一、图书推荐系统

1. 系统架构

已有：资源库（图书名+图书频率，大概三万行多点）

索引表建立：构建倒排索引表（map），key-value中key存放分词之后的词，value存放set合集，set就是所有与key相关的图书名；构建书频表（map）

搜索：根据搜索词依据最小编辑距离算法求得搜索词与图书名之间的跳跃度。

2. Reactor+threadpool模式

参考Muduo，《Linux多线程服务器端编程》， 陈硕大神个人开发的 C++ 的 TCP 网络编程库。muduo 基于 Reactor 模式实现，Reactor 模式也是目前大多数 Linux 端高性能网络编程框架和网络应用所选择的主要架构，例如 Redis 和 Java 的 Netty 库等。所谓 Reactor 模式，是指有一个**循环**的过程，**不断监听对应事件是否触发，事件触发时调用对应的 callback 进行处理。**

负责事件循环的部分在 muduo 被命名为 EventLoop。

负责监听事件是否触发的部分，在 muduo 中叫做 `Poller`。muduo 提供了 epoll 和 poll 两种来实现，默认是 epoll 实现。

陈硕认为，TCP 网络编程的本质是处理三个半事件，即：

1. 连接的建立
2. 连接的断开：包括主动断开和被动断开
3. 消息到达，文件描述符可读。
4. 消息发送完毕。这个算半个事件。

而**Proactor**的整体结构和reacotor的处理方式大同小异，不同的是**Proactor**采用的是异步非阻塞IO的方式实现，对数据的读写由异步处理，无需用户线程来处理，服务程序更专注于业务事件的处理，而非IO阻塞。

二、企业私有云

三、C语言

1. memmove与memcpy

```
void *memcpy(void *dst, const void *src, size_t count);
void *memmove(void *dst, const void *src, size_t count);
```

作用都是拷贝一定的数据长度的内存的内容，但是当内存发生局部重叠的时候，memmove保证拷贝的结果是正确的，memcpy不保证拷贝的结果是正确的。

```
void* my_memcpy(void* dst, const void* src, size_t n)
{
    char *tmp = (char*)dst;
    char *s_src = (char*)src;

    while(n--) {
        *tmp++ = *s_src++;
    }
    return dst;
}

void* my_memmove(void* dst, const void* src, size_t n)
{
    char* s_dst;
    char* s_src;
    s_dst = (char*)dst;
    s_src = (char*)src;
    if(s_dst>s_src && (s_src+n>s_dst)) {      //-------------------------第二种内存覆盖的
情形。
        s_dst = s_dst+n-1;
        s_src = s_src+n-1;
        while(n--) {
            *s_dst-- = *s_src--;
        }
    }else {
        while(n--) {
            *s_dst++ = *s_src++;
        }
    }
    return dst;
}
```

四、数据结构与算法

1. Hash表

哈希表的动态扩容：

常见的情况是在填装因子(loaderFactor) > 0.75是进行扩容

JAVA中HashMap的实现方式：每次扩容时，哈希表的容量增加为原先的两倍。是需要先将原哈希表中所有键值对都转移到新的哈希表中，这个过程是比较慢的，此时插入该元素的性能相当低。

Redis中的实现方式：采取的是**分摊转移**的方式。即当插入一个新元素x触发了扩容时，先转移第一个不为空的桶到新的哈希表，然后将该元素插入。而下一次再次插入时，继续转移旧哈希表中第一个不为空的桶，再插入元素。直至旧哈希表为空为止。

2. B树、B+树

B树(logBN)：

B树是将各种数据信息保存在所有节点中的，B+树是将各种信息保存在叶子中的。

但是B树也有优点，其优点在于，由于B树的每一个节点都包含key和value，因此经常访问的元素可能离根节点更近，因此访问也更迅速

B+树(logBN)：

- 由于B+树在内部节点上不好含数据信息，因此在内存页中能够存放更多的key。 数据存放的更加紧密，具有更好的空间局部性。因此访问叶子几点上关联的数据也具有更好的缓存命中率。
- B+树的叶子结点都是相链的，因此对整棵树的便利只需要一次线性遍历叶子结点即可。而且由于数据顺序排列并且相连，所以便于区间查找和搜索。而B树则需要进行每一层的递归遍历。相邻的元素可能在内存中不相邻，所以**缓存命中性**没有B+树好。

为什么查询性能B+树更优

B树是为了减少磁盘IO而创建出来的，因为二叉树虽然快，但是内存空间有限，一下子读取不了那么多。那就必须按照磁盘页的大小，依次读取节点到内存中。在数据量相同的情况下，因为B树每个节点上都存在数据。而不一样的是，B+树只有在叶子节点才会存在数据，所以呢 同样的情况下，B+树的这种结构，一次性能够放入内存的节点数量就可以增加了。因为B+树中间节点放的是引用地址嘛，这样读取性能又能够翻一翻。

为什么看似效率更低的B+树比B树更广泛应用

实际上B+树的效率更高。在实际应用中，数据库中存放的数据很多，不像我们测试时一样，每次只是几行数据代表全部。**而数据库的一条经验法则就是： 尽可能多的一次性将数据放进内存中处理**。举个例子，内存容量以及性能允许一次性存放300条数据，但是数据库中有3000条数据要处理。那么我们每次放300条，需要放10次，I/O操作就是10。众所周知，**I/O操作（硬盘到内存）往往比单纯的检索更费时间**，因此I/O操作越少越好。
设：300条数据查找时间为300，I/O操作需要30，那么`300*10+30*10 = 3300`
如果每次放入30条数据：`30*100 + 20*100 = 5000`，这里由于每次传输的数据少，将每次传输时间假设少于`30`。可见I/O操作的耗时。

