Part one —— C/C++

【1】静态变量和全局变量的区别 —— 都放在静态存储区
作用域不同——全局变量只要在一个源文件中定义，就可以配合extern声明作用于**所有的源文件**；而静态变量只初始化一次

****引申——将局部变量改为静态变量，改变的是变量的存储方式（也就是生存期，从函数执行期间到程序运行期间）。
将全局变量修改为静态全局变量改变的是变量的作用域，限制变量的使用范围。

静态过程变量（在函数中定义的static变量）——也是存放在静态区（在.bss区或者是.data区），并且在符号表中创建一个有唯一名字的本地链接器符号

```
int f()
{
    static int x = 0;
    return x;
}
int g()
{
    static int x = 0;
    return x;
}
```

实际会在静态区出现**x.1和x.2**对两个静态变量加以区分！！！
总之任何带有static属性声明的全局变量或者函数都是**模块私有**的~~~~就跟private和public修饰一样

【2】C++的封装、继承和多态
（1）封装的思想——把属性和方法合成一个整体，封装了成员变量和成员函数，同时又可以**隐藏一些成员，降低类之间的耦合**程度。
（2）继承 —— 子类得以**继承父类的各种属性和方法**，不用写相同的代码， 同时**又可以重新定义某些属性和方法**，使子类获得一些与父类不同的功能
（3）多态 —— 同一个调用语句在父类和子类间穿梭时具有不同的表现形式，可以用同一段代码处理不同类型的对象，极大的提高了代码的重用。—— 扩展到grpc源码（使用抽象类作为公共接口）、**stl源码**（**静态多态的典范**，使用偏特化的思想为合适的数据类型选择最快的实现方式）

【3】多态——静态多态和动态多态
（1）静态多态——编译期间确定程序执行哪一个函数 —— 偏特化、全特化，STL中大量用到了静态多态提供**泛型设计！！！**的思想
实现方式：**模板+函数重载**
几个类之间可以没有关系，但要有共同的接口 —— 例如：多个形状类，都有公共的draw接口 那么就可以做一个模板，在模板中去调用draw方法
为什么是编译时绑定——所有的模板都需要在用之前产生一个实例（编译时完成），所以在产生实例时就是解析了多态该执行哪个方法，效率比较高

```
class Line
{
public:
    void pp()
    {
        cout << "Line" << endl;
    }
};

class Circle
{
public:
    void pp()
    {
        cout << "Circle" << endl;
    }
};

template<typename Geometry>
void RunPp(Geometry &geo)
{
    geo.pp();
}
```

（2）动态多态 —— 虚函数

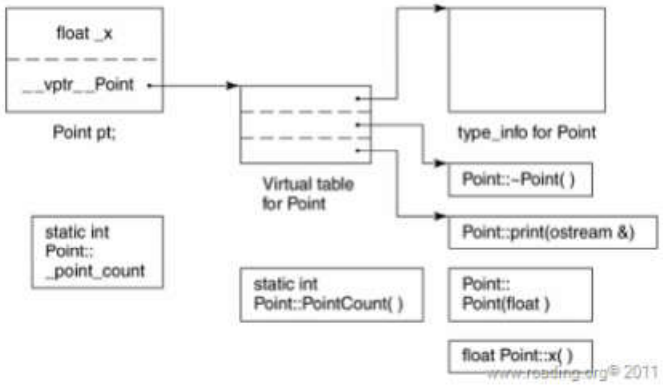
【4】虚析构函数的作用
通过父类指针，可以释放子类以及父类的资源

【5】C++中的类型转换
（1）static_cast和dynamic_cast——放在一起是为了对比　其中static是编译时期的静态类型检测，dynamic是运行时检测真实调用对象类型

static_cast可以完成 **基础数据类型**转换、**同一个继承体系**中的（**指针或引用**）类型转换、任意类型指针和void *的转换
dynamic_cast是RTTI 操作符，提供运行时检查，如果不能转换，返回NULL

（2）reinterpret_cast —— 仅仅是比特位的拷贝，重新解释
（3）const_cast —— 去const属性 —— 为了函数调用而出现了，让const对象能够作为参数，**调用non_const**参数的函数（该函数中不改变对象的值）

【6】C++对象模型



C++的对象模型

所有的**非静态数据成员**存储在对象本身中， 而所有的**静态数据成员和成员函数**都位于对象之外。 另用一张**虚函数表**存储所有**指向虚函数的指针**，在**表头位置**附加该类的**type_info对象用于RTTI**，对象中只需要保存一根指向虚函数表的指针

所以一个类对象的大小包括：1、非静态数据成员 2、内存对齐 3、因支持virtual而产生的开销

【7】虚函数和纯虚函数的区别
虚函数——实现运行时多态，为子类提供了默认的函数实现，子类可以**重写**实现特殊化。**提供接口声明和默认实现**，子类可以重写。
纯虚函数——包含纯虚函数的是抽象类，**不能new对象**，只有实现了纯虚函数的子类才可以new对象。**提供接口的声明，并且强制子类实现**。

【8】Volatile关键字的作用 —— 声明的变量，编译器对访问该变量的代码不再进行优化，每次访问该变量时，总是从**所在的内存中去读取**数据。
声明形式 —— int volatile num

多线程下的volatile —— 当两个线程都会用到一个变量，且该变量会被改变时，用volatile声明该变量 —— **防止优化编译器把变量从内存装入寄存器中**，强制要求程序从内存中真正取出，而不是使用寄存器中的值。

【9】#define和typedef的区别： 两点：处理方式（预处理器和编译器）、效果（机械替换和声明新类型）
（1）define是由预处理器处理，执行的是机械的替换，不做正确性检查，只有在编译是才会报错。
（2）typedef由编译器处理，在作用域内给一个已经存在的类型起一个别名，用来声明一种类型。

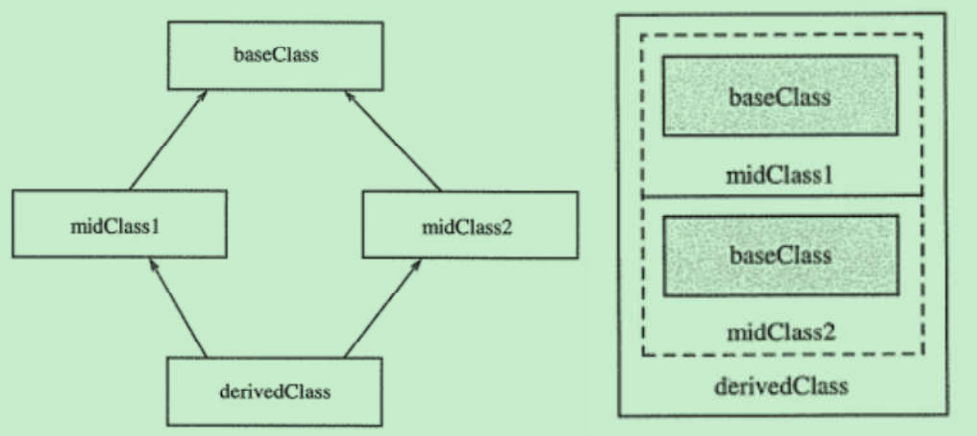
**********区别：
#define int_ptr int *
int_ptr a,b; —— 执行宏替换以后， int *a,b; 相当于声明一个指针a和整型b，因为不是一个新类型别名

typedef int* int_ptr;
int_ptr a,b; —— 声明了两个整型指针a和b

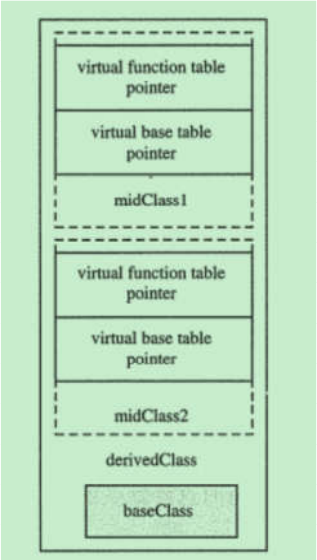
区别同样表现在const上：

```
typedef int * pint ;
#define PINT int *

那么：
const pint p ;//p不可更改，但p指向的内容可更改
const PINT p ;//p可更改，但是p指向的内容不可更改。

pint是一种指针类型 const pint p 就是把指针给锁住了 p不可更改
而const PINT p 是const int * p 锁的是指针p所指的对象。
```

