间人攻击？如何防止攻击？ 中等

简述 HTTP 短链接与长链接的区别 简单

3.2. TCP短连接

模拟一下TCP短连接的情况：client向server发起连接请求，server接到请求，然后双方建立连接。client向server发送消息，server回应client，然后一次请求就完成了。这时候双方任意都可以发起close操作，不过一般都是client先发起close操作。上述可知，短连接一般只会在 client/server间传递一次请求操作。

短连接的优点是：管理起来比较简单，存在的连接都是有用的连接，不需要额外的控制手段。

3.3. TCP长连接

我们再模拟一下长连接的情况：client向server发起连接，server接受client连接，双方建立连接，client与server完成一次请求后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

TCP的保活功能主要为服务器应用提供。如果客户端已经消失而连接未断开，则会使得服务器上保留一个半开放的连接，而服务器又在等待来自客户端的数据，此时服务器将永远等待客户端的数据。保活功能就是试图在服务端器端检测到这种半开放的连接。

如果一个给定的连接在两小时内没有任何动作，服务器就向客户发送一个探测报文段，根据客户端主机响应探测4个客户端状态：

- 客户主机依然正常运行，且服务器可达。此时客户的TCP响应正常，服务器将保活定时器复位。
- 客户主机已经崩溃，并且关闭或者正在重新启动。上述情况下客户端都不能响应TCP。服务端将无法收到客户端对探测的响应。服务器总共发送10个这样的探测，每个间隔75秒。若服务器没有收到任何一个响应，它就认为客户端已经关闭并终止连接。
- 客户端崩溃并已经重新启动。服务器将收到一个对其保活探测的响应，这个响应是一个复位，使得服务器终止这个连接。
- 客户机正常运行，但是服务器不可达。这种情况与第二种状态类似。

4. 长连接和短连接的优点和缺点

由上可以看出，长连接可以省去较多的TCP建立和关闭的操作，减少浪费，节约时间。对于频繁请求资源的客户端适合使用长连接。在长连接的应用场景下，client端一般不会主动关闭连接，当client与server之间的连接一直不关闭，随着客户端连接越来越多，server会保持过多连接。这时候server端需要采取一些策略，如关闭一些长时间没有请求发生的连接，这样可以避免一些恶意连接导致server端服务受损；如果条件允许则可以限制每个客户端的最大长连接数，这样可以完全避免恶意的客户端拖垮整体后端服务。

短连接对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。但如果客户请求频繁，将在TCP的建立和关闭操作上浪费较多时间和带宽。

长连接和短连接的产生在于client和server采取的关闭策略。不同的应用场景适合采用不同的策略。

由上可以看出，**长连接**可以**省去较多的TCP建立和关闭的操作，减少浪费，节约时间。**对于频繁请求资源的客户来说，较适用长连接。不过这里**存在一个问题**，**存活功能的探测周期太长**，还有就是它只是探测TCP连接的存活，属于比较斯文的做法，遇到恶意的连接时，保活功能就不够使了。在长连接的应用场景下，client端一般不会主动关闭它们之间的连接，**Client与server之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越多，server早晚有扛不住的时候**，这时候server端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可 以避免一些恶意连接导致server端服务受损；如果条件再允许就可以以客户端机器为颗粒度，限制每个客户端的最大长连接数，这样可以完全避免某个蛋疼的客户端连累后端服务。

短连接对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。但如果客户**请求频繁**，将在**TCP的建立和关闭操作上浪费时间和带宽。**

长连接和短连接的产生在于client和server采取的关闭策略，具体的应用场景采用具体的策略，没有十全十美的选择，只有合适的选择。

长连接短连接操作过程

```
短连接的操作步骤是：
建立连接--数据传输--关闭连接...建立连接--数据传输--关闭连接
长连接的操作步骤是：
建立连接--数据传输...（保持连接）...数据传输--关闭连接
```

什么时候用长连接，短连接?

长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况，。每个TCP连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，次处理时直接发送数据包就OK了，不用建立TCP连接。例如：数据库的连接用长连接， 如果用短连接频繁的通信会造成socket错误，而且频繁的socket 创建也是对资源的浪费。

而像WEB网站的http服务一般都用**短链接**，因为长连接对于服务端来说会耗费一定的资源，而像WEB网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操作情况下需用短连好。

简述 TCP 的报文头部结构 简单

数据库

MySQL 为什么使用 B+ 树来作索引，对比 B 树它的优点和缺点是什么？ 中等 参考1 参考2

数据库的事务隔离级别有哪些？各有哪些优缺点？ 中等 参考1 参考2

什么是数据库事务，MySQL 为什么会使用 InnoDB 作为默认选项 中等 参考1 参考2

简述乐观锁以及悲观锁的区别以及使用场景

简单 参考1 参考2

何谓悲观锁与乐观锁

> 乐观锁对应于生活中乐观的人总是想着事情往好的方向发展，悲观锁对应于生活中悲观的人总是想着事情往坏的方向发展。这两种人各有优缺点，不能不以场景而定说一种人好于另外一种人。

悲观锁

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（**共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程**）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中`synchronized`和`ReentrantLock`等独占锁就是悲观锁思想的实现。

乐观锁

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。**乐观锁适用于多读的应用类型，这样可以提高吞吐量**，像数据库提供的类似于**write_condition机制**，其实都是提供的乐观锁。在Java中`java.util.concurrent.atomic`包下面的原子变量类就是使用了乐观锁的一种实现方式**CAS**实现的。

两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像**乐观锁适用于写比较少的情况下（多读场景）**，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以**一般多写的场景下用悲观锁就比较合适。**

乐观锁常见的两种实现方式

> **乐观锁一般会使用版本号机制或CAS算法实现。**

1. 版本号机制

一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

举一个简单的例子： 假设数据库中帐户信息表中有一个 version 字段，当前值为 1 ； 而当前帐户余额字段 （ balance ） 为 $100 。

1. 操作员 A 此时将其读出 （ version=1 ），并从其帐户余额中扣除 $50 （ $100-$50 ）。
2. 在操作员 A 操作的过程中，操作员B 也读入此用户信息 （ version=1 ），并从其帐户余额中扣除 $20 （ $100-$20 ）。
3. 操作员 A 完成了修改工作，将数据版本号加一 （ version=2 ），连同帐户扣除后余额 （ balance=$50 ），提交至数据库更新，此时由于提交数据版本大于数据库记录当前版本，数据被更新，数据库记录 version 更新为 2 。
4. 操作员 B 完成了操作，也将版本号加一 （ version=2 ）试图向数据库提交数据 （ balance=$80 ），但此时比对数据库记录版本时发现，操作员 B 提交的数据版本号为 2 ，数据库记录当前版本也为 2 ，不满足 “ 提交版本必须大于记录当前版本才能执行更新 “ 的乐观锁策略，因此，操作员 B 的提交被驳回。

这样，就避免了操作员 B 用基于 version=1 的旧数据修改的结果覆盖操作员A 的操作结果的可能。

2. CAS算法

即**compare and swap（比较与交换）**，是一种有名的**无锁算法。**无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。**CAS算法**涉及到三个操作数

- 需要读写的内存值 V
- 进行比较的值 A
- 拟写入的新值 B

当且仅当 V 的值等于 A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个**自旋操作**，即**不断的重试。**

乐观锁的缺点

> ABA 问题是乐观锁一个常见的问题

1 ABA 问题

如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 **"ABA"问题。**

JDK 1.5 以后的 `AtomicStampedReference` 类就提供了此种能力，其中的 `compareAndSet` 方法就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

2 循环时间长开销大

自旋CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。 如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

3 只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5开始，提供了 `AtomicReference`类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作.所以我们可以使用锁或者利用 `AtomicReference`类把多个共享变量合并成一个共享变量来操作。

CAS与synchronized的使用情景

> **简单的来说CAS适用于写比较少的情况下（多读场景，冲突一般较少），synchronized适用于写比较多的情况下（多写场景，冲突一般较多）**

1. 对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。
2. 对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

补充： Java并发编程这个领域中synchronized关键字一直都是元老级的角色，很久之前很多人都会称它为 **“重量级锁”** 。但是，在JavaSE 1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的 **偏向锁** 和 **轻量级锁** 以及其它**各种优化**之后变得在某些情况下并不是那么重了。synchronized的底层实现主要依靠 **Lock-Free** 的队列，基本思路是 **自旋后阻塞**，**竞争切换后继续竞争锁**，**稍微牺牲了公平性，但获得了高吞吐量。**在线程冲突较少的情况下，可以获得和CAS类似的性能；而线程冲突严重的情况下，性能远高于CAS。

什么情况下会发生死锁，如何解决死锁?

中等 参考1 参考2

Redis 有几种数据结构？Zset 是如何实现的?

中等 参考1 参考2

聚簇索引和非聚簇索引有什么区别?

简单 参考1 参考2

简述脏读和幻读的发生场景，InnoDB 是如何解决幻读的?

中等 参考1 参考2

唯一索引与普通索引的区别是什么？使用索引会有哪些优缺点?

简单 参考1 参考2

简述 Redis 持久化中 RDB 以及 AOF 方案的优缺点

困难 参考1 参考2

简述 MySQL 的间隙锁

简单 参考1

Redis 如何实现分布式锁?

困难 参考1 参考2

MySQL 有什么调优的方式?

困难 参考1 参考2

简述 Redis 中如何防止缓存雪崩和缓存击穿

中等 参考1 参考2

简述 MySQL 的主从同步机制，如果同步失败会怎么样?

中等

MySQL 的索引什么情况下会失效?

简单

什么是 SQL 注入攻击？如何防止这类攻击?

中等 参考1

简述数据库中的 ACID 分别是什么？ 简单

简述 Redis 中跳表的应用以及优缺点 中等

简述数据库中什么情况下进行分库，什么情况下进行分表?

中等

数据库如何设计索引，如何优化查询?

困难

假设Redis 的 master 节点宕机了，你会怎么进行数据恢复?

中等

假设建立联合索引 (a, b, c) 如果对字段 a 和 c 查询，会用到这个联合索引吗?

中等

MySQL 有哪些常见的存储引擎？它们的区别是什么?

中等

数据库反范式设计会出现什么问题?

中等

简述 MySQL MVCC 的实现原理 困难

操作系统

进程和线程之间有什么区别?

简单 参考1 参考2

进程间有哪些通信方式?

困难 参考1 参考2

1.无名管道(pipe): 管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

2.高级管道(popen): 将另一个程序当做一个新的进程在当前程序进程中启动，则它算是当前程序的子进程，这种方式我们成为高级管道方式。

3.有名管道 (named pipe) : 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

4.消息队列(message queue) : 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

5.信号量(semophore) : 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。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

6.信号 (sinal) : 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

7.共享内存(shared memory) : 共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

8.套接字(socket) : 套解字也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。

简述 select, poll, epoll 的使用场景以及区别，epoll 中水平触发以及边缘触发有什么不同？ epoll和poll有什么区别?

困难 参考1 参考2

(1)select==>时间复杂度O(n)

它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以**select具有O(n)的无差别轮询复杂度**，同时处理的流越多，无差别轮询时间就越长。

(2)poll==>时间复杂度O(n)

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，**但是它没有最大连接数的限制**，原因是它是基于链表来存储的.

(3)epoll==>时间复杂度O(1)

epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是**事件驱动（每个事件关联上fd）**的，此时我们对这些流的操作都是有意义的。**（复杂度降低到了O(1)）**

select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。**但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的**，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

epoll跟select都能提供多路I/O复用的解决方案。在现在的Linux内核里有都能够支持，其中epoll是Linux所特有，而select则应该是POSIX所规定，一般操作系统均有实现

select:

select本质上是通过设置或者检查存放fd标志位的数据结构来进行下一步处理。这样所带来的缺点是:

1、 单个进程可监视的fd数量被限制，即能监听端口的大小有限。

一般来说这个数目和系统内存关系很大，具体数目可以cat /proc/sys/fs/file-max察看。32位机默认是1024个。64位机默认是2048.

2、 对socket进行扫描时是线性扫描，即采用轮询的方法，效率较低:

当套接字比较多的时候，每次select()都要通过遍历FD_SETSIZE个Socket来完成调度,不管哪个Socket是活跃的,都遍历一遍。这会浪费很多CPU时间。如果能给套接字注册某个回调函数，当他们活跃时，自动完成相关操作，那就避免了轮询，这正是epoll与kqueue做的。

3、需要维护一个用来存放大量fd的数据结构，这样会使得用户空间和内核空间在传递该结构时复制开销大

poll:

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。这个过程经历了多次无谓的遍历。

它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点:

1、大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。

2、poll还有一个特点是“水平触发”，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。

epoll:

epoll有EPOLLLT和EPOLLET两种触发模式，LT是默认的模式，ET是“高速”模式。LT模式下，只要这个fd还有数据可读，每次 epoll_wait都会返回它的事件，提醒用户程序去操作，而在ET（边缘触发）模式中，它只会提示一次，直到下次再有数据流入之前都不会再提示了，无 论fd中是否还有数据可读。所以在ET模式下，read一个fd的时候一定要把它的buffer读光，也就是说一直读到read的返回值小于请求值，或者 遇到EAGAIN错误。还有一个特点是，epoll使用“事件”的就绪通知方式，通过epoll_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll_wait便可以收到通知。

epoll为什么要有EPOLLET触发模式?

如果采用EPOLLLT模式的话，系统中一旦有大量你不需要读写的就绪文件描述符，它们每次调用epoll_wait都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率.。而采用EPOLLET这种边沿触发模式的话，当被监控的文件描述符上有可读写事件发生时，epoll_wait()会通知处理程序去读写。如果这次没有把数据全部读写完(如读写缓冲区太小)，那么下次调用epoll_wait()时，它不会通知你，也就是它只会通知你一次，直到该文件描述符上出现第二次可读写事件才会通知你！！！
这种模式比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符

epoll的优点：

1、**没有最大并发连接的限制，能打开的FD的上限远大于1024（1G的内存上能监听约10万个端口）**；
2、效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数； 即Epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。

3、 内存拷贝，利用mmap()文件映射内存加速与内核空间的消息传递；即epoll使用mmap减少复制开销。 **select、poll、epoll 区别总结：**

1、支持一个进程所能打开的最大连接数

select

单个进程所能打开的最大连接数有FD_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是32*32*，同理*64*位机器上*FD_SETSIZE*为*32*64），当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试。

poll

poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的

epoll

虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接

2、FD剧增后带来的IO效率问题

select

因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”。

poll

同上

epoll

因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。

3、 消息传递方式

select

内核需要将消息传递到用户空间，都需要内核拷贝动作

poll

同上

epoll

epoll通过内核和用户空间共享一块内存来实现的。

总结：

综上，在选择select，poll，epoll时要根据具体的使用场合以及这三种方式的自身特点。**

1、表面上看epoll的性能最好，但是在连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多函数回调。

2、select低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善

关于这三种IO多路复用的用法，前面三篇总结写的很清楚，并用服务器回射echo程序进行了测试。连接如下所示：

select：IO多路复用之select总结

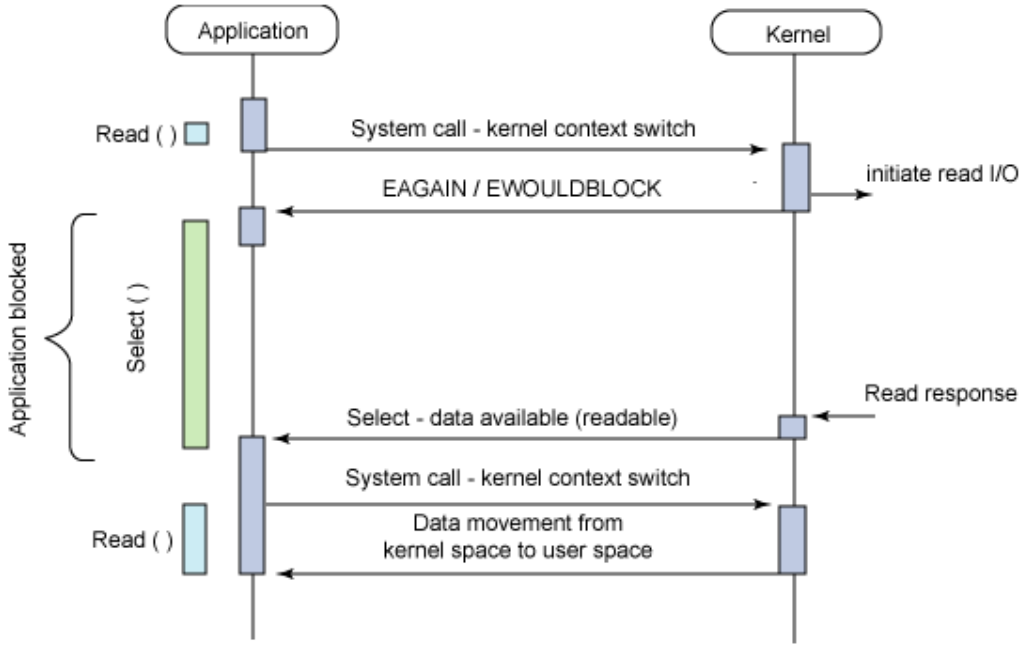
poll：O多路复用之poll总结

epoll：IO多路复用之epoll总结

今天对这三种IO多路复用进行对比，参考网上和书上面的资料，整理如下：

1、select实现

select的调用过程如下所示：



（1）使用copy_from_user从用户空间拷贝fd_set到内核空间

（2）注册回调函数__pollwait

（3）遍历所有fd，调用其对应的poll方法（对于socket，这个poll方法是sock_poll，sock_poll根据情况会调用到tcp_poll,udp_poll或者datagram_poll）