那么，由于B+树不将数据信息存放进节点，也就是说节点信息的容量变小，那么一次性可以放进内存的节点数变多，意味着I/O操作变少，以此提高性能

3. 红黑树

性质：

节点红/黑

根黑

空叶黑

红色的孩子都是黑色

从根出发，到所有叶子经过的黑色节点数量都是一样的

插入调整：

插入到根，直接染色

插入节点父亲是黑色，不调整

父亲叔叔红色，置黑，爷爷染红，将爷爷看作新的节点向上调整

父亲红叔叔黑色或者空，腰型，旋转

父亲红色叔叔黑色，裙型，旋转，新的爷爷节点染黑，其孩子均染红

4. 八大排序

	原地排序	稳定性	最好	最坏	平均
冒泡	√	√	O(n)	O(n方)	O(n方)
插入	√	√	O(n)	O(n方)	O(n方)
选择	√	×	O(n方)	O(n方)	O(n方)
归并	o(n)	√			O(nlogn)
快速	√	×			O(nlogn)
桶（外部）	×	√			O(n)
计数	×	√			O(n+k)
基数	×	√			O(dn)

五、Linux系统

1. 操作系统基本操作

(1) 基本操作

su—切换用户

useradd、userdel—增加、删除用户

(2) 进程相关

ps—查看系统中的进程

top—动态显示系统中的进程

kill—中止进程

2. 进程与线程

(1) 进程与线程定义

进程是资源分配的最小单位，线程是CPU调度的最小单位。从Linux内核的实现，也就是进程和线程是如何创建的来出发。在 Linux 中进程和线程实际上都是用一个结构体 `task_struct` 来表示一个执行任务的实体。进程创建调用 `fork` 系统调用，而线程创建则是 `pthread_create` 方法，但是这两个方法最终都会调用到 `do_fork` 来做具体的创建操作 ，区别就在于传入的参数不同。

(2) 孤儿进程

若父进程先于子进程退出，则子进程成为孤儿进程，此时将自动被PID为1（init，Linux中的所有进程都是有init进程创建并运行的。首先Linux内核启动，然后在用户空间中启动init进程，再启动其他系统进程。在系统启动完成完成后，init将变为守护进程监视系统其他进程。）的进程接管。但在init进程清理子进程之前，它会一直消耗系统的资源。

(3) 僵尸进程

如果子进程先退出，系统不会清理掉子进程的环境，而必须由父进程调用wait或waitpid函数来完成清理工作，如果父进程不做清理工作，则已经退出的子进程将成为僵尸进程。如果系统中的僵尸进程过多，会影响系统的性能。

(4) 进程的结束方式

main函数的自然返回

调用exit函数

调用_exit函数

调用abort函数

接收到能导致进程终止的信号

PS：其中exit结束时直接终止，并不会返回值，后面的都无法执行；但是return可以返回值，接着执行后面的。

(5) 打印"-"

请问下面的程序一共输出多少个“-”

```
int main()
{
    int i=0;
    for(i=0;i<2;i++)
    {
        fork();
        printf("-");
    }
    return 0;
}
```

打印8个-，因为第一次fork时并没有刷新缓冲区，故缓冲区内的-也被子进程复制过去。

printf("-")换成printf("-\n")打印6个-，当i=0时，两个fork返回值各打印一次-，此时已经分成了父进程和子进程，执行完一次后，i=1，此时父进程和子进程各自再次fork，故又打印出4个-。

(6) 守护进程vs后台进程

当控制终端被关闭时，相应的进程都会自动关闭。守护进程便可突破这种限制，它被执行时开始运转，独立于控制终端和会话在运行，直到整个系统关闭时才退出。独立于控制终端和会话在运行，和运行前的环境隔离开。

(7) 线程等待

子线程之间可以相互等待，但不能等待主线程

(8) 线程取消

给目标线程发送cancel信号，发送后即返回，但如何处理cancel信号由目标线程决定。可能直接结束、忽略、运行至取消点

(9) 线程资源清理

为什么需要资源清理：

线程间在访问一些共享资源时，需要独占的使用这些资源，先加锁，加锁成功，才能访问资源，访问资源结束后，解锁。假如一个线程加锁成功之后，访问资源时线程终止，不会去解锁，资源没有被释放，其他线程想访问资源的时候，永远都不能加锁成功，其他线程就没办法正常运行。

```
//资源清理函数，这两个函数必须配对使用，并且在同一个级别的代码段中。
pthread_cleanup_push(void(*cleanfun)(void*),void* arg);
pthread_cleanup_pop(int execute);
```

清理函数得到执行的三种情况：

1. 线程被cancel
2. 线程通过pthread_exit()退出
3. 线程通过pthread_cleanup_pop(1)，显示弹栈

清理函数不会得到执行的两种情况：

1. 线程通过pthread_cleanup_pop(0)
2. 线程通过return结束

(10) 线程的同步-条件变量

```
//静态初始化
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

//动态初始化
pthread_cond_t cond;
pthread_cond_init(&cond,NULL);
```

使用说明：

1.等待条件变量的成立（等待在条件变量上）

(1)无条件等待-pthread_cond_wait(&cond,&mutex);

(a)上半部

1.在条件变量上排队

2.解锁

3.睡眠

(b)下半部

1.被唤醒

2.加锁（如果没有锁则加锁成功；如果有锁，线程会阻塞等待）

3.从pthread_cond_wait函数返回（线程从条件变量上被唤醒，不代表一定会立刻从pthread_cond_wait函数返回）

(2)计时等待-pthread_cond_timedwait(&cond,&mutex,&timespec)

2.使条件变量成立（条件变量的激活）

(1)激活等待在队列上的第一个线程-pthread_cond_signal(&cond);

(2)激活所有等待在队列上的线程-pthread_cond_broadcast(&cond);

(11) 线程的互斥-线程锁

```
//静态初始化
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
//动态初始化
pthread_mutex_t mutex;
pthread_mutex_init(&mutex,NULL);
//加锁
pthread_mutex_lock(&mutex);
//解锁
pthread_mutex_unlock(&mutex);
//测试加锁
pthread_mutex_trylock(&mutex);
//锁的销毁，只有没被使用的锁才可以被销毁
pthread_mutex_destroy(&mutex);
```

锁的属性：

1. 快速锁，当一个线程加锁以后，其余请求锁的线程将形成一个阻塞等待队列，并在解锁后按优先级获得锁。这种锁策略保证了资源分配的公平性。
2. 嵌套锁，允许同一个线程对同一个锁成功获得多次， 并通过多次 unlock 解锁。 如果是不同线程请求，则在加锁线程解锁时重新竞争。

