C++面试真题题库

牛客网出品

C++工程师校招面试题库导读

一、学习说明

本题库均来自海量真实校招面试题目大数据进行的整理，后续也会不断更新，可永久免费在线观看，如需下载，也可在页面 https://www.nowcoder.com/interview/center 上直接进行**下载**（下载需要用牛币兑换，一次兑换可享受永久下载权限，因为后续会更新）

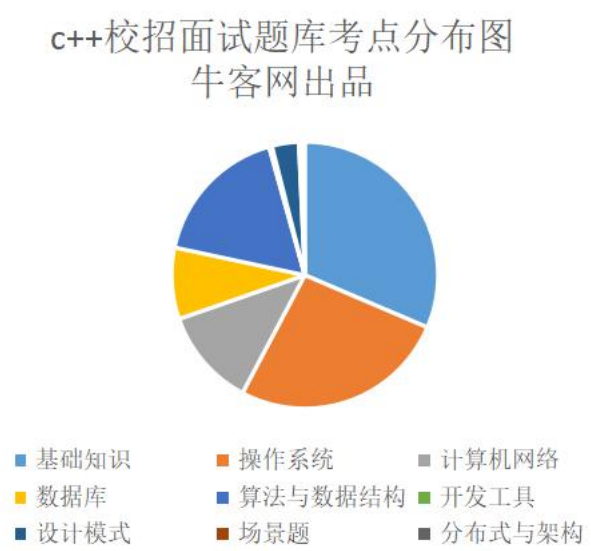
需要严肃说明的是：面试题库作为帮助同学准备面试的辅助资料，但是绝对不能作为备考唯一途径，因为面试是一个考察真实水平的，不是背会了答案就可以的，需要你透彻理解的，否则追问问题答不出来反而减分，毕竟技术面试中面试官最痛恨的就是背答案这个事情了。

学完这个题库，把此题库都理解透彻应对各家企业面试完全没有问题。（当然要加上好的项目以及透彻掌握和你有足够强的手撕代码的能力）

另外，此面试题库中不包括面试中问到的项目，hr 面以及个人技术发展类。

- 项目是比较个性化的，没办法作为一个题库来给大家参考，但是如果你有一个非常有含金量的项目的话，是非常加分的，而且你的项目可能也会被问的多一些；
- hr 面的话一般来说技术面通过的话个人没有太大的和公司不符合的问题都能通过；
- 技术发展类的话这个就完全看自己啦，主要考察的会是你对技术的热爱和学习能力，比如会问一些你是如何学习 xxx 技术的，或者能表达出你对技术的热爱的地方等等。此处不做赘述。

那么抛开这些，c++工程师中技术面中考察的占比如下：



需要注意的是：此图不绝对，因为实际面试中面试官会根据你的简历去问，比如你的项目多可能就问的项目多一些，或者你说哪里精通可能面试官就多去问你这些。而且此图是根据题库数据整理出来，并不是根据实际单场面试整理，比如基础部分不会考那么多，会从中抽着考

但是面试中必考的点且占比非常大的有 c 基础和**算法**。
决定你是否能拿 sp offer（高薪 offer）以及是否进名企的是项目和**算法**。

可以看出，算法除了是面试必过门槛以外，更是决定你是否能进名企或高薪 offer 的决定性因素。

另外关于算法部分，想要系统的学习算法思想，实现高频面试题最优解等详细讲解的话可以报名算法名企校招冲刺班或算法高薪校招冲刺班，你将能学到更先进的算法思想以及又一套系统的校招高频笔试面试题目的解题套路和方法论。

多出来的服务如下：

如果有什么问题，也可以加 qq 咨询 1440073724，如果是早鸟的话，还可以领取优惠码哦

二、面试技巧

面试一般分为技术面和 hr 面，形式的话很少有群面，少部分企业可能会有一个交叉面，不过总的来说，技术面基本就是考察你的专业技术水平的，hr 面的话主要是看这个人的综合素质以及家庭情况符不符合公司要求，一般来讲，技术的话只要通过了技术面 hr 面基本上是没有问题（也有少数企业 hr 面会刷很多人）

那我们主要来说技术面，技术面的话主要是考察专业技术知识和水平，我们是可以有一定的技巧的，但是一定是基于有一定的能力水平的。

所以也慎重的告诉大家，技巧不是投机取巧，是起到辅助效果的，技术面最主要的还是要有实力，这里是基于实力水平之上的技巧。

这里告诉大家面试中的几个技巧：

1、简历上做一个引导：
在词汇上做好区分，比如熟悉 Java，了解 python，精通 c 语言

这样的话对自己的掌握程度有个区分，也好让面试官有个着重去问，python 本来写的也只是了解，自然就不会多问你深入的一些东西了。

2、在面试过程中做一个引导：
面试过程中尽量引导到自己熟知的一个领域，比如问到你说一下 DNS 寻址，然后你简单回答（甚至这步也可以省略）之后，可以说一句，自己对这块可能不是特别熟悉，对计算机网络中的运输层比较熟悉，如果有具体的，甚至可以再加一句，比如 TCP 和 UDP

这样的话你可以把整个面试过程往你熟知的地方引导，也能更倾向于体现出你的优势而不是劣势，但是此方法仅限于掌握合适的度，比如有的知识点是必会的而你想往别处引就有点说不过去了，比如让你说几个 c++的关键字，你一个也说不上来，那可能就真的没辙了。

3、在自我介绍中做一个引导：
一般面试的开头都会有一个自我介绍，在这个位置你也可以尽情的为自己的优势方面去引导。

4、面试过程中展示出自信：
面试过程中的态度也要掌握好，不要自卑，也不要傲娇，自信的回答出每个问题，尤其遇到不会的问题，要么做一些引导，实在不能引导也可以先打打擦边球，和面试官交流一下问题，看起来像是没听懂题意，这个过程也可以再自己思考一下，如果觉得这个过程可以免了的话也直接表明一下这个地方不太熟悉或者还没有掌握好，千万不要强行回答。

面试前的准备：
最重要的肯定是系统的学习了，有一个知识的框架，基础知识的牢靠程度等。
其中算法尤其重要，越来越多公司还会让你现场或者视频面试中手写代码；

另一大重要的和加分项就是项目，在面试前，一定要练习回答自己项目的三个问题：

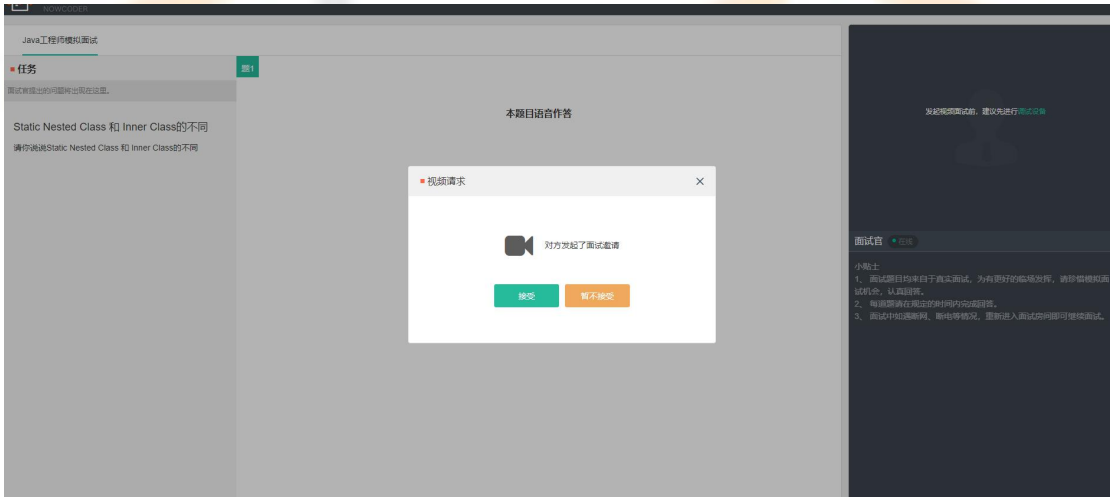
- 这是一个怎样的项目
- 用到了什么技术，为什么用这项技术（以及每项技术很细的点以及扩展）
- 过程中遇到了什么问题，怎么解决的。

那么话说回来，这个的前提是你要有一个好的项目，牛客网 CEO 叶向宇有带大家做项目，感兴趣的可以去了解一下

- 竞争力超过 70%求职者的项目：https://www.nowcoder.com/courses/semester/medium （专属优惠码：DjPgy3x，每期限量前 100 个）
- 竞争力超过 80%求职者的项目：https://www.nowcoder.com/courses/semester/senior （专属优惠码：DMVSexJ，每期限量前 100 个）

知识都掌握好后，剩下的就是一个心态和模拟练习啦，因为你面试的少的话现场难免紧张，而且没在那个环境下可能永远不知道自己回答的怎么样。

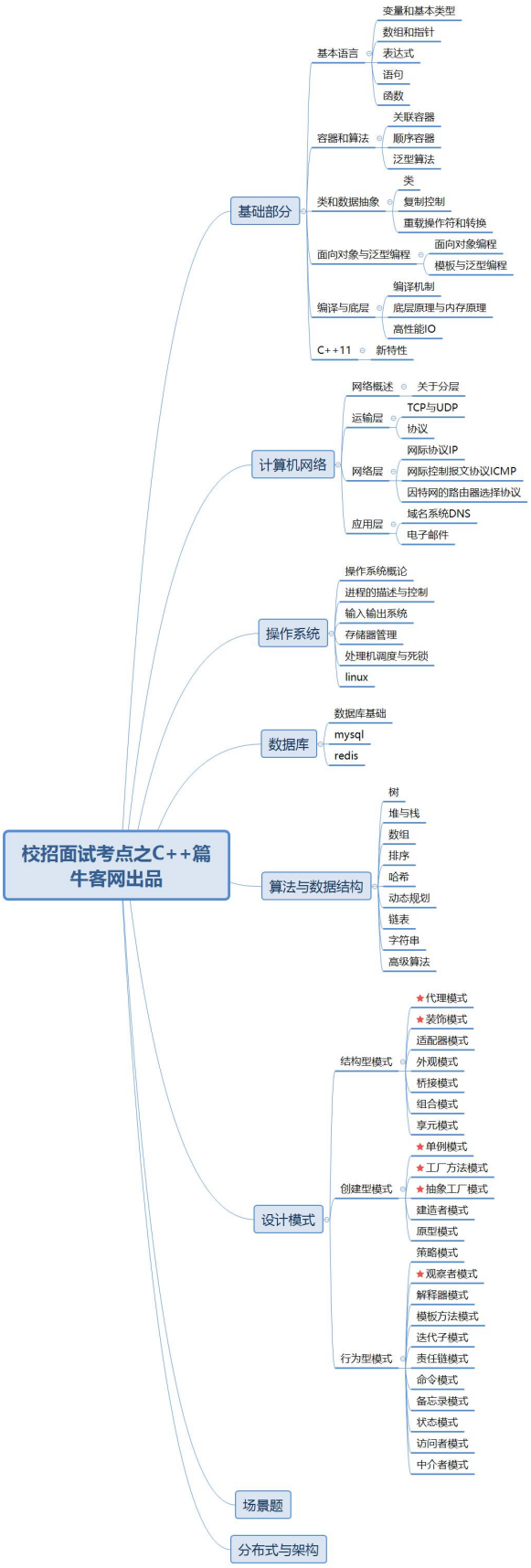
因为哪怕当你都会了的情况下，你的表达和心态就显得更重要了，会了但是没有表达的很清晰就很吃亏了，牛客网这边有 AI 模拟面试，完全模拟了真实面试环境，正好大家可以真正的去练习一下，还能收获一份面试报告：
https://www.nowcoder.com/interview/ai/index



面试后需要做的：
面试完了的话就不用太在意结果了，有限的时间就应该做事半功倍的事情，当然，要保持电话邮箱畅通，不然别给你发 offer 你都不知道。

抛开这些，我们需要做的是及时将面试中的问题记录下来，尤其是自己回答的不够好的问题，一定要花时间去研究，并解决这些问题，下次面试再遇到相同的问题就能很好的解决，当然，即使不遇到，你这个习惯坚持住，后面也可以作为一个经历去跟面试官说，能表现出你对技术的喜爱和钻研的一个态度，同时，每次面试后你会发现自己的不足，查缺补漏的好机会，及时调整，在不断的调整和查缺补漏的过程中，你会越来越好。

三、面试考点导图


校招面试考点之C++篇
牛客网出品
基础部分
基本语言
变量和基本类型
数组和指针
表达式
语句
函数
容器和算法
关联容器
顺序容器
泛型算法
类和数据抽象
类
复制控制
重载操作符和转换
面向对象与泛型编程
面向对象编程
模板与泛型编程
编译与底层
编译机制
底层原理与内存原理
高性能IO
C++11
新特性
计算机网络
网络概述
关于分层
运输层
TCP与UDP
协议
网络层
网际协议IP
网际控制报文协议ICMP
因特网的路由器选择协议
应用层
域名系统DNS
电子邮件
操作系统
操作系统概论
进程的描述与控制
输入输出系统
存储器管理
处理机调度与死锁
linux
数据库
数据库基础
mysql
redis
算法与数据结构
树
堆与栈
数组
排序
哈希
动态规划
链表
字符串
高级算法
设计模式
结构型模式
★代理模式
★装饰模式
适配器模式
外观模式
桥接模式
组合模式
享元模式
创建型模式
★单例模式
★工厂方法模式
★抽象工厂模式
建造者模式
原型模式
行为型模式
策略模式
★观察者模式
解释器模式
模板方法模式
迭代子模式
责任链模式
命令模式
备忘录模式
状态模式
访问者模式
中介者模式
场景题
分布式与架构

四、一对一答疑讲解戳这里

如果你对校招求职或者职业发展很困惑，欢迎与牛客网专业老师沟通，老师会帮你一对一讲解答疑哦（可以扫下方二维码或者添加微信号：niukewang985）

目录

一、基础知识

1、基本语言

1、说一下 static 关键字的作用

考察点：C/C++

参考回答：

1. 全局静态变量

在全局变量前加上关键字 static，全局变量就定义成一个全局静态变量.

静态存储区，在整个程序运行期间一直存在。

初始化：未经初始化的全局静态变量会被自动初始化为 0（自动对象的值是任意的，除非他被显式初始化）；

作用域：全局静态变量在声明他的文件之外是不可见的，准确地说是从定义之处开始，到文件结尾。

2. 局部静态变量

在局部变量之前加上关键字 static，局部变量就成为一个局部静态变量。

内存中的位置：静态存储区

初始化：未经初始化的全局静态变量会被自动初始化为 0（自动对象的值是任意的，除非他被显式初始化）；

作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域结束。但是当局部静态变量离开作用域后，并没有销毁，而是仍然驻留在内存当中，只不过我们不能再对它进行访问，直到该函数再次被调用，并且值不变；

3. 静态函数

在函数返回类型前加 static，函数就定义为静态函数。函数的定义和声明在默认情况下都是 extern 的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。

函数的实现使用 static 修饰，那么这个函数只可在本 cpp 内使用，不会同其他 cpp 中的同名函数引起冲突；

warning：不要再头文件中声明 static 的全局函数，不要在 cpp 内声明非 static 的全局函数，如果你要在多个 cpp 中复用该函数，就把它的声明提到头文件里去，否则 cpp 内部声明需加上 static 修饰；

4. 类的静态成员

在类中，静态成员可以实现多个对象之间的数据共享，并且使用静态数据成员还不会破坏隐藏的原则，即保证了安全性。因此，静态成员是类的所有对象中共享的成员，而不是某个对象的成员。对多个对象来说，静态数据成员只存储一处，供所有对象共用

5. 类的静态函数

静态成员函数和静态数据成员一样，它们都属于类的静态成员，它们都不是对象成员。因此，对静态成员的引用不需要用对象名。

在静态成员函数的实现中不能直接引用类中说明的非静态成员，可以引用类中说明的静态成员（这点非常重要）。如果静态成员函数中要引用非静态成员时，可通过对象来引用。从中可看出，调用静态成员函数使用如下格式：<类名>::<静态成员函数名>(<参数表>);

2、说一下 C++和 C 的区别

考察点:C 基础

参考回答：

设计思想上：

C++是面向对象的语言，而 C 是面向过程的结构化编程语言

语法上：

C++具有重载、继承和多态三种特性

C++相比 C，增加多许多类型安全的功能，比如强制类型转换、

C++支持范式编程，比如模板类、函数模板等

3、说一下 C++中 static 关键字的作用

考点：static 关键字

参考回答：对于函数定义和代码块之外的变量声明，static 修改标识符的链接属性，由默认的 external 变为 internal，作用域和存储类型不改变，这些符号只能在声明它们的源文件中访问。

对于代码块内部的变量声明，static 修改标识符的存储类型，由自动变量改为静态变量，作用域和链接属性不变。这种变量在程序执行之前就创建，在程序执行的整个周期都存在。

对于被 static 修饰的普通函数，其只能在定义它的源文件中使用，不能在其他源文件中被引用

对于被 static 修饰的类成员变量和成员函数，它们是属于类的，而不是某个对象，所有对象共享一个静态成员。静态成员通过<类名>::<静态成员>来使用。

4、请说一下 static 的作用

考察点：C/C++

参考回答：

1. 全局静态变量

在全局变量前加上关键字 static，全局变量就定义成一个全局静态变量。

静态存储区，在整个程序运行期间一直存在。

初始化：未经初始化的全局静态变量会被自动初始化为 0（自动对象的值是任意的，除非他被显式初始化）。

作用域：全局静态变量在声明他的文件之外是不可见的，准确地说是从定义之处开始，到文件结尾。

2. 局部静态变量

在局部变量之前加上关键字 static，局部变量就成为一个局部静态变量。

内存中的位置：静态存储区。

初始化：未经初始化的全局静态变量会被自动初始化为 0（自动对象的值是任意的，除非他被显式初始化）。

作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域结束。但是当局部静态变量离开作用域后，并没有销毁，而是仍然驻留在内存当中，只不过我们不能再对它进行访问，直到该函数再次被调用，并且值不变。

3. 静态函数

在函数返回类型前加 static，函数就定义为静态函数。函数的定义和声明在默认情况下都是 extern 的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。

函数的实现使用 static 修饰，那么这个函数只可在本 cpp 内使用，不会同其他 cpp 中的同名函数引起冲突。

warning：不要再头文件中声明 static 的全局函数，不要在 cpp 内声明非 static 的全局函数，如果你要在多个 cpp 中复用该函数，就把它的声明提到头文件里去，否则 cpp 内部声明需加上 static 修饰。

4. 类的静态成员

在类中，静态成员可以实现多个对象之间的数据共享，并且使用静态数据成员还不会破坏隐藏的原则，即保证了安全性。因此，静态成员是类的所有对象中共享的成员，而不是某个对象的成员。对多个对象来说，静态数据成员只存储一处，供所有对象共用。

5. 类的静态函数

静态成员函数和静态数据成员一样，它们都属于类的静态成员，它们都不是对象成员。因此，对静态成员的引用不需要用对象名。

在静态成员函数的实现中不能直接引用类中说明的非静态成员，可以引用类中说明的静态成员（这点非常重要）。如果静态成员函数中要引用非静态成员时，可通过对象来引用。从中可看出，调用静态成员函数使用如下格式：<类名>::<静态成员函数名>(<参数表>);

5、说一说 c++中四种 cast 转换

考察点：C/C++

参考回答：

C++中四种类型转换是：static_cast, dynamic_cast, const_cast, reinterpret_cast

1、const_cast

用于将 const 变量转为非 const

2、static_cast

用于各种隐式转换，比如非 const 转 const，void*转指针等，static_cast 能用于多态向上转化，如果向下转能成功但是不安全，结果未知；

3、dynamic_cast

用于动态类型转换。只能用于含有虚函数的类，用于类层次间的向上和向下转化。只能转指针或引用。向下转化时，如果是非法的对于指针返回 NULL，对于引用抛异常。要深入了解内部转换的原理。

向上转换：指的是子类向基类的转换

向下转换：指的是基类向子类的转换

它通过判断在执行到该语句的时候变量的运行时类型和要转换的类型是否相同来判断是否能够进行向下转换。

4、reinterpret_cast

几乎什么都可以转，比如将 int 转指针，可能会出问题，尽量少用；

5、为什么不使用 C 的强制转换？

C 的强制转换表面上看起来功能强大什么都能转，但是转化不够明确，不能进行错误检查，容易出错。

6、请说一下 C/C++ 中指针和引用的区别？

考察点：C/C++基础知识

参考回答：

1.指针有自己的一块空间，而引用只是一个别名；

2.使用 sizeof 看一个指针的大小是 4，而引用则是被引用对象的大小；

3.指针可以被初始化为 NULL，而引用必须被初始化且必须是一个已有对象 的引用；

4.作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引 用的修改都会改变引用所指向的对象；

5.可以有 const 指针，但是没有 const 引用；

6.指针在使用中可以指向其它对象，但是引用只能是一个对象的引用，不能 被改变；

7.指针可以有多级指针（**p），而引用至于一级；

8.指针和引用使用++运算符的意义不一样；

9.如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄露。

7、给定三角形 ABC 和一点 P(x,y,z)，判断点 P 是否在 ABC 内，给出思路并手写代码

参考回答：

根据面积法，如果 P 在三角形 ABC 内，那么三角形 ABP 的面积+三角形 BCP 的面积+三角形 ACP 的面积应该等于三角形 ABC 的面积。算法如下：

```
#include <iostream>

#include <math.h>

using namespace std;


#define ABS_FLOAT_0 0.0001


struct point_float

{
```

```
    float x;
    float y;
    };

    /**
    * @brief 计算三角形面积
    */
    float GetTriangleSquar(const point_float pt0, const point_float pt1, const
point_float pt2)
    {
    point_float AB,   BC;
    AB.x = pt1.x - pt0.x;
    AB.y = pt1.y - pt0.y;
    BC.x = pt2.x - pt1.x;
    BC.y = pt2.y - pt1.y;
    return fabs((AB.x * BC.y - AB.y * BC.x)) / 2.0f;
    }

    /**
    * @brief 判断给定一点是否在三角形内或边上
    */
    bool IsInTriangle(const point_float A, const point_float B, const point_float C,
const point_float D)
    {
    float SABC, SADB, SBDC, SADC;
    SABC = GetTriangleSquar(A, B, C);
```

```
    SADB = GetTriangleSquar(A, D, B);

    SBDC = GetTriangleSquar(B, D, C);

    SADC = GetTriangleSquar(A, D, C);


    float SumSuqar = SADB + SBDC + SADC;


    if ((-ABS_FLOAT_0 < (SABC - SumSuqar)) && ((SABC - SumSuqar) < ABS_FLOAT_0))

    {

        return true;

    }

    else

    {

        return false;

    }

    }
```

8、请你说一下你理解的 c++中的 smart pointer 四个智能指针：
shared_ptr,unique_ptr,weak_ptr,auto_ptr

考察点：C++智能指针

参考回答：

C++里面的四个智能指针: auto_ptr, shared_ptr, weak_ptr, unique_ptr 其中后三个是 c++11 支持，并且第一个已经被 11 弃用。

为什么要使用智能指针：

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

1. auto_ptr（c++98 的方案，cpp11 已经抛弃）

采用所有权模式。

```
auto_ptr< string> p1 (new string ("I reigned lonely as a cloud.” ));

auto_ptr<string> p2;

p2 = p1; //auto_ptr 不会报错.
```

此时不会报错，p2 剥夺了 p1 的所有权，但是当程序运行时访问 p1 将会报错。所以 auto_ptr 的缺点是：存在潜在的内存崩溃问题！

2. unique_ptr（替换 auto_ptr）

unique_ptr 实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露(例如“以 new 创建对象后因为发生异常而忘记调用 delete”)特别有用。

采用所有权模式，还是上面那个例子

```
unique_ptr<string> p3 (new string ("auto"));   //#4

unique_ptr<string> p4;                         //#5

p4 = p3;//此时会报错！！
```

编译器认为 p4=p3 非法，避免了 p3 不再指向有效数据的问题。因此，unique_ptr 比 auto_ptr 更安全。

另外 unique_ptr 还有更聪明的地方：当程序试图将一个 unique_ptr 赋值给另一个时，如果源 unique_ptr 是个临时右值，编译器允许这么做；如果源 unique_ptr 将存在一段时间，编译器将禁止这么做，比如：

```
unique_ptr<string> pu1(new string ("hello world"));

unique_ptr<string> pu2;

pu2 = pu1;                                     // #1 not allowed

unique_ptr<string> pu3;

pu3 = unique_ptr<string>(new string ("You"));   // #2 allowed
```

其中#1 留下悬挂的 unique_ptr(pu1)，这可能导致危害。而#2 不会留下悬挂的 unique_ptr，因为它调用 unique_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。这种随情况而已的行为表明，unique_ptr 优于允许两种赋值的 auto_ptr 。

注：如果确实想执行类似与#1 的操作，要安全的重用这种指针，可给它赋新值。C++有一个标准库函数 std::move()，让你能够将一个 unique_ptr 赋给另一个。例如：

```
unique_ptr<string> ps1, ps2;

ps1 = demo("hello");

ps2 = move(ps1);

ps1 = demo("alexia");

cout << *ps2 << *ps1 << endl;
```

3. shared_ptr

shared_ptr 实现共享式拥有概念。多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。从名字 share 就可以看出了资源可以被多个指针共享，它使用计数机制来表明资源被几个指针共享。可以通过成员函数 use_count()来查看资源的所有者个数。除了可以通过 new 来构造，还可以通过传入 auto_ptr, unique_ptr,weak_ptr 来构造。当我们调用 release()时，当前指针会释放资源所有权，计数减一。当计数等于 0 时，资源会被释放。

shared_ptr 是为了解决 auto_ptr 在对象所有权上的局限性(auto_ptr 是独占的)，在使用引用计数的机制上提供了可以共享所有权的智能指针。

成员函数：

use_count 返回引用计数的个数

unique 返回是否是独占所有权(use_count 为 1)

swap 交换两个 shared_ptr 对象(即交换所拥有的对象)

reset 放弃内部对象的所有权或拥有对象的变更，会引起原有对象的引用计数的减少

get 返回内部对象(指针)，由于已经重载了()方法，因此和直接使用对象是一样的.如 shared_ptr<int> sp(new int(1)); sp 与 sp.get()是等价的

4. weak_ptr

weak_ptr 是一种不控制对象生命周期的智能指针，它指向一个 shared_ptr 管理的对象. 进行该对象的内存管理的是那个强引用的 shared_ptr. weak_ptr 只是提供了对管理对象的一个访问手段。weak_ptr 设计的目的是为配合 shared_ptr 而引入的一种智能指针来协助 shared_ptr 工作，它只可以从一个 shared_ptr 或另一个 weak_ptr 对象构造，它的构造和析构不会引起引用记数的增加或减少。weak_ptr 是用来解决 shared_ptr 相互引用时的死锁问题，如果说两个 shared_ptr 相互引用,那么这两个指针的引用计数永远不可能下降为 0,资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用计数，和 shared_ptr 之间可以相互转化，shared_ptr 可以直接赋值给它，它可以通过调用 lock 函数来获得 shared_ptr。

```
class B;

class A

{

public:

    shared_ptr<B> pb_;

    ~A()

    {

        cout<<"A delete\n";

    }

};

class B

{

public:

    shared_ptr<A> pa_;

    ~B()

    {

        cout<<"B delete\n";

    }

};

void fun()

{
```

```
        shared_ptr<B> pb(new B());
        shared_ptr<A> pa(new A());
        pb->pa_ = pa;
        pa->pb_ = pb;
        cout<<pb.use_count()<<endl;
        cout<<pa.use_count()<<endl;
    }
    int main()
    {
        fun();
        return 0;
    }
```

可以看到 fun 函数中 pa ，pb 之间互相引用，两个资源的引用计数为 2，当要跳出函数时，智能指针 pa，pb 析构时两个资源引用计数会减一，但是两者引用计数还是为 1，导致跳出函数时资源没有被释放（A B 的析构函数没有被调用），如果把其中一个改为 weak_ptr 就可以了，我们把类 A 里面的 shared_ptr pb_; 改为 weak_ptr pb_; 运行结果如下，这样的话，资源 B 的引用开始就只有 1，当 pb 析构时，B 的计数变为 0，B 得到释放，B 释放的同时也会使 A 的计数减一，同时 pa 析构时使 A 的计数减一，那么 A 的计数为 0，A 得到释放。

注意的是我们不能通过 weak_ptr 直接访问对象的方法，比如 B 对象中有一个方法 print(),我们不能这样访问，pa->pb_->print(); 英文 pb_是一个 weak_ptr，应该先把它转化为 shared_ptr,如：shared_ptr p = pa->pb_.lock(); p->print();

9、怎么判断一个数是二的倍数，怎么求一个数中有几个 1，说一下你的思路并手写代码

考察点：算法

参考回答：

1、判断一个数是不是二的倍数，即判断该数二进制末位是不是 0：

a % 2 == 0 或者 a & 0x0001 == 0。

2、求一个数中 1 的位数，可以直接逐位除十取余判断：

```
int fun(long x)
{
    int _count = 0;
    while(x)
    {
        if(x % 10 == 1)
            ++_count;
        x /= 10;
    }
    return _count;
}

int main()
{
    cout << fun(123321) << endl;
    return 0;
}
```

10、请回答一下数组和指针的区别

知识点：数组 指针

参考回答：

指针和数组的主要区别如下：

指针	数组
保存数据的地址	保存数据
间接访问数据，首先获得指针的内容，然后将其作为地址，从该地址中提取数据	直接访问数据，
通常用于动态的数据结构	通常用于固定数目且数据类型相同的元素
通过 Malloc 分配内存，free 释放内存	隐式的分配和删除
通常指向匿名数据，操作匿名函数	自身即为数据名

11、请你回答一下野指针是什么？

考点：指针

参考回答：

野指针就是指向一个已删除的对象或者未申请访问受限内存区域的指针

12、请你介绍一下 C++中的智能指针

考点：shared_ptr

参考回答：

智能指针主要用于管理在堆上分配的内存，它将普通的指针封装为一个栈对象。当栈对象的生存周期结束后，会在析构函数中释放掉申请的内存，从而防止内存泄漏。C++ 11 中最常用的智能指针类型为 shared_ptr,它采用引用计数的方法，记录当前内存资源被多少个智能指针引用。该引用计数的内存在堆上分配。当新增一个时引用计数加 1，当过期时引用计数减一。只有引用计数为 0 时，智能指针才会自动释放引用的内存资源。对 shared_ptr 进行初始化时不能将一个

普通指针直接赋值给智能指针，因为一个是指针，一个是类。可以通过 make_shared 函数或者通过构造函数传入普通指针。并可以通过 get 函数获得普通指针。

13、请你回答一下智能指针有没有内存泄露的情况

考点：循环引用 weak_ptr

参考回答：

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

```
class Parent {
private:
    std::shared_ptr<Child> ChildPtr;
public:
    void setChild(std::shared_ptr<Child> child) {
        this->ChildPtr = child;
    }

    void doSomething() {
        if (this->ChildPtr.use_count()) {

        }
    }

    ~Parent() {
    }
};
```

```
class Child {
private:
    std::shared_ptr<Parent> ParentPtr;
public:
    void setPartent(std::shared_ptr<Parent> parent) {
        this->ParentPtr = parent;
    }
    void doSomething() {
        if (this->ParentPtr.use_count()) {

        }
    }
    ~Child() {
    }
};
```

```
int main() {
    std::weak_ptr<Parent> wpp;
    std::weak_ptr<Child> wpc;
    {
        std::shared_ptr<Parent> p(new Parent);
        std::shared_ptr<Child> c(new Child);
        p->setChild(c);
        c->setPartent(p);
        wpp = p;
        wpc = c;
        std::cout << p.use_count() << std::endl; // 2
        std::cout << c.use_count() << std::endl; // 2
    }
    std::cout << wpp.use_count() << std::endl;  // 1
    std::cout << wpc.use_count() << std::endl;  // 1
    return 0;
}
```

上述代码中，parent 有一个 shared_ptr 类型的成员指向孩子，而 child 也有一个 shared_ptr 类型的成员指向父亲。然后在创建孩子和父亲对象时也使用了智能指针 c 和 p，随后将 c 和 p 分别又赋值给 child 的智能指针成员 parent 和 parent 的智能指针成员 child。从而形成了一个循环引用：

14、请你来说一下智能指针的内存泄漏如何解决

参考回答：

为了解决循环引用导致的内存泄漏，引入了 weak_ptr 弱指针，weak_ptr 的构造函数不会修改引用计数的值，从而不会对对象的内存进行管理，其类似一个普通指针，但不指向引用计数的共享内存，但是其可以检测到所管理的对象是否已经被释放，从而避免非法访问。

15、请你理解的 c++中的引用和指针

考察点：C/C++

参考回答：定义：

1、引用：

C++是 C 语言的继承，它可进行过程化程序设计，又可以进行以抽象数据类型为特点的基于对象的程序设计，还可以进行以继承和多态为特点的面向对象的程序设计。引用就是 C++对 C 语言的重要扩充。引用就是某一变量的一个别名，对引用的操作与对变量直接操作完全一样。引用的声明方法：类型标识符 &引用名=目标变量名；引用引入了对象的一个同义词。定义引用的表示方法与定义指针相似，只是用&代替了*。

2、指针：

指针利用地址，它的值直接指向存在电脑存储器中另一个地方的值。由于通过地址能找到所需的变量单元，可以说，地址指向该变量单元。因此，将地址形象化的称为“指针”。意思是通过它能找到以它为地址的内存单元。

区别：

1、指针有自己的一块空间，而引用只是一个别名；

2、使用 sizeof 看一个指针的大小是 4，而引用则是被引用对象的大小；

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

4、作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引用的修改都会改变引用所指向的对象；

5、可以有 const 指针，但是没有 const 引用；

6、指针在使用中可以指向其它对象，但是引用只能是一个对象的引用，不能 被改变；

7、指针可以有多级指针（**p），而引用至于一级；

8、指针和引用使用++运算符的意义不一样；

9、如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄露。

16、请你来说一下 C++中的智能指针

参考回答：

C++里面的四个智能指针: auto_ptr, shared_ptr, weak_ptr, unique_ptr 其中后三个是 c++11 支持，并且第一个已经被 11 弃用。

为什么要使用智能指针：

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

1、auto_ptr（c++98 的方案，cpp11 已经抛弃）

采用所有权模式。

```
auto_ptr< string> p1 (new string ("I reigned lonely as a cloud.”));
```

```
auto_ptr<string> p2;

p2 = p1; //auto_ptr 不会报错.
```

此时不会报错，p2 剥夺了 p1 的所有权，但是当程序运行时访问 p1 将会报错。所以 auto_ptr 的缺点是：存在潜在的内存崩溃问题！

2、unique_ptr（替换 auto_ptr）

unique_ptr 实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露(例如“以 new 创建对象后因为发生异常而忘记调用 delete”)特别有用。

采用所有权模式，还是上面那个例子

```
unique_ptr<string> p3 (new string ("auto"));           //#4

unique_ptr<string> p4;                          //#5

p4 = p3;//此时会报错！！
```

编译器认为 p4=p3 非法，避免了 p3 不再指向有效数据的问题。因此，unique_ptr 比 auto_ptr 更安全。

另外 unique_ptr 还有更聪明的地方：当程序试图将一个 unique_ptr 赋值给另一个时，如果源 unique_ptr 是个临时右值，编译器允许这么做；如果源 unique_ptr 将存在一段时间，编译器将禁止这么做，比如：

```
unique_ptr<string> pu1(new string ("hello world"));

unique_ptr<string> pu2;

pu2 = pu1;                                          // #1 not allowed

unique_ptr<string> pu3;

pu3 = unique_ptr<string>(new string ("You"));          // #2 allowed
```

其中#1 留下悬挂的 unique_ptr(pu1)，这可能导致危害。而#2 不会留下悬挂的 unique_ptr，因为它调用 unique_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。这种随情况而已的行为表明，unique_ptr 优于允许两种赋值的 auto_ptr 。

注：如果确实想执行类似与#1 的操作，要安全的重用这种指针，可给它赋新值。C++有一个标准库函数 std::move()，让你能够将一个 unique_ptr 赋给另一个。例如：

```
unique_ptr<string> ps1, ps2;

ps1 = demo("hello");

ps2 = move(ps1);

ps1 = demo("alexia");

cout << *ps2 << *ps1 << endl;
```

3、shared_ptr

shared_ptr 实现共享式拥有概念。多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。从名字 share 就可以看出了资源可以被多个指针共享，它使用计数机制来表明资源被几个指针共享。可以通过成员函数 use_count()来查看资源的所有者个数。除了可以通过 new 来构造，还可以通过传入 auto_ptr, unique_ptr,weak_ptr 来构造。当我们调用 release()时，当前指针会释放资源所有权，计数减一。当计数等于 0 时，资源会被释放。

shared_ptr 是为了解决 auto_ptr 在对象所有权上的局限性(auto_ptr 是独占的)，在使用引用计数的机制上提供了可以共享所有权的智能指针。

成员函数：

use_count 返回引用计数的个数

unique 返回是否是独占所有权(use_count 为 1)

swap 交换两个 shared_ptr 对象(即交换所拥有的对象)

reset 放弃内部对象的所有权或拥有对象的变更，会引起原有对象的引用计数的减少

get 返回内部对象(指针)，由于已经重载了()方法，因此和直接使用对象是一样的.如 shared_ptr<int> sp(new int(1)); sp 与 sp.get()是等价的

4、weak_ptr

weak_ptr 是一种不控制对象生命周期的智能指针，它指向一个 shared_ptr 管理的对象.进行该对象的内存管理的是那个强引用的 shared_ptr. weak_ptr 只是提供了对管理对象的一个访问手段。weak_ptr 设计的目的是为配合 shared_ptr 而引入的一种智能指针来协助 shared_ptr 工作，它只可以从一个 shared_ptr 或另一个 weak_ptr 对象构造，它的构造和析构不会引起引用记数的增加或减少。weak_ptr 是用来解决 shared_ptr 相互引用时的死锁问题，

如果说两个 shared_ptr 相互引用,那么这两个指针的引用计数永远不可能下降为 0,资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用计数，和 shared_ptr 之间可以相互转化，shared_ptr 可以直接赋值给它，它可以通过调用 lock 函数来获得 shared_ptr。

```
class B;
class A
{
public:
    shared_ptr<B> pb_;
    ~A()
    {
        cout<<"A delete\n";
    }
};
class B
{
public:
    shared_ptr<A> pa_;
    ~B()
    {
        cout<<"B delete\n";
    }
};
void fun()
{
    shared_ptr<B> pb(new B());
    shared_ptr<A> pa(new A());
    pb->pa_ = pa;
```

```
    pa->pb_ = pb;
    cout<<pb.use_count()<<endl;
    cout<<pa.use_count()<<endl;
}
int main()
{
    fun();
    return 0;
}
```

可以看到 fun 函数中 pa ，pb 之间互相引用，两个资源的引用计数为 2，当要跳出函数时，智能指针 pa，pb 析构时两个资源引用计数会减一，但是两者引用计数还是为 1，导致跳出函数时资源没有被释放（A B 的析构函数没有被调用），如果把其中一个改为 weak_ptr 就可以了，我们把类 A 里面的 shared_ptr pb_; 改为 weak_ptr pb_; 运行结果如下，这样的话，资源 B 的引用开始就只有 1，当 pb 析构时，B 的计数变为 0，B 得到释放，B 释放的同时也会使 A 的计数减一，同时 pa 析构时使 A 的计数减一，那么 A 的计数为 0，A 得到释放。

注意的是我们不能通过 weak_ptr 直接访问对象的方法，比如 B 对象中有一个方法 print(),我们不能这样访问，pa->pb_->print(); 英文 pb_是一个 weak_ptr，应该先把它转化为 shared_ptr,如：shared_ptr p = pa->pb_.lock(); p->print();

17、请你回答一下为什么析构函数必须是虚函数？为什么 C++默认的析构函数不是虚函数

考点：基础

参考回答：

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

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

18、请你来说一下函数指针

考察点：C/C++

参考回答：

1、定义

函数指针是指向函数的指针变量。

函数指针本身首先是一个指针变量，该指针变量指向一个具体的函数。这正如用指针变量可指向整型变量、字符型、数组一样，这里是指向函数。

C 在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。有了指向函数的指针变量后，可用该指针变量调用函数，就如同用指针变量可引用其他类型变量一样，在这些概念上是大体一致的。

2、用途：

调用函数和做函数的参数，比如回调函数。

3、示例：

```
char  * fun(char  * p)  {…}      // 函数 fun

char  * (*pf)(char  * p);        // 函数指针 pf

pf = fun;                        // 函数指针 pf 指向函数 fun

pf(p);                          // 通过函数指针 pf 调用函数 fun
```

19、请你来说一下 fork 函数

考察点：STL

参考回答：

Fork：创建一个和当前进程映像一样的进程可以通过 fork( )系统调用：

```
#include <sys/types.h>

#include <unistd.h>

pid_t fork(void);
```

成功调用 fork( )会创建一个新的进程，它几乎与调用 fork( )的进程一模一样，这两个进程都会继续运行。在子进程中，成功的 fork( )调用会返回 0。在父进程中 fork( )返回子进程的 pid。如果出现错误，fork( )返回一个负值。

最常见的 fork( )用法是创建一个新的进程，然后使用 exec( )载入二进制映像，替换当前进程的映像。这种情况下，派生（fork）了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。

在早期的 Unix 系统中，创建进程比较原始。当调用 fork 时，内核会把所有的内部数据结构复制一份，复制进程的页表项，然后把父进程的地址空间中的内容逐页的复制到子进程的地址空间中。但从内核角度来说，逐页的复制方式是十分耗时的。现代的 Unix 系统采取了更多的优化，例如 Linux，采用了写时复制的方法，而不是对父进程空间进程整体复制。

20、请你来说一下 C++中析构函数的作用

考察点：C++

参考回答：

析构函数与构造函数对应，当对象结束其生命周期，如对象所在的函数已调用完毕时，系统会自动执行析构函数。

析构函数名也应与类名相同，只是在函数名前面加一个位取反符~，例如~stud( )，以区别于构造函数。它不能带任何参数，也没有返回值（包括 void 类型）。只能有一个析构函数，不能重载。

如果用户没有编写析构函数，编译系统会自动生成一个缺省的析构函数（即使自定义了析构函数，编译器也总是会为我们合成一个析构函数，并且如果自定义了析构函数，编译器在执行时会先调用自定义的析构函数再调用合成的析构函数），它也不进行任何操作。所以许多简单的类中没有用显式的析构函数。

如果一个类中有指针，且在使用的过程中动态的申请了内存，那么最好显示构造析构函数在销毁类之前，释放掉申请的内存空间，避免内存泄漏。

类析构顺序：1）派生类本身的析构函数；2）对象成员析构函数；3）基类析构函数。

21、请你来说一下静态函数和虚函数的区别

参考回答：

静态函数在编译的时候就已经确定运行时机，虚函数在运行的时候动态绑定。虚函数因为用了虚函数表机制，调用的时候会增加一次内存开销

22、请你来说一说重载和覆盖

参考回答：

重载：两个函数名相同，但是参数列表不同（个数，类型），返回值类型没有要求，在同一作用域中
重写：子类继承了父类，父类中的函数是虚函数，在子类中重新定义了这个虚函数，这种情况是重写

23、请你来说一说 static 关键字

参考回答：

1.加了 static 关键字的全局变量只能在本文件中使用。例如在 a.c 中定义了 static int a=10;那么在 b.c 中用 extern int a 是拿不到 a 的值得，a 的作用域只在 a.c 中。
2.static 定义的静态局部变量分配在数据段上，普通的局部变量分配在栈上，会因为函数栈帧的释放而被释放掉。
3. 对一个类中成员变量和成员函数来说，加了 static 关键字，则此变量/函数就没有了 this 指针了，必须通过类名才能访问

24、请你说一说 strcpy 和 strlen

参考回答：

strcpy 是字符串拷贝函数，原型：

```
char *strcpy(char* dest, const char *src);
```

从 src 逐字节拷贝到 dest，直到遇到'\0'结束，因为没有指定长度，可能会导致拷贝越界，造成缓冲区溢出漏洞,安全版本是 strncpy 函数。
strlen 函数是计算字符串长度的函数，返回从开始到'\0'之间的字符个数。