（4）以tcp_poll为例，其核心实现就是__pollwait，也就是上面注册的回调函数。

（5）__pollwait的主要工作就是把current（当前进程）挂到设备的等待队列中，不同的设备有不同的等待队列，对于tcp_poll来说，其等待队列是sk->sk_sleep（注意把进程挂到等待队列中并不代表进程已经睡眠了）。在设备收到一条消息（网络设备）或填写完文件数据（磁盘设备）后，会唤醒设备等待队列上睡眠的进程，这时current便被唤醒了。

（6）poll方法返回时会返回一个描述读写操作是否就绪的mask掩码，根据这个mask掩码给fd_set赋值。

（7）如果遍历完所有的fd，还没有返回一个可读写的mask掩码，则会调用schedule_timeout是调用select的进程（也就是current）进入睡眠。当设备驱动发生自身资源可读写后，会唤醒其等待队列上睡眠的进程。如果超过一定的超时时间（schedule_timeout指定），还是没人唤醒，则调用select的进程会重新被唤醒获得CPU，进而重新遍历fd，判断有没有就绪的fd。

（8）把fd_set从内核空间拷贝到用户空间。

总结：

select的几大缺点：

（1）每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大

（2）同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大

（3）select支持的文件描述符数量太小了，默认是1024

2 poll实现

poll的实现和select非常相似，只是描述fd集合的方式不同，poll使用pollfd结构而不是select的fd_set结构，其他的都差不多,管理多个描述符也是进行轮询，根据描述符的状态进行处理，**但是poll没有最大文件描述符数量的限制。**poll和select同样存在一个缺点就是，包含大量文件描述符的数组被整体复制于用户态和内核的地址空间之间，而不论这些文件描述符是否就绪，它的开销随着文件描述符数量的增加而线性增大。

3、epoll

epoll既然是对select和poll的改进，就应该能避免上述的三个缺点。那epoll都是怎么解决的呢？在此之前，我们先看一下epoll和select和poll的调用接口上的不同，select和poll都只提供了一个函数——select或者poll函数。而epoll提供了三个函数，epoll_create,epoll_ctl和epoll_wait，epoll_create是创建一个epoll句柄；epoll_ctl是注册要监听的事件类型；epoll_wait则是等待事件的产生。

对于第一个缺点，epoll的解决方案在epoll_ctl函数中。每次注册新的事件到epoll句柄中时（在epoll_ctl中指定EPOLL_CTL_ADD），会把所有的fd拷贝进内核，而不是在epoll_wait的时候重复拷贝。epoll保证了每个fd在整个过程中只会拷贝一次。

对于第二个缺点，epoll的解决方案不像select或poll一样每次都把current轮流加入fd对应的设备等待队列中，而只在epoll_ctl时把current挂一遍（这一遍必不可少）并为每个fd指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调函数，而这个回调函数会把就绪的fd加入一个就绪链表）。epoll_wait的工作实际上就是在这个就绪链表中查看有没有就绪的fd（利用schedule_timeout()实现睡一会，判断一会的效果，和select实现中的第7步是类似的）。

对于第三个缺点，epoll没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

总结：

（1）select，poll实现需要自己不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll其实也需要调用epoll_wait不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪fd放入就绪链表中，并唤醒在epoll_wait中进入睡眠的进程。虽然都要睡眠和交替，但是select和poll在“醒着”的时候要遍历整个fd集合，而epoll在“醒着”的时候只要判断

一下就绪链表是否为空就行了，这节省了大量的CPU时间。这就是回调机制带来的性能提升。

（2）select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，而且把current往等待队列上挂也只挂一次（在epoll_wait的开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内部定义的等待队列）。这也能节省不少的开销。

简述 Linux 进程调度的算法

困难 参考1

简述操作系统如何进行内存管理

中等 参考1 参考2

简述 Linux 系统态与用户态，什么时候会进入系统态?

简单 参考1

线程间有哪些通信方式?

中等

互斥锁、 读写锁、条件变量、信号量、 自旋锁、屏障

简述操作系统中的缺页中断

中等 参考1

简述同步与异步的区别，阻塞与非阻塞的区别

困难

简述操作系统如何进行内存管理

中等 参考1 参考2

简述几个常用的 Linux 命令以及他们的功能

简单

什么时候会由用户态陷入内核态?

简单

BIO、NIO 有什么区别？怎么判断写文件时 Buffer 已经写满？简述 Linux 的 IO模型，Linux下IO模型有几种，各自的含义是什么。

困难

简述操作系统中 malloc 的实现原理

中等

前言

从操作系统角度来看，进程分配内存有2种方式，分别由2个系统调用完成：brk和mmap（不考虑共享内存）。

1. brk是将数据段(.data)的最高地址指针_edata往高地址推
2. mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。 **这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。**

MALLOC概述

在C语言中只能通过malloc()和其派生的函数进行动态的申请内存，而实现的根本是通过**系统调用实现的（在linux下是通过sbrk()系统调用实现）**。

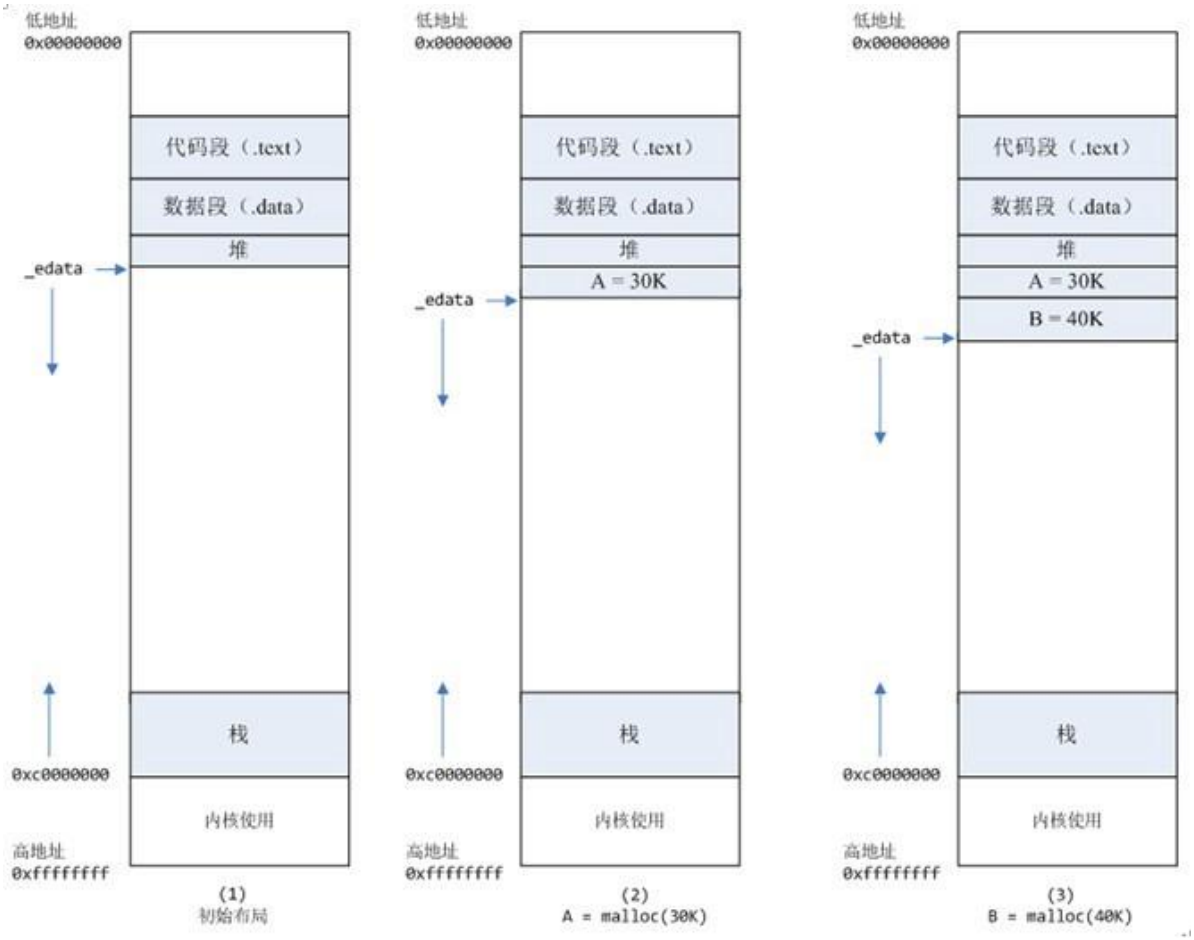
malloc()到底从哪里得到了内存空间？**答案是从堆里面获得空间。也就是说函数返回的指针是指向堆里面的一块内存。**操作系统中有一个记录空闲内存地址的链表。当**操作系统收到程序的申请时，就会遍历该链表，然后就寻找第一个空间大于所申请空间的堆结点，然后就将该结点从空闲结点链表中删除，**并将该结点的空间分配给程序。

malloc()在**运行期动态分配分配内存**,**free()释放由其分配的内存。**malloc()在**分配用户传入的大小的时候，还分配的一个相关的用于管理的额外内存，不过，用户是看不到的。**所以，

实际的大小 = 管理空间 + 用户空间

小于128K的内存分配

malloc小于128k的内存，使用brk分配内存，将_edata往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系)，如下图(32位系统)：



1. 进程启动的时候，其（虚拟）内存空间的初始布局如图1所示

其中，mmap内存映射文件是在堆和栈的中间（例如libc-2.2.93.so，其它数据文件等），为了简单起见，省略了内存映射文件。

_edata指针（glibc里面定义）指向数据段的最高地址。

1. 进程调用A=malloc(30K)以后，内存空间如图2

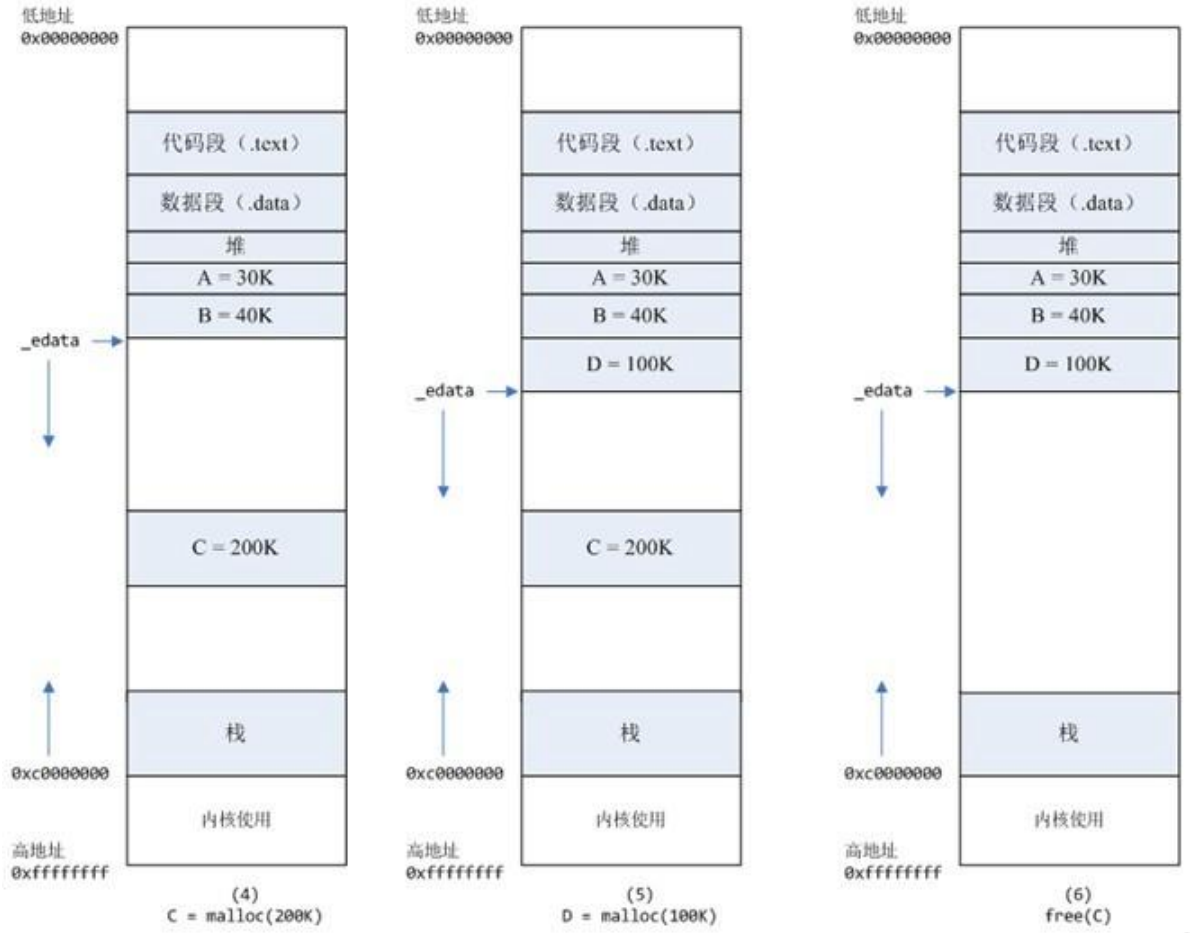
malloc函数会调用brk系统调用，将_edata指针往高地址推30K，就完成虚拟内存分配。

你可能会问：只要把_edata+30K就完成内存分配了？

事实是这样的，_edata+30K只是完成虚拟地址的分配，A这块内存现在还是没有物理页与之对应的，等到进程第一次读写A这块内存的时候，发生**缺页中断，这个时候，内核才分配A这块内存对应的物理页。**也就是说，如果用malloc分配了A这块内容，然后从来不访问它，那么，A对应的物理页是不会被分配的。

大于128K的内存分配

malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为0)，如下图：



1. 进程调用C=malloc(200K)以后，内存空间如图4：
2. 默认情况下，malloc函数分配内存，如果请求内存大于128K（可由M_MMAP_THRESHOLD选项调节），那就不是去推_edata指针了，而是利用mmap系统调用，从堆和栈的中间分配一块虚拟内存。这样子做主要是因为:brk分配的内存需要等到高地址内存释放以后才能释放（例如，在B释放之前，A是不可能释放的，这就是内存碎片产生的原因，什么时候紧缩看下面），而mmap分配的内存可以单独释放。

