上品科技研发综合题库

2018

# 数据结构与算法

## 数组

## 链表

## 队列

## 栈

## 图

## 面试题分类（上机题）

### 模拟

#### 1.6.1.1 位运算

##### 统计一个整数二进制表示中bit为1的个数

解法一:（能想到这种解法的说明了解位运算,但是没有考虑负数的源码标志位为1的情况,酌情给分数）

1. /\*\*
2. \*
3. \* 分析:
4. \* 一个数的二进制中有多少个 1
5. \* [1] 将该数字和 1 做与运算,判断该数的末位数字是否为 1(&运算特性)
6. \* [2] 如果与运算结果为 1 , count+=1,于此同时将 a 向右移动一位;
7. \* [3] 直到 a变成 0 为止;
8. \* @param args
9. \*/
10. **public** **static** **void** main(String[] args) {
11. **int** a = 11;
12. // 1011
13. **int** count = 0;
14. **while** (a != 0) {
15. **if** ((a & 1) == 1) {
16. count += 1;
17. }
18. a >>= 1;
19. }
20. System.out.println(count);
21. }

解法一存在的问题: 可能会陷入死循环(输入负数会陷入到死循环之中,因为负数右移时,在最高位补一)

解法二:（让1最多移动32位，因为你判断的是一个int 类型4字节整数）

1. **public** **static** **void** main(String[] args) {
2. **int** n = 11;
3. // 1011
4. //    1
5. **int** count = 0;
6. **int** flag = 1;
7. **for** (**int** i = 0; i < 32; i++) {
8. **if** ((n & flag) != 0) {
9. count += 1;
10. }
11. flag = flag << 1;
12. }
13. System.out.println(count);
14. }

解法三 :（让n右移和1做&运算,在int范围内移动,即只移动32位）

1. **public** **static** **void** main(String[] args) {
2. **int** n = 11;
3. **int** count = 0;
4. **for** (**int** i = 0; i < 32; i++) {
5. **if** (((n >> i) & 1) != 0) {
6. count += 1;
7. }
8. }
9. System.out.println(count);
10. }

解法四:

1. **public** **static** **void** main(String[] args) {
2. **int** n = 11;
3. /\*\* 不说明原理,自己理一遍就懂什么意思了。
4. \* n   11  1011
5. \* n-1 10  1010
6. \* &----------- count=1
7. \*         1010
8. \* n   10  1010
9. \* n-1  9  1001
10. \* &-----------
11. \*         1000 count=2
12. \* n    9  1000
13. \* n-1  8  0111
14. \* &----------- count=3
15. \*         0000
16. \*/
17. **int** count = 0;
18. **while** (n != 0) {
19. count += 1;
20. n = n & (n - 1);
21. }
22. System.out.println(count);
23. }

### 链表

#### 链表的基础操作

##### 寻找单链表倒数第N个节点

1. **public** **class** Solution {
2. **class** ListNode {
3. **int** val;
4. ListNode next = null;
6. ListNode(**int** val) {
7. **this**.val = val;
8. }
9. **public** ListNode FindKthToTail(ListNode head, **int** k) {
10. /\*\*
11. \* step 1:创建2个节点,都指向头结点;
12. \*/
13. ListNode first = head;
14. ListNode second = head;
15. /\*\*
16. \* step 2:让第二个节点指向第K个节点;
17. \*/
18. **for** (**int** i = 0; i < k; i++) {
19. /\*\*
20. \* 如果出现链表长度<k的情况,方法直接返回;
21. \*/
22. **if** (second == null) {
23. **return** null;
24. }
25. second = second.next;
26. }
27. /\*\*
28. \* step 3:
29. \*/
30. **while** (second!= null) {
31. first = first.next;
32. second = second.next;
33. }
34. **return** first;
35. }
36. }