3. 检错锁，如果同一个线程请求同一个锁， 则返回 EDEADLK，否则与 PTHREAD_MUTEX_TIMED 类型动作相同。这样就保证当不允许多次加锁时不会出现最简单情况下的死锁。如果锁的类型是快速锁，一个线程加锁之后，又加锁，则此时就是死锁。

死锁的四个必要条件：

互斥、请求与保持、不可剥夺、循环等待

(12) 线程安全

如果一个函数能够安全的同时被多个线程调用而得到正确的结果，那么，我们说这个函数是线程安全的。 简单来说线程安全就是多个线程并发同一段代码时，不会出现不同的结果，我们就可以说该线程是安全的。

(13) 线程池

一个进程最多能创建多少个线程：

3. 同步与互斥

同步：实现进程/线程间有序执行/对资源的有序访问

4. Socket网络编程

TCP/IP参考模型：应用层、传输层、网络层、网络接口层

OSI参考模型：应用层、表示层、会话层、传输层、网络层、数据链路层、物理层

抓包工具：WireShark(Windows)，tcpdump(Linux)

(1) TCP/IP

特点：

1. 面向连接
2. 全双工
3. 可靠交付
4. 传输字节流，没有边界

协议格式：

三次握手：

第一次握手是客户端向服务器发送连接请求，第二次是服务器确认刚刚客户端的连接请求，第三次是客户端收到服务器的确认后向服务器发送确认。TCP的三次握手其实就是保证我们这是一个可靠的连接，最关键的一个问题就是我们为什么需要第三次握手，因为建立连接前两次就足够了，那么保证可靠就全靠第三次握手，这里有两点原因：（1）服务器向客户端发送确认后，如果客户端没有收到，服务器会进入Established状态，认为连接已经建立完毕，但客户端并没有建立连接。（2）之前失效的连接请求报文段在网络延迟后突然又传送到了服务器，服务器收到失效的连接请求报文段后，就误认为是客户端再次发出的一个连接请求，从而进入Established状态。

未连接队列：

在三次握手协议中，服务器会维护一个未连接队列，该队列为每个客户端的SYN包开设一个条目，该条目表明服务器已收到SYN包，并向客户发出确认，正在等待客户的确认包。这些条目所标识的连接在服务器处于 Syn_RECV状态，当服务器收到客户的确认包时，删除该条目，服务器进入ESTABLISHED状态。

四次挥手：

因为TCP是全双工的连接，当客户端断开了连接之后，两个方向的关闭都需要经历请求断开+确认断开，故需要四次挥手。

服务器可以主动断开连接吗：

可以主动断开连接，但是一般情况下并不会主动断开连接。主动断开的一方不会立即进入CLOSED状态，需要经历2MSL（Maximum Segment Lifetime）的TIME_WAIT状态。当服务器主动断开连接后，会进入TIME_WAIT状态，此时服务器无法进行其他操作。

为什么需要TIME_WAIT状态：
（1）确保客户端发送的确认报文正确到达，若没有正确到达，服务器会在等待超时后再次发送请求断开报文。
（2）确保此次连接中因意外情况延迟到达的报文消逝，避免在串入到下次连接中。

MTU(最大传输单元，网络层)—1500字节

MSS(最大分段大小，应用层)

状态：

建立连接

服务器	Closed	Listen	SYN_RECV	ESTABLISHED
客户端	Closed	SYN_SENT	ESTABLISHED	

断开连接

服务器	ESTABLISHED	CLOSE_WAIT	LAST_ACK	CLOSED	
客户端	ESTABLISHED	FIN_WAIT1	FIN_WAIT2	TIME_WAIT	CLOSED

(2) UDP

特点：

1. 无连接
2. 不可靠交付
3. 速度快
4. 传输数据包

头部格式：

(3)客户端突然断开

进程会忽然关闭，如何解决。

服务器给客户端发送文件，如果发送过程中，客户端断开，服务器仍在发送文件，第一次send会返回-1，第二次send时调用send的子进程会收到SIGPIPE信号，子进程收到信号终止之后，父子进程间的管道会变成就绪状态，父进程的epoll会一直唤醒。为了避免子进程收到信号终止，每次send判断send的返回值是否为-1，如果是-1，就停止发送，直接返回。

5. 进程间通信

管道、共享内存、信号量、消息队列、信号、socket

(1) 管道

无名管道(PIPE)：

1. 只能用于具有**亲缘关系**的进程之间
2. **动态存在于内存中**，程序结束即消失
3. **半双工**，读端保存在fds[0]，写端保存在fds[1]
4. 数据流，管道中的数据没有边界
5. 管道先关闭读端时，对方进程再去写管道时会受到SIGPIPE信号
6. 管道先关闭写段时，对方进程再去读管道，read不会阻塞，返回0

命名管道(FIFO)：

1. 可以用于**没有亲缘关系**的进程之间
2. 是**保存在磁盘**上的文件，不会随着进程的结束而消失

区别：

1. 是否只可以作用不具有具有亲缘关系的进程之间
2. 是否随着进程的结束而消失

(2) 共享内存(shm)

定义：

共享内存是一段特殊的内存区域，可以被多个进程共享，进程把这段共享内存 映射到自己的进程地址空间里，才能使用。进程间需要通信时，就把数据写到 共享内存里。 共享存创建之后，一直存在，不会随着进程的结束而消失，直到 用命令把它删除，或者系统重启。

缺点：

1. 方向性，如果用于两个进程通信，可能会因为同时对内存区进行操作从而导致数据混乱，我们可以参考管道的解决方法，开辟两个共享内存，分别用于单向的传输，可以有效解决数据混乱。
2. 因为共享内存一般都是比较大的空间，那么我们用完一定要记得释放，防止内存泄漏。

(3) 信号量(sem)

函数 semget 创建一个信号量集或访问一个已存在的信号量集。

PV操作

临界资源—可以共享，但必须互斥访问

临界区—访问临界资源的代码

(4) 消息队列(msg)