分配虚拟内存的细节

malloc()在运行期动态分配分配内存,free()释放由其分配的内存。malloc()在分配用户传入的大小的时候，还分配的一个相关的用于管理的额外内存，不过，用户是看不到的。所以，

实际的大小 = 管理空间 + 用户空间

在64位系统中，malloc(0)的有效内存大小为24，32位中为12,准确的说是至少是这么多，并且这些内存是可以用的

```c
#include <stdio.h>
#include <stdlib.h>


int main()
{
    char *p = (char*)malloc(0);
    char *p1 = (char*)malloc(5);
    char *p2 = (char*)malloc(25);
    char *p3 = (char*)malloc(39);
    char *p4 = (char*)malloc(41);
    printf("p size %d\n", malloc_usable_size(p));
    printf("p1 size %d\n", malloc_usable_size(p1));
    printf("p2 size %d\n", malloc_usable_size(p2));
    printf("p3 size %d\n", malloc_usable_size(p3));
    printf("p4 size %d\n", malloc_usable_size(p4));


    free(p);
    free(p1);
    free(p2);
    free(p3);
    free(p4);
    return 0;
}
```

结果为

```
p size 24
p1 size 24
p2 size 40
p3 size 40
p4 size 56
```

此外，**堆中的内存块总是成块分配的，并不是申请多少字节，就拿出多少个字节的内存来提供使用。堆中内存块的大小通常与内存对齐有关（8Byte(for 32bit system)或16Byte(for 64bit system)。**

因此，**在64位系统下，当(申请内存大小+sizeof(struct mem_control_block))% 16 == 0的时候，刚好可以完成一次满额的分配，但是当其!=0的时候，就会多分配内存块。**

```c
#include <stdio.h>
#include <stdlib.h>
#define LEN 24
int main()
{
    char* p = NULL;
    unsigned long lastaddr  = 0;
    int i   = 0;
    while (i < 5)
    {
        p = malloc(sizeof(char) * LEN);
        printf("%d:%10p, size:%d\n", i++, p, (int)p - lastaddr);
        lastaddr = (int)p;
    }
    return 0;
}
```

```
0:  0x8f9010, size:9408528
1:  0x8f9030, size:32
2:  0x8f9050, size:32
3:  0x8f9070, size:32
4:  0x8f9090, size:32
```

Malloc24时，实际的可用大小为32

```c
#include <stdio.h>
#include <stdlib.h>
#define LEN 25
int main()
{
    char* p = NULL;
    unsigned long lastaddr  = 0;
    int i   = 0;
    while (i < 5)
    {
        p = malloc(sizeof(char) * LEN);
        printf("%d:%10p, size:%d\n", i++, p, (int)p - lastaddr);
        lastaddr = (int)p;
    }
    return 0;
}
```

```
0: 0x128f010, size:19460112
1: 0x128f040, size:48
2: 0x128f070, size:48
3: 0x128f0a0, size:48
4: 0x128f0d0, size:48
```

Malloc25时，实际的可用大小为40


```c
#include <stdio.h>
#include <stdlib.h>
#define LEN 24
int main()
{
    char* p = NULL;
    unsigned long lastaddr  = 0;
    int i   = 0;
    while (i < 5)
    {
        p = malloc(sizeof(char) * LEN);
        printf("%d:%10p, size:%d\n", i++, p, (int)p - lastaddr);
        lastaddr = (int)p;
    }
    return 0;
}
```

```
0:  0x8f9010, size:9408528
1:  0x8f9030, size:32
2:  0x8f9050, size:32
3:  0x8f9070, size:32
4:  0x8f9090, size:32
```

Malloc24时，实际的可用大小为32

```c
#include <stdio.h>
#include <stdlib.h>
#define LEN 25
int main()
{
    char* p = NULL;
    unsigned long lastaddr  = 0;
    int i   = 0;
    while (i < 5)
    {
        p = malloc(sizeof(char) * LEN);
        printf("%d:%10p, size:%d\n", i++, p, (int)p - lastaddr);
        lastaddr = (int)p;
    }
    return 0;
}
```

```
0: 0x128f010, size:19460112
1: 0x128f040, size:48
2: 0x128f070, size:48
3: 0x128f0a0, size:48
4: 0x128f0d0, size:48
```

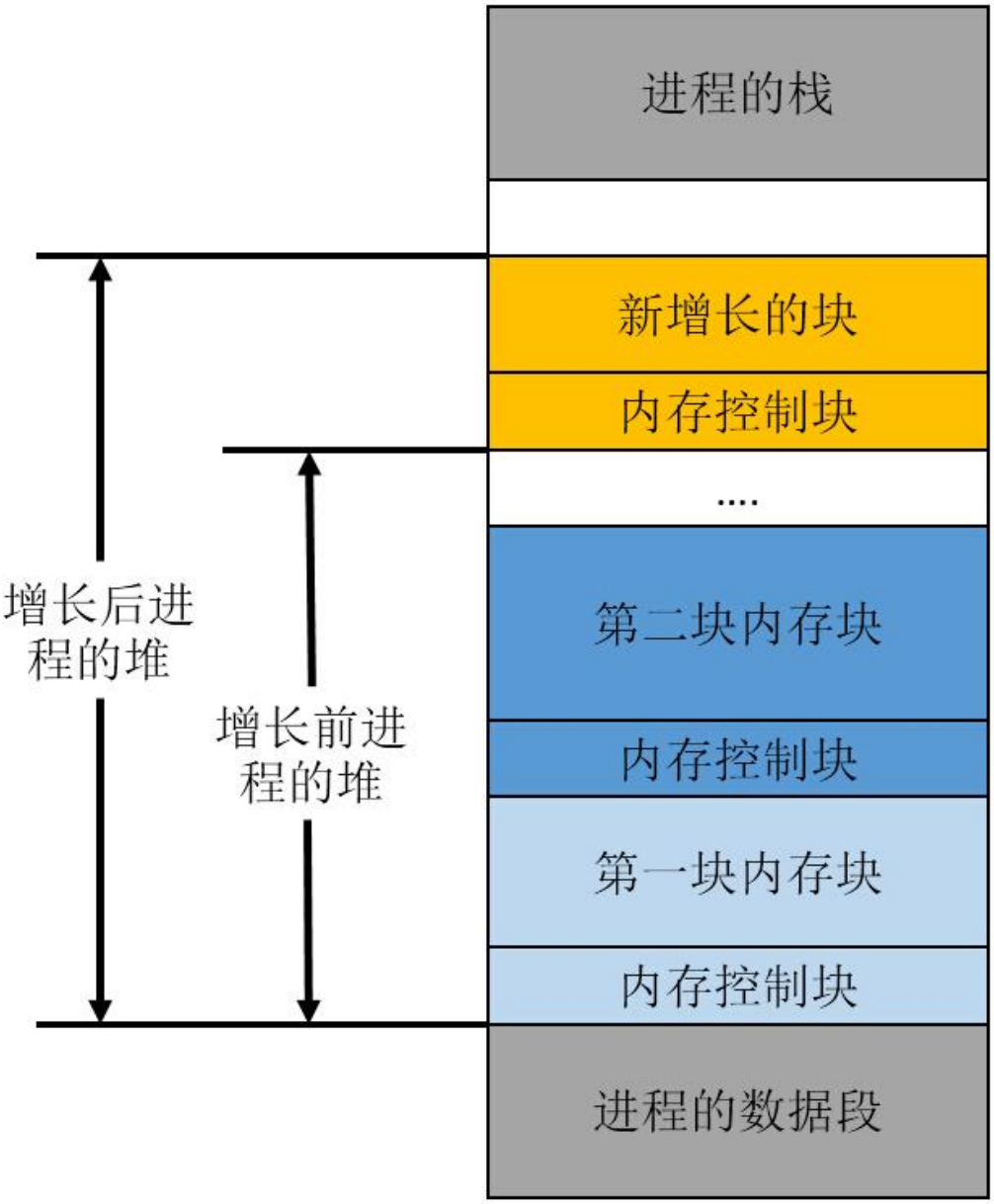
Malloc25时，实际的可用大小为40

在linux系统下面一个程序的堆的管理是通过内存块进行管理的，也就是将堆分成了很多大小不一的内存块。这些块怎么管理尼，比如怎么查询块的大小，怎么查询块是否正在被程序使用，怎么知道这个块的地址。为了解决内存块的管理所以要设计一个管理内存块的数据结构，详细的数据结构如下：

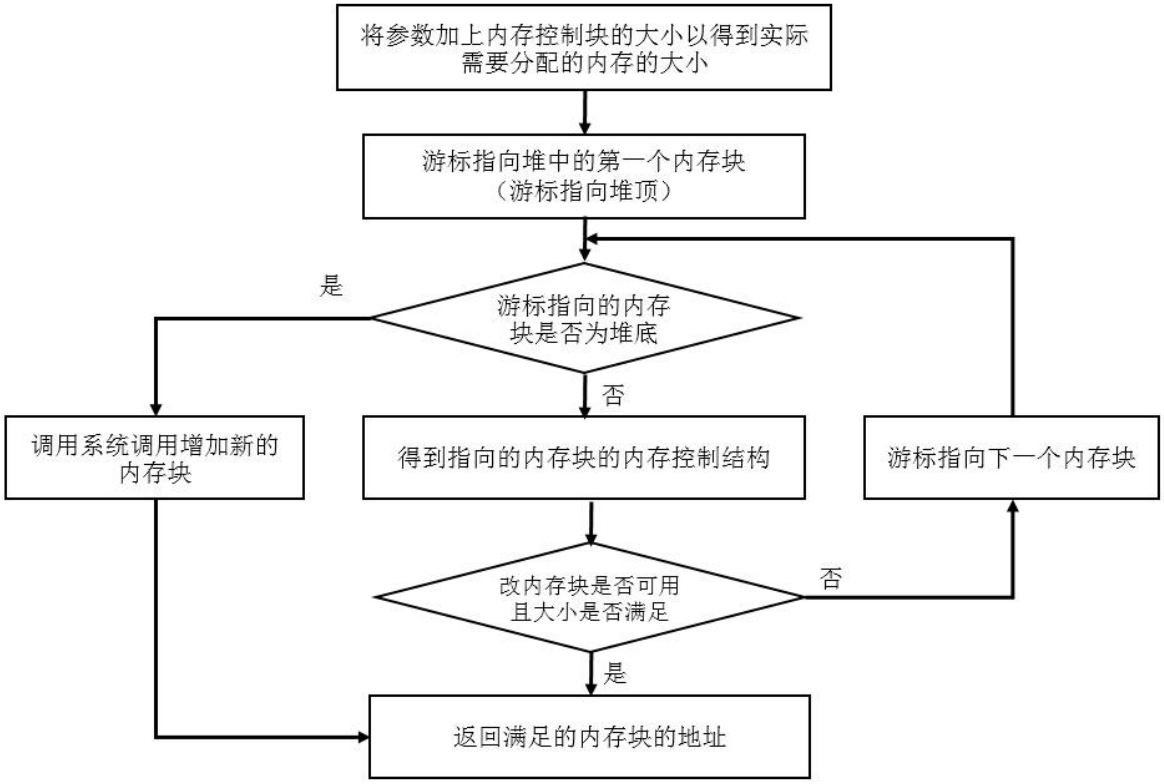
```
/**内存控制块数据结构，用于管理所有的内存块
* is_available: 标志着该块是否可用。1表示可用，0表示不可用
* size: 该块的大小
**/

struct mem_control_block {
    int is_available;
    int size;
};
```

有了管理内存块的数据结构，那么在内存中堆的组织形式也好理解了，也就是堆是由很多内存块组成的，所以有了如下的示意图：



综合上面的知识，可以很容易想到malloc()实现的大体思路。**首先挨个检查堆中的内存是否可用，如果可用那么大小是否能满足需求，要是都满足的话就直接用。当遍历了堆中的所有内存块时，要是没有能满足需求的块时就只能通过系统调用向操作系统申请新的内存，然后将新的内存添加到堆中。**思路很简单，malloc()实现流程图如下所示：



看完上面的思路，也会很容易的想到free()函数的实现思路，只要将内存管理块设置为可用就可以了。这样下次调用malloc()函数的时候就可以将该内存块作为可分配块再次进行分配了。

最后，贴上malloc()和free()实现的代码：

malloc()实现：

```
/**内存控制块数据结构，用于管理所有的内存块
* is_available: 标志着该块是否可用。1表示可用，0表示不可用
* size: 该块的大小
**/

struct mem_control_block {
    int is_available;
    int size;
};
/**在实现malloc时要用到linux下的全局变量
*managed_memory_start：该指针指向进程的堆底，也就是堆中的第一个内存块
*last_valid_address：该指针指向进程的堆顶，也就是堆中最后一个内存块的末地址
**/
void *managed_memory_start;
void *last_valid_address;

/**malloc()功能是动态的分配一块满足参数要求的内存块
*numbytes：该参数表明要申请多大的内存空间
*返回值：函数执行结束后将返回满足参数要求的内存块首地址，要是没有分配成功则返回NULL
**/

void *malloc(size_t numbytes) {

    //游标，指向当前的内存块
    void *current_location;
    //保存当前内存块的内存控制结构
```

```
    struct mem_control_block *current_location_mcb;
    //保存满足条件的内存块的地址用于函数返回
    void *memory_location;
    memory_location = NULL;
    //计算内存块的实际大小，也就是函数参数指定的大小+内存控制块的大小
    numbytes = numbytes + sizeof(struct mem_control_block);

    //利用全局变量得到堆中的第一个内存块的地址
    current_location = managed_memory_start;

    //对堆中的内存块进行遍历，找合适的内存块
   while (current_location != last_valid_address) //检查是否遍历到堆顶了
    {
        //取得当前内存块的内存控制结构
        current_location_mcb = (struct mem_control_block*)current_location;

        //判断该块是否可用
        if (current_location_mcb->is_available)
            //检查该块大小是否满足
            if (current_location_mcb->size >= numbytes)
            {
                //满足的块将其标志为不可用
                current_location_mcb->is_available = 0;
                //得到该块的地址，结束遍历
                memory_location = current_location;
                break;
            }
        //取得下一个内存块
        current_location = current_location + current_location_mcb->size;
    }
    //在堆中已有的内存块中没有找到满足条件的内存块时执行下面的函数
    if (!memory_location)
    {      //向操作系统申请新的内存块
        if (sbrk(numbytes) == -1)
            return NULL;//申请失败，说明系统没有可用内存
        memory_location = last_valid_address;
        last_valid_address = last_valid_address + numbytes;
        current_location_mcb = (struct mem_control_block)memory_location;
        current_location_mcb->is_available = 0;
        current_location_mcb->size = numbytes;
    }
    //到此已经得到所要的内存块，现在要做的是越过内存控制块返回内存块的首地址
    memory_location = memory_location + sizeof(struct mem_control_block);
    return memory_location;
}
```

free实现：

```
/**free()功能是将参数指向的内存块进行释放
*firstbyte：要释放的内存块首地址
*返回值：空
**/

void free(void *firstbyte)
{
    struct mem_control_block *mcb;
    //取得该块的内存控制块的首地址
```

```
    mcb = firstbyte - sizeof(struct mem_control_block);
    //将该块标志设为可用
    mcb->is_available = 1;
    return;
}
```

缺页中断

用ps -o majflt,minflt -C program命令查看缺页中断的次数。

majflt代表majorfault，中文名叫大错误，minflt代表minor fault，中文名叫小错误。

这2个数值表示一个进程自启动以来所发生的缺页中断的次数。

发生缺页中断后，执行了哪些操作?

当一个进程发生缺页中断的时候，进程会陷入内核态，执行以下操作：

1. 检查要访问的虚拟地址是否合法
2. 查找/分配一个物理页
3. 填充物理页内容（读取磁盘，或者直接置0，或者啥也不干）
4. 建立映射关系（虚拟地址到物理地址）
5. 重新执行发生缺页中断的那条指令

如果第3步，需要读取磁盘，那么这次缺页中断就是majflt，否则就是minflt。

Linux 下如何查看端口被哪个进程占用?

