1.两道算法：

- **计算n x m的棋盘格子（n为横向的格子数，m为竖向的格子数）沿着各自边缘线从左上角走到右下角，总共有多少种走法，要求不能走回头路，即：只能往右和往下走，不能往左和往上走【**动态规划】

```
int Go(int m, int n ) {
    if (m < 0 || n < 0) {
        return 0;
    }
    else if (n == 1 || m == 1) {
        return 1;
    }
    else
        return Go(n - 1, m) + Go(n, m - 1);
}
```

- **传入一个int型数组，返回该数组能否分成两组，使得两组中各元素加起来的和相等，并且，所有5的倍数必须在其中一个组中，所有3的倍数在另一个组中（不包括5的倍数），能满足以上条件，返回true；不满足时返回false【**动态规划】

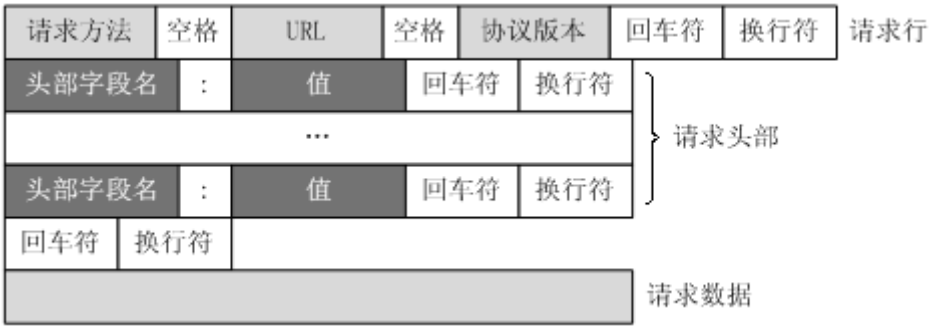
```
import java.util.ArrayList;
import java.util.Scanner;

//思想：将能整除3或者5的各自分为一组，记为sum1和sum2，剩余的保存在数组nums里
//现有两组，剩余nums里的数相加或减的值result 等于 abs(sum1 - sum2)即可
//最终nums里的数字全部组合，若 result == abs(sum1 - sum2)，则返回true，否则，返回false

public class Main {
    public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    while(scanner.hasNext()){
        int n = Integer.parseInt(scanner.nextLine());
        int[] arr = new int[n];
        String[] s = scanner.nextLine().split("\\s");
        for(int i=0;i<n;i++){
            arr[i] = Integer.parseInt(s[i]);
        }
        System.out.println(depart(arr,n));
    }
}

private static boolean depart(int[] arr,int n) {
    int[] nums = new int[n];    // 存不能能整除3或者5
    int count = 0;
    int sum1 = 0,sum2 = 0;
    for(int i=0;i<n;i++){
        if(arr[i] % 3 == 0){
            sum1 += arr[i];     // 能能整除3的数之和
        }else if(arr[i] % 5 == 0){
            sum2 += arr[i];     // 存不整除5的数之和
        }else{
```

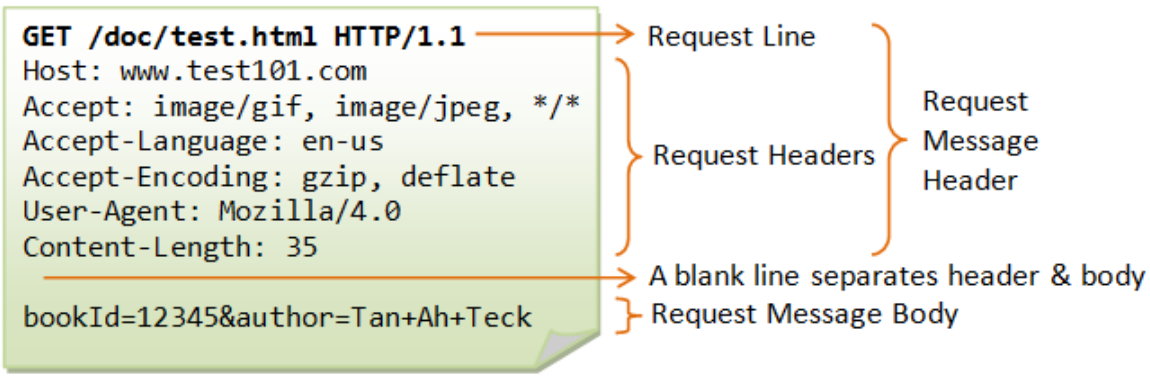
```
                nums[count++] = arr[i];
            }
        }

        int sum = Math.abs(sum1 - sum2);
        int i = 0;
        int result = 0;  //不能能整除3或者5 的组合值
        return f(i,count,nums,sum,result);
    }

    private static boolean f(int i, int count, int[] nums, int sum,int result) {
        if(i == count){
            return result == sum;
        }else{
            int result1 = result + nums[i];
             int result2 = result - nums[i];
            i=i+1;
            return (f(i,count,nums,sum,result1) || f(i,count,nums,sum,result2));
        }
    }

    }
```

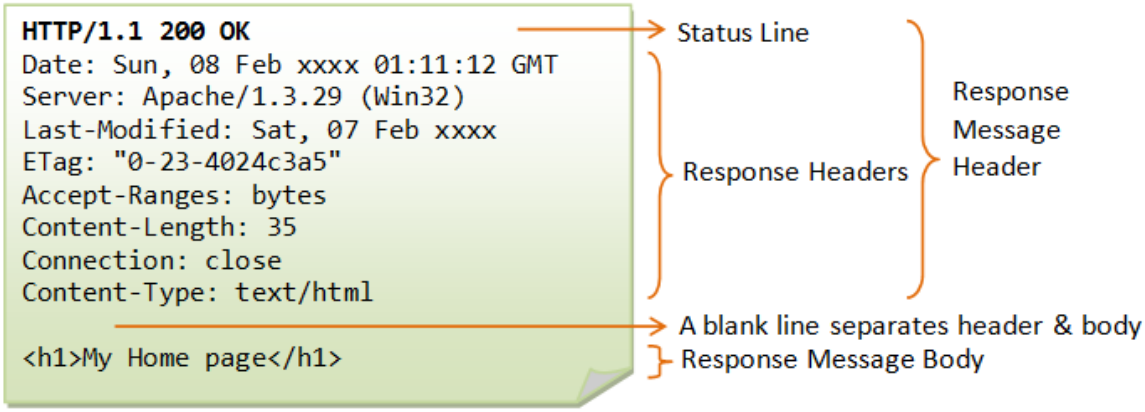
2.常见的拥塞控制算法

慢开始、拥塞避免、快重传、快恢复

3.http协议的具体内容，怎么实现的，里面的字段，文本协议的定义







第一部分：请求行，用来说明请求类型,要访问的资源以及所使用的HTTP版本.