定义：

特点：

1. 队列结构，先进先出，
2. 命令 删除 ipcrm -q msgid
3. 解耦（解除耦合）—两个类之间的联系的紧密程度

接收消息的特点：

1. 可以按照**特定的消息类型**来接收消息。
2. 当我们希望接收的特定类型的消息不存在的时候，msgrcv会阻塞。
3. msgrcv的第四个参数，mtype的值如果填的是0，表示按照先后顺序去接收所有类型的消息。
4. msgrcv的第四个参数，mtype的值如果是一个小于0的数字，接收消息的时候会把小于这个数字的绝对值的消息类型取出来，取出来的时候优先会取消息类型最小的消息。

(5) 信号

收到信号后的相应方式：

1. 接收默认处理：终止进程、忽略信号、暂停运行、继续运行
2. 忽略信号
3. 捕捉信号并处理
4. SIGKILL和SIGSTOP不能被捕捉，也不能被忽略

信号处理函数：

signal函数

sigaction函数

1. 信号的注册，用signal或者sigaction，注册一次，一直生效，如果想改为注册一次，只生效一次，可以用sigaction函数，sa_flags加上SA_RESETHAND，处理了一次信号之后，对于该信号的处理方式改回默认处理，下一次再有该信号到达时，按照默认方式处理。
2. 如果正在执行一个信号的处理函数，此时又来了一个/多个相同的信号，原来的信号处理函数不会被打断，当原来的信号处理函数执行完毕之后，才会处理新到的信号，并且只会处理一次。如果此时又来了不同类型的另一个信号，会先去处理新的信号，处理完新的信号，再继续执行原来的信号处理函数。signal和sigaction默认行为是一样。如果把sigaction结构体中的sa_flags加上SA_NODEFER，信号和信号之间都会互相打断，不断重

入，递归的效果。

3. 如果程序阻塞在read系统调用上面，此时有信号发生，如果是signal函数注册的信号处理函数，执行完信号处理函数之后，会回到read上面继续阻塞，如果是sigaction函数注册的，执行完信号处理函数之后，不会阻塞在read上面，read函数会返回-1。如果把sigaction结构体中的sa_flags加上了SA_NODEFER加上SA_RESTART，执行完信号处理函数之后，会回到read上面继续阻塞。

信号的阻塞或挂起：

1. sigaction，设置sa_mask掩码，只能在信号处理函数内阻塞其他信号。
2. sigprocmask 全局的阻塞，在信号处理函数之外，也可以阻塞信号。

\1. 可以按照特定的消息类型来接收消息。
\2. 当我们希望接收的特定类型的消息不存在的时候，msgrcv会
阻塞。
\3. msgrcv的第四个参数，mtype的值如果填的是0，表示按照
先后顺序去接收所有类型的消息。
\4. msgrcv的第四个参数，mtype的值如果是一个小于0的数
字，接收消息的时候会把小于这个数字的绝对值的消息类型
取出来，取出来的时候优先会取消息类型最小的消息。

(6) 零拷贝

系统的性能瓶颈往往是网络IO
read+send的缺点：先从内核缓冲区读到用户态缓冲区，又从用户态缓冲区拷贝到socket缓冲区，多次拷贝，效率低。
mmap方式：将内核缓冲区与用户共享，这样能减少内核缓冲区到用户缓冲区的之间的复制。
sendfile：
sendfile在内核里把数据从一个描述符缓冲区拷贝到另一个描述符缓冲区，不需要经历，从内核拷贝到用户空间的过程，因此效率更高。
sendfile的一个限制是只能用于发送端，另一个限制是不能发送超过2G的文件。
splice：
splice可以从两个文件描述符之间拷贝数据而不用经过用户地址空间
splice需要用管道做一个缓冲，每次先拷贝到管道中，再从管道中拷贝到文件缓冲区中

6. 多进程编程、多线程编程

互斥锁、条件变量、读写锁、线程池

7. 五大IO模型

同步、异步、阻塞、非阻塞、信号驱动

同步/异步关注的是用户态进程/线程(程序)的状态，其要访问的数据是否就绪，进程/线程是否需要等待；

阻塞/非阻塞关注的是消息通信机制，访问数据的方式，同步需要主动读写数据，在读写数据的过程中还是会阻塞；异步只需要关注I/O操作完成的通知，并不主动读写数据，由**操作系统内核**完成数据的读写。

同步：主动读写数据

异步：关注I/O操作完成的通知，由操作系统内核完成数据的读写

阻塞：访问的数据未就绪，进程/线程需要等待

非阻塞：访问的数据就绪，进程/线程不需要等待

8. IO复用机制

select、epoll、poll

select：
（1）最大并发数受到限制，最多有1024位
（2）效率低，每次都会线性扫描整个fd_set,集合越大速度越慢
（3）每次调用select，都会发生用户态的数据拷贝到内核空间，select返回时，又需要从内核空间拷贝到用户空间
epoll：
（1）无最大并发连接的限制
（2）只有活跃的socket才会主动地去调用callback函数
（3）epoll_ctl时把描述符添加到红黑树，不需要每次都拷贝
只有当高并发但是少数socket活跃时epoll的效率高于select。

（1）select，单个进程打开的FD是有限制的，默认值为1024；对socket进行扫描时是线性扫描；每次调用select都会发生用户态与内核态的数据拷贝。
（2）poll，因为是基于链表存储的所以没有最大连接数的限制；对socket进行扫描时是线性扫描；水平触发方式，若本次报告了fd后没有被处理，下次poll时会再次报告fd。
（3）epoll，连接数有上限但是很大，1G内存的机器上可以打开10W左右的连接；只有活跃的socket才会主动调用callback函数；边沿触发方式；底层使用红黑树存储，不需要每次都进行拷贝。