25、请你说一说你理解的虚函数和多态

参考回答：

多态的实现主要分为静态多态和动态多态，静态多态主要是重载，在编译的时候就已经确定；动态多态是用虚函数机制实现的，在运行期间动态绑定。举个例子：一个父类类型的指针指向一个子类对象时候，使用父类的指针去调用子类中重写了的父类中的虚函数的时候，会调用子类重写过后的函数，在父类中声明为加了 virtual 关键字的函数，在子类中重写时候不需要加 virtual 也是虚函数。
虚函数的实现：在有虚函数的类中，类的最开始部分是一个虚函数表的指针，这个指针指向一个虚函数表，表中放了虚函数的地址，实际的虚函数在代码段(.text)中。当子类继承了父类的时候也会继承其虚函数表，当子类重写父类中虚函数时候，会将其继承到的虚函数表中的地址替换为重新写的函数地址。使用了虚函数，会增加访问内存开销，降低效率。

26、请你来回答一下++i 和 i++的区别

参考回答：

++i 先自增 1，再返回，i++先返回 i,再自增 1

27、请你来说一说++i 和 i++的实现

参考回答：

1. ++i 实现：

```
int&   int::operator++ ()
{
        *this +=1;
        return *this;
}
```

2. i++ 实现：

```
const int   int::operator (int)
{
        int oldValue = *this;
      ++(*this);
          return oldValue;
  }
```

28、请你来写个函数在 main 函数执行前先运行

参考回答：

```
__attribute((constructor))void before()
{
    printf("before main\n");
}
```

29、有段代码写成了下边这样，如果在只修改一个字符的前提下，使代码输出 20 个 hello?

```
for(int i = 0; i < 20; i--)
cout << "hello" << endl;
```

参考回答：

```
for(int i = 0; i + 20; i--)

cout << "hello" << endl;
```

30、请你来说一下智能指针 shared_ptr 的实现

参考回答：

核心要理解引用计数，什么时候销毁底层指针，还有赋值，拷贝构造时候的引用计数的变化，析构的时候要判断底层指针的引用计数为 0 了才能真正释放底层指针的内存

```
template <typename T>

class SmartPtr

{

private:

    T *ptr;    //底层真实的指针

    int *use_count;//保存当前对象被多少指针引用计数

public:

    SmartPtr(T *p); //SmartPtr<int>p(new int(2));

    SmartPtr(const SmartPtr<T>&orig);//SmartPtr<int>q(p);

    SmartPtr<T>&operator=(const SmartPtr<T> &rhs);//q=p

    ~SmartPtr();

    T operator*();  //为了能把智能指针当成普通指针操作定义解引用操作

    T*operator->();  //定义取成员操作

    T* operator+(int i);//定义指针加一个常数

    int operator-(SmartPtr<T>&t1,SmartPtr<T>&t2);//定义两个指针相减

    void getcount()

    {
```

```
        return *use_count
    }
};
template <typename T>
int SmartPtr<T>::operator-(SmartPtr<T> &t1, SmartPtr<T> &t2)
{
    return t1.ptr-t2.ptr;
}
template <typename T>
SmartPtr<T>::SmartPtr(T *p)
{
    ptr=p;
    try
    {
        use_count=new int(1);
    }catch (...)
    {
        delete ptr;    //申请失败释放真实指针和引用计数的内存
        ptr= nullptr;
        delete use_count;
        use_count= nullptr;

    }
```

```
}

template <typename T>

SmartPtr<T>::SmartPtr(const SmartPtr<T> &orig) //复制构造函数

{

    use_count=orig.use_count;//引用计数保存在一块内存，所有的 SmarPtr 对象的引用计数都指向这里

    this->ptr=orig.ptr;

    ++(*use_count);//当前对象的引用计数加 1



}



template <typename T>

SmartPtr<T>& SmartPtr<T>::operator=(const SmartPtr<T> &rhs)

{

//重载=运算符，例如 SmartPtr<int>p,q; p=q;这个语句中，首先给 q 指向的对象的引用计数加 1，因为 p 重新指向了 q 所指的对象，所以 p 需要先给原来的对象的引用计数减 1，如果减一后为 0，先释放掉 p 原来指向的内存，然后讲 q 指向的对象的引用计数加 1 后赋值给 p

    ++*(rhs.use_count);

    if((--*(use_count))==0)

    {



        delete ptr;

        ptr= nullptr;

        delete use_count;

        use_count= nullptr;

    }

    ptr=rhs.ptr;
```

```
    *use_count=*(rhs.use_count);

    return *this;



}

template <typename T>

SmartPtr<T>::~SmartPtr()

{

    getcount();

    if(--(*use_count)==0)  //SmartPtr 的对象会在其生命周期结束的时候调用其析构函数，在析构函数中检测当前对象的引用计数是不是只有正在结束生命周期的这个 SmartPtr 引用，如果是，就释放掉，如果不是，就还有其他的 SmartPtr 引用当前对象，就等待其他的 SmartPtr 对象在其生命周期结束的时候调用析构函数释放掉

    {

        getcount();

        delete ptr;

        ptr= nullptr;

        delete use_count;

        use_count=nullptr;

    }

}



template <typename T>

T SmartPtr<T>::operator*()

{

    return *ptr;

}

template <typename T>
```

```
T*  SmartPtr<T>::operator->()
{
    return ptr;
}
template <typename T>
T* SmartPtr<T>::operator+(int i)
{
    T *temp=ptr+i;
    return temp;
}
}
```

31、以下四行代码的区别是什么？

const char * arr = "123";
char * brr = "123";
const char crr[] = "123";
char drr[] = "123";

参考回答：

```
const char * arr = "123";
//字符串 123 保存在常量区，const 本来是修饰 arr 指向的值不能通过 arr 去修改，但是字符串“123”在常量区，本来就不能改变，所以加不加 const 效果都一样
char * brr = "123";
//字符串 123 保存在常量区,这个 arr 指针指向的是同一个位置,同样不能通过 brr 去修改"123"的值
const char crr[] = "123";
//这里 123 本来是在栈上的，但是编译器可能会做某些优化，将其放到常量区
char drr[] = "123";
```

```
//字符串 123 保存在栈区，可以通过 drr 去修改
```

32、请你来说一下 C++里是怎么定义常量的？常量存放在内存的哪个位置？

参考回答：

常量在 C++里的定义就是一个 top-level const 加上对象类型，常量定义必须初始化。对于局部对象，常量存放在栈区，对于全局对象，常量存放在全局/静态存储区。对于字面值常量，常量存放在常量存储区。

33、请你来回答一下 const 修饰成员函数的目的是什么？

参考回答：

const 修饰的成员函数表明函数调用不会对对象做出任何更改，事实上，如果确认不会对对象做更改，就应该为函数加上 const 限定，这样无论 const 对象还是普通对象都可以调用该函数。

34、如果同时定义了两个函数，一个带 const，一个不带，会有问题吗？

参考回答：

不会，这相当于函数的重载。

35、请你来说一说隐式类型转换

参考回答：

首先，对于内置类型，低精度的变量给高精度变量赋值会发生隐式类型转换，其次，对于只存在单个参数的构造函数的对象构造来说，函数调用可以直接使用该参数传入，编译器会自动调用其构造函数生成临时对象。

36、说说你了解的类型转换

参考回答：

reinterpret_cast：可以用于任意类型的指针之间的转换，对转换的结果不做任何保证
dynamic_cast：这种其实也是不被推荐使用的，更多使用 static_cast，dynamic 本身只能用于存在虚函数的父子关系的强制类型转换，对于指针，转换失败则返回 nullptr，对于引用，转换失败会抛出异常
const_cast：对于未定义 const 版本的成员函数，我们通常需要使用 const_cast 来去除 const

引用对象的 const，完成函数调用。另外一种使用方式，结合 static_cast，可以在非 const 版本的成员函数内添加 const，调用完 const 版本的成员函数后，再使用 const_cast 去除 const 限定。
static_cast：完成基础数据类型；同一个继承体系中类型的转换；任意类型与空指针类型 void* 之间的转换。

37、请你来说一说 C++函数栈空间的最大值

参考回答：

默认是 1M，不过可以调整

38、请你来说一说 extern “C”

参考回答：

C++调用 C 函数需要 extern C，因为 C 语言没有函数重载。

39、请你回答一下 new/delete 与 malloc/free 的区别是什么

参考回答：

首先，new/delete 是 C++的关键字，而 malloc/free 是 C 语言的库函数，后者使用必须指明申请内存空间的大小，对于类类型的对象，后者不会调用构造函数和析构函数

40、请你说说你了解的 RTTI

参考回答：

运行时类型检查，在 C++层面主要体现在 dynamic_cast 和 typeid,VS 中虚函数表的-1 位置存放了指向 type_info 的指针。对于存在虚函数的类型，typeid 和 dynamic_cast 都会去查询 type_info

41、请你说说虚函数表具体是怎样实现运行时多态的?

参考回答：

子类若重写父类虚函数，虚函数表中，该函数的地址会被替换，对于存在虚函数的类的对象，在 VS 中，对象的对象模型的头部存放指向虚函数表的指针，通过该机制实现多态。

42、请你说说 C 语言是怎么进行函数调用的？

参考回答：

每一个函数调用都会分配函数栈，在栈内进行函数执行过程。调用前，先把返回地址压栈，然后把当前函数的 esp 指针压栈。

43、请你说说 C 语言参数压栈顺序？

参考回答：

从右到左

44、请你说说 C++如何处理返回值？

参考回答：

生成一个临时变量，把它的引用作为函数参数传入函数内。

45、请你回答一下 C++中拷贝赋值函数的形参能否进行值传递？

参考回答：

不能。如果是这种情况下，调用拷贝构造函数的时候，首先要将实参传递给形参，这个传递的时候又要调用拷贝构造函数。。如此循环，无法完成拷贝，栈也会满。

46、请你回答一下 malloc 与 new 区别

参考回答：

malloc 需要给定申请内存的大小，返回的指针需要强转。
new 会调用构造函数，不用指定内存大小，返回的指针不用强转。

47、请你说一说 select

参考回答：

select 在使用前，先将需要监控的描述符对应的 bit 位置 1，然后将其传给 select,当有任何一个事件发生时，select 将会返回所有的描述符，需要在应用程序自己遍历去检查哪个描述符上有事件发生，效率很低，并且其不断在内核态和用户态进行描述符的拷贝，开销很大

48、请你说说 fork,wait,exec 函数

参考回答：

父进程产生子进程使用 fork 拷贝出来一个父进程的副本，此时只拷贝了父进程的页表，两个进程都读同一块内存，当有进程写的时候使用写实拷贝机制分配内存，exec 函数可以加载一个 elf 文件去替换父进程，从此父进程和子进程就可以运行不同的程序了。fork 从父进程返回子进程的 pid，从子进程返回 0.调用了 wait 的父进程将会发生阻塞，直到有子进程状态改变,执行成功返回 0，错误返回-1。exec 执行成功则子进程从新的程序开始运行，无返回值，执行失败返回-1

49、请你回答一下静态函数和虚函数的区别

参考回答：

静态函数在编译的时候就已经确定运行时机，虚函数在运行的时候动态绑定。虚函数因为用了虚函数表机制，调用的时候会增加一次内存开销

50、请你说一说重载和覆盖

参考回答：

重载：两个函数名相同，但是参数列表不同（个数，类型），返回值类型没有要求，在同一作用域中
重写：子类继承了父类，父类中的函数是虚函数，在子类中重新定义了这个虚函数，这种情况是重写

51、请你说一说 static 关键字

参考回答：

1.加了 static 关键字的全局变量只能在本文件中使用。例如在 a.c 中定义了 static int a=10;那么在 b.c 中用 extern int a 是拿不到 a 的值得，a 的作用域只在 a.c 中。
2.static 定义的静态局部变量分配在数据段上，普通的局部变量分配在栈上，会因为函数栈帧的释放而被释放掉。
3. 对一个类中成员变量和成员函数来说，加了 static 关键字，则此变量/函数就没有了 this 指针了，必须通过类名才能访问

52、请你说一说 strcpy 和 strlen

参考回答：

strcpy 是字符串拷贝函数，原型：

char *strcpy(char* dest, const char *src);

从 src 逐字节拷贝到 dest，直到遇到'\0'结束，因为没有指定长度，可能会导致拷贝越界，造成缓冲区溢出漏洞,安全版本是 strncpy 函数。
strlen 函数是计算字符串长度的函数，返回从开始到'\0'之间的字符个数。

2、容器和算法

1、请你来说一下 map 和 set 有什么区别，分别又是怎么实现的？

考察点：C++ STL 基础知识，STL 底层实现

参考回答：

map 和 set 都是 C++的关联容器，其底层实现都是红黑树（RB-Tree）。由于 map 和 set 所开放的各种操作接口，RB-tree 也都提供了，所以几乎所有的 map 和 set 的操作行为，都只是转调 RB-tree 的操作行为。

map 和 set 区别在于：

（1）map 中的元素是 key-value（关键字—值）对：关键字起到索引的作用，值则表示与索引相关联的数据；Set 与之相对就是关键字的简单集合，set 中每个元素只包含一个关键字。

（2）set 的迭代器是 const 的，不允许修改元素的值；map 允许修改 value，但不允许修改 key。其原因是因为 map 和 set 是根据关键字排序来保证其有序性的，如果允许修改 key 的话，那么首先需要删除该键，然后调节平衡，再插入修改后的键值，调节平衡，如此一来，严重破坏了 map 和 set 的结构，导致 iterator 失效，不知道应该指向改变前的位置，还是指向改变后的位置。所以 STL 中将 set 的迭代器设置成 const，不允许修改迭代器的值；而 map 的迭代器则不允许修改 key 值，允许修改 value 值。

（3）map 支持下标操作，set 不支持下标操作。map 可以用 key 做下标，map 的下标运算符[]将关键码作为下标去执行查找，如果关键码不存在，则插入一个具有该关键码和 mapped_type 类型默认值的元素至 map 中，因此下标运算符[ ]在 map 应用中需要慎用，const_map 不能用，只希望确定某一个关键值是否存在而不希望插入元素时也不应该使用，mapped_type 类型没有默认值也不应该使用。如果 find 能解决需要，尽可能用 find。

2、请你来介绍一下 STL 的 allocaotr

考点：STL 分配器

参考回答：

STL 的分配器用于封装 STL 容器在内存管理上的底层细节。在 C++中，其内存配置和释放如下：

new 运算分两个阶段：(1)调用::operator new 配置内存;(2)调用对象构造函数构造对象内容

delete 运算分两个阶段：(1)调用对象希构函数；(2)掉员工::operator delete 释放内存

为了精密分工，STL allocator 将两个阶段操作区分开来：内存配置有 alloc::allocate()负责，内存释放由 alloc::deallocate()负责；对象构造由::construct()负责，对象析构由::destroy()负责。

同时为了提升内存管理的效率，减少申请小内存造成的内存碎片问题，SGI STL 采用了两级配置器，当分配的空间大小超过 128B 时，会使用第一级空间配置器；当分配的空间大小小于 128B 时，将使用第二级空间配置器。第一级空间配置器直接使用 malloc()、realloc()、free()函数进行内存空间的分配和释放，而第二级空间配置器采用了内存池技术，通过空闲链表来管理内存。

3、请你来说一说 STL 迭代器删除元素

参考回答：

这个主要考察的是迭代器失效的问题。1.对于序列容器 vector,deque 来说，使用 erase(itertor)后，后边的每个元素的迭代器都会失效，但是后边每个元素都会往前移动一个位置，但是 erase 会返回下一个有效的迭代器；2.对于关联容器 map set 来说，使用了 erase(iterator)后，当前元素的迭代器失效，但是其结构是红黑树，删除当前元素的，不会影响到下一个元素的迭代器，所以在调用 erase 之前，记录下一个元素的迭代器即可。3.对于 list 来说，它使用了不连续分配的内存，并且它的 erase 方法也会返回下一个有效的 iterator，因此上面两种正确的方法都可以使用。

4、请你说一说 STL 中 MAP 数据存放形式

参考回答：

红黑树。unordered map 底层结构是哈希表

5、请你讲讲 STL 有什么基本组成

参考回答：

STL 主要由：以下几部分组成：
容器 迭代器 仿函数 算法 分配器 配接器

他们之间的关系：分配器给容器分配存储空间，算法通过迭代器获取容器中的内容，仿函数可以协助算法完成各种操作，配接器用来套接适配仿函数

6、请你说说 STL 中 map 与 unordered_map

参考回答：

1、Map

映射，map 的所有元素都是 pair，同时拥有实值（value）和键值（key）。pair 的第一元素被视为键值，第二元素被视为实值。所有元素都会根据元素的键值自动被排序。不允许键值重复。

底层实现：红黑树

适用场景：有序键值对不重复映射

2、Multimap

多重映射。multimap 的所有元素都是 pair，同时拥有实值（value）和键值（key）。pair 的第一元素被视为键值，第二元素被视为实值。所有元素都会根据元素的键值自动被排序。允许键值重复。

底层实现：红黑树

适用场景：有序键值对可重复映射

7、请你说一说 vector 和 list 的区别，应用，越详细越好

参考回答：

1、概念：

1）Vector

连续存储的容器，动态数组，在堆上分配空间

底层实现：数组

两倍容量增长：

vector 增加（插入）新元素时，如果未超过当时的容量，则还有剩余空间，那么直接添加到最后（插入指定位置），然后调整迭代器。

如果没有剩余空间了，则会重新配置原有元素个数的两倍空间，然后将原空间元素通过复制的方式初始化新空间，再向新空间增加元素，最后析构并释放原空间，之前的迭代器会失效。

性能：

访问： O(1)

插入：在最后插入（空间够）：很快

在最后插入（空间不够）：需要内存申请和释放，以及对之前数据进行拷贝。

在中间插入（空间够）：内存拷贝

在中间插入（空间不够）：需要内存申请和释放，以及对之前数据进行拷贝。

删除：在最后删除：很快

在中间删除：内存拷贝

适用场景：经常随机访问，且不经常对非尾节点进行插入删除。

2、List

动态链表，在堆上分配空间，每插入一个元数都会分配空间，每删除一个元素都会释放空间。

底层：双向链表

性能：

访问：随机访问性能很差，只能快速访问头尾节点。

插入：很快，一般是常数开销

删除：很快，一般是常数开销

适用场景：经常插入删除大量数据

2、区别：

1）vector 底层实现是数组；list 是双向 链表。

2）vector 支持随机访问，list 不支持。

3）vector 是顺序内存，list 不是。

4）vector 在中间节点进行插入删除会导致内存拷贝，list 不会。

5）vector 一次性分配好内存，不够时才进行 2 倍扩容；list 每次插入新节点都会进行内存申请。

6）vector 随机访问性能好，插入删除性能差；list 随机访问性能差，插入删除性能好。

3、应用

vector 拥有一段连续的内存空间，因此支持随机访问，如果需要高效的随即访问，而不在乎插入和删除的效率，使用 vector。

list 拥有一段不连续的内存空间，如果需要高效的插入和删除，而不关心随机访问，则应使用 list。

8、请你来说一下 STL 中迭代器的作用，有指针为何还要迭代器

考察点：

参考回答：

1、迭代器

Iterator（迭代器）模式又称 Cursor（游标）模式，用于提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。或者这样说可能更容易理解：Iterator 模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序（由 iterator 提供的方法）访问聚合对象中的各个元素。

由于 Iterator 模式的以上特性：与聚合对象耦合，在一定程度上限制了它的广泛运用，一般仅用于底层聚合支持类，如 STL 的 list、vector、stack 等容器类及 ostream_iterator 等扩展 iterator。

2、迭代器和指针的区别

迭代器不是指针，是类模板，表现的像指针。他只是模拟了指针的一些功能，通过重载了指针的一些操作符，->、*、++、--等。迭代器封装了指针，是一个“可遍历 STL（ Standard Template Library）容器内全部或部分元素”的对象， 本质是封装了原生指针，是指针概念的一种提升（lift），提供了比指针更高级的行为，相当于一种智能指针，他可以根据不同类型的数据结构来实现不同的++，--等操作。

迭代器返回的是对象引用而不是对象的值，所以 cout 只能输出迭代器使用*取值后的值而不能直接输出其自身。

3、迭代器产生原因

Iterator 类的访问方式就是把不同集合类的访问逻辑抽象出来，使得不用暴露集合内部的结构而达到循环遍历集合的效果。

9、请你说一说 epoll 原理

参考回答：

调用顺序：

```
int epoll_create(int size);

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

```
int epoll_wait(int epfd, struct epoll_event *events,int maxevents, int timeout);
```

首先创建一个 epoll 对象，然后使用 epoll_ctl 对这个对象进行操作，把需要监控的描述添加进去，这些描述如将会以 epoll_event 结构体的形式组成一颗红黑树，接着阻塞在 epoll_wait，进入大循环，当某个 fd 上有事件发生时，内核将会把其对应的结构体放入到一个链表中，返回有事件发生的链表。

10、请你说一说 STL 迭代器是怎么删除元素的呢

参考回答：

这个主要考察的是迭代器失效的问题。1.对于序列容器 vector,deque 来说，使用 erase(itertor)后，后边的每个元素的迭代器都会失效，但是后边每个元素都会往前移动一个位置，但是 erase 会返回下一个有效的迭代器；2.对于关联容器 map set 来说，使用了 erase(iterator)后，当前元素的迭代器失效，但是其结构是红黑树，删除当前元素的，不会影响到下一个元素的迭代器，所以在调用 erase 之前，记录下一个元素的迭代器即可。3.对于 list 来说，它使用了不连续分配的内存，并且它的 erase 方法也会返回下一个有效的 iterator，因此上面两种正确的方法都可以使用。

11、请你说一说 STL 中 MAP 数据存放形式

参考回答：

红黑树。unordered map 底层结构是哈希表

12、n 个整数的无序数组，找到每个元素后面比它大的第一个数，要求时间复杂度为 O(N)

参考回答：

```
vector<int> findMax(vector<int>num)
{


    if(num.size()==0)return num;
    vector<int>res(num.size());
    int i=0;
    stack<int>s;
    while(i<num.size())
    {
```

```
            if(s.empty()||num[s.top()]>=num[i])
            {
                s.push(i++);
            }
            else
            {
                res[s.top()]=num[i];
                s.pop();
            }
        }
        while(!s.empty())
        {
            res[s.top()]=INT_MAX;
            s.pop();
        }
        for(int i=0; i<res.size(); i++)
            cout<<res[i]<<endl;
        return res;
    }
```

13、请你回答一下 STL 里 resize 和 reserve 的区别

参考回答：

resize()：改变当前容器内含有元素的数量(size())，eg: vector<int>v; v.resize(len);v 的 size 变为 len,如果原来 v 的 size 小于 len，那么容器新增（len-size）个元素，元素的值为默认为 0.当 v.push_back(3);之后，则是 3 是放在了 v 的末尾，即下标为 len，此时容器是 size 为 len+1；
reserve()：改变当前容器的最大容量（capacity）,它不会生成元素，只是确定这个容器允许放

入多少对象，如果 reserve(len)的值大于当前的 capacity()，那么会重新分配一块能存 len 个对象的空间，然后把之前 v.size()个对象通过 copy construtor 复制过来，销毁之前的内存；测试代码如下：

```
#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> a;

a.reserve(100);

a.resize(50);

cout<<a.size()<<"  "<<a.capacity()<<endl;

        //50  100

a.resize(150);

cout<<a.size()<<"  "<<a.capacity()<<endl;

        //150  200

a.reserve(50);

cout<<a.size()<<"  "<<a.capacity()<<endl;

        //150  200

a.resize(50);

cout<<a.size()<<"  "<<a.capacity()<<endl;

        //50  200

}
```

14、请你说一说 stl 里面 set 和 map 怎么实现的

考察点：C/C++，STL

参考回答：

1、set

集合，所有元素都会根据元素的值自动被排序，且不允许重复。

底层实现：红黑树

set 底层是通过红黑树（RB-tree）来实现的，由于红黑树是一种平衡二叉搜索树，自动排序的效果很不错，所以标准的 STL 的 set 即以 RB-Tree 为底层机制。又由于 set 所开放的各种操作接口，RB-tree 也都提供了，所以几乎所有的 set 操作行为，都只有转调用 RB-tree 的操作行为而已。

适用场景：有序不重复集合

2、map

映射。map 的所有元素都是 pair，同时拥有实值（value）和键值（key）。pair 的第一元素被视为键值，第二元素被视为实值。所有元素都会根据元素的键值自动被排序。不允许键值重复。

底层：红黑树

适用场景：有序键值对不重复映射

3、类和数据抽象

1、请你来说一下 C++中类成员的访问权限

参考回答：

C++通过 public、protected、private 三个关键字来控制成员变量和成员函数的访问权限，它们分别表示公有的、受保护的、私有的，被称为成员访问限定符。在类的内部（定义类的代码内部），无论成员被声明为 public、protected 还是 private，都是可以互相访问的，没有访问权限的限制。在类的外部（定义类的代码之外），只能通过对象访问成员，并且通过对象只能访问 public 属性的成员，不能访问 private、protected 属性的成员

2、请你来说一下 C++中 struct 和 class 的区别

考察点：C++

参考回答：

在 C++中，可以用 struct 和 class 定义类，都可以继承。区别在于：structural 的默认继承权限和默认访问权限是 public，而 class 的默认继承权限和默认访问权限是 private。

另外，class 还可以定义模板类形参，比如 template <class T, int i>。

3、请你回答一下 C++类内可以定义引用数据成员吗？

参考回答：

可以，必须通过成员函数初始化列表初始化。

4、面向对象与泛型编程

1、请你回答一下什么是右值引用，跟左值又有什么区别？

考察点：C++

参考回答：右值引用是 C++11 中引入的新特性 ，它实现了转移语义和精确传递。它的主要目的有两个方面：

1. 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。
2. 能够更简洁明确地定义泛型函数。

左值和右值的概念：

左值：能对表达式取地址、或具名对象/变量。一般指表达式结束后依然存在的持久对象。

右值：不能对表达式取地址，或匿名对象。一般指表达式结束就不再存在的临时对象。

右值引用和左值引用的区别：

1. 左值可以寻址，而右值不可以。
2. 左值可以被赋值，右值不可以被赋值，可以用来给左值赋值。

3. 左值可变,右值不可变（仅对基础类型适用，用户自定义类型右值引用可以通过成员函数改变）。

5、编译与底层

1、请你来说一下一个 C++源文件从文本到可执行文件经历的过程？

参考回答：

对于 C++源文件，从文本到可执行文件一般需要四个过程：

预处理阶段：对源代码文件中文件包含关系（头文件）、预编译语句（宏定义）进行分析和替换，生成预编译文件。

编译阶段：将经过预处理后的预编译文件转换成特定汇编代码，生成汇编文件

汇编阶段：将编译阶段生成的汇编文件转化成机器码，生成可重定位目标文件

链接阶段：将多个目标文件及所需要的库连接成最终的可执行目标文件

2、请你来回答一下 include 头文件的顺序以及双引号”“和尖括号<>的区别？

考点：编译过程

参考回答：

Include 头文件的顺序:对于 include 的头文件来说,如果在文件 a.h 中声明一个在文件 b.h 中定义的变量，而不引用 b.h。那么要在 a.c 文件中引用 b.h 文件，并且要先引用 b.h，后引用 a.h,否则汇报变量类型未声明错误。

双引号和尖括号的区别：编译器预处理阶段查找头文件的路径不一样。

对于使用双引号包含的头文件，查找头文件路径的顺序为：

当前头文件目录

编译器设置的头文件路径（编译器可使用-I 显式指定搜索路径）

系统变量 CPLUS_INCLUDE_PATH/C_INCLUDE_PATH 指定的头文件路径

对于使用尖括号包含的头文件，查找头文件的路径顺序为：

编译器设置的头文件路径（编译器可使用-I 显式指定搜索路径）

系统变量 CPLUS_INCLUDE_PATH/C_INCLUDE_PATH 指定的头文件路径

3、请你回答一下 malloc 的原理，另外 brk 系统调用和 mmap 系统调用的作用分别是什么？

考点：malloc 底层原理、内存管理

参考回答：

Malloc 函数用于动态分配内存。为了减少内存碎片和系统调用的开销，malloc 其采用内存池的方式，先申请大块内存作为堆区，然后将堆区分为多个内存块，以块作为内存管理的基本单位。当用户申请内存时，直接从堆区分配一块合适的空闲块。Malloc 采用隐式链表结构将堆区分成连续的、大小不一的块，包含已分配块和未分配块；同时 malloc 采用显示链表结构来管理所有的空闲块，即使用一个双向链表将空闲块连接起来，每一个空闲块记录了一个连续的、未分配的地址。

当进行内存分配时，Malloc 会通过隐式链表遍历所有的空闲块，选择满足要求的块进行分配；当进行内存合并时，malloc 采用边界标记法，根据每个块的前后块是否已经分配来决定是否进行块合并。

Malloc 在申请内存时，一般会通过 brk 或者 mmap 系统调用进行申请。其中当申请内存小于 128K 时，会使用系统函数 brk 在堆区中分配；而当申请内存大于 128K 时，会使用系统函数 mmap 在映射区分配。

4、请你说一说 C++的内存管理是怎样的？

考点：虚拟内存分布

参考回答：

在 C++中，虚拟内存分为代码段、数据段、BSS 段、堆区、文件映射区以及栈区六部分。

代码段:包括只读存储区和文本区，其中只读存储区存储字符串常量，文本区存储程序的机器代码。

数据段：存储程序中已初始化的全局变量和静态变量

bss 段：存储未初始化的全局变量和静态变量（局部+全局），以及所有被初始化为 0 的全局变量和静态变量。

堆区： 调用 new/malloc 函数时在堆区动态分配内存，同时需要调用 delete/free 来手动释放申请的内存。

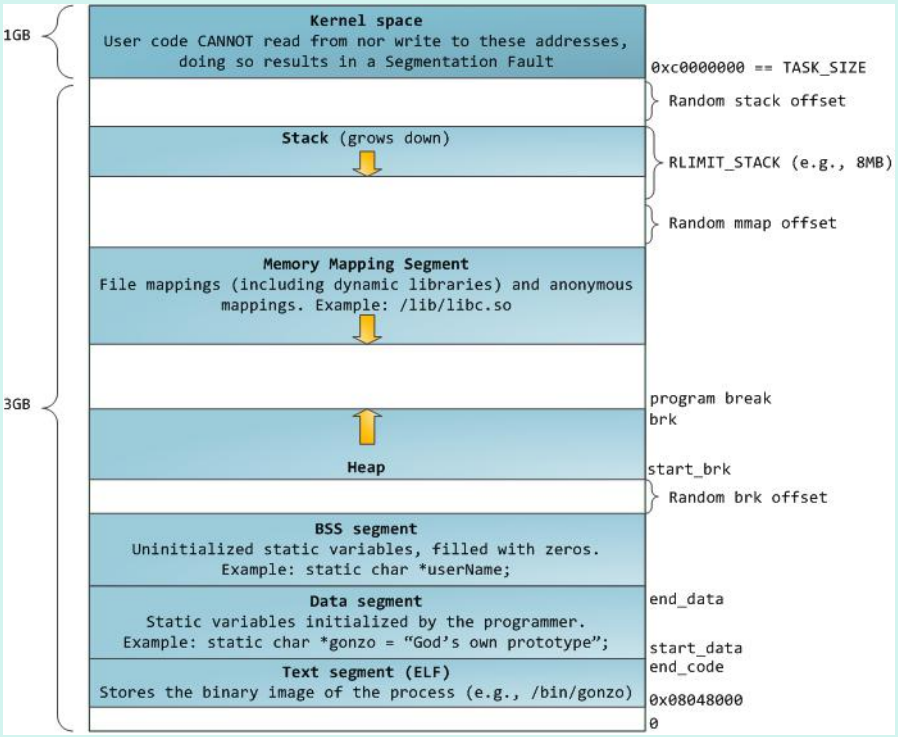
映射区:存储动态链接库以及调用 mmap 函数进行的文件映射

栈：使用栈空间存储函数的返回地址、参数、局部变量、返回值

5、请你来说一下 C++/C 的内存分配

知识点：内存分配

参考回答：



32bitCPU 可寻址 4G 线性空间，每个进程都有各自独立的 4G 逻辑地址，其中 0~3G 是用户态空间，3~4G 是内核空间，不同进程相同的逻辑地址会映射到不同的物理地址中。其逻辑地址其划分如下：

各个段说明如下：

3G 用户空间和 1G 内核空间

静态区域：

text segment(代码段):包括只读存储区和文本区，其中只读存储区存储字符串常量，文本区存储程序的机器代码。

data segment(数据段)：存储程序中已初始化的全局变量和静态变量

bss segment：存储未初始化的全局变量和静态变量（局部+全局），以及所有被初始化为 0 的全局变量和静态变量，对于未初始化的全局变量和静态变量，程序运行 main 之前时会统一清零。即未初始化的全局变量编译器会初始化为 0

动态区域：

heap（堆）： 当进程未调用 malloc 时是没有堆段的，只有调用 malloc 时采用分配一个堆，并且在程序运行过程中可以动态增加堆大小(移动 break 指针)，从低地址向高地址增长。分配小内存时使用该区域。 堆的起始地址由 mm_struct 结构体中的 start_brk 标识，结束地址由 brk 标识。

memory mapping segment(映射区):存储动态链接库等文件映射、申请大内存（malloc 时调用 mmap 函数）

stack（栈）：使用栈空间存储函数的返回地址、参数、局部变量、返回值，从高地址向低地址增长。在创建进程时会有一个最大栈大小，Linux 可以通过 ulimit 命令指定。

6、请你回答一下如何判断内存泄漏？

考点：内存泄漏

参考回答：

内存泄漏通常是由于调用了 malloc/new 等内存申请的操作,但是缺少了对应的 free/delete。为了判断内存是否泄露，我们一方面可以使用 linux 环境下的内存泄漏检查工具 Valgrind,另一方面我们在写代码时可以添加内存申请和释放的统计功能,统计当前申请和释放的内存是否一致,以此来判断内存是否泄露。

7、请你来说一下什么时候会发生段错误

考点：内存访问 段错误

参考回答：

段错误通常发生在访问非法内存地址的时候，具体来说分为以下几种情况：

使用野指针

试图修改字符串常量的内容

8、请你来回答一下什么是 memory leak，也就是内存泄漏

考察点：内存泄露

参考回答：

内存泄漏(memory leak)是指由于疏忽或错误造成了程序未能释放掉不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。

内存泄漏的分类：

1. 堆内存泄漏 （Heap leak）。对内存指的是程序运行中根据需要分配通过 malloc,realloc new 等从堆中分配的一块内存，再是完成后必须通过调用对应的 free 或者 delete 删掉。如果程序的设计的错误导致这部分内存没有被释放，那么此后这块内存将不会被使用，就会产生 Heap Leak.

2. 系统资源泄露（Resource Leak）。主要指程序使用系统分配的资源比如 Bitmap,handle ,SOCKET 等没有使用相应的函数释放掉，导致系统资源的浪费，严重可导致系统效能降低，系统运行不稳定。

3. 没有将基类的析构函数定义为虚函数。当基类指针指向子类对象时，如果基类的析构函数不是 virtual，那么子类的析构函数将不会被调用，子类的资源没有正确是释放，因此造成内存泄露。

9、请你来回答一下 new 和 malloc 的区别

考察点：C/C++

参考回答：

1、new 分配内存按照数据类型进行分配，malloc 分配内存按照指定的大小分配；

2、new 返回的是指定对象的指针，而 malloc 返回的是 void*，因此 malloc 的返回值一般都需要进行类型转化。

3、new 不仅分配一段内存，而且会调用构造函数，malloc 不会。

4、new 分配的内存要用 delete 销毁，malloc 要用 free 来销毁；delete 销毁的时候会调用对象的析构函数，而 free 则不会。

5、new 是一个操作符可以重载，malloc 是一个库函数。

6、malloc 分配的内存不够的时候，可以用 realloc 扩容。扩容的原理？new 没用这样操作。

7、new 如果分配失败了会抛出 bad_malloc 的异常，而 malloc 失败了会返回 NULL。

8、申请数组时： new[]一次分配所有内存，多次调用构造函数，搭配使用 delete[],delete[]多次调用析构函数，销毁数组中的每个对象。而 malloc 则只能 sizeof(int) * n。

10、请你来说一下共享内存相关 api

考察点：STL

参考回答：

Linux 允许不同进程访问同一个逻辑内存，提供了一组 API，头文件在 sys/shm.h 中。

1）新建共享内存 shmget

int shmget(key_t key,size_t size,int shmflg);

key：共享内存键值，可以理解为共享内存的唯一性标记。

size：共享内存大小

shmflag：创建进程和其他进程的读写权限标识。

返回值：相应的共享内存标识符，失败返回-1

2）连接共享内存到当前进程的地址空间 shmat

void *shmat(int shm_id,const void *shm_addr,int shmflg);

shm_id：共享内存标识符

shm_addr：指定共享内存连接到当前进程的地址，通常为 0，表示由系统来选择。

shmflg：标志位

返回值：指向共享内存第一个字节的指针，失败返回-1

3）当前进程分离共享内存 shmdt

int shmdt(const void *shmaddr);

4）控制共享内存 shmctl

和信号量的 semctl 函数类似，控制共享内存

int shmctl(int shm_id,int command,struct shmid_ds *buf);

shm_id：共享内存标识符

command：有三个值

IPC_STAT:获取共享内存的状态，把共享内存的 shmid_ds 结构复制到 buf 中。

IPC_SET:设置共享内存的状态，把 buf 复制到共享内存的 shmid_ds 结构。

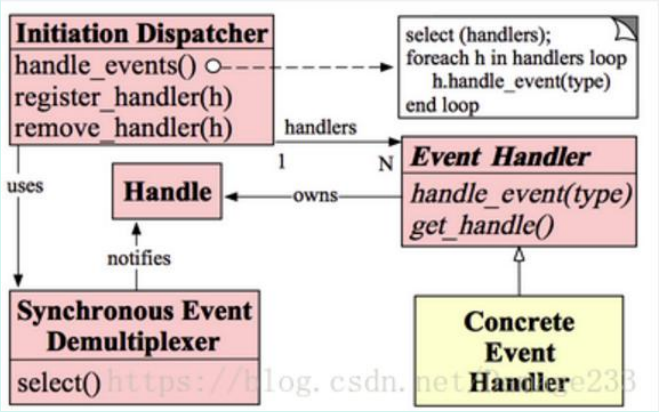
IPC_RMID:删除共享内存

buf：共享内存管理结构体。

11、请你来说一下 reactor 模型组成

参考回答：

reactor 模型要求主线程只负责监听文件描述上是否有事件发生，有的话就立即将该事件通知工作线程，除此之外，主线程不做任何其他实质性的工作，读写数据、接受新的连接以及处理客户请求均在工作线程中完成。其模型组成如下：



1）Handle：即操作系统中的句柄，是对资源在操作系统层面上的一种抽象，它可以是打开的文件、一个连接(Socket)、Timer 等。由于 Reactor 模式一般使用在网络编程中，因而这里一般指 Socket Handle，即一个网络连接。

2）Synchronous Event Demultiplexer（同步事件复用器）：阻塞等待一系列的 Handle 中的事件到来，如果阻塞等待返回，即表示在返回的 Handle 中可以不阻塞的执行返回的事件类型。这个模块一般使用操作系统的 select 来实现。

3）Initiation Dispatcher：用于管理 Event Handler，即 EventHandler 的容器，用以注册、移除 EventHandler 等；另外，它还作为 Reactor 模式的入口调用 Synchronous Event Demultiplexer 的 select 方法以阻塞等待事件返回，当阻塞等待返回时，根据事件发生的 Handle 将其分发给对应的 Event Handler 处理，即回调 EventHandler 中的 handle_event()方法。

4）Event Handler：定义事件处理方法：handle_event()，以供 InitiationDispatcher 回调使用。

5）Concrete Event Handler：事件 EventHandler 接口，实现特定事件处理逻辑。

12、请自己设计一下如何采用单线程的方式处理高并发

考点：I/O 复用 异步回调

参考回答：

在单线程模型中，可以采用 I/O 复用来提高单线程处理多个请求的能力，然后再采用事件驱动模型，基于异步回调来处理事件来

13、请你说说 C++如何处理内存泄漏？

参考回答：

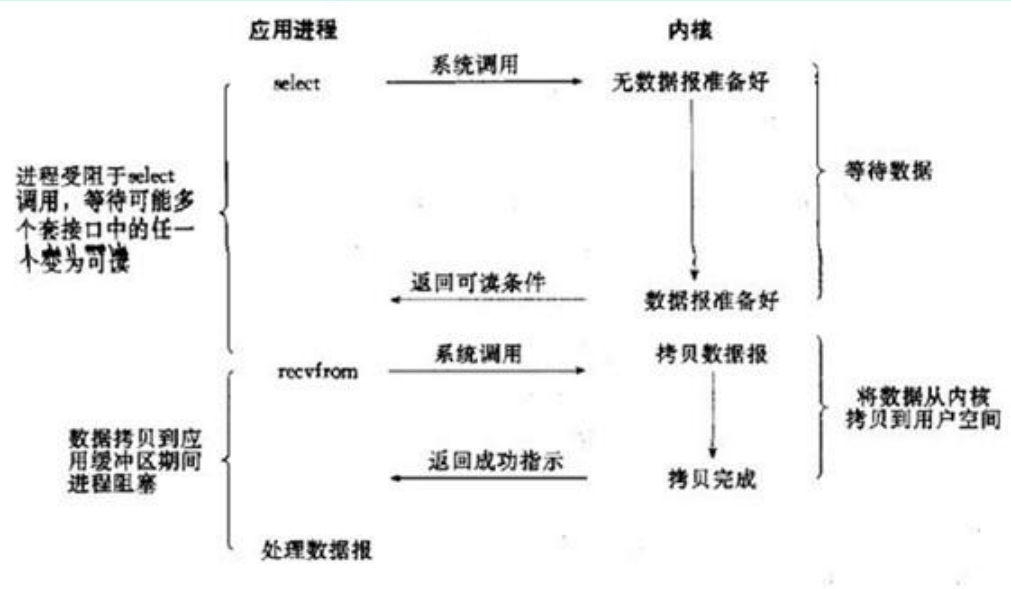
使用 varglind，mtrace 检测

14、请你说说 select，epoll 的区别，原理，性能，限制都说一说

1）IO 多路复用

IO 复用模型在阻塞 IO 模型上多了一个 select 函数，select 函数有一个参数是文件描述符集合，意思就是对这些的文件描述符进行循环监听，当某个文件描述符就绪的时候，就对这个文件描述符进行处理。

这种 IO 模型是属于阻塞的 IO。但是由于它可以对多个文件描述符进行阻塞监听，所以它的效率比阻塞 IO 模型高效。



IO 多路复用就是我们说的 select，poll，epoll。select/epoll 的好处就在于单个 process 就可以同时处理多个网络连接的 IO。它的基本原理就是 select，poll，epoll 这个 function 会不断的轮询所负责的所有 socket，当某个 socket 有数据到达了，就通知用户进程。

当用户进程调用了 select，那么整个进程会被 block，而同时，kernel 会“监视”所有 select 负责的 socket，当任何一个 socket 中的数据准备好了，select 就会返回。这个时候用户进程再调用 read 操作，将数据从 kernel 拷贝到用户进程。

所以，I/O 多路复用的特点是通过一种机制一个进程能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回。

I/O 多路复用和阻塞 I/O 其实并没有太大的不同，事实上，还更差一些。因为这里需要使用两个 system call (select 和 recvfrom)，而 blocking IO 只调用了一个 system call (recvfrom)。但是，用 select 的优势在于它可以同时处理多个 connection。

所以，如果处理的连接数不是很高的话，使用 select/epoll 的 web server 不一定比使用 multi-threading + blocking IO 的 web server 性能更好，可能延迟还更大。select/epoll 的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）

在 IO multiplexing Model 中，实际中，对于每一个 socket，一般都设置成为 non-blocking，但是，如上图所示，整个用户的 process 其实是一直被 block 的。只不过 process 是被 select 这个函数 block，而不是被 socket IO 给 block。

2、select

select:是最初解决 IO 阻塞问题的方法。用结构体 fd_set 来告诉内核监听多个文件描述符，该结构体被称为描述符集。由数组来维持哪些描述符被置位了。对结构体的操作封装在三个宏定义中。通过轮寻来查找是否有描述符要被处理。

存在的问题：

1. 内置数组的形式使得 select 的最大文件数受限与 FD_SIZE；

2. 每次调用 select 前都要重新初始化描述符集，将 fd 从用户态拷贝到内核态，每次调用 select 后，都需要将 fd 从内核态拷贝到用户态；

3. 轮寻排查当文件描述符个数很多时，效率很低；

3、poll

poll：通过一个可变长度的数组解决了 select 文件描述符受限的问题。数组中元素是结构体，该结构体保存描述符的信息，每增加一个文件描述符就向数组中加入一个结构体，结构体只需要拷贝一次到内核态。poll 解决了 select 重复初始化的问题。轮寻排查的问题未解决。

4、epoll

epoll：轮寻排查所有文件描述符的效率不高，使服务器并发能力受限。因此，epoll 采用只返回状态发生变化的文件描述符，便解决了轮寻的瓶颈。

epoll 对文件描述符的操作有两种模式：LT（level trigger）和 ET（edge trigger）。LT 模式是默认模式

1. LT 模式

LT(level triggered)是缺省的工作方式，并且同时支持 block 和 no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的 fd 进行 IO 操作。如果你不作任何操作，内核还是会继续通知你的。

2. ET 模式

ET(edge-triggered)是高速工作方式，只支持 no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过 epoll 告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了(比如，你在发送，接收或者接收请求，或者发送接收的数据少于一定量时导致了一个 EWOULDBLOCK 错误）。但是请注意，如果一直不对这个 fd 作 IO 操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once)

ET 模式在很大程度上减少了 epoll 事件被重复触发的次数，因此效率要比 LT 模式高。epoll 工作在 ET 模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。

3、LT 模式与 ET 模式的区别如下：
LT 模式：当 epoll_wait 检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用 epoll_wait 时，会再次响应应用程序并通知此事件。
ET 模式：当 epoll_wait 检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用 epoll_wait 时，不会再次响应应用程序并通知此事件。

15、请你说一说 C++ STL 的内存优化

参考回答：

1）二级配置器结构

STL 内存管理使用二级内存配置器。

1、第一级配置器

第一级配置器以 malloc()，free()，realloc()等 C 函数执行实际的内存配置、释放、重新配置等操作，并且能在内存需求不被满足的时候，调用一个指定的函数。

一级空间配置器分配的是大于 128 字节的空间

如果分配不成功，调用句柄释放一部分内存

如果还不能分配成功，抛出异常

2、第二级配置器

在 STL 的第二级配置器中多了一些机制，避免太多小区块造成的内存碎片，小额区块带来的不仅是内存碎片，配置时还有额外的负担。区块越小，额外负担所占比例就越大。

3、分配原则

如果要分配的区块大于 128bytes，则移交给第一级配置器处理。

如果要分配的区块小于 128bytes，则以内存池管理（memory pool），又称之次层配置（sub-allocation）：每次配置一大块内存，并维护对应的 16 个空闲链表（free-list）。下次若有相同大小的内存需求，则直接从 free-list 中取。如果有小额区块被释放，则由配置器回收到 free-list 中。

当用户申请的空间小于 128 字节时，将字节数扩展到 8 的倍数，然后在自由链表中查找对应大小的子链表

如果在自由链表查找不到或者块数不够，则向内存池进行申请，一般一次申请 20 块

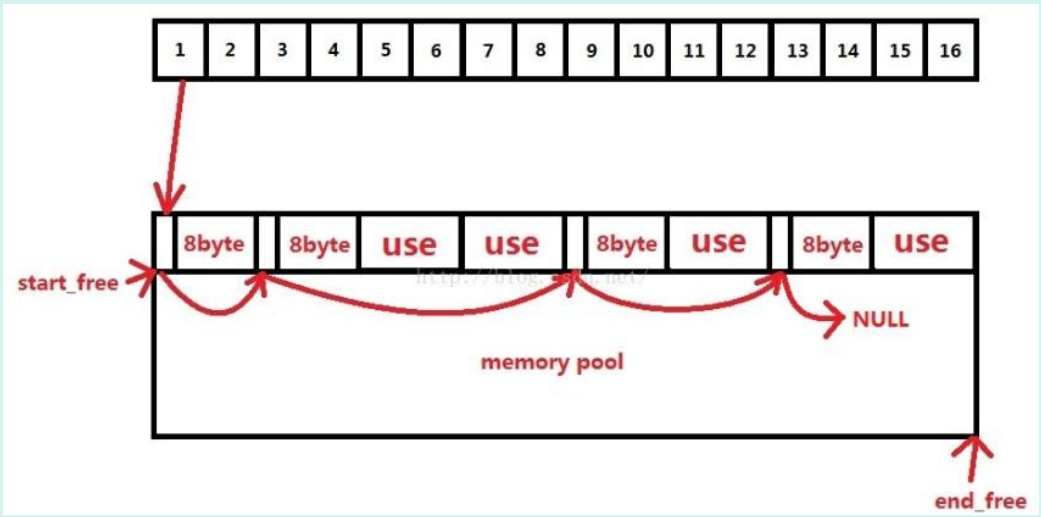
如果内存池空间足够，则取出内存

如果不够分配 20 块，则分配最多的块数给自由链表，并且更新每次申请的块数

如果一块都无法提供，则把剩余的内存挂到自由链表，然后向系统 heap 申请空间，如果申请失败，则看看自由链表还有没有可用的块，如果也没有，则最后调用一级空间配置器

2）二级内存池

二级内存池采用了 16 个空闲链表，这里的 16 个空闲链表分别管理大小为 8、16、24.....120、128 的数据块。这里空闲链表节点的设计十分巧妙，这里用了一个联合体既可以表示下一个空闲数据块（存在于空闲链表中）的地址，也可以表示已经被用户使用的数据块（不存在空闲链表中）的地址。



1、空间配置函数 allocate

首先先要检查申请空间的大小，如果大于 128 字节就调用第一级配置器，小于 128 字节就检查对应的空闲链表，如果该空闲链表中有可用数据块，则直接拿来用（拿取空闲链表中的第一个可用数据块，然后把该空闲链表的地址设置为该数据块指向的下一个地址），如果没有可用数据块，则调用 refill 重新填充空间。

2、空间释放函数 deallocate

首先先要检查释放数据块的大小，如果大于 128 字节就调用第一级配置器，小于 128 字节则根据数据块的大小来判断回收后的空间会被插入到哪个空闲链表。

3、重新填充空闲链表 refill

在用 allocate 配置空间时，如果空闲链表中没有可用数据块，就会调用 refill 来重新填充空间，新的空间取自内存池。缺省取 20 个数据块，如果内存池空间不足，那么能取多少个节点就取多少个。

从内存池取空间给空闲链表用是 chunk_alloc 的工作，首先根据 end_free-start_free 来判断内存池中的剩余空间是否足以调出 nobjs 个大小为 size 的数据块出去，如果内存连一个数据块的空间都无法供应，需要用 malloc 取堆中申请内存。

假如山穷水尽，整个系统的堆空间都不够用了，malloc 失败，那么 chunk_alloc 会从空闲链表中找是否有大的数据块，然后将该数据块的空间分给内存池（这个数据块会从链表中去除）。

3、总结：

1. 使用 allocate 向内存池请求 size 大小的内存空间，如果需要请求的内存大小大于 128bytes，直接使用 malloc。

2. 如果需要的内存大小小于 128bytes，allocate 根据 size 找到最适合的自由链表。

a. 如果链表不为空，返回第一个 node，链表头改为第二个 node。

b. 如果链表为空，使用 blockAlloc 请求分配 node。

x. 如果内存池中有大于一个 node 的空间,分配竟可能多的 node(但是最多 20 个)，将一个 node 返回，其他的 node 添加到链表中。

y. 如果内存池只有一个 node 的空间，直接返回给用户。

z. 若果如果连一个 node 都没有，再次向操作系统请求分配内存。

①分配成功，再次进行 b 过程。

②分配失败，循环各个自由链表，寻找空间。

I. 找到空间，再次进行过程 b。

II. 找不到空间，抛出异常。

3. 用户调用 deallocate 释放内存空间，如果要求释放的内存空间大于 128bytes，直接调用 free。

4. 否则按照其大小找到合适的自由链表，并将其插入。

6、C++11

1、请问 C++11 有哪些新特性？

考点：C++语法

参考回答 1：

C++11 最常用的新特性如下：

auto 关键字：编译器可以根据初始值自动推导出类型。但是不能用于函数传参以及数组类型的推导

nullptr 关键字：nullptr 是一种特殊类型的字面值，它可以被转换成任意其它的指针类型；而 NULL 一般被宏定义为 0，在遇到重载时可能会出现问题。

智能指针：C++11 新增了 std::shared_ptr、std::weak_ptr 等类型的智能指针，用于解决内存管理的问题。

初始化列表：使用初始化列表来对类进行初始化

右值引用：基于右值引用可以实现移动语义和完美转发，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

atomic 原子操作用于多线程资源互斥操作

新增 STL 容器 array 以及 tuple

参考回答 2：

1、关键字及新语法
    1.1、auto 关键字及用法
    1.2、nullptr 关键字及用法
    1.3、for 循环语法
2、STL 容器
    2.1、std::array
    2.2、std::forward_list
    2.3、std::unordered_map
    2.4、std::unordered_set
3、多线程
    3.1、std::thread
    3.2、st::atomic
    3.3、std::condition_variable
4、智能指针内存管理
    4.1、std::shared_ptr
    4.2、std::weak_ptr
5、其他

5.1、std::function、std::bind 封装可执行对象

5.2、lamda 表达式

等等，这些只是比较常用的。

2、请你详细介绍一下 C++11 中的可变参数模板、右值引用和 lambda 这几个新特性。

考点：C++11 可变参数模板、右值引用和 lambda 新特性

参考回答：

可变参数模板：

C++11 的可变参数模板，对参数进行了高度泛化，可以表示任意数目、任意类型的参数，其语法为：在 class 或 typename 后面带上省略号”。

例如：