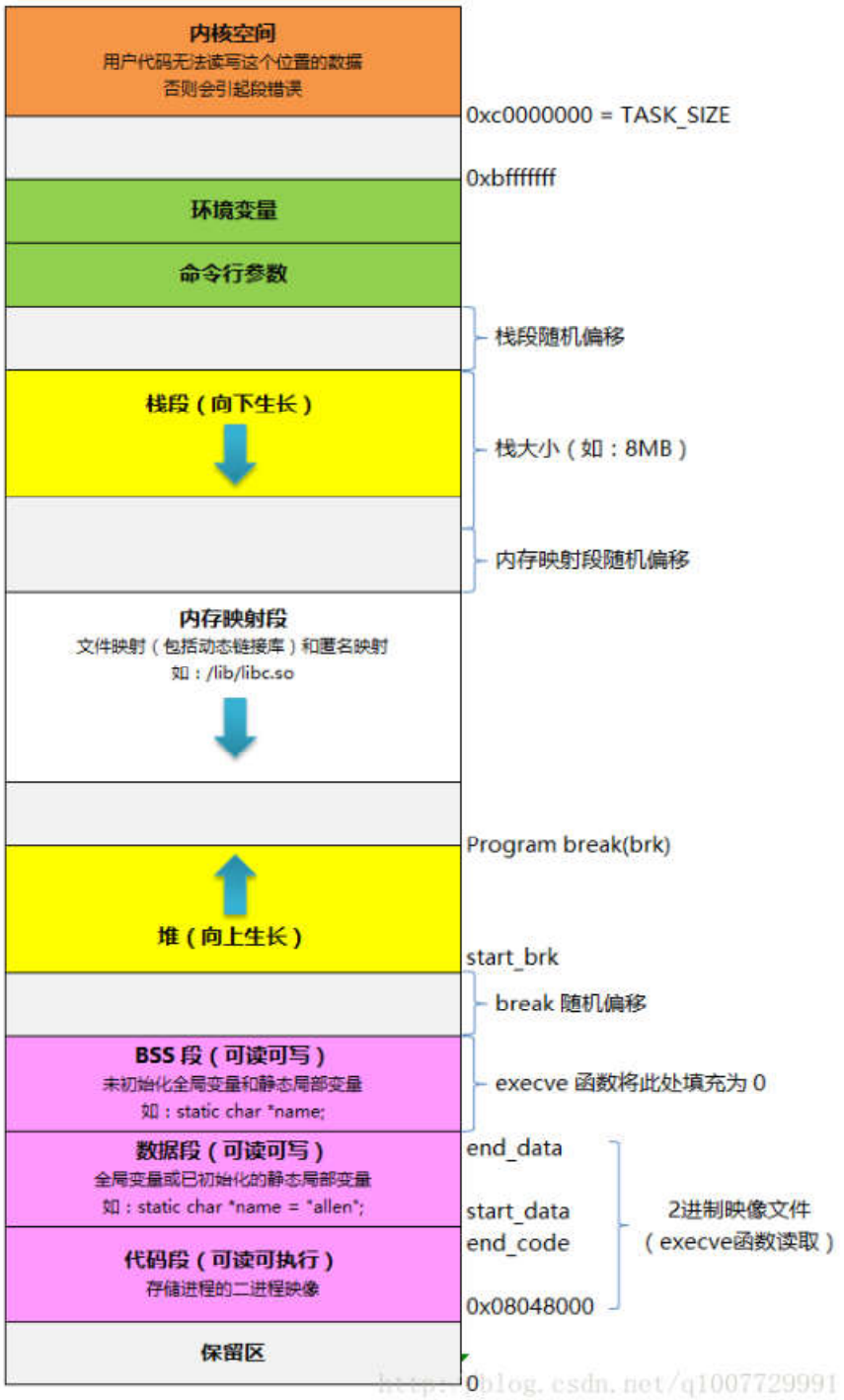
【10】菱形继承 —— 产生的二义性问题： 基类实际上会在中间类中各存在一个实例，所以使用基类中的成员变量a时，要指定B::a，还是C::a否则就会不明确
既造成调用困难，又给内存带来不必要的消耗



解决方法：虚继承 —— 只有一个基类实例，在B和C类中各有一张虚函数表， 可以消除二义性，增加的开销只是虚函数指针



【11】可执行程序的进程空间



【12】NULL和nullptr的区别： nullptr相对于NULL的优越性 —— 避免函数重载时的二义性 —— fun(int a) fun(char *a) 在fun(NULL)时，NULL是0，调用int参数的重载函数，而fun(nullptr)则会调用fun(char *a)重载函数
NULL在C++中是**整数0**（在C中是void * 0）
nullptr是C++11中引入的一个字面值常量 std::nullptr， 是一个void *型的指针，可以转换成任意类型的指针

【13】extern C的作用：C动态库制作时考虑到可能被C++调用，需要在接口头文件上添加externC
*****但是C又不支持extern C，那么是否要为C和C++各准备一套对应的头文件呢？？？如memset，这个C语言的函数

```
#ifdef  __cplusplus
extern "C" {
#endif
void *memset(void *,int,size_t);

#ifdef  __cplusplus
}
#endif
```

深入探究所谓的**符号重定义**：
在**链接**期间，多个目标文件中含有**相同名字**的**全局符号的定义**，这就是一个符号重定义的问题（也就是存在**多个同名强符号**）

弱符号之间出现类型不一致的问题怎么办？？？ —— 参考了Fortran语言Common block机制（语言不支持动态内存分配，不同目标文件申明的临时空间大小不一致时，以最大的为准），这里也一样，多个弱符号出现类型不一致时，以**占空间最大的**那个为准
——最本质的原因：**链接器LD不支持符号类型**，即并没有存储符号的类型

不同**编译器**编译出来的**目标文件**，可以被链接成可执行文件吗？？？
如果要链接，首先是链接器要支持不同编译器产生的**目标文件格式**；其次，两个目标文件还需要有相同的符号修饰标准、变量内存布局和函数调用方式（这些统称为**ABI**，也就是**二进制兼容性**）
【14】迭代器失效问题：
（1）vector：如果插入引起内存重新分配，则所有迭代器失效；如果没有重新分配，则被插入位置之后的迭代器失效； 删除操作会导致，被删除元素后的迭代器失效。（主要按照是否改变内存中的位置来看！！！！）
（2）deque：对于插入/删除在非首尾的任何位置，迭代器都会失效；如果是在首尾插入/删除，则只有指向该元素的迭代器失效（同样是内存中的位置是否移动！！）
（3）List和map等，只有删除该元素，会导致该元素的迭代器失效

【15】手写实现shared_ptr —— **面试时的核心点：保存了两根指针**，一根指针指向对象本身，另一根指针指向对象的引用计次（注意交叉引用问题即可）

```
template<typename T>
class SharedPtr{
public:
    SharedPtr( ): _ptr((T *)0), _refCount(0)
    {   }
    SharedPtr(T *obj) : _ptr(obj), _refCount(new int(1))
    {   }
    SharedPtr(SharedPtr &other): _ptr(other._ptr),_refCount(&(++*other._refCount))
    {   }
    ~SharedPtr()
    {
        if(_ptr && --*_refCount == 0)
        {
            delete _ptr;
            _ptr = NULL;
            delete _refCount;
            _refCount = NULL;
        }
    }

    SharedPtr &operator=(SharedPtr &other)
    {
        if(this == &other)
            return *this;
        ++*other._refCount;
        if(--*_refCount == 0)
        {
            delete _ptr;
            _ptr = NULL;
            delete _refCount;
            _refCount = NULL;
        }
        _ptr = other._ptr;
        _refCount = other._refCount;
        return *this;
    }

    T *operator->()
    {
        if(_refCount == 0)
            return NULL;
        return _ptr;
    }

    T &operator*()
    {
        if(_refCount == 0)
```

```
            return (T *) 0;
        return _ptr;
    }
pri  ate:
    T *_ptr;
    int *_refCount;
}
```

**********为什么shared_ptr的引用计次要放在堆上？？？
shard_ptr作为智能指针，解决的是多个指针指向同一个对象时候的内存管理问题，多个智能指针本身作为多个对象，要对同一个引用计次进行管理，且用原子操作保证互斥访问，那么引用计次必然就只是一份，所以当然要new 放在堆上啦

【16】union的字节对齐： sizeof(A) = 24 虽然union所占的空间是按照最大的5*4=20，但是字节对齐方式是按照最大的double（8字节）进行对齐，所以是24

```
union A{
    int a[5];
    char b;
    double c;
};
```