水平触发：满足条件就会通知，比如100字节，读了50字节，剩余50字节，还会触发

边沿触发：每当状态变化时触发一次，比如100字节，触发一次读了50字节，剩余50字节，不会再触发

9. GDB调试

编译选项加-g，打开调试选项

操作指令：

run—运行程序

break—设置断点

delete—删除指定的断点

continue—运行到下一个断点

next—单步调试，跳过函数声明—F10

step—单步调试，不跳过函数声明—F11

info—查看断点信息

ignore—忽略断点

tbreak—设置临时断点，触发断点以后会被自动删除

disable—停止指定的断点

调试段错误：

1. ulimit -a 查看当前系统的各项属性的大小限制
2. ulimit -c unlimited 设置core file size为不限制大小
3. gdb ./test core 再次运行程序，会产生core文件，通过gdb可执行core文件，进行调试。通过**bt(breaktrace)**可以看到程序段错误时的现场。

10. Makefile工具链

规则、目标、依赖文件

11. 高性能IO两种模式

Reactor

Proactor（linux下缺少异步IO支持，基本没有Proactor）

12. 内存管理

它使得应用程序认为它拥有连续可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

虚拟地址和物理地址

页—4K

MMU（内存管理单元，完成虚拟地址到物理地址的映射）

Lazy模式—只有写数据的时候才会真正的分配物理页

缺页异常—写数据的时候，如果虚拟地址和物理地址此时还没有建立映射关系，会发生缺页异常，系统才会分配物理页，建立映射关系。

TLB（快表）—保存的内容是页表的副本，每次要找虚拟地址和物理地址的映射关系时，先访问快表，如果快表中存在映射关系，就不需要到内存中的快表找了，这样加快了访问速度，因为TLB的速度比内存快。

13. 排查内存泄漏

Linux环境下工具：valgrind

编译选项：valgrind --tool=memcheck --leak-check=yes ./test

可以探测到的问题：使用未初始化的内存、内存读写越界、内存覆盖、动态内存管理错误、内存泄漏

其他选项：

callgrind ------> 它主要用来检查程序中函数调用过程中出现的问题。

cachegrind ------> 它主要用来检查程序中缓存使用出现的问题。

helgrind ------> 它主要用来检查多线程程序中出现的竞争问题。

massif ------> 它主要用来检查程序中堆栈使用中出现的问题。

extension ------> 可以利用core提供的功能，自己编写特定的内存调试工具

六、C++

1. C++11标准

2. STL

(1) 容器

1. 线性容器：

（1）vector

动态数组

不可以头部插入和删除：底层是数组，移动数据耗费的时间复杂度过高

三个指针：

1. _M_start：指向第一个元素的位置
2. _M_finish：指向最后一个元素的**下一个位置**
3. _M_end_of_storage：指向当前**分配空间的最后一个位置的下一个位置**

类型萃取：帮助我们提取出自定义类型进行深拷贝，而内置类型统一进行浅拷贝，也就是所谓的值拷贝。

迭代器失效问题：扩容会导致数据资源转移，而迭代指向的是内存地址，进行资源转移后迭代器依然指向之前的内存地址。当删除容器内的元素后，会导致删除之后的元素前移，从而导致删除之后的所有迭代器失效

（2）deque

双端队列

（3）forward_list

单向链表

支持前向访问迭代器

（4）list

双向链表

2. 关联式容器：

（1）map

映射	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
map	红黑树	key有序	key不可重复	key不可修改	O(logN)	O(logN)
multimap	红黑树	key有序	key可重复	key不可修改	O(logN)	O(logN)
unordered_map	哈希表	key无序	key不可重复	key不可修改	O(1)	O(1)

insert和下标插入的区别

insert：返回值是一个pair，pair的first是一个迭代器，pair的second成员是一个bool类型变量，如果关键字已在map中，返回false；如果不在，则返回true插入成功。

下标插入：关键字已在map中，则更新value；如果不在，则插入新值。

（2）set

集合	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
set	红黑树	有序	否	否	O(logN)	O(logN)
multiset	红黑树	有序	是	否	O(logN)	O(logN)
unordered_set	哈希表	无序	否	否	O(1)	O(1)

(2) 迭代器

随机访问迭代器：支持vector、deque

双向迭代器：支持list、set、map

前向迭代器：支持unordered_set、unordered_map

输出迭代器

输入迭代器

流迭代器

(3) 适配器

容器适配器：stack、queue、priority_queue

迭代器适配器：流迭代器、反向迭代器、插入迭代器

函数适配器：std::bind()、std::mem_fn

(4) 算法

通过迭代器对容器中的元素进行操作，在操作时只和迭代器进行交互，实现了算法与容器的解耦

非修改式算法(for_each)

修改式算法(copy/remove if)

(5) 函数对象

std::function()

可调用对象的包装器。

可调用对象包括lambda表达式、函数指针、类成员指针、仿函数（具有成员函数的类对象）、可被转换成为函数指针的类对象。

std::function的实例可以对任何可以调用的目标实体进行存储、复制、和调用操作，这些目标实体包括普通函数、Lambda表达式、函数指针、以及其它函数对象等。

std::bind()

是用来绑定函数调用的某些参数的，bind()接受的第一个参数f，必须是一个可调用的对象可以是函数、函数指针、函数对象和成员函数指针，绑定完成后，bind会返回一个函数对象，它内部保存了f的拷贝，返回值类型被自动推导为f的返回值类型。

绑定类的非静态成员函数时，参数列表需要加上this指针。

占位符(placeholders)，按照序号匹配参数位置，也就是代表函数形参的位置。

bind绑定是以值传递的方式进行，即使参数时某个变量的引用，也不会因此修改。

可通过std::cref()引用包装器使用变量的引用。

std::function()+std::bind()实现回调函数

实现多态，比纯虚函数 + 继承要简单，灵活

基于对象的方式实现多态

例子：

店员那里留下了你的电话，过了几天店里有货了，店员就打了你的电话，然后你接到电话后就到店里去取了货。

在这个例子里，你的电话号码就叫`回调函数`，你把电话留给店员就叫`登记回调函数`，店里后来有货了叫做`触发回调事件`，店员给你打电话叫做`调用回调函数`，你到店里去取货叫做`响应回调事件`。

std::mem_fn()

lambda表达式（匿名函数）

```
//捕获列表，参数列表，函数选项，返回类型，函数体
[capture](parameters) mutable ->return-type{statement}

//捕获值列表能捕获的值是所有在此作用域可以访问的值，包括这个作用域里面的临时变量，类的可访问成员，全局变量。
//捕获值的方式分两种，一种是按值捕获，一种是按引用捕获。
//捕获列表的形式
1.[var]表示值传递方式捕捉变量var；
2.[=]表示值传递方式捕捉所有父作用域的变量（包括this）；
3.[&var]表示引用传递捕捉变量var；
4.[&]表示引用传递方式捕捉所有父作用域的变量（包括this）；
5.[this]表示值传递方式捕捉当前的this指针；
6.捕获列表为空时表示不捕捉；
7.捕获列表也可以联合使用值传递和引用传递方式。
```