第二部分：请求头部，用来说明服务器要使用的附加信息

HOST将指出请求的目的地.User-Agent,服务器端和客户端脚本都能访问它,它是浏览器类型检测逻辑的重要基础.该信息由你的浏览器来定义,并且在每个请求中自动发送等等

第三部分：空行，请求头部后面的空行是必须的

即使第四部分的请求数据为空，也必须有空行。

第四部分：请求数据也叫主体，可以添加任意的其他数据。

文本协议是在进行网络传输时，传输的是类似JSON，XML这样的文本文件，而不是二进制文件（可靠性高，常见的二进制协议有TCP、UDP、IP等），可读性比较高，例如HTTP、FTP、TELNET。

4.首部和报文是如何区分的

首部和报文中间会有一个空行

5.RESTful

REST全称是Representational State Transfer，表述性状态转移。如果一个架构符合REST的约束条件和原则，我们就称它为RESTful架构。使用这样一个风格的约束可以提高可见性。

遵循了REST原则后：

看Url就知道要什么
看http method就知道干什么
看http status code就知道结果如何

6.数据库范式

第一范式：属性是最小的字段

第二范式：非主属性完全依赖于键码

分解前

Sno	Sname	Sdept	Mname	Cname	Grade
1	学生-1	学院-1	院长-1	课程-1	90
2	学生-2	学院-2	院长-2	课程-2	80
2	学生-2	学院-2	院长-2	课程-1	100
3	学生-3	学院-2	院长-2	课程-2	95

以上学生课程关系中，{Sno, Cname} 为键码，有如下函数依赖：

- Sno -> Sname, Sdept
- Sdept -> Mname
- Sno, Cname-> Grade

Grade 完全函数依赖于键码，它没有任何冗余数据，每个学生的每门课都有特定的成绩。

Sname, Sdept 和 Mname 都部分依赖于键码，当一个学生选修了多门课时，这些数据就会出现多次，造成大量冗余数据。

分解后

关系-1

Sno	Sname	Sdept	Mname
1	学生-1	学院-1	院长-1
2	学生-2	学院-2	院长-2
3	学生-3	学院-2	院长-2

有以下函数依赖：

- Sno -> Sname, Sdept
- Sdept -> Mname

关系-2

Sno	Cname	Grade
1	课程-1	90
2	课程-2	80
2	课程-1	100
3	课程-2	95

第三范式：非主属性不能传递依赖于主属性

上面的 关系-1 中存在以下传递函数依赖：

- Sno -> Sdept -> Mname

可以进行以下分解：

关系-11

Sno	Sname	Sdept
1	学生-1	学院-1
2	学生-2	学院-2
3	学生-3	学院-2

关系-12

Sdept	Mname
学院-1	院长-1
学院-2	院长-2

7.索引的优缺点

优点：

第一，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。
第二，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。
第三，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。
第四，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。
第五，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：

第一，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。
第二，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。
第三，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。

8.删除数据库表中的所有数据

```
#立即执行，不能回滚
Truncate from tablename

#删除数据不删除表
Delete from tablename where 1=1

#速度,一般来说: truncate >delete 。
```

9.limit优化

```
#查询如下内容
select * from product limit 866613, 20

#通过查询id来优化速度，因为id是主键，利用索引会更快找到
select id from product limit 866613, 20

#上面那个只可以找到一列，如果要所有列可以如下
SELECT * FROM product
WHERE ID > =(select id from product limit 866613, 1) limit 20
```

10.多线程的创建

①. 继承Thread类创建线程类

- 定义Thread类的子类，并重写该类的run方法，该run方法的方法体就代表了线程要完成的任务。因此把run()方法称为执行体。
- 创建Thread子类的实例，即创建了线程对象。
- 调用线程对象的start()方法来启动该线程。

②. 通过Runnable接口创建线程类

- 定义runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。
- 创建 Runnable实现类的实例，并依此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。
- 调用线程对象的start()方法来启动该线程。

③. 通过Callable和Future创建线程

- 创建Callable接口的实现类，并实现call()方法，该call()方法将作为线程执行体，并且有返回值。
- 创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。
- 使用FutureTask对象作为Thread对象的target创建并启动新线程。
- 调用FutureTask对象的get()方法来获得子线程执行结束后的返回值。

```
//SumTotal实现了Callable接口
SumTotal sumTotal = new SumTotal();
//创建FutureTask
FutureTask<Integer> futureTask = new FutureTask<Integer>(sumTotal);
//执行线程任务
Thread thread = new Thread(futureTask);
thread.setName("线程1");
thread.start();

//获取线程任务返回值
try {
    int sum = futureTask.get();
    System.out.println(sum);
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
```

11.Spring的设计思想，它的特点

控制反转IOC和面向切面编程AOP

控制反转IOC指的就是代码的控制权不是单单写死在了后端接口中，比如在接口实现层决定new 哪一个接口，这样的话如果要更改new哪一个接口，每次都需要找到指定的实现处。控制反转而是反转了这个控制，由第三方（即Spring）来管理。

aop底层是通过反射机制和代理模式来实现的，通过反射来创建一个代理类，给我们的真实类添加一些额外的功能，比如设置某个切入点，在前置和后置增加一些预处理或者日志记录等操作，而增加这些功能不改变之前真实类的原有功能。

12.什么是IOC

控制反转指的就是代码的控制权不是单单写死在了后端接口中，比如在接口实现层决定new 哪一个接口，这样的话如果要更改new哪一个接口，每次都需要找到指定的实现处。控制反转而是反转了这个控制，由第三方（即Spring）来管理。

13.跨域

跨域，指的是浏览器不能执行其他网站的脚本。

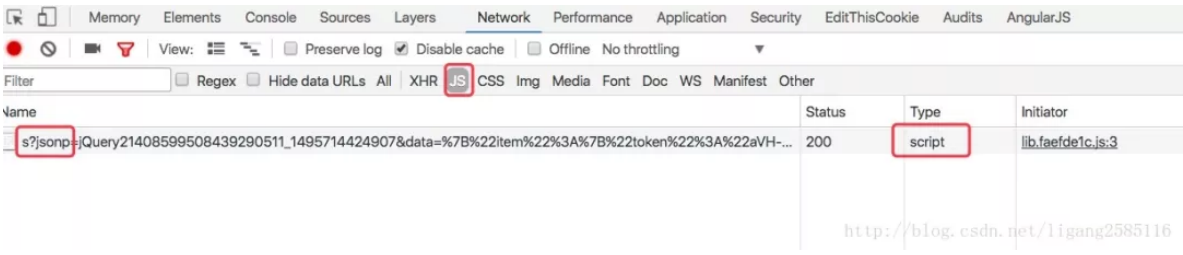
它是由浏览器的同源策略造成的，所谓同源是指，域名，协议，端口均相同，简而言之域名不同就是跨域。

方式一：图片ping或script标签跨域

图片ping常用于跟踪用户点击页面或动态广告曝光次数。
script标签可以得到从其他来源数据，这也是JSONP依赖的根据。

方式二：JSONP跨域

JSONP（JSON with Padding）是数据格式JSON的一种“使用模式”，可以让网页从别的网域要数据。根据 XmlHttpRequest 对象受到同源策略的影响，而利用 元素的这个开放策略，网页可以得到从其他来源动态产生的JSON数据，而这种使用模式就是所谓的 JSONP。用JSONP抓到的数据并不是JSON，而是任意的JavaScript，用 JavaScript解释器运行而不是用JSON解析器解析。所有，通过Chrome查看所有JSONP发送的Get请求都是js类型，而非XHR。