```
Template<class ... T>
 void func(T ... args)
{
cout<<”num is”<<sizeof ...(args)<<endl;
}
func();//args 不含任何参数
func(1);//args 包含一个 int 类型的实参
func(1,2.0)//args 包含一个 int 一个 double 类型的实参
```

其中 T 叫做模板参数包，args 叫做函数参数包

省略号作用如下：

1）声明一个包含 0 到任意个模板参数的参数包

2）在模板定义得右边，可以将参数包展成一个个独立的参数

C++11 可以使用递归函数的方式展开参数包，获得可变参数的每个值。通过递归函数展开参数包，需要提供一个参数包展开的函数和一个递归终止函数。例如：

```
#include using namespace std;
// 最终递归函数
void print()
```

```
{
cout << "empty" << endl;
}
// 展开函数
template void print(T head, Args... args)
 {
cout << head << ","; print(args...);
 }
int main()
{
 print(1, 2, 3, 4); return 0;
}
```

参数包 Args ...在展开的过程中递归调用自己，没调用一次参数包中的参数就会少一个，直到所有参数都展开为止。当没有参数时就会调用非模板函数 printf 终止递归过程。

右值引用:

C++中，左值通常指可以取地址，有名字的值就是左值，而不能取地址，没有名字的就是右值。而在指 C++11 中，右值是由两个概念构成，将亡值和纯右值。纯右值是用于识别临时变量和一些不跟对象关联的值，比如 1+3 产生的临时变量值，2、true 等，而将亡值通常是指具有转移语义的对象，比如返回右值引用 T&&的函数返回值等。

C++11 中，右值引用就是对一个右值进行引用的类型。由于右值通常不具有名字，所以我们一般只能通过右值表达式获得其引用，比如：

T && a=ReturnRvale();

假设 ReturnRvalue()函数返回一个右值，那么上述语句声明了一个名为 a 的右值引用，其值等于 ReturnRvalue 函数返回的临时变量的值。

基于右值引用可以实现转移语义和完美转发新特性。

移动语义：

对于一个包含指针成员变量的类，由于编译器默认的拷贝构造函数都是浅拷贝，所有我们一般需要通过实现深拷贝的拷贝构造函数，为指针成员分配新的内存并进行内容拷贝，从而避免悬挂指针的问题。

但是如下列代码所示：

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

class HasPtrMem {
public:
    HasPtrMem(): d(new int(0)) {
        cout << "Construct: " << ++n_cstr << endl;
    }
    HasPtrMem(const HasPtrMem & h): d(new int(*h.d)) {
        cout << "Copy construct: " << ++n_cptr << endl;
    }
    ~HasPtrMem() {
        cout << "Destruct: " << ++n_dstr << endl;
    }
    int * d;
    static int n_cstr;
    static int n_dstr;
    static int n_cptr;
};

int HasPtrMem::n_cstr = 0;
int HasPtrMem::n_dstr = 0;
int HasPtrMem::n_cptr = 0;

HasPtrMem GetTemp() { return HasPtrMem(); }

int main() {
    HasPtrMem a = GetTemp();
}
```

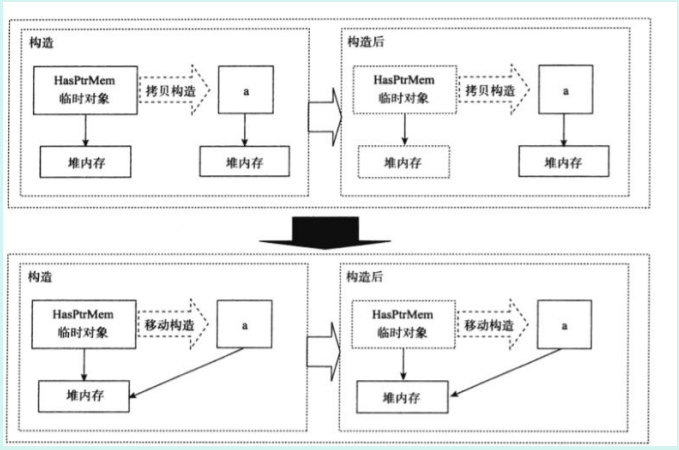
当类 HasPtrMem 包含一个成员函数 GetTemp,其返回值类型是 HasPtrMem,如果我们定义了深拷贝的拷贝构造函数，那么在调用该函数时需要调用两次拷贝构造函数。第一次是生成 GetTemp 函数返回时的临时变量，第二次是将该返回值赋值给 main 函数中的变量 a。与此对应需要调用三次析构函数来释放内存。

而在上述过程中，使用临时变量构造 a 时会调用拷贝构造函数分配对内存，而临时对象在语句结束后会释放它所使用的堆内存。这样重复申请和释放内存，在申请内存较大时会严重影响性能。因此 C++使用移动构造函数，从而保证使用临时对象构造 a 时不分配内存，从而提高性能。

如下列代码所示，移动构造函数接收一个右值引用作为参数，使用右值引用的参数初始化其指针成员变量。

```
HasPtrMem(HasPtrMem && h): d(h.d) { // 移动构造函数
    h.d = nullptr;                  // 将临时值的指针成员置空
    cout << "Move construct: " << ++n_mvtr << endl;
}
```

其原理就是使用在构造对象 a 时，使用 h.d 来初始化 a，然后将临时对象 h 的成员变量 d 指向 nullptr，从而保证临时变量析构时不会释放对内存。



完美转发：

完美转发是指在函数模板中，完全依照模板的参数的类型，将参数传递给函数模板中调用的另一个函数，即传入转发函数的是左值对象，目标函数就能获得左值对象，转发函数是右值对象，目标函数就能获得右值对象，而不产生额外的开销。

因此转发函数和目标函数参数一般采用引用类型，从而避免拷贝的开销。其次，由于目标函数可能需要能够既接受左值引用，又接受右值引用，所以考虑转发也需要兼容这两种类型。

C++11 采用引用折叠的规则，结合新的模板推导规则实现完美转发。其引用折叠规则如下：

TR 的类型定义	声明 v 的类型	v 的实际类型
T&	TR	A&
T&	TR&	A&
T&	TR&&	A&
T&&	TR	A&&
T&&	TR&	A&
T&&	TR&&	A&&

因此，我们将转发函数和目标函数的参数都设置为右值引用类型，

```
void IamForwording(T && t) {
   IrunCodeActually(static_cast<T &&>(t));
}
```

当传入一个 X 类型的左值引用时，转发函数将被实例为：

```
void IamForwording(X& && t) {
   IrunCodeActually(static_cast<X& &&>(t));
}
```

经过引用折叠，变为：

```
void IamForwording(X& t) {
   IrunCodeActually(static_cast<X&>(t));
}
```

当传入一个 X 类型的右值引用时，转发函数将被实例为：

```
void IamForwording(X&& && t) {
   IrunCodeActually(static_cast<X&& &&>(t));
}
```

经过引用折叠，变为：

```
void IamForwording(X&& t) {
   IrunCodeActually(static_cast<X&&>(t));
}
```

除此之外，还可以使用 forward()函数来完成左值引用到右值引用的转换：

```
template <typename T>
void IamForwording(T && t) {
   IrunCodeActually(forward(t));
```

Lambda 表达式:

Lambda 表达式定义一个匿名函数，并且可以捕获一定范围内的变量，其定义如下：

[capture](params)mutable->return-type{statement}

其中，

[capture]：捕获列表，捕获上下文变量以供 lambda 使用。同时[]是 lambda 寅初复，编译器根据该符号来判断接下来代码是否是 lambda 函数。

(Params)：参数列表，与普通函数的参数列表一致，如果不需要传递参数，则可以连通括号一起省略。

mutable 是修饰符，默认情况下 lambda 函数总是一个 const 函数，Mutable 可以取消其常量性。在使用该修饰符时，参数列表不可省略。

->return-type:返回类型是返回值类型

{statement}:函数体，内容与普通函数一样，除了可以使用参数之外，还可以使用所捕获的变量。

Lambda 表达式与普通函数最大的区别就是其可以通过捕获列表访问一些上下文中的数据。其形式如下:

❑[var] 表示值传递方式捕捉变量 var。
❑[=] 表示值传递方式捕捉所有父作用域的变量（包括 this）。
❑[&var] 表示引用传递捕捉变量 var。
❑[&] 表示引用传递捕捉所有父作用域的变量（包括 this）。
❑[this] 表示值传递方式捕捉当前的 this 指针。

Lambda 的类型被定义为“闭包”的类，其通常用于 STL 库中，在某些场景下可用于简化仿函数的使用，同时 Lambda 作为局部函数，也会提高复杂代码的开发加速，轻松在函数内重用代码，无须费心设计接口。

二、操作系统

1、请你说一下进程与线程的概念，以及为什么要有进程线程，其中有什么区别，他们各自又是怎么同步的

考察点：操作系统

参考回答：

基本概念：

进程是对运行时程序的封装，是系统进行资源调度和分配的的基本单位，实现了操作系统的并发；

线程是进程的子任务，是 CPU 调度和分派的基本单位，用于保证程序的实时性，实现进程内部的并发;线程是操作系统可识别的最小执行和调度单位。每个线程都独自占用一个虚拟处理器：独自的寄存器组，指令计数器和处理器状态。每个线程完成不同的任务，但是共享同一地址空间（也就是同样的动态内存，映射文件，目标代码等等），打开的文件队列和其他内核资源。

区别：

1.一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。线程依赖于进程而存在。

2.进程在执行过程中拥有独立的内存单元，而多个线程共享进程的内存。（资源分配给进程，同一进程的所有线程共享该进程的所有资源。同一进程中的多个线程共享代码段（代码和常量），数据段（全局变量和静态变量），扩展段（堆存储）。但是每个线程拥有自己的栈段，栈段又叫运行时段，用来存放所有局部变量和临时变量。）

3.进程是资源分配的最小单位，线程是 CPU 调度的最小单位；

4.系统开销： 由于在创建或撤消进程时，系统都要为之分配或回收资源，如内存空间、I／o 设备等。因此，操作系统所付出的开销将显著地大于在创建或撤消线程时的开销。类似地，在进行进程切换时,涉及到整个当前进程 CPU 环境的保存以及新被调度运行的进程的 CPU 环境的设置。而线程切换只须保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作。可见，进程切换的开销也远大于线程切换的开销。

5.通信：由于同一进程中的多个线程具有相同的地址空间，致使它们之间的同步和通信的实现，也变得比较容易。进程间通信 IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。在有的系统中，线程的切换、同步和通信都无须操作系统内核的干预

6.进程编程调试简单可靠性高，但是创建销毁开销大；线程正相反，开销小，切换速度快，但是编程调试相对复杂。

7.进程间不会相互影响 ；线程一个线程挂掉将导致整个进程挂掉

8.进程适应于多核、多机分布；线程适用于多核

进程间通信的方式：

进程间通信主要包括管道、系统 IPC（包括消息队列、信号量、信号、共享内存等）、以及套接字 socket。

1.管道：

管道主要包括无名管道和命名管道:管道可用于具有亲缘关系的父子进程间的通信，有名管道除了具有管道所具有的功能外，它还允许无亲缘关系进程间的通信

1.1 普通管道 PIPE：

1)它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端

2)它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）

3)它可以看成是一种特殊的文件，对于它的读写也可以使用普通的 read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

1.2 命名管道 FIFO：

1)FIFO 可以在无关的进程之间交换数据

2)FIFO 有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

2. 系统 IPC：

2.1 消息队列

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列 ID）来标记。(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特点)具有写权限得进程可以按照一定得规则向消息队列中添加新信息；对消息队列有读权限得进程则可以从消息队列中读取信息；

特点：

1)消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。

2)消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。

3)消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。

2.2 信号量 semaphore

信号量（semaphore）与已经介绍过的 IPC 结构不同，它是一个计数器，可以用来控制多个进程对共享资源的访问。信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。

特点：

1)信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

2)信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。

3)每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

4)支持信号量组。

2.3 信号 signal

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。

2.4 共享内存（Shared Memory）

它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等

特点：

1)共享内存是最快的一种 IPC，因为进程是直接对内存进行存取

2)因为多个进程可以同时操作，所以需要进行同步

3)信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问

3.套接字 SOCKET：

socket 也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同主机之间的进程通信。

线程间通信的方式:

临界区：通过多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问；

互斥量 Synchronized/Lock：采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问

信号量 Semphare：为控制具有有限数量的用户资源而设计的，它允许多个线程在同一时刻去访问同一个资源，但一般需要限制同一时刻访问此资源的最大线程数目。

事件(信号)，Wait/Notify：通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作

2、请你说一说 Linux 虚拟地址空间

考察点：Linux 基础知识

参考回答：

为了防止不同进程同一时刻在物理内存中运行而对物理内存的争夺和践踏，采用了虚拟内存。

虚拟内存技术使得不同进程在运行过程中，它所看到的是自己独自占有了当前系统的 4G 内存。所有进程共享同一物理内存，每个进程只把自己目前需要的虚拟内存空间映射并存储到物理内存上。 事实上，在每个进程创建加载时，内核只是为进程“创建”了虚拟内存的布局，具体就是初始化进程控制表中内存相关的链表，实际上并不立即就把虚拟内存对应位置的程序数据和代码（比如.text .data 段）拷贝到物理内存中，只是建立好虚拟内存和磁盘文件之间的映射就好（叫做存储器映射），等到运行到对应的程序时，才会通过缺页异常，来拷贝数据。还有进程运行过程中，要动态分配内存，比如 malloc 时，也只是分配了虚拟内存，即为这块虚拟内存对应的页表项做相应设置，当进程真正访问到此数据时，才引发缺页异常。

请求分页系统、请求分段系统和请求段页式系统都是针对虚拟内存的，通过请求实现内存与外存的信息置换。

虚拟内存的好处：

1.扩大地址空间；

2.内存保护：每个进程运行在各自的虚拟内存地址空间，互相不能干扰对方。虚存还对特定的内存地址提供写保护，可以防止代码或数据被恶意篡改。

3.公平内存分配。采用了虚存之后，每个进程都相当于有同样大小的虚存空间。

4.当进程通信时，可采用虚存共享的方式实现。

5.当不同的进程使用同样的代码时，比如库文件中的代码，物理内存中可以只存储一份这样的代码，不同的进程只需要把自己的虚拟内存映射过去就可以了，节省内存

6.虚拟内存很适合在多道程序设计系统中使用，许多程序的片段同时保存在内存中。当一个程序等待它的一部分读入内存时，可以把 CPU 交给另一个进程使用。在内存中可以保留多个进程，系统并发度提高

7.在程序需要分配连续的内存空间的时候，只需要在虚拟内存空间分配连续空间，而不需要实际物理内存的连续空间，可以利用碎片

虚拟内存的代价：

1.虚存的管理需要建立很多数据结构，这些数据结构要占用额外的内存

2.虚拟地址到物理地址的转换，增加了指令的执行时间。

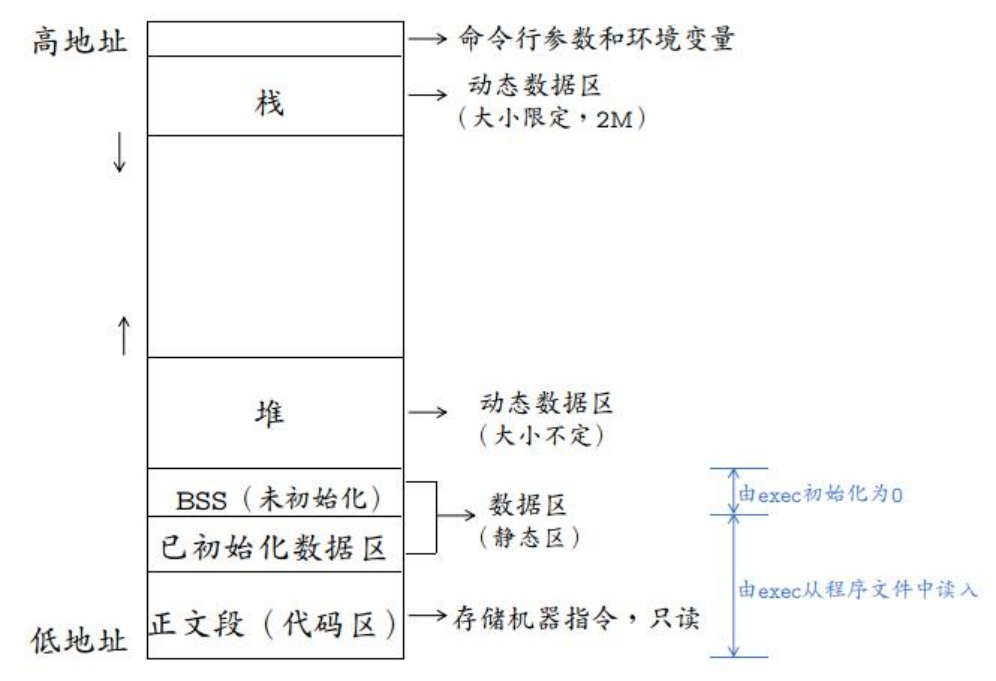
3.页面的换入换出需要磁盘 I/O，这是很耗时的

4.如果一页中只有一部分数据，会浪费内存。

3、请你说一说操作系统中的程序的内存结构

考察点：操作系统

参考回答：



一个程序本质上都是由 BSS 段、data 段、text 段三个组成的。可以看到一个可执行程序在存储（没有调入内存）时分为代码段、数据区和未初始化数据区三部分。

BSS 段（未初始化数据区）：通常用来存放程序中未初始化的全局变量和静态变量的一块内存区域。BSS 段属于静态分配，程序结束后静态变量资源由系统自动释放。

数据段：存放程序中已初始化的全局变量的一块内存区域。数据段也属于静态内存分配

代码段：存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域属于只读。在代码段中，也有可能包含一些只读的常数变量

text 段和 data 段在编译时已经分配了空间，而 BSS 段并不占用可执行文件的大小，它是由链接器来获取内存的。

bss 段（未进行初始化的数据）的内容并不存放在磁盘上的程序文件中。其原因是内核在程序开始运行前将它们设置为 0。需要存放在程序文件中的只有正文段和初始化数据段。

data 段（已经初始化的数据）则为数据分配空间，数据保存到目标文件中。

数据段包含经过初始化的全局变量以及它们的值。BSS 段的大小从可执行文件中得到，然后链接器得到这个大小的内存块，紧跟在数据段的后面。当这个内存进入程序的地址空间后全部清零。包含数据段和 BSS 段的整个区段此时通常称为数据区。

可执行程序在运行时又多出两个区域：栈区和堆区。

栈区：**由编译器自动释放，存放函数的参数值、局部变量等。每当一个函数被调用时，该函数的返回类型和一些调用的信息被存放到栈中。然后这个被调用的 函数再为他的自动变量和临时变量在栈上分配空间。每调用一个函数一个新的栈就会被使用。栈区是从高地址位向低地址位增长的，是一块连续的内存区域，最大容 量是由系统预先定义好的，申请的栈空间超过这个界限时会提示溢出，用户能从栈中获取的空间较小。**

堆区：用于动态分配内存，位于 BSS 和栈中间的地址区域。由程序员申请分配和释放。堆是从低地址位向高地址位增长，采用链式存储结构。频繁的 malloc/free 造成内存空间的不连续，产生碎片。当申请堆空间时库函数是按照一定的算法搜索可用的足够大的空间。因此堆的效率比栈要低的多。

4、请你说一说操作系统中的缺页中断

考察点：操作系统

参考回答：

malloc()和 mmap()等内存分配函数，在分配时只是建立了进程虚拟地址空间，并没有分配虚拟内存对应的物理内存。当进程访问这些没有建立映射关系的虚拟内存时，处理器自动触发一个缺页异常。

缺页中断：在请求分页系统中，可以通过查询页表中的状态位来确定所要访问的页面是否存在于内存中。每当所要访问的页面不在内存是，会产生一次缺页中断，此时操作系统会根据页表中的外存地址在外存中找到所缺的一页，将其调入内存。

缺页本身是一种中断，与一般的中断一样，需要经过 4 个处理步骤：

1、保护 CPU 现场

2、分析中断原因

3、转入缺页中断处理程序进行处理

4、恢复 CPU 现场，继续执行

但是缺页中断是由于所要访问的页面不存在于内存时，由硬件所产生的一种特殊的中断，因此，与一般的中断存在区别：

1、在指令执行期间产生和处理缺页中断信号

2、一条指令在执行期间，可能产生多次缺页中断

3、缺页中断返回是，执行产生中断的一条指令，而一般的中断返回是，执行下一条指令。

5、请你回答一下 fork 和 vfork 的区别

考察点：操作系统

参考回答：

fork 的基础知识：

fork:创建一个和当前进程映像一样的进程可以通过 fork( )系统调用：

```
#include <sys/types.h>

#include <unistd.h>

pid_t fork(void);
```

成功调用 fork( )会创建一个新的进程，它几乎与调用 fork( )的进程一模一样，这两个进程都会继续运行。在子进程中，成功的 fork( )调用会返回 0。在父进程中 fork( )返回子进程的 pid。如果出现错误，fork( )返回一个负值。

最常见的 fork( )用法是创建一个新的进程，然后使用 exec( )载入二进制映像，替换当前进程的映像。这种情况下，派生（fork）了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。

在早期的 Unix 系统中，创建进程比较原始。当调用 fork 时，内核会把所有的内部数据结构复制一份，复制进程的页表项，然后把父进程的地址空间中的内容逐页的复制到子进程的地址空间中。但从内核角度来说，逐页的复制方式是十分耗时的。现代的 Unix 系统采取了更多的优化，例如 Linux，采用了写时复制的方法，而不是对父进程空间进程整体复制。

vfork 的基础知识：

在实现写时复制之前，Unix 的设计者们就一直很关注在 fork 后立刻执行 exec 所造成的地址空间的浪费。BSD 的开发者们在 3.0 的 BSD 系统中引入了 vfork( )系统调用。

```
#include <sys/types.h>

#include <unistd.h>

pid_t vfork(void);
```

除了子进程必须要立刻执行一次对 exec 的系统调用，或者调用_exit( )退出，对 vfork( )的成功调用所产生的结果和 fork( )是一样的。vfork( )会挂起父进程直到子进程终止或者运行了一个新的可执行文件的映像。通过这样的方式，vfork( )避免了地址空间的按页复制。在这个过程中，父进程和子进程共享相同的地址空间和页表项。实际上 vfork( )只完成了一件事：复制内部的内核数据结构。因此，子进程也就不能修改地址空间中的任何内存。

vfork( )是一个历史遗留产物，Linux 本不应该实现它。需要注意的是，即使增加了写时复制，vfork( )也要比 fork( )快，因为它没有进行页表项的复制。然而，写时复制的出现减少了对于替换 fork( )争论。实际上，直到 2.2.0 内核，vfork( )只是一个封装过的 fork( )。因为对 vfork( )的需求要小于 fork( )，所以 vfork( )的这种实现方式是可行的。

补充知识点：写时复制

Linux 采用了写时复制的方法，以减少 fork 时对父进程空间进程整体复制带来的开销。

写时复制是一种采取了惰性优化方法来避免复制时的系统开销。它的前提很简单：如果有多个进程要读取它们自己的那部门资源的副本，那么复制是不必要的。每个进程只要保存一个指向这个资源的指针就可以了。只要没有进程要去修改自己的“副本”，就存在着这样的幻觉：每个进程好像独占那个资源。从而就避免了复制带来的负担。如果一个进程要修改自己的那份资源“副本”，那么就会复制那份资源，并把复制的那份提供给进程。不过其中的复制对进程来说是透明的。这个进程就可以修改复制后的资源了，同时其他的进程仍然共享那份没有修改过的资源。所以这就是名称的由来：在写入时进行复制。

写时复制的主要好处在于：如果进程从来就不需要修改资源，则不需要进行复制。惰性算法的好处就在于它们尽量推迟代价高昂的操作，直到必要的时刻才会去执行。

在使用虚拟内存的情况下，写时复制（Copy-On-Write）是以页为基础进行的。所以，只要进程不修改它全部的地址空间，那么就不必复制整个地址空间。在 fork( )调用结束后，父进程和子进程都相信它们有一个自己的地址空间，但实际上它们共享父进程的原始页，接下来这些页又可以被其他的父进程或子进程共享。

写时复制在内核中的实现非常简单。与内核页相关的数据结构可以被标记为只读和写时复制。如果有进程试图修改一个页，就会产生一个缺页中断。内核处理缺页中断的方式就是对该页进行一次透明复制。这时会清除页面的 COW 属性，表示着它不再被共享。

现代的计算机系统结构中都在内存管理单元（MMU）提供了硬件级别的写时复制支持，所以实现是很容易的。

在调用 fork( )时，写时复制是有很大优势的。因为大量的 fork 之后都会跟着执行 exec，那么复制整个父进程地址空间中的内容到子进程的地址空间完全是在浪费时间：如果子进程立刻

执行一个新的二进制可执行文件的映像，它先前的地址空间就会被交换出去。写时复制可以对这种情况进行优化。

fork 和 vfork 的区别：

1. fork()的子进程拷贝父进程的数据段和代码段；vfork()的子进程与父进程共享数据段

2. fork()的父子进程的执行次序不确定；vfork()保证子进程先运行，在调用 exec 或 exit 之前与父进程数据是共享的，在它调用 exec 或 exit 之后父进程才可能被调度运行。

3. vfork()保证子进程先运行，在它调用 exec 或 exit 之后父进程才可能被调度运行。如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。

4.当需要改变共享数据段中变量的值，则拷贝父进程。

6、请问如何修改文件最大句柄数？

考察点：linux 调优、高并发问题

参考回答：

linux 默认最大文件句柄数是 1024 个，在 linux 服务器文件并发量比较大的情况下，系统会报"too many open files"的错误。故在 linux 服务器高并发调优时，往往需要预先调优 Linux 参数，修改 Linux 最大文件句柄数。

有两种方法：

1. ulimit -n <可以同时打开的文件数>，将当前进程的最大句柄数修改为指定的参数（注：该方法只针对当前进程有效，重新打开一个 shell 或者重新开启一个进程，参数还是之前的值）

首先用 ulimit -a 查询 Linux 相关的参数，如下所示：

```
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 94739
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
```

```
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 94739
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

其中，open files 就是最大文件句柄数，默认是 1024 个。

修改 Linux 最大文件句柄数： ulimit -n 2048， 将最大句柄数修改为 2048 个。

2. 对所有进程都有效的方法，修改 Linux 系统参数

vi /etc/security/limits.conf 添加