(6) 空间配置器

3. 设计模式

设计模式原则：

单一职责原则—单个类，最好只做一件事，只有一个引起它变化的原因。

开闭原则—对扩展开放，对修改闭合。

里氏替换原则—派生类必须能够替换其基类。

接口分离原则—使用多个小的专门的接口，而不要使用一个大的总接口。

依赖倒置原则—面向接口编程，依赖于抽象。

类与类之间的关系：

继承、关联、聚合、组合、依赖

设计模式分类：

创建型设计模式：单例、工厂

结构型设计模式：适配器、组合、代理模式（写时复制COW，区分是读还是写）

行为设计模式：迭代器模式、观察者模式

(1) 单例模式

实现步骤：

1. 将构造函数私有化
2. 在类中定义一个静态的指向本类型的指针变量
3. 定义一个返回值为类指针的静态成员函数

代码实现：

```
class Singleton{
public:
    static Singleton *getInstance(){
        if(nullptr == _pInstance){
            _pInstance = new Singleton();
        }
        return _pInstance;
    }
    static void destroy(){
        if(_pInstance){
            delete _pInstance;
            _pInstance = nullptr;
        }
    }
private:
```

```
    Singleton(){
        cout << "Singleton()" << endl;
    }
    ~Singleton(){
        cout << "~Singleton()" << endl;
    }
private:
    static Singleton *_pInstance;
};

Singleton *Singleton::_pInstance = nullptr;

int main()
{
    Singleton *ps1 = Singleton::getInstance();
    Singleton *ps2 = Singleton::getInstance();
    cout << "ps1 = " << ps1 << endl
         << "ps2 = " << ps2 << endl;
    Singleton::destroy();
    return 0;
}
```

自动释放：

1. 友元

```
class AutoRelease
{
    public:
    AutoRelease()
    {
        cout << "AutoRelease()" << endl;
    }
    ~AutoRelease()
    {
        cout << "~AutoRelease()" << endl;
        if (Singleton::_pInstance)
        {
            delete Singleton::_pInstance; //1、调用析构函数 2、operator delete
            Singleton::_pInstance = nullptr;
        }
    }
};内部类+静态成员的方法
```

2. 内部类+静态成员的方法

   当设置为私有时，要创建一个全局AutoRelease类对象(如果不是全局对象也属于Singleton类对象的一部分，这时如果想执行析构函数，两个对象的析构函数就会相互等待)

3. atexit函数

先注册，进程结束时才执行

线程不安全

饿汉/饱汉(懒汉):

1. 饿汉

   _pInstance初始化时直接getInstance()

2. 饱汉

   _pInstance初始化时直接nullptr，多线程下不安全

3. pthread_once

   保证多线程下安全

```
class Singleton
{
public:
    static Singleton *getInstance()
  {
    pthread_once(&_once, init);
    return _pInstance;
  }
    static void init()
  {
    _pInstance = new Singleton();
    atexit(destroy);
  }
    static void destroy()
  {
    if(_pInstance)
    {
      delete _pInstance;
      _pInstance = nullptr;
    }
  }
private:
    static Singleton *_pInstance;
  static pthread_once_t _once;
}
Singleton *Singleton::_pInstance = nullptr; //饱汉模式(懒汉模式)
/* Singleton *Singleton::_pInstance = getInstance();//饿汉模式 */
pthread_once_t Singleton::_once = PTHREAD_ONCE_INIT;
```

(2) 工厂模式

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，工厂模式使其创建过程延迟到子类进行。

在工厂模式中创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

扩展性高、屏蔽了产品的具体实现、但每次增加一个产品都需要增加一个具体类和对象实现工厂导致系统中类的个数成倍增加。

(3) 观察者模式

用途：

观察者模式定义了一个一对多的依赖关系，让一个或多个观察者对象监察一个主题对象，这样一个主题对象在状态上的变化能够通知所有的依赖于此对象的那些观察者对象，使得这些观察者对象能够自动更新。

一个主题Subject可以有任意数目依赖它的Observer，一旦Subject的状态发生改变，所有Observe都可以得到通知。

架构：

```
class Subject
{
public:
    virtual ~Subject() {}

    void attach(Observer*);//添加到观察者队列
    void detach(Observer*);//从观察者队列移除
    void notify();//通知观察者队列上的每个观察者

    virtual void setStatus(Status status);
    Status getStatus() const { return _status; }
protected:
    std::list<Observer*> _obList;//观察者队列
    Status _status;
};

class Observer
{
public:
    Observer(const string & name);
    virtual ~Observer() {}

    virtual void update() = 0;
protected:
    string _name;
};
```

(4) 代理模式

解决自定义string写时复制区分读写问题，自定义string时需要重载下载访问运算符，需要区分读写。

七、数据库

1. MySQL

（1）MySQL有哪些常用存储引擎，各自有什么特点。

MyISAM：查询速度比较快；支持表锁；支持全文索引；不支持事务；采用非聚集索引（数据和索引分开存放）
InnoDB：5.5以及之后的版本默认的存储引擎；支持事务；支持表锁和行锁；支持MVCC（多版本并发控制）；支持外键约束；采用聚集索引（索引和数据保存在一起）
Memory：数据保存在内存中，故数据库重启后数据会消失；支持表锁；用temporary关键字可以创建临时表，但只在当前的连接中看到；用memory创建的表，在不同的连接中都可以看到。