缺点：

- 只能使用Get请求
- 不能注册success、error等事件监听函数，不能很容易的确定JSONP请求是否失败
- JSONP是从其他域中加载代码执行，容易受到跨站请求伪造的攻击，其安全性无法确保

14.慢查询优化，limit优化

同9

15.操作系统临界资源的访问，控制

互斥量和信号量

16.网络编程有没有做过，socket编程，如何做一个服务器

socket+多线程来实现

服务器：创建一个socket对象提供给客户端连接，当监听到有客户端连接时，立即将这个socket加入到一个socket数组中，并且同时创建一个线程来监听客户端发来的信息，如果客户端发来信息就广播给其他客户端。

客户端：有两个线程，一个线程用来发送数据给服务器，一个线程用来读取从服务器广播过来的信息。

17.拥塞控制

如果网络出现拥塞，分组将会丢失，此时发送方会继续重传，从而导致网络拥塞程度更高。因此当出现拥塞时，应当控制发送方的速率。这一点和流量控制很像，但是出发点不同。流量控制是为了让接收方能来得及接收，而拥塞控制是为了降低整个网络的拥塞程度。

TCP 主要通过四个算法来进行拥塞控制：慢开始、拥塞避免、快重传、快恢复。

发送方需要维护一个叫做拥塞窗口（cwnd）的状态变量，注意拥塞窗口与发送方窗口的区别：拥塞窗口只是一个状态变量，实际决定发送方能发送多少数据的是发送方窗口。

为了便于讨论，做如下假设：

- 接收方有足够大的接收缓存，因此不会发生流量控制；
- 虽然 TCP 的窗口基于字节，但是这里设窗口的大小单位为报文段。

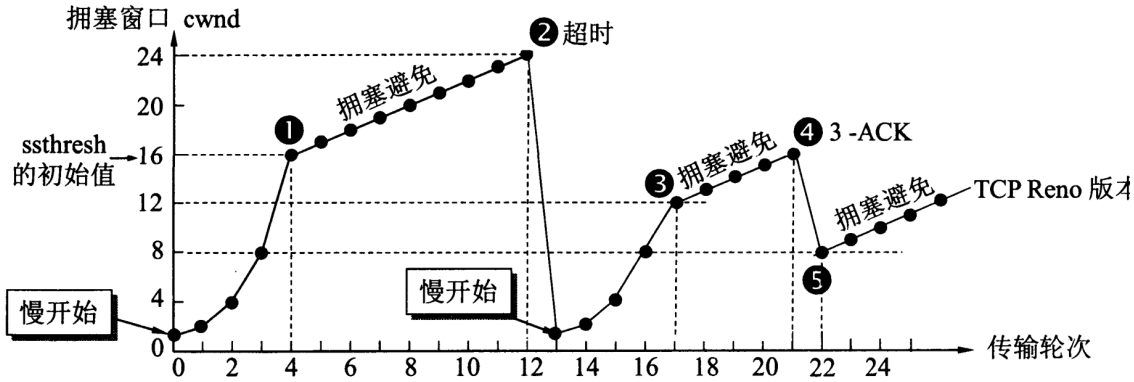


图 5-25　TCP 拥塞窗口 cwnd 在拥塞控制时的变化情况

1. 慢开始与拥塞避免

发送的最初执行慢开始，令 cwnd = 1，发送方只能发送 1 个报文段；当收到确认后，将 cwnd 加倍，因此之后发送方能够发送的报文段数量为：2、4、8 ...

注意到慢开始每个轮次都将 cwnd 加倍，这样会让 cwnd 增长速度非常快，从而使得发送方发送的速度增长速度过快，网络拥塞的可能性也就更高。设置一个慢开始门限 ssthresh，当 cwnd >= ssthresh 时，进入拥塞避免，每个轮次只将 cwnd 加 1。

如果出现了超时，则令 ssthresh = cwnd / 2，然后重新执行慢开始。

2. 快重传与快恢复

在接收方，要求每次接收到报文段都应该对最后一个已收到的有序报文段进行确认。例如已经接收到 M1 和 M2，此时收到 M4，应当发送对 M2 的确认。

在发送方，如果收到三个重复确认，那么可以知道下一个报文段丢失，此时执行快重传，立即重传下一个报文段。例如收到三个 M2，则 M3 丢失，立即重传 M3。

在这种情况下，只是丢失个别报文段，而不是网络拥塞。因此执行快恢复，令 ssthresh = cwnd / 2 ，cwnd = ssthresh，注意到此时直接进入拥塞避免。

慢开始和快恢复的快慢指的是 cwnd 的设定值，而不是 cwnd 的增长速率。慢开始 cwnd 设定为 1，而快恢复 cwnd 设定为 ssthresh。

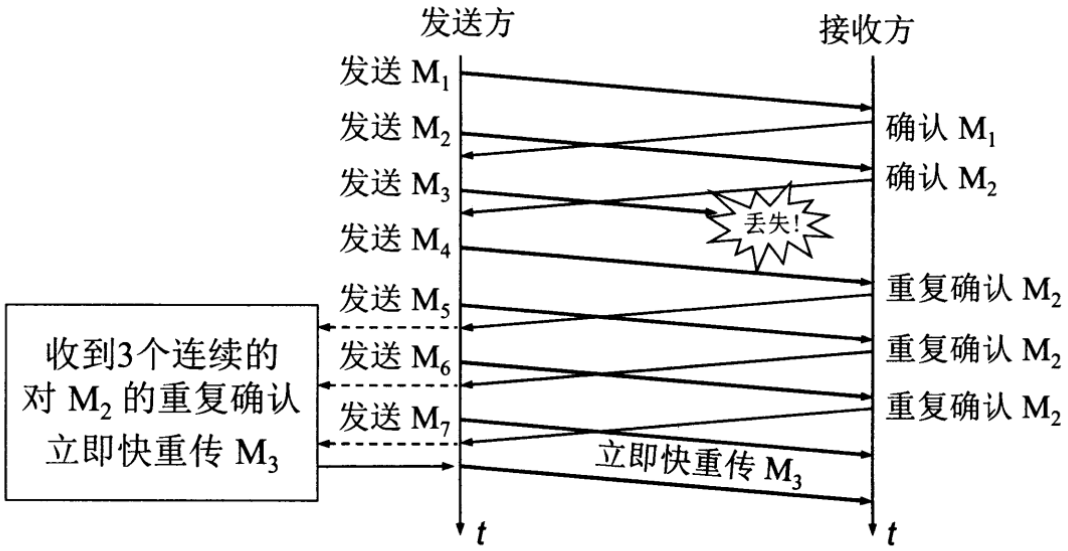


图 5-26　快重传的示意图

18.有一个大文件，40亿个无符号整数，有重复的，乱序，对文件排序，单机实现，内存一个G

如果用位图表示，一个数分配2位计算是否存在，40亿就是80亿bit，80亿为800M，所以40亿个数一共占100MB；

如果一个数分配2B计算出现的次数，2B*40亿=2B*400M=800MB

附加题：如何判断某个数是否为40亿个中的一个?