##### 反转单链表

1. **public** **class** Solution {
2. **class** ListNode {
3. **int** val;
4. ListNode next = null;
6. ListNode(**int** val) {
7. **this**.val = val;
8. }
9. /\*\* 自己画一个图显示一下即可懂原理;
10. \* 1-->2-->3-->4-->5
11. \*
12. \* @param head
13. \* @return
14. \*/
15. **public** ListNode reverseList(ListNode head) {
16. ListNode pre = null;
17. ListNode current = head;
18. ListNode next = null;
19. **while** (current != null) {
20. next = current.next;
21. current.next = pre;
22. pre = current;
23. current = next;
24. }
25. **return** pre;
26. }
27. }

##### 判断一个链表中是否存在环

1. **public** **class** Solution {
2. **class** ListNode {
3. **int** val;
4. ListNode next = null;
6. ListNode(**int** val) {
7. **this**.val = val;
8. **this**.next = null;
9. }
10. }
11. 1---->2---- >3----> 4---->5
12. ^ |
13. | v
14. 7<-----6
15. **public** boolean detectLoop(ListNode head) {
16. ListNode fast = head, slow = head;
17. **while** (fast != null && fast.next != null) {
18. fast = fast.next.next;
19. slow = slow.next;
20. **if** (fast == slow) {
21. **return** **true**;
22. }
23. }
24. **return** **false**;
25. }
26. }

### 队列

#### 1.6.3.1 使用堆栈（Stack）来模拟队列（FIFO）功能，

要求数据必须存储在堆栈内部，需要实现enqueue（入队），dequeue（ 出队），isEmpty（判空）三个功能;

### Hash算法

（1）4亿个int数，如何找出重复的数

用hash方法，创建一个2的32方个长度的bit的数组：b[2的32次方]，每取一个int数x，可i=hash(x) 得到数组下标，b[hash(x)] 设置为1 表示存在。

（2）4亿个url，找出重复的

考虑内存不够，通过hash算法，将url分配到1000个文件中，不同的文件肯定不重复，再分别找出重复的。

（3）统计100G的ip文件中出现ip次数最多的100个ip

### 栈

#### 1.6.5.1 分别用数组和链表实现Stack

[代码及分析](https://github.com/553899811/Algorithm-And-DataStructure/tree/master/Data%20Structures/Stack)

#### 1.6.5.2 给出入栈顺序，哪一个不是出栈顺序（重点）

设有一个栈，元素依次进栈的顺序是A,B,C,D,E。下列不可能的出栈顺序有?

A:ABCDE

B:BCDEA

C:EABCD

D:EDCBA

解析: 正确理解题意是关键,在一边push的时候可以一边pop();

A:ABCDE (可能的出栈顺序,push进去一个随即pop出来就是这个顺序)

B:BCDEA (可能的出栈顺序,先push进去A之后,BCDE按照push进去随即pop出来,最后pop出A,就是这个顺序)

C:EABCD (不可能的出栈顺序)

D:EDCBA (将所有元素先全部push进去,然后一个一个pop出来)

#### 1.6.5.3 用两个栈来模拟一个队列(FIFO,先进先出原则),完成队列的push和pop操作

1. import java.util.NoSuchElementException;
2. **public** **class** Solution<T> {
3. Stack<T> t1 = **new** Stack<T>();
4. Stack<T> t2 = **new** Stack<>();
6. **public** **void** push(T data) {
7. t1.push(data);
8. }
10. **public** T pop() {
11. **if** (t1.isEmpty() && t2.isEmpty()) {
12. **throw** **new** NoSuchElementException("队列中无元素");
13. }
14. **if** (t2.isEmpty()) {
15. **while** (!t1.isEmpty()) {
16. t2.push(t1.pop());
17. }
18. }
19. **return** t2.pop();
20. }
21. }

### 搜索与查找

#### 1.6.6.1 二分查找

（1）二分查找法

要求原数组是有序序列;