（2）MySQL中的锁是怎么分类的？各自的特性如何？
（1）按照对数据的锁定范围划分：
（a）行级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。
（b）表级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。
（2）按照对数据的访问类型划分：
（a）读锁：同一份数据，多个读操作可以同时进行而互不影响
（b）写锁：当前操作没有完成之前，它会阻断其他读锁和写锁。

（3）数据库设计的三大范式是什么？反范式设计是什么？各自的优缺点有哪些？
（1）第一范式：表中的每一个字段原子性不可再分。
（2）第二范式：所有非主键字段完全依赖主键，不能产生部分依赖。
（3）第三范式：需要确保数据表中的所有非主键字段直接依赖主键，不能产生传递依赖。
（4）反范式设计：允许存在少量冗余，以空间换时间。
（5）范式设计优缺点：
（a）优点：可以尽量减少数据的冗余，符合范式设计的表更小，更新数据时速度也会更快。
（b）缺点：范式化的表尝尝需要很多join关联，增加了查询的代价；不容易做索引优化。
（6）反范式设计优缺点：
（a）优点：可以减少表的关联，提高效率，能够更好地做索引优化。
（b）缺点：存在数据冗余，使得数据的维护和修改成本更高。

（4）什么是聚集索引，什么是辅助索引？回表和覆盖索引分别是什么？
（1）聚集索引：
（a）数据跟索引保存在一个文件中，InnoDB存储引擎的表都是采用这种数据结构。
（b）聚集索引中叶子结点保存key值和一整行完整的数据。
（c）InnoDB的表中，有且只有一个聚集索引。
（2）辅助索引：
（a）叶子结点上面保存的是key值和对应的主键ID
（b）每张表可以有多个辅助索引
（3）回表：当通过辅助索引来寻找数据的时候，InnoDB会遍历辅助索引，并通过辅助索引的叶子结点，获取主键。然后再通过聚集索引来找到一个完整的行记录。
（4）覆盖索引：从辅助索引中就可以得到要查询的信息，而不需要回表。

1. **使用数据库有什么优势?**
   可以大大提高应用开发的效率，方便用户使用，减轻数据库系统管理人员维护负担。因为在数据库系统中应用程序不必考虑数据的定义、存储和数据存储的具体途径，这些工作都由数据库管理系统完成。开发人员就可以专注于应用逻辑的设计，而不必聚焦于数据管理的许许多多复杂的细节。

2. **设计表时，数据类型如何选择?**

   优先考虑数字类型，其次是日期或者二进制类型，最后是字符类型。对于相同级别的数据类型，应该优先选择占用空间小的数据类型。

3. **MyISAM与InnoDB各自有什么特点?**
   MyISAM：查询速度比较快，支持表锁，支持全文索引，不支持事务，采用非聚集索引
   InnnoDB：支持事务，支持表锁和行锁，支持MVCC（多版本并发控制），支持外键约束，采用聚集索引

4. **MyISAM索引与InnoDB索引的区别?**
   （1）MMyISAM是非聚集索引；InnoDB是聚集索引。
   （2）MyISAM索引的叶子结点存储的是行数据地址，需要再寻址一次才能得到数据；InnoDB的主键索引的叶子结点存储着行数据，因此主键索引非常高效。
   （3）InnoDB非主键索引的叶子结点存储的是主键和其他带索引的列数据，因此查询时做到覆盖索引会非常高效。

5. **创建索引的原则有哪些?**
   （1）最左前缀匹配原则
   （2）较频繁作为查询条件的字段才可以去创建索引
   （3）更新频繁字段不适合创建索引
   （4）若是不能有效区分数据的列不适合做索引列
   （5）尽量的扩展索引，不要新建索引
   （6）定义有外键的数据列一定要建立索引
   （7）对于那些查询中很少涉及的列，重复值较多的列不适合建立索引
   （8）对于定义为text、image和bit的数据类型的列不要建立索引

6. **索引的使用场景有哪些?**
   （1）where
   （2）order by
   （3）join
   （4）索引覆盖

7. **创建索引时需要注意什么?**
   （1）非空字段，应该指定列为NOT NULL，除非想要存储NULL。含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。
   （2）取值离散大的字段的列放到联合索引的前面，可以通过count（）函数查看字段的差异值。
   （3）索引字段越小越好，数据库的数据存储以页为单位一页存储的数据越多一次IO操作获取的数据越大，效率越高。

8. **使用索引查询一定能提高查询的性能吗? 为什么?**

   几乎在所有的情况下是可以提高查询到的性能的，索引就是事先排好序，从而在查找时可以应用二分查找等高效率的算法。

9. **什么是最左前缀原则? 什么是最左匹配原则**
   最左前缀原则：先看第一列，在第一列满足的条件下再看下一列，以此类推

最左匹配原则：最左优先，以最左边的为起点任何连续的索引都能匹配上。同时遇到范围查询(>、<、between、like)就会停止匹配。

10. **非聚簇索引一定会回表查询吗?**
    不一定，这涉及到查询语句所要求的字段是否全部命中了索引，如果全部命中了索引，那么就不必再进行回表查询。
11. **MySQL中InnoDB引擎的行锁是怎么实现的?**
    基于索引实现的，对索引到的列完成行锁锁定。
12. **B树和B+树的区别?**
    （1）在B树中，你可以将键和值存放在内部节点和叶子结点；但在B+树中，内部节点都是键，没有值，叶子结点同时存放键和值。
    （2）B+树的叶子结点有一条链相连，而B树的叶子结点各自独立。
13. **如何定位及优化SQL语句的性能问题？创建的索引有没有被使用到?或者说怎么才可以知道这条语句运行很慢的原因?**
    使用执行计划

    通过type字段可以判断是否使用了常见的索引
14. **为什么要尽量设定一个主键?**

    主键是数据库确保数据行在整张表唯一性的保障，设定了主键之后，在后续的删查改可以更加快速以及确保操作数据范围的安全。
15. **统计过慢查询吗？对慢查询都怎么优化过?**

    set global slow_query_log 开启慢查询

    直接分析慢查询日志，利用explain关键字模拟优化器执行SQL语句，分析慢查询语句。

    常见的慢查询优化：

    （1）索引没起到作用：模糊匹配中占位符在第一个位置

    （2）优化数据库结构

    （3）分解关联查询：大查询分解为多个小查询

    （4）优化limit分页

2. Redis

（1）什么是Redis?有哪些有优缺点?