用位图，一位表示一个数，一个数占据4KB的内存，10亿个数就是就是100MB（100*2^20），40亿就是400MB。

附加题：在2.5亿个整数中找出不重复的整数

每个数分配2bit，00表示不存在，01表示出现一次，10表示多次，11无意义进行。然后扫描这2.5亿个整数，查看Bitmap中相对应位，如果是00变01，01变10，10保持不变。所描完后，查看bitmap，把对应位是01的整数输出即可。

附加题：100亿个url，每个URL占有16B，计算重复以及排序。

由于数据量很大，16*100亿B = 1600亿Byte,约等于160G。10亿Byte约等于1G大小。这种题目都是有数据资源限制的。如果说机器内存只有16G，那我们可以通过Hash算法，把原始大文件，hash分成10个小文件。或者hash成更多更小的文件，以满足机器资源限制。因为通过hash，我们可以保证相同的URL一定可以被分配到相同的小文件上。然后从每个文件里统计计算每个URL出现就可以了，可以使用hashMap存放统计结果。

附加题：100亿个url，每个URL占有16B，按照重复次数选择TOP100。

我们在存的时候，也是先做hash，看新进来的词汇会分配到那个节点（或者说小文件、分区），更新统计这个节点（分区）的hashmap统计结果，也就是给这个新加的词汇出现次数+1，然后这个文件，对应的hashmap统计有一个小根堆（100个node），存放着当前这个小文件top 100的最热统计，最后我们汇总多个小文件的小根堆，得到总的top 100.

19.赛马场8个跑道，64匹马，找出64匹中跑的最快的4匹，最少多少场

64分8组比**8场**，淘汰每组后四名；

8个第一比**1场**，淘汰后四名所在组；

剩余16匹马中有一个确定冠军，除此之外还剩第一名所在组后三位，第二名所在组前三位，第三名所在组前两位，第四名所在组第一位，共计9匹马未定，随机选8匹赛**1场**，取前三名；

前三名+上一场漏掉的马赛**1场**，再取前三名加上固定冠军就是最快的四匹马。

一共十一场

20.平时除了学习，有没有对一些开源的项目或者技术有研究和学习，了解到什么程度

21.数据库的事务隔离级别

未提交读（READ UNCOMMITTED）

事务中的修改，即使没有提交，对其它事务也是可见的。（更新数据时加入了共享锁即读锁，只能读不能写，解决丢失修改）

提交读（READ COMMITTED）

一个事务只能读取已经提交的事务所做的修改。换句话说，一个事务所做的修改在提交之前对其它事务是不可见的。（读数据时加入了共享锁但是读操作完了以后就释放共享锁，使得不能重复读，更新数据时加入了排他锁即写锁，解决脏读的问题）

可重复读（REPEATABLE READ）

保证在同一个事务中多次读取同一数据的结果是一样的。（读数据时加入了共享锁，一直到事务执行完毕才释放共享锁，这样就可以解决不可重复读问题，更新数据时加入了排他锁即写锁。）

可串行化（SERIALIZABLE）

强制事务串行执行，这样多个事务互不干扰，不会出现并发一致性问题。

该隔离级别需要加锁实现，因为要使用加锁机制保证同一时间只有一个事务执行，也就是保证事务串行执行。（读数据时对**整个数据表**加入了共享锁，更新数据时**整个数据表**加入了排他锁即写锁，解决幻影读问题）

22.线程不安全举个例子？线程安全的方式?

某些属性被多个线程共享导致数据结果无法保证，比如数据丢失修改，或者死锁的发生（A等待B释放资源，B等待A释放资源）。

线程安全：synchronized或者lock

23.ThreadLocal场景应用

线程变量，意思是ThreadLocal中填充的变量属于**当前**线程，该变量对其他线程而言是隔离的。ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。可以用来存储线程间的事务信息。

```
public class ThreadLocalTest01 {
    public static void main(String[] args) {
        //新建一个ThreadLocal
        ThreadLocal<String> local = new ThreadLocal<>();
        //新建一个随机数类
        Random random = new Random();
        //使用java8的Stream新建5个线程
        IntStream.range(0, 5).forEach(a-> new Thread(()-> {
            //为每一个线程设置相应的local值
            local.set(a+"  "+random.nextInt(10));
            System.out.println("线程和local值分别是  "+ local.get());
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start());
    }
}
/*线程和local值分别是  0  6
  线程和local值分别是  1  4
  线程和local值分别是  2  3
  线程和local值分别是  4  9
  线程和local值分别是  3  5 */
```

24.concurrentHashmap

将数组分为多个段，每个段对应一个小的hashtable来加锁。

- java 1.7中：第一次先找到段，第二次到达段中的hashtable来遍历。put操作会对段加锁，若其他线程已获取锁会不断自旋（一直查询）。size操作通过每个段维护一个count来统计个数，首先不加锁统计所有count，若连续统计2次一致则正确，否则一致计算直到连续3次都不对则加锁统计。
- java1.8中：已经抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全。数据结构采用：数组+链表+红黑树。

25.实现一个LRU

```
import java.util.HashMap;
import java.util.Map;

public class LRU {
    private Node head;//记录下一个要移除的node
    private Node end;//记录插入的尾巴节点
```

```
    private Map<String, Node> map=new HashMap<>();//加快查找速度，不用遍历链表，时间复杂度为O(1)
    private int limit;//最大容量

    public LRU(int limit) {
        this.limit = limit;
    }

    public void add(String key , String val) {
        Node node = map.get(key);
        if(node == null){
            if(map.size() >= limit){
                map.remove(head.key);
                removeNode(head);
            }
            Node newNode = new Node(key, val);
            map.put(key,newNode);
            addNode(newNode);
        }else {
            reflush(node);
        }
    }

    @Override
    public String toString() {
        return "LRU{" +
                "map=" + map +
                ", limit=" + limit +
                '}';
    }

    public void remove(String key) {
        Node node = map.remove(key);
        removeNode(node);
    }
    public String get(String key) {
        Node node = map.get(key);
        if(node == null){
            return null;
        }
        reflush(node);
        return node.val;
    }
    private void reflush(Node node) {
        if(node == end){
            return;
        }
        removeNode(node);
        addNode(node);
    }

    private void removeNode(Node node) {
        boolean isHeadOrEnd=false;
        if(node == end){
            end=end.pre;
            isHeadOrEnd=true;
        }
        if(node == head){
```

```
            head=head.next;
            isHeadOrEnd=true;
        }
        if(isHeadOrEnd){
            return;
        }
        if(node.next != null){
            node.next.pre=node.pre;
        }
        if(node.pre != null){
            node.pre.next=node.next;
        }
    }

    private void addNode(Node newNode) {
        if(end != null){
            end.next=newNode;
            newNode.pre=end;
        }
        end=newNode;
        if(head == null){
            head=end;
        }
    }
}
/*这里用双向队列，在移除时需要改变node的指向，如果不是双端队列，会比较麻烦*/
class Node{
    public String key;
    public String val;
    public Node pre;
    public Node next;

    public Node(String key, String val) {
        this.key = key;
        this.val = val;
    }
}
```