```
*    soft    nofile    65536
*    hard    nofile    65536
```

将最大句柄数改为 65536

修改以后保存，注销当前用户，重新登录，修改后的参数就生效了

7、请你说一说并发(concurrency)和并行(parallelism)

考察点：操作系统

参考回答：

并发（concurrency）：指宏观上看起来两个程序在同时运行，比如说在单核 cpu 上的多任务。但是从微观上看两个程序的指令是交织着运行的，你的指令之间穿插着我的指令，我的指令之间穿插着你的，在单个周期内只运行了一个指令。这种并发并不能提高计算机的性能，只能提高效率。

并行（parallelism）：指严格物理意义上的同时运行，比如多核 cpu，两个程序分别运行在两个核上，两者之间互不影响，单个周期内每个程序都运行了自己的指令，也就是运行了两条指令。这样说来并行的确提高了计算机的效率。所以现在的 cpu 都是往多核方面发展。

8、请问 MySQL 的端口号是多少，如何修改这个端口号

考察点：Linux

参考回答：

查看端口号：

使用命令 show global variables like 'port';查看端口号 ，mysql 的默认端口是 3306。（补充：sqlserver 默认端口号为：1433；oracle 默认端口号为：1521；DB2 默认端口号为：5000；PostgreSQL 默认端口号为：5432）

修改端口号：

修改端口号：编辑/etc/my.cnf 文件，早期版本有可能是 my.conf 文件名，增加端口参数，并且设定端口，注意该端口未被使用，保存退出。

9、请你说一说操作系统中的页表寻址

考察点：操作系统

参考回答：

页式内存管理，内存分成固定长度的一个个页片。操作系统为每一个进程维护了一个从虚拟地址到物理地址的映射关系的数据结构，叫页表，页表的内容就是该进程的虚拟地址到物理地址的一个映射。页表中的每一项都记录了这个页的基地址。通过页表，由逻辑地址的高位部分先找到逻辑地址对应的页基地址，再由页基地址偏移一定长度就得到最后的物理地址，偏移的长度由逻辑地址的低位部分决定。一般情况下，这个过程都可以由硬件完成，所以效率还是比较高的。页式内存管理的优点就是比较灵活，内存管理以较小的页为单位，方便内存换入换出和扩充地址空间。

Linux 最初的两级页表机制：

两级分页机制将 32 位的虚拟空间分成三段，低十二位表示页内偏移，高 20 分成两段分别表示两级页表的偏移。

* PGD(Page Global Directory)：最高 10 位，全局页目录表索引

* PTE(Page Table Entry)：中间 10 位，页表入口索引

当在进行地址转换时，结合在 CR3 寄存器中存放的页目录(page directory, PGD)的这一页的物理地址，再加上从虚拟地址中抽出高 10 位叫做页目录表项(内核也称这为 pgd)的部分作为偏移，即定位到可以描述该地址的 pgd;从该 pgd 中可以获取可以描述该地址的页表的物理地址，再加上从虚拟地址中抽取中间 10 位作为偏移，即定位到可以描述该地址的 pte；在这个 pte 中即可获取该地址对应的页的物理地址，加上从虚拟地址中抽取的最后 12 位，即形成该页的页内

偏移，即可最终完成从虚拟地址到物理地址的转换。从上述过程中，可以看出，对虚拟地址的分级解析过程，实际上就是不断深入页表层次，逐渐定位到最终地址的过程，所以这一过程被叫做 page talbe walk。

Linux 的三级页表机制：

当 X86 引入物理地址扩展(Pisycal Addrress Extension, PAE)后，可以支持大于 4G 的物理内存(36 位),但虚拟地址依然是 32 位,原先的页表项不适用,它实际多 4 bytes 被扩充到 8 bytes，这意味着，每一页现在能存放的 pte 数目从 1024 变成 512 了(4k/8)。相应地，页表层级发生了变化，Linus 新增加了一个层级，叫做页中间目录(page middle directory, PMD)，变成：

字段	描述	位数
cr3	指向一个 PDPT	crs 寄存器存储
PGD	指向 PDPT 中 4 个项中的一个	位 31~30
PMD	指向页目录中 512 项中的一个	位 29~21
PTE	指向页表中 512 项中的一个	位 20~12
page offset	4KB 页中的偏移	位 11~0

现在就同时存在 2 级页表和 3 级页表，在代码管理上肯定不方便。巧妙的是，Linux 采取了一种抽象方法：所有架构全部使用 3 级页表：即 PGD -> PMD -> PTE。那只使用 2 级页表(如非 PAE 的 X86)怎么办？

办法是针对使用 2 级页表的架构，把 PMD 抽象掉，即虚设一个 PMD 表项。这样在 page table walk 过程中，PGD 本直接指向 PTE 的，现在不了，指向一个虚拟的 PMD，然后再由 PMD 指向 PTE。这种抽象保持了代码结构的统一。

Linux 的四级页表机制：

硬件在发展，3 级页表很快又捉襟见肘了，原因是 64 位 CPU 出现了，比如 X86_64， 它的硬件是实实在在支持 4 级页表的。它支持 48 位的虚拟地址空间 1。如下：

字段	描述	位数
PML4	指向一个 PDPT	位 47~39
PGD	指向 PDPT 中 4 个项中的一个	位 38~30
PMD	指向页目录中 512 项中的一个	位 29~21

PTE　　指向页表中 512 项中的一个　位 20~12

page offset　4KB 页中的偏移　　位 11~0

Linux 内核针为使用原来的 3 级列表(PGD->PMD->PTE)，做了折衷。即采用一个唯一的，共享的顶级层次，叫 PML4。这个 PML4 没有编码在地址中，这样就能套用原来的 3 级列表方案了。不过代价就是，由于只有唯一的 PML4，寻址空间被局限在(239=)512G，而本来 PML4 段有 9 位，可以支持 512 个 PML4 表项的。现在为了使用 3 级列表方案，只能限制使用一个， 512G 的空间很快就又不够用了，解决方案呼之欲出。

在 2004 年 10 月,当时的 X86_64 架构代码的维护者 Andi Kleen 提交了一个叫做 4level page tables for Linux 的 PATCH 系列，为 Linux 内核带来了 4 级页表的支持。在他的解决方案中，不出意料地，按照 X86_64 规范，新增了一个 PML4 的层级，在这种解决方案中，X86_64 拥一个有 512 条目的 PML4，512 条目的 PGD，512 条目的 PMD，512 条目的 PTE。对于仍使用 3 级目录的架构来说，它们依然拥有一个虚拟的 PML4,相关的代码会在编译时被优化掉。 这样，就把 Linux 内核的 3 级列表扩充为 4 级列表。这系列 PATCH 工作得不错，不久被纳入 Andrew Morton 的-mm 树接受测试。不出意外的话，它将在 v2.6.11 版本中释出。但是，另一个知名开发者 Nick Piggin 提出了一些看法，他认为 Andi 的 Patch 很不错，不过他认为最好还是把 PGD 作为第一级目录，把新增加的层次放在中间，并给出了他自己的 Patch:alternate 4-level page tables patches。Andi 更想保持自己的 PATCH，他认为 Nick 不过是玩了改名的游戏，而且他的 PATCH 经过测试很稳定，快被合并到主线了，不宜再折腾。不过 Linus 却表达了对 Nick Piggin 的支持，理由是 Nick 的做法 conceptually least intrusive。毕竟作为 Linux 的扛把子，稳定对于 Linus 来说意义重大。最终，不意外地，最后 Nick Piggin 的 PATCH 在 v2.6.11 版本中被合并入主线。在这种方案中，4 级页表分别是：PGD -> PUD -> PMD -> PTE。

10、请你说一说有了进程，为什么还要有线程？

考点：线程和进程

参考回答：

线程产生的原因：

进程可以使多个程序能并发执行，以提高资源的利用率和系统的吞吐量；但是其具有一些缺点：

进程在同一时间只能干一件事

进程在执行的过程中如果阻塞，整个进程就会挂起，即使进程中有些工作不依赖于等待的资源，仍然不会执行。

因此，操作系统引入了比进程粒度更小的线程，作为并发执行的基本单位，从而减少程序在并发执行时所付出的时空开销，提高并发性。和进程相比，线程的优势如下：

从资源上来讲，线程是一种非常"节俭"的多任务操作方式。在 linux 系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种"昂贵"的多任务工作方式。

从切换效率上来讲，运行于一个进程中的多个线程，它们之间使用相同的地址空间，而且线程间彼此切换所需时间也远远小于进程间切换所需要的时间。据统计，一个进程的开销大约是一个线程开销的 30 倍左右。（

从通信机制上来讲，线程间方便的通信机制。对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过进程间通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进城下的线程之间贡献数据空间，所以一个线程的数据可以直接为其他线程所用，这不仅快捷，而且方便。

除以上优点外，多线程程序作为一种多任务、并发的工作方式，还有如下优点：

1、使多 CPU 系统更加有效。操作系统会保证当线程数不大于 CPU 数目时，不同的线程运行于不同的 CPU 上。

2、改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序才会利于理解和修改。

11、请问单核机器上写多线程程序，是否需要考虑加锁，为什么？

考点：多核 多线程 锁

参考回答：

在单核机器上写多线程程序，仍然需要线程锁。因为线程锁通常用来实现线程的同步和通信。在单核机器上的多线程程序，仍然存在线程同步的问题。因为在抢占式操作系统中，通常为每个线程分配一个时间片，当某个线程时间片耗尽时，操作系统会将其挂起，然后运行另一个线程。如果这两个线程共享某些数据，不使用线程锁的前提下，可能会导致共享数据修改引起冲突。

12、请问线程需要保存哪些上下文，SP、PC、EAX 这些寄存器是干嘛用的

考点：线程切换

参考回答：

线程在切换的过程中需要保存当前线程 Id、线程状态、堆栈、寄存器状态等信息。其中寄存器主要包括 SP PC EAX 等寄存器，其主要功能如下：

SP:堆栈指针，指向当前栈的栈顶地址

PC:程序计数器，存储下一条将要执行的指令

EAX:累加寄存器，用于加法乘法的缺省寄存器

13、请你说一说线程间的同步方式，最好说出具体的系统调用

参考回答：

信号量

信号量是一种特殊的变量，可用于线程同步。它只取自然数值，并且只支持两种操作：

P(SV):如果信号量 SV 大于 0，将它减一；如果 SV 值为 0，则挂起该线程。

V(SV)：如果有其他进程因为等待 SV 而挂起，则唤醒，然后将 SV+1；否则直接将 SV+1。

其系统调用为：

sem_wait（sem_t *sem）：以原子操作的方式将信号量减 1，如果信号量值为 0，则 sem_wait 将被阻塞，直到这个信号量具有非 0 值。

sem_post（sem_t *sem)：以原子操作将信号量值+1。当信号量大于 0 时，其他正在调用 sem_wait 等待信号量的线程将被唤醒。

互斥量

互斥量又称互斥锁，主要用于线程互斥，不能保证按序访问，可以和条件锁一起实现同步。当进入临界区 时，需要获得互斥锁并且加锁；当离开临界区时，需要对互斥锁解锁，以唤醒其他等待该互斥锁的线程。其主要的系统调用如下：

pthread_mutex_init:初始化互斥锁

pthread_mutex_destroy：销毁互斥锁

pthread_mutex_lock：以原子操作的方式给一个互斥锁加锁，如果目标互斥锁已经被上锁，pthread_mutex_lock 调用将阻塞，直到该互斥锁的占有者将其解锁。

pthread_mutex_unlock:以一个原子操作的方式给一个互斥锁解锁。

条件变量

条件变量，又称条件锁，用于在线程之间同步共享数据的值。条件变量提供一种线程间通信机制：当某个共享数据达到某个值时，唤醒等待这个共享数据的一个/多个线程。即，当某个共享变量等于某个值时，调用 signal/broadcast。此时操作共享变量时需要加锁。其主要的系统调用如下：

pthread_cond_init:初始化条件变量

pthread_cond_destroy：销毁条件变量

pthread_cond_signal：唤醒一个等待目标条件变量的线程。哪个线程被唤醒取决于调度策略和优先级。

pthread_cond_wait：等待目标条件变量。需要一个加锁的互斥锁确保操作的原子性。该函数中在进入 wait 状态前首先进行解锁，然后接收到信号后会再加锁，保证该线程对共享资源正确访问。

14、请你说一下多线程和多进程的不同

参考回答：

进程是资源分配的最小单位，而线程时 CPU 调度的最小单位。多线程之间共享同一个进程的地址空间，线程间通信简单，同步复杂，线程创建、销毁和切换简单，速度快，占用内存少，适用于多核分布式系统，但是线程间会相互影响，一个线程意外终止会导致同一个进程的其他线程也终止，程序可靠性弱。而多进程间拥有各自独立的运行地址空间，进程间不会相互影响，程序可靠性强，但是进程创建、销毁和切换复杂，速度慢，占用内存多，进程间通信复杂，但是同步简单，适用于多核、多机分布。

15、请你说一说进程和线程的区别

考点：进程 线程

参考回答:

1）进程是 cpu 资源分配的最小单位，线程是 cpu 调度的最小单位。

2）进程有独立的系统资源，而同一进程内的线程共享进程的大部分系统资源,包括堆、代码段、数据段，每个线程只拥有一些在运行中必不可少的私有属性，比如 tcb,线程 Id,栈、寄存器。

3）一个进程崩溃，不会对其他进程产生影响；而一个线程崩溃，会让同一进程内的其他线程也死掉。

4）进程在创建、切换和销毁时开销比较大，而线程比较小。进程创建的时候需要分配系统资源，而销毁的的时候需要释放系统资源。进程切换需要分两步：切换页目录、刷新 TLB 以使用新的地址空间；切换内核栈和硬件上下文（寄存器）；而同一进程的线程间逻辑地址空间是一样的，不需要切换页目录、刷新 TLB。

5）进程间通信比较复杂，而同一进程的线程由于共享代码段和数据段，所以通信比较容易。

16、游戏服务器应该为每个用户开辟一个线程还是一个进程，为什么？

参考回答：

游戏服务器应该为每个用户开辟一个进程。因为同一进程间的线程会相互影响，一个线程死掉会影响其他线程，从而导致进程崩溃。因此为了保证不同用户之间不会相互影响，应该为每个用户开辟一个进程

17、请你说一说 OS 缺页置换算法

参考回答：

当访问一个内存中不存在的页，并且内存已满，则需要从内存中调出一个页或将数据送至磁盘对换区，替换一个页，这种现象叫做缺页置换。当前操作系统最常采用的缺页置换算法如下：

先进先出(FIFO)算法：置换最先调入内存的页面，即置换在内存中驻留时间最久的页面。按照进入内存的先后次序排列成队列，从队尾进入，从队首删除。

最近最少使用（LRU）算法：置换最近一段时间以来最长时间未访问过的页面。根据程序局部性原理，刚被访问的页面，可能马上又要被访问；而较长时间内没有被访问的页面，可能最近不会被访问。

当前最常采用的就是 LRU 算法。

18、请你说一说进程和线程区别

考点：进程 线程

参考回答：

1）进程是 cpu 资源分配的最小单位，线程是 cpu 调度的最小单位。

2）进程有独立的系统资源，而同一进程内的线程共享进程的大部分系统资源,包括堆、代码段、数据段，每个线程只拥有一些在运行中必不可少的私有属性，比如 tcb,线程 Id,栈、寄存器。

3）一个进程崩溃，不会对其他进程产生影响；而一个线程崩溃，会让同一进程内的其他线程也死掉。

4）进程在创建、切换和销毁时开销比较大，而线程比较小。进程创建的时候需要分配系统资源，而销毁的的时候需要释放系统资源。进程切换需要分两步：切换页目录、刷新 TLB 以使用新的地址空间；切换内核栈和硬件上下文（寄存器）；而同一进程的线程间逻辑地址空间是一样的，不需要切换页目录、刷新 TLB。

5）进程间通信比较复杂，而同一进程的线程由于共享代码段和数据段，所以通信比较容易。

19、请你说一下多进程和多线程的使用场景

考点：进程与线程的区别

参考回答：

多进程模型的优势是 CPU

多线程模型主要优势为线程间切换代价较小，因此适用于 I/O 密集型的工作场景，因此 I/O 密集型的工作场景经常会由于 I/O 阻塞导致频繁的切换线程。同时，多线程模型也适用于单机多核分布式场景。

多进程模型，适用于 CPU 密集型。同时，多进程模型也适用于多机分布式场景中，易于多机扩展。

20、请你说一说死锁发生的条件以及如何解决死锁

考点：死锁

参考回答：

死锁是指两个或两个以上进程在执行过程中，因争夺资源而造成的下相互等待的现象。死锁发生的四个必要条件如下：

互斥条件：进程对所分配到的资源不允许其他进程访问，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源；

请求和保持条件：进程获得一定的资源后，又对其他资源发出请求，但是该资源可能被其他进程占有，此时请求阻塞，但该进程不会释放自己已经占有的资源

不可剥夺条件：进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用后自己释放

环路等待条件：进程发生死锁后，必然存在一个进程-资源之间的环形链

解决死锁的方法即破坏上述四个条件之一，主要方法如下：

资源一次性分配，从而剥夺请求和保持条件

可剥夺资源：即当进程新的资源未得到满足时，释放已占有的资源，从而破坏不可剥夺的条件

资源有序分配法：系统给每类资源赋予一个序号，每个进程按编号递增的请求资源，释放则相反，从而破坏环路等待的条件

21、请问虚拟内存和物理内存怎么对应

参考回答：

1、概念：

物理地址(physical address)

用于内存芯片级的单元寻址，与处理器和 CPU 连接的地址总线相对应。

虽然可以直接把物理地址理解成插在机器上那根内存本身，把内存看成一个从 0 字节一直到最大空量逐字节的编号的大数组，然后把这个数组叫做物理地址，但是事实上，这只是一个硬件提供给软件的抽像，内存的寻址方式并不是这样。所以，说它是“与地址总线相对应”，是更贴切一些，不过抛开对物理内存寻址方式的考虑，直接把物理地址与物理的内存一一对应，也是可以接受的。也许错误的理解更利于形而上的抽像。

虚拟地址(virtual memory)

这是对整个内存（不要与机器上插那条对上号）的抽像描述。它是相对于物理内存来讲的，可以直接理解成“不直实的”，“假的”内存，例如，一个 0x08000000 内存地址，它并不对就物理地址上那个大数组中 0x08000000 - 1 那个地址元素；

之所以是这样，是因为现代操作系统都提供了一种内存管理的抽像，即虚拟内存（virtual memory）。进程使用虚拟内存中的地址，由操作系统协助相关硬件，把它“转换”成真正的物理地址。这个“转换”，是所有问题讨论的关键。

有了这样的抽像，一个程序，就可以使用比真实物理地址大得多的地址空间。甚至多个进程可以使用相同的地址。不奇怪，因为转换后的物理地址并非相同的。

——可以把连接后的程序反编译看一下，发现连接器已经为程序分配了一个地址，例如，要调用某个函数 A，代码不是 call A，而是 call 0x0811111111 ，也就是说，函数 A 的地址已经被定下来了。没有这样的“转换”，没有虚拟地址的概念，这样做是根本行不通的。

2、地址转换

第一步：CPU 段式管理中——逻辑地址转线性地址

CPU 要利用其段式内存管理单元，先将为个逻辑地址转换成一个线程地址。

一个逻辑地址由两部份组成，【段标识符：段内偏移量】。

段标识符是由一个 16 位长的字段组成，称为段选择符。其中前 13 位是一个索引号。后面 3 位包含一些硬件细节，如图：

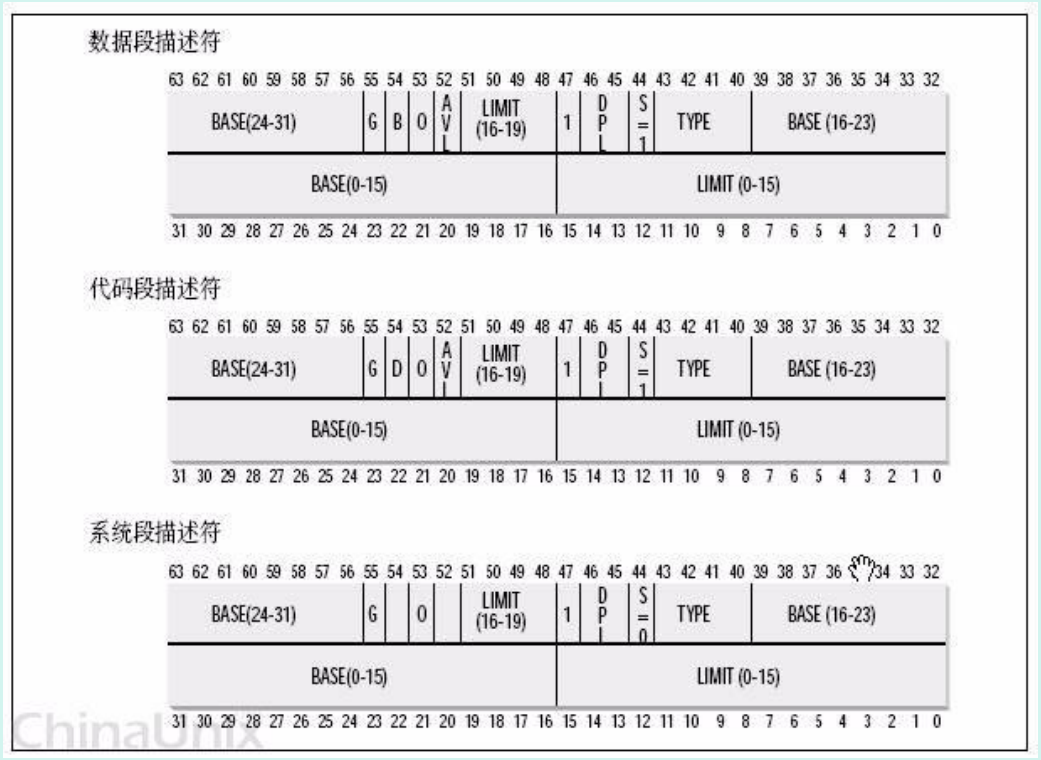
通过段标识符中的索引号从 GDT 或者 LDT 找到该段的段描述符，段描述符中的 base 字段是段的起始地址

段描述符：Base 字段，它描述了一个段的开始位置的线性地址。

一些全局的段描述符，就放在“全局段描述符表(GDT)”中，一些局部的，例如每个进程自己的，就放在所谓的“局部段描述符表(LDT)”中。

GDT 在内存中的地址和大小存放在 CPU 的 gdtr 控制寄存器中，而 LDT 则在 ldtr 寄存器中。

段起始地址 + 段内偏移量 = 线性地址

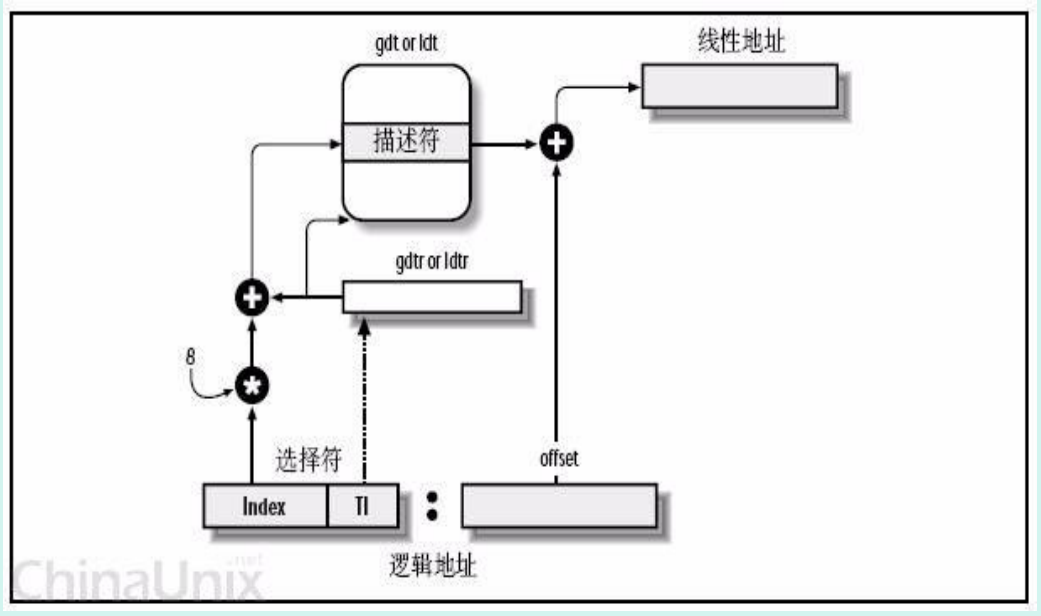


首先，给定一个完整的逻辑地址[段选择符：段内偏移地址]，

1、看段选择符的 T1=0 还是 1，知道当前要转换是 GDT 中的段，还是 LDT 中的段，再根据相应寄存器，得到其地址和大小。我们就有了一个数组了。

2、拿出段选择符中前 13 位，可以在这个数组中，查找到对应的段描述符，这样，它了 Base，即基地址就知道了。

3、把 Base + offset，就是要转换的线性地址了。



第一步：页式管理——线性地址转物理地址

再利用其页式内存管理单元，转换为最终物理地址。

linux 假的段式管理

Intel 要求两次转换，这样虽说是兼容了，但是却是很冗余，但是这是 intel 硬件的要求。

其它某些硬件平台，没有二次转换的概念，Linux 也需要提供一个高层抽像，来提供一个统一的界面。

所以，Linux 的段式管理，事实上只是“哄骗”了一下硬件而已。

按照 Intel 的本意，全局的用 GDT，每个进程自己的用 LDT——不过 Linux 则对所有的进程都使用了相同的段来对指令和数据寻址。即用户数据段，用户代码段，对应的，内核中的是内核数据段和内核代码段。

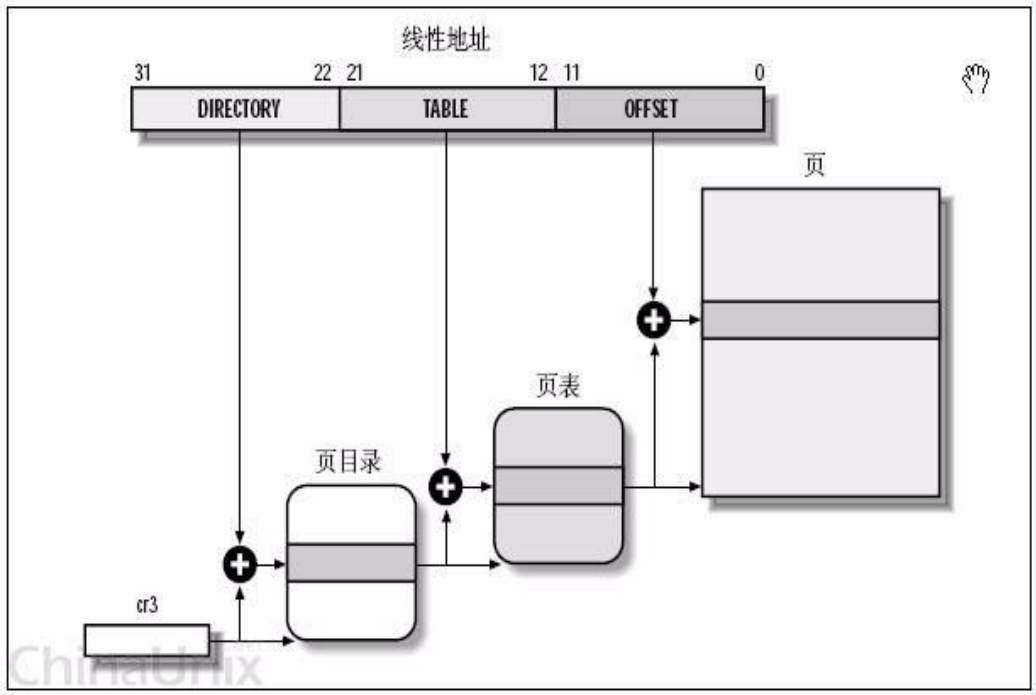
在 Linux 下，逻辑地址与线性地址总是一致的，即逻辑地址的偏移量字段的值与线性地址的值总是相同的。

linux 页式管理

CPU 的页式内存管理单元，负责把一个线性地址，最终翻译为一个物理地址。

线性地址被分为以固定长度为单位的组，称为页(page)，例如一个 32 位的机器，线性地址最大可为4G,可以用4KB为一个页来划分,这页,整个线性地址就被划分为一个tatol_page[2^20]的大数组，共有 2 的 20 个次方个页。

另一类“页”，我们称之为物理页，或者是页框、页桢的。是分页单元把所有的物理内存也划分为固定长度的管理单位，它的长度一般与内存页是一一对应的。



每个进程都有自己的页目录，当进程处于运行态的时候，其页目录地址存放在 cr3 寄存器中。

每一个 32 位的线性地址被划分为三部份，【页目录索引(10 位)：页表索引(10 位)：页内偏移(12 位)】

依据以下步骤进行转换：

从 cr3 中取出进程的页目录地址（操作系统负责在调度进程的时候，把这个地址装入对应寄存器）；

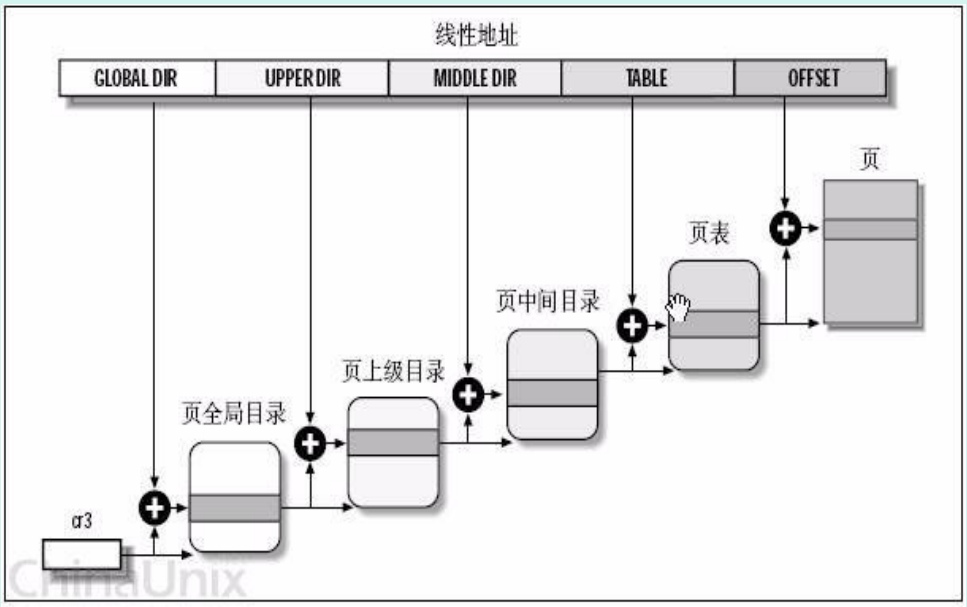
根据线性地址前十位，在数组中，找到对应的索引项，因为引入了二级管理模式，页目录中的项，不再是页的地址，而是一个页表的地址。（又引入了一个数组），页的地址被放到页表中去了。

根据线性地址的中间十位，在页表（也是数组）中找到页的起始地址；

将页的起始地址与线性地址中最后 12 位相加。

目的：

内存节约：如果一级页表中的一个页表条目为空，那么那所指的二级页表就根本不会存在。这表现出一种巨大的潜在节约，因为对于一个典型的程序，4GB 虚拟地址空间的大部份都会是未分配的；



32 位，PGD = 10bit，PUD = PMD = 0，table = 10bit，offset = 12bit

64 位，PUD 和 PMD ≠ 0

22、请你说一说操作系统中的结构体对齐，字节对齐

考察点：操作系统

参考回答：

1、原因：

1）平台原因（移植原因）：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

2）性能原因：数据结构（尤其是栈）应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

2、规则

1）数据成员对齐规则：结构(struct)(或联合(union))的数据成员，第一个数据成员放在offset 为 0 的地方，以后每个数据成员的对齐按照#pragma pack 指定的数值和这个数据成员自身长度中，比较小的那个进行。

2）结构(或联合)的整体对齐规则：在数据成员完成各自对齐之后，结构(或联合)本身也要进行对齐，对齐将按照#pragma pack 指定的数值和结构(或联合)最大数据成员长度中，比较小的那个进行。

3）结构体作为成员：如果一个结构里有某些结构体成员，则结构体成员要从其内部最大元素大小的整数倍地址开始存储。

3、定义结构体对齐

可以通过预编译命令#pragma pack(n)，n=1,2,4,8,16 来改变这一系数，其中的 n 就是指定的“对齐系数”。

4、举例

```
#pragma pack(2)
struct AA {
    int a;        //长度 4 > 2 按 2 对齐；偏移量为 0；存放位置区间[0,3]
    char b;  //长度 1 < 2 按 1 对齐；偏移量为 4；存放位置区间[4]
    short c;      //长度 2 = 2 按 2 对齐;偏移量要提升到 2 的倍数 6;存放位置区间[6,7]
    char d;  //长度 1 < 2 按 1 对齐；偏移量为 7；存放位置区间[8]；共九个字节
};
#pragma pack()
```

23、请问进程间怎么通信

考察点：操作系统

参考回答：

进程间通信主要包括管道、系统 IPC（包括消息队列、信号量、信号、共享内存等）、以及套接字 socket。

1.管道：

管道主要包括无名管道和命名管道:管道可用于具有亲缘关系的父子进程间的通信，有名管道除了具有管道所具有的功能外，它还允许无亲缘关系进程间的通信

1.1 普通管道 PIPE：

1）它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端

2）它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）

3）它可以看成是一种特殊的文件，对于它的读写也可以使用普通的 read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

1.2 命名管道 FIFO：

1）FIFO 可以在无关的进程之间交换数据

2）FIFO 有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

2. 系统 IPC：

2.1 消息队列

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列 ID）来标记。(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特点)具有写权限得进程可以按照一定得规则向消息队列中添加新信息；对消息队列有读权限得进程则可以从消息队列中读取信息；

特点：

1）消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。

2）消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。

3）消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。

2.2 信号量 semaphore

信号量（semaphore）与已经介绍过的 IPC 结构不同，它是一个计数器，可以用来控制多个进程对共享资源的访问。信号量用于实现进程间的互斥与同步,而不是用于存储进程间通信数据。

特点：

1）信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

2）信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。

3）每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

4）支持信号量组。

2.3 信号 signal

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。

2.4 共享内存（Shared Memory）

它使得多个进程可以访问同一块内存空间,不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等

特点：

1）共享内存是最快的一种 IPC，因为进程是直接对内存进行存取

2）因为多个进程可以同时操作，所以需要进行同步

3）信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问

3. 套接字 SOCKET：

socket 也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同主机之间的进程通信。

24、请你说一下虚拟内存置换的方式

考察点：操作系统

参考回答：

比较常见的内存替换算法有：FIFO，LRU，LFU，LRU-K，2Q。

1、FIFO（先进先出淘汰算法）

思想：最近刚访问的，将来访问的可能性比较大。

实现：使用一个队列，新加入的页面放入队尾，每次淘汰队首的页面，即最先进入的数据，最先被淘汰。

弊端：无法体现页面冷热信息

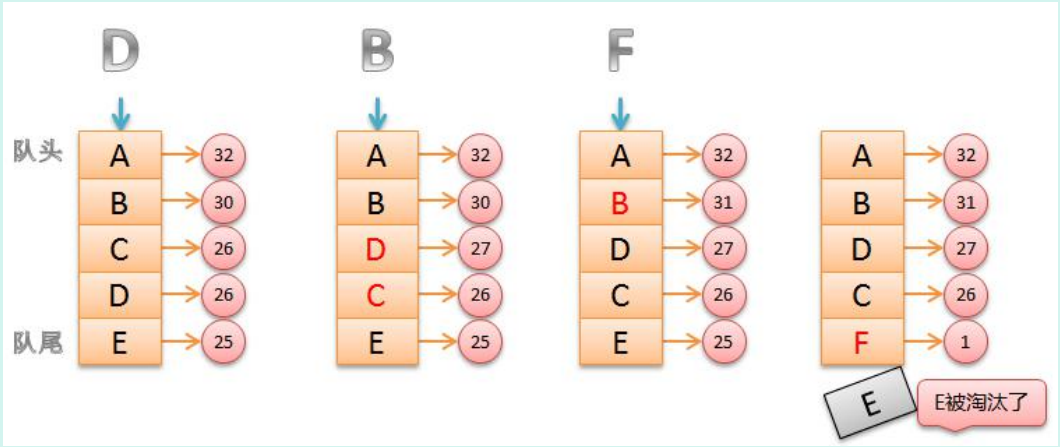
2、LFU（最不经常访问淘汰算法）

思想：如果数据过去被访问多次，那么将来被访问的频率也更高。

实现：每个数据块一个引用计数，所有数据块按照引用计数排序，具有相同引用计数的数据块则按照时间排序。每次淘汰队尾数据块。

开销：排序开销。

弊端：缓存颠簸。



3、LRU（最近最少使用替换算法）

思想：如果数据最近被访问过，那么将来被访问的几率也更高。

实现：使用一个栈，新页面或者命中的页面则将该页面移动到栈底，每次替换栈顶的缓存页面。

优点：LRU 算法对热点数据命中率是很高的。

缺陷：

1）缓存颠簸，当缓存（1，2，3）满了，之后数据访问（0，3，2，1，0，3，2，1。。。）。

2）缓存污染，突然大量偶发性的数据访问，会让内存中存放大量冷数据。

4、LRU-K（LRU-2、LRU-3）

思想：最久未使用 K 次淘汰算法。

LRU-K 中的 K 代表最近使用的次数，因此 LRU 可以认为是 LRU-1。LRU-K 的主要目的是为了解决 LRU 算法“缓存污染”的问题，其核心思想是将“最近使用过 1 次”的判断标准扩展为“最近使用过 K 次”。

相比 LRU，LRU-K 需要多维护一个队列，用于记录所有缓存数据被访问的历史。只有当数据的访问次数达到 K 次的时候，才将数据放入缓存。当需要淘汰数据时，LRU-K 会淘汰第 K 次访问时间距当前时间最大的数据。

实现：

1）数据第一次被访问，加入到访问历史列表；

2）如果数据在访问历史列表里后没有达到 K 次访问，则按照一定规则（FIFO，LRU）淘汰；

3）当访问历史队列中的数据访问次数达到 K 次后，将数据索引从历史队列删除，将数据移到缓存队列中，并缓存此数据，缓存队列重新按照时间排序；

4）缓存数据队列中被再次访问后，重新排序；

5）需要淘汰数据时，淘汰缓存队列中排在末尾的数据，即：淘汰“倒数第 K 次访问离现在最久”的数据。

针对问题：

LRU-K 的主要目的是为了解决 LRU 算法“缓存污染”的问题，其核心思想是将“最近使用过 1 次”的判断标准扩展为“最近使用过 K 次”。

5、2Q

类似 LRU-2。使用一个 FIFO 队列和一个 LRU 队列。

实现：

1）新访问的数据插入到 FIFO 队列；

2）如果数据在 FIFO 队列中一直没有被再次访问，则最终按照 FIFO 规则淘汰；

3）如果数据在 FIFO 队列中被再次访问，则将数据移到 LRU 队列头部；

4）如果数据在 LRU 队列再次被访问，则将数据移到 LRU 队列头部；

5）LRU 队列淘汰末尾的数据。

针对问题：LRU 的缓存污染

弊端：

当 FIFO 容量为 2 时，访问负载是：ABCABCABC 会退化为 FIFO，用不到 LRU。

25、请你说一下多线程，线程同步的几种方式

考察点：操作系统

参考回答：

概念：

进程是对运行时程序的封装，是系统进行资源调度和分配的的基本单位，实现了操作系统的并发；

线程是进程的子任务，是 CPU 调度和分派的基本单位，用于保证程序的实时性，实现进程内部的并发;线程是操作系统可识别的最小执行和调度单位。每个线程都独自占用一个虚拟处理器：独自的寄存器组，指令计数器和处理器状态。每个线程完成不同的任务，但是共享同一地址空间（也就是同样的动态内存，映射文件，目标代码等等），打开的文件队列和其他内核资源。

线程间通信的方式:

1、临界区：

通过多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问；

2、互斥量 Synchronized/Lock：

采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问

3、信号量 Semphare：

为控制具有有限数量的用户资源而设计的，它允许多个线程在同一时刻去访问同一个资源，但一般需要限制同一时刻访问此资源的最大线程数目。

4、事件(信号)，Wait/Notify：

通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作

26、请你讲述一下互斥锁（mutex）机制，以及互斥锁和读写锁的区别

考察点：操作系统

参考回答：

1、互斥锁和读写锁区别：

互斥锁：mutex，用于保证在任何时刻，都只能有一个线程访问该对象。当获取锁操作失败时，线程会进入睡眠，等待锁释放时被唤醒。

读写锁：rwlock，分为读锁和写锁。处于读操作时，可以允许多个线程同时获得读操作。但是同一时刻只能有一个线程可以获得写锁。其它获取写锁失败的线程都会进入睡眠状态，直到写锁释放时被唤醒。 注意：写锁会阻塞其它读写锁。当有一个线程获得写锁在写时，读锁也不能被其它线程获取；写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）。适用于读取数据的频率远远大于写数据的频率的场合。

互斥锁和读写锁的区别：

1）读写锁区分读者和写者，而互斥锁不区分

2）互斥锁同一时间只允许一个线程访问该对象，无论读写；读写锁同一时间内只允许一个写者，但是允许多个读者同时读对象。

2、Linux 的 4 种锁机制：

互斥锁：mutex，用于保证在任何时刻，都只能有一个线程访问该对象。当获取锁操作失败时，线程会进入睡眠，等待锁释放时被唤醒

读写锁：rwlock，分为读锁和写锁。处于读操作时，可以允许多个线程同时获得读操作。但是同一时刻只能有一个线程可以获得写锁。其它获取写锁失败的线程都会进入睡眠状态，直到写锁释放时被唤醒。 注意：写锁会阻塞其它读写锁。当有一个线程获得写锁在写时，读锁也不能被其它线程获取；写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）。适用于读取数据的频率远远大于写数据的频率的场合。

自旋锁：spinlock，在任何时刻同样只能有一个线程访问对象。但是当获取锁操作失败时，不会进入睡眠，而是会在原地自旋，直到锁被释放。这样节省了线程从睡眠状态到被唤醒期间的消耗，在加锁时间短暂的环境下会极大的提高效率。但如果加锁时间过长，则会非常浪费 CPU 资源。

RCU：即 read-copy-update，在修改数据时，首先需要读取数据，然后生成一个副本，对副本进行修改。修改完成后，再将老数据 update 成新的数据。使用 RCU 时，读者几乎不需要同步开销，既不需要获得锁，也不使用原子指令，不会导致锁竞争，因此就不用考虑死锁问题了。而对于写者的同步开销较大，它需要复制被修改的数据，还必须使用锁机制同步并行其它写者的修改操作。在有大量读操作，少量写操作的情况下效率非常高。

27、请回答一下进程和线程的区别

考察点：操作系统

参考回答:

1、一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。线程依赖于进程而存在。

2、进程在执行过程中拥有独立的内存单元，而多个线程共享进程的内存。（资源分配给进程，同一进程的所有线程共享该进程的所有资源。同一进程中的多个线程共享代码段（代码和常量），数据段（全局变量和静态变量），扩展段（堆存储）。但是每个线程拥有自己的栈段，栈段又叫运行时段，用来存放所有局部变量和临时变量。）

3、进程是资源分配的最小单位，线程是 CPU 调度的最小单位。

4、系统开销： 由于在创建或撤消进程时，系统都要为之分配或回收资源，如内存空间、I／o 设备等。因此，操作系统所付出的开销将显著地大于在创建或撤消线程时的开销。类似地，在进行进程切换时，涉及到整个当前进程 CPU 环境的保存以及新被调度运行的进程的 CPU 环境的设置。而线程切换只须保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作。可见，进程切换的开销也远大于线程切换的开销。

5、通信：由于同一进程中的多个线程具有相同的地址空间，致使它们之间的同步和通信的实现，也变得比较容易。进程间通信 IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。在有的系统中，线程的切换、同步和通信都无须操作系统内核的干预 。

6、进程编程调试简单可靠性高，但是创建销毁开销大；线程正相反，开销小，切换速度快，但是编程调试相对复杂。

7、进程间不会相互影响 ；线程一个线程挂掉将导致整个进程挂掉。

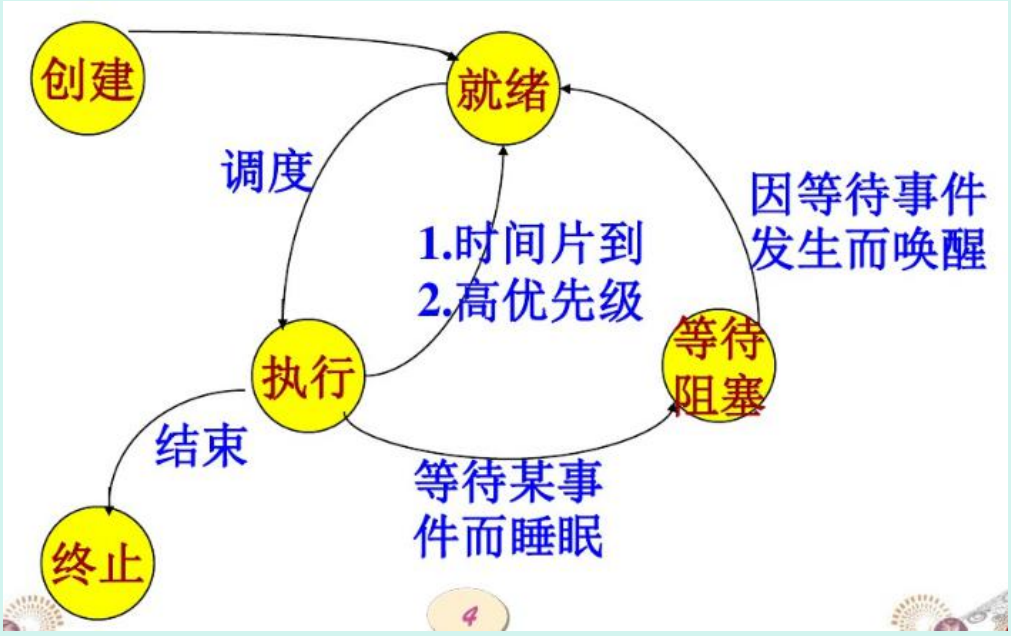
8、进程适应于多核、多机分布；线程适用于多核。

28、请你说一说进程状态转换图，动态就绪，静态就绪，动态阻塞，静态阻塞

考察点：操作系统

参考回答：

1、进程的五种基本状态：



1）创建状态：进程正在被创建

2）就绪状态：进程被加入到就绪队列中等待 CPU 调度运行

3）执行状态：进程正在被运行

4）等待阻塞状态：进程因为某种原因，比如等待 I/0，等待设备，而暂时不能运行。

5）终止状态：进程运行完毕

2、交换技术

当多个进程竞争内存资源时，会造成内存资源紧张，并且，如果此时没有就绪进程，处理机会空闲，I/O 速度比处理机速度慢得多，可能出现全部进程阻塞等待 I/O。

针对以上问题，提出了两种解决方法：

1）交换技术：换出一部分进程到外存，腾出内存空间。

2）虚拟存储技术：每个进程只能装入一部分程序和数据。

在交换技术上，将内存暂时不能运行的进程，或者暂时不用的数据和程序，换出到外存，来腾出足够的内存空间，把已经具备运行条件的进程，或进程所需的数据和程序换入到内存。

从而出现了进程的挂起状态：进程被交换到外存，进程状态就成为了挂起状态。

3、活动阻塞，静止阻塞，活动就绪，静止就绪

1）活动阻塞：进程在内存，但是由于某种原因被阻塞了。

2）静止阻塞：进程在外存，同时被某种原因阻塞了。

3）活动就绪：进程在内存，处于就绪状态，只要给 CPU 和调度就可以直接运行。

4）静止就绪：进程在外存，处于就绪状态，只要调度到内存，给 CPU 和调度就可以运行。

从而出现了：

活动就绪 —— 静止就绪 （内存不够，调到外存）

活动阻塞 —— 静止阻塞 （内存不够，调到外存）

执行 —— 静止就绪 （时间片用完）

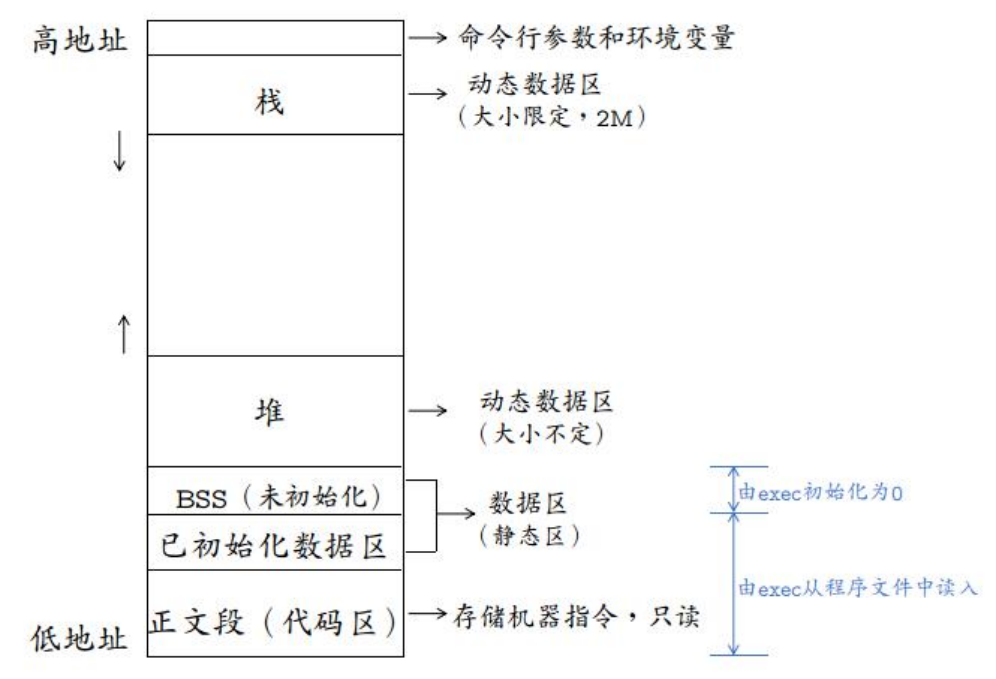
29、A* a = new A; a->i = 10;在内核中的内存分配上发生了什么？

考察点：操作系统

参考回答：

1、程序内存管理：

一个程序本质上都是由 BSS 段、data 段、text 段三个组成的。可以看到一个可执行程序在存储（没有调入内存）时分为代码段、数据区和未初始化数据区三部分。



BSS 段（未初始化数据区）：通常用来存放程序中未初始化的全局变量和静态变量的一块内存区域。BSS 段属于静态分配，程序结束后静态变量资源由系统自动释放。

数据段：存放程序中已初始化的全局变量的一块内存区域。数据段也属于静态内存分配

代码段：存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域属于只读。在代码段中，也有可能包含一些只读的常数变量

text 段和 data 段在编译时已经分配了空间，而 BSS 段并不占用可执行文件的大小，它是由链接器来获取内存的。

bss 段（未进行初始化的数据）的内容并不存放在磁盘上的程序文件中。其原因是内核在程序开始运行前将它们设置为 0。需要存放在程序文件中的只有正文段和初始化数据段。

data 段（已经初始化的数据）则为数据分配空间，数据保存到目标文件中。

数据段包含经过初始化的全局变量以及它们的值。BSS 段的大小从可执行文件中得到，然后链接器得到这个大小的内存块，紧跟在数据段的后面。当这个内存进入程序的地址空间后全部清零。包含数据段和 BSS 段的整个区段此时通常称为数据区。

可执行程序在运行时又多出两个区域：栈区和堆区。

栈区：由编译器自动释放，存放函数的参数值、局部变量等。每当一个函数被调用时，该函数的返回类型和一些调用的信息被存放到栈中。然后这个被调用的 函数再为他的自动变量和临时变量在栈上分配空间。每调用一个函数一个新的栈就会被使用。栈区是从高地址位向低地址位增长的，是一块连续的内存区域，最大容 量是由系统预先定义好的，申请的栈空间超过这个界限时会提示溢出，用户能从栈中获取的空间较小。

堆区：用于动态分配内存，位于 BSS 和栈中间的地址区域。由程序员申请分配和释放。堆是从低地址位向高地址位增长，采用链式存储结构。频繁的 malloc/free 造成内存空间的不连续，产生碎片。当申请堆空间时库函数是按照一定的算法搜索可用的足够大的空间。因此堆的效率比栈要低的多。

2、A* a = new A; a->i = 10：

1）A *a：a 是一个局部变量，类型为指针，故而操作系统在程序栈区开辟 4/8 字节的空间（0x000m），分配给指针 a。

2）new A：通过 new 动态的在堆区申请类 A 大小的空间（0x000n）。

3)a = new A:将指针 a 的内存区域填入栈中类 A 申请到的地址的地址。即*(0x000m)=0x000n。

4）a->i：先找到指针 a 的地址 0x000m，通过 a 的值 0x000n 和 i 在类 a 中偏移 offset，得到 a->i 的地址 0x000n + offset，进行*(0x000n + offset) = 10 的赋值操作，即内存 0x000n + offset 的值是 10。

30、给你一个类，里面有 static，virtual，之类的，来说一说这个类的内存分布

考察点：操作系统

参考回答：

1、static 修饰符

1）static 修饰成员变量

对于非静态数据成员，每个类对象都有自己的拷贝。而静态数据成员被当做是类的成员，无论这个类被定义了多少个，静态数据成员都只有一份拷贝，为该类型的所有对象所共享(包括其派生类)。所以，静态数据成员的值对每个对象都是一样的，它的值可以更新。

因为静态数据成员在全局数据区分配内存，属于本类的所有对象共享，所以它不属于特定的类对象，在没有产生类对象前就可以使用。

2）static 修饰成员函数

与普通的成员函数相比，静态成员函数由于不是与任何的对象相联系，因此它不具有 this 指针。从这个意义上来说，它无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，只能调用其他的静态成员函数。

Static 修饰的成员函数，在代码区分配内存。

2、C++继承和虚函数

C++多态分为静态多态和动态多态。静态多态是通过重载和模板技术实现，在编译的时候确定。动态多态通过虚函数和继承关系来实现，执行动态绑定，在运行的时候确定。

动态多态实现有几个条件：

(1) 虚函数；

(2) 一个基类的指针或引用指向派生类的对象；

基类指针在调用成员函数(虚函数)时，就会去查找该对象的虚函数表。虚函数表的地址在每个对象的首地址。查找该虚函数表中该函数的指针进行调用。

每个对象中保存的只是一个虚函数表的指针，C++内部为每一个类维持一个虚函数表，该类的对象的都指向这同一个虚函数表。

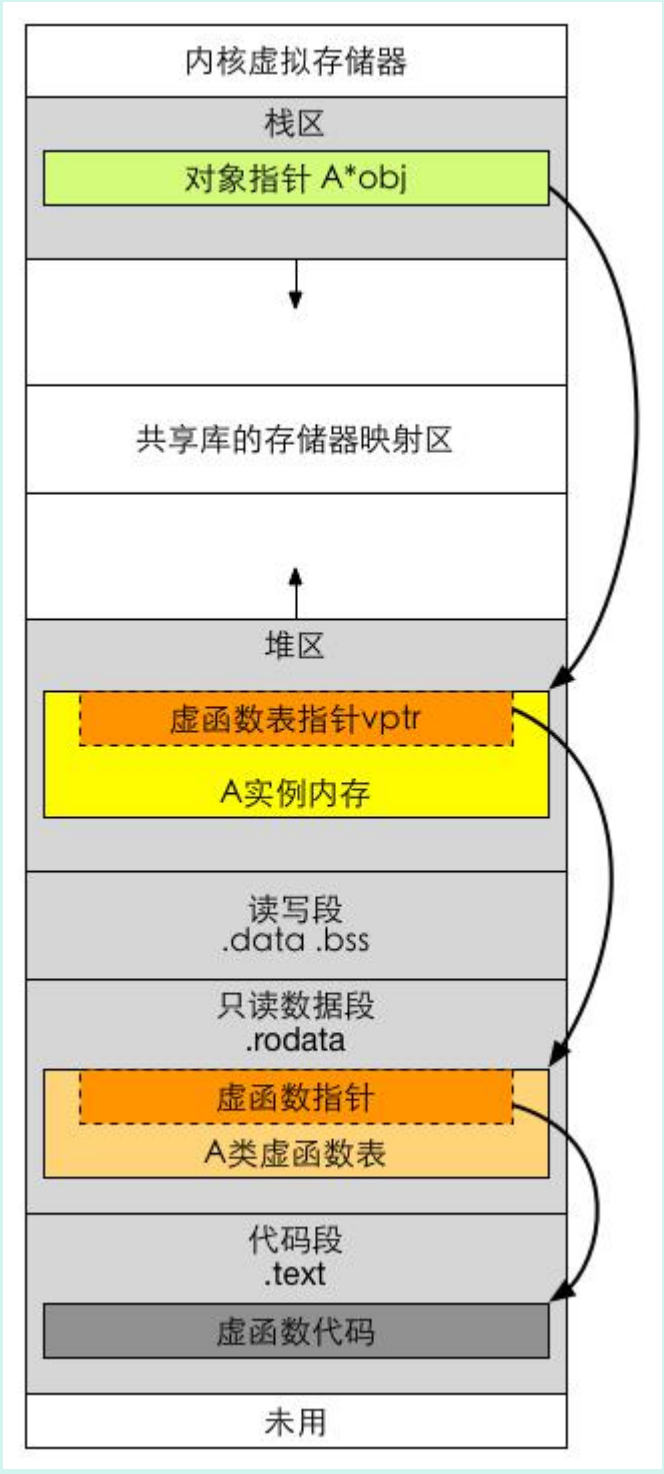
虚函数表中为什么就能准确查找相应的函数指针呢？因为在类设计的时候，虚函数表直接从基类也继承过来，如果覆盖了其中的某个虚函数，那么虚函数表的指针就会被替换，因此可以根据指针准确找到该调用哪个函数。

3、virtual 修饰符

如果一个类是局部变量则该类数据存储在栈区,如果一个类是通过 new/malloc 动态申请的，则该类数据存储在堆区。

如果该类是 virutal 继承而来的子类，则该类的虚函数表指针和该类其他成员一起存储。虚函数表指针指向只读数据段中的类虚函数表，虚函数表中存放着一个个函数指针，函数指针指向代码段中的具体函数。

如果类中成员是 virtual 属性，会隐藏父类对应的属性。



31、请你回答一下软链接和硬链接区别

考点：软链接 硬链接

参考回答：

为了解决文件共享问题，Linux 引入了软链接和硬链接。除了为 Linux 解决文件共享使用，还带来了隐藏文件路径、增加权限安全及节省存储等好处。若 1 个 inode 号对应多个文件名，则为硬链接，即硬链接就是同一个文件使用了不同的别名，使用 ln 创建。若文件用户数据块中存放的内容是另一个文件的路径名指向，则该文件是软连接。软连接是一个普通文件，有自己独立的 inode，但是其数据块内容比较特殊。

32、请问什么是大端小端以及如何判断大端小端

考点：大端 小端 联合体

参考回答：

大端是指低字节存储在高地址；小端存储是指低字节存储在低地址。我们可以根据联合体来判断该系统是大端还是小端。因为联合体变量总是从低地址存储。