1. **public** **static** **int** BinarySearch(**int**[] arr, **int** key) {
2. **int** left = 0, right = arr.length - 1;
3. **int** middle = 0;
4. **while** (left <= right) {
5. middle = (left + right) >> 1;
6. **if** (arr[middle] == key) {
7. **return** middle;
8. } **else** **if** (arr[middle] > key) {
9. right = middle - 1;
10. } **else** {
11. left = middle + 1;
12. }
13. }
14. **return** -1;

}

### 排序

#### 冒泡排序

省略

#### 快速排序

<https://blog.csdn.net/code_AC/article/details/74158681>

（理解快速排序的思路为首要前提,其次是对一个实例的完整化口述过程）

1. **public** **class** Solution {
3. /\*\*
4. \* @param arr
5. \* @param left
6. \* @param right
7. \*/
8. **public** **static** **void** QuickSort(**int**[] arr, **int** left, **int** right) {
9. **if** (left >= right) {
10. **return**;
11. }
12. **int** i = left, j = right, key = arr[left];
13. **while** (i < j) {
14. **while** (i < j && key <= arr[j]) {
15. j--;
16. }
17. arr[i] = arr[j];
18. **while** (i < j && key > arr[i]) {
19. i++;
20. }
21. arr[j] = arr[i];
22. }
23. arr[i] = key;
24. QuickSort(arr, left, i - 1);
25. QuickSort(arr, i + 1, right);
26. }
28. **public** **static** **void** main(String[] args) {
29. **int**[] arr = **new** **int**[]{
30. 2, -9, 1, 90, -89, 89, 23
31. };
32. QuickSort(arr, 0, arr.length - 1);
33. **for** (**int** i = 0; i < arr.length; i++) {
34. System.out.print(arr[i] + " ");
35. }
36. }
37. }

时间复杂度: O(N\*Log(N))

#### 堆排序

找出一个数组中第100个小的数字（堆思想解决）

### 数论

（1）1000个苹果放10个篮子（每个篮子中苹果没有个数限制），怎么放，所有可能的个数是？

（我的理解是m个苹果放n个篮子[篮子其实也会分相同与不同之说]）

（2）如何判断一个正整数是2的整数次幂

解法1：

1. /\*\*
2. \* 思路:
3. \* 是2的整数次幂,表示可以一直被2整除直到1;
4. \* @param n
5. \* @return
6. \*/
7. **public** **static** boolean isPower(**int** n) {
8. **if** (n == 1) {
9. **return** **true**;
10. } **else** {
11. **while** (n != 1) {
12. **if** (n % 2 == 0) {
13. n >>= 1;
14. } **else** {
15. **return** **false**;
16. }
17. }
18. **return** **true**;
19. }
20. }

解法2:

1. /\*\*
2. \* 举例说明:
3. \* 16 --> 二进制表示: 10000
4. \* 15 --> 二进制表示: 01111
5. \* -----------------------&
6. \*                   00000
7. \* @param n
8. \* @return
9. \*/
10. **public** **static** boolean isPower(**int** n) {
11. **if** ((n & (n - 1)) == 0) {
12. **return** **true**;
13. }
14. **return** **false**;
15. }

（3）求两个数的最大公约数和最小公倍数

最大公约数:

1. /\*\*
2. \* 辗转相除法:
3. \* a=25,b=15
4. \* 25/15=1......10
5. \* 15/10=1......5
6. \* 10/5 =2......0
7. \* @param a
8. \* @param b
9. \* @return
10. \*/
11. **public** **static** **int** gcd(**int** a, **int** b) {
12. **return** b == 0 ? a : gcd(b, a % b);
13. }

最小公倍数：（两个数的最小公倍数等于两数之积除以两个数的最大公约数）

Lcm(a,b)=a\*b/gcd(a,b);

### 树

#### 二叉树定义

1. /\*\*
2. \* 二叉树节点定义:
3. \*/
4. **class** Node {
5. **int** val;
6. Node left, right;
8. Node(**int** val) {
9. **this**.val = val;
10. **this**.left = null;
11. **this**.right = null;
12. }
13. }