26.实现一个有过期时间的LRU

可以在刚刚的node结点中加入一个属性记录创建时间，然后在LRU的构造方法中设置一个while，每隔一段时间检测（当前时间-记录的创建时间）是否大于过期时间，是的话就执行removeNode。

27.HTTP请求/响应的步骤

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.oakcms.cn。

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。

例如：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;

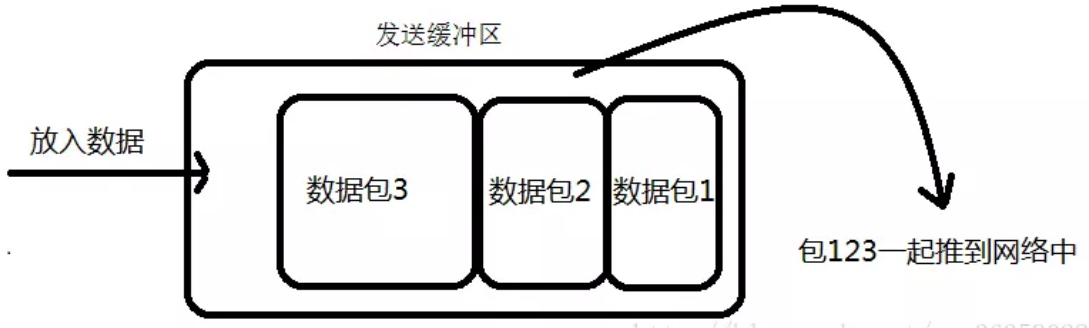
6、浏览器将该 html 文本并显示内容;

28.网络粘包、少包怎么处理

所谓粘包问题主要还是因为接收方不知道消息之间的界限，不知道一次性提取多少字节的数据所造成的。

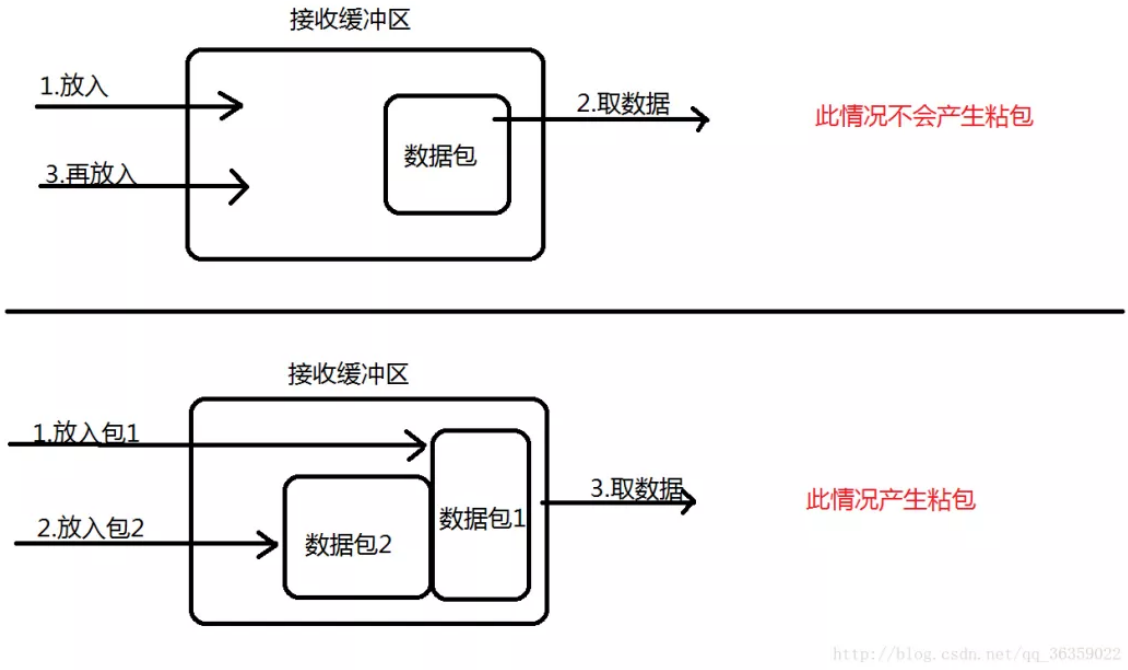
①. 发送方产生粘包

采用TCP协议传输数据的客户端与服务器经常是保持一个长连接的状态（一次连接发一次数据不存在粘包），双方在连接不断开的情况下，可以一直传输数据；但当发送的数据包过于的小时，那么TCP协议默认的会启用Nagle算法，将这些较小的数据包进行合并发送（缓冲区数据发送是一个堆压的过程）；这个合并过程就是在发送缓冲区中进行的，也就是说数据发送出来它已经是粘包的状态了。



②. 接收方产生粘包

接收方采用TCP协议接收数据时的过程是这样的：数据到底接收方，从网络模型的下方传递至传输层，传输层的TCP协议处理是将其放置接收缓冲区，然后由应用层来主动获取（C语言用recv、read等函数）；这时会出现一个问题，就是我们在程序中调用的读取数据函数不能及时的把缓冲区中的数据拿出来，而下一个数据又到来并有一部分放入的缓冲区末尾，等我们读取数据时就是一个粘包。（放数据的速度 > 应用层拿数据速度）



解决方法： 数据包添加首部、添加分界符号、设置为统一长度（不够的话填充0）

29.写个代码，关于map迭代器失效的

```
vector<int> container;

//会失效。
//对于序列式容器，比如vector，删除当前的iterator会使后面所有元素的iterator都失效，不可以再for中iter++了，已经找不到那个iter了。这是因为顺序容器内存是连续分配（分配一个数组作为内存），删除一个元素导致后面所有的元素会向前移动一个位置。（删除了一个元素，该元素后面的所有元素都要挪位置，所以，iter++，已经指向的是未知内存）。

for (iter = container.begin(); iter != container.end(); iter++)
    {
            if (*iter > 3)
              container.erase(iter);
    }


//不会失效
for (iter = container.begin(); iter != container.end(); )
    {
            if (*iter > 3)
              container.erase(iter);    //// vector元素自动向前挪动了(关联的map容器元素不会自动挪动)，  所以不能把iter进行++,而map迭代器失效可以直接在这里++解决。
            else{
```

```
            iter++;
        }
```