```
//判断系统是大端还是小端:通过联合体，因为联合体的所有成员都从低地址开始存放
int fun1()
{
    union test
    {
        int i;
        char c;
    };

    test t;
    t.i = 1;

    //如果是大端,则t.c为0x00,则t.c!=1,返回0 是小端,则t.c为0x01,则t.c==1,返回1
    return (t.c==1);
}
```

33、请你回答一下静态变量什么时候初始化

考点：静态变量

参考回答：

静态变量存储在虚拟地址空间的数据段和 bss 段，C 语言中其在代码执行之前初始化，属于编译期初始化。而 C++中由于引入对象，对象生成必须调用构造函数，因此 C++规定全局或局部静态对象当且仅当对象首次用到时进行构造

34、请你说一说用户态和内核态区别

参考回答：

用户态和内核态是操作系统的两种运行级别，两者最大的区别就是特权级不同。用户态拥有最低的特权级，内核态拥有较高的特权级。运行在用户态的程序不能直接访问操作系统内核数据结构和程序。内核态和用户态之间的转换方式主要包括：系统调用，异常和中断。

35、请问如何设计 server，使得能够接收多个客户端的请求

参考回答：

多线程，线程池，io 复用

36、死循环+来连接时新建线程的方法效率有点低，怎么改进？

参考回答：

提前创建好一个线程池，用生产者消费者模型，创建一个任务队列，队列作为临界资源，有了新连接，就挂在到任务队列上，队列为空所有线程睡眠。改进死循环：使用 select epoll 这样的技术

37、请问怎么唤醒被阻塞的 socket 线程？

参考回答：

给阻塞时候缺少的资源

38、请问怎样确定当前线程是繁忙还是阻塞？

参考回答：

使用 ps 命令查看

39、空闲的进程和阻塞的进程状态会不会在唤醒的时候误判？

参考回答：

略

40、请问就绪状态的进程在等待什么？

参考回答：

被调度使用 cpu 的运行权

41、请你说一说多线程的同步，锁的机制

参考回答：

同步的时候用一个互斥量，在访问共享资源前对互斥量进行加锁，在访问完成后释放互斥量上的锁。对互斥量进行加锁以后，任何其他试图再次对互斥量加锁的线程将会被阻塞直到当前线程释放该互斥锁。如果释放互斥锁时有多个线程阻塞，所有在该互斥锁上的阻塞线程都会变成可运行状态，第一个变为运行状态的线程可以对互斥量加锁，其他线程将会看到互斥锁依然被锁住，只能回去再次等待它重新变为可用。在这种方式下，每次只有一个线程可以向前执行。

42、两个进程访问临界区资源，会不会出现都获得自旋锁的情况？

参考回答：

单核 cpu，并且开了抢占可以造成这种情况。

43、假设临界区资源释放，如何保证只让一个线程获得临界区资源而不是都获得？

参考回答：

略

44、windows 消息机制知道吗，请说一说

参考回答：

当用户有操作(鼠标，键盘等)时，系统会将这些时间转化为消息。每个打开的进程系统都为其维护了一个消息队列，系统会将这些消息放到进程的消息队列中，而应用程序会循环从消息队列中取出来消息，完成对应的操作。

45、C++的锁你知道几种?

参考回答：

锁包括互斥锁，条件变量，自旋锁和读写锁

46、说一说你用到的锁

参考回答：

生产者消费者问题利用互斥锁和条件变量可以很容易解决,条件变量这里起到了替代信号量的作用

47、请你说一说死锁产生的必要条件？

参考回答：

1.互斥条件：一个资源每次只能被一个进程使用。
2.请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
3.不剥夺条件:进程已获得的资源，在未使用完之前，不能强行剥夺。
4.循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

48、请你说一说什么是线程和进程，多线程和多进程通信方式

考察点：操作系统

1）概念：

进程是对运行时程序的封装，是系统进行资源调度和分配的的基本单位，实现了操作系统的并发；

线程是进程的子任务，是 CPU 调度和分派的基本单位，用于保证程序的实时性，实现进程内部的并发；线程是操作系统可识别的最小执行和调度单位。每个线程都独自占用一个虚拟处理器：独自的寄存器组，指令计数器和处理器状态。每个线程完成不同的任务，但是共享同一地址空间（也就是同样的动态内存，映射文件，目标代码等等），打开的文件队列和其他内核资源。

2）进程间通信的方式:

进程间通信主要包括管道、系统 IPC（包括消息队列、信号量、信号、共享内存等）、以及套接字 socket。

1、管道：

管道主要包括无名管道和命名管道:管道可用于具有亲缘关系的父子进程间的通信，有名管道除了具有管道所具有的功能外，它还允许无亲缘关系进程间的通信

普通管道 PIPE：

它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端

它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）

它可以看成是一种特殊的文件，对于它的读写也可以使用普通的 read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

命名管道 FIFO：

FIFO 可以在无关的进程之间交换数据

FIFO 有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

2、消息队列

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列 ID）来标记。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特点。具有写权限得进程可以按照一定得规则向消息队列中添加新信息，对消息队列有读权限得进程则可以从消息队列中读取信息。消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。

消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。

消息队列可以实现消息的随机查询，消息不一定要以先进先出的次序读取，也可以按消息的类型读取。

3、信号量 semaphore

信号量（semaphore）与已经介绍过的 IPC 结构不同，它是一个计数器，可以用 来控制多个进程对共享资源的访问。信号量用于实现进程间的互斥与同步，而不是用于 存储进程间通信数据。

信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。

信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。

每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

支持信号量组。

4 信号 signal

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。

5 共享内存（Shared Memory）

它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等。

共享内存是最快的一种 IPC，因为进程是直接对内存进行存取

因为多个进程可以同时操作，所以需要进行同步

信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问

6、套接字 SOCKET：

socket 也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同主机之间的进程通信。

3）线程间通信的方式:

1、临界区：

通过多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问；

2、互斥量 Synchronized/Lock：

采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问

3、信号量 Semphare：

为控制具有有限数量的用户资源而设计的，它允许多个线程在同一时刻去访问同一个资源，但一般需要限制同一时刻访问此资源的最大线程数目。

4、事件(信号)，Wait/Notify：

通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作。

49、请你说一说内存溢出和内存泄漏

考察点：C/C++

参考回答：

1、内存溢出

指程序申请内存时，没有足够的内存供申请者使用。内存溢出就是你要的内存空间超过了系统实际分配给你的空间，此时系统相当于没法满足你的需求，就会报内存溢出的错误

内存溢出原因：

内存中加载的数据量过于庞大，如一次从数据库取出过多数据

集合类中有对对象的引用，使用完后未清空，使得不能回收

代码中存在死循环或循环产生过多重复的对象实体

使用的第三方软件中的 BUG

启动参数内存值设定的过小

2、内存泄漏

内存泄漏是指由于疏忽或错误造成了程序未能释放掉不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。

内存泄漏的分类：

1、堆内存泄漏 （Heap leak）。对内存指的是程序运行中根据需要分配通过 malloc,realloc new 等从堆中分配的一块内存，再是完成后必须通过调用对应的 free 或者 delete 删掉。如果程序的设计的错误导致这部分内存没有被释放，那么此后这块内存将不会被使用，就会产生 Heap Leak。

2、系统资源泄露（Resource Leak）。主要指程序使用系统分配的资源比如 Bitmap,handle ,SOCKET 等没有使用相应的函数释放掉，导致系统资源的浪费，严重可导致系统效能降低，系统运行不稳定。

3、没有将基类的析构函数定义为虚函数。当基类指针指向子类对象时，如果基类的析构函数不是 virtual，那么子类的析构函数将不会被调用，子类的资源没有正确是释放，因此造成内存泄露。

50、进程和线程的区别，你都使用什么线程模型

考察点：

参考回答：

1）进程和线程区别

1、一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。线程依赖于进程而存在。

2、进程在执行过程中拥有独立的内存单元，而多个线程共享进程的内存。（资源分配给进程，同一进程的所有线程共享该进程的所有资源。同一进程中的多个线程共享代码段（代码和常量），数据段（全局变量和静态变量），扩展段（堆存储）。但是每个线程拥有自己的栈段，栈段又叫运行时段，用来存放所有局部变量和临时变量。）

3、进程是资源分配的最小单位，线程是 CPU 调度的最小单位。

4、系统开销： 由于在创建或撤消进程时，系统都要为之分配或回收资源，如内存空间、I／o 设备等。因此，操作系统所付出的开销将显著地大于在创建或撤消线程时的开销。类似地，在进行进程切换时,涉及到整个当前进程CPU环境的保存以及新被调度运行的进程的CPU环境的设置。而线程切换只须保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作。可见，进程切换的开销也远大于线程切换的开销。

5、通信：由于同一进程中的多个线程具有相同的地址空间，致使它们之间的同步和通信的实现，也变得比较容易。进程间通信 IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。在有的系统中，线程的切换、同步和通信都无须操作系统内核的干预 。

6、进程编程调试简单可靠性高，但是创建销毁开销大；线程正相反，开销小，切换速度快，但是编程调试相对复杂。

7、进程间不会相互影响；线程一个线程挂掉将导致整个进程挂掉。

8、进程适应于多核、多机分布；线程适用于多核。

2、常用线程模型

1、Future 模型

该模型通常在使用的时候需要结合 Callable 接口配合使用。

Future 是把结果放在将来获取，当前主线程并不急于获取处理结果。允许子线程先进行处理一段时间，处理结束之后就把结果保存下来，当主线程需要使用的时候再向子线程索取。

Callable 是类似于 Runnable 的接口，其中 call 方法类似于 run 方法，所不同的是 run 方法不能抛出受检异常没有返回值，而 call 方法则可以抛出受检异常并可设置返回值。两者的方法体都是线程执行体。

2、fork&join 模型

该模型包含递归思想和回溯思想，递归用来拆分任务，回溯用合并结果。 可以用来处理一些可以进行拆分的大任务。其主要是把一个大任务逐级拆分为多个子任务，然后分别在子线程中执行，当每个子线程执行结束之后逐级回溯，返回结果进行汇总合并，最终得出想要的结果。

这里模拟一个摘苹果的场景：有 100 棵苹果树，每棵苹果树有 10 个苹果，现在要把他们摘下来。为了节约时间，规定每个线程最多只能摘 10 棵苹树以便于节约时间。各个线程摘完之后汇总计算总苹果树。

3、actor 模型

actor 模型属于一种基于消息传递机制并行任务处理思想，它以消息的形式来进行线程间数据传输，避免了全局变量的使用，进而避免了数据同步错误的隐患。actor 在接受到消息之后可以自己进行处理，也可以继续传递（分发）给其它 actor 进行处理。在使用 actor 模型的时候需要使用第三方 Akka 提供的框架。

4、生产者消费者模型

生产者消费者模型都比较熟悉，其核心是使用一个缓存来保存任务。开启一个/多个线程来生产任务，然后再开启一个/多个来从缓存中取出任务进行处理。这样的好处是任务的生成和处理分隔开，生产者不需要处理任务，只负责向生成任务然后保存到缓存。而消费者只需要从缓存中取出任务进行处理。使用的时候可以根据任务的生成情况和处理情况开启不同的线程来处理。比如，生成的任务速度较快，那么就可以灵活的多开启几个消费者线程进行处理，这样就可以避免任务的处理响应缓慢的问题。

5、master-worker 模型

master-worker 模型类似于任务分发策略，开启一个 master 线程接收任务，然后在 master 中根据任务的具体情况进行分发给其它 worker 子线程，然后由子线程处理任务。如需返回结果，则 worker 处理结束之后把处理结果返回给 master。

51、请你来说一说协程

考察点：

参考回答：

1、概念：

协程，又称微线程，纤程，英文名 Coroutine。协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

例如：

```
def A() :
    print '1'
    print '2'
    print '3'

def B() :
    print 'x'
    print 'y'
    print 'z'
```

由协程运行结果可能是 12x3yz。在执行 A 的过程中，可以随时中断，去执行 B，B 也可能在执行过程中中断再去执行 A。但协程的特点在于是一个线程执行。

2）协程和线程区别

那和多线程比，协程最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

3）其他

在协程上利用多核 CPU 呢——多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

Python 对协程的支持还非常有限，用在 generator 中的 yield 可以一定程度上实现协程。虽然支持不完全，但已经可以发挥相当大的威力了。

52、系统调用是什么，你用过哪些系统调用

参考回答：

1）概念：

在计算机中，系统调用（英语：system call），又称为系统呼叫，指运行在使用者空间的程序向操作系统内核请求需要更高权限运行的服务。系统调用提供了用户程序与操作系统之间的接口（即系统调用是用户程序和内核交互的接口）。

操作系统中的状态分为管态（核心态）和目态（用户态）。大多数系统交互式操作需求在内核态执行。如设备 IO 操作或者进程间通信。特权指令：一类只能在核心态下运行而不能在用户态下运行的特殊指令。不同的操作系统特权指令会有所差异，但是一般来说主要是和硬件相关的一些指令。用户程序只在用户态下运行，有时需要访问系统核心功能，这时通过系统调用接口使用系统调用。

应用程序有时会需要一些危险的、权限很高的指令，如果把这些权限放心地交给用户程序是很危险的(比如一个进程可能修改另一个进程的内存区，导致其不能运行)，但是又不能完全不给这些权限。于是有了系统调用，危险的指令被包装成系统调用，用户程序只能调用而无权自己运行那些危险的指令。另外，计算机硬件的资源是有限的，为了更好的管理这些资源，所有的资源都由操作系统控制，进程只能向操作系统请求这些资源。操作系统是这些资源的唯一入口，这个入口就是系统调用。

2）系统调用举例：

对文件进行写操作，程序向打开的文件写入字符串“hello world”，open 和 write 都是系统调用。如下：

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<errno.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/stat.h>
```

```
#include<fcntl.h>

int main(int argc, char *argv[])

{

    if (argc<2)

        return 0;

    //用读写追加方式打开一个已经存在的文件

    int fd = open(argv[1], O_RDWR | O_APPEND);

    if (fd == -1)

    {

        printf("error is %s\n", strerror(errno));

    }

    else

    {

        //打印文件描述符号

        printf("success fd = %d\n", fd);

        char buf[100];

        memset(buf, 0, sizeof(buf));

        strcpy(buf, "hello world\n");

        write(fd, buf, strlen(buf));

        close(fd);

    }

    return 0;

}
```

还有写数据 write，创建进程 fork，vfork 等都是系统调用。

53、请你来手写一下 fork 调用示例

考察点：fork，进程

参考回答：

1、概念：

Fork：创建一个和当前进程映像一样的进程可以通过 fork( )系统调用：

成功调用 fork( )会创建一个新的进程，它几乎与调用 fork( )的进程一模一样，这两个进程都会继续运行。在子进程中，成功的 fork( )调用会返回 0。在父进程中 fork( )返回子进程的 pid。如果出现错误，fork( )返回一个负值。

最常见的 fork( )用法是创建一个新的进程，然后使用 exec( )载入二进制映像，替换当前进程的映像。这种情况下，派生（fork）了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。

在早期的 Unix 系统中，创建进程比较原始。当调用 fork 时，内核会把所有的内部数据结构复制一份，复制进程的页表项，然后把父进程的地址空间中的内容逐页的复制到子进程的地址空间中。但从内核角度来说，逐页的复制方式是十分耗时的。现代的 Unix 系统采取了更多的优化，例如 Linux，采用了写时复制的方法，而不是对父进程空间进程整体复制。

2、fork 实例

```
int main(void)
{
    pid_t pid;
    signal(SIGCHLD, SIG_IGN);
    printf("before fork pid:%d\n", getpid());
    int abc = 10;
    pid = fork();
    if (pid == -1) {          //错误返回
        perror("tile");
        return -1;
    }
    if (pid > 0) {            //父进程空间
        abc++;
```

```
        printf("parent:pid:%d \n", getpid());
        printf("abc:%d \n", abc);
        sleep(20);
    }
    else if (pid == 0) {      //子进程空间
        abc++;
        printf("child:%d,parent: %d\n", getpid(), getppid());
        printf("abc:%d", abc);
    }
    printf("fork after...\n");
}
```

54、请你来说一说用户态到内核态的转化原理

考察点：

参考回答：

1）用户态切换到内核态的 3 种方式

1、系统调用

这是用户进程主动要求切换到内核态的一种方式，用户进程通过系统调用申请操作系统提供的服务程序完成工作。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如 Linux 的 ine 80h 中断。

2、异常

当 CPU 在执行运行在用户态的程序时，发现了某些事件不可知的异常，这是会触发由当前运行进程切换到处理此。异常的内核相关程序中，也就到了内核态，比如缺页异常。

3、外围设备的中断

当外围设备完成用户请求的操作之后，会向 CPU 发出相应的中断信号，这时 CPU 会暂停执行下一条将要执行的指令，转而去执行中断信号的处理程序，如果先执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了有用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。

2）切换操作

从出发方式看，可以在认为存在前述 3 种不同的类型，但是从最终实际完成由用户态到内核态的切换操作上来说，涉及的关键步骤是完全一样的，没有任何区别，都相当于执行了一个中断响应的过程，因为系统调用实际上最终是中断机制实现的，而异常和中断处理机制基本上是一样的，用户态切换到内核态的步骤主要包括：

1、从当前进程的描述符中提取其内核栈的 ss0 及 esp0 信息。

2、使用 ss0 和 esp0 指向的内核栈将当前进程的 cs,eip，eflags，ss,esp 信息保存起来，这个过程也完成了由用户栈找到内核栈的切换过程，同时保存了被暂停执行的程序的下一条指令。

3、将先前由中断向量检索得到的中断处理程序的 cs，eip 信息装入相应的寄存器，开始执行中断处理程序，这时就转到了内核态的程序执行了。

55、请你说一下源码到可执行文件的过程

参考回答：

1）预编译

主要处理源代码文件中的以“#”开头的预编译指令。处理规则见下

1、删除所有的#define，展开所有的宏定义。

2、处理所有的条件预编译指令，如“#if”、“#endif”、“#ifdef”、“#elif”和“#else”。

3、处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其他文件。

4、删除所有的注释，“//”和“/**/”。

5、保留所有的#pragma 编译器指令，编译器需要用到他们，如：#pragma once 是为了防止有文件被重复引用。

6、添加行号和文件标识，便于编译时编译器产生调试用的行号信息，和编译时产生编译错误或警告是能够显示行号。

2）编译

把预编译之后生成的 xxx.i 或 xxx.ii 文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相应的汇编代码文件。

1、词法分析：利用类似于“有限状态机”的算法，将源代码程序输入到扫描机中，将其中的字符序列分割成一系列的记号。

2、语法分析：语法分析器对由扫描器产生的记号，进行语法分析，产生语法树。由语法分析器输出的语法树是一种以表达式为节点的树。

3、语义分析：语法分析器只是完成了对表达式语法层面的分析，语义分析器则对表达式是否有意义进行判断，其分析的语义是静态语义——在编译期能分期的语义，相对应的动态语义是在运行期才能确定的语义。

4、优化：源代码级别的一个优化过程。

5、目标代码生成：由代码生成器将中间代码转换成目标机器代码，生成一系列的代码序列——汇编语言表示。

6、目标代码优化：目标代码优化器对上述的目标机器代码进行优化：寻找合适的寻址方式、使用位移来替代乘法运算、删除多余的指令等。

3）汇编

将汇编代码转变成机器可以执行的指令(机器码文件)。 汇编器的汇编过程相对于编译器来说更简单，没有复杂的语法，也没有语义，更不需要做指令优化，只是根据汇编指令和机器指令的对照表一一翻译过来，汇编过程有汇编器 as 完成。经汇编之后，产生目标文件(与可执行文件格式几乎一样)xxx.o(Windows 下)、xxx.obj(Linux 下)。

4）链接

将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链接：

1、静态链接：

函数和数据被编译进一个二进制文件。在使用静态库的情况下，在编译链接可执行文件时，链接器从库中复制这些函数和数据并把它们和应用程序的其它模块组合起来创建最终的可执行文件。

空间浪费：因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以如果多个程序对同一个目标文件都有依赖，会出现同一个目标文件都在内存存在多个副本；

更新困难：每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。

运行速度快：但是静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西，在执行的时候运行速度快。

2、动态链接：

动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。

共享库：就是即使需要每个程序都依赖同一个库，但是该库不会像静态链接那样在内存中存在多分，副本，而是这多个程序在执行时共享同一份副本；

更新方便：更新时只需要替换原来的目标文件，而无需将所有的程序再重新链接一遍。当程序下一次运行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标。

性能损耗：因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损失。

56、请你来说一下微内核与宏内核

考察点：

参考回答：

宏内核：除了最基本的进程、线程管理、内存管理外，将文件系统，驱动，网络协议等等都集成在内核里面，例如 linux 内核。

优点：效率高。

缺点：稳定性差，开发过程中的 bug 经常会导致整个系统挂掉。

微内核：内核中只有最基本的调度、内存管理。驱动、文件系统等都是用户态的守护进程去实现的。

优点：稳定，驱动等的错误只会导致相应进程死掉，不会导致整个系统都崩溃

缺点：效率低。典型代表 QNX，QNX 的文件系统是跑在用户态的进程，称为 resmgr 的东西，是订阅发布机制,文件系统的错误只会导致这个守护进程挂掉。不过数据吞吐量就比较不乐观了。

57、请你说一下僵尸进程

考察点：

参考回答：

1）正常进程

正常情况下，子进程是通过父进程创建的，子进程再创建新的进程。子进程的结束和父进程的运行是一个异步过程，即父进程永远无法预测子进程到底什么时候结束。 当一个进程完成它的工作终止之后，它的父进程需要调用 wait()或者 waitpid()系统调用取得子进程的终止状态。

unix 提供了一种机制可以保证只要父进程想知道子进程结束时的状态信息， 就可以得到：在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存等。 但是仍然为其保留一定的信息，直到父进程通过 wait / waitpid 来取时才释放。保存信息包括：

1 进程号 the process ID

2 退出状态 the termination status of the process

3 运行时间 the amount of CPU time taken by the process 等

2）孤儿进程

一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被 init 进程(进程号为 1)所收养，并由 init 进程对它们完成状态收集工作。

3）僵尸进程

一个进程使用 fork 创建子进程，如果子进程退出，而父进程并没有调用 wait 或 waitpid 获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵尸进程。

僵尸进程是一个进程必然会经过的过程：这是每个子进程在结束时都要经过的阶段。

如果子进程在 exit()之后，父进程没有来得及处理，这时用 ps 命令就能看到子进程的状态是“Z”。如果父进程能及时 处理，可能用 ps 命令就来不及看到子进程的僵尸状态，但这并不等于子进程不经过僵尸状态。

如果父进程在子进程结束之前退出，则子进程将由 init 接管。init 将会以父进程的身份对僵尸状态的子进程进行处理。

危害：

如果进程不调用 wait / waitpid 的话， 那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵死进程，将因为没有可用的进程号而导致系统不能产生新的进程。

外部消灭：

通过 kill 发送 SIGTERM 或者 SIGKILL 信号消灭产生僵尸进程的进程，它产生的僵死进程就变成了孤儿进程，这些孤儿进程会被 init 进程接管，init 进程会 wait()这些孤儿进程，释放它们占用的系统进程表中的资源

内部解决：

1、子进程退出时向父进程发送 SIGCHILD 信号，父进程处理 SIGCHILD 信号。在信号处理函数中调用 wait 进行处理僵尸进程。

2、fork 两次，原理是将子进程成为孤儿进程，从而其的父进程变为 init 进程，通过 init 进程可以处理僵尸进程。

58、请问 GDB 调试用过吗，什么是条件断点

考察点：GDB 调试

参考回答：

1、GDB 调试

GDB 是自由软件基金会（Free Software Foundation）的软件工具之一。它的作用是协助程序员找到代码中的错误。如果没有 GDB 的帮助，程序员要想跟踪代码的执行流程，唯一的办法就是添加大量的语句来产生特定的输出。但这一手段本身就可能会引入新的错误，从而也就无法对那些导致程序崩溃的错误代码进行分析。

GDB 的出现减轻了开发人员的负担，他们可以在程序运行的时候单步跟踪自己的代码，或者通过断点暂时中止程序的执行。此外，他们还能够随时察看变量和内存的当前状态，并监视关键的数据结构是如何影响代码运行的。

2、条件断点

条件断点是当满足条件就中断程序运行，命令：break line-or-function if expr。

例如：(gdb)break 666 if testsize==100

59、请你来介绍一下 5 种 IO 模型

参考回答：

1.阻塞 IO:调用者调用了某个函数，等待这个函数返回，期间什么也不做，不停的去检查这个函数有没有返回，必须等这个函数返回才能进行下一步动作
2.非阻塞 IO:非阻塞等待，每隔一段时间就去检测 IO 事件是否就绪。没有就绪就可以做其他事。
3.信号驱动 IO:信号驱动 IO:linux 用套接口进行信号驱动 IO，安装一个信号处理函数，进程继续运行并不阻塞，当 IO 时间就绪，进程收到 SIGIO 信号。然后处理 IO 事件。
4.IO 复用/多路转接 IO:linux 用 select/poll 函数实现 IO 复用模型，这两个函数也会使进程阻塞，但是和阻塞 IO 所不同的是这两个函数可以同时阻塞多个 IO 操作。而且可以同时对多个读操作、写操作的 IO 函数进行检测。知道有数据可读或可写时，才真正调用 IO 操作函数
5.异步 IO:linux 中，可以调用 aio_read 函数告诉内核描述字缓冲区指针和缓冲区的大小、文件偏移及通知的方式，然后立即返回，当内核将数据拷贝到缓冲区后，再通知应用程序。

60、请你说一说异步编程的事件循环

参考回答：

事件循环就是不停循环等待时间的发生，然后将这个事件的所有处理器，以及他们订阅这个事件的时间顺序依次依次执行。当这个事件的所有处理器都被执行完毕之后，事件循环就会开始继续等待下一个事件的触发，不断往复。当同时并发地处理多个请求时，以上的概念也是正确的，可以这样理解：在单个的线程中，事件处理器是一个一个按顺序执行的。即如果某个事件绑定了两个处理器，那么第二个处理器会在第一个处理器执行完毕后，才开始执行。在这个事件的所有处理器都执行完毕之前，事件循环不会去检查是否有新的事件触发。在单个线程中，一切都是有顺序地一个一个地执行的！

61、请你回答一下操作系统为什么要分内核态和用户态

参考回答：

为了安全性。在 cpu 的一些指令中，有的指令如果用错，将会导致整个系统崩溃。分了内核态和用户态后，当用户需要操作这些指令时候，内核为其提供了 API，可以通过系统调用陷入内核，让内核去执行这些操作。

62、请你回答一下为什么要有 page cache，操作系统怎么设计的 page cache

参考回答：

加快从磁盘读取文件的速率。page cache 中有一部分磁盘文件的缓存，因为从磁盘中读取文件比较慢，所以读取文件先去 page cache 中去查找，如果命中，则不需要去磁盘中读取，大大加快读取速度。在 Linux 内核中，文件的每个数据块最多只能对应一个 Page Cache 项，它通过两个数据结构来管理这些 Cache
项，一个是 radix tree，另一个是双向链表。Radix tree 是一种搜索树，Linux
内核利用这个数据结构来通过文件内偏移快速定位 Cache 项

43、server 端监听端口，但还没有客户端连接进来，此时进程处于什么状态？

参考回答：

这个需要看服务端的编程模型，如果如上一个问题的回答描述的这样，则处于阻塞状态，如果使用了 epoll,select 等这样的 io 复用情况下，处于运行状态

63、请问如何设计 server，使得能够接收多个客户端的请求

参考回答：

多线程，线程池，io 复用

64、死循环+来连接时新建线程的方法效率有点低，怎么改进？

参考回答：

提前创建好一个线程池，用生产者消费者模型，创建一个任务队列，队列作为临界资源，有了新连接，就挂在到任务队列上，队列为空所有线程睡眠。改进死循环：使用 select epoll 这样的技术

65、就绪状态的进程在等待什么？

参考回答：

被调度使用 cpu 的运行权

66、请你说一下多线程的同步，锁的机制

参考回答：

同步的时候用一个互斥量，在访问共享资源前对互斥量进行加锁，在访问完成后释放互斥量上的锁。对互斥量进行加锁以后，任何其他试图再次对互斥量加锁的线程将会被阻塞直到当前线程释放该互斥锁。如果释放互斥锁时有多个线程阻塞，所有在该互斥锁上的阻塞线程都会变成可运行状态，第一个变为运行状态的线程可以对互斥量加锁，其他线程将会看到互斥锁依然被锁住，只能回去再次等待它重新变为可用。在这种方式下，每次只有一个线程可以向前执行。

67、两个进程访问临界区资源，会不会出现都获得自旋锁的情况？

参考回答：

单核 cpu，并且开了抢占可以造成这种情况。

68、假设临界区资源释放，如何保证只让一个线程获得临界区资源而不是都获得？

参考回答：

略

69、请问怎么实现线程池

参考回答：

1.设置一个生产者消费者队列，作为临界资源
2.初始化 n 个线程，并让其运行起来，加锁去队列取任务运行
3.当任务队列为空的时候，所有线程阻塞
4.当生产者队列来了一个任务后，先对队列加锁，把任务挂在到队列上，然后使用条件变量去通知阻塞中的一个线程

70、Linux 下怎么得到一个文件的 100 到 200 行

公司：京东

参考回答：

```
sed -n '100,200p' inputfile

awk 'NR>=100&&NR<=200{print}' inputfile

head -200 inputfile|tail -100
```

71、请你来说一下 awk 的使用

考察点：linux

公司：腾讯

1）作用：

样式扫描和处理语言。它允许创建简短的程序，这些程序读取输入文件、为数据排序、处理数据、对输入执行计算以及生成报表，还有无数其他的功能。

2）用法：

awk [-F field-separator] 'commands' input-file(s)

3）内置变量

ARGC	命令行参数个数
ARGV	命令行参数排列
ENVIRON	支持队列中系统环境变量的使用
FILENAME	awk 浏览的文件名
FNR	浏览文件的记录数
FS	设置输入域分隔符，等价于命令行 -F 选项
NF	浏览记录的域的个数
NR	已读的记录数
OFS	输出域分隔符

ORS	输出记录分隔符

RS	控制记录分隔符

4）实例：

1、找到当前文件夹下所有的文件和子文件夹,并显示文件大小

```
> ls -l | awk '{print $5 "\t" $9}'
```

读入有'\n'换行符分割的一条记录，然后将记录按指定的域分隔符划分域，填充域。$0 则表示所有域,$1 表示第一个域,$n 表示第 n 个域。默认域分隔符是"空白键" 或 "[tab]键"。

2、找到当前文件夹下所有的文件和子文件夹，并显示文件大小，并显示排序

```
> ls -l | awk 'BEGIN {COUNT = -1; print "BEGIN COUNT"}
 {COUNT = COUNT + 1; print COUNT"\t"$5"\t"$9}
END {print "END, COUNT = "COUNT}'
```

先处理 BEGIN， 然后进行文本分析，进行第二个{}的操作，分析完进行 END 操作。

3、找到当前文件夹下所有的子文件夹,并显示排序

```
> ls -l | awk 'BEGIN {print "BEGIN COUNT"} /4096/{print NR"\t"$5"\t"$9}
END {print "END"}'
```

* /4096/ 正则匹配式子

* 使用 print $NF 可以打印出一行中的最后一个字段，使用$(NF-1)则是打印倒数第二个字段，其他以此类推。

72、请你来说一下 linux 内核中的 Timer 定时器机制

参考回答:

1）低精度时钟

Linux 2.6.16 之前，内核只支持低精度时钟，内核定时器的工作方式：

1、系统启动后，会读取时钟源设备(RTC, HPET, PIT…)，初始化当前系统时间。

2、内核会根据 HZ(系统定时器频率，节拍率)参数值，设置时钟事件设备，启动 tick(节拍)中断。HZ 表示 1 秒种产生多少个时钟硬件中断，tick 就表示连续两个中断的间隔时间。

3、设置时钟事件设备后，时钟事件设备会定时产生一个 tick 中断，触发时钟中断处理函数，更新系统时钟,并检测 timer wheel，进行超时事件的处理。

在上面工作方式下，Linux 2.6.16 之前，内核软件定时器采用 timer wheel 多级时间轮的实现机制，维护操作系统的所有定时事件。timer wheel 的触发是基于系统 tick 周期性中断。

所以说这之前，linux 只能支持 ms 级别的时钟，随着时钟源硬件设备的精度提高和软件高精度计时的需求，有了高精度时钟的内核设计。

2）高精度时钟

Linux 2.6.16 ，内核支持了高精度的时钟，内核采用新的定时器 hrtimer，其实现逻辑和 Linux 2.6.16 之前定时器逻辑区别：

hrtimer 采用红黑树进行高精度定时器的管理，而不是时间轮；

高精度时钟定时器不在依赖系统的 tick 中断，而是基于事件触发。

旧内核的定时器实现依赖于系统定时器硬件定期的 tick，基于该 tick，内核会扫描 timer wheel 处理超时事件，会更新 jiffies，wall time(墙上时间，现实时间)，process 的使用时间等等工作。

新的内核不再会直接支持周期性的 tick，新内核定时器框架采用了基于事件触发，而不是以前的周期性触发。新内核实现了 hrtimer(high resolution timer)：于事件触发。

hrtimer 的工作原理：

通过将高精度时钟硬件的下次中断触发时间设置为红黑树中最早到期的 Timer 的时间，时钟到期后从红黑树中得到下一个 Timer 的到期时间，并设置硬件，如此循环反复。

在高精度时钟模式下，操作系统内核仍然需要周期性的 tick 中断，以便刷新内核的一些任务。hrtimer 是基于事件的，不会周期性出发 tick 中断，所以为了实现周期性的 tick 中断(dynamic tick)：系统创建了一个模拟 tick 时钟的特殊 hrtimer，将其超时时间设置为一个 tick 时长，在超时回来后，完成对应的工作，然后再次设置下一个 tick 的超时时间，以此达到周期性 tick 中断的需求。

引入了 dynamic tick,是为了能够在使用高精度时钟的同时节约能源,这样会产生 tickless 情况下，会跳过一些 tick。

新内核对相关的时间硬件设备进行了统一的封装，定义了主要有下面两个结构：

时钟源设备(closk source device)：抽象那些能够提供计时功能的系统硬件，比如 RTC(Real Time Clock)、TSC(Time Stamp Counter)，HPET，ACPI PM-Timer，PIT 等。不同时钟源提供的精度不一样，现在 pc 大都是支持高精度模式(high-resolution mode)也支持低精度模式(low-resolution mode)。

时钟事件设备(clock event device)：系统中可以触发 one-shot（单次）或者周期性中断的设备都可以作为时钟事件设备。

当前内核同时存在新旧 timer wheel 和 hrtimer 两套 timer 的实现，内核启动后会进行从低精度模式到高精度时钟模式的切换，hrtimer 模拟的 tick 中断将驱动传统的低精度定时器系统（基于时间轮）和内核进程调度。

三、计算机网络

1. 你的研究方向是无线传感器网络，请问怎么确保节点传输存储的可靠性

考察点：无线传感器网络

参考回答：

略

2. 请你说一下 TCP 怎么保证可靠性，并且简述一下 TCP 建立连接和断开连接的过程

考察点：计算机网络，TCP

参考回答：

TCP 保证可靠性：

（1）序列号、确认应答、超时重传

数据到达接收方，接收方需要发出一个确认应答，表示已经收到该数据段，并且确认序号会说明了它下一次需要接收的数据序列号。如果发送发迟迟未收到确认应答，那么可能是发送的数据丢失，也可能是确认应答丢失，这时发送方在等待一定时间后会进行重传。这个时间一般是 2*RTT(报文段往返时间）+一个偏差值。

（2）窗口控制与高速重发控制/快速重传（重复确认应答）

TCP 会利用窗口控制来提高传输速度，意思是在一个窗口大小内，不用一定要等到应答才能发送下一段数据，窗口大小就是无需等待确认而可以继续发送数据的最大值。如果不使用窗口控制，每一个没收到确认应答的数据都要重发。

使用窗口控制，如果数据段 1001-2000 丢失，后面数据每次传输，确认应答都会不停地发送序号为 1001 的应答，表示我要接收 1001 开始的数据，发送端如果收到 3 次相同应答，就会立刻进行重发；但还有种情况有可能是数据都收到了，但是有的应答丢失了，这种情况不会进行重发，因为发送端知道，如果是数据段丢失，接收端不会放过它的，会疯狂向它提醒.....