简单

```
lsof -i
lsof -i:端口号

netstat -tunlp |grep 端口号
```

Linux 中虚拟内存和物理内存有什么区别？有什么优点?

简单

虚拟地址

即使是现代操作系统中，内存依然是计算机中很宝贵的资源，看看你电脑几个T固态硬盘，再看看内存大小就知道了。

虚拟内存是Linux管理内存的一种技术。它使得每个应用程序都认为自己拥有独立且连续的可用的内存空间（一段连续完整的地址空间），而实际上，它通常是被映射到多个物理内存段，还有部分暂时存储在外部磁盘存储器上，在需要时再加载到内存中来。

每个进程所能使用的虚拟地址大小和CPU位数有关，在32位的系统上，虚拟地址空间大小是4G，在64位系统上，是2^64=？（算不过来了）。而实际的物理内存可能远远小于虚拟地址空间的大小。

虚拟地址和进程息息相关，不同进程里的同一个虚拟地址指向的物理地址不一定一样，所以离开进程谈虚拟地址没有任何意义。

注意：网上很多文章将虚拟内存等同于交换空间，其实描述不够严谨，交换空间只是虚拟内存这张大蓝图中的一部分。

为了充分利用和管理系统内存资源，Linux采用虚拟内存管理技术，在32位的系统上，利用虚拟内存技术让每个进程都有4GB 互不干涉的虚拟地址空间。进程初始化分配和操作的都是基于这个「虚拟地址」，只有当进程需要实际访问内存资源的时候才会建立虚拟地址和物理地址的映射，调入物理内存页。

打个不是很恰当的比方，这个原理其实和现在的某某网盘一样。假如你的网盘空间是1TB，真以为就一口气给了你这么大空间吗？那还是太年轻，都是在你往里面放东西的时候才给你分配空间，你放多少就分多少实际空间给你，但你和你朋友看起来就像大家都拥有1TB空间一样。

虚拟内存和物理内存的关系

下面这张表很直观的表述了它们之间的关系

```
  进程X                                                                                                进程Y
+-------+                                                                                            +-------+
| VPFN7 |--+                                                                                         | VPFN7 |
+-------+  |       进程X的                                          进程Y的                           +-------+
| VPFN6 |  |      Page Table                                       Page Table                    +-| VPFN6 |
+-------+  |      +------+                                         +------+                      |  +-------+
| VPFN5 |  +----->| .... |---+                                +-------| .... |<---+  | |  | VPFN5 |
+-------+         +------+   |        +------+    |       +------+    |  |  +-------+
| VPFN4 |    +--->| .... |---+-+      | PFN4 |    |       | .... |    |  | |  | VPFN4 |
+-------+    |    +------+   | |      +------+    |       +------+    |  |  +-------+
| VPFN3 |--+ |    | .... |   | | +--->| PFN3 |<---+  +----| .... |<---+--+ |  | VPFN3 |
+-------+  | |    +------+   | | |    +------+       |    +------+    |       +-------+
| VPFN2 |  +-+--->| .... |---+-+-+    | PFN2 |<------+    | .... |    |    |  | VPFN2 |
+-------+    |    +------+   | |      +------+            +------+    |       +-------+
| VPFN1 |    |               | +----->| FPN1 |                        +----|  | VPFN1 |
+-------+    |               |        +------+                             +-------+
| VPFN0 |----+               +------->| PFN0 |                             |  | VPFN0 |
+-------+                            +------+                             +-------+
 虚拟内存                              物理内存                                虚拟内存
PFN(the page frame number)： 页编号
```

当进程执行一个程序时，需要先从先内存中读取该进程的指令，然后执行，获取指令时用到的就是虚拟地址，这个地址是程序链接时确定的（内核加载并初始化进程时会调整动态库的地址范围），为了获取到实际的数据，CPU需要将虚拟地址转换成物理地址，CPU转换地址时需要用到进程的page table，而page table里面的数据由操作系统维护。

注意：Linux内核代码访问内存时用的都是实际的物理地址，所以不存在虚拟地址到物理地址的转换，只有应用层程序才需要。

为了转换方便，Linux将虚拟内存和物理内存都拆分为固定大小的页，x86的系统一般内存页大小是4K，每个页都会分配一个唯一的编号，这就是页编号（PFN）.

从上面的图中可以看出，虚拟内存和物理内存的page之间通过page table进行映射。进程X和Y的虚拟内存是相互独立的，且page table也是独立的，它们之间共享物理内存。进程可以随便访问自己的虚拟地址空间，而page table和物理内存由内核维护。当进程需要访问内存时，CPU会根据进程的page table将虚拟地址翻译成物理地址，然后进行访问。

注意：并不是每个虚拟地址空间的page都有对应的Page Table相关联，只有虚拟地址被分配给进程后，也即进程调用类似malloc函数之后，系统才会为相应的虚拟地址在Page Table中添加记录，如果进程访问一个没有和Page Table关联的虚拟地址，系统将会抛出SIGSEGV信号，导致进程退出，这也是为什么我们访问野指针时会经常出现segmentfault的原因。换句话说，虽然每个进程都有4G（32位系统）的虚拟地址空间，但只有向系统申请了的那些地址空间才能用，访问未分配的地址空间将会出segmentfault错误。Linux会将虚拟地址0不映射到任何地方，这样我们访问空指针就一定会报segmentfault错误。

page table

page table可以简单的理解为一个memory mapping的链表（当然实际结构很复杂），里面的每个memory mapping都将一块虚拟地址映射到一个特定的资源（物理内存或者外部存储空间）。每个进程拥有自己的page table，和其它进程的page table没有关系。

memory mapping

每个memory mapping就是对一段虚拟内存的描述，包括虚拟地址的起始位置，长度，权限(比如这段内存里的数据是否可读、写、执行), 以及关联的资源(如物理内存page，swap空间上的page，磁盘上的文件内容等)。

当进程申请内存时，系统将返回虚拟内存地址，同时为相应的虚拟内存创建memory mapping并将它放入page table，但这时系统不一定会分配相应的物理内存，系统一般会在进程真正访问这段内存的时候才会分配物理内存并关联到相应的memory mapping，这就是所谓的延时分配/按需分配。

每个memory mapping都有一个标记，用来表示所关联的物理资源类型，一般分两大类，那就是anonymous和file backed，在这两大类中，又分了一些小类，比如anonymous下面有更具体的shared和copy on write类型, file backed下面有更具体的device backed类型。下面是每个类型所代表的意思：

file backed

这种类型表示memory mapping对应的物理资源存放在磁盘上的文件中，它所包含的信息包括文件的位置、offset、rwx权限等。

当进程第一次访问对应的虚拟page的时候，由于在memory mapping中找不到对应的物理内存，CPU会报page fault中断，然后操作系统就会处理这个中断并将文件的内容加载到物理内存中，然后更新memory mapping，这样下次CPU就能访问这块虚拟地址了。以这种方式加载到内存的数据一般都会放到page cache中，关于page cache会在后面介绍到.

一般程序的可执行文件，动态库都是以这种方式映射到进程的虚拟地址空间的。

device backed

和file backed类似，只是后端映射到了磁盘的物理地址，比如当物理内存被swap out后，将被标记为device backed。

anonymous

程序自己用到的数据段和堆栈空间，以及通过mmap分配的共享内存，它们在磁盘上找不到对应的文件，所以这部分内存页被叫做anonymous page。anonymous page和file backed最大的差别是当内存吃紧时，系统会直接删除掉file backed对应的物理内存，因为下次需要的时候还能从磁盘加载到内存，但anonymous page不能被删除，只能被swap out。

shared

不同进程的Page Table里面的多个memory mapping可以映射到相同的物理地址，通过虚拟地址（不同进程里的虚拟地址可能不一样）可以访问到相同的内容，当一个进程里面修改内存的内容后，在另一个进程中可以立即读取到。这种方式一般用来实现进程间高速的共享数据（如mmap）。当标记为shared的memory mapping被删除回收时，需要更新物理page上的引用计数，便于物理page的计数变0后被回收。

copy on write

copy on write基于shared技术，当读这种类型的内存时，系统不需要做任何特殊的操作，而当要写这块内存时，系统将会生成一块新的内存并拷贝原来内存中的数据到新内存中，然后将新内存关联到相应的memory mapping，然后执行写操作。Linux下很多功能都依赖于copy on write技术来提高性能，比如fork等。

通过上面的介绍，我们可以简单的将内存的使用过程总结如下：

进程向系统发出内存申请请求

系统会检查进程的虚拟地址空间是否被用完，如果有剩余，给进程分配虚拟地址

系统为这块虚拟地址创建相应的memory mapping（可能多个），并将它放进该进程的page table

系统返回虚拟地址给进程，进程开始访问该虚拟地址

CPU根据虚拟地址在该进程的page table中找到了相应的memory mapping，但是该mapping没有和物理内存关联，于是产生缺页中断

操作系统收到缺页中断后，分配真正的物理内存并将它关联到相应的memory mapping

中断处理完成后，CPU就可以访问该内存了

当然缺页中断不是每次都会发生，只有系统觉得有必要延迟分配内存的时候才用的着，也即很多时候在上面的第3步系统会分配真正的物理内存并和memory mapping关联。

虚拟内存的优点

更大的地址空间：并且是连续的，使得程序编写、链接更加简单

进程隔离：不同进程的虚拟地址之间没有关系，所以一个进程的操作不会对其它进程造成影响

数据保护：每块虚拟内存都有相应的读写属性，这样就能保护程序的代码段不被修改，数据块不能被执行等，增加了系统的安全性

内存映射：有了虚拟内存之后，可以直接映射磁盘上的文件（可执行文件或动态库）到虚拟地址空间，这样可以做到物理内存延时分配，只有在需要读相应的文件的时候，才将它真正的从磁盘上加载到内存中来，而在内存吃紧的时候又可以将这部分内存清空掉，提高物理内存利用效率，并且所有这些对应用程序来说是都透明的

共享内存：比如动态库，只要在内存中存储一份就可以了，然后将它映射到不同进程的虚拟地址空间中，让进程觉得自己独占了这个文件。进程间的内存共享也可以通过映射同一块物理内存到进程的不同虚拟地址空间来实现共享

物理内存管理：物理地址空间全部由操作系统管理，进程无法直接分配和回收，从而系统可以更好的利用内存，平衡进程间对内存的需求

其它：有了虚拟地址空间后，交换空间和COW（copy on write）等功能都能很方便的实现

其它概念

操作系统只要实现了虚拟内存和物理内存之间的映射关系，就能正常工作了，但要使内存访问更高效，还有很多东西需要考虑，在这里我们可以看看跟内存有关的一些其它概念以及它们的作用。

MMU（Memory Management Unit）

MMU是CPU的一个用来将进程的虚拟地址转换成物理地址的模块，简单点说，这个模块的输入是进程的page table和虚拟地址，输出是物理地址。将虚拟地址转换成物理地址的速度直接影响着系统的速度，所以CPU包含了这个模块用来加速。

TLB（Translation Lookaside Buffer）

上面介绍到，MMU的输入是page table，而page table又存在内存里面，跟CPU的cache相比，内存的速度很慢，所以为了进一步加快虚拟地址到物理地址的转换速度，Linux发明了TLB，它存在于CPU的L1 cache里面，用来缓存已经找到的虚拟地址到物理地址的映射，这样下次转换前先查一下TLB，如果已经在里面了就不需要调用MMU了.

进程空间从高位到低位都有些什么?

简单

进程的内存布局在结构上是有规律的，具体来说对于 linux 系统上的进程，其内存空间一般可以粗略地分为以下几大段【1】，从高内存到低内存排列：

1、内核态内存空间，其大小一般比较固定（可以编译时调整），但 32 位系统和 64 位系统的值不一样。

2、用户态的堆栈，大小不固定，可以用 ulimit -s 进行调整，默认一般为 8M，从高地址向低地址增长。

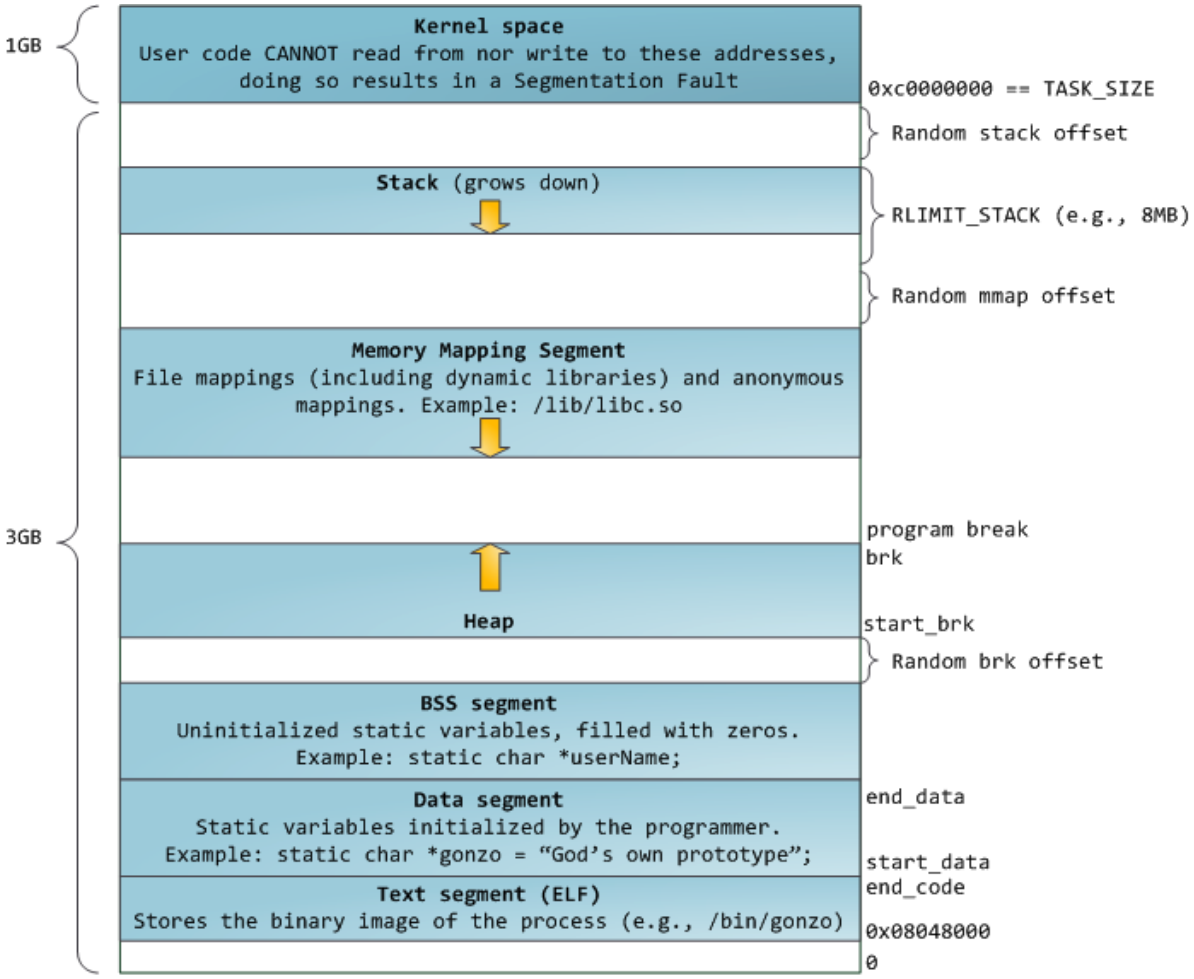
3、mmap 区域，进程茫茫内存空间里的主要部分，既可以从高地址到低地址延伸(所谓 flexible layout)，也可以从低到高延伸(所谓 legacy layout)，看进程具体情况【2】【3】。

4、brk 区域，紧邻数据段(甚至贴着)，从低位向高位伸展，但它的大小主要取决于 mmap 如何增长，一般来说，即使是 32 位的进程以传统方式延伸，也有差不多 1 GB 的空间（准确地说是 TASK_SIZE/3 - 代码段数据段，参看 arch/x86/include/asm/processor.h 里的定义)【4】