vector在插入元素时也会失效，因为如果vector申请的内存区域已经满了，就会扩容到另一块区域，那么迭代器自增就会失效。

30.select poll epoll，epoll的两种触发模式，各有什么应用场景

多路复用：单个线程，通过记录跟踪每个I/O流(socket)的状态，来同时管理多个I/O流

select模型：说的通俗一点就是各个客户端连接的文件描述符也就是套接字，都被放到了一个集合中，调用select函数之后会一直监视这些文件描述符中有哪些可读，如果有可读的描述符那么我们的工作进程就去读取资源。

poll模型：poll 和 select 的实现非常类似，本质上的区别就是存放 fd 集合的数据结构不一样。select 在一个进程内可以维持最多 1024 个连接，poll 在此基础上做了加强，可以维持任意数量的连接。但 select 和 poll 方式有一个很大的问题就是，我们不难看出来 select 是通过轮训的方式来查找是否可读或者可写，打个比方，如果同时有100万个连接都没有断开，而只有一个客户端发送了数据，所以这里它还是需要循环这么多次，造成资源浪费。

epoll模型：epoll 是 select 和 poll 的增强版，epoll 同 poll 一样，文件描述符数量无限制。epoll是基于内核的反射机制，在有活跃的 socket 时，系统会调用我们提前设置的回调函数。而 poll 和 select 都是遍历。

epoll 的描述符事件有两种触发模式：LT（level trigger）和 ET（edge trigger）。

1. LT 模式

当 epoll_wait() 检测到描述符事件到达时，将此事件通知进程，进程可**以不立即处理该事件**，下次调用 epoll_wait() 会再次通知进程。是默认的一种模式，并且同时支持 Blocking 和 No-Blocking。

eg：客户端发送数据，I/O函数会提醒描述符fd有数据---->recv读数据，若一次没有读完，I/O函数会一直提醒服务端fd上有数据，直到recv缓冲区里的数据读完

2. ET 模式

和 LT 模式不同的是，通知之后进程**必须立即处理**事件，下次再调用 epoll_wait() 时不会再得到事件到达的通知。

eg：客户端发送数据，I/O函数只会提醒一次服务端fd上有数据，以后将不会再提醒

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。只支持 No-Blocking，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。

31.多线程多进程区别

多线程模型适用于I/O密集型场景，因为I/O密集型场景因为I/O阻塞导致频繁切换，线程只占用栈，程序计数器，一组寄存器等少量资源，切换效率高，单机多核分布式；

多进程模型适用于需要频繁的计算场景，多机分布式，占有内存多，切换比较复杂，数据是分开的且共享复杂。

32.为什么c++要用指针，用指针什么好处

主要是灵活，方便。

- 利用指针可以实现动态内存分配，指针指向一个分配的内存地址。
- 指针还用于表示和实现各种复杂的数据结构
- 可以提高程序的编译效率和执行速度，使程序更加简洁，比如不用直接传递一个大块的数据区域，而是传递一个指针即可。

33.数据库主从复制，binlog有哪些格式

MySQL 主从复制概念

MySQL 主从复制是指数据可以从一个MySQL数据库服务器主节点复制到一个或多个从节点。MySQL默认采用异步复制方式，这样从节点不用一直访问主服务器来更新自己的数据，数据的更新可以在远程连接上进行，从节点可以复制主数据库中的所有数据库或者特定的数据库，或者特定的表。

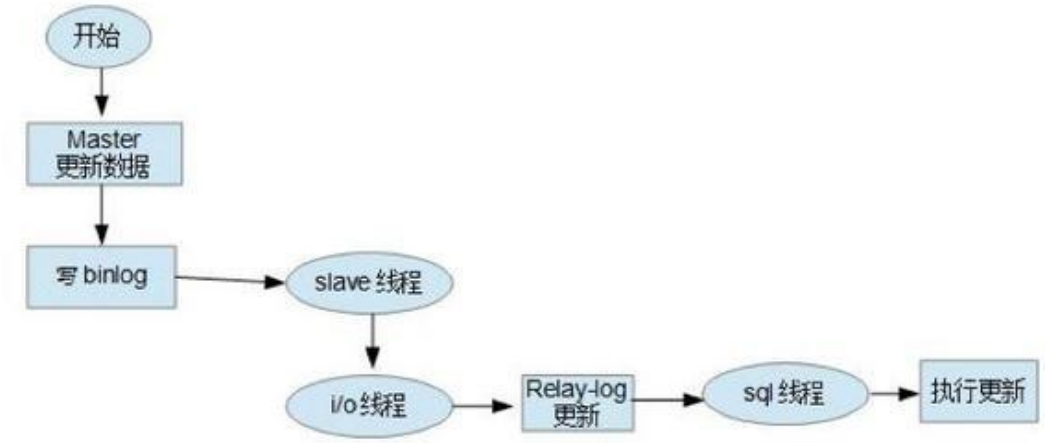
MySQL 主从复制主要用途

l 读写分离

在开发工作中，有时候会遇见某个sql 语句需要锁表，导致暂时不能使用读的服务，这样就会影响现有业务，使用主从复制，让主库负责写，从库负责读，这样，即使主库出现了锁表的情景，通过读从库也可以保证业务的正常运作。

l 数据实时备份，当系统中某个节点发生故障时，可以方便的故障切换

复制过程如下：

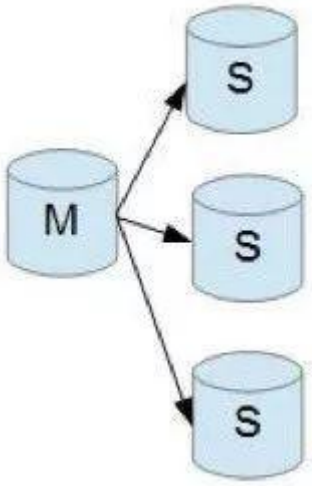


- 从节点上的**I/O 线程**连接主节点，并请求日志文件的指定位置后的日志内容；
- 主节点接收到来自从节点的I/O请求后，主节点的**binary log dump 线程**读取指定日志指定位置之后的日志信息，返回给从节点。返回信息中包括bin log和bin log position；从节点的I/O进程接收到内容后，将接收到的日志内容更新到本机的relay log中，并将读取到的binary log文件名和位置保存到master-info 文件中；
- 从节点的 **SQL线程**检测到relay-log 中新增加了内容后，会将relay-log的内容解析成在从节点上实际执行过的操作，并在本数据库中执行。

MySQL 主从形式

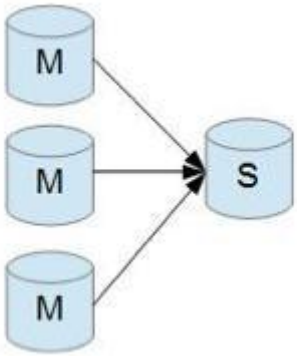
一主一从

一主多从，提高系统的读性能



一主一从和一主多从是最常见的主从架构，实施起来简单并且有效，不仅可以实现HA，而且还能读写分离，进而提升集群的并发能力。

多主一从 （从5.7开始支持）



多主一从可以将多个mysql数据库备份到一台存储性能比较好的服务器上。

双主复制

双主复制，也就是互做主从复制，每个master既是master，又是另外一台服务器的slave。这样任何一方所做的变更，都会通过复制应用到另外一方的数据库中。

MySQL主从复制有三种方式：基于SQL语句的复制（记录的是会修改数据的SQL语句），基于行的复制（记录的是更新的行，所以日志量会很大），混合模式复制。

34.说说多进程的系统调用（fork），那你说说多线程的系统调用。

通过fork函数分叉一个新的进程。

35.事务acid，sql语句里怎么写一个事务

1. 原子性（Atomicity）

事务被视为不可分割的最小单元，事务的所有操作要么全部提交成功，要么全部失败回滚。

回滚可以用回滚日志（Undo Log）来实现，回滚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即可。