（3）拥塞控制

如果把窗口定的很大，发送端连续发送大量的数据，可能会造成网络的拥堵（大家都在用网，你在这狂发，吞吐量就那么大，当然会堵），甚至造成网络的瘫痪。所以 TCP 在为了防止这种情况而进行了拥塞控制。

慢启动：定义拥塞窗口，一开始将该窗口大小设为 1，之后每次收到确认应答（经过一个 rtt），将拥塞窗口大小*2。

拥塞避免：设置慢启动阈值，一般开始都设为 65536。拥塞避免是指当拥塞窗口大小达到这个阈值，拥塞窗口的值不再指数上升，而是加法增加（每次确认应答/每个 rtt，拥塞窗口大小+1），以此来避免拥塞。

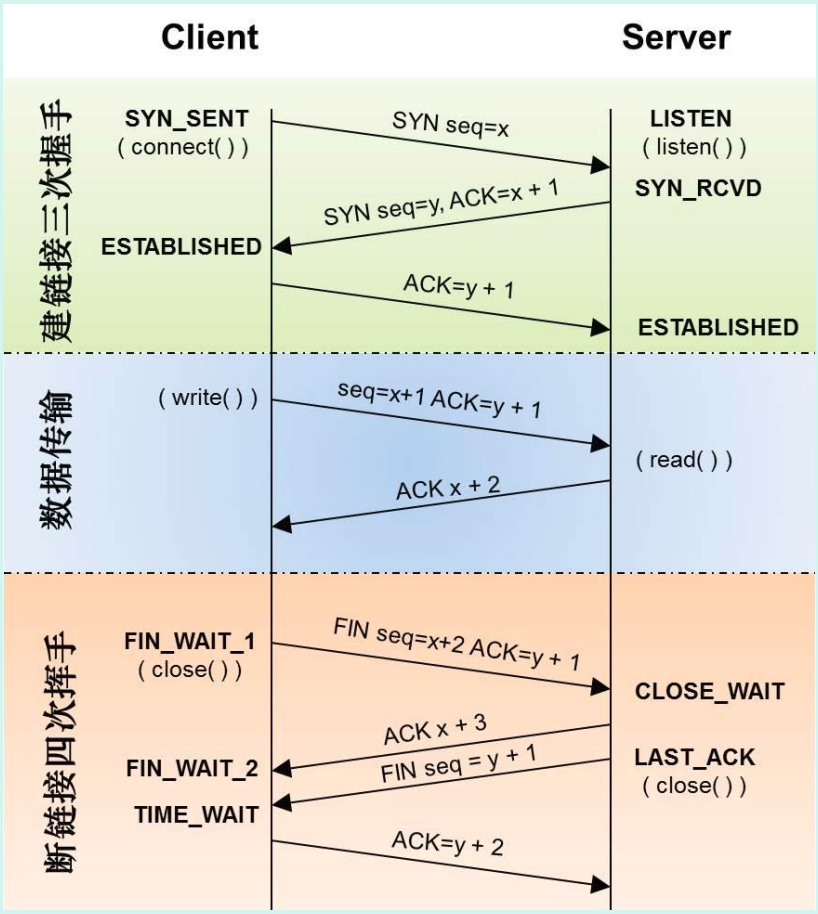
将报文段的超时重传看做拥塞，则一旦发生超时重传，我们需要先将阈值设为当前窗口大小的一半，并且将窗口大小设为初值 1，然后重新进入慢启动过程。

快速重传：在遇到 3 次重复确认应答（高速重发控制）时，代表收到了 3 个报文段，但是这之前的 1 个段丢失了，便对它进行立即重传。

然后，先将阈值设为当前窗口大小的一半，然后将拥塞窗口大小设为慢启动阈值+3 的大小。

这样可以达到：在 TCP 通信时，网络吞吐量呈现逐渐的上升，并且随着拥堵来降低吞吐量，再进入慢慢上升的过程，网络不会轻易的发生瘫痪。

TCP 建立连接和断开连接的过程：



三次握手：

1. Client 将标志位 SYN 置为 1,随机产生一个值 seq=J,并将该数据包发送给 Server,Client 进入 SYN_SENT 状态，等待 Server 确认。

2. Server 收到数据包后由标志位 SYN=1 知道 Client 请求建立连接，Server 将标志位 SYN 和 ACK 都置为 1，ack=J+1，随机产生一个值 seq=K，并将该数据包发送给 Client 以确认连接请求，Server 进入 SYN_RCVD 状态。

3. Client 收到确认后，检查 ack 是否为 J+1，ACK 是否为 1，如果正确则将标志位 ACK 置为 1，ack=K+1，并将该数据包发送给 Server，Server 检查 ack 是否为 K+1，ACK 是否为 1，如果正确则连接建立成功，Client 和 Server 进入 ESTABLISHED 状态，完成三次握手，随后 Client 与 Server 之间可以开始传输数据了。

四次挥手：

由于 TCP 连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个 FIN 来终止这一方向的连接，收到一个 FIN 只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个 TCP 连接上仍然能够发送数据，直到这一方向也发送了 FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭。

1.数据传输结束后，客户端的应用进程发出连接释放报文段，并停止发送数据，客户端进入FIN_WAIT_1状态，此时客户端依然可以接收服务器发送来的数据。

2.服务器接收到FIN后，发送一个ACK给客户端，确认序号为收到的序号+1，服务器进入CLOSE_WAIT状态。客户端收到后进入FIN_WAIT_2状态。

3.当服务器没有数据要发送时，服务器发送一个FIN报文，此时服务器进入LAST_ACK状态，等待客户端的确认

4.客户端收到服务器的FIN报文后，给服务器发送一个ACK报文，确认序列号为收到的序号+1。此时客户端进入TIME_WAIT状态，等待2MSL（MSL：报文段最大生存时间），然后关闭连接。

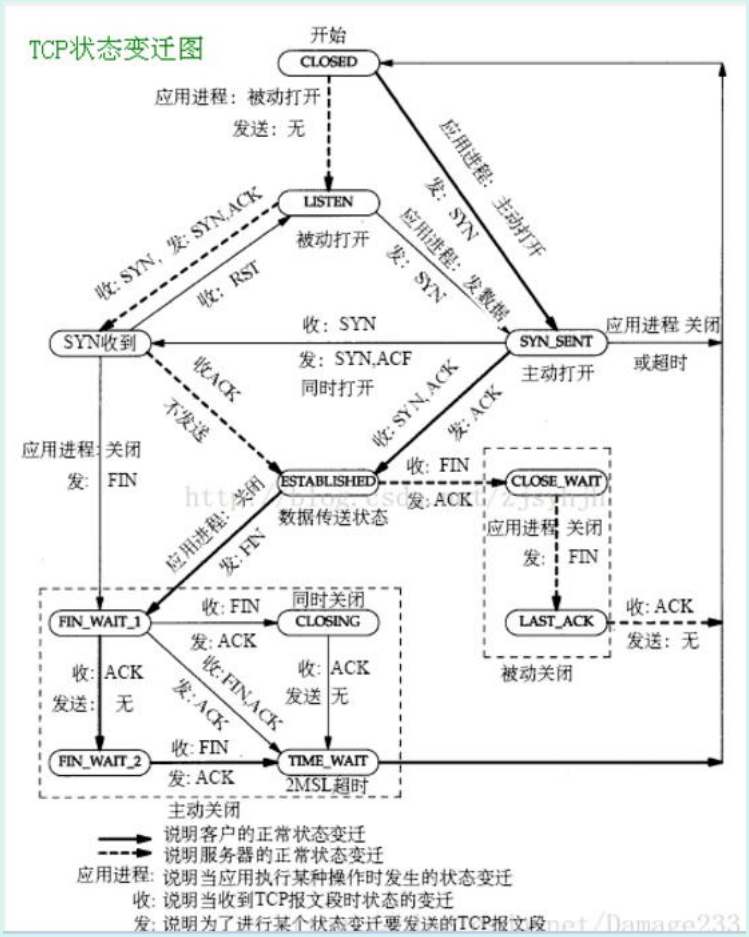
3、请你说一说TCP的模型，状态转移

考点：TCP模型 TCP状态转移

参考回答：

四层TCP/IP模型如下：

其状态转移图如下：



4、请回答一下 HTTP 和 HTTPS 的区别，以及 HTTPS 有什么缺点？

考点：HTTP HTTPS

参考回答：

HTTP 协议和 HTTPS 协议区别如下：

1）HTTP 协议是以明文的方式在网络中传输数据，而 HTTPS 协议传输的数据则是经过 TLS 加密后的，HTTPS 具有更高的安全性

2）HTTPS 在 TCP 三次握手阶段之后，还需要进行 SSL 的 handshake，协商加密使用的对称加密密钥

3）HTTPS 协议需要服务端申请证书，浏览器端安装对应的根证书

4）HTTP 协议端口是 80，HTTPS 协议端口是 443

HTTPS 优点：

HTTPS 传输数据过程中使用密钥进行加密，所以安全性更高

HTTPS 协议可以认证用户和服务器，确保数据发送到正确的用户和服务器

HTTPS 缺点：

HTTPS 握手阶段延时较高：由于在进行 HTTP 会话之前还需要进行 SSL 握手，因此 HTTPS 协议握手阶段延时增加

HTTPS 部署成本高：一方面 HTTPS 协议需要使用证书来验证自身的安全性，所以需要购买 CA 证书；另一方面由于采用 HTTPS 协议需要进行加解密的计算，占用 CPU 资源较多，需要的服务器配置或数目高

5、请你说一说 HTTP 和 HTTPS 的不同

参考回答：

HTTP 协议和 HTTPS 协议区别如下：

1）HTTP 协议是以明文的方式在网络中传输数据，而 HTTPS 协议传输的数据则是经过 TLS 加密后的，HTTPS 具有更高的安全性

2）HTTPS 在 TCP 三次握手阶段之后，还需要进行 SSL 的 handshake，协商加密使用的对称加密密钥

3）HTTPS 协议需要服务端申请证书，浏览器端安装对应的根证书

4）HTTP 协议端口是 80，HTTPS 协议端口是 443

6、请你说一说 HTTP 返回码

考点:HTTP 协议

参考回答：

HTTP 协议的响应报文由状态行、响应头部和响应包体组成，其响应状态码总体描述如下：

1xx：指示信息--表示请求已接收，继续处理。

2xx：成功--表示请求已被成功接收、理解、接受。

3xx：重定向--要完成请求必须进行更进一步的操作。

4xx：客户端错误--请求有语法错误或请求无法实现。

5xx：服务器端错误--服务器未能实现合法的请求。

常见状态代码、状态描述的详细说明如下。

200 OK：客户端请求成功。

206 partial content 服务器已经正确处理部分 GET 请求，实现断点续传或同时分片下载，该请求必须包含 Range 请求头来指示客户端期望得到的范围

300 multiple choices（可选重定向）:被请求的资源有一系列可供选择的反馈信息，由浏览器/用户自行选择其中一个。

301 moved permanently（永久重定向）：该资源已被永久移动到新位置，将来任何对该资源的访问都要使用本响应返回的若干个 URI 之一。

302 move temporarily(临时重定向)：请求的资源现在临时从不同的 URI 中获得，

304：not modified :如果客户端发送一个待条件的 GET 请求并且该请求以经被允许，而文档内容未被改变，则返回 304,该响应不包含包体（即可直接使用缓存）。

403 Forbidden：服务器收到请求，但是拒绝提供服务。

t Found：请求资源不存在，举个例子：输入了错误的 URL。

7、请你说一说 IP 地址作用，以及 MAC 地址作用

考点：MAC 协议 IP 协议

参考回答：

MAC 地址是一个硬件地址，用来定义网络设备的位置，主要由数据链路层负责。而 IP 地址是 IP 协议提供的一种统一的地址格式，为互联网上的每一个网络和每一台主机分配一个逻辑地址，以此来屏蔽物理地址的差异。

8、请介绍一下操作系统中的中断

考点：操作系统 系统中断

参考回答：

中断是指 CPU 对系统发生的某个事件做出的一种反应，CPU 暂停正在执行的程序，保存现场后自动去执行相应的处理程序，处理完该事件后再返回中断处继续执行原来的程序。中断一般三类，一种是由 CPU 外部引起的，如 I/O 中断、时钟中断，一种是来自 CPU 内部事件或程序执行中引起的中断，例如程序非法操作，地址越界、浮点溢出），最后一种是在程序中使用了系统调用引起的。而中断处理一般分为中断响应和中断处理两个步骤，中断响应由硬件实施，中断处理主要由软件实施。

9、请回答 OSI 七层模型和 TCP/IP 四层模型，每层列举 2 个协议

考点：网络模型 协议

参考回答：

OSI 七层模型及其包含的协议如下:

物理层: 通过媒介传输比特,确定机械及电气规范,传输单位为 bit，主要包括的协议为: IEE802.3 CLOCK RJ45

数据链路层: 将比特组装成帧和点到点的传递,传输单位为帧,主要包括的协议为 MAC VLAN PPP

网络层：负责数据包从源到宿的传递和网际互连，传输单位为包,主要包括的协议为 IP ARP ICMP

传输层：提供端到端的可靠报文传递和错误恢复，传输单位为报文,主要包括的协议为 TCP UDP

会话层：建立、管理和终止会话，传输单位为 SPDU，主要包括的协议为 RPC NFS

表示层: 对数据进行翻译、加密和压缩,传输单位为 PPDU，主要包括的协议为 JPEG ASII

应用层: 允许访问 OSI 环境的手段,传输单位为 APDU，主要包括的协议为 FTP HTTP DNS

TCP/IP 4 层模型包括：

网络接口层：MAC VLAN

网络层:IP ARP ICMP

传输层:TCP UDP

应用层:HTTP DNS SMTP

10、请你说一说 TCP 的三次握手和四次挥手的过程及原因

考点：TCP 协议

参考回答：

TCP 的三次握手过程如下：

C-> SYN -> S

S->SYN/ACK->C

C->ACK->S

三次握手的原因：三次握手可以防止已经失效的连接请求报文突然又传输到服务器端导致的服务器资源浪费。例如，客户端先发送了一个 SYN，但是由于网络阻塞，该 SYN 数据包在某个节点长期滞留。然后客户端又重传 SYN 数据包并正确建立 TCP 连接，然后传输完数据后关闭该连接。该连接释放后失效的 SYN 数据包才到达服务器端。在二次握手的前提下，服务器端会认为这是客户端发起的又一次请求，然后发送 SYN ，并且在服务器端创建 socket 套接字，一直等待客户端发送数据。但是由于客户端并没有发起新的请求，所以会丢弃服务端的 SYN 。此时服务器会一直等待客户端发送数据从而造成资源浪费。

TCP 的四次挥手过程如下：

C->FIN->S

S->ACK->C

S->FIN->C

C->ACK->S

四次挥手的原因：由于连接的关闭控制权在应用层，所以被动关闭的一方在接收到 FIN 包时，TCP 协议栈会直接发送一个 ACK 确认包，优先关闭一端的通信。然后通知应用层，由应用层决定什么时候发送 FIN 包。应用层可以使用系统调用函数 read==0 来判断对端是否关闭连接。

11、搜索 baidu，会用到计算机网络中的什么层？每层是干什么的

考察点：网络

浏览器中输入 URL

浏览器要将 URL 解析为 IP 地址，解析域名就要用到 DNS 协议，首先主机会查询 DNS 的缓存，如果没有就给本地 DNS 发送查询请求。DNS 查询分为两种方式，一种是递归查询，一种是迭代查询。如果是迭代查询，本地的 DNS 服务器，向根域名服务器发送查询请求，根域名服务器告知该域名的一级域名服务器，然后本地服务器给该一级域名服务器发送查询请求，然后依次类推直到查询到该域名的 IP 地址。DNS 服务器是基于 UDP 的，因此会用到 UDP 协议。

得到 IP 地址后，浏览器就要与服务器建立一个 http 连接。因此要用到 http 协议，http 协议报文格式上面已经提到。http 生成一个 get 请求报文，将该报文传给 TCP 层处理，所以还会用到 TCP 协议。如果采用 https 还会使用 https 协议先对 http 数据进行加密。TCP 层如果有需要先将 HTTP 数据包分片，分片依据路径 MTU 和 MSS。TCP 的数据包然后会发送给 IP 层，用到 IP 协议。IP 层通过路由选路，一跳一跳发送到目的地址。当然在一个网段内的寻址是通过以太网协议实现(也可以是其他物理层协议，比如 PPP，SLIP)，以太网协议需要直到目的 IP 地址的物理地址，有需要 ARP 协议。

其中：

1、DNS 协议，http 协议，https 协议属于应用层

应用层是体系结构中的最高层。应用层确定进程之间通信的性质以满足用户的需要。这里的进程就是指正在运行的程序。应用层不仅要提供应用进程所需要的信息交换和远地操作，而且还要作为互相作用的应用进程的用户代理，来完成一些为进行语义上有意义的信息交换所必须的功能。应用层直接为用户的应用进程提供服务。

2、TCP/UDP 属于传输层

传输层的任务就是负责主机中两个进程之间的通信。因特网的传输层可使用两种不同协议：即面向连接的传输控制协议 TCP，和无连接的用户数据报协议 UDP。面向连接的服务能够提供可靠的交付，但无连接服务则不保证提供可靠的交付，它只是“尽最大努力交付”。这两种服务方式都很有用，备有其优缺点。在分组交换网内的各个交换结点机都没有传输层。

3、IP 协议，ARP 协议属于网络层

网络层负责为分组交换网上的不同主机提供通信。在发送数据时，网络层将运输层产生的报文段或用户数据报封装成分组或包进行传送。在 TCP/IP 体系中，分组也叫作 IP 数据报，或简称为数据报。网络层的另一个任务就是要选择合适的路由，使源主机运输层所传下来的分组能够交付到目的主机。

4、数据链路层

当发送数据时，数据链路层的任务是将在网络层交下来的 IP 数据报组装成帧，在两个相邻结点间的链路上传送以帧为单位的数据。每一帧包括数据和必要的控制信息（如同步信息、地址信息、差错控制、以及流量控制信息等）。控制信息使接收端能够知道一个帧从哪个比特开始和到哪个比特结束。控制信息还使接收端能够检测到所收到的帧中有无差错。

5、物理层

物理层的任务就是透明地传送比特流。在物理层上所传数据的单位是比特。传递信息所利用的一些物理媒体，如双绞线、同轴电缆、光缆等，并不在物理层之内而是在物理层的下面。因此也有人把物理媒体当做第 0 层。

12、请你说一说 TCP 拥塞控制？以及达到什么情况的时候开始减慢增长的速度？

考察点：网络

参考回答：

拥塞控制是防止过多的数据注入网络，使得网络中的路由器或者链路过载。流量控制是点对点的通信量控制，而拥塞控制是全局的网络流量整体性的控制。发送双方都有一个拥塞窗口——cwnd。

1、慢开始

最开始发送方的拥塞窗口为 1，由小到大逐渐增大发送窗口和拥塞窗口。每经过一个传输轮次，拥塞窗口 cwnd 加倍。当 cwnd 超过慢开始门限，则使用拥塞避免算法，避免 cwnd 增长过大。

2、拥塞避免

每经过一个往返时间 RTT，cwnd 就增长 1。

在慢开始和拥塞避免的过程中，一旦发现网络拥塞，就把慢开始门限设为当前值的一半，并且重新设置 cwnd 为 1，重新慢启动。（乘法减小，加法增大）

3、快重传

接收方每次收到一个失序的报文段后就立即发出重复确认，发送方只要连续收到三个重复确认就立即重传（尽早重传未被确认的报文段）。

4、快恢复

当发送方连续收到了三个重复确认，就乘法减半（慢开始门限减半），将当前的 cwnd 设置为慢开始门限，并且采用拥塞避免算法（连续收到了三个重复请求，说明当前网络可能没有拥塞）。

采用快恢复算法时，慢开始只在建立连接和网络超时才使用。

达到什么情况的时候开始减慢增长的速度？

采用慢开始和拥塞避免算法的时候

1. 一旦 cwnd>慢开始门限，就采用拥塞避免算法，减慢增长速度

2. 一旦出现丢包的情况，就重新进行慢开始，减慢增长速度

采用快恢复和快重传算法的时候

1. 一旦 cwnd>慢开始门限，就采用拥塞避免算法，减慢增长速度

2. 一旦发送方连续收到了三个重复确认，就采用拥塞避免算法，减慢增长速度

13、请问 TCP 用了哪些措施保证其可靠性

考察点：计算机网络，TCP

参考回答：

1、序列号、确认应答、超时重传

数据到达接收方，接收方需要发出一个确认应答，表示已经收到该数据段，并且确认序号会说明了它下一次需要接收的数据序列号。如果发送发迟迟未收到确认应答，那么可能是发送的数据丢失，也可能是确认应答丢失，这时发送方在等待一定时间后会进行重传。这个时间一般是 2*RTT(报文段往返时间）+一个偏差值。

2、窗口控制与高速重发控制/快速重传（重复确认应答）

TCP 会利用窗口控制来提高传输速度，意思是在一个窗口大小内，不用一定要等到应答才能发送下一段数据，窗口大小就是无需等待确认而可以继续发送数据的最大值。如果不使用窗口控制，每一个没收到确认应答的数据都要重发。

使用窗口控制，如果数据段 1001-2000 丢失，后面数据每次传输，确认应答都会不停地发送序号为 1001 的应答，表示我要接收 1001 开始的数据，发送端如果收到 3 次相同应答，就会立刻进行重发；但还有种情况有可能是数据都收到了，但是有的应答丢失了，这种情况不会进行重发，因为发送端知道，如果是数据段丢失，接收端不会放过它的，会疯狂向它提醒.....

3、拥塞控制

如果把窗口定的很大，发送端连续发送大量的数据，可能会造成网络的拥堵（大家都在用网，你在这狂发，吞吐量就那么大，当然会堵），甚至造成网络的瘫痪。所以 TCP 在为了防止这种情况而进行了拥塞控制。

慢启动：定义拥塞窗口，一开始将该窗口大小设为 1，之后每次收到确认应答（经过一个 rtt），将拥塞窗口大小*2。

拥塞避免：设置慢启动阈值，一般开始都设为 65536。拥塞避免是指当拥塞窗口大小达到这个阈值，拥塞窗口的值不再指数上升，而是加法增加（每次确认应答/每个 rtt，拥塞窗口大小+1），以此来避免拥塞。

将报文段的超时重传看做拥塞，则一旦发生超时重传，我们需要先将阈值设为当前窗口大小的一半，并且将窗口大小设为初值 1，然后重新进入慢启动过程。

快速重传：在遇到 3 次重复确认应答（高速重发控制）时，代表收到了 3 个报文段，但是这之前的 1 个段丢失了，便对它进行立即重传。

然后，先将阈值设为当前窗口大小的一半，然后将拥塞窗口大小设为慢启动阈值+3 的大小。

这样可以达到：在 TCP 通信时，网络吞吐量呈现逐渐的上升，并且随着拥堵来降低吞吐量，再进入慢慢上升的过程，网络不会轻易的发生瘫痪。

TCP 三次握手

深信服

TCP 的三次握手过程如下：

1）首先客户端发送 seq=c 的 SYN 数据包

服务器端响应一个 seq=s,ack=c+1 的 SYN+ACK 数据包

最后客户端回复一个 seq=c+1,ack=s+1 的 ACK 数据包，三次握手完成

14、请你说说 TCP/IP 数据链路层的交互过程

参考回答：

网络层等到数据链层用 mac 地址作为通信目标,数据包到达网络等准备往数据链层发送的时候，首先会去自己的 arp 缓存表(存着 ip-mac 对应关系)去查找改目标 ip 的 mac 地址，如果查到了，就讲目标 ip 的 mac 地址封装到链路层数据包的包头。如果缓存中没有找到，会发起一个广播：who is ip XXX tell ip XXX,所有收到的广播的机器看这个 ip 是不是自己的，如果是自己的，则以单拨的形式将自己的 mac 地址回复给请求的机器

15、请你说说传递到 IP 层怎么知道报文该给哪个应用程序，它怎么区分 UDP 报文还是 TCP 报文

参考回答：

根据端口区分；

看 ip 头中的协议标识字段，17 是 udp，6 是 tcp

16、请问你有没有基于做过 socket 的开发？具体网络层的操作该怎么做？（其实就是网络编程的基本步骤）

参考回答：

服务端：socket-bind-listen-accept
客户端：socket-connect

17、请问 server 端监听端口，但还没有客户端连接进来，此时进程处于什么状态？

参考回答：

这个需要看服务端的编程模型，如果如上一个问题的回答描述的这样，则处于阻塞状态，如果使用了 epoll,select 等这样的 io 复用情况下，处于运行状态

18、请问 TCP 三次握手是怎样的？

参考回答：

1.客户端发送 syn0 给服务器
2.服务器收到 syn0，回复 syn1,ack(syn0+1)
3.客户端收到 syn1，回复 ack(syn1+1)

19、请问 tcp 握手为什么两次不可以？为什么不用四次？

参考回答：

两次不可以：tcp 是全双工通信，两次握手只能确定单向数据链路是可以通信的，并不能保证反向的通信正常
不用四次：
本来握手应该和挥手一样都是需要确认两个方向都能联通的，本来模型应该是：
1.客户端发送 syn0 给服务器
2.服务器收到 syn0，回复 ack(syn0+1)
3.服务器发送 syn1
4.客户端收到 syn1，回复 ack(syn1+1)
因为 tcp 是全双工的，上边的四部确认了数据在两个方向上都是可以正确到达的，但是 2，3 步没有没有上下的联系，可以将其合并，加快握手效率，所有就变成了 3 步握手。

22、请你来说一下 TCP 拥塞控制？

参考回答：

发送方维持一个叫做拥塞窗口 cwnd（congestion window）的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于拥塞窗口，另外考虑到接受方的接收能力，发送窗口可能小于拥塞窗口。慢开始算法的思路就是，不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。
过程 cwnd 的大小呈指数增长，直到超过慢启动门限，然后进入拥塞避免阶段，cwnd 的大小线性增长，当出现网络拥塞(三个重复的 ack 或者超时)时候，将慢启动门限设置为出现拥塞时候大小的一半，cwnd 的大小重新从 0 开始进入慢启动阶段。
快重传和快恢复：快重传要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时捎带确认。快重传算法规定，发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期

20、TCP 和 UDP 的区别和各自适用的场景

考察点：网络

参考回答：

1）TCP 和 UDP 区别

1） 连接

TCP 是面向连接的传输层协议，即传输数据之前必须先建立好连接。

UDP 无连接。

2） 服务对象

TCP 是点对点的两点间服务，即一条 TCP 连接只能有两个端点；

UDP 支持一对一，一对多，多对一，多对多的交互通信。

3） 可靠性

TCP 是可靠交付：无差错，不丢失，不重复，按序到达。

UDP 是尽最大努力交付，不保证可靠交付。

4）拥塞控制，流量控制

TCP 有拥塞控制和流量控制保证数据传输的安全性。

UDP 没有拥塞控制，网络拥塞不会影响源主机的发送效率。

5） 报文长度

TCP 是动态报文长度，即 TCP 报文长度是根据接收方的窗口大小和当前网络拥塞情况决定的。

UDP 面向报文，不合并，不拆分，保留上面传下来报文的边界。

6) 首部开销

TCP 首部开销大，首部 20 个字节。

UDP 首部开销小，8 字节。（源端口，目的端口，数据长度，校验和）

2）TCP 和 UDP 适用场景

从特点上我们已经知道，TCP 是可靠的但传输速度慢，UDP 是不可靠的但传输速度快。因此在选用具体协议通信时，应该根据通信数据的要求而决定。

若通信数据完整性需让位与通信实时性，则应该选用 TCP 协议（如文件传输、重要状态的更新等）；反之，则使用 UDP 协议（如视频传输、实时通信等）。

21、请你来说一下 TCP 三次握手四次挥手的过程，为什么 tcp 连接握手需要三次, time_wait 状态

考察点：TCP，三次握手，四次挥手，time_wait

公司：腾讯

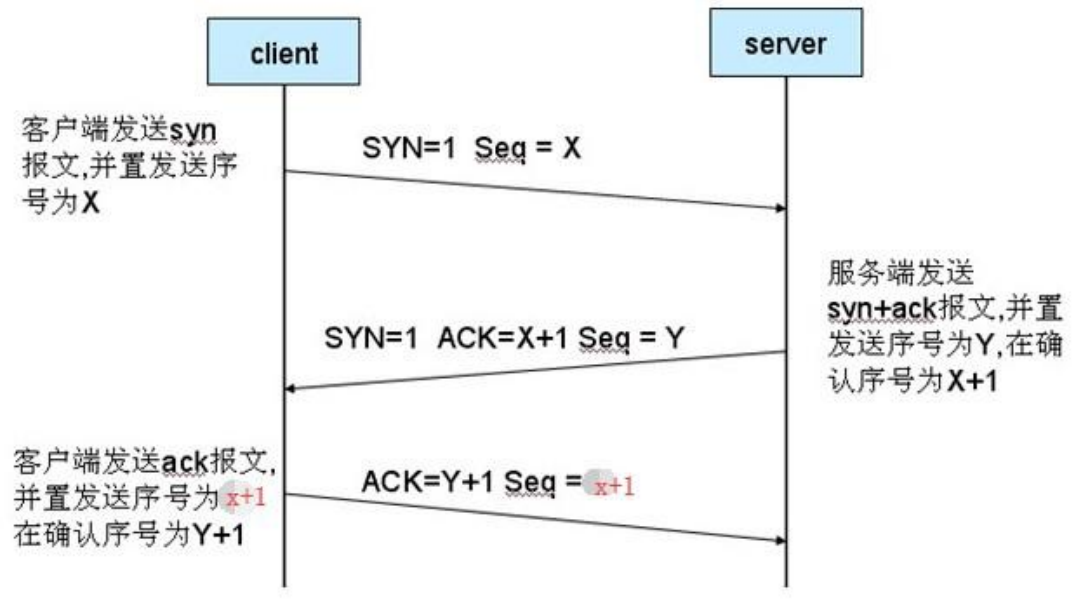
1）TCP 连接（三次握手）过程：

客户端 A：发送 SYN 连接报文，序列号为 x，进入 SYNC-SENT 状态。

服务端 B：发送 SYN 连接确认报文（SYN=1，ACK = 1），序列号为 y（seq = y），确认报文 x（ack = x + 1），进入 SYNC-RCVD 状态。

客户端 A：发送 ACK 确认报文（ACK = 1），序列号为 x+1（seq = x + 1），确认报文 y+1（ack = y + 1），进入 ESTABLISHED 状态。

服务器 B：收到后进入 ESTABLISHED 状态。



2）三次握手原因：

三次握手是为了防止，客户端的请求报文在网络滞留，客户端超时重传了请求报文，服务端建立连接，传输数据，释放连接之后，服务器又收到了客户端滞留的请求报文，建立连接一直等待客户端发送数据。

服务器对客户端的请求进行回应(第二次握手)后，就会理所当然的认为连接已建立，而如果客户端并没有收到服务器的回应呢？此时，客户端仍认为连接未建立，服务器会对已建立的连接保存必要的资源，如果大量的这种情况，服务器会崩溃。

3） TCP 释放（四次分手）过程：

服务端 A：发送 FIN 报文（FIN = 1），序列号为 u（seq = u），进入 FIN-WAIT 1 状态。

客户端 B：发送 ACK 确认报文（ACK = 1），序列号为 v（seq = v），确认报文 u（ack = u + 1），进入 CLOSE-WAIT 状态，继续传送数据。

服务端 A：收到上述报文进入 FIN-WAIT2 状态，继续接受 B 传输的数据。

客户端 B：数据传输完毕后，发送 FIN 报文（FIN = 1，ACK = 1），序列号为 w（seq = w），确认报文 u（ack = u + 1），进入 LAST-ACK 状态。

服务端 A：发送 ACK 确认报文（ACK = 1），序列号为 u+1（seq = u + 1），确认报文 w（ack = w + 1），进入 TIME-WAIT 状态，等待 2MSL（最长报文段寿命），进入 CLOSED 状态。

客户端 B：收到后上述报文后进入 CLOSED 状态。

4）为什么 TCP 协议终止链接要四次？

1、当客户端确认发送完数据且知道服务器已经接收完了，想要关闭发送数据口（当然确认信号还是可以发），就会发 FIN 给服务器。

2、服务器收到客户端发送的 FIN，表示收到了，就会发送 ACK 回复。

3、但这时候服务器可能还在发送数据，没有想要关闭数据口的意思，所以服务器的 FIN 与 ACK 不是同时发送的，而是等到服务器数据发送完了，才会发送 FIN 给客户端。

4、客户端收到服务器发来的 FIN，知道服务器的数据也发送完了，回复 ACK， 客户端等待 2MSL 以后，没有收到服务器传来的任何消息，知道服务器已经收到自己的 ACK 了，客户端就关闭链接，服务器也关闭链接了。

5）2MSL 意义：

1、保证最后一次握手报文能到 B，能进行超时重传。

2、2MSL 后，这次连接的所有报文都会消失，不会影响下一次连接。

22、请你来说一说 http 协议

考察点：网络

参考回答：

1）HTTP 协议：

HTTP 协议是 Hyper Text Transfer Protocol（超文本传输协议）的缩写，是用于从万维网（WWW:World Wide Web）服务器传输超文本到本地浏览器的传送协议。

HTTP 是一个基于 TCP/IP 通信协议来传递数据（HTML 文件，图片文件，查询结果等）。

HTTP 是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于 1990 年提出，经过几年的使用与发展，得到不断地完善和扩展。目前在 WWW 中使用的是 HTTP/1.0 的第六版，HTTP/1.1 的规范化工作正在进行之中，而且 HTTP-NG（Next Generation of HTTP）的建议已经提出。

HTTP 协议工作于客户端-服务端架构为上。浏览器作为 HTTP 客户端通过 URL 向 HTTP 服务端即 WEB 服务器发送所有请求。Web 服务器根据接收到的请求后，向客户端发送响应信息。

2）HTTP 协议特点

1、简单快速：

客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有 GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。

2、灵活：

HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。

3、无连接：

无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

4、无状态：

HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

5、支持 B/S 及 C/S 模式。

6、默认端口 80

7、基于 TCP 协议

3）HTTP 过程概述：

HTTP 协议定义 Web 客户端如何从 Web 服务器请求 Web 页面，以及服务器如何把 Web 页面传送给客户端。HTTP 协议采用了请求/响应模型。客户端向服务器发送一个请求报文，请求报文包含请求的方法、URL、协议版本、请求头部和请求数据。服务器以一个状态行作为响应，响应的内容包括协议的版本、成功或者错误代码、服务器信息、响应头部和响应数据。

HTTP 请求/响应的步骤如下：

1、客户端连接到 Web 服务器

一个 HTTP 客户端，通常是浏览器，与 Web 服务器的 HTTP 端口（默认为 80）建立一个 TCP 套接字连接。例如，http://www.baidu.com。

2、发送 HTTP 请求

通过 TCP 套接字，客户端向 Web 服务器发送一个文本的请求报文，一个请求报文由请求行、请求头部、空行和请求数据 4 部分组成。

3、服务器接受请求并返回 HTTP 响应

Web 服务器解析请求，定位请求资源。服务器将资源复本写到 TCP 套接字，由客户端读取。一个响应由状态行、响应头部、空行和响应数据 4 部分组成。

4、释放连接 TCP 连接

若 connection 模式为 close，则服务器主动关闭 TCP 连接，客户端被动关闭连接，释放 TCP 连接;若 connection 模式为 keepalive，则该连接会保持一段时间，在该时间内可以继续接收请求;

5、客户端浏览器解析 HTML 内容

客户端浏览器首先解析状态行，查看表明请求是否成功的状态代码。然后解析每一个响应头，响应头告知以下为若干字节的 HTML 文档和文档的字符集。客户端浏览器读取响应数据 HTML，根据 HTML 的语法对其进行格式化，并在浏览器窗口中显示。

4、举例：

在浏览器地址栏键入 URL，按下回车之后会经历以下流程：

1、浏览器向 DNS 服务器请求解析该 URL 中的域名所对应的 IP 地址;

2、解析出 IP 地址后，根据该 IP 地址和默认端口 80，和服务器建立 TCP 连接;

3、浏览器发出读取文件（URL 中域名后面部分对应的文件）的 HTTP 请求，该请求报文作为 TCP 三次握手的第三个报文的数据发送给服务器;

4、服务器对浏览器请求作出响应，并把对应的 html 文本发送给浏览器;

5、释放 TCP 连接;

6、浏览器将该 html 文本并显示内容;

23、请你来说一下 GET 和 POST 的区别

参考回答：

1、概括

对于 GET 方式的请求，浏览器会把 http header 和 data 一并发送出去，服务器响应 200（返回数据）；

而对于 POST，浏览器先发送 header，服务器响应 100 continue，浏览器再发送 data，服务器响应 200 ok（返回数据）

2、区别：

1、get 参数通过 url 传递，post 放在 request body 中。

2、get 请求在 url 中传递的参数是有长度限制的，而 post 没有。

3、get 比 post 更不安全，因为参数直接暴露在 url 中，所以不能用来传递敏感信息。

4、get 请求只能进行 url 编码，而 post 支持多种编码方式。

5、get 请求会浏览器主动 cache，而 post 支持多种编码方式。

6、get 请求参数会被完整保留在浏览历史记录里，而 post 中的参数不会被保留。

7、GET 和 POST 本质上就是 TCP 链接，并无差别。但是由于 HTTP 的规定和浏览器/服务器的限制，导致他们在应用过程中体现出一些不同。

8、GET 产生一个 TCP 数据包；POST 产生两个 TCP 数据包。

24、请你来说一下 socket 编程中服务器端和客户端主要用到哪些函数

考察点：socket 网络编程

1）基于 TCP 的 socket：

1、服务器端程序：

1 创建一个 socket，用函数 socket()

2 绑定 IP 地址、端口等信息到 socket 上，用函数 bind()

3 设置允许的最大连接数，用函数 listen()

4 接收客户端上来的连接，用函数 accept()

5 收发数据，用函数 send()和 recv()，或者 read()和 write()

6 关闭网络连接

2、客户端程序：

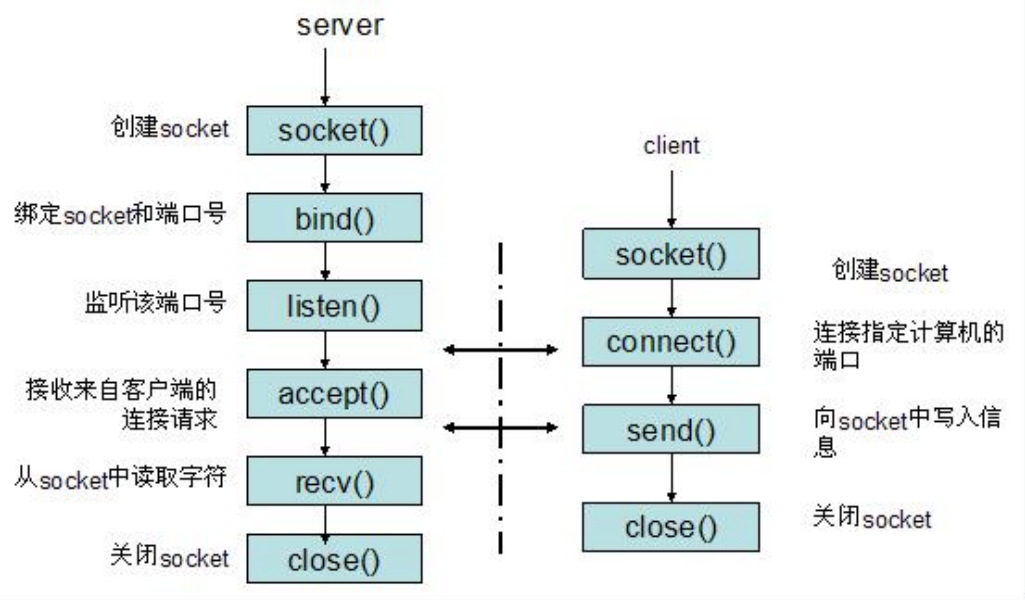
1 创建一个 socket，用函数 socket()

2 设置要连接的对方的 IP 地址和端口等属性

3 连接服务器，用函数 connect()

4 收发数据，用函数 send()和 recv()，或 read()和 write()

5 关闭网络连接



2）基于 UDP 的 socket：

1、服务器端流程

1 建立套接字文件描述符，使用函数 socket()，生成套接字文件描述符。

2 设置服务器地址和侦听端口，初始化要绑定的网络地址结构。

3 绑定侦听端口，使用 bind()函数，将套接字文件描述符和一个地址类型变量进行绑定。

4 接收客户端的数据，使用 recvfrom()函数接收客户端的网络数据。

5 向客户端发送数据，使用 sendto()函数向服务器主机发送数据。

6 关闭套接字，使用 close()函数释放资源。UDP 协议的客户端流程

2、客户端流程

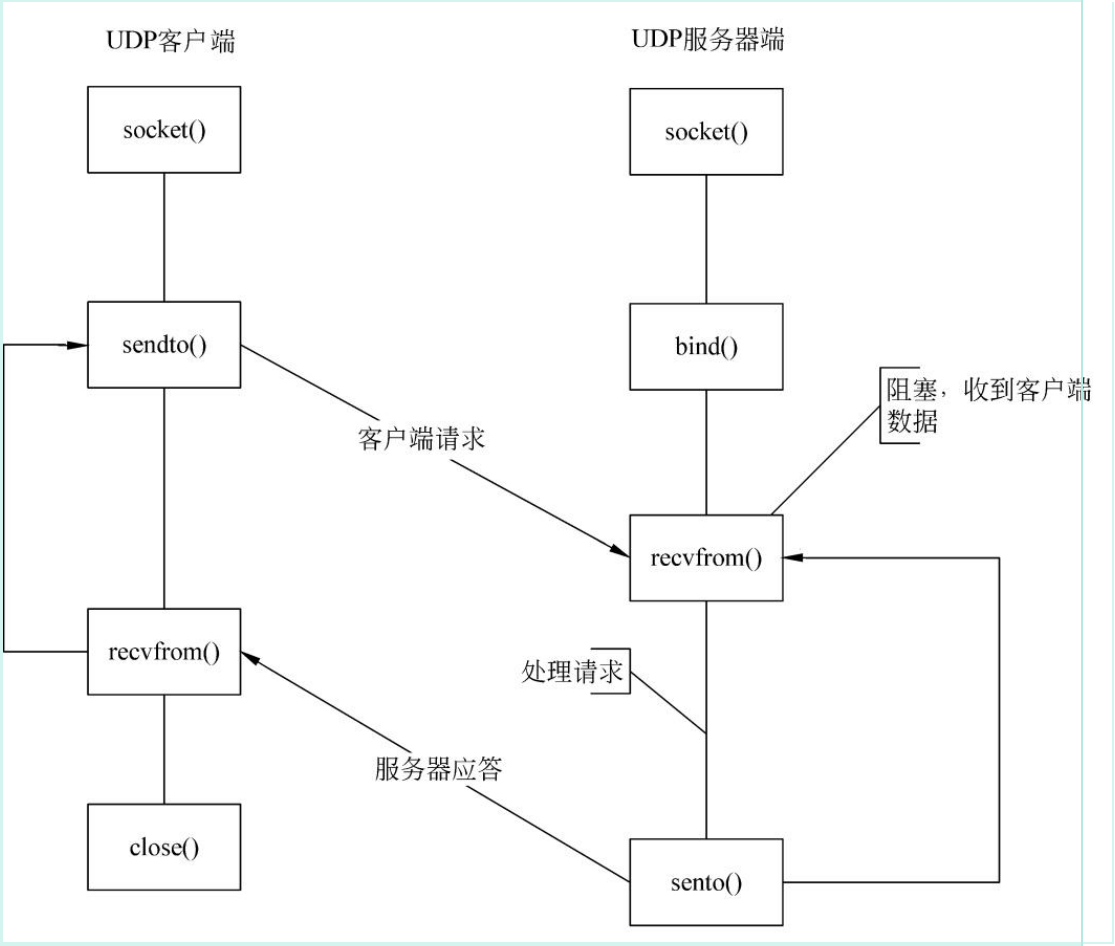
1 建立套接字文件描述符，socket()。

2 设置服务器地址和端口，struct sockaddr。

3 向服务器发送数据，sendto()。

4 接收服务器的数据，recvfrom()。

5 关闭套接字，close()。



25、请你来说一下数字证书是什么，里面都包含那些内容

考察点：密码学，数字证书

1）概念：

数字证书是数字证书在一个身份和该身份的持有者所拥有的公/私钥对之间建立了一种联系，由认证中心（CA）或者认证中心的下级认证中心颁发的。根证书是认证中心与用户建立信任关系的基础。在用户使用数字证书之前必须首先下载和安装。

认证中心是一家能向用户签发数字证书以确认用户身份的管理机构。为了防止数字凭证的伪造，认证中心的公共密钥必须是可靠的，认证中心必须公布其公共密钥或由更高级别的认证中心提供一个电子凭证来证明其公共密钥的有效性，后一种方法导致了多级别认证中心的出现。

2）数字证书颁发过程：

数字证书颁发过程如下：用户产生了自己的密钥对，并将公共密钥及部分个人身份信息传送给一家认证中心。认证中心在核实身份后，将执行一些必要的步骤，以确信请求确实由用户发送而来，然后，认证中心将发给用户一个数字证书，该证书内附了用户和他的密钥等信息，同时还附有对认证中心公共密钥加以确认的数字证书。当用户想证明其公开密钥的合法性时，就可以提供这一数字证书。

3）内容：

数字证书的格式普遍采用的是 X.509V3 国际标准，一个标准的 X.509 数字证书包含以下一些内容：

1、证书的版本信息；

2、证书的序列号，每个证书都有一个唯一的证书序列号；

3、证书所使用的签名算法；

4、证书的发行机构名称，命名规则一般采用 X.500 格式；

5、证书的有效期，通用的证书一般采用 UTC 时间格式；

6、证书所有人的名称，命名规则一般采用 X.500 格式；

7、证书所有人的公开密钥；

8、证书发行者对证书的签名。

26、请你来介绍一下 UDP 的 connect 函数

考察点：UDP

参考回答：

除非套接字已连接，否则异步错误是不会反悔到 UDP 套接字的。我们确实可以给 UDP 套接字调用 connect，然而这样做的结果却与 TCP 连接不同的是没有三路握手过程。内核只是检查是否存在立即可知的错误，记录对端的 IP 地址和端口号，然后立即返回调用进程。

对于已连接 UDP 套接字，与默认的未连接 UDP 套接字相比，发生了三个变化。

其实一旦 UDP 套接字调用了 connect 系统调用，那么这个 UDP 上的连接就变成一对一的连接，但是通过这个 UDP 连接传输数据的性质还是不变的，仍然是不可靠的 UDP 连接。一旦变成一对一的连接，在调用系统调用发送和接受数据时也就可以使用 TCP 那一套系统调用了。

1、我们再也不能给输出操作指定目的 IP 地址和端口号。也就是说，我们不使用 sendto，而改用 write 或 send。写到已连接 UDP 套接字上的任何内容都自动发送到由 connect 指定的协

议地址。可以给已连接的 UDP 套接字调用 sendto，但是不能指定目的地址。sendto 的第五个参数必须为空指针，第六个参数应该为 0.