#### 1.6.9.2 遍历二叉树

1. 先序遍历二叉树
2. /\*\*
3. \* 先序遍历
4. \* 根左右
5. \*
6. \* @param root
7. \*/
8. **public** **static** **void** preOrderTree(Node root) {
9. **if** (root != null) {
10. System.out.println(root.val);
11. preOrderTree(root.left);
12. preOrderTree(root.right);
13. }
14. }
15. 中序遍历二叉树
16. /\*\*
17. \* 中序遍历
18. \* 左根右
19. \*
20. \* @param root
21. \*/
22. **public** **static** **void** inOrderTree(Node root) {
23. **if** (root != null) {
24. inOrderTree(root.left);
25. System.out.println(root.val);
26. inOrderTree(root.right);
27. }
28. }
29. 后序遍历二叉树
30. /\*\*
31. \* 后序遍历
32. \* 左右根
33. \*
34. \* @param root
35. \*/
36. **public** **static** **void** postOrderTree(Node root) {
37. **if** (root != null) {
38. postOrderTree(root.left);
39. postOrderTree(root.right);
40. System.out.println(root.val);
41. }
42. }

### 递归

#### 跳台阶问题

1. **public** **static** **int** step(**int** n) {
2. **if** (n == 1 || n == 2) {
3. **return** n;
4. } **else** {
5. **return** step(n - 1) + step(n - 2);
6. }
7. }

考虑当n—>很大时,此时这种解法递归深度很深,时间复杂度大，引起超时;

1. **public** **static** **int** step(**int** n) {
2. **int** left = 1, right = 2;
3. **int** tmp = 0;
4. **for** (**int** i = 2; i < n; i++) {
5. tmp = right;
6. right = left + right;
7. left = tmp;
8. }
9. **return** right;
10. }

# JAVA基础

# 基础-面向对象特性

## Object方法之equals()

* 源码：
* **public** boolean equals(Object obj) {
* **return** (**this** == obj);
* }

## Object方法之hashCode()

* 源码：

1. /\*\*
2. \* Returns a hash code value for the object. This method is
3. \* supported for the benefit of hash tables such as those provided by
4. \* {@link java.util.HashMap}.
5. \* <p>
6. \* The general contract of {@code hashCode} is:
7. \* <ul>
8. \* <li>Whenever it is invoked on the same object more than once during
9. \*     an execution of a Java application, the {@code hashCode} method
10. \*     must consistently return the same integer, provided no information
11. \*     used in {@code equals} comparisons on the object is modified.
12. \*     This integer need not remain consistent from one execution of an
13. \*     application to another execution of the same application.
14. \* <li>If two objects are equal according to the {@code equals(Object)}
15. \*     method, then calling the {@code hashCode} method on each of
16. \*     the two objects must produce the same integer result.
17. \* 如果两个对象equals(Object)是相等的，那么调用hashCode方法结果必须产生相同的整型
18. \* <li>It is <em>not</em> required that if two objects are unequal
19. \*     according to the {@link java.lang.Object#equals(java.lang.Object)}
20. \*     method, then calling the {@code hashCode} method on each of the
21. \*     two objects must produce distinct integer results.  However, the
22. \*     programmer should be aware that producing distinct integer results
23. \*     for unequal objects may improve the performance of hash tables.
24. \* 不要求两个根据equals（Object）方法比较的不相等的对象，调用hashCode方法必须产生不 同的整型，然而，程序员应该意识到为不同的对象产生不同的整型可以提高哈希表的性能
25. \* </ul>
26. \* <p>
27. \* As much as is reasonably practical, the hashCode method defined by
28. \* class {@code Object} does return distinct integers for distinct
29. \* objects. (This is typically implemented by converting the internal
30. \* address of the object into an integer, but this implementation
31. \