5、数据段，主要是进程里初始化和未初始化的全局数据总和，当然还有编译器生成一些辅助数据结构等等)，大小取决于具体进程，其位置紧贴着代码段。

6、代码段，主要是进程的指令，包括用户代码和编译器生成的辅助代码，其大小取决于具体程序，但起始位置根据 32 位还是 64 位一般固定(-fPIC, -fPIE等除外【5】)。

以上各段(除了代码段数据段)其起始位置根据系统是否起用 randomize_va_space 一般稍有变化，各段之间因此可能有随机大小的间隔，千言万语不如一幅图（x86-32位下）：



32位下bash进程的示例：

```
root@sylas-VirtualBox:~# cat /proc/$$/maps
08048000-08124000 r-xp 00000000 08:01 1966084    /bin/bash
08124000-08125000 r--p 000db000 08:01 1966084    /bin/bash
08125000-0812a000 rw-p 000dc000 08:01 1966084    /bin/bash
0812a000-0812f000 rw-p 00000000 00:00 0
09738000-09989000 rw-p 00000000 00:00 0          [heap]
b72e0000-b72eb000 r-xp 00000000 08:01 2627982    /lib/i386-linux-gnu/libnss_files-2.15.so
b72eb000-b72ec000 r--p 0000a000 08:01 2627982    /lib/i386-linux-gnu/libnss_files-2.15.so
b72ec000-b72ed000 rw-p 0000b000 08:01 2627982    /lib/i386-linux-gnu/libnss_files-2.15.so
b72ed000-b72f7000 r-xp 00000000 08:01 2622624    /lib/i386-linux-gnu/libnss_nis-2.15.so
b72f7000-b72f8000 r--p 00009000 08:01 2622624    /lib/i386-linux-gnu/libnss_nis-2.15.so
b72f8000-b72f9000 rw-p 0000a000 08:01 2622624    /lib/i386-linux-gnu/libnss_nis-2.15.so
b72f9000-b730f000 r-xp 00000000 08:01 2627976    /lib/i386-linux-gnu/libnsl-2.15.so
b730f000-b7310000 r--p 00015000 08:01 2627976    /lib/i386-linux-gnu/libnsl-2.15.so
b7310000-b7311000 rw-p 00016000 08:01 2627976    /lib/i386-linux-gnu/libnsl-2.15.so
b7311000-b7313000 rw-p 00000000 00:00 0
b731c000-b7323000 r--s 00000000 08:01 1181007    /usr/lib/i386-linux-gnu/gconv/gconv-modules.cache
b7323000-b7523000 r--p 00000000 08:01 1187343    /usr/lib/locale/locale-archive
b7523000-b7524000 rw-p 00000000 00:00 0
b7524000-b76c7000 r-xp 00000000 08:01 2627974    /lib/i386-linux-gnu/libc-2.15.so
b76c7000-b76c9000 r--p 001a3000 08:01 2627974    /lib/i386-linux-gnu/libc-2.15.so
b76c9000-b76ca000 rw-p 001a5000 08:01 2627974    /lib/i386-linux-gnu/libc-2.15.so
b76ca000-b76ce000 rw-p 00000000 00:00 0
b76ce000-b76d1000 r-xp 00000000 08:01 2622623    /lib/i386-linux-gnu/libdl-2.15.so
b76d1000-b76d2000 r--p 00002000 08:01 2622623    /lib/i386-linux-gnu/libdl-2.15.so
b76d2000-b76d3000 rw-p 00003000 08:01 2622623    /lib/i386-linux-gnu/libdl-2.15.so
b76d3000-b76ef000 r-xp 00000000 08:01 2622537    /lib/i386-linux-gnu/libtinfo.so.5.9
b76ef000-b76f1000 r--p 0001b000 08:01 2622537    /lib/i386-linux-gnu/libtinfo.so.5.9
b76f1000-b76f2000 rw-p 0001d000 08:01 2622537    /lib/i386-linux-gnu/libtinfo.so.5.9
b76f8000-b76ff000 r-xp 00000000 08:01 2627981    /lib/i386-linux-gnu/libnss_compat-2.15.so
b76ff000-b7700000 r--p 00006000 08:01 2627981    /lib/i386-linux-gnu/libnss_compat-2.15.so
b7700000-b7701000 rw-p 00007000 08:01 2627981    /lib/i386-linux-gnu/libnss_compat-2.15.so
b7701000-b7702000 r--p 005e0000 08:01 1187343    /usr/lib/locale/locale-archive
b7702000-b7704000 rw-p 00000000 00:00 0
b7704000-b7705000 r-xp 00000000 00:00 0          [vdso]
b7705000-b7725000 r-xp 00000000 08:01 2627980    /lib/i386-linux-gnu/ld-2.15.so
b7725000-b7726000 r--p 0001f000 08:01 2627980    /lib/i386-linux-gnu/ld-2.15.so
b7726000-b7727000 rw-p 00020000 08:01 2627980    /lib/i386-linux-gnu/ld-2.15.so
bf9ae000-bf9cf000 rw-p 00000000 00:00 0          [stack]
```


平时用到哪些Linux命令。

1.1 关机和重启 关机 shutdown -h now 立刻关机 shutdown -h 5 5分钟后关机 poweroff 立刻关机 重启 shutdown -r now 立刻重启 shutdown -r 5 5分钟后重启 reboot 立刻重启

1.2 帮助命令 --help命令 shutdown --help： ifconfig --help：查看网卡信息

man命令（命令说明书） man shutdown 注意：man shutdown打开命令说明书之后，使用按键q退出

二、目录操作命令 2.1 目录切换 cd 命令：cd 目录

cd / 切换到根目录 cd /usr 切换到根目录下的usr目录 cd ../ 切换到上一级目录 或者 cd .. cd ~ 切换到home目录 cd - 切换到上次访问的目录

2.2 目录查看 ls [-al] 命令：ls [-al]

ls 查看当前目录下的所有目录和文件 ls -a 查看当前目录下的所有目录和文件（包括隐藏的文件） ls -l 或 ll 列表查看当前目录下的所有目录和文件（列表查看，显示更多信息） ls /dir 查看指定目录下的所有目录和文件 如：ls /usr

2.3 目录操作【增，删，改，查】 2.3.1 创建目录【增】 mkdir 命令：mkdir 目录

mkdir aaa 在当前目录下创建一个名为aaa的目录 mkdir /usr/aaa 在指定目录下创建一个名为aaa的目录

2.3.2 删除目录或文件【删】rm 命令：rm [-rf] 目录

删除文件： rm 文件 删除当前目录下的文件 rm -f 文件 删除当前目录的的文件（不询问）

删除目录： rm -r aaa 递归删除当前目录下的aaa目录 rm -rf aaa 递归删除当前目录下的aaa目录（不询问）

全部删除： rm -rf * 将当前目录下的所有目录和文件全部删除 rm -rf /* 【自杀命令！慎用！慎用！慎用！】将根目录下的所有文件全部删除

注意：rm不仅可以删除目录，也可以删除其他文件或压缩包，为了方便大家的记忆，无论删除任何目录或文件，都直接使用 rm -rf 目录/文件/压缩包

2.3.3 目录修改【改】mv 和 cp 一、重命名目录 命令：mv 当前目录 新目录 例如：mv aaa bbb 将目录aaa改为bbb 注意：mv的语法不仅可以对目录进行重命名而且也可以对各种文件，压缩包等进行 重命名的操作

二、剪切目录 命令：mv 目录名称 目录的新位置 示例：将/usr/tmp目录下的aaa目录剪切到 /usr目录下面 mv /usr/tmp/aaa /usr 注意：mv语法不仅可以对目录进行剪切操作，对文件和压缩包等都可执行剪切操作

三、拷贝目录 命令：cp -r 目录名称 目录拷贝的目标位置 -r代表递归 示例：将/usr/tmp目录下的aaa目录复制到 /usr目录下面 cp /usr/tmp/aaa /usr 注意：cp命令不仅可以拷贝目录还可以拷贝文件，压缩包等，拷贝文件和压缩包时不 用写-r递归

2.3.4 搜索目录【查】find 命令：find 目录 参数 文件名称 示例：find /usr/tmp -name 'a*' 查找/usr/tmp目录下的所有以a开头的目录或文件

三、文件操作命令 3.1 文件操作【增，删，改，查】 3.1.1 新建文件【增】touch 命令：touch 文件名 示例：在当前目录创建一个名为aa.txt的文件 touch aa.txt

3.1.2 删除文件 【删】 rm 命令：rm -rf 文件名

3.1.3 修改文件【改】 vi或vim 【vi编辑器的3种模式】 基本上vi可以分为三种状态，分别是命令模式（command mode）、插入模式（Insert mode）和底行模式（last line mode），各模式的功能区分如下：

1. 命令行模式command mode） 控制屏幕光标的移动，字符、字或行的删除，查找，移动复制某区段及进入Insert mode下，或者到 last line mode。 命令行模式下的常用命令： 【1】控制光标移

动：↑，↓，j【2】删除当前行：dd【3】查找：/字符【4】进入编辑模式：i o a【5】进入底行模式：：

2. 编辑模式（Insert mode） 只有在Insert mode下，才可以做文字输入，按「ESC」键可回到命令行模式。 编辑模式下常用命令： 【1】ESC 退出编辑模式到命令行模式；
3. 底行模式（last line mode） 将文件保存或退出vi，也可以设置编辑环境，如寻找字符串、列出行号……等。 底行模式下常用命令： 【1】退出编辑： :q 【2】强制退出： :q! 【3】保存并退出： :wq

打开文件

命令：vi 文件名 示例：打开当前目录下的aa.txt文件 vi aa.txt 或者 vim aa.txt

注意：使用vi编辑器打开文件后，并不能编辑，因为此时处于命令模式，点击键盘i/a/o进入编辑模式。

编辑文件

使用vi编辑器打开文件后点击按键：i，a或者o即可进入编辑模式。

i:在光标所在字符前开始插入 a:在光标所在字符后开始插入 o:在光标所在行的下面另起一新行插入

保存或者取消编辑

保存文件：

第一步：ESC 进入命令行模式 第二步：: 进入底行模式 第三步：wq 保存并退出编辑

取消编辑：

第一步：ESC 进入命令行模式 第二步：: 进入底行模式 第三步：q! 撤销本次修改并退出编辑

3.1.4 文件的查看【查】 文件的查看命令：cat/more/less/tail

cat：看最后一屏

示例：使用cat查看/etc/sudo.conf文件，只能显示最后一屏内容 cat sudo.conf

more：百分比显示

示例：使用more查看/etc/sudo.conf文件，可以显示百分比，回车可以向下一行，空格可以向下一页，q可以退出查看 more sudo.conf

less：翻页查看

示例：使用less查看/etc/sudo.conf文件，可以使用键盘上的PgUp和PgDn向上 和向下翻页，q结束查看 less sudo.conf

tail：指定行数或者动态查看

示例：使用tail -10 查看/etc/sudo.conf文件的后10行，Ctrl+C结束
tail -10 sudo.conf

3.2 权限修改 rwx：r代表可读，w代表可写，x代表该文件是一个可执行文件，如果rwx任意位置变为-则代表不可读或不可写或不可执行文件。

示例：给aaa.txt文件权限改为可执行文件权限，aaa.txt文件的权限是-rw-------

第一位：-就代表是文件，d代表是文件夹 第一段（3位）：代表拥有者的权限 第二段（3位）：代表拥有者所在的组，组员的权限 第三段（最后3位）：代表的是其他用户的权限

421 421 421

- rw- --- ---

命令：chmod +x aaa.txt 或者采用8421法 命令：chmod 100 aaa.txt 四、压缩文件操作 4.1 打包和压缩 Windows的压缩文件的扩展名 .zip/.rar linux中的打包文件：aa.tar
linux中的压缩文件：bb.gz
linux中打包并压缩的文件：.tar.gz

Linux中的打包文件一般是以.tar结尾的，压缩的命令一般是以.gz结尾的。 而一般情况下打包和压缩是一起进行的，打包并压缩后的文件的后缀名一般.tar.gz。

命令：tar -zcvf 打包压缩后的文件名 要打包的文件 其中：z：调用gzip压缩命令进行压缩 c：打包文件 v：显示运行过程 f：指定文件名

示例：打包并压缩/usr/tmp 下的所有文件 压缩后的压缩包指定名称为xxx.tar tar -zcvf ab.tar aa.txt bb.txt 或：tar -zcvf ab.tar *

4.2 解压 命令：tar [-zxvf] 压缩文件
其中：x：代表解压 示例：将/usr/tmp 下的ab.tar解压到当前目录下

示例：将/usr/tmp 下的ab.tar解压到根目录/usr下 tar -xvf ab.tar -C /usr------C代表指定解压的位置

五、查找命令 5.1 grep grep命令是一种强大的文本搜索工具

使用实例：

ps -ef | grep sshd 查找指定ssh服务进程 ps -ef | grep sshd | grep -v grep 查找指定服务进程，排除gerp身 ps -ef | grep sshd -c 查找指定进程个数 5.2 find find命令在目录结构中搜索文件，并对搜索结果执行指定的操作。

find 默认搜索当前目录及其子目录，并且不过滤任何结果（也就是返回所有文件），将它们全都显示在屏幕上。

使用实例：

find . -name "*.log" -ls* 在当前目录查找以*.log*结尾的文件，并显示详细信息。 *find /root/ -perm 600* 查找*/root/*目录下权限为*600*的文件 *find . -type f -name* ".log" 查找当目录，以.log结尾的普通文件 find . -type d | sort 查找当前所有目录并排序 find . -size +100M 查找当前目录大于100M的文件 5.3 locate locate 让使用者可以很快速的搜寻某个路径。默认每天自动更新一次，所以使用locate 命令查不到最新变动过的文件。为了避免这种情况，可以在使用locate之前，先使用updatedb命令，手动更新数据库。如果数据库中没有查询的数据，则会报出locate: can not stat () `/var/lib/mlocate/mlocate.db': No such file or directory该错误！updatedb即可！

yum -y install mlocate 如果是精简版CentOS系统需要安装locate命令

使用实例：

updatedb locate /etc/sh 搜索etc目录下所有以sh开头的文件 locate pwd 查找和pwd相关的所有文件 5.4 whereis whereis命令是定位可执行文件、源代码文件、帮助文件在文件系统中的位置。这些文件的属性应属于原始代码，二进制文件，或是帮助文件。

使用实例：

whereis ls 将和ls文件相关的文件都查找出来 5.5 which which命令的作用是在PATH变量指定的路径中，搜索某个系统命令的位置，并且返回第一个搜索结果。

使用实例：

which pwd 查找pwd命令所在路径 which java 查找path中java的路径 六、su、sudo 6.1 su su用于用户之间的切换。但是切换前的用户依然保持登录状态。如果是root 向普通或虚拟用户切换不需要密码，反之普通用户切换到其它任何用户都需要密码验证。

su test:切换到test用户，但是路径还是/root目录 su - test : 切换到test用户，路径变成了/home/test su : 切换到root用户，但是路径还是原来的路径 su - : 切换到root用户，并且路径是/root su不足：如果某个用户需要使用root权限、则必须要把root密码告诉此用户。

退出返回之前的用户：exit

6.2 sudo sudo是为所有想使用root权限的普通用户设计的。可以让普通用户具有临时使用root权限的权利。只需输入自己账户的密码即可。

进入sudo配置文件命令：

vi /etc/sudoer或者visudo 案例： 允许hadoop用户以root身份执行各种应用命令，需要输入hadoop用户的密码。 hadoop ALL=(ALL) ALL