2、不必使用 recvfrom 以获悉数据报的发送者，而改用 read、recv 或 recvmsg。在一个已连接 UDP 套接字上，由内核为输入操作返回的数据报只有那些来自 connect 指定协议地址的数据报。这样就限制一个已连接 UDP 套接字能且仅能与一个对端交换数据报。

3、由已连接 UDP 套接字引发的异步错误会返回给它们所在的进程，而未连接的 UDP 套接字不接收任何异步错误。

来自任何其他 IP 地址或断开的数据报不投递给这个已连接套接字，因为它们要么源 IP 地址要么源 UDP 端口不与该套接字 connect 到的协议地址相匹配。

UDP 客户进程或服务器进程只在使用自己的 UDP 套接字与确定的唯一对端进行通信时，才可以调用 connect。调用 connect 的通常是 UDP 客户，不过有些网络应用中的 UDP 服务器会与单个客户长时间通信 TFTP，这种情况下，客户和服务器都可能调用 connect。

27、请你讲述一下 TCP 三次握手，四次挥手，以及为什么用三次握手？

参考回答：

三次握手
1.客户端发送 syn0 给服务器
2.服务器收到 syn0，回复 syn1,ack(syn0+1)
3.客户端收到 syn1，回复 ack(syn1+1)
四次挥手(这里以客户端主动断开为例)
1.客户端发送 fin
2.服务端收到 fin,回复 ack,然后服务器去处理其他事
3.服务器事情处理完，回复 fin
4.客户端回复 ack
为什么用三次握手
本来握手应该和挥手一样都是需要确认两个方向都能联通的，本来模型应该是：
1.客户端发送 syn0 给服务器
2.服务器收到 syn0，回复 ack(syn0+1)
3.服务器发送 syn1
3.客户端收到 syn1，回复 ack(syn1+1)
因为 tcp 是全双工的，上边的四部确认了数据在两个方向上都是可以正确到达的，但是 2，3 步没有没有上下的联系，可以将其合并，加快握手效率，所有就变成了 3 步握手。

28、请你说一下阻塞，非阻塞，同步，异步

参考回答：

阻塞和非阻塞：调用者在事件没有发生的时候，一直在等待事件发生，不能去处理别的任务这是阻塞。调用者在事件没有发生的时候，可以去处理别的任务这是非阻塞。

同步和异步：调用者必须循环自去查看事件有没有发生，这种情况是同步。调用者不用自己去查看事件有没有发生，而是等待着注册在事件上的回调函数通知自己，这种情况是异步。

29、请你讲述一下 Socket 编程的 send() recv() accept() socket()函数？

参考回答：

send 函数用来向 TCP 连接的另一端发送数据。客户程序一般用 send 函数向服务器发送请求，而服务器则通常用 send 函数来向客户程序发送应答,send 的作用是将要发送的数据拷贝到缓冲区，协议负责传输。
recv 函数用来从 TCP 连接的另一端接收数据，当应用程序调用 recv 函数时，recv 先等待 s 的发送缓冲中的数据被协议传送完毕，然后从缓冲区中读取接收到的内容给应用层。
accept 函数用了接收一个连接，内核维护了半连接队列和一个已完成连接队列，当队列为空的时候，accept 函数阻塞，不为空的时候 accept 函数从上边取下来一个已完成连接，返回一个文件描述符。

30、请你说一下 http 协议会话结束标志怎么截出来？

参考回答：

看 tcp 连接是否有断开的四部挥手阶段。

31、请你说一说三次握手

参考回答：

1.客户端发送 syn0 给服务器
2.服务器收到 syn0，回复 syn1,ack(syn0+1)
3.客户端收到 syn1，回复 ack(syn1+1)

32、请你说一说四次挥手

参考回答：

1.客户端发送 syn0 给服务器
2.服务器收到 syn0，回复 ack(syn0+1)
3.服务器发送 syn1
4.客户端收到 syn1，回复 ack(syn1+1)

33、请你说一说 TCP/IP 数据链路层的交互过程

参考回答：

网络层等到数据链层用 mac 地址作为通信目标，数据包到达网络等准备往数据链层发送的时候，首先会去自己的 arp 缓存表(存着 ip-mac 对应关系)去查找改目标 ip 的 mac 地址，如果查到了，就讲目标 ip 的 mac 地址封装到链路层数据包的包头。如果缓存中没有找到，会发起一个广播：

who is ip XXX tell ip XXX,所有收到的广播的机器看这个 ip 是不是自己的，如果是自己的，则以单拨的形式将自己的 mac 地址回复给请求的机器

四、数据库

1、数据库基础

1.请你说一下数据库事务以及四个特性

考察点：数据库

参考回答：

事务（Transaction）是由一系列对系统中数据进行访问与更新的操作所组成的一个程序执行逻辑单元。事务是 DBMS 中最基础的单位，事务不可分割。

事务具有 4 个基本特征，分别是：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Duration），简称 ACID。

1. 原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，[删删删]因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

2. 一致性（Consistency）

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

拿转账来说，假设用户 A 和用户 B 两者的钱加起来一共是 5000，那么不管 A 和 B 之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是 5000，这就是事务的一致性。

3. 隔离性（Isolation）

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

即要达到这么一种效果：对于任意两个并发的事务 T1 和 T2，在事务 T1 看来，T2 要么在 T1 开始之前就已经结束，要么在 T1 结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

多个事务并发访问时，事务之间是隔离的，一个事务不应该影响其它事务运行效果。

这指的是在并发环境中，当不同的事务同时操纵相同的数据时，每个事务都有各自的完整数据空间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。

不同的隔离级别：

Read Uncommitted（读取未提交[添加中文释义]内容）：最低的隔离级别，什么都不需要做，一个事务可以读到另一个事务未提交的结果。所有的并发事务问题都会发生。

Read Committed（读取提交内容）：只有在事务提交后，其更新结果才会被其他事务看见。可以解决脏读问题。

Repeated Read（可重复读）：在一个事务中，对于同一份数据的读取结果总是相同的，无论是否有其他事务对这份数据进行操作，以及这个事务是否提交。可以解决脏读、不可重复读。

Serialization（可串行化）：事务串行化执行，隔离级别最高，牺牲了系统的并发性。可以解决并发事务的所有问题。

4. 持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

例如我们在使用 JDBC 操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

2、请你说一说数据库的三大范式

考点：数据库

参考回答：

第一范式：当关系模式 R 的所有属性都不能再分解为更基本的数据单位时，称 R 是满足第一范式，即属性不可分

第二范式：如果关系模式 R 满足第一范式，并且 R 得所有非主属性都完全依赖于 R 的每一个候选关键属性，称 R 满足第二范式

第三范式：设 R 是一个满足第一范式条件的关系模式，X 是 R 的任意属性集，如果 X 非传递依赖于 R 的任意一个候选关键字，称 R 满足第三范式，即非主属性不传递依赖于键码

3、请你介绍一下数据库的 ACID 特性

考点：数据库 ACID 特性

参考回答：

1)原子性：事务被视为不可分割的最小单元，事物的所有操作要不成功，要不失败回滚，而回滚可以通过日志来实现，日志记录着事务所执行的修改操作，在回滚时反向执行这些修改操作。

2)一致性：数据库在事务执行前后都保持一致性状态，在一致性状态下，所有事务对一个数据的读取结果都是相同的。

3)隔离性：一个事务所做的修改在最终提交以前，对其他事务是可不见的。

4)持久性：一旦事务提交，则其所做的修改将会永远保存到数据库中。

4、请你说一说数据库索引

参考回答：

索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。如果想按特定职员的姓来查找他或她，则与在表中搜索所有的行相比，索引有助于更快地获取信息。
索引的一个主要目的就是加快检索表中数据的方法，亦即能协助信息搜索者尽快的找到符合限制条件的记录 ID 的辅助数据结构。

5、请你说一说数据库事务

参考回答：

数据库事务(Database Transaction) ，是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。 事务处理可以确保除非事务性单元内的所有操作都成功完成，否则不会永久更新面向数据的资源。通过将一组相关操作组合为一个要么全部成功要么全部失败的单元，可以简化错误恢复并使应用程序更加可靠。一个逻辑工作单元要成为事务，必须满足所谓的 ACID（原子性、一致性、隔离性和持久性）属性。事务是数据库运行中的逻辑工作单位，由 DBMS 中的事务管理子系统负责事务的处理。

6、请你说一说数据库事务隔离

参考回答：

同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如 A 正在从一张银行卡中取钱，在 A 取钱的过程结束前，B 不能向这张卡转账。

7、请你说一说 inner join 和 left join

参考回答：

left join(左联接) 返回包括左表中的所有记录和右表中联结字段相等的记录
right join(右联接) 返回包括右表中的所有记录和左表中联结字段相等的记录
inner join(等值连接) 只返回两个表中联结字段相等的行

8、请你聊一聊数据库事物的一致性

考察点：数据库

参考回答：

事务（Transaction）是由一系列对系统中数据进行访问与更新的操作所组成的一个程序执行逻辑单元。事务是 DBMS 中最基础的单位，事务不可分割。

事务具有 4 个基本特征，分别是：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Duration），简称 ACID。

1）原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，[删删删]因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

2）一致性（Consistency）

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

拿转账来说，假设用户 A 和用户 B 两者的钱加起来一共是 5000，那么不管 A 和 B 之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是 5000，这就是事务的一致性。

3）隔离性（Isolation）

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

即要达到这么一种效果：对于任意两个并发的事务 T1 和 T2，在事务 T1 看来，T2 要么在 T1 开始之前就已经结束，要么在 T1 结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

多个事务并发访问时，事务之间是隔离的，一个事务不应该影响其它事务运行效果。这指的是在并发环境中，当不同的事务同时操纵相同的数据时，每个事务都有各自的完整数据空间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。

不同的隔离级别：

Read Uncommitted（读取未提交[添加中文释义]内容）：最低的隔离级别，什么都不需要做，一个事务可以读到另一个事务未提交的结果。所有的并发事务问题都会发生。

Read Committed（读取提交内容）：只有在事务提交后，其更新结果才会被其他事务看见。可以解决脏读问题。

Repeated Read（可重复读）：在一个事务中，对于同一份数据的读取结果总是相同的，无论是否有其他事务对这份数据进行操作，以及这个事务是否提交。可以解决脏读、不可重复读。

Serialization（可串行化）：事务串行化执行，隔离级别最高，牺牲了系统的并发性。可以解决并发事务的所有问题。

4）持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

例如我们在使用 JDBC 操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

9、请你说说索引是什么，多加索引一定会好吗

参考回答：

1、索引

数据库索引是为了增加查询速度而对表字段附加的一种标识，是对数据库表中一列或多列的值进行排序的一种结构。

DB 在执行一条 Sql 语句的时候，默认的方式是根据搜索条件进行全表扫描，遇到匹配条件的就加入搜索结果集合。如果我们对某一字段增加索引，查询时就会先去索引列表中一次定位到特定值的行数，大大减少遍历匹配的行数，所以能明显增加查询的速度。

优点：

通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

缺点：

创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

2、添加索引原则

在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

定义为 text、image 和 bit 数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

10、k-v 存储中，key 有哪些要求？

参考回答：

略

11、介绍数据库中的 WAL 技术

参考回答：

略

2、 Mysql

1.请你说一说 mysql 的四种隔离状态

考点：mysql 隔离状态

参考回答：

Mysql 主要包含四种隔离状态：

事务隔离级别	脏读	不可重复读	幻读
读未提交（read-uncommitted）	是	是	是
不可重复读（read-committed）	否	是	是
可重复读（repeatable-read）	否	否	是
串行化（serializable）	否	否	否

2、请你介绍一下 mysql 的 MVCC 机制

考点：MVCC

参考回答：

MVCC 是一种多版本并发控制机制，是 MySQL 的 InnoDB 存储引擎实现隔离级别的一种具体方式，用于实现提交读和可重复读这两种隔离级别。MVCC 是通过保存数据在某个时间点的快照来实现该机制，其在每行记录后面保存两个隐藏的列，分别保存这个行的创建版本号和删除版本号，然后 Innodb 的 MVCC 使用到的快照存储在 Undo 日志中，该日志通过回滚指针把一个数据行所有快照连接起来。

3、请问 SQL 优化方法有哪些

考点：数据库 优化

参考回答：

通过建立索引对查询进行优化

对查询进行优化，应尽量避免全表扫描

4、请你说一下 MySQL 引擎和区别

考察点：数据库

参考回答：

1、MySQL 引擎

MySQL 中的数据用各种不同的技术存储在文件（或者内存）中。这些技术中的每一种技术都使用不同的存储机制、索引技巧、锁定水平并且最终提供广泛的不同的功能和能力。通过选择不同的技术，你能够获得额外的速度或者功能，从而改善你的应用的整体功能。

数据库引擎是用于存储、处理和保护数据的核心服务。利用数据库引擎可控制访问权限并快速处理事务，从而满足企业内大多数需要处理大量数据的应用程序的要求。 使用数据库引擎创建用于联机事务处理或联机分析处理数据的关系数据库。这包括创建用于存储数据的表和用于查看、管理和保护数据安全的数据库对象（如索引、视图和存储过程）。

MySQL 存储引擎主要有： MyIsam、InnoDB、Memory、Blackhole、CSV、Performance_Schema、Archive、Federated、Mrg_Myisam。

但是最常用的是 InnoDB 和 Mylsam。

2、InnoDB

InnoDB 是一个事务型的存储引擎，有行级锁定和外键约束。

Innodb 引擎提供了对数据库 ACID 事务的支持，并且实现了 SQL 标准的四种隔离级别，关于数据库事务与其隔离级别的内容请见数据库事务与其隔离级别这类型的文章。该引擎还提供了行级锁和外键约束，它的设计目标是处理大容量数据库系统，它本身其实就是基于 MySQL 后台的完整数据库系统，MySQL 运行时 Innodb 会在内存中建立缓冲池，用于缓冲数据和索引。但是该引擎不支持 FULLTEXT 类型的索引，而且它没有保存表的行数，当 SELECT COUNT(*) FROM TABLE 时需要扫描全表。当需要使用数据库事务时，该引擎当然是首选。由于锁的粒度更小，写操作不会锁定全表，所以在并发较高时，使用 Innodb 引擎会提升效率。但是使用行级锁也不是绝对的，如果在执行一个 SQL 语句时 MySQL 不能确定要扫描的范围，InnoDB 表同样会锁全表。

适用场景：

经常更新的表，适合处理多重并发的更新请求。

支持事务。

可以从灾难中恢复（通过 bin-log 日志等）。

外键约束。只有他支持外键。

支持自动增加列属性 auto_increment。

索引结构：

InnoDB 也是 B+Treee 索引结构。Innodb 的索引文件本身就是数据文件，即 B+Tree 的数据域存储的就是实际的数据，这种索引就是聚集索引。这个索引的 key 就是数据表的主键，因此 InnoDB 表数据文件本身就是主索引。

InnoDB 的辅助索引数据域存储的也是相应记录主键的值而不是地址，所以当以辅助索引查找时，会先根据辅助索引找到主键，再根据主键索引找到实际的数据。所以 Innodb 不建议使用

过长的主键，否则会使辅助索引变得过大。建议使用自增的字段作为主键，这样 B+Tree 的每一个结点都会被顺序的填满，而不会频繁的分裂调整，会有效的提升插入数据的效率。

3、Mylsam

MyIASM 是 MySQL 默认的引擎，但是它没有提供对数据库事务的支持，也不支持行级锁和外键，因此当 INSERT 或 UPDATE 数据时即写操作需要锁定整个表，效率便会低一些。MyIsam 存储引擎独立于操作系统，也就是可以在 windows 上使用，也可以比较简单的将数据转移到 linux 操作系统上去。

适用场景：

不支持事务的设计，但是并不代表着有事务操作的项目不能用 MyIsam 存储引擎，可以在 service 层进行根据自己的业务需求进行相应的控制。

不支持外键的表设计。

查询速度很快，如果数据库 insert 和 update 的操作比较多的话比较适用。

整天对表进行加锁的场景。

MyISAM 极度强调快速读取操作。

MyIASM 中存储了表的行数，于是 SELECT COUNT(*) FROM TABLE 时只需要直接读取已经保存好的值而不需要进行全表扫描。如果表的读操作远远多于写操作且不需要数据库事务的支持，那么 MyIASM 也是很好的选择。

缺点：就是不能在表损坏后主动恢复数据。

索引结构：

MyISAM 索引结构：MyISAM 索引用的 B+ tree 来储存数据，MyISAM 索引的指针指向的是键值的地址，地址存储的是数据。B+Tree 的数据域存储的内容为实际数据的地址，也就是说它的索引和实际的数据是分开的，只不过是用索引指向了实际的数据，这种索引就是所谓的非聚集索引。

3、InnoDB 和 Mylsam 的区别：

1）事务：MyISAM 类型不支持事务处理等高级处理，而 InnoDB 类型支持，提供事务支持已经外部键等高级数据库功能。

2）性能：MyISAM 类型的表强调的是性能，其执行数度比 InnoDB 类型更快。

3）行数保存：InnoDB 中不保存表的具体行数，也就是说，执行 select count() fromtable 时，InnoDB 要扫描一遍整个表来计算有多少行，但是 MyISAM 只要简单的读出保存好的行数即可。注意的是，当 count()语句包含 where 条件时，两种表的操作是一样的。

4）索引存储：对于 AUTO_INCREMENT 类型的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中，可以和其他字段一起建立联合索引。MyISAM 支持全文索引（FULLTEXT）、压缩索引，InnoDB 不支持。

MyISAM 的索引和数据是分开的，并且索引是有压缩的，内存使用率就对应提高了不少。能加载更多索引，而 Innodb 是索引和数据是紧密捆绑的，没有使用压缩从而会造成 Innodb 比 MyISAM 体积庞大不小。

InnoDB 存储引擎被完全与 MySQL 服务器整合，InnoDB 存储引擎为在主内存中缓存数据和索引而维持它自己的缓冲池。InnoDB 存储它的表＆索引在一个表空间中，表空间可以包含数个文件（或原始磁盘分区）。这与 MyISAM 表不同，比如在 MyISAM 表中每个表被存在分离的文件中。InnoDB 表可以是任何尺寸，即使在文件尺寸被限制为 2GB 的操作系统上。

5）服务器数据备份：InnoDB 必须导出 SQL 来备份，LOAD TABLE FROM MASTER 操作对 InnoDB 是不起作用的，解决方法是首先把 InnoDB 表改成 MyISAM 表，导入数据后再改成 InnoDB 表，但是对于使用的额外的 InnoDB 特性(例如外键)的表不适用。

MyISAM 应对错误编码导致的数据恢复速度快。MyISAM 的数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。

InnoDB 是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十 G 的时候就相对痛苦了。

6）锁的支持：MyISAM 只支持表锁。InnoDB 支持表锁、行锁 行锁大幅度提高了多用户并发操作的新能。但是 InnoDB 的行锁，只是在 WHERE 的主键是有效的，非主键的 WHERE 都会锁全表的。

3、Redis

1、请你回答一下 mongodb 和 redis 的区别

考点：mongodb 数据库 redis 数据库

参考回答：

内存管理机制上：Redis 数据全部存在内存，定期写入磁盘，当内存不够时，可以选择指定的 LRU 算法删除数据。MongoDB 数据存在内存，由 linux 系统 mmap 实现，当内存不够时，只将热点数据放入内存，其他数据存在磁盘。

支持的数据结构上：Redis 支持的数据结构丰富，包括 hash、set、list 等。

MongoDB 数据结构比较单一，但是支持丰富的数据表达，索引，最类似关系型数据库，支持的查询语言非常丰富

2、请你说一下 mysql 引擎以及其区别

考点：mysql

参考回答：

在 Mysql 数据库中,常用的引擎为 Innodb 和 MyIASM,其中 Innodb 是一个事务型的存储引擎，有行级锁定和外键约束，提供了对数据库 ACID 事物的支持，实现了 SQL 标准的四种隔离级别，即读未提交，不可重复读，可重复读以及串行,其涉及目标就是处理大数据容量的数据库系统。而 MyIASM 引擎是 Mysql 默认的引擎，不提供数据库事务的支持，也不支持行级锁和外键，因此当写操作时需要锁定整个表，效率较低。不过其保存了表的行数，当金星 select count(*)form table 时，可直接读取已经保存的值，不需要进行全表扫描。因此当表的读操作远多于写操作，并且不需要事务支持时，可以优先选择 MyIASM

3、请你来说一说 Redis 的定时机制怎么实现的

考点：redis 定时机制

参考回答：

Redis 服务器是一个事件驱动程序，服务器需要处理以下两类事件：文件事件（服务器对套接字操作的抽象）和时间事件（服务器对定时操作的抽象）。Redis 的定时机制就是借助时间事件实现的。

一个时间事件主要由以下三个属性组成：id：时间事件标识号；when：记录时间事件的到达时间；timeProc：时间事件处理器，当时间事件到达时，服务器就会调用相应的处理器来处理时间。一个时间事件根据时间事件处理器的返回值来判断是定时事件还是周期性事件

一个时间事件主要由以下三个属性组成：id：时间事件标识号；when：记录时间事件的到达时间；timeProc：时间事件处理器，当时间事件到达时，服务器就会调用相应的处理器来处理时间。一个时间事件根据时间事件处理器的返回值来判断是定时事件还是周期性事件。

4、请你来说一说 Redis 是单线程的，但是为什么这么高效呢?

考点：Redis I/O 复用

参考回答：

虽然 Redis 文件事件处理器以单线程方式运行,但是通过使用 I/O 多路复用程序来监听多个套接字，文件事件处理器既实现了高性能的网络通信模型，又可以很好地与 Redis 服务器中其他同样以单线程运行的模块进行对接，这保持了 Redis 内部单线程设计的简单性。

5、请问 Redis 的数据类型有哪些，底层怎么实现?

参考回答：

1）字符串：整数值、embstr 编码的简单动态字符串、简单动态字符串（SDS）

2）列表：压缩列表、双端链表

3）哈希：压缩列表、字典

4）集合：整数集合、字典

5）有序集合：压缩列表、跳跃表和字典

6、请问 Redis 的 rehash 怎么做的，为什么要渐进 rehash，渐进 rehash 又是怎么实现的?

参考回答：

因为 redis 是单线程，当 K 很多时，如果一次性将键值对全部 rehash，庞大的计算量会影响服务器性能，甚至可能会导致服务器在一段时间内停止服务。不可能一步完成整个 rehash 操作，所以 redis 是分多次、渐进式的 rehash。渐进性哈希分为两种：

1）操作 redis 时，额外做一步 rehash

对 redis 做读取、插入、删除等操作时，会把位于 table[dict->rehashidx]位置的链表移动到新的 dictht 中，然后把 rehashidx 做加一操作，移动到后面一个槽位。

2）后台定时任务调用 rehash

后台定时任务 rehash 调用链，同时可以通过 server.hz 控制 rehash 调用频率

7、请你来说一下 Redis 和 memcached 的区别

考点：Redis memcached

参考回答：

1）数据类型 ：redis 数据类型丰富，支持 set liset 等类型；memcache 支持简单数据类型，需要客户端自己处理复杂对象

2）持久性：redis 支持数据落地持久化存储；memcache 不支持数据持久存储。）

3）分布式存储：redis 支持 master-slave 复制模式；memcache 可以使用一致性 hash 做分布式。

4）value 大小不同：memcache 是一个内存缓存，key 的长度小于 250 字符，单个 item 存储要小于 1M，不适合虚拟机使用

5）数据一致性不同：redis 使用的是单线程模型，保证了数据按顺序提交；memcache 需要使用 cas 保证数据一致性。CAS（Check and Set）是一个确保并发一致性的机制，属于“乐观锁”范畴；原理很简单：拿版本号，操作，对比版本号，如果一致就操作，不一致就放弃任何操作

6）cpu 利用：redis 单线程模型只能使用一个 cpu，可以开启多个 redis 进程

8、请问 Redis 怎么实现的定期删除功能

考察点：redis 数据库

参考回答：略

9、 请你说一说 Redis 对应的命令和数据类型...

考察点：redis 数据库

参考回答：略

五、算法与数据结构

1、树

1、请你来说一说红黑树和 AVL 树的定义，特点，以及二者区别

考察点：数据结构，红黑树，二叉平衡树

参考回答：

平衡二叉树（AVL 树）：

平衡二叉树又称为 AVL 树，是一种特殊的二叉排序树。其左右子树都是平衡二叉树，且左右子树高度之差的绝对值不超过 1。一句话表述为：以树中所有结点为根的树的左右子树高度之差的绝对值不超过 1。将二叉树上结点的左子树深度减去右子树深度的值称为平衡因子 BF，那么平衡二叉树上的所有结点的平衡因子只可能是-1、0 和 1。只要二叉树上有一个结点的平衡因子的绝对值大于 1，则该二叉树就是不平衡的。

红黑树：

红黑树是一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是红或黑（非红即黑）。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因此，红黑树是一种弱平衡二叉树，相对于要求严格的 AVL 树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，通常使用红黑树。

性质：

1. 每个节点非红即黑

2. 根节点是黑的；

3. 每个叶节点（叶节点即树尾端 NULL 指针或 NULL 节点）都是黑的；

4. 如果一个节点是红色的，则它的子节点必须是黑色的。

5. 对于任意节点而言，其到叶子点树 NULL 指针的每条路径都包含相同数目的黑节点；

区别：

AVL 树是高度平衡的，频繁的插入和删除，会引起频繁的 rebalance，导致效率下降；红黑树不是高度平衡的，算是一种折中，插入最多两次旋转，删除最多三次旋转。

2、请你聊一聊哈夫曼编码

考点：哈夫曼编码

参考回答：

哈夫曼编码是哈夫曼树的一种应用，广泛用于数据文件压缩。哈夫曼编码算法用字符在文件中出现的频率来建立使用 0，1 表示个字符的最优表示方式，其具体算法如下：

(1)哈夫曼算法以自底向上的方式构造表示最优前缀码的二叉树 T。

(2)算法以|C|个叶结点开始，执行|C|－1 次的“合并”运算后产生最终所要求的树 T。

(3)假设编码字符集中每一字符 c 的频率是 f(c)。以 f 为键值的优先队列 Q 用在贪心选择时有效地确定算法当前要合并的 2 棵具有最小频率的树。一旦 2 棵具有最小频率的树合并后，产生一棵新的树，其频率为合并的 2 棵树的频率之和，并将新树插入优先队列 Q。经过 n－1 次的合并后，优先队列中只剩下一棵树，即所要求的树 T。

3、请你回答一下 map 底层为什么用红黑树实现

考察点：STL

参考回答：

1、红黑树：

红黑树是一种二叉查找树,但在每个节点增加一个存储位表示节点的颜色,可以是红或黑(非红即黑）。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因此，红黑树是一种弱平衡二叉树，相对于要求严格的 AVL 树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，通常使用红黑树。

性质：

1. 每个节点非红即黑

2. 根节点是黑的;

3. 每个叶节点（叶节点即树尾端 NULL 指针或 NULL 节点）都是黑的;

4. 如果一个节点是红色的，则它的子节点必须是黑色的。

5. 对于任意节点而言，其到叶子点树 NULL 指针的每条路径都包含相同数目的黑节点;

2、平衡二叉树（AVL 树）：

红黑树是在 AVL 树的基础上提出来的。

平衡二叉树又称为 AVL 树，是一种特殊的二叉排序树。其左右子树都是平衡二叉树，且左右子树高度之差的绝对值不超过 1。

AVL 树中所有结点为根的树的左右子树高度之差的绝对值不超过 1。

将二叉树上结点的左子树深度减去右子树深度的值称为平衡因子 BF，那么平衡二叉树上的所有结点的平衡因子只可能是-1、0 和 1。只要二叉树上有一个结点的平衡因子的绝对值大于 1，则该二叉树就是不平衡的。

3、红黑树较 AVL 树的优点：

AVL 树是高度平衡的，频繁的插入和删除，会引起频繁的 rebalance，导致效率下降；红黑树不是高度平衡的，算是一种折中，插入最多两次旋转，删除最多三次旋转。

所以红黑树在查找，插入删除的性能都是 O(logn)，且性能稳定，所以 STL 里面很多结构包括 map 底层实现都是使用的红黑树。

4、请你介绍一下 B+树

参考回答：

B+是一种多路搜索树，主要为磁盘或其他直接存取辅助设备而设计的一种平衡查找树，在 B+树中，每个节点的可以有多个孩子，并且按照关键字大小有序排列。所有记录节点都是按照键值的大小顺序存放在同一层的叶节点中。相比 B 树，其具有以下几个特点：

每个节点上的指针上限为 2d 而不是 2d+1（d 为节点的出度）

内节点不存储 data,只存储 key

叶子节点不存储指针

5、请你说一说 map 和 unordered_map 的底层实现

考点：map unordered_map

map 底层是基于红黑树实现的，因此 map 内部元素排列是有序的。而 unordered_map 底层则是基于哈希表实现的，因此其元素的排列顺序是杂乱无序的。

6、请你回答一下 map 和 unordered_map 优点和缺点

考点：map unordered_map 红黑树 哈希表

参考回答：

对于 map，其底层是基于红黑树实现的，优点如下：

1)有序性，这是 map 结构最大的优点，其元素的有序性在很多应用中都会简化很多的操作

2)map 的查找、删除、增加等一系列操作时间复杂度稳定，都为 logn

缺点如下：

1）查找、删除、增加等操作平均时间复杂度较慢，与 n 相关

对于 unordered_map 来说，其底层是一个哈希表，优点如下：

查找、删除、添加的速度快，时间复杂度为常数级 O(c)

缺点如下：

因为 unordered_map 内部基于哈希表，以（key,value）对的形式存储，因此空间占用率高

Unordered_map 的查找、删除、添加的时间复杂度不稳定，平均为 O(c)，取决于哈希函数。极端情况下可能为 O(n)

7、请你回答一下 epoll 怎么实现的

参考回答：

Linux epoll 机制是通过红黑树和双向链表实现的。 首先通过 epoll_create()系统调用在内核中创建一个 eventpoll 类型的句柄，其中包括红黑树根节点和双向链表头节点。然后通过 epoll_ctl()系统调用，向 epoll 对象的红黑树结构中添加、删除、修改感兴趣的事件，返回 0

标识成功，返回-1 表示失败。最后通过 epoll_wait()系统调用判断双向链表是否为空，如果为空则阻塞。当文件描述符状态改变，fd 上的回调函数被调用，该函数将 fd 加入到双向链表中，此时 epoll_wait 函数被唤醒，返回就绪好的事件。

8、请你说一说 C++两种 map

参考回答：

unordered_map（哈希表)和 map（红黑树）

9、请问红黑树了解吗

参考回答：

略

10、请你说一说红黑树的性质还有左右旋转

考察点：算法

1）平衡二叉树（AVL 树）：

红黑树是在 AVL 树的基础上提出来的。

平衡二叉树又称为 AVL 树，是一种特殊的二叉排序树。其左右子树都是平衡二叉树，且左右子树高度之差的绝对值不超过 1。

AVL 树中所有结点为根的树的左右子树高度之差的绝对值不超过 1。

将二叉树上结点的左子树深度减去右子树深度的值称为平衡因子 BF，那么平衡二叉树上的所有结点的平衡因子只可能是-1、0 和 1。只要二叉树上有一个结点的平衡因子的绝对值大于 1，则该二叉树就是不平衡的。

2）红黑树：

红黑树是在 AVL 树的基础上发展而来的。红黑树是一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是红或黑（非红即黑）。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因此，红黑树是一种弱平衡二叉树，相对于要求严格的 AVL 树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，通常使用红黑树。

性质：

1. 每个节点非红即黑

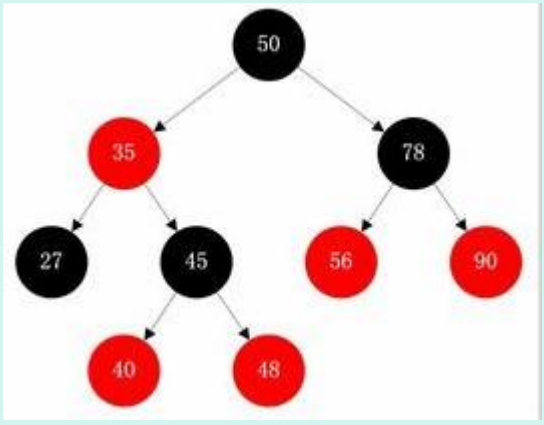
2. 根节点是黑的;

3. 每个叶节点（叶节点即树尾端 NULL 指针或 NULL 节点）都是黑的;

4. 如果一个节点是红色的，则它的子节点必须是黑色的。

5. 对于任意节点而言，其到叶子点树 NULL 指针的每条路径都包含相同数目的黑节点;

从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。它可以在 O(log n)时间内做查找，插入和删除，这里的 n 是树中元素的数目。恢复红黑属性需要少量(O(log n))的颜色变更(这在实践中是非常快速的)并且不超过三次树旋转(对于插入是两次)。这允许插入和删除保持为 O(log n) 次，



3）红黑树较 AVL 树的优点：

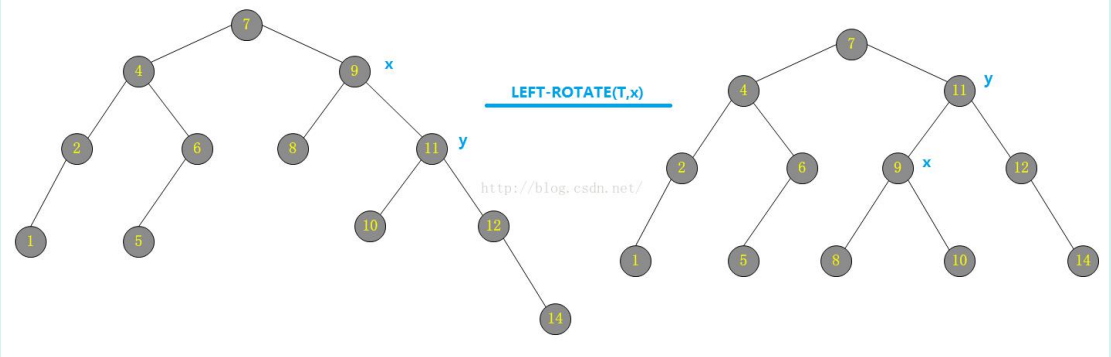
AVL 树是高度平衡的，频繁的插入和删除，会引起频繁的 rebalance，导致效率下降；红黑树不是高度平衡的，算是一种折中，插入最多两次旋转，删除最多三次旋转。

所以红黑树在查找，插入删除的性能都是 O(logn)，且性能稳定，所以 STL 里面很多结构包括 map 底层实现都是使用的红黑树。

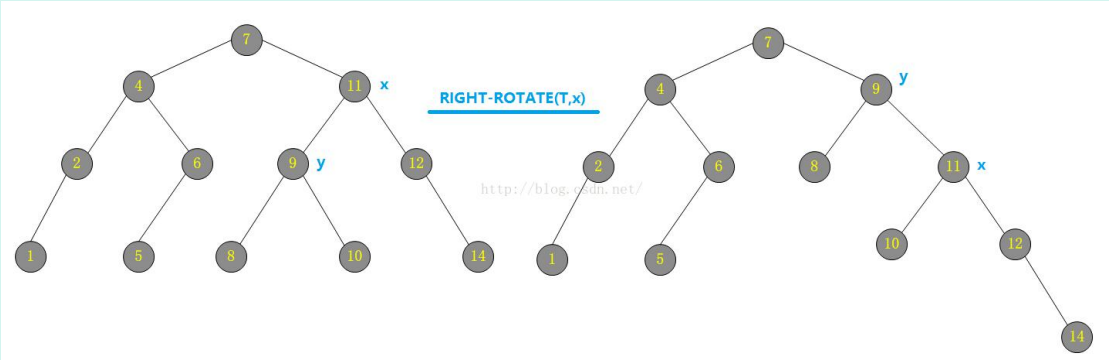
4）红黑树旋转：

旋转：红黑树的旋转是一种能保持二叉搜索树性质的搜索树局部操作。有左旋和右旋两种旋转,通过改变树中某些结点的颜色以及指针结构来保持对红黑树进行插入和删除操作后的红黑性质。

左旋：对某个结点 x 做左旋操作时，假设其右孩子为 y 而不是 T.nil：以 x 到 y 的链为“支轴”进行。使 y 成为该子树新的根结点，x 成为 y 的左孩子，y 的左孩子成为 x 的右孩子。



右旋：对某个结点 x 做右旋操作时，假设其左孩子为 y 而不是 T.nil：以 x 到 y 的链为“支轴”进行。使 y 成为该子树新的根结点，x 成为 y 的右孩子，y 的右孩子成为 x 的左孩子。



11、请你说一说红黑树的原理以及 erase 以后迭代器的具体分布情况？

参考回答：略

12、请你实现二叉树的层序遍历并输出

参考回答：

```
void layerTrace(BTreeNode *T)
{
    if(T== nullptr)return;
    BTreeNode *p=T;
    queue<BTreeNode*>q;
    q.push(p);
    while(!q.empty())
    {
        p=q.front();
        q.pop();
        cout<<<<p->data;
        if(p->left!= nullptr)q.push(p->left);
        if(p->right!= nullptr)q.push(p->right);
```

```
        }
    }```
```

13、手写代码：二叉树序列化反序列化

参考回答：

> 序列化：必须保存一个中序遍历结果，然后外加一个前序或者后序遍历结果

>反序列化：根据两次遍历生成的结果恢复二叉树，代码如下(前序和中序)：

```
```TreeNode* helper(vector<int>pre,int startPre,int endPre,vector<int>in,int startIn,int endIn)
        {
            if(startPre>endPre||startIn>endIn)
                return nullptr;
            TreeNode * root=new TreeNode(pre[startPre]);
            for(int i=startIn;i<=endIn;++i)
            {
                if(in[i]==pre[startPre])
                {

root->left=helper(pre,startPre+1,startPre+i-startIn,in,startIn,i-1);

root->right=helper(pre,i-startIn+startPre+1,endPre,in,i+1,endIn);
                    break;
                }
            }
            return root;
```
```

```
    }
    TreeNode* reConstructBinaryTree(vector<int> pre,vector<int> vin)
    {
        TreeNode *root=helper(pre,0,pre.size()-1,vin,0,vin.size()-1);
        return root;
    }
```

2、堆与栈

1、请说一说你理解的 stack overflow，并举个简单例子导致栈溢出

考察点：栈溢出

参考回答：

栈溢出概念：

栈溢出指的是程序向栈中某个变量中写入的字节数超过了这个变量本身所申请的字节数,因而导致栈中与其相邻的变量的值被改变。

栈溢出的原因：

1. 局部数组过大。当函数内部的数组过大时，有可能导致堆栈溢出。局部变量是存储在栈中的，因此这个很好理解。解决这类问题的办法有两个，一是增大栈空间,二是改用动态分配，使用堆（heap）而不是栈（stack）。

2. 递归调用层次太多。递归函数在运行时会执行压栈操作，当压栈次数太多时，也会导致堆栈溢出。

3. 指针或数组越界。这种情况最常见，例如进行字符串拷贝，或处理用户输入等等。

栈溢出例子：

```
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv[]) {
        char buf[256];
        strcpy(buf,argv[1]);
```

```
        printf("Input:%s\n",buf);

        return 0;

}
```

上述代码中的 strcpy(buf,argv[1]);这一行发生了缓冲区溢出错误，因为源缓冲区内容是用户输入的。

2、请你回答一下栈和堆的区别，以及为什么栈要快

知识点：堆 栈

参考回答：

堆和栈的区别：

堆是由低地址向高地址扩展；栈是由高地址向低地址扩展

堆中的内存需要手动申请和手动释放；栈中内存是由 OS 自动申请和自动释放，存放着参数、局部变量等内存

堆中频繁调用 malloc 和 free,会产生内存碎片，降低程序效率；而栈由于其先进后出的特性，不会产生内存碎片

堆的分配效率较低，而栈的分配效率较高

栈的效率高的原因：

栈是操作系统提供的数据结构，计算机底层对栈提供了一系列支持：分配专门的寄存器存储栈的地址，压栈和入栈有专门的指令执行；而堆是由 C/C++函数库提供的，机制复杂，需要一些列分配内存、合并内存和释放内存的算法，因此效率较低。

3、手写代码：两个栈实现一个队列

考点：算法

参考回答：

```
class Solution
{
public:
    void push(int node) {
        stack1.push(node);
    }

    int pop() {
        if(stack2.size()!=0){
            int tmp = stack2.top();
            stack2.pop();
            return tmp;
        }
        else{
            while(stack1.size()!=0){
                int tmp = stack1.top();
                stack1.pop();
                stack2.push(tmp);
            }
            return pop();
        }

    }
```

```
    private:
        stack<int> stack1;
        stack<int> stack2;
};
```

4、请你来说一下堆和栈的区别

考察点：C++

参考回答：

1）申请方式：

栈由系统自动分配和管理，堆由程序员手动分配和管理。

2）效率：

栈由系统分配，速度快，不会有内存碎片。

堆由程序员分配，速度较慢，可能由于操作不当产生内存碎片。

3）扩展方向

栈从高地址向低地址进行扩展，堆由低地址向高地址进行扩展。

4）程序局部变量是使用的栈空间，new/malloc 动态申请的内存是堆空间，函数调用时会进行形参和返回值的压栈出栈，也是用的栈空间。

5、请你说一说小根堆特点

考察点：数据结构，小根堆

参考回答：

堆是一棵完全二叉树（如果一共有 h 层，那么 1~h-1 层均满，在 h 层可能会连续缺失若干个右叶子）。

1）小根堆

若根节点存在左子女则根节点的值小于左子女的值；若根节点存在右子女则根节点的值小于右子女的值。

2）大根堆

若根节点存在左子女则根节点的值大于左子女的值；若根节点存在右子女则根节点的值大于右子女的值。

3、数组

1、请你回答一下 Array&List， 数组和链表的区别

考察点：数据结构

参考回答：

数组的特点：

数组是将元素在内存中连续存放，由于每个元素占用内存相同，可以通过下标迅速访问数组中任何元素。数组的插入数据和删除数据效率低，插入数据时，这个位置后面的数据在内存中都要向后移。删除数据时，这个数据后面的数据都要往前移动。但数组的随机读取效率很高。因为数组是连续的，知道每一个数据的内存地址，可以直接找到给地址的数据。如果应用需要快速访问数据，很少或不插入和删除元素，就应该用数组。数组需要预留空间，在使用前要先申请占内存的大小，可能会浪费内存空间。并且数组不利于扩展，数组定义的空间不够时要重新定义数组。

链表的特点：

链表中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起。比如：上一个元素有个指针指到下一个元素，以此类推，直到最后一个元素。如果要访问链表中一个元素，需要从第一个元素开始，一直找到需要的元素位置。但是增加和删除一个元素对于链表数据结构就非常简单了，只要修改元素中的指针就可以了。如果应用需要经常插入和删除元素你就需要用链表数据结构了。不指定大小，扩展方便。链表大小不用定义，数据随意增删。

各自的优缺点

数组的优点：

1. 随机访问性强
2. 查找速度快

数组的缺点:

1. 插入和删除效率低
2. 可能浪费内存

3. 内存空间要求高，必须有足够的连续内存空间。

4. 数组大小固定，不能动态拓展

链表的优点:

1. 插入删除速度快

2. 内存利用率高，不会浪费内存

3. 大小没有固定，拓展很灵活。

链表的缺点:

不能随机查找，必须从第一个开始遍历，查找效率低

2、一个长度为 N 的整形数组，数组中每个元素的取值范围是[0,n-1],判断该数组否有重复的数，请说一下你的思路并手写代码

参考回答：

把每个数放到自己对应序号的位置上，如果其他位置上有和自己对应序号相同的数，那么即为有重复的数值。时间复杂度为 O(N),同时为了节省空间复杂度，可以在原数组上进行操作，空间复杂度为 O(1)

```
bool IsDuplicateNumber(int *array, int n)
{
    if(array==NULL) return false;
    int i,temp;
    for(i=0;i<n;i++)
    {
        while(array[i]!=i)
        {
            if(array[array[i]]==array[i])
             return true;
            temp=array[array[i]];
```

```
                array[array[i]]=array[i];
                array[i]=temp;
            }
        }
        return false;
    }
```

4、排序

1、请你来手写一下快排的代码

参考回答：

```
int once_quick_sort(vector<int> &data, int left, int right)
{
        int key = data[left];
        while (left < right)
        {
                while (left < right && key <= data[right])
                {
                        right--;
                }
                if (left < right)
                {
                        data[left++] = data[right];
                }
```

```
                while (left < right && key > data[left])
                {
                        left++;
                }
                if (left < right)
                {
                        data[right--] = data[left];
                }
        }
        data[left] = key;
        return left;
}
int quick_sort(vector<int> & data, int left, int right)
{
        if (left >= right )
        {
                return 1;
        }
        int middle = 0;
        middle = once_quick_sort(data, left, right);
        quick_sort(data, left, middle-1);
        quick_sort(data, middle + 1, right);
};
```

2、请你手写一下快排的代码

参考回答：

```
int once_quick_sort(vector<int> &data, int left, int right)
{
	int key = data[left];
	while (left < right)
	{
		while (left < right && key <= data[right])
		{
			right--;
		}
		if (left < right)
		{
			data[left++] = data[right];
		}
		while (left < right && key > data[left])
		{
			left++;
		}
		if (left < right)
		{
			data[right--] = data[left];
		}
	}
	data[left] = key;
	return left;
}
```

```
int quick_sort(vector<int> & data, int left, int right)
{
        if (left >= right )
        {
                return 1;
        }
        int middle = 0;
        middle = once_quick_sort(data, left, right);
        quick_sort(data, left, middle-1);
        quick_sort(data, middle + 1, right);
}
```

3、请问求第 k 大的数的方法以及各自的复杂度是怎样的，另外追问一下，当有相同元素时，还可以使用什么不同的方法求第 k 大的元素

参考回答：

首先使用快速排序算法将数组按照从大到小排序，然后取第 k 个，其时间复杂度最快为 O(nlogn)

使用堆排序，建立最大堆，然后调整堆，知道获得第 k 个元素，其时间复杂度为 O(n+klogn)

首先利用哈希表统计数组中个元素出现的次数，然后利用计数排序的思想，线性从大到小扫描过程中，前面有 k-1 个数则为第 k 大的数

利用快排思想，从数组中随机选择一个数 i，然后将数组分成两部分 Dl,Dr，Dl 的元素都小于 i,Dr 的元素都大于 i。然后统计 Dr 元素个数，如果 Dr 元素个数等于 k-1,那么第 k 大的数即为 k，如果 Dr 元素个数小于 k,那么继续求 Dl 中第 k-Dr 大的元素；如果 Dr 元素个数大于 k,那么继续求 Dr 中第 k 大的元素。

当有相同元素的时候，

首先利用哈希表统计数组中个元素出现的次数，然后利用计数排序的思想，线性从大到小扫描过程中，前面有 k-1 个数则为第 k 大的数，平均情况下时间复杂度为 O(n)

4、请你来介绍一下各种排序算法及时间复杂度

考点：排序算法

参考回答：

插入排序：对于一个带排序数组来说，其初始有序数组元素个数为 1，然后从第二个元素，插入到有序数组中。对于每一次插入操作，从后往前遍历当前有序数组，如果当前元素大于要插入的元素，则后移一位；如果当前元素小于或等于要插入的元素，则将要插入的元素插入到当前元素的下一位中。

希尔排序：先将整个待排序记录分割成若干子序列，然后分别进行直接插入排序，待整个序列中的记录基本有序时，在对全体记录进行一次直接插入排序。其子序列的构成不是简单的逐段分割，而是将每隔某个增量的记录组成一个子序列。希尔排序时间复杂度与增量序列的选取有关，其最后一个值必须为 1.