【17】智能指针的循环引用问题 —— shared_ptr造成的自引用问题，或者两个类之间的相互引用，导致引用次数增加
解决方法 —— 将其中一个智能指针换成weak_ptr（不会修改引用计数）

```
class A
{
public:
    shared_ptr<B> m_b;
};

class B
{
public:
    shared_ptr<A> m_a;
};

int main()
{
    while (true)
    {
        shared_ptr<A> a(new A); //new出来的A的引用计数此时为1
        shared_ptr<B> b(new B); //new出来的B的引用计数此时为1
        a->m_b = b; //B的引用计数增加为2
        b->m_a = a; //A的引用计数增加为2
    }
```

【18】右值引用 —— https://blog.csdn.net/li1914309758/article/details/81663488
（1）通过右值引用，右值重获新生，生命周期和右值引用变量的生命周期一样长
T &&t = getVar( )，getVar()产生的临时值是一个右值，生命周期被延长
（2）右值引用独立于右值和左值，右值引用类型的变量可以是右值，也可以是左值。他是右值还是左值取决于它的初始化

```cpp
template<typename T>
void f(T&& t){}

f(10); //t是右值

int x = 10;
f(x); //t是左值
```

【19】主流编译器下，int和long都占4个字节，不管是32位还是64位

【20】为什么声明常放在.h中，而实现常放在.cpp中
要避免在头文件中定义函数，否则会引发重定义的问题
https://www.cnblogs.com/magicsoar/p/3702365.html

【21】预处理 —— 编译 —— 链接过程：
https://www.cnblogs.com/magicsoar/p/3760201.html

【22】this指针存在哪？？？？

的。那么C++中类成员函数是如何知道哪个对象调用了它？并正确显示调用它的对象的数据呢？ 很多书上也都已经介绍了，当一个对象调用某成员函数时会隐式传入一个参数， 这个参数就是this指针。this指针中存放的就是这个对象的首地址。这和C中通过向函数传递结构体变量的地址是不是很像？！只是传参形式不一样罢了！ 在C中我们是手工把结构体变量和函数关联起来的，而C++则是编译器帮我们把类数据和成员函数关联起来的并通过名称粉碎和编译时检查等形式防止外部的任意访问。那么这个this指针存放在哪里呢？其实编译器在生成程序时加入了获取对象首地址的相关代码，并把获取的首地址存放在了寄存器ECX中(VC++编译器是放在ECX中，其它编译器有可能不同)。也就是成员函数的其它参数正常都是存放在栈中。而this指针参数则是存放在寄存器中。类的静态成员函数因为没有this指针这个参数，所以类的静态成员函数也就无法调用类的非静态成员变量。让我们调试，并对EXC进行手动修改会发现一点有趣的现象。

#1:this指针是什么时候创建的?

this在成员函数的开始执行前构造的，在成员的执行结束后清除。

#2:this指针存放在何处? 堆,栈,全局变量,还是其他?

this指针会因编译器不同，而放置的位置不同。可能是栈，也可能是寄存器，甚至全局变量。

https://www.jb51.net/article/40721.htm

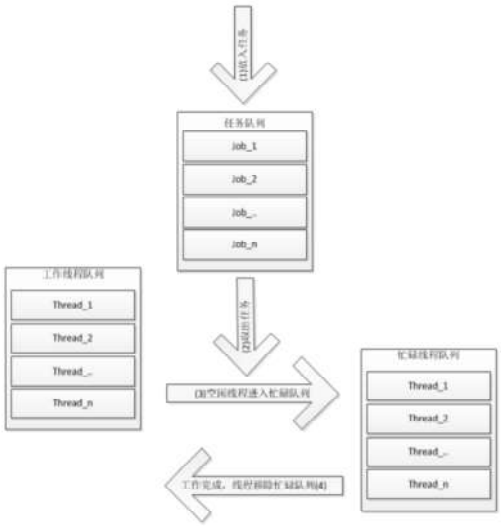
【23】volatile的作用
（1）阻止编译器为了提高速度将变量缓存到寄存器内而不写回
（2）阻止编译器调整volatile操作变量的指令顺序 —— 阻止编译器乱序 ，但是！！！不能防止运行期乱序，所以还是乖乖用六种内存序的语言级别的屏障

【24】C语言函数指针实现多态： https://www.cnblogs.com/mqxnongmin/p/10668922.html

Part two —— 操作系统

【1】静态库和动态库的区别
在Linux下：（1）静态库是.a文件，动态库是.so文件，生成方法不同
（2）在发布程序的时候，静态库会被打包进应用程序，而动态库不会　这 就导致了
静态库1、包含静态库的程序体积很大　2、静态库修改后要重新编译程序　3、静态库加载速度更快　4、因为编译时就被加载，所以是位置确定的代码！！！
动态库1、程序不包含动态库，动态库需要一起发布　2、动态库修改后只需要替换动态库，不用重新编译程序　3、动态库加载速度稍慢　4、程序装载进内存时才会链接进来确定位置，所以是位置无关代码（**PIC——position independent code**）　5、系统要使用动态库，要设置LD_LIBRARY_PATH

【2】关于线程池—— c++实现：https://www.cnblogs.com/yangang92/p/5485868.html
为什么要用线程池 —— 线程本身就是可重用资源，对于多任务处理时，不需要每次都开线程进行初始化，可以提前分配好一定数量的线程

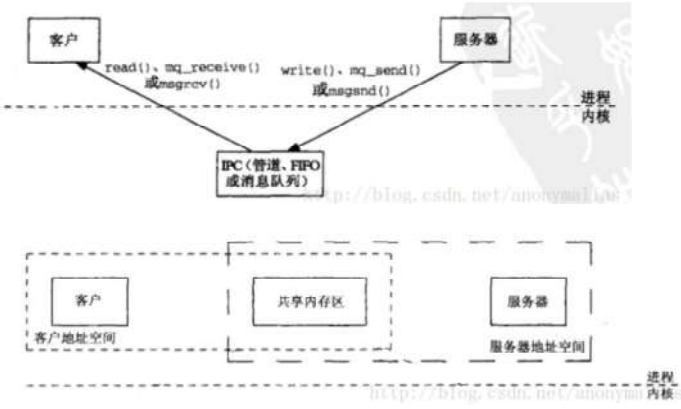


实现过程：（1）用条件变量和互斥量封装一个状态，保护线程池状态
（2）任务对象 和 线程池结构体：　—— 任务对象本身用一个链表进行连接，保存了第一个任务和最后一个任务（方便任务添加）

```
//封装线程池中的对象需要执行的任务对象
typedef struct task
{
    void *(*run)(void *args);  //函数指针，需要执行的任务
    void *arg;                 //参数
    struct task *next;         //任务队列中下一个任务
}task_t;
```

```
//下面是线程池结构体
typedef struct threadpool
{
    condition_t ready;     //状态量
    task_t *first;         //任务队列中第一个任务
    task_t *last;          //任务队列中最后一个任务
    int counter;           //线程池中已有线程数
    int idle;              //线程池中kongxi线程数
    int max_threads;       //线程池最大线程数
    int quit;              //是否退出标志
}threadpool_t;
```

【3】为什么共享内存是最快的IPC?
像管道、FIFO、消息队列，**写数据**时需要把数据**从进程复制进内核**，IPC**读数据**时，又要把数据**从内核复制到进程**。进程间通信借助内核完成，有两次数据复制过程。
而共享内存将**同一块内存区**映射到**不同进程的虚拟地址空间**中，进程直接对该内存进行读写即可（需要借助信号量进行**同步操作**）



【4】什么时候会发生上下文切换 —— 时间片耗尽 、 sleep等、中断、抢占式内核优先级更高的进程
开发者不可控的发生Context Switch：时间片轮转、抢占式内核优先级更高的进程、中断、异常、系统调用返回
开发者主动申请Context Switch：休眠当前进程/线程，sleep/pthread_cond_wait/mutex竞争等，唤醒其他线程/进程 pthread_cond_signal

what happens in context switch 进程上下文切换时发生了什么 —— （1）挂起当前进程，保存当前进程在CPU中的状态（也就是上下文）　（2）在内存中找到下一个调度的进程，并将其上下文在CPU中恢复　（3）跳到程序计数器PC指向的位置，继续执行程序，恢复该进程

进程切换 —— 需要切换页表，保存上下文信息在内核栈中（主要是寄存器中的信息），而其他数据本来就还在内存中
线程切换 —— 换一组栈和寄存器就行，不同线程使用不同的栈，将现场保护信息存在自身线程的栈中，就可以进行线程切换了