案例： 只允许hadoop用户以root身份执行ls 、cat命令，并且执行时候免输入密码。 配置文件中： hadoop ALL=NOPASSWD: /bin/ls, /bin/cat 七、系统服务 service iptables status --查看iptables服务的状态 service iptables start --开启iptables服务 service iptables stop --停止iptables服务 service iptables restart --重启iptables服务

chkconfig iptables off --关闭iptables服务的开机自启动 chkconfig iptables on --开启iptables服务的开机自启动 八、网络管理 8.1 主机名配置 [root@node1 ~]# vi /etc/sysconfig/network NETWORKING=yes HOSTNAME=node1 8.2 IP 地址配置 [root@node1 ~]# vi /etc/sysconfig/network-scripts/ifcfg-eth0 8.3 域名映射 /etc/hosts文件用于在通过主机名进行访问时做ip地址解析之用。所以，你想访问一个什么样的主机名，就需要把这个主机名和它对应的ip地址。

[root@node1 ~]# vi /etc/hosts

192.168.52.201 node1 192.168.52.202 node2 192.168.52.203 node3 九、定时任务指令crontab 配置 crontab是Unix和Linux用于设置定时任务的指令。通过crontab命令，可以在固定间隔时间,执行指定的系统指令或shell脚本。时间间隔的单位可以是分钟、小时、日、月、周及以上的任意组合。

crontab安装：

yum install crontabs 服务操作说明：

service crond start ## 启动服务 service crond stop ## 关闭服务 service crond restart ## 重启服务 9.1 命令格式 crontab [-u user] file

crontab [-u user] [-e | -l | -r]

参数说明：

-u user：用来设定某个用户的crontab服务

file：file是命令文件的名字,表示将file做为crontab的任务列表文件

并载入crontab。

-e：编辑某个用户的crontab文件内容。如果不指定用户，则表示编辑当前

用户的crontab文件。

-l：显示某个用户的crontab文件内容。如果不指定用户，则表示显示当前

用户的crontab文件内容。

-r：删除定时任务配置，从/var/spool/cron目录中删除某个用户的crontab

文件，如果不指定用户，则默认删除当前用户的crontab文件。

命令示例：

crontab file [-u user] ## 用指定的文件替代目前的crontab crontab -l [-u user] ## 列出用户目前的crontab crontab -e [-u user] ## 编辑用户目前的crontab 9.2 配置说明、实例 命令：* * * * * command

解释：分 时 日 月 周 命令

第1列表示分钟1～59 每分钟用*或者 */1表示

第2列表示小时0～23（0表示0点）

第3列表示日期1～31

第4列表示月份1～12

第5列标识号星期0～6（0表示星期天）

第6列要运行的命令

配置实例：

先打开定时任务所在的文件： crontab -e

每分钟执行一次date命令 */1 * * * * date >> /root/date.txt

每晚的21:30重启apache。 30 21 * * * service httpd restart

每月1、10、22日的4 : 45重启apache。
45 4 1,10,22 * * service httpd restart

每周六、周日的1 : 10重启apache。 10 1 * * 6,0 service httpd restart

每天18 : 00至23 : 00之间每隔30分钟重启apache。 0,30 18-23 * * * service httpd restart 晚上11点到早上7点之间，每隔一小时重启apache

- 23-7/1 * * * service httpd restart 十、其他命令 10.1 查看当前目录：pwd 命令：pwd 查看当前目录路径

10.2 查看进程：ps -ef 命令：ps -ef 查看所有正在运行的进程

10.3 结束进程：kill 命令：kill pid 或者 kill -9 pid(强制杀死进程) pid:进程号

10.4 网络通信命令： ifconfig：查看网卡信息

命令：ifconfig 或 ifconfig | more

ping：查看与某台机器的连接情况

命令：ping ip

netstat -an：查看当前系统端口

命令：netstat -an

搜索指定端口 命令：netstat -an | grep 8080

10.5 配置网络 命令：setup

10.6 重启网络 命令：service network restart

10.7 切换用户 命令：su - 用户名

10.8 关闭防火墙 命令：chkconfig iptables off

或者：

iptables -L; iptables -F; service iptables stop 10.9 修改文件权限 命令：chmod 777

10.10 清屏 命令：ctrl + l

10.11 vi模式下快捷键 esc后:

保存并退出快捷键：shift+z+z

光标跳到最后一行快捷键：shift+g

删除一行：dd

复制一行内容：y+y

粘贴复制的内容：p

Linux系统下你关注过哪些内核参数，说说你知道的。

用一行命令查看文件的最后五行。

tail -n 5 文件名称

用一行命令输出正在运行的java进程。

介绍下你理解的操作系统中线程切换过程。

top 命令之后有哪些内容，有什么作用。htop?

一、top前5行统计信息 第1行：top - 05:43:27 up 4:52, 2 users, load average: 0.58, 0.41, 0.30 第1行是任务队列信息，其参数如下：

内容 含义 05:43:27 表示当前时间 up 4:52 系统运行时间 格式为时：分 2 users 当前登录用户数 load average: 0.58, 0.41, 0.30 系统负载，即任务队列的平均长度。 三个数值分别为 1分钟、5分钟、15分钟前到现在的平均值。 load average: 如果这个数除以逻辑CPU的数量，结果高于5的时候就表明系统在超负荷运转了。

第2行：Tasks: 159 total, 1 running, 158 sleeping, 0 stopped, 0 zombie 第3行：%Cpu(s): 37.0 us, 3.7 sy, 0.0 ni, 59.3 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st 第2、3行为进程和CPU的信息 当有多个CPU时，这些内容可能会超过两行，其参数如下：

内容 含义 159 total 进程总数 1 running 正在运行的进程数 158 sleeping 睡眠的进程数 0 stopped 停止的进程数 0 zombie 僵尸进程数 37.0 us 用户空间占用CPU百分比 3.7 sy 内核空间占用CPU百分比 0.0 ni 用户进程空间内改变过优先级的进程占用CPU百分比 59.3 id 空闲CPU百分比 0.0 wa 等待输入输出的CPU时间百分比 0.0 hi 硬中断（Hardware IRQ）占用CPU的百分比 0.0 si 软中断（Software Interrupts）占用CPU的百分比 0.0 st

第4行：KiB Mem: 1530752 total, 1481968 used, 48784 free, 70988 buffers 第5行：KiB Swap: 3905532 total, 267544 used, 3637988 free. 617312 cached Mem 第4、5行为内存信息 其参数如下：

内容 含义 KiB Mem: 1530752 total 物理内存总量 1481968 used 使用的物理内存总量 48784 free 空闲内存总量 70988 buffers（buff/cache） 用作内核缓存的内存量 KiB Swap: 3905532 total 交换区总量 267544 used 使用的交换区总量 3637988 free 空闲交换区总量 617312 cached Mem 缓冲的交换区总量。 3156100 avail Mem 代表可用于进程下一次分配的物理内存数量 上述最后提到的缓冲的交换区总量，这里解释一下，所谓缓冲的交换区总量，即内存中的内容被换出到交换区，而后又被换入到内存，但使用过的交换区尚未被覆盖，该数值即为这些内容已存在于内存中的交换区的大小。相应的内存再次被换出时可不必再对交换区写入。

计算可用内存数有一个近似的公式： 第四行的free + 第四行的buffers + 第五行的cached

二、进程信息 列名 含义 PID 进程id PPID 父进程id RUSER Real user name UID 进程所有者的用户id USER 进程所有者的用户名 GROUP 进程所有者的组名 TTY 启动进程的终端名。不是从终端启动的进程则显示为 ? PR 优先级 NI nice值。负值表示高优先级，正值表示低优先级 P 最后使用的CPU，仅在多CPU环境下有意义 %CPU 上次更新到现在的CPU时间占用百分比 TIME 进程使用的CPU时间总计，单位秒 TIME+ 进程使用的CPU时间总计，单位1/100秒 %MEM 进程使用的物理内存百分比 VIRT 进程使用的虚拟内存总量，单位kb。VIRT=SWAP+RES SWAP 进程使用的虚拟内存中，被换出的大小，单位kb RES 进程使用的、未被换出的物理内存大小，单位kb。RES=CODE+DATA CODE 可执行代码占用的物理内存大小，单位kb DATA 可执行代码以外的部分(数据段+栈)占用的物理内存大小，单位kb SHR 共享内存大小，单位kb nFLT 页面错误次数 nDRT 最后一次写入到现在，被修改过的页面数。 S 进程状态。D=不可中断的睡眠状态 R=运行 S=睡眠 T=跟踪/停止 Z=僵尸进程 COMMAND 命令名/命令行 WCHAN 若该进程在睡眠，则显示睡眠中的系统函数名 Flags 任务标志

线上CPU爆高，请问你如何找到问题所在。

出现CPU负载过高的原因

程序陷入死循环 线程死锁，相互等待，导致假死状态 现象模拟

解决步骤

top 命令查看当前系统负载信息 top -H -p pid 查看指定进程中每个线程的资源占用情况 jstack pid > ./dump.log 将指定进程中线程的堆栈信息输出到文件 利用MAT工具分析内存占用

系统设计

电商系统中，如何实现秒杀功能？如何解决商品的超卖问题？ 困难

如何设计一个线程池 中等

简述 Zookeeper 基础原理以及使用场景 中等

简述什么是两阶段提交?

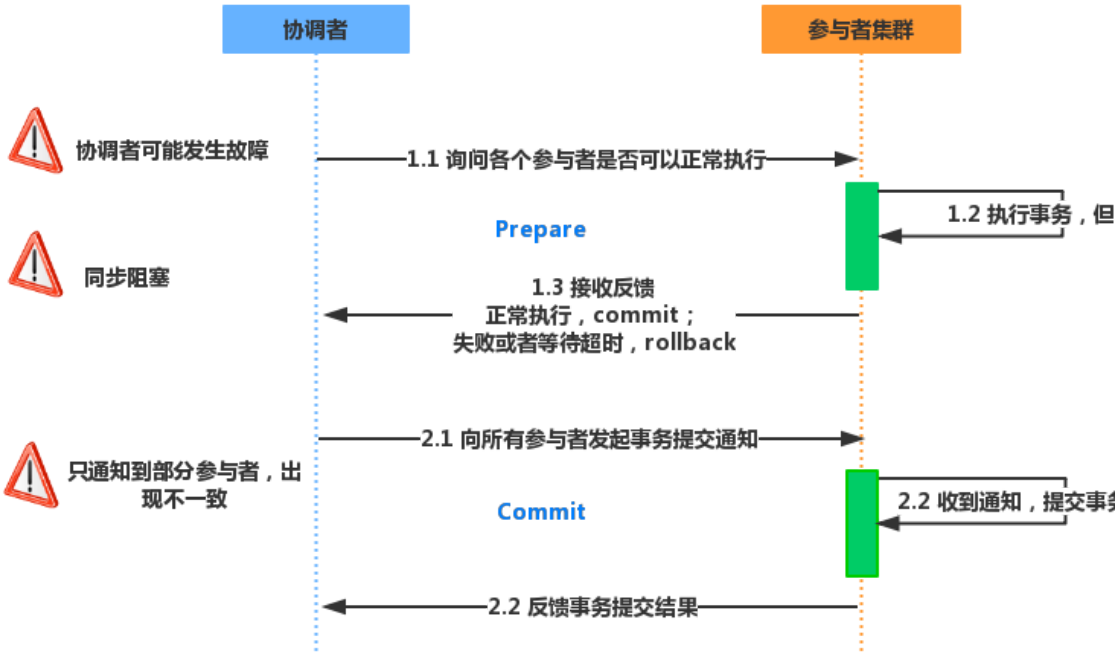
简单

两阶段提交协议是协调所有分布式原子事务参与者，并决定提交或取消（回滚）的分布式算法。

1.协议参与者

在两阶段提交协议中，系统一般包含两类机器（或节点）：一类为协调者（coordinator），通常一个系统中只有一个；另一类为事务参与者（participants，cohorts或workers），一般包含多个，在数据存储系统中可以理解为数据副本的个数。协议中假设每个节点都会记录写前日志（write-ahead log）并持久性存储，即使节点发生故障日志也不会丢失。协议中同时假设节点不会发生永久性故障而且任意两个

节点都可以互相通信。



2.两个阶段的执行

1.请求阶段（commit-request phase，或称表决阶段，voting phase） 在请求阶段，协调者将通知事务参与者准备提交或取消事务，然后进入表决过程。 在表决过程中，参与者将告知协调者自己的决策：同意（事务参与者本地作业执行成功）或取消（本地作业执行故障）。

2.提交阶段（commit phase） 在该阶段，协调者将基于第一个阶段的投票结果进行决策：提交或取消。 当且仅当所有的参与者同意提交事务协调者才通知所有的参与者提交事务，否则协调者将通知所有的参与者取消事务。 参与者在接收到协调者发来的消息后将执行响应的操作。

（3）两阶段提交的缺点

1.同步阻塞问题。执行过程中，所有参与节点都是事务阻塞型的。 当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。

2.单点故障。由于协调者的重要性，一旦协调者发生故障。 参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）

3.数据不一致。在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了commit请求。 而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据部一致性的现象。

（4）两阶段提交无法解决的问题

当协调者出错，同时参与者也出错时，两阶段无法保证事务执行的完整性。 考虑协调者再发出commit消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。 那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

什么是设计模式，描述几个常用的设计模式 中等

简述 CAP 理论 简单

一致性（C）

一致性是指“all nodes see the same data at the same time”，即更新操作成功后，所有节点在同一时间的数据完全一致。

一致性可以分为客户端和服务端两个不同的视角：

- 从客户端角度来看，一致性主要指多个用户并发访问时更新的数据如何被其他用户获取的问题；
- 从服务端来看，一致性则是用户进行数据更新时如何将数据复制到整个系统，以保证数据的一致。

一致性是在并发读写时才会出现的问题，因此在理解一致性的问题时，一定要注意结合考虑并发读写的场景。

可用性（A）

可用性是指“reads and writes always succeed”，即用户访问数据时，系统是否能在正常响应时间返回结果。

好的可用性主要是指系统能够很好地为用户服务，不出现用户操作失败或者访问超时等用户体验不好的情况。在通常情况下，可用性与分布式数据冗余、负载均衡等有着很大的关联。

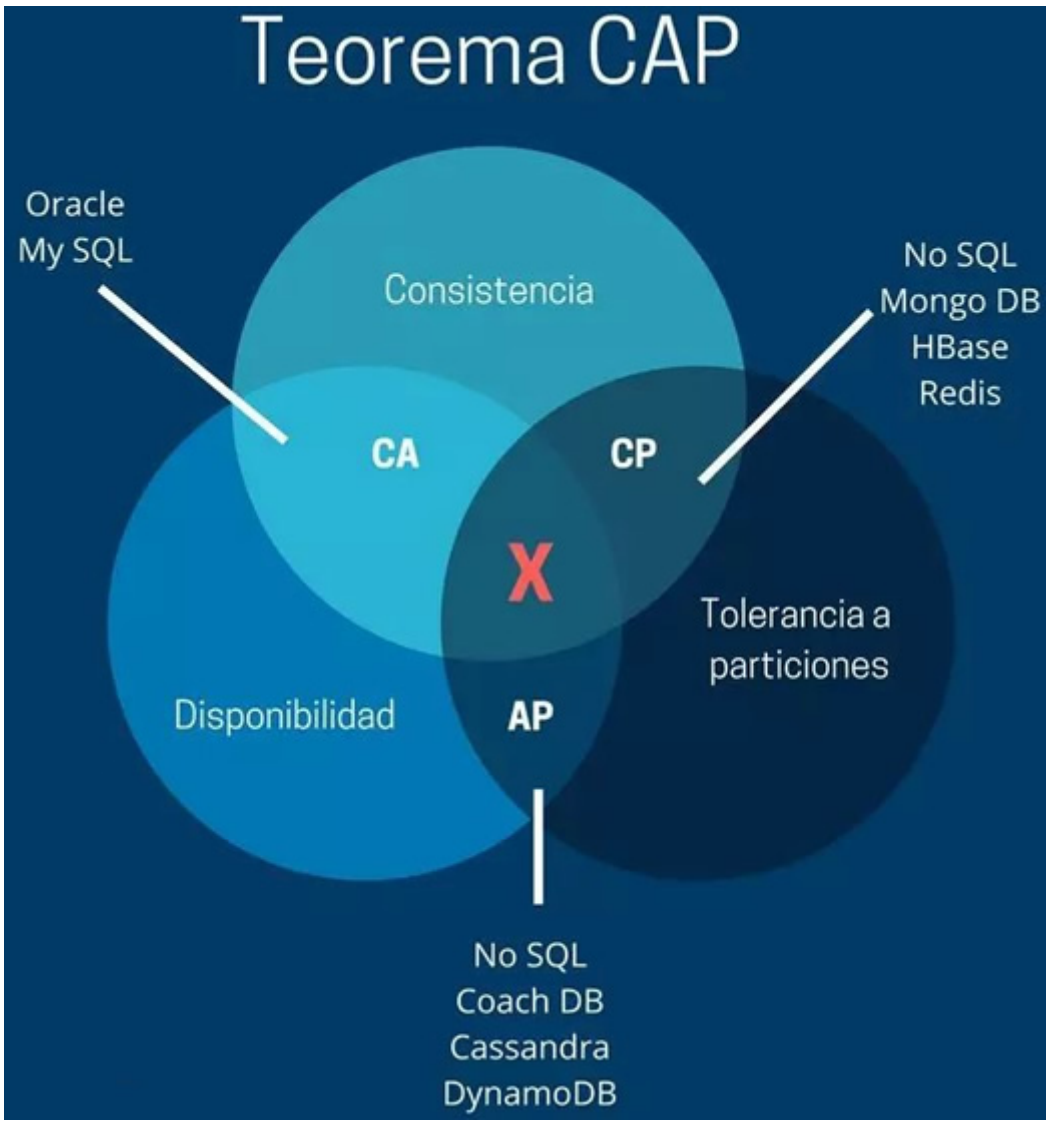
分区容错性（P）

分区容错性是指“the system continues to operate despite arbitrary message loss or failure of part of the system”，即分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。