归并排序：该算法采用分治法；对于包含 m 个元素的待排序序列，将其看成 m 个长度为 1 的子序列。然后两两合归并，得到 n/2 个长度为 2 或者 1 的有序子序列；然后再两两归并，直到得到 1 个长度为 m 的有序序列。

冒泡排序：对于包含 n 个元素的带排序数组，重复遍历数组，首先比较第一个和第二个元素，若为逆序，则交换元素位置；然后比较第二个和第三个元素，重复上述过程。每次遍历会把当前前 n-i 个元素中的最大的元素移到 n-i 位置。遍历 n 次，完成排序。

快速排序：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

选择排序：每次循环，选择当前无序数组中最小的那个元素，然后将其与无序数组的第一个元素交换位置，从而使有序数组元素加 1，无序数组元素减 1.初始时无序数组为空。

堆排序：堆排序是一种选择排序，利用堆这种数据结构来完成选择。其算法思想是将带排序数据构造一个最大堆（升序）/最小堆（降序），然后将堆顶元素与待排序数组的最后一个元素交换位置，此时末尾元素就是最大/最小的值。然后将剩余 n-1 个元素重新构造成最大堆/最小堆。

各个排序的时间复杂度、空间复杂度及稳定性如下：

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性	复杂性
直接插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
希尔排序	$O(nlog_2n)$	$O(n^2)$	$O(n)$	$O(1)$	不稳定	较复杂
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定	较复杂
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定	较复杂

5、请你说一说你知道的排序算法及其复杂度

考察点：算法

参考回答：

常见的排序算法及其复杂度：

1、冒泡排序：

从数组中第一个数开始，依次遍历数组中的每一个数，通过相邻比较交换，每一轮循环下来找出剩余未排序数的中的最大数并“冒泡”至数列的顶端。

稳定性：稳定

平均时间复杂度：O(n ^ 2)

2、插入排序：

从待排序的 n 个记录中的第二个记录开始，依次与前面的记录比较并寻找插入的位置，每次外循环结束后，将当前的数插入到合适的位置。

稳定性：稳定

平均时间复杂度：O(n ^ 2)

3、希尔排序（缩小增量排序）：

希尔排序法是对相邻指定距离(称为增量)的元素进行比较，并不断把增量缩小至 1，完成排序。

希尔排序开始时增量较大，分组较多，每组的记录数目较少，故在各组内采用直接插入排序较快，后来增量 di 逐渐缩小，分组数减少，各组的记录数增多，但由于已经按 di-1 分组排序，文件叫接近于有序状态，所以新的一趟排序过程较快。因此希尔 排序在效率上比直接插入排序有较大的改进。

在直接插入排序的基础上，将直接插入排序中的 1 全部改变成增量 d 即可，因为希尔排序最后一轮的增量 d 就为 1。

稳定性：不稳定

平均时间复杂度：希尔排序算法的时间复杂度分析比较复杂，实际所需的时间取决于各次排序时增量的个数和增量的取值。时间复杂度在 O(n ^ 1.3)到 O(n ^ 2)之间。

4、选择排序：

从所有记录中选出最小的一个数据元素与第一个位置的记录交换;然后在剩下的记录当中再找最小的与第二个位置的记录交换，循环到只剩下最后一个数据元素为止。

稳定性：不稳定

平均时间复杂度：0(n ^ 2)

5、快速排序

1）从待排序的 n 个记录中任意选取一个记录（通常选取第一个记录）为分区标准;

2）把所有小于该排序列的记录移动到左边，把所有大于该排序码的记录移动到右边，中间放所选记录，称之为第一趟排序;

3）然后对前后两个子序列分别重复上述过程，直到所有记录都排好序。

稳定性：不稳定

平均时间复杂度：0(nlogn)

6、堆排序：

堆：

1、完全二叉树或者是近似完全二叉树。

2、大顶堆：父节点不小于子节点键值，小顶堆：父节点不大于子节点键值。左右孩子没有大小的顺序。

堆排序在选择排序的基础上提出的，步骤：

1、建立堆

2、删除堆顶元素，同时交换堆顶元素和最后一个元素，再重新调整堆结构，直至全部删除堆中元素。

稳定性：不稳定

平均时间复杂度：0(nlogn)

7、归并排序：

采用分治思想，现将序列分为一个个子序列，对子序列进行排序合并，直至整个序列有序。

稳定性：稳定

平均时间复杂度：0(nlogn)

8、计数排序：

思想：如果比元素 x 小的元素个数有 n 个，则元素 x 排序后位置为 n+1。

步骤：

1）找出待排序的数组中最大的元素；

2）统计数组中每个值为 i 的元素出现的次数，存入数组 C 的第 i 项；

3）对所有的计数累加（从 C 中的第一个元素开始，每一项和前一项相加）；

4）反向填充目标数组：将每个元素 i 放在新数组的第 C(i)项，每放一个元素就将 C(i)减去 1。

稳定性：稳定

时间复杂度：O(n+k)，k 是待排序数的范围。

9、桶排序：

步骤：

1）设置一个定量的数组当作空桶子；

2）寻访序列，并且把记录一个一个放到对应的桶子去；

3）对每个不是空的桶子进行排序。

4）从不是空的桶子里把项目再放回原来的序列中。

时间复杂度：O(n+C) ，C 为桶内排序时间。

6、请问海量数据如何去取最大的 k 个

考察点：算法，海量数据求 Top K 问题

参考回答：

1.直接全部排序（只适用于内存够的情况）

当数据量较小的情况下，内存中可以容纳所有数据。则最简单也是最容易想到的方法是将数据全部排序，然后取排序后的数据中的前 K 个。

这种方法对数据量比较敏感，当数据量较大的情况下，内存不能完全容纳全部数据，这种方法便不适应了。即使内存能够满足要求，该方法将全部数据都排序了，而题目只要求找出 top K 个数据，所以该方法并不十分高效，不建议使用。

2.快速排序的变形 （只使用于内存够的情况）

这是一个基于快速排序的变形，因为第一种方法中说到将所有元素都排序并不十分高效，只需要找出前 K 个最大的就行。

这种方法类似于快速排序，首先选择一个划分元，将比这个划分元大的元素放到它的前面，比划分元小的元素放到它的后面，此时完成了一趟排序。如果此时这个划分元的序号 index 刚好等于 K，那么这个划分元以及它左边的数，刚好就是前 K 个最大的元素；如果 index ＞K，那么前 K 大的数据在 index 的左边，那么就继续递归的从 index-1 个数中进行一趟排序；如果 index ＜ K，那么再从划分元的右边继续进行排序，直到找到序号 index 刚好等于 K 为止。再将前 K 个数进行排序后，返回 Top K 个元素。这种方法就避免了对除了 Top K 个元素以外的数据进行排序所带来的不必要的开销。

3. 最小堆法

这是一种局部淘汰法。先读取前 K 个数，建立一个最小堆。然后将剩余的所有数字依次与最小堆的堆顶进行比较，如果小于或等于堆顶数据，则继续比较下一个；否则，删除堆顶元素，并将新数据插入堆中，重新调整最小堆。当遍历完全部数据后，最小堆中的数据即为最大的 K 个数。

4. 分治法

将全部数据分成 N 份，前提是每份的数据都可以读到内存中进行处理，找到每份数据中最大的 K 个数。此时剩下 N*K 个数据，如果内存不能容纳 N*K 个数据，则再继续分治处理，分成 M 份，找出每份数据中最大的 K 个数，如果 M*K 个数仍然不能读到内存中，则继续分治处理。直到剩余的数可以读入内存中，那么可以对这些数使用快速排序的变形或者归并排序进行处理。

5. Hash 法

如果这些数据中有很多重复的数据，可以先通过 hash 法，把重复的数去掉。这样如果重复率很高的话，会减少很大的内存用量，从而缩小运算空间。处理后的数据如果能够读入内存，则可以直接排序；否则可以使用分治法或者最小堆法来处理数据。

7、请你说一说 Top(K)问题

考察点：算法

参考回答：

1、直接全部排序（只适用于内存够的情况）

当数据量较小的情况下，内存中可以容纳所有数据。则最简单也是最容易想到的方法是将数据全部排序，然后取排序后的数据中的前 K 个。

这种方法对数据量比较敏感，当数据量较大的情况下，内存不能完全容纳全部数据，这种方法便不适应了。即使内存能够满足要求，该方法将全部数据都排序了，而题目只要求找出 top K 个数据，所以该方法并不十分高效，不建议使用。

2、快速排序的变形 （只使用于内存够的情况）

这是一个基于快速排序的变形，因为第一种方法中说到将所有元素都排序并不十分高效，只需要找出前 K 个最大的就行。

这种方法类似于快速排序，首先选择一个划分元，将比这个划分元大的元素放到它的前面，比划分元小的元素放到它的后面，此时完成了一趟排序。如果此时这个划分元的序号 index 刚好等于 K，那么这个划分元以及它左边的数，刚好就是前 K 个最大的元素；如果 index ＞K，那么前 K 大的数据在 index 的左边，那么就继续递归的从 index-1 个数中进行一趟排序；如果 index ＜ K，那么再从划分元的右边继续进行排序，直到找到序号 index 刚好等于 K 为止。再将前 K 个数进行排序后，返回 Top K 个元素。这种方法就避免了对除了 Top K 个元素以外的数据进行排序所带来的不必要的开销。

3、最小堆法

这是一种局部淘汰法。先读取前 K 个数，建立一个最小堆。然后将剩余的所有数字依次与最小堆的堆顶进行比较，如果小于或等于堆顶数据，则继续比较下一个；否则，删除堆顶元素，并将新数据插入堆中，重新调整最小堆。当遍历完全部数据后，最小堆中的数据即为最大的 K 个数。

4、分治法

将全部数据分成 N 份，前提是每份的数据都可以读到内存中进行处理，找到每份数据中最大的 K 个数。此时剩下 N*K 个数据，如果内存不能容纳 N*K 个数据，则再继续分治处理，分成 M 份，找出每份数据中最大的 K 个数，如果 M*K 个数仍然不能读到内存中，则继续分治处理。直到剩余的数可以读入内存中，那么可以对这些数使用快速排序的变形或者归并排序进行处理。

5、Hash 法

如果这些数据中有很多重复的数据，可以先通过 hash 法，把重复的数去掉。这样如果重复率很高的话，会减少很大的内存用量，从而缩小运算空间。处理后的数据如果能够读入内存，则可以直接排序；否则可以使用分治法或者最小堆法来处理数据。

8、请问快排的时间复杂度最差是多少？什么时候时间最差

参考回答：

O（N2),元素本来倒序排列用时最多

9、请问稳定排序哪几种？

参考回答：

基数排序、冒泡排序、直接插入排序、折半插入排序、归并排序

10、请你介绍一下快排算法；以及什么是稳定性排序，快排是稳定性的吗；快排算法最差情况推导公式

考察点：快排

1、快排算法

根据哨兵元素，用两个指针指向待排序数组的首尾，首指针从前往后移动找到比哨兵元素大的，尾指针从后往前移动找到比哨兵元素小的，交换两个元素，直到两个指针相遇，这是一趟排序，经常这趟排序后，比哨兵元素大的在右边，小的在左边。经过多趟排序后，整个数组有序。

稳定性：不稳定

平均时间复杂度：O(nlogn)

2、稳定排序

假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，r[i]=r[j]，且 r[i]在 r[j]之前，而在排序后的序列中，r[i]仍在 r[j]之前，则称这种排序算法是稳定的；否则称为不稳定的。

快排算法是不稳定的排序算法。例如：

待排序数组:int a[] ={1, 2, 2, 3, 4, 5, 6};

若选择 a[2]（即数组中的第二个 2）为枢轴，而把大于等于比较子的数均放置在大数数组中，则 a[1]（即数组中的第一个 2）会到 pivot 的右边， 那么数组中的两个 2 非原序。

若选择 a[1]为比较子，而把小于等于比较子的数均放置在小数数组中，则数组中的两个 2 顺序也非原序。

3、快排最差情况推倒

在快速排序的早期版本中呢，最左面或者是最右面的那个元素被选为枢轴，那最坏的情况就会在下面的情况下发生啦：

1）数组已经是正序排过序的。 （每次最右边的那个元素被选为枢轴）

2）数组已经是倒序排过序的。 （每次最左边的那个元素被选为枢轴）

3）所有的元素都相同（1、2 的特殊情况）

因为这些案例在用例中十分常见，所以这个问题可以通过要么选择一个随机的枢轴，或者选择一个分区中间的下标作为枢轴，或者（特别是对于相比更长的分区）选择分区的第一个、中间、最后一个元素的中值作为枢轴。有了这些修改，那快排的最差的情况就不那么容易出现了，但是如果输入的数组最大（或者最小元素）被选为枢轴，那最坏的情况就又来了。

快速排序，在最坏情况退化为冒泡排序，需要比较 O(n2)次（n(n - 1)/2 次）。

5、哈希

1、请你来说一说 hash 表的实现，包括 STL 中的哈希桶长度常数。

考点：STL 哈希表实现

参考回答：

hash 表的实现主要包括构造哈希和处理哈希冲突两个方面：

对于构造哈希来说，主要包括直接地址法、平方取中法、除留余数

法等。

对于处理哈希冲突来说，最常用的处理冲突的方法有开放定址法、再哈希法、链地址法、建立公共溢出区等方法。SGL 版本使用链地址法，使用一个链表保持相同散列值的元素。

虽然链地址法并不要求哈希桶长度必须为质数，但 SGI STL 仍然以质数来设计哈希桶长度，并且将 28 个质数（逐渐呈现大约两倍的关系）计算好，以备随时访问，同时提供一个函数，用来查询在这 28 个质数之中，“最接近某数并大于某数”的质数。

2、请你回答一下 hash 表如何 rehash，以及怎么处理其中保存的资源

考点：哈希表 负载因子 rehash

参考回答：

C++的 hash 表中有一个负载因子 loadFactor，当 loadFactor<=1 时，hash 表查找的期望复杂度为 O(1). 因此，每次往 hash 表中添加元素时，我们必须保证是在 loadFactor <1 的情况下，才能够添加。

因此，当 Hash 表中 loadFactor==1 时，Hash 就需要进行 rehash。rehash 过程中，会模仿 C++的 vector 扩容方式，Hash 表中每次发现 loadFactor ==1 时，就开辟一个原来桶数组的两倍空间，称为新桶数组，然后把原来的桶数组中元素全部重新哈希到新的桶数组中。

3、请你说一下哈希表的桶个数为什么是质数，合数有何不妥？

参考回答：

哈希表的桶个数使用质数，可以最大程度减少冲突概率，使哈希后的数据分布的更加均匀。如果使用合数，可能会造成很多数据分布会集中在某些点上，从而影响哈希表效率。

算法：

给定一个数字数组，返回哈夫曼树的头指针

```
struct BTreeNode* CreateHuffman(ElemType a[], int n)
{
    int i, j;
    struct BTreeNode **b, *q;
    b = malloc(n*sizeof(struct BTreeNode));
    for (i = 0; i < n; i++)
    {
        b[i] = malloc(sizeof(struct BTreeNode));
        b[i]->data = a[i];
        b[i]->left = b[i]->right = NULL;
    }
    for (i = 1; i < n; i++)
    {

        int k1 = -1, k2;
        for (j = 0; j < n; j++)
        {
            if (b[j] != NULL && k1 == -1)
            {
                k1 = j;
                continue;
            }
            if (b[j] != NULL)
            {
```

```
                k2 = j;
                break;
            }
        }
        for (j = k2; j < n; j++)
        {
            if (b[j] != NULL)
            {
                if (b[j]->data < b[k1]->data)
                {
                    k2 = k1;
                    k1 = j;
                }
                else if (b[j]->data < b[k2]->data)
                    k2 = j;
            }
        }
        q = malloc(sizeof(struct BTreeNode));
        q->data = b[k1]->data + b[k2]->data;
        q->left = b[k1];
        q->right = b[k2];

        b[k1] = q;
        b[k2] = NULL;
    }
```

```
        free(b);
        return q;
    }
```

4、请你说一下解决 hash 冲突的方法

考点：哈希

参考回答：

当哈希表关键字集合很大时，关键字值不同的元素可能会映象到哈希表的同一地址上，这样的现象称为哈希冲突。目前常用的解决哈希冲突的方法如下：

开放定址法：当发生地址冲突时，按照某种方法继续探测哈希表中的其他存储单元，直到找到空位置为止。

再哈希法：当发生哈希冲突时使用另一个哈希函数计算地址值，直到冲突不再发生。这种方法不易产生聚集，但是增加计算时间，同时需要准备许多哈希函数。

链地址法：将所有哈希值相同的 Key 通过链表存储。key 按顺序插入到链表中

建立公共溢出区：采用一个溢出表存储产生冲突的关键字。如果公共溢出区还产生冲突，再采用处理冲突方法处理。

5、请你说一说哈希冲突的解决方法

考察点：hash 冲突，数据结构

1、开放定址

开放地址法有个非常关键的特征，就是所有输入的元素全部存放在哈希表里，也就是说，位桶的实现是不需要任何的链表来实现的，换句话说，也就是这个哈希表的装载因子不会超过 1。它的实现是在插入一个元素的时候，先通过哈希函数进行判断，若是发生哈希冲突，就以当前地址为基准，根据再寻址的方法（探查序列），去寻找下一个地址，若发生冲突再去寻找，直至找到一个为空的地址为止。所以这种方法又称为再散列法。

有几种常用的探查序列的方法：

①线性探查

dii=1，2，3，…，m-1；这种方法的特点是：冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。

②二次探查

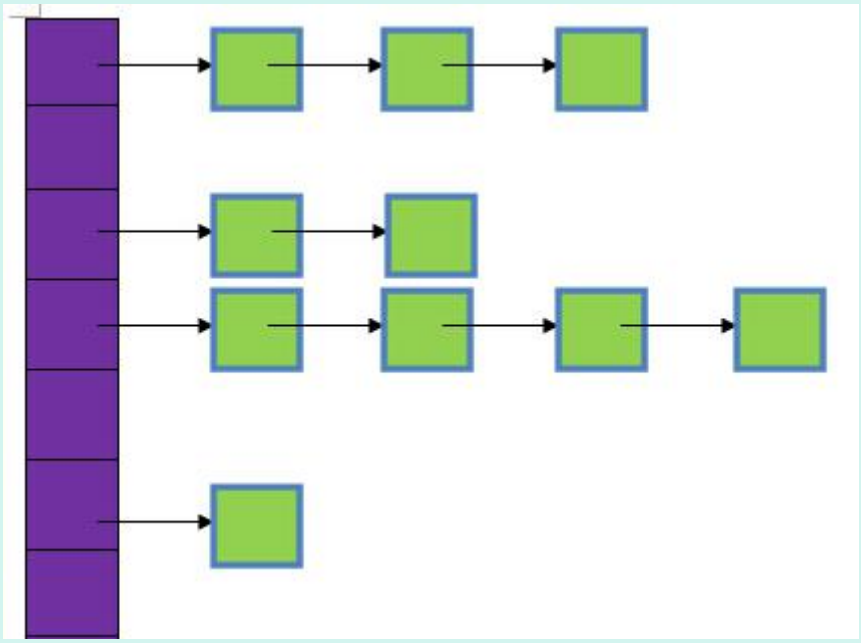
di=12，-12，22，-22，…，k2，-k2　　（ k<=m/2 ）；这种方法的特点是：冲突发生时，在表的左右进行跳跃式探测，比较灵活。

③　伪随机探测

di=伪随机数序列；具体实现时，应建立一个伪随机数发生器，（如 i=(i+p) % m），生成一个位随机序列，并给定一个随机数做起点，每次去加上这个伪随机数++就可以了。

2、链地址

每个位桶实现的时候，采用链表或者树的数据结构来去存取发生哈希冲突的输入域的关键字，也就是被哈希函数映射到同一个位桶上的关键字。



紫色部分即代表哈希表，也称为哈希数组，数组的每个元素都是一个单链表的头节点，链表是用来解决冲突的，如果不同的 key 映射到了数组的同一位置处，就将其放入单链表中，即链接在桶后。

3、公共溢出区

建立一个公共溢出区域，把 hash 冲突的元素都放在该溢出区里。查找时，如果发现 hash 表中对应桶里存在其他元素，还需要在公共溢出区里再次进行查找。

4、再 hash

再散列法其实很简单，就是再使用哈希函数去散列一个输入的时候，输出是同一个位置就再次散列，直至不发生冲突位置。

缺点：每次冲突都要重新散列，计算时间增加。

6、动态规划

1、请你手写代码：最长公共连续子序列

参考回答：

```
int substr(string & str1, string &str2)
{
	int len1 = str1.length();
	int len2 = str2.length();
	vector<vector<int>>dp(len1,vector<int>(len2,0));
	for (int i = 0; i < len1; i++)
	{
		dp[i][0] = str1[i]==str1[0]?1:0;
	}
	for (int j = 0; j <= len2; j++)
	{
		dp[0][j] = str1[0]==str2[j]?1:0;
	}
	for (int i = 1; i < len1; i++)
	{
		for (int j = 1; j < len2; j++)
		{
			if (str1[i] == str2[j])
			{
				dp[i][j] = dp[i - 1][j - 1]+1;
			}
		}
```

```
        }
        int longest = 0;
        int longest_index = 0;
        for (int i = 0; i < len1; i++)
        {
                for (int j = 0; j < len2; j++)
                {
                        if (longest < dp[i][j])
                        {
                                longest = dp[i][j];
                                longest_index = i;
                        }
                }
        }
        //字符串为从第 i 个开始往前数 longest 个
        for (int i = longest_index-longest+1; i <=longest_index; i++)
        {
                cout << str1[i] << endl;
        }
        return longest;
}
```

2、手写代码：求一个字符串最长回文子串

考点：动态规划

参考回答：

```
int LongestPalindromicSubstring(string & a)
```

```
{
        int len = a.length();
        vector<vector<int>>dp(len, vector<int>(len, 0));
        for (int i = 0; i < len; i++)
        {
                dp[i][i] = 1;
        }
        int max_len = 1;
        int start_index = 0;
        for (int i= len - 2; i >= 0; i--)
        {
                for (int j = i + 1; j < len; j++)
                {
                        if (a[i] == a[j])
                        {
                                if (j - i == 1)
                                {
                                        dp[i][j] = 2;

                                }
                                else
                                {
                                        if (j - i > 1)
                                        {
                                                dp[i][j] = dp[i + 1][j - 1] + 2;
```

```
                                        }
                                }
                                if (max_len < dp[i][j])
                                {
                                        max_len =   dp[i][j];
                                        start_index = i;
                                }
                        }
                        else
                        {
                                dp[i][j] = 0;
                        }
                }
        }
        cout << "max len is " << max_len << endl;
        cout << "star index is" << start_index << endl;
        return max_len;
}
```

3、手写代码：查找最长回文子串

考点：动态规划

参考回答：

```
int LongestPalindromicSubstring(string & a)
{
int len = a.length();
vector<vector<int>>dp(len, vector<int>(len, 0));
```

```
for (int i = 0; i < len; i++)
{
    dp[i][i] = 1;
}
int max_len = 1;
int start_index = 0;
for (int i= len - 2; i >= 0; i--)
{
    for (int j = i + 1; j < len; j++)
    {
        if (a[i] == a[j])
        {
            if (j - i == 1)
            {
                dp[i][j] = 2;
            }
            else
            {
                if (j - i > 1)
                {
                    dp[i][j] = dp[i + 1][j - 1] + 2;
                }
            }
            if (max_len < dp[i][j])
            {
```

```
                max_len =  dp[i][j];
                start_index = i;
            }
        }
        else
        {
            dp[i][j] = 0;
        }
    }
}
cout << "max len is " << max_len << endl;
cout << "star index is" << start_index << endl;
return max_len;
}
```

7、链表

1、请你手写代码，如何合并两个有序链表

考点：链表合并

参考回答：

```
class Solution {
public:
     ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if(l1 == NULL)
        {
```

```
            return l2;
        }
        if(l2 == NULL)
        {
             return l1;
        }

        if(l1->val < l2->val)
        {
            l1->next=mergeTwoLists(l1->next,l2);
            return l1;
        }
        else
        {
            l2->next=mergeTwoLists(l1,l2->next);
            return l2;
        }
    }
};
```

2、手写代码：反转链表

参考回答：

```
Void reversal_list(mylist * a_list)
```

```
{
    mylist * forward_node = nullptr;
    mylist * cur_node = a_list->next;
    mylist* next_node = cur_node->next;
    if(cur_node == nullptr)
    {
        return ;
    }
    while(1)
    {
        cur_node->next = forward_node;
        forward_node = cur_node;
        cur_node = next_node;
        if(cur_node == nullptr)
        {
            break;
        }
        next_node = cur_node->next;
    }
    a_list->next = forward_node;
}
```

3、判断一个链表是否为回文链表，说出你的思路并手写代码

参考回答：

思路：使用栈存储链表前半部分，然后一个个出栈，与后半部分元素比较，如果链表长度未知，可以使用快慢指针的方法，将慢指针指向的元素入栈，然后如果快指针指向了链表尾部，此时慢指针指向了链表中间

```
bool is_palindromic_list2(mylist *a_list)
{
    if(a_list == nullptr)
    {
        return false;
    }
    stack<int>list_value;
    mylist * fast =a_list;
    mylist *slow =a_list;
    while(fast->next!=nullptr && fast->next->next!=nullptr)
    {
        list_value.push(slow->next->value);
        slow = slow->next;
        fast = fast->next->next;
    }
     cout<<"middle elem value is "<<slow->next->value<<endl;
    if(fast->next != nullptr)
    {
        cout<<"the list has odd num of node"<<endl;
        slow =slow->next;
    }

    int cur_value;
```

```
        while(!list_value.empty())
        {
            cur_value = list_value.top();
            cout<<"stack top value is"<<cur_value<<endl;
            cout<<"list value is "<<slow->next->value<<endl;

            if(cur_value != slow->next->value)
            {
                return false;
            }
            list_value.pop();
            slow = slow->next;
        }
        return true;
    }
```

4、请你手写链表反转

```
    参考回答：
    struct ListNode {
    int val;
    struct ListNode *next;
    ListNode(int x) :val(x), next(NULL) {}
    }
    ListNode* ReverseList(ListNode* pHead)
    {
```

```
        if(!pHead||!pHead->next)return pHead;
        ListNode *pre=nullptr;
        ListNode *p=pHead;
        ListNode *next=pHead->next;
        while(p)
        {
            p->next=pre;
            pre=p;
            p=next;
            if(next)
                next=next->next;
        }
        return pre;
    }
```

5、什么是单向链表，如何判断两个单向链表是否相交

参考回答：

1、单向链表

单向链表（单链表）是链表的一种，其特点是链表的链接方向是单向的，对链表的访问要通过顺序读取从头部开始；链表是使用指针进行构造的列表；又称为结点列表，因为链表是由一个个结点组装起来的；其中每个结点都有指针成员变量指向列表中的下一个结点。

列表是由结点构成，head 指针指向第一个成为表头结点，而终止于最后一个指向 nuLL 的指针。

2、判断两个链表是否相交

1）方法 1：

链表相交之后，后面的部分节点全部共用，可以用 2 个指针分别从这两个链表头部走到尾部，最后判断尾部指针的地址信息是否一样，若一样则代表链表相交！

2）方法 2：

可以把其中一个链表的所有节点地址信息存到数组中，然后把另一个链表的每一个节点地址信息遍历数组，若相等，则跳出循环，说明链表相交。进一步优化则是进行 hash 排序，建立 hash 表。

7、高级算法

1、如果让你做自然语言理解（NLU），在处理语言规则的时候大概会用到什么算法

考察点：自然语言处理

参考回答：

略

2、请问加密方法都有哪些

考察点：密码学

1、单向加密

单向加密又称为不可逆加密算法，其密钥是由加密散列函数生成的。单向散列函数一般用于产生消息摘要，密钥加密等，常见的有：

MD5（Message Digest Algorithm 5）：是 RSA 数据安全公司开发的一种单向散列算法，非可逆，相同的明文产生相同的密文；

SHA（Secure Hash Algorithm）：可以对任意长度的数据运算生成一个 160 位的数值。其变种由 SHA192，SHA256，SHA384 等；

CRC-32，主要用于提供校验功能；

算法特征：

输入一样，输出必然相同；

雪崩效应，输入的微小改变，将会引起结果的巨大变化；

定长输出，无论原始数据多大，结果大小都是相同的；

不可逆，无法根据特征码还原原来的数据；

2、对称加密

采用单钥密码系统的加密方法，同一个密钥可以同时用作信息的加密和解密，这种加密方法称为对称加密，也称为单密钥加密。

特点：

1、加密方和解密方使用同一个密钥；

2、加密解密的速度比较快，适合数据比较长时的使用；

3、密钥传输的过程不安全，且容易被破解，密钥管理也比较麻烦；

优点：对称加密算法的优点是算法公开、计算量小、加密速度快、加密效率高。

缺点：对称加密算法的缺点是在数据传送前，发送方和接收方必须商定好秘钥，然后使双方都能保存好秘钥。其次如果一方的秘钥被泄露，那么加密信息也就不安全了。另外，每对用户每次使用对称加密算法时，都需要使用其他人不知道的唯一秘钥，这会使得收、发双方所拥有的钥匙数量巨大，密钥管理成为双方的负担。

3、非对称加密

非对称密钥加密也称为公钥加密，由一对公钥和私钥组成。公钥是从私钥提取出来的。可以用公钥加密，再用私钥解密，这种情形一般用于公钥加密，当然也可以用私钥加密，用公钥解密。常用于数字签名，因此非对称加密的主要功能就是加密和数字签名。

特征：

1）秘钥对，公钥(public key)和私钥(secret key)

2）主要功能：加密和签名

发送方用对方的公钥加密，可以保证数据的机密性（公钥加密）。

发送方用自己的私钥加密，可以实现身份验证（数字签名）。

3）公钥加密算法很少用来加密数据，速度太慢，通常用来实现身份验证。

常用的非对称加密算法

RSA：由 RSA 公司发明，是一个支持变长密钥的公共密钥算法，需要加密的文件块的长度也是可变的；既可以实现加密，又可以实现签名。

DSA（Digital Signature Algorithm）：数字签名算法，是一种标准的 DSS（数字签名标准）。

ECC（Elliptic Curves Cryptography）：椭圆曲线密码编码。

3、什么是 LRU 缓存

参考回答：

LRU(最近最少使用)算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高
实现：使用一个链表保存缓存数据，将新数据插入到头部，每当缓存命中时，则将命中的数据移动到链表头部，当链表满的时候，将链表尾部的数据丢弃。

4、请你说一说洗牌算法

参考回答：

1、Fisher-Yates Shuffle 算法

最早提出这个洗牌方法的是 Ronald A. Fisher 和 Frank Yates，即 Fisher-Yates Shuffle，其基本思想就是从原始数组中随机取一个之前没取过的数字到新的数组中，具体如下：

1）初始化原始数组和新数组，原始数组长度为 n(已知)。

2）从还没处理的数组（假如还剩 k 个）中，随机产生一个[0, k)之间的数字 p（假设数组从 0 开始）。

3）从剩下的 k 个数中把第 p 个数取出。

4）重复步骤 2 和 3 直到数字全部取完。

5）从步骤 3 取出的数字序列便是一个打乱了的数列。

时间复杂度为 O(n*n)，空间复杂度为 O(n)。

2）Knuth-Durstenfeld Shuffle

Knuth 和 Durstenfeld 在 Fisher 等人的基础上对算法进行了改进，在原始数组上对数字进行交互，省去了额外 O(n)的空间。该算法的基本思想和 Fisher 类似，每次从未处理的数据中随机取出一个数字，然后把该数字放在数组的尾部，即数组尾部存放的是已经处理过的数字。

算法步骤为：

1. 建立一个数组大小为 n 的数组 arr，分别存放 1 到 n 的数值；

2. 生成一个从 0 到 n - 1 的随机数 x；

3. 输出 arr 下标为 x 的数值，即为第一个随机数；

4. 将 arr 的尾元素和下标为 x 的元素互换；

5. 同 2，生成一个从 0 到 n - 2 的随机数 x；

6. 输出 arr 下标为 x 的数值，为第二个随机数；

7. 将 arr 的倒数第二个元素和下标为 x 的元素互换；

……

如上，直到输出 m 个数为止

时间复杂度为 O(n)，空间复杂度为 O(1)，缺点必须知道数组长度 n。

8、字符串

1、给你一个字符串，找出第一个不重复的字符，如“abbbabcd”，则第一个不重复就是 c

参考回答：

使用哈希的思想，建立 256 个 bool 数组 array，初始都为 false,从头开始扫描字符串，扫到一个，将以其 ascii 码为下标的元素置 true。例如扫描到 A 的时候，执行：array['A']=true。第二边扫描，扫到一个字母就以其 ascii 码为下标，去 array 数组中看其值，如果是 true,返回改字母，如果是 false，继续扫描下一个字母。

六、项目相关

1、请你回答一下 git 中 Merge 和 rebase 区别

考点：git

参考回答：

Merge 会自动根据两个分支的共同祖先和两个分支的最新提交 进行一个三方合并，然后将合并中修改的内容生成一个新的 commit,即 merge 合并两个分支并生成一个新的提交,并且仍然后保存原来分支的 commit 记录

Rebase 会从两个分支的共同祖先开始提取当前分支上的修改，然后将当前分支上的所有修改合并到目标分支的最新提交后面，如果提取的修改有多个，那 git 将依次应用到最新的提交后面。Rebase 后只剩下一个分支的 commit 记录

七、设计模式

1、请问你用过哪些设计模式，介绍一下单例模式的多线程安全问题

参考回答：

常见的设计模式如下：

单例模式：单例模式主要解决一个全局使用的类频繁的创建和销毁的问题。单例模式下可以确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。单例模式有三个要素：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。

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

观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

单例模式的多线程安全问题：

在单例模式的实现中，如果不采取任何措施，在多线程下是不安全的，可能会同时创建多个实例。因此，为了保证单例模式在多线程下的线程安全，一般采用下面几种方式实现单例模式：

1)饿汉式：基于 class loader 机制避免多线程的同步问题，不过，instance 在类装载时就实例化，可能会产生垃圾对象。

```
public class Singleton {
    private static Singleton sin=new Singleton();    ///直接初始化一个实例对象
    private Singleton(){    ///private类型的构造函数，保证其他类对象不能直接new一个该对象的实例
    }
    public static Singleton getSin(){    ///该类唯一的一个public方法
        return sin;
    }
}
```

2)懒汉式：通过双重锁机制实现线程安全。

```
public class Singleton {
     private static Singleton instance;
     private Singleton (){
     }
     public static Singleton getInstance(){     //对获取实例的方法进行同步
       if (instance == null){
           synchronized(Singleton.class){
               if (instance == null)
                   instance = new Singleton();
           }
       }
       return instance;
     }

 }
```

2、请问你了解哪些设计模式？

参考回答：

常见的设计模式如下：

单例模式：单例模式主要解决一个全局使用的类频繁的创建和销毁的问题。单例模式下可以确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。单例模式有三个要素：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。

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

观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

装饰器模式：对已经存在的某些类进行装饰，以此来扩展一些功能，从而动态的为一个对象增加新的功能。装饰器模式是一种用于代替继承的技术，无需通过继承增加子类就能扩展对象的新功能。使用对象的关联关系代替继承关系，更加灵活，同时避免类型体系的快速膨胀。

3、请问如何保证单例模式只有唯一实例？你知道的都有哪些方法？

参考回答：

单例的实现主要是通过以下两个步骤：

将该类的构造方法定义为私有方法，这样其他处的代码就无法通过调用该类的构造方法来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例；

在该类内提供一个静态方法，当我们调用这个方法时，如果类持有的引用不为空就返回这个引用，如果类保持的引用为空就创建该类的实例并将实例的引用赋予该类保持的引用。

单例模式的实现主要有两种一种是饿汉式，一种是懒汉式。饿汉式线程安全的单例模式如下：

```
public class Singleton {

    private final static Singleton INSTANCE = new Singleton();

    private Singleton(){}

    public static Singleton getInstance(){
        return INSTANCE;
    }
}
```

懒汉式线程安全的单例模式如下

```
public class Singleton {

    private static volatile Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

4、请你说一说 OOP 的设计模式的五项原则

参考回答：

1、单一职责原则

单一职责有 2 个含义，一个是避免相同的职责分散到不同的类中，另一个是避免一个类承担太多职责。减少类的耦合，提高类的复用性。

2、接口隔离原则

表明客户端不应该被强迫实现一些他们不会使用的接口，应该把胖接口中额方法分组，然后用多个接口代替它，每个接口服务于一个子模块。简单说，就是使用多个专门的接口比使用单个接口好很多。

该原则观点如下：
1）一个类对另外一个类的依赖性应当是建立在最小的接口上

2）客户端程序不应该依赖它不需要的接口方法。

3、开放-封闭原则

open 模块的行为必须是开放的、支持扩展的，而不是僵化的。

closed 在对模块的功能进行扩展时，不应该影响或大规模影响已有的程序模块。一句话概括：一个模块在扩展性方面应该是开放的而在更改性方面应该是封闭的。

核心思想就是对抽象编程，而不对具体编程。

4、替换原则

子类型必须能够替换掉他们的父类型、并出现在父类能够出现的任何地方。

主要针对继承的设计原则

1）父类的方法都要在子类中实现或者重写，并且派生类只实现其抽象类中生命的方法，而不应当给出多余的,方法定义或实现。

2）在客户端程序中只应该使用父类对象而不应当直接使用子类对象，这样可以实现运行期间绑定。

5、依赖倒置原则

上层模块不应该依赖于下层模块，他们共同依赖于一个抽象，即：父类不能依赖子类，他们都要依赖抽象类。

抽象不能依赖于具体，具体应该要依赖于抽象。

5、请你说说工厂模式的优点？

参考回答：

解耦，代码复用，更改功能容易。

6、请你说一下观察者模式

参考回答：

观察者模式中分为观察者和被观察者，当被观察者发生装填改变时，观察者会受到通知。主要为了解决对象状态改变给其他对象通知的问题，其实现类似于观察者在被观察者那注册了一个回调函数。

7、请你介绍一下单例模式

参考回答：

C++的实现有两种，一种通过局部静态变量，利用其只初始化一次的特点，返回对象。另外一种，则是定义全局的指针，getInstance 判断该指针是否为空，为空时才实例化对象

8、单例模式中的懒汉加载，如果并发访问该怎么做？

参考回答：

使用锁机制，防止多次访问,可以这样，第一次判断为空不加锁，若为空，再进行加锁判断是否为空，若为空则生成对象。

9、装饰器模式和单例模式，使用单例模式应该注意什么

考察点：设计模式

参考回答：

1、装饰器模式

装饰器模式主要是为了动态的为一个对象增加新的功能，装饰器模式是一种用于代替继承的技术，无需通过继承增加子类就能扩展对象的新功能。这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。使用对象的关联关系代替继承关系，更加灵活，同时避免类型体系的快速膨胀。

优点：装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

缺点：多层装饰比较复杂。

使用场景：1、扩展一个类的功能。 2、动态增加功能，动态撤销。

2、单例模式

单例模式是一种常用的软件设计模式，其定义是单例对象的类只能允许一个实例存在。这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

优点： 1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。 2、避免对资源的多重占用（比如写文件操作）。

缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

使用场景： 1、要求生产唯一序列号。 2、WEB 中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。 3、创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接等。

实现：

单例模式要求类能够有返回对象一个引用(永远是同一个)和一个获得该实例的方法（必须是静态方法，通常使用 getInstance 这个名称）。

单例的实现主要是通过以下两个步骤：

将该类的构造方法定义为私有方法，这样其他处的代码就无法通过调用该类的构造方法来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例；

在该类内提供一个静态方法，当我们调用这个方法时，如果类持有的引用不为空就返回这个引用，如果类保持的引用为空就创建该类的实例并将实例的引用赋予该类保持的引用。

注意事项：getInstance() 方法中需要使用同步锁 synchronized (Singleton.class) 防止多线程同时进入造成 instance 被多次实例化。

3）单例模式举例（常手撕）

```
class SingleTon

{

private:

SingleTon() {};

SingleTon(const SingleTon&) {}; // 禁止拷贝

SingleTon& operator=(const SingleTon&) {}; // 禁止赋值



static T* uniqueInstance;

static pthread_mutex_t mutex;
```

```
public:
static T* GetInstance()
{
    pthread_mutex_lock(&mutex);
    if (uniqueInstance == nullptr)
    {
        uniqueInstance = new T();
    }
    pthread_mutex_unlock(&mutex);
    return uniqueInstance;
}
};
template <class T>
pthread_mutex_t SingleTon<T>::mutex = PTHREAD_MUTEX_INITIALIZER;
template <class T>
T* SingleTon<T>::uniqueInstance = nullptr;
```

八、场景题

1、给你两个球，100 层楼，每个球在一定高度扔下去会碎，怎么用最少的次数给判断是几层楼能把求摔碎？

参考回答：

略

九、分布式与架构

1、分布式缓存和分布式存储的设计

参考回答：略

十、惊喜福利

此面试题库将根据当下面试形式大数据随时更新，如果你已获得下载权限，那么你可以终身在牛币兑换中心里去兑换此面试题库的电子版，如果电子版有更新，会通过牛客站内信进行通知（前提是你已获得下载权限）。

牛币兑换中心：https://www.nowcoder.com/coin/index
还能兑换各种惊喜周边哦