【5】父进程没有wait、waitpid，怎么杀死僵尸进程？？ —— 杀死父进程，让僵尸进程交给init进程托管 进行回收

【6】互斥锁和条件变量
使用互斥锁可以解决一些资源竞争问题，但互斥锁只有加锁和解锁两种状态，限制了互斥锁的用途 —— 谁抢到谁知道呢
而条件变量，是对互斥锁的补充，允许一个线程阻塞并等待另一个线程发送的信号，当收到信号时，阻塞的线程被唤醒并试图锁定与之相关的互斥锁 —— 有人在等，有人通知

【7】协程！！！
简单讲就是轻量级的线程，在**用户态**切换，更低的切换开销**避免用户态和内核态的切换**。其次就是**协程栈更小**，系统可以支持更多的协程，**并发数可以更大**其次就是因为开销低了并发模型就可以很简单的使用多协程的模型。

"协程"（Coroutine）概念最早由 Melvin Conway 于 1958 年提出。协程可以理解为纯用户态的线程，其通过协作而不是抢占来进行切换。相对于进程或者线程，协程所有的操作都可以在用户态完成，创建和切换的消耗更低。总的来说，协程为协同任务提供了一种运行时抽象，这种抽象非常适合于协同多任务调度和数据流处理。在现代操作系统和编程语言中，因为用户态线程切换代价比内核态线程小，协程成为了一种轻量级的多任务模型。

从编程角度上看，协程的思想本质上就是控制流的主动让出（yield）和恢复（resume）机制，迭代器常被用来实现协程，所以大部分的语言实现的协程中都有 yield 关键字，比如 Python、PHP、Lua。但也有特殊比如 Go 就使用的是通道来通信。

C协程的实现：setjmp和longjmp

异步 、并发、协程原理： http://www.iigrowing.cn/?p=6736

多线程或多进程是并行的基本条件，但单线程也可用协程做到并发。通常情况下，用多进程来实现分布式和负载平衡，减轻单进程垃圾回收压力；用多线程抢夺更多的处理器资源；用协程来提高处理器时间片利用率。现代系统中，多核 CPU 可以同时运行多个不同的进程或者线程。所以并发程序可以是并行的，也可以不是。

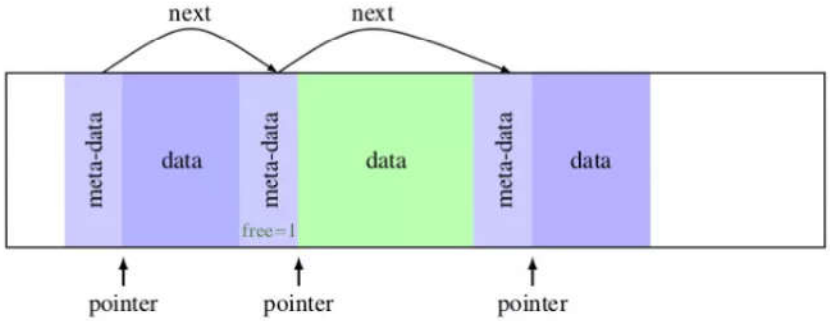
【8】线程安全的单例模式
（1）Double checked locking

```
singleton *get nstance()
{
    if(p == nullptr)
    {
        loc ();
        if(p == nullptr)
            p = ne  singleton;
        unloc ();
    }
    return p;
}
```

实际上由于编译器乱序，DCL是靠不住的
可以使用pthread提供的pthread_once实现一个更安全的单例模式

```
template<typename T>
class Singleton : boost::noncopyable
{
 public:
  static T& instance()
  {
    pthread_once(&ponce_, &Singleton::init);
    return *value_;
  }

 private:
  Singleton();
  ~Singleton();

  static void init()
  {
    value_ = new T();
  }

 private:
  static pthread_once_t ponce_;
  static T* value_;
};

// 必须在头文件中定义 static 变量
template<typename T>
pthread_once_t Singleton<T>::ponce_ = PTHREAD_ONCE_INIT;

template<typename T>
T* Singleton<T>::value_ = NULL;
```

【9】malloc的实现
（1）介绍一下进程地址空间，也就是内存布局
（2）malloc分配内存时，先去扫描free()释放的空闲内存块，看是否能找到满足尺寸要求的内存块（首次适配算法（时间效率高）和最优适配算法（空间效率高）），实际上堆内存空间会以块的形式组织起来



（3）当malloc找不到可用内存时，就需要扩展堆，就是brk和sbrk系统调用，扩展堆的地址空间起到分配内存的效果
—— 物理内存的实际使用时机（缺页异常调度算法）
（4）再扩展tcmalloc的实现，小内存分配，大内存分配，线程局部缓存，admvise( )，回收后的再分配
（5）最后总结，各种malloc的实现版本，都不能完全解决内碎片和外碎片的问题，要均衡考虑

Part three —— 服务器编程

【1】fork 、 vfork 、 clone
（1）fork —— 一次调用两次返回、**copy on write**（一般接exec系列的系统调用，没必要直接拷贝，否则重新装载程序白copy了内存）没有发生写入的资源是共享的，发生了写入的资源是独立的，提高进程创建速度　父子进程的**执行顺序不确定**
（2）vfork —— **子进程先调用**，父进程等待子进程exec或者exit后才执行，是在copy on write机制还没实现前使用的，二者**共享同一地址**空间，所以也认为vfork（）创建出来的是 轻量级进程，共享资源的进程，也叫线程
（3）clone —— linux提供的**创建线程的系统调用**，虽然也有pthread系列函数，**也可以用来创建进程**。可以通过**参数**控制选择**共享虚拟地址空间**，从而**创建线程**，也可以**不共享**，**创建进程**（甚至创建的进程与原进程可以是**兄弟进程**关系），父子进程的**执行顺序**也是由**标志位**控制

【2】僵死进程和孤儿进程的危害

（1）孤儿进程在父进程退出后，会交由Linux的1号进程/sbin/init进程托管，init会循环wait，回收这些退出的孤儿进程，所以本质上没什么危害。
（2）僵死进程还**保留着子进程的PCB**，包括（**进程号，退出状态，CPU占用时间**等），如果不被回收，就会一直占用内存资源，大量的僵死进程可能导致没有新的进程号分配给新进程，且系统资源被大量占用

【3】linux下查看进程端口号 —— 先查pid： ps -ef 或者 -aux | grep 进程名 得到进程pid 再用pid查看进程的端口占用情况： netstat -apn | grep pid

【4】进程间通信方式中哪种效率最高 —— 共享内存，有一个进程创建，映射一段被其他进程访问的内存（**最快IPC，因为进程直接对内存进行存取**），没有多余的拷贝操作（通常消息队列和管道等，都是通过系统调用完成send和recv的操作），通常和信号量一起使用，信号量用来同步对共享内存的访问 （用**信号量封装P操作和V操作**）

【5】CP、MV、RM命令的实际内容 —— （阿里中间件面试：mv为什么比cp快？？？） （字节跳动头条信息流：rm以及rm -f一个正在被使用的文件，其实两者没差）
（1）unlink系统调用——**删除文件名**，原目录下不再有这个文件。 只有当**文件的链接数为1（无硬链接）**&&**没有进程打开该文件**时，才会**删除文件内容**。如果删除期间，有进程打开了该文件，则文件内容会在所有进程都关闭了这个文件时，才会被真正删除。
rm -f f只是忽略不存在的文件，从不给出提示