分区容错性和扩展性紧密相关。在分布式应用中，可能因为一些分布式的原因导致系统无法正常运转。分区容错性高指在部分节点故障或出现丢包的情况下，集群系统仍然能提供服务，完成数据的访问。分区容错可视为在系统中采用多副本策略。

相互关系

CAP 理论认为分布式系统只能兼顾其中的两个特性，即出现 CA、CP、AP 三种情况，如图所示。



CA without P

如果不要求 Partition Tolerance，即不允许分区，则强一致性和可用性是可以保证的。其实分区是始终存在的问题，因此 CA 的分布式系统更多的是允许分区后各子系统依然保持 CA。

CP without A

如果不要求可用性，相当于每个请求都需要在各服务器之间强一致，而分区容错性会导致同步时间无限延长，如此 CP 也是可以保证的。很多传统的数据库分布式事务都属于这种模式。

AP without C

如果要可用性高并允许分区，则需放弃一致性。一旦分区发生，节点之间可能会失去联系，为了实现高可用，每个节点只能用本地数据提供服务，而这样会导致全局数据的不一致性。

总结

在实践中，可根据实际情况进行权衡，或者在软件层面提供配置方式，由用户决定如何选择 CAP 策略。

CAP 理论可用在不同的层面，可以根据 CAP 原理定制局部的设计策略，例如，在分布式系统中，每个节点自身的数据是能保证 CA 的，但在整体上又要兼顾 AP 或 CP。

如何设计一个消息队列 中等

假如明天是活动高峰？QPS 预计会翻10倍，你要怎么做？ 中等

简述一致性哈希算法的实现方式及原理 困难

算法

53. 最大子序和 简单

贪心

```
func maxSubArray(nums []int) int {
    if len(nums) == 1{
        return nums[0]
    }
    var currSum, maxSum = 0, nums[0]
    for _, v := range nums {
        if currSum > 0 {
            currSum += v
        } else {
            currSum = v
        }
        if maxSum < currSum {
            maxSum = currSum
        }
    }
    return maxSum
}

```

动态

```
func maxSubArray(nums []int) int {
    max_sum := nums[0]
    for i := 1; i < len(nums); i++ {
        if nums[i - 1] > 0 {
           nums[i] += nums[i - 1]
        }
        if nums[i] > max_sum {
            max_sum = nums[i]
        }
    }
    return max_sum
}

```

10亿个数中如何高效地找到最大的一个数以及最大的第 K 个数

困难 参考1

方法一、先拿10000个数建堆，然后一次添加剩余元素，如果大于堆顶的数（10000中最小的），将这个数替换堆顶，并调整结构使之仍然是一个最小堆，这样，遍历完后，堆中的10000个数就是所需的最大的10000个。建堆时间复杂度是O（mlogm），算法的时间复杂度为O（nmlogm）（n为10亿，m为10000）。

方法二(优化的方法)：可以把所有10亿个数据分组存放，比如分别放在1000个文件中。这样处理就可以分别在每个文件的10^6个数据中找出最大的10000个数，合并到一起在再找出最终的结果。

70. 爬楼梯 简单

```
func climbStairs(n int) int {
    sli := []int{1, 2, 3}
    if n <= 3 {
        return sli[n-1]
    }
    for i := 3; i < n; i++ {
        sli = append(sli, sli[i-1]+sli[i-2])
    }
    return sli[len(sli)-1]
}
```

△ 23次21. 合并两个有序链表 简单

```
func mergeTwoLists(l1 *ListNode, l2 *ListNode) *ListNode {
    if l1 == nil {
        return l2
    }
    if l2 == nil {
        return l1
    }
    if l1.Val > l2.Val {
        l2.Next = mergeTwoLists(l1, l2.Next)
        return l2
    } else {
        l1.Next = mergeTwoLists(l1.Next, l2)
        return l1
    }
}

```

△ 22次470. 用 Rand7() 实现 Rand10() 中等

```
func rand10() int {
     for {
        r1 := rand7()
        r2 := rand7()
        num := r1 + (r2 - 1) * 7
        if num <= 40 {
            return num % 10 + 1
        }
    }
}

```

△ 21次AVL 树和红黑树有什么区别?

简单 参考1

add by zhj: AVL树和红黑树都是平衡二叉树，虽然AVL树是最早发明的平衡二叉树，但直接把平衡二叉树等价于AVL树，我认为非常不合适。

但很多地方都在这么用。两者的比较如下

平衡二叉树类型	平衡度	调整频率	适用场景
AVL树	高	高	查询多，增/删少
红黑树	低	低	增/删频繁

△ 21次使用递归及非递归两种方式实现快速排序 中等 参考1 参考2

△ 20次如何通过一个不均匀的硬币得到公平的结果？ 简单 参考1

△ 19次10亿条数据包括 id，上线时间，下线时间，请绘制每一秒在线人数的曲线图 中等

△ 19次给定一个包含 40 亿个无符号整数的大型文件，使用最多 1G 内存，对此文件进行排序 困难 参考1

△ 19次83. 删除排序链表中的重复元素 简单

△ 18次112. 路径总和 简单

△ 17次215. 数组中的第K个最大元素 中等

△ 16次有序链表插入的时间复杂度是多少？ 简单

△ 16次146. LRU 缓存机制 中等

△ 15次哈希表常见操作的时间复杂度是多少？遇到哈希冲突是如何解决的？ 简单 参考1

△ 14次141. 环形链表 简单

△ 13次232. 用栈实现队列 简单

△ 12次常用的限流算法有哪些？简述令牌桶算法原理 中等 参考1 参考2

△ 12次189. 旋转数组 中等

△ 12次4. 寻找两个正序数组的中位数 困难

△ 12次简述你熟悉的几个排序算法以及优缺点 中等 参考1 参考2

△ 11次简述常见的负载均衡算法 简单 参考1

△ 11次300. 最长递增子序列 中等

△ 11次64 匹马，8 个赛道，找出前 4 匹马最少需要比几次 困难 参考1

△ 9次25. K 个一组翻转链表 困难

△ 9次5. 最长回文子串 中等

```
package main

import "fmt"

func longestPalindrome(s string) string {
   // 空字符串或长度为1的字符串肯定是回文字符串，直接返回即可
   if len(s) < 2 {
      return s
   }
   // 初始化最长回文子串：取首字符构成的子串作为最长回文子串
   maxSubPalindrome := s[:1]
   // 设置两个游标变量，i从头到尾-1依次遍历，将每个字符作为子串的首字符下标，
   // j作为子串的尾字符下标，依次检查s[i:j+1]是否为最长回文字符串
   for i := 0; i < len(s)-1; i++ {
      for j := i + 1; j < len(s); j++ {
         if j-i+1 > len(maxSubPalindrome) && isPalindrome(s[i:j+1]) {
            maxSubPalindrome = s[i : j+1]
         }
      }
   }

   return maxSubPalindrome
}

// 判断某个子串是否是回文字符串
func isPalindrome(subStr string) bool {
   for i := 0; i < len(subStr)/2; i++ {
      if subStr[len(subStr)-i-1] != subStr[i] {
         return false
      }
   }
   return true
}
// 自测用例
func main() {
   fmt.Println(longestPalindrome("a"))
}
```

△ 9次125. 验证回文串 简单

```
func isPalindrome(s string) bool {
    sB:=[]byte(s)
    m:=0
    for i:=0;i<len(sB);i++{
        sB[i]|=32//位运算将大写转换为小写
        //过滤符合条件的小写字母和数字，将其前移
        if sB[i]>=97&&sB[i]<=122||sB[i]>=48&&sB[i]<=57{
            sB[m]=sB[i]
```

```
            m++//m为操作后有效长度尾部
        }
    }
    l,r:=0,m-1//双指针回文比较
    for l<r{
        if sB[l]!=sB[r]{
            return false
        }
        l++
        r--
    }
    return true
}
```

△ 9次23. 合并K个升序链表 困难

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func mergeKLists(lists []*ListNode) *ListNode {
    n := len(lists)
    switch n {
    case 0:
        return nil
    case 1:
        return lists[0]
    case 2:
        //针对两个链表进行归并排序
        return merge(lists[0], lists[1])
    default:
        key := n / 2
        //数组拆分,使下一次递归的lists的长度=2

        //优化思路: mergeKLists(lists[:key]),使用Goroutine+channel进行并发合并(归并排序的特点)
        return mergeKLists([]*ListNode{mergeKLists(lists[:key]), mergeKLists(lists[key:])})
    }

}

//merge 对两个有序链表进行归并排序
func merge(left *ListNode, right *ListNode) *ListNode {
    //head: 新的链表的head指针,保持不变
    //tail: 新链表的尾指针
    var head, tail *ListNode

    if left == nil {
        return right
    }

    if right == nil {
        return left
```

```
    }

    if left.Val < right.Val {
        head, tail, left = left, left, left.Next
    } else {
        head, tail, right = right, right, right.Next
    }

    //循环,直到某一个链表已遍历完
    for left != nil && right != nil {
        //找到下一个节点,添加到新链表的尾
        if left.Val < right.Val {
            tail.Next, left = left, left.Next
        } else {
            tail.Next, right = right, right.Next
        }
        //更新tail
        tail = tail.Next
    }

    //剩下的节点字节拼接到新链表尾部
    if left != nil {
        tail.Next = left
    }
    if right != nil {
        tail.Next = right
    }

    return head
}

```

△ 8次面试题 04.04. 检查平衡性 简单

```
func isBalanced(root *TreeNode) bool {
    if root == nil {
        return true
    }
    flag := true
    run(root,&flag)

    return flag
}

func run(root *TreeNode,flag *bool) int {
    if *flag == false {
        return 0
    }

    if root == nil {
        return 1
    }


    l := run(root.Left,flag)
    r := run(root.Right,flag)

```

```
    if abs(l-r) > 1 {
        *flag = false
    }

    if l > r {
        return l+1
    }
    return r+1
}

func abs(a int) int {
    if a < 0 {
        a *= -1
    }
    return a
}

```

k8s&docker

非技术

△ 19次 对加班有什么看法?

△ 15次 你的优势和劣势是什么?

△ 11次 与同事沟通的时候，如果遇到冲突了如何解决?

△ 11次 项目中最难的地方是哪里？你学习到了什么?

△ 11次 最近在看什么书吗，有没有接触过什么新技术?

△ 9次 未来的职业规划是什么?

△ 5次 最近一年内遇到的最有挑战的事情是什么?

前端

△ 31次Vue 中双向数据绑定的实现原理是怎样的？ 中等 参考1 参考2

△ 20次简述 Javascript 原型以及原型链 中等 参考1

△ 17次什么是闭包，什么是立即执行函数，它的作用是什么？简单说一下闭包的使用场景 简单 参考1

△ 17次简述虚拟 dom 实现原理 简单

△ 16次简述浏览器的缓存机制 中等

△ 15次简述 diff 算法的实现机制和使用场景 中等

△ 15次简述 Vue 的生命周期 中等

△ 15次CSS 实现三列布局 简单

△ 15次手写题库 https://github.com/Mayandev/fe-interview-handwrite 困难

△ 15次深拷贝与浅拷贝区别是什么？ 简单

△ 14次简述强缓存与协商缓存的区别和使用场景 中等

△ 13次简述图片的懒加载原理 简单

△ 13次简述浏览器的渲染过程，重绘和重排在渲染过程中的哪一部分？ 中等

△ 12次简述 ES6 的新特性 简单

△ 11次简述 Javascript 中的防抖与节流的原理并尝试实现 中等

△ 11次简述 Flex 布局的原理和使用场景 中等

△ 11次简述 Vue 和 React 的区别 中等

△ 10次简述 JavaScript 事件循环机制 中等 参考1

△ 10次箭头函数和普通函数的区别是什么？ 简单

△ 10次async 和 defer 有什么区别？ 简单

△ 10次CSS 的选择器优先级是怎样？ 简单

△ 9次简述 Javascript 中 this 的指向有哪些 简单

△ 9次MVC 模型和 MVVM 模型的区别 中等

△ 9次简述常见异步编程方案 (promise, generator, async) 的原理 中等

△ 9次简述 Javascript 事件冒泡和事件捕获原理

△ 9次什么是跨域，什么情况下会发生跨域请求？ 中等 参考1

△ 8次localstorage 与 cookie 的区别是什么？ 简单

△ 8次简述 CSS 盒模型 简单 参考1

△ 8次什么是 DOM 事件流?

△ 8次CSS 的伪类和伪元素的区别是什么？ 简单

CI、CD

请简述devops是什么?

答:DevOps是一种实现Dev(开发)与Ops(运维)工作流有效联合的思想。

简述集中式版本控制系统与分布式版本控制系统的区别?

答:集中式控制系统,必须联网才能工作,如果中央服务器挂了那么就完蛋了 分布式版本控制系统可以不用联网工作,因为参与开发的每人都有一个版本库, 并且分布式的没有中央服务器,可靠性高,也可以有中央服务器但是只是用来合 大家修改的代码.

请简述git本地仓库有哪三大区, 中间的那个区主要有什么作用?

答:工作区和暂存区和本地仓库,中间的区主要用来暂存你修改或更新的代码, 但是还不是最终的结果,又不想提交到版本库里,因为修改一点就提交到版本库 那样会造成版本库很乱,不方便管理,并且提交到版本库里之后删除不了.所以可以吧 修改好但又不确定是不是最终的修改的都可以先放到暂存区.

请说明下面git命令的作用