2. 一致性（Consistency）

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，所有事务对同一个数据的读取结果都是相同的。

3. 隔离性（Isolation）

一个事务所做的修改在最终提交以前，对其它事务是不可见的。

4. 持久性（Durability）

一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统发生崩溃，事务执行的结果也不能丢失。

```
BEGIN
insert into a values（'1','str'）
insert into b values（'2','str'）
update a set name='stu' where id='1'
ROLLBACK
COMMIT
```

36.回滚操作是系统自动的，还是要你手动回滚。

都可以

37.说说数据库容灾

即在异地部署一个一模一样的数据库，一个数据库所处的地理位置发生自然灾害了导致当前数据库发生灾难，另一个数据库会立马顶替工作。例如两城三中心（两个城市，一个生产中心，一个同城灾备中心，一个异地灾备中心），或者三地五中心。

38.数据库引擎，innodb的行锁和表锁，不通过索引用的什么锁

innodb和myisam引擎。

MyISAM：有三个文件，一个存放索引，一个存放数据，一个存放表结构。

查找数据时会先根据索引文件判断索引对应的行记录地址，然后再查找行记录文件找到所对应的行记录。叶子结点存放的数据是**一个行记录的地址指针。**

InnoDB：有两个文件，一个存放表结构，一个是数据和索引的合并文件。

表数据文件本身就是用B+树来组织的索引结构文件，所以必须要有主键，而且设置为自增的整型。叶子结点存放的数据是**一个完整的行记录数据**。

innodb行级锁是依赖于索引实现的，where条件后的字段只有是索引才可以应用于行级锁，否则是表级锁。

不通过索引就是表级锁。

39.ipc，为什么共享内存最快

共享内存的消息复制只有两次。一是，从输入文件到共享内存；二是，从共享内存到输出文件。

而管道需要四次，一是复制文件到输出缓冲区，二是把输出缓冲区文件复制到管道，三是从管道复制到输入缓冲区，四是从输入缓冲区复制。

40.知道什么加密算法，说说https过程，怎么验证签名。有哪些对称加密、非对称加密算法，哈希算法有哪些。

（对称加密）AES、DES

（非对称加密）RSA、DSA

https过程：数据加密、验证签名、保证报文完整性。

常用的哈希算法有 SHA-1算法和MD5算法

数字签名：

一个报文通过hash得到报文摘要，然后被私钥加密成为数字签名，接收方得到数字签名后用公钥进行解密得到报文摘要，将接受的报文hash后的摘要和解密得到的摘要比对来验证数字签名。

数字证书：

为了保证接收方的公钥确实是发送方的公钥，通过第三方机构证书中心来认证。证书中心用**自己的私钥**，对发送方的公钥和一些相关信息一起加密，生成"数字证书"（Digital Certificate），随数字签名一起发送。接收方通过用发送发的公钥以及证书中心的公钥来解密。

41.微信支付输入密码，怎么保证安全。密码怎么安全存在db

使用不可逆的加密算法加密（MD5和SHA-1）存在于DB中，以后直接加密完判断即可。

42.说说reactor模式

43.怎么保证网络数据传输没有被更改。

首先生成MD5码，然后对数据部分进行加密，这样无法破解数据的话，即使修改了加密数据，也不能生成正确的MD5码。接收方收到数据后，先解密，再生成MD5码后再进行对比进行校验

44.消息队列串包怎么解决

比如传进来三个数据1、2、3，三个消费者分别读取三个数据来操作，但是消费者2可能执行快，使得消费者2就会去消费消费者1应该消费的数据。可以通过设置三个任务队列，每个消费者一个任务队列，这样就不会乱套。

45.https中间被劫持会怎样

https由于安装了ssl证书，所以能够在客户端和服务器端建立一条加密的连接，防止被窃取和劫持。它能确定用户访问的是真实的主体，而不是假冒的第三方，区别于钓鱼网专站。

如果被劫持了的话，自己发送的所有数据都会到假冒第三方的私人服务器上。

46.怎么避免删库跑路

通过数据库容灾，两地三中心。

数据库每天都备份，备份密钥交给不同的人管理。

设定权限，只有大boss才可以删库。

47.学生注册系统，怎么保证分配的学号是唯一的

利用uuid或者snowflake算法得到一个学号，就可以保证全局唯一。

48.new的对象可以free吗

不可以，如果对一个类直接free，那么不会调用析构函数，引起内存的泄漏

49.C++中的内存区域

1、**栈区（stack）**— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

2、**堆区（heap）** — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。

3、**全局区（静态区）（static）**—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域（RW）， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域（ZI）。 - 程序结束后有系统释放

4、**文字常量区** —常量字符串就是放在这里的。 程序结束后由系统释放

5、**程序代码区**—存放函数体的二进制代码。

50.手撕多种排序算法

51.c++如何实现多态

子类型多态（虚函数）、强制多态（自定义数据类型/基本数据类型强转）、参数多态（类模板、函数模板）、重载多态（函数重载、操作符重载）

52.设计一个哈希算法需要考虑什么

- 映射到每个位置的概率近似
- 冲突小
- hash的过程相互独立互不干扰

53.哈希冲突解决

链表法、再哈希法、线性探测法、再平方法

54.快排什么时候会退化?

如果一个待排的数组是完全有序或者完全倒序的情况下

55.为什么对小数据量直接插入排序要比快排更好呢?

元素少的时候（小于50），插入排序的比较次数会比快排的比较次数少，并且也没有额外的开销（如果是递归还需要栈的开销，非递归也需要额外的栈来记录）。

56.内存分配算法

首次适应算法、最坏适应算法(找最大的满足大小的空闲分区，把最大内存分开，容易导致之后难以申请大内存，且申请大块不用的内存)、最好适应算法（找最小的满足大小的空闲分区，但是容易产生细小的不能使用的碎片）

57.进程调度的策略有哪些？我们计算机用的策略是什么

优先级调度、先来先服务、最短时间优先、时间片轮转；时间片轮转

58.进程的地址空间

按顺序依次是 代码段、数据段、堆和栈

59.多线程和多进程的优劣?