（2）rm 实际就是调用**unlink系统调用**，在调用之前进行了一些权限检查

（3）rename， 重命名文件， 实际系统调用rename( )，如果文件名已存在，**会删除原来的文件**，然后新键一个同名文件

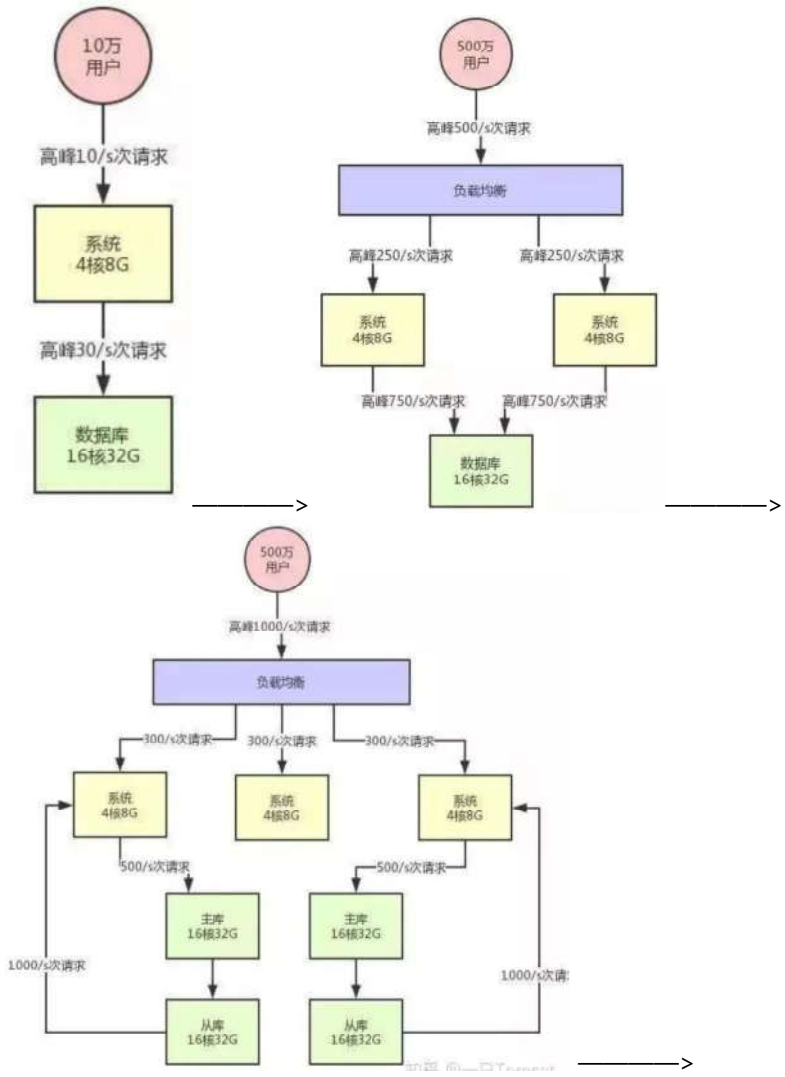
（4）MV其实就是用来文件重命名，就是调用的rename( )系统调用 —— 只需要**改变文件的inode**

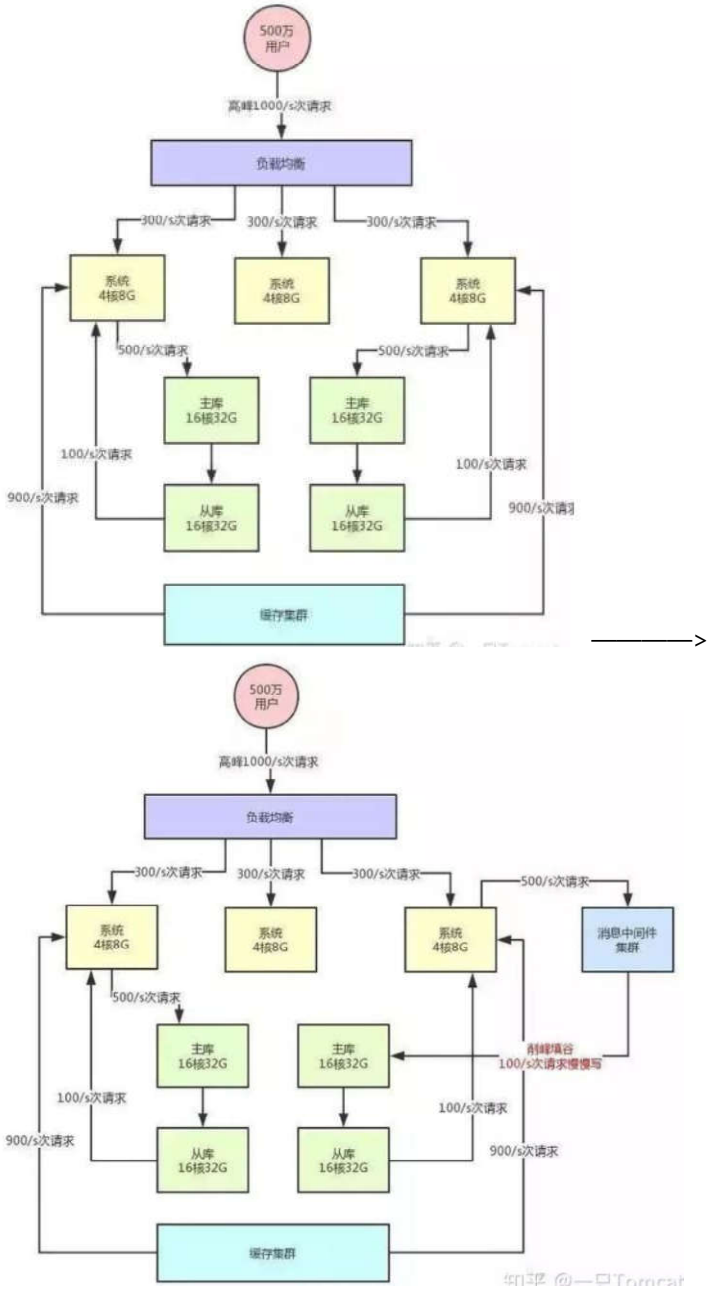
（5）CP则是**改变inode指向的Block**，当文件存在时，清空原来的数据，再写入新的数据

【6】Top 命令参数说明
load参数 —— 表示最近1,5,15分钟内系统的平均负载，单核cpu情况下，0.0表示没任何负载，1.0表示满负载，理想负载为0.7
Tasks参数 —— 进程相关信息，可以看到总进程数、正在运行的进程数、睡眠进程数和僵尸进程数等——僵尸进程怎么处理，杀死父进程，让/sbin/init进程去回收
CPU使用情况：包括用户空间、内核空间、中断等情况占用的cpu时间百分比

【7】sleep函数： 让出CPU资源，不会释放锁资源

********【8】系统是如何支持高并发的？？？ https://zhuanlan.zhihu.com/p/71524957?utm_source=wechat_session&utm_medium=social&utm_oi=817651029813387264
假设系统部署在一台机器上，背后连接了一个数据库，部署在一台服务器上，则系统的初始架构如下：


10万
用户
高峰10/s次请求
系统
4核8G
高峰30/s次请求
数据库
16核32G
500万
用户
高峰500/s次请求
负载均衡
高峰250/s次请求
高峰250/s次请求
系统
4核8G
系统
4核8G
高峰750/s次请求
高峰750/s次请求
数据库
16核32G
500万
用户
高峰1000/s次请求
负载均衡
300/s次请求
300/s次请求
300/s次请求
系统
4核8G
系统
4核8G
系统
4核8G
500/s次请求
500/s次请求
主库
16核32G
主库
16核32G
1000/s次请求
1000/s次请
从库
16核32G
从库
16核32G



————>

（1）用户量增长 —— 这时数据库还能支撑并发请求，只是服务器可能负载较高
系统处理一些复杂的业务逻辑会占用较高的CPU —— 添加**负载均衡层**，将**请求均匀**打到系统层；系统层采用**集群化部署**多台机器，初步抗住并发压力。

（2）用户量继续增长 —— 此时数据库负载过高，高峰期性能较差，还可能把数据库搞挂
数据库减压 —— **分库分表，读写分离**，让主库支撑写请求，每台主库再挂一台服务器从库，让从库去承担读请求

（3）如果用户量越来越大（**写少读多**） —— 其实是可以不断加机器的，如在系统层面不断加机器承载更高的并发请求，数据库层面扩容服务器，通过分库分表扩容机器，读写分离承载更高的并发读写，但是成本太高！！！ **缓存系统**的设计就是为了承载高并发而生。
在写数据库时，**同时写入缓存集群**，让缓存集群去承载大部分的读请求，比如2000次的读请求，可能1800次读取的都是缓存中不怎么变化的数据
——不盲目进行数据库扩容，数据库服务器成本极高，且本身不是用来承载高并发的；针对**写少读多**的请求，引入**缓存集群**，用缓存集群去承载大量的读请求

（4）如何处理数据库的**写入压力**呢 —— 引入**消息中间件集群**，进行**写请求异步化处理**，达到**削峰填谷**的效果

【9】Linux 远程登录 —— SSH基本过程
https://blog.csdn.net/m0_37822234/article/details/82494556

【10】线程和协程 —— 协程为什么能开更多，为什么高效

多个线程相对独立，有自己的上下文，但是**切换受系统控制，发生在内核态**；协程虽然也有自己的上下文，但是切换有自己控制，由当前协程切换到其他协程由**当前协程控制，发送在用户态**。