```
git init
答:初始化git仓库

git add *                    答:把文件或修改的代码添加到暂存区

git rm xxx                   答:删除暂存区的代码或文件

git status                   答:查看工作区里有无修改或新添加的代码文件还未添加到暂存区

git commit -m "xxx"          答:把暂存区的代码或文件提交到版本库

git checkout -- xxx          答:撤销修改

git reset HEAD               答:回退版本

git log                      答:查看提交的历史版本信息

git reflog                   答:查看所有的操作历史。

git reset --hard xxx         答:还原版本

git branch                   答:查看有多少个分支默认只要一个master分支

git branch xxx               答:创建自定义的分支

git checkout xxx             答:切换自定义的分支

git merge xxx                答:合并分支里的新内容
```

请简述github与gitlab分别是什么，各自的应用场景

```
答:github是开源项目代码托管平台 gitlab是一个开源的版本管理系统

github提供了公有仓库和私有仓库但是私有仓库是要付费的.可通过web界面访问 可以实现远程代码管理,跨区域管理

gitlab是一个自托管的git项目仓库可通过web界面访问,他和github应用场景相似, 可以实现远程代码管理,
跨区域管理
```

1. 什么是DevOps 用最简单的术语来说，DevOps是产品开发过程中开发（Dev）和运营（Ops）团队之间的灰色区域。DevOps是一种在产品开发周期中强调沟通，集成和协作的文化。因此，它消除了软件开发团队和运营团队之间的孤岛，使他们能够快速，连续地集成和部署产品。
2. 什么是持续集成 持续集成（Continuous integration，缩写为 CI）是一种软件开发实践，团队开发成员经常集成他们的工作。利用自动测试来验证并断言其代码不会与现有代码库产生冲突。理想情况下，代码更改应该每天在CI工具的帮助下，在每次提交时进行自动化构建（包括编译，发布，自动化测试），从而尽早地发现集成错误，以确保合并的代码没有破坏主分支。

3. 什么是持续交付 持续交付（Continuous delivery，缩写为 CD）以及持续集成为交付代码包提供了完整的流程。在此阶段，将使用自动构建工具来编译工件，并使其准备好交付给最终用户。它的目标在于让软件的构建、测试与发布变得更快以及更频繁。这种方式可以减少软件开发的成本与时间，减少风险。

4. 什么是持续部署 持续部署（Continuous deployment）通过集成新的代码更改并将其自动交付到发布分支，从而将持续交付提升到一个新的水平。更具体地说，一旦更新通过了生产流程的所有阶段，便将它们直接部署到最终用户，而无需人工干预。因此，要成功利用连续部署，软件工件必须先经过严格建立的自动化测试和工具，然后才能部署到生产环境中。

5. 什么是持续测试及其好处 连续测试是一种在软件交付管道中尽早、逐步和适当地应用自动化测试的实践。在典型的CI/CD工作流程中，将小批量发布构建。因此，为每个交付手动执行测试用例是不切实际的。自动化的连续测试消除了手动步骤，并将其转变为自动化例程，从而减少了人工。因此，对于DevOps文化而言，自动连续测试至关重要。 持续测试的好处

    - 确保构建的质量和速度。
    - 支持更快的软件交付和持续的反馈机制。
    - 一旦系统中出现错误，请立即检测。
    - 降低业务风险。在潜在问题变成实际问题之前进行评估。

6. 什么是版本控制及其用途？ 版本控制（或源代码控制）是一个存储库，源代码中的所有更改都始终存储在这个代码仓库中。版本控件提供了代码开发的操作历史记录，追踪文件的变更内容、时间、人等信息忠实地了记录下来。版本控制是持续集成和持续构建的源头。

7. 什么是Git？ Git是一个分布式版本控制系统，可跟踪代码存储库中的更改。利用GitHub流，Git围绕着一个基于分支的工作流，该工作流随着团队项目的不断发展而简化了团队协作。

实施DevOps的原因

1. DevOps为什么重要？DevOps如何使团队在软件交付方面受益？ 在当今的数字化世界中，组织必须重塑其产品部署系统，使其更强大，更灵活，以跟上竞争的步伐。 这就是DevOps概念出现的地方。DevOps在为整个软件开发管道（从构思到部署，再到最终用户）产生移动性和敏捷性方面发挥着至关重要的作用。DevOps是将不断更新和改进产品的更简化，更高效的流程整合在一起的解决方案。
2. 解释DevOps对开发人员有何帮助 在没有DevOps的世界中，开发人员的工作流程将首先建立新代码，交付并集成它们，然后，操作团队有责任打包和部署代码。之后，他们将不得不等待反馈。而且如果出现问题，由于错误，他们将不得不重新执行一次，在项目中涉及的不同团队之间的无数手动沟通。 由于CI/CD实践已经合并并自动化了其余任务，因此应用DevOps可以将开发人员的任务简化为仅构建代码。随着流程变得更加透明并且所有团队成员都可以访问，将工程团队和运营团队相结合有助于建立更好的沟通和协作。
3. 为什么DevOps最近在软件交付方面变得越来越流行？ DevOps在过去几年中受到关注，主要是因为它能够简化组织运营的开发，测试和部署流程，并将其转化为业务价值。 技术发展迅速。因此，组织必须采用一种新的工作流程-DevOps和Agile方法-来简化和刺激其运营，而不能落后于其他公司。DevOps的功能通过Facebook和Netflix的持续部署方法所取得的成功得到了清晰体现，该方法成功地促进了其增长，而没有中断正在进行的运营。
4. CI/CD有什么好处？ CI和CD的结合将所有代码更改统一到一个单一的存储库中，并通过自动化测试运行它们，从而在所有阶段全面开发产品，并随时准备部署。 CI/CD使组织能够按照客户期望的那样快速，高效和自动地推出产品更新。 简而言之，精心规划和执行良好的CI/CD管道可加快发布速度和可靠性，同时减轻产品的代码更改和缺陷。这最终将导致更高的客户满意度。
5. 持续交付有什么好处？ 通过手动发布代码更改，团队可以完全控制产品。在某些情况下，该产品的新版本将更有希望：具有明确业务目的的促销策略。 通过自动执行重复性和平凡的任务，IT专业人员可以拥有更多的思考能力来专注于改进产品，而不必担心集成进度。
6. 持续部署有哪些好处？ 通过持续部署，开发人员可以完全专注于产品，因为他们在管道中的最后任务是审查拉取请求并将其合并到分支。通过在自动测试后立即发布新功能和修复，此方法可实现快速部署并缩短部署持续时间。 客户将是评估每个版本质量的人。新版本的错误修复更易于处理，因为现在每个版本都以小批量交付。

如何有效实施DevOps

1. 定义典型的DevOps工作流程 典型的DevOps工作流程可以简化为4个阶段：
    - 版本控制：这是存储和管理源代码的阶段。版本控件包含代码的不同版本。
    - 持续集成：在这一步中，开发人员开始构建组件，并对其进行编译，验证，然后通过代码审查，单元测试和集成测试进行测试。
    - 持续交付：这是持续集成的下一个层次，其中发布和测试过程是完全自动化的。CD确保将新版本快速，可持续地交付给最终用户。
    - 持续部署：应用程序成功通过所有测试要求后，将自动部署到生产服务器上以进行发布，而无需任何人工干预。
2. DevOps的核心操作是什么？ DevOps在开发和基础架构方面的核心运营是 Software development: Infrastructure:
    - Provisioning
    - Configuration
    - Orchestration
    - Deployment
    - Code building
    - Code coverage
    - Unit testing
    - Packaging
    - Deployment
3. 在实施DevOps之前，团队需要考虑哪些预防措施？ 当组织尝试应用这种新方法时，对DevOps做法存在一些误解，有可能导致悲惨的失败：
    - DevOps不仅仅是简单地应用新工具和/或组建新的“部门”并期望它能正常工作。实际上，DevOps被认为是一种文化，开发团队和运营团队遵循共同的框架。
    - 企业没有为其DevOps实践定义清晰的愿景。对开发团队和运营团队而言，应用DevOps计划是一项显着的变化。因此，拥有明确的路线图，将DevOps集成到你的组织中的目标和期望将消除任何混乱，并从早期就提供清晰的指导方针。
    - 在整个组织中应用DevOps做法之后，管理团队需要建立持续的学习和改进文化。系统中的故障和问题应被视为团队从错误中学习并防止这些错误再次发生的宝贵媒介。
4. SCM团队在DevOps中扮演什么角色？ 软件配置管理（SCM）是跟踪和保留开发环境记录的实践，包括在操作系统中进行的所有更改和调整。 在DevOps中，将SCM作为代码构建在基础架构即代码实践的保护下。 SCM为开发人员简化了任务，因为他们不再需要手动管理配置过程。现在，此过程以机器可读的形式构建，并且会自动复制和标准化。
5. 质量保证（QA）团队在DevOps中扮演什么角色？ 随着DevOps实践在创新组织中变得越来越受欢迎，QA团队的职责和相关性在当今的自动化世界中已显示出下降的迹象。 但是，这可以被认为是神话。DevOps的增加并不等于QA角色的结束。这仅意味着他们的工作环境和所需的专业知识正在发生变化。因此，他们的主要重点是专业发展以跟上这种不断变化的趋势。 在DevOps中，质量保证团队在确保连续交付实践的稳定性以及执行自动重复性测试无法完成的探索性测试任务方面发挥战略作用。他们在评估测试和检测最有价值的测试方面的见识仍然在缓解发布的最后步骤中的错误方面起着至关重要的作用。
6. DevOps使用哪些工具？描述你使用任何这些工具的经验 在典型的DevOps生命周期中，有不同的工具来支持产品开发的不同阶段。因此，用于DevOps的最常用工具可以分为6个关键阶段： 持续开发：Git, SVN, Mercurial, CVS, Jira 持续整合：Jenkins, Bamboo, CircleCI 持续交付：Nexus, Archiva, Tomcat 持续部署：Puppet, Chef, Docker 持续监控：Splunk, ELK Stack, Nagios 连续测试：Selenium，Katalon Studio
7. 如何在DevOps实践中进行变更管理 典型的变更管理方法需要与DevOps的现代实践适当集成。第一步是将变更集中到一个平台中，以简化变更，问题和事件管理流程。 接下来，企业应建立高透明度标准，以确保每个人都在同一页面上，并确保内部信息和沟通的准确性。 对即将到来的变更进行分层并建立可靠的策略，将有助于最大程度地降低风险并缩短变更周期。最后，组织应将自动化应

用到其流程中，并与DevOps软件集成。

如何有效实施CI/CD

1. CI/CD的一些核心组件是什么？ 稳定的CI/CD管道需要用作版本控制系统的存储库管理工具。这样开发人员就可以跟踪软件版本中的更改。 在版本控制系统中，开发人员还可以在项目上进行协作，在版本之间进行比较并消除他们犯的任何错误，从而减轻对所有团队成员的干扰。 连续测试和自动化测试是成功建立无缝CI / CD管道的两个最关键的关键。自动化测试必须集成到所有产品开发阶段（包括单元测试，集成测试和系统测试），以涵盖所有功能，例如性能，可用性，性能，负载，压力和安全性。
2. CI/CD的一些常见做法是什么？ 以下是建立有效的CI / CD管道的一些最佳实践：
    - 发展DevOps文化
    - 实施和利用持续集成
    - 以相同的方式部署到每个环境
    - 失败并重新启动管道
    - 应用版本控制
    - 将数据库包含在管道中
    - 监控你的持续交付流程
    - 使你的CD流水线流畅
3. 什么时候是实施CI/CD的最佳时间？ 向DevOps的过渡需要彻底重塑其软件开发文化，包括工作流，组织结构和基础架构。因此，组织必须为实施DevOps的重大变化做好准备。
4. 有哪些常见的CI/CD服务器 Visual Studio Visual Studio支持具有敏捷计划，源代码控制，包管理，测试和发布自动化以及持续监视的完整开发的DevOps系统。 TeamCity TeamCity是一款智能CI服务器，可提供框架支持和代码覆盖，而无需安装任何额外的插件，也无需模块来构建脚本。 Jenkins 它是一个独立的CI服务器，通过共享管道和错误跟踪功能支持开发和运营团队之间的协作。它也可以与数百个仪表板插件结合使用。 GitLab GitLab的用户可以自定义平台，以进行有效的持续集成和部署。GitLab帮助CI / CD团队加快代码交付，错误识别和恢复程序的速度。 Bamboo Bamboo是用于产品发布管理自动化的连续集成服务器。Bamboo跟踪所有工具上的所有部署，并实时传达错误。
5. 描述持续集成的有效工作流程 实施持续集成的成功工作流程包括以下实践：
    - 实施和维护项目源代码的存储库
    - 自动化构建和集成
    - 使构建自检
    - 每天将更改提交到基准
    - 构建所有添加到基准的提交
    - 保持快速构建
    - 在生产环境的克隆中运行测试
    - 轻松获取最新交付物
    - 使构建结果易于所有人监视
    - 自动化部署

每种术语之间的差异

1. 敏捷和DevOps之间有哪些主要区别？ 基本上，DevOps和敏捷是相互补充的。敏捷更加关注开发新软件和以更有效的方式管理复杂过程的价值和原则。同时，DevOps旨在增强由开发人员和运营团队组成的不同团队之间的沟通，集成和协作。 它需要采用敏捷方法和DevOps方法来形成无缝工作的产品开发生命周期：敏捷原理有助于塑造和引导正确的开发方向，而DevOps利用这些工具来确保将产品完全交付给客户。
2. 持续集成，持续交付和持续部署之间有什么区别？ 持续集成（CI）是一种将代码版本连续集成到共享存储库中的实践。这种做法可确保自动测试新代码，并能快速检测和修复错误。 持续交付使CI进一步迈出了一步，确保集成后，随时可以在一个按钮内就可以释放代码库。因此，CI可以视为持续

交付的先决条件，这是CI / CD管道的另一个重要组成部分。 对于连续部署，不需要任何手动步骤。这些代码通过测试后，便会自动推送到生产环境。 所有这三个组件：持续集成，持续交付和持续部署是实施DevOps的重要阶段。 一方面，连续交付更适合于活跃用户已经存在的应用程序，这样事情就可以变慢一些并进行更好的调整。另一方面，如果你打算发布一个全新的软件并且将整个过程指定为完全自动化的，则连续部署是你产品的更合适选择。

3. 连续交付和连续部署之间有哪些根本区别？ 在连续交付的情况下，主分支中的代码始终可以手动部署。通过这种做法，开发团队可以决定何时发布新的更改或功能，以最大程度地使组织受益。 同时，连续部署将在测试阶段之后立即将代码中的所有更新和修补程序自动部署到生产环境中，而无需任何人工干预。
4. 持续集成和持续交付之间的区别是什么？ 持续集成有助于确保软件组件紧密协作。整合应该经常进行；最好每小时或每天一次。持续集成有助于提高代码提交的频率，并降低连接多个开发人员的代码的复杂性。最终，此过程减少了不兼容代码和冗余工作的机会。 持续交付是CI / CD流程中的下一步。由于代码不断集成到共享存储库中，因此可以持续测试该代码。在等待代码完成之前，没有间隙可以进行测试。这样可确保找到尽可能多的错误，然后将其连续交付给生产。
5. DevOps和持续交付之间有什么区别？ DevOps更像是一种组织和文化方法，可促进工程团队和运营团队之间的协作和沟通。 同时，持续交付是成功将DevOps实施到产品开发工作流程中的重要因素。持续交付实践有助于使新发行的版本更加乏味和可靠，并建立更加无缝和短的流程。 DevOps的主要目的是有效地结合Dev和Ops角色，消除所有孤岛，并实现独立于持续交付实践的业务目标。 另一方面，如果已经有DevOps流程，则连续交付效果最佳。因此，它扩大了协作并简化了组织的统一产品开发周期。
6. 敏捷，精益IT和DevOps之间有什么区别？ 敏捷是仅专注于软件开发的方法。敏捷旨在迭代开发，建立持续交付，缩短反馈循环以及在整个软件开发生命周期（SDLC）中改善团队协作。 精益IT是一种旨在简化产品开发周期价值流的方法。精益专注于消除不必要的过程，这些过程不会增加价值，并创建流程来优化价值流。 DevOps专注于开发和部署-产品开发过程的Dev和Ops。其目标是有效整合自动化工具和IT专业人员之间的角色，以实现更简化和自动化的流程。

准备好DevOps面试中吗？

希望这些问题和建议的答案能使你快速掌握DevOps和CI/CD的相关知识，帮助你在面试之前对DevOps和CI/CD有系统性的概念和理解。

本文分享自微信公众号 - DevOps攻城狮（DevOps-Engineer），作者：Peter Shen

原文出处及转载信息见文内详细说明，如有侵权，请联系 yunjia_community@tencent.com 删除。

原始发表时间：2020-04-18

本文参与腾讯云自媒体分享计划，欢迎正在阅读的你也加入，一起分享。

方法论