1. Redis是一款key-value型的非关系型数据库(NoSQL)，全程为Remote Dictionary Service（远程字典服务）。
2. 优点：高性能，官方公布的数据为每秒读11万次，每秒最高写8.1万次；支持丰富的数据类型；每个操作都是原子性的；丰富的特性，比如key的过期时间。Redis的数据存在内存中，所以读写速度非常快。
3. 缺点：数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。缓存击穿、缓存穿透、缓存雪崩

（2）为什么要用 Redis /为什么要用缓存

1. 关系型数据库的缺点：通常通过数据驱动来链接数据库进行增删查改，数据全都存储于硬盘之上，当网站的访问量非常大时，我们的数据库压力就变大了。数据库的连接池、处理数据的能力就会面临很大的挑战。
2. Redis/缓存的优点：缓存就是在内存中存储的数据备份，当数据没有发生本质变化的时候，我们避免数据的查询操作直接连接数据库，而是去内存中读取数据，这样就大大降低了数据库的读写次数，而且从内存中读取数据的速度要比从数据库查询快很多。

   高性能：

普通数据库从硬盘读取，Redis从缓存中读取。

高并发：

直接操作缓存能够承受的请求是远远大于直接访问数据库的，我们可以考虑把数据库中的部分数据转移到内存中，这样用户的一部分请求会直接到缓存这里而不用经过数据库。

（3）Redis为什么这么快

- 完全基于内存，Redis将数据存储在内存里面，读写数据的时候都不会受到硬盘I/O速度的限制，所以速度极快，类似于HashMap。
- 数据结构简单，对数据操作也简单，Redis的数据结构是专门进行设计的。
- 采用单线程，避免了不必要的上下文切换和竞争条件。

（4）Redis有哪些数据类型

string、list、set、zset、hash

（5）什么是Redis持久化?

Redis的数据是存储在内存中，为了避免进程退出导致数据的永久丢失，需要**定期**将Redis中的数据以某种形式从内存中**保存到硬盘**，当下次Redis重启时，利用持久化文件实现**数据恢复**，也可以用与灾难备份。

（6）Redis 的持久化机制是什么？各自的优缺点?

1．RDB(Redis DataBase)：将Redis在内存中的数据库记录定时dump到磁盘上。

优点：RDB可以最大化Redis的性能，**父进程在保存RDB文件时唯一要做的就是fork出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘I/O操作**。RDB在恢复大数据集时的速度比AOF的恢复速度要快。

缺点：Redis需要设置不同的保存点来控制保存RDB文件的频率，但是，因为RDB文件需要保存整个数据集的状态，所以它并不是一个轻松地操作，系统不能过于频繁的保存。所以一旦发生故障停机，我们可能会丢失好几分钟的数据。

2．AOF：将每次执行的写命令保存到硬盘，在服务器启动时，通过重新执行这些命令来还原数据集。

优点：AOF的默认策略为每秒钟fsync一次，所以就算发生故障停机，也最多只会丢失一秒钟的数据。即使日志因为某些原因而包含了未写入完整的命令（比如磁盘已满，写入中停机等情况），也可以采用redis-check-aof修复。Redis也可以在AOF文件体积变得过大时，自动地在后台对AOF进行重写， 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。

缺点：对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB 。

（7）Redis key的过期时间和永久有效分别怎么设置?

expire key ttl：经过ttl秒后，key将自动被删除

pexpire key ttl：经过ttl毫秒后，key将自动被删除

persist key：永久有效

（8）Redis事务的概念

Redis事务可以一次执行多个命令，本质是一组命令的集合。一个事务中的所有命令都会序列化，按照顺序的串行化执行，而不会被其他命令插入，不允许加塞。

（9）Redis事务的三个阶段

1. 开始事务
2. 命令入队
3. 执行事务

（10）Redis事务相关命令和事务的特征?

1. 相关命令：

开启事务：MULTI

执行事务：EXEC

丢弃事务：DISCARD

监视一个（多个）事务：WATCH key [key ...]（如果事务执行前key被其他命令所改动，那么事务将被打断

取消监视：UNWATCH

2. 特征：

收到EXEC命令进入事务执行，事务中任意命令执行失败，其余的命令依然被执行。

（11）缓存穿透是什么？如何解决?

要查询的数据不存在，缓存无法命中所以查询数据库，但数据库中也不存在该数据，这样每次对数据的查询都会穿过缓存区查询一次数据库。

解决方案：查询时做一些校验和过滤，判断是否为正常的查询；缓存空对象，就算数据库中不存在这个数据，我们也在缓存中保存这个key，但是把value值记录为不存在，下次再访问时就可以快速查询完毕；预先将数据库里面的所有key存入一个大的map里，然后再过滤器中过滤掉那些不存在的key，不过需要考虑map的更新频率问题。

（12）缓存击穿是什么？有什么解决方案?

当热点数据key从缓存内失效时，大量访问同时请求这个数据，就会将查询下沉到数据库层，此时数据库的负载压力就会骤增。

解决方案：延长热点key的过期时间或者设置永不过期；利用互斥锁保证同一时刻只有一个客户端可以查询底层数据库的这个数据，一旦查到数据就会缓存至Redis内，避免其他大量请求同时穿过Redis访问底层数据库。

（13）缓存雪崩的概念？如何应对?

缓存中大量的数据在同一时间失效，此时相当于没有缓存，所有对数据的请求直接到数据库，会带来很大压力。

解决方案：将缓存失效时间分散开，我们可以在原有的失效时间基础上增加一个随机值，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的时间；或者不设置缓存的过期时间，有更新操作时就把热点的缓存全部更新。