线程上下文切换和进程上下文切换：
https://blog.csdn.net/qq_40280705/article/details/82178688

Part four —— 网络编程

【1】OmniPeek抓TCP包：

```
Packet Info
    Packet Number:          4
    Flags:                  0x00000000
    Status:                 0x00000000
    Packet Length:          64
    Timestamp:              20:06:05.468825800 03/08/2019
Ethernet Type 2
    Destination:            00:00:5E:00:01:01  UscInforma:00:01:01 [0-5]
    Source:                 1C:1B:0D:EE:82:F3 [6-11]
    Protocol Type:          0x0800  IP [12-13]
IP Version 4 Header - Internet Protocol Datagram
    Version:                4 [14 Mask 0xF0]
    Header Length:          5  (20 bytes) [14 Mask 0x0F]
    Diff. Services:         %00000000 [15]
                              0000 00.. Default
                              .... ..00 Not-ECT
    Total Length:           41 [16-17]
    Identifier:             32553 [18-19]
    Fragmentation Flags:    %010 [20 Mask 0xE0]
                              0.. Reserved
                              .1. Do Not Fragment
                              ..0 Last Fragment
    Fragment Offset:        0  (0 bytes) [20-21 Mask 0x1FFF]
    Time To Live:           64 [22]
    Protocol:               6  TCP - Transmission Control Protocol [23]
    Header Checksum:        0x0000  Checksum invalid. Should be:  0xC704 [24-25]
    Source IP Address:      10.193.7.220 [26-29]
    Dest. IP Address:       202.108.23.152 [30-33]
TCP - Transport Control Protocol
    Source Port:            3264  ccmail [34-35]
    Destination Port:       443  https [36-37]
    Sequence Number:        3978307073 [38-41]
    Ack Number:             1866120379 [42-45]
    TCP Offset:             5  (20 bytes) [46 Mask 0xF0]
    Reserved:               %0000 [46 Mask 0x0F]
    TCP Flags:              %00010000  ...A.... [47]
                              0... .... (No Congestion Window Reduction)
                              .0.. .... (No ECN-Echo)
                              ..0. .... (No Urgent pointer)
                              ...1 .... Ack
                              .... 0... (No Push)
                              .... .0.. (No Reset)
                              .... ..0. (No SYN)
                              .... ...0 (No FIN)
    Window:                 258 [48-49]
    TCP Checksum:           0xF4BC  Checksum invalid. Should be:  0x6C9D [50-51]
    Urgent Pointer:         0 [52-53]
    No TCP Options [54-59]
SSL - Secure Sockets Layer
    Unrecognized SSL Layer
        -- SSL Data Can Not Be Recongnized --
        Length:             6    bytes
        SSL Data:           (6 bytes) [54-59]
FCS - Frame Check Sequence
    FCS:                    0x3393346E  Calculated
```

【2】session和cookie的区别和联系

（1）cookie放在客户的浏览器上，用于保存客户的身份；session放在服务器上，服务器为每一个客户分配一个session ID

（2）cookie放在本地并不安全，可能会被别人拿到并冒用，所以安全性session更高

（3）session放在服务器上，如果过多会比较占用服务器的性能，所以为减轻服务器性能，考虑用cookie

（4）现在大部分服务器都采用token验证，只需要在服务器上保存一个签名，用CPU的计算时间去换取session的存储空间

session和cookie如何配合，完成客户端身份的验证？？？
https://www.cnblogs.com/hzb462606/p/8977444.html
浏览器端 给 服务器发Http请求， 服务器给它生成一个session放在服务器上记录相关的状态信息，然后再生成一个sessionId来标识这个session，最后生成一个cookie将sessionId放在cookie中，然后给浏览器回送http请求，添加Set-Cookie字段（并发送cookie过去），浏览器端保存这个cookie到本地，以后再向这个服务器发送http请求时就带上cookie就可以保存状态了。

对比Token验证技术 —— 采用token可以不保存sessionId，只是生成token和验证token，用**CPU的计算时间换取服务器的session存储**空间

***【3】TCP粘包问题及解决方法 —— 包粘在一起发送，包头接包尾 —— **配合Tencent taf使用的实际经验**！！！
原因（1）发送方：TCP默认开启了Nagle算法，导致网络中只能存在一个小分组，其他分组在上一个小分组到达前粘包
（2）接收方：应用程序没有立即处理缓冲区中的包，导致粘连

解决方法： —— （1）发送方：关闭Nagle算法，使用TCP_NODELAY选项 ； 对于不希望粘包的信息，携带PUSH标志位，让接收方收到以后，立即从缓冲区去读数据
（2）应用层：为包添加约定的结束标志符（格式化数据）、固定包的发送长度

【4】TCP初始序列号为什么要随机产生?
（1）为了安全考虑，防止被黑客轻易知道序列号后制造tcp序列号攻击 （2）防止socket连接失效以后，socket被重用使得原来的包被错误接受

【5】如果服务器调用完sock，bind和listen，调用sleep(1000)，那么请求方的connect()会返回吗—— 会返回，调用listen后，回复连接信号SYN的操作都是由内核完成，就算在sleep()后没有调用accept()函数，仍然会返回 —— 所以说connect()的返回是在收到服务器端的SYN信号后，回复SYN后，就返回了

【6】（1）两个epoll监控同一个文件描述符 —— 都会返回！！！
（2）epoll监控一个socket，如果调用close(socket)，会自动从epoll集合中删除掉——但是只有所有指向该socket的文件描述符都被关闭了，才会被删除（close与shutdown的区别！！close是引用计次减1）

【7】服务器**大量Time_wait**状态的原因？？？ 解决方法？？？
原因：常见于爬虫服务器或者WEB服务器，作为**主动断开连接**的一方，需要等待2MSL（一般是60s或者120s），导致产生大量的Time_wait状态
解决方法： 优化服务器参数 —— 修改/etc/sysctl.conf文件，包括修改**开启重用**（允许将TIME_WAIT的sockets重新用于新的TCP连接）和**开启快速回收**（快速回收TCP连接中的sockets）

Time_wait状态：可以通过优化服务器参数解决，因为发生Time_wait的情况是服务器可控的，要么是对端连接异常，要么是自己没有迅速回收资源。

而**大量的Close_wait**状态，必然是自身忘记关闭连接或者程序忙于读写而忘记关闭连接。
解决方法 —— 对Socket进行Read操作，读到0，说明对端已关闭，如果返回负数，且errno不是EAGAIN，也将socket关闭

【8】四次挥手是否可以3次 —— 可以，当服务器回复主动断开的客户端的SYN时，也没有数据发送了，连带发送自身SYN，此时就是3次挥手。

【9】长连接和短连接
短连接： 一次通信建立一次TCP连接，每次都需要经过三次握手建立连接和四次挥手断开连接的过程，如HTTP1.0，每个HTTP请求都需要建立一次TCP连接
长连接： 在连接期间会保持状态，双方维护心跳包来保持连接状态；如HTTP1.1，在头信息中加入了Connection：keep-alive后，一次TCP连接就可以支持多个HTTP请求。所以长连接多用于：操作频繁（读写）、点对点通讯，连接数不能太多（服务器没这么多资源保持很多的长连接）

【10】输入URL后的过程
https://blog.csdn.net/dididudidudu/article/details/80181505

【11】局域网下设备的上网流程
DHCP（动态分配一个局域网下的IP地址） —— 数据从设备到路由器 —— 路由器支持NAT映射（未注册地址和网络地址之间的翻译） —— 源IP和源端口修改后，将数据发送到目的IP

具有NAT功能的路由器维护了一个**地址翻译表**，NAT寻址算法工作流程：
1）当内部网络上的计算机发送一个TCP或者UDP包给网络外的计算机时，路由器将源IP和端口号保存为地址翻译表中的一个可用项；
**********2）路由器用路由器的IP地址替换源IP地址，用**虚拟端口号**替换源端口号，虚拟端口号指向（1）中的地址翻译表项，然后数据包被转发出去；
3）路由器从外部计算机接收到一个TCP或UDP包时，使用包中的目的端口号去访问地址翻译表项，最后找到内部的计算机。