多进程：互不影响，一个进程有问题不干扰另一个进程，不存在同步和竞争的问题，构建会有大量开销

多线程：共享同一进程的全局变量，通信十分方便

60.对称加密和非对称加密的优缺点

对称：

- 优点：运算速度快；
- 缺点：无法安全地将密钥传输给通信方。

非对称：

- 优点：可以更安全地将公开密钥传输给通信发送方；
- 缺点：运算速度慢。

61.sql注入

可以通过正则表达式过滤传入的参数，或者PreparedStatement（不会直接提交SQL语句）

62.XSS攻击

SS攻击又称CSS,全称Cross Site Script （跨站脚本攻击），其原理是攻击者向有XSS漏洞的网站中输入恶意的 HTML 代码，当用户浏览该网站时，这段 HTML 代码会自动执行，从而达到攻击的目的。

XSS防范的总体思路是：对输入(和URL参数)进行过滤，对输出进行编码。

63.虚函数什么时候用，为什么要用

虚函数简单的说就是**为了让基类指针能够指向派生类中与基类同名的函数而引进的**

举个简单的例子，

1：你定义了一个“图形类”这样的基类，然后再类中定义了一个求图形周长的函数（不是虚函数）；

2：现在再定义这个“图形类”的一个派生类“三角形类”，中也含有一个求三角形周长的函数（不是虚函数）；

3：再定义一个这个“图形类”的一个派生类“矩形类”，中也含有一个求矩形周长的函数（不是虚函数）；

4：现在回到主函数，你定义了这个“图形类”的一个指针（即基类的指针），根据C++的规定，基类的对象指针可以指向它的公有派生类的对象，但是当其指向公有的派生类对象时，它只能访问派生类中从基类继承来的成员，而不能访问公有派生类中定义的成员。

所以，你定义的这个指针是不能够指向“三角形”和“矩形”类中定义的那个周长函数，但是，如果你在基类中将这个周长函数定义为虚函数，则这样的代码是允许的，而且能够到达预期目的

64.什么是覆盖索引?

查询的字段被联合索引包含，而且只需要在一棵索引树上就能获取SQL所需的所有列数据，无需回表，速度更快。

例如：假设id和name是一个联合索引

如果select id,name from xx **符合覆盖索引**

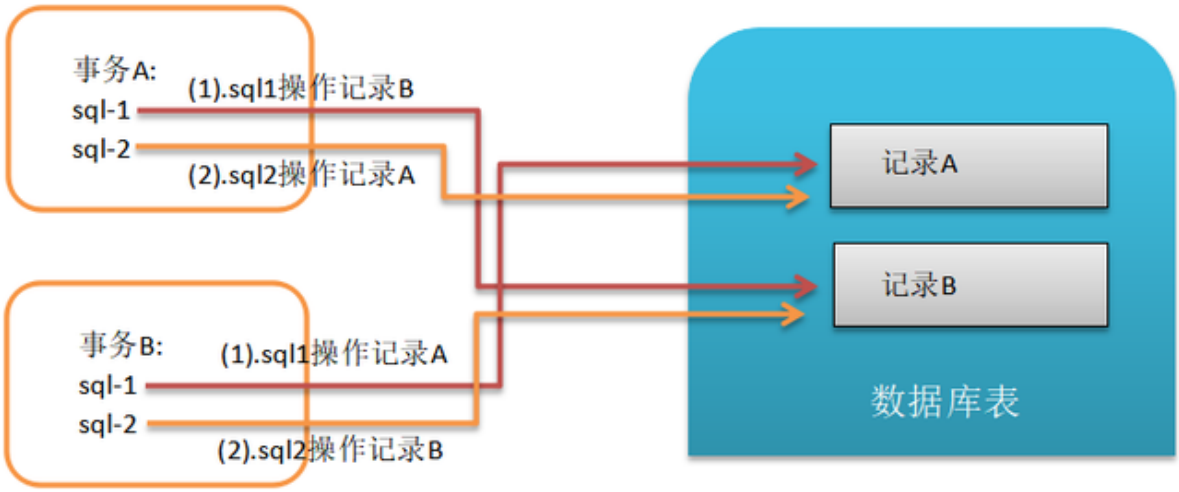
如果select sex,name from xx **不符合覆盖索引，因为sex不在索引中，需要回表查询**

65.什么是回表查询?

先定位主键值，再定位行数据。INNODB引擎就是回表查询，主索引和辅助索引结合。

66.数据库中的死锁

当多个事务试图以不同的顺序锁定资源时，就可能会产生死锁。



图中，有两个事务A与B，事务A与B同时执行sql1，则事务A会锁住记录B；事务B会锁住记录A，两条记录同时被锁住。接下来事务A执行sql2，发现记录A被事务B锁住，事务B执行sql2发现记录B被事务A锁住，陷入死循环，产生死锁。

解决：尽量避免同时锁定两个资源，如操作A和B两张表时，总是按先A后B的顺序处理

67.查询比较多的时候为什么用myisam?

- innodb寻址要映射到块，再到行，MYISAM记录的直接是文件的OFFSET，定位比INNODB要快。
- INNODB还需要维护MVCC一致。
- myisam缓存只需要缓存索引块，不用整行数据。

68.快排的优化方式

- 哨兵的位置通过三数取中法来确定，而不是只设置为头或者尾，因为最佳的划分是将待排序的序列分成等长的子序列。

  举例：待排序序列为：8 1 4 9 6 3 5 2 7 0

  左边为：8，右边为0，中间为6.

  我们这里取三个数排序后，中间那个数作为枢轴，则枢轴为6

- 当待排序序列的长度分割到一定大小后，使用插入排序
- 在一次分割结束后，可以把与Key相等的元素聚在一起，继续下次分割时，不用再对与key相等元素分割

  具体过程：在处理过程中，会有两个步骤

  第一步，在划分过程中，把与key相等元素放入数组的两端

  第二步，划分结束后，把与key相等的元素移到枢轴周围

69.父进程和子进程会共享什么，不共享什么

子进程会继承父进程什么资源

1.实际UID和GID,以及有效的GID和UID
2.所有环境变量
3.进程组ID和会话ID
4.当前工作路径。除非用chdir()修改
5.打开的文件
6.信号响应函数
7.整个内存空间，包括栈、堆、数据段、代码段、标准I/O缓冲区等

子进程不会继承父进程什么资源

1.进程的进程号PID。PID号是身份证号码，每个进程的PID号都不一样

2.记录锁。父进程对某个文件加了锁，子进程不会继承这个锁

3.挂起的信号

70.字节对齐

为了提高效率，计算机从内存中取数据是按照一个固定长度的。以32位机为例，它每次取32个位，也就是4个字节（每字节8个位)。字节对齐有什么好处？以int型数据为例，如果它在内存中存放的位置按4字节对齐，也就是说1个int的数据全部落在计算机一次取数的区间内，那么只需要取一次就可以了.

71.一条SQL语句是如何执行的?

1. 连接器（获取、维持和管理数据库连接）
2. 查询缓存（之前执行过的查询，MySQL以"Key - Value"的形式存在内存（key为SQL，value为结果集））
3. 分析器（分析SQL语句）
4. 优化器（优化SQL语句的执行过程，例如使用哪个索引）
5. 执行器（执行SQL）
6. 存储引擎（提供读写接口，供执行器调用并获取结果集）