其实工作原理都和缓存相关，ARP也是

【12】关于MTU —— 最大传输单元
https://yq.aliyun.com/articles/222535

【13】ARP详细过程
首先，源主机先查找目标主机的IP在**ARP缓存**中是否存在对于的MAC地址，如果存在，直接将数据包发送至该MAC地址；否则，向本网段发起**一个**ARP请求的广播包，其中包含了**源IP+源MAC+目标IP**，网段内所有主机都会收到这个广播，检查目标IP与自身IP是否一致，如果一致，将发送端的IP和MAC信息**添加到自己的ARP缓存中**，然后给源主机回一个ARP响应包，其中包含自己的MAC地址，源主机收到响应后，将目的主机的IP+MAC**添加到自己的ARP缓存**中

Part five —— 数据库

【1】SELECT .... FOR UPDATE的作用
for update只适用于InnoDB引擎（因为**加的是行级锁**），MYSQL会对查询结果的每一行添加行级排他锁，其他线程对该记录的更新会被阻塞。

【2】聚簇索引和非聚簇索引
聚簇索引 —— 将数据与索引放到了一块，索引结构的叶子节点保存了行数据 —— 显然行的数据只保存一份，所以聚簇索引具有唯一性
innodb基于聚簇索引建表，主要组织在一棵B+树上，在聚簇索引之上建立的都是辅助索引，此时辅助索引上存的不再是行的物理位置，而是主键值，所以辅助索引访问数据总是要二次查找（从辅助索引找到主键值 —— 根据主键值去找数据）
聚簇索引默认是主键 —— 没有主键？？找一个**唯一且非空**的索引 —— 没有这种键？？隐式定义一个主键作为聚簇索引

非聚簇索引 —— 数据与索引分开存储，索引的叶子节点指向了数据对应的位置

【3】Mysql如何实现主从一致性
参考单机的WAL机制，双机时也一样，只有从机实现了日志落盘、主机也实现了日志落盘才算事务完成

【4】Mysql的几种索引类型和使用场景 —— 其实就是招聘笔记里的索引结构
1、全文索引 FULLTEXT —— 主要是为了解决WHERE name LIKE "%word%"这类针对文本的查询问题

2、HASH索引 —— 可以一次定位，不像树形索引那样需要逐层查找，具有极高的效率， 使用场景：只有在 = 和 in 条件下才高效，不适合范围查找、排序和组合索引

3、B/B+树索引 —— Mysql默认和最常用的所有类型，依次遍历node，获取leaf，适合范围查找和排序

【5】最左匹配原则及其实现原理——实际上最左匹配是针对联合索引来说的，对于联合索引，底层会依据最左端的字段建立B+树，所以要将最常用，最有区分度的字段放在第一个

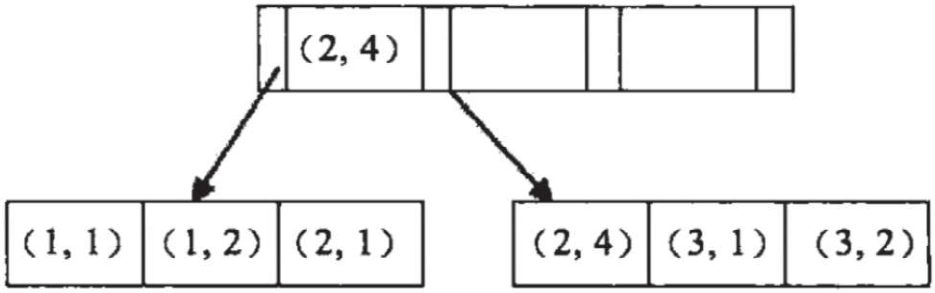


图 9-22　多个键值的 B+ 树

Part six —— 数据结构与算法

【1】**稳定排序**的意义？ —— 稳定排序可以让第一个关键字的排序结果**服务于第二个关键字　应用场景**：本次考试同分的排名按照学号排序（或者按照上一次考试成绩排序）
【2】40亿不重复数据，找到一个数是否存在 —— Bitmap位图　数字除以64得到的商就是位图中的下标，余数就是下标中的第几个比特位
https://blog.csdn.net/qq_31828515/article/details/56853478

【3】堆的实现 —— https://blog.csdn.net/look_poet_of_water/article/details/80155690
主要三个函数：
（1）插入：元素插入数组最后一个位置处，不断与父节点值比较，如果更大，则父节点下移，直到父节点值更大 或者到达堆顶
（2）删除：将堆顶元素和最后一个位置元素交换（然后将堆的size--），对于父节点开始进行调整，找到两个子节点的较大值，跟父节点比较，如果父节点小，就将父节点跟该节点交换，否则调整完毕，同样需要注意判断是否到达边界
（3）堆调整：从k = H->size/2开始（最后一个父节点开始执行堆调整的过程，直到k递减到0，堆调整完毕，也就是建堆完毕）

【4】单向链表是否是回文串——O(n)时间 O(1)空间 —— 第一次遍历得到链表长度 —— 第二次在中间将链表断开，后半段链表逆置， 对比两段链表是否相同~

【5】关于卡特兰数及其相关算法题
什么是卡特兰数 —— F(n) = C（2n,n）/ (n+1)　递推公式： **h(n)=h(n-1)*(4*n-2)/(n+1);**

令h(1)=1,h(0)=1，catalan数（卡特兰数）满足递归式：

h(n)= h(0)*h(n-1)+h(1)*h(n-2) + ... + h(n-1)h(0) (其中n>=2)

**********常见应用及其推导： —— 0,0走到n,n问题（本质和出栈序列数量问题一致）、n个节点组成的二叉树数量、圈上2n各点连成不相交的通路问题
（1）n个节点组成的二叉树的个数 —— F(n)
1个节点时，只有一种情况，F(1) = 1；
2个节点时，根节点固定，剩下一个节点可以是(1,0)或者(0,1)，F(2) = 2 = F(0)*F(1) + F(1)*F(0)；
3个节点时，根节点固定，剩下两个节点可以是(2,0)、(0,2)、(1,1)，F(3) = 5 = F(0)*F(2) + F(1)*F(1)+F(2)*F(0); ——得到递推公式，该公式有通项，即卡特兰数
如果有n个节点，那么左右子树节点数量的分布情况不就是(n-1,0).....(0,n-1)吗，而子树的数量前面已经给出！！！就是递推公式

（2）一个栈里面有1...n，问出栈顺序有多少种 —— F(n)
每个数必须进栈一次出栈一次，进栈看成状态1，出栈看成状态0，n个数的所有状态对应n个1和n个0组成的二进制数，因此输出总序列数=n个1+n个0组成的2n位二进制数的个数，1的累计数不小于0（**其实跟从（0,0）走到（n,n）只能向右或者向上走，不超过y=x这条线的问题是一样的**！！！！），也可用dp去解！！！！
对于不符合要求的数，必然是在奇数位2m+1上，有m+1个0和m个1，则对于剩下的数字，有n-m个1和n-m-1个0，如果将剩下数字中的0和1互换，那么将得到一个由n+1个0和n-1个1组成的二进制数，即一个不符合要求的数对应于一个由n+1个0和n-1个1组成的数，不符合要求的数为C(2n,n+1)，而总数是C(2n,n)。所以F(n)=C(2n,n)-C(2n,n+1) = C（2n,n）/ (n+1)

【6】k个链表的归并排序 —— 将K个链表投放入小顶堆，然后每次取堆顶元素，将堆顶元素的下一个结点加入堆中
需要关注代码 + priority_queue的使用
https://blog.csdn.net/u012677715/article/details/78718111

【7】文本编辑器的数据结构 —— 用双向链表 + 字符数组，空间时间的均衡考虑*****

【8】找到数组中出现次数大于 n / k 的元素，其实就是找到超过一半的数的变种 —— 要求：空间复杂度O(k)，时间复杂度O(n*k)
https://blog.csdn.net/u013309870/article/details/69788342
每次在数组中删除掉k个不同元素，当剩余元素个数小于k时，出现超过了n/k次的元素必备留下

Part Seven —— 高并发下的无锁设计

【1】一写多读情况下的双Buffer缓冲设计
双Buffer的备份机制，避免了同时读写同一变量，额外设置一个备份变量，写的线程只向备份变量写入，而所有的读线程都访问主变量即可，当写操作完成后，触发主变量和备份变量的更新，达到数据更新的目的。

【2】锁保护的是一段代码，如果只需要对单个变量做修改 —— 用原子变量代替锁，效率会高很多

【3】线程安全下的HashMap
https://blog.csdn.net/V_Axis/article/details/78616700

【4】单例模式的double-check下，可能因为new对象时的三步（1、operator new分配空间，2、placement new在空间上构造对象，3、将内存地址类型转换后返回给指针），这里可能由于编译器乱序，2、3被颠倒，导致竞争问题出现（即一个对象还没有被构造，但是指针已经不为nullptr，如果被另一个线程使用，就有可能Core），所以最稳妥的方式还是用pthread_init_once来做，或者先new给一个局部对象，然后加个barrier( )，再将局部对象的地址给单例指针

Part Eight —— 逻辑思维与算法（工程能力）

【1】随机洗牌　rand() 只可以用这一个函数 完成一个54张牌的洗牌程序，rand() % 54，然后将尾元素和它互换！！！！尾元素--

【2】1000个棋子，每次可取1-8个，问先手必胜策略 —— 反推，当对面取时，如果剩9个，则必胜， 再往前推 —— 剩18个不就必胜吗，所以先将剩余棋子取成9的倍数，然后不管对面取m个，自己取9-m个，则必胜

【3】2000万高考学生的成绩排名 —— 众多高考学生的成绩只局限在少数区间，所以只要在用一个vector记录每个考分有多少人就行了，不需要排序

【4】利用不均匀概率产生等概率 —— 一枚不均匀硬币，正反概率不同，如何得到等概率 —— 正反、反正概率是相同的，这就是一对等概率

【5】等概率产生不等概率 —— 一枚均匀硬币，正反概率相同，如何得到不等概率 —— 模拟二进制数，假设硬币结果是函数foo()
那么：foo()*2^4 + foo()*2^3 + foo()*2^2 + foo()*2^1 + foo() 可以得到0-31等概率，那么取%3，就得到不等概率

【6】小范围随机数生成一个大范围随机数，如0-4 生成 0-12 ？？？ —— 把0-4的结果，作为一个5进制数，把第一次生成的结果作为低位，第二次生成的结果作为高位，就得到24种结果，再对12取个模就可以了

再比如一个骰子产生1　的随机数　　所有的　到n其实都是映射问题，把骰子扔 次，就能得到　种等概率的情况，抛弃掉其中一种，然后将这些组合分配好做一个映射到 种情况就行

【每次投掷等概率产生6个结果中的一个】这个假设在，无论你投多少次，产生结果的态数都是6的N次方，不包含7这个因子。

也就是小到大的话，如果无法引入大数这个因子，用于也无法解决不舍弃部分情况直接等概率生成的问题

【7】大范围随机数生成小范围随机数，如0-13生成0-9的随机数？？？
大到小不一样，为了引入9这个因子，可以去组合啊，13*12*11*10*9/5/4/3/2/1，这就引入了9这个因子，把组合数分情况就能得到完全随机的情况，不过可能需要一定的存储空间

【8】 1 — 10000，少了两个数，如何快速找出这两个数， 做减法找到 x+y，再找平方x^2+y^2就可以了

【9】如何判断一棵树是否是完全二叉树 —— 按层遍历

（1）如果一个节点有右孩子没有左孩子，肯定不是完全二叉树
（2）如果一个节点只有左孩子没有右孩子，那么他右边的节点必须都是叶子节点，否则不是完全二叉树

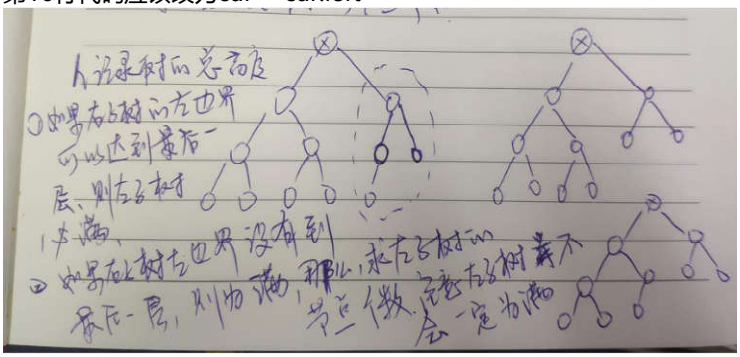
【10】memcpy的实现 —— https://blog.csdn.net/goodwillyang/article/details/45559925
需要考虑重叠、定义临时变量把头接过来、考虑不同的指针类型，不同指针类型不能直接++

```
void *memcpy(void *dest, const void *src, size_t count)
{
 char *d;
 const char *s;

 if (dest > (src+size)) || (dest < src))
    {
    d = dest;
    s = src;
    while (count--)
        *d++ = *s++;
    }
 else /* overlap */
    {
    d = (char *)(dest + count - 1); /* offset of pointer is from 0 */
    s = (char *)(src + count -1);
    while (count --)
        *d-- = *s--;
    }

 return dest;
}
```

【11】atoi实现 —— https://www.cnblogs.com/ralap7/p/9171613.html

【12】如何利用一个能返回【0,1】平均随机点的函数，等概率生成一个圆内的点
（1）极坐标，随机产生一个0-360的角度，随机产生一个0-1上的半径r
看似完全随机，但是r越小其实越密集！！！
（2）正解：随机产生一个0-360的角度，随机产生一个0-1上的半径平方r^2

【13】小于O(N)的时间，求完全二叉树的节点数量 —— 二叉树的遍历和求二叉树的高度（其实也是遍历）的时间复杂度都是O(N)
第16行代码应该改为cur = cur.left



```java
public int count(TreeNode root) {
        if(root==null) return 0;
        int count = 0;
        TreeNode cur = root.left;
        //记录左子树的深度
        int ldepth = 0;
        while(cur!=null) {
            ldepth++;
            cur = cur.left;
        }
        cur = root.right;
        //记录右子树深度
        int rdepth = 0;
        while(cur!=null) {
            rdepth++;
            cur = cur.right;
        }

        //如果ldepth==rdepth,则说明左子树是满二叉树，可用公式求解
        if(ldepth==rdepth) {
            //这里不用减一，因为把根节点算进去了
            count = (int)Math.pow(2, ldepth)+count(root.right);
        } else { //此时ldepth>rdepth,则说明右子树是一棵满二叉树
            count = (int)Math.pow(2, rdepth) + count(root.left);
        }
        return count;
    }
```

【14】二维数组的最长增长子序列 —— 深搜时间复杂度 O(n^2*m^2) —— 加速，以空间换时间，经过一个点的最大值是固定的，所以记录下来，后面就不要算了

```
#define len 110
int m,n;
int dp[len][len];
int vis[len][len];
int judge(int i,int j)
{
   if(i>=0&&i<m&&j>=0&&j<n)
      return 1;
   return 0;
}
```

```
int setdp(int a[len][len],int i,int j)
{
   if (vis[i][j]==1)
   {
      return dp[i][j];
   }
   vis[i][j]=1;
   int t;
   if(judge(i-1,j)&&a[i][j]>a[i-1][j])
   {

      t=setdp(a,i-1,j)+1;
      if(dp[i][j]<t)
         dp[i][j]=t;
   }
   if(judge(i+1,j)&&a[i][j]>a[i+1][j])
   {

      t=setdp(a,i+1,j)+1;
      if(dp[i][j]<t)
         dp[i][j]=t;
   }
   if(judge(i,j-1)&&a[i][j]>a[i][j-1])
   {

      t=setdp(a,i,j-1)+1;
      if(dp[i][j]<t)
         dp[i][j]=t;
   }
   if(judge(i,j+1)&&a[i][j]>a[i][j+1])
   {

      t=setdp(a,i,j+1)+1;
      if(dp[i][j]<t)
         dp[i][j]=t;
   }
   return dp[i][j];

}
```

【15】单链表快排，值交换即可 —— https://blog.csdn.net/otuhacker/article/details/10366563
单链表归并 —— https://www.cnblogs.com/qiaozhoulin/p/4585401.html

【16】2个鸡蛋100层楼问题 dp —— https://www.xuebuyuan.com/723672.html

【17】带冻结期的股票买卖问题 leetcode 309

dp[i][j] i是天数，j是状态，0代表未持有股票，1代表持有股票，2代表冷冻期

-prices[i]代表买这一天的股票花的钱

```java
class Solution {
    public int maxProfit(int[] prices) {
        int n = prices.length;
        if(n == 0) return 0;
        int[][] dp = new int[n + 1][4];
        dp[0][0] = 0;
        dp[0][1] = -prices[0];
        dp[0][2] = 0;
        for(int i = 1; i < n; i++) {
            dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][2]);
            dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i]);
            dp[i][2] = dp[i - 1][1] + prices[i];
        }
        return Math.max(dp[n - 1][0], dp[n - 1][2]);
    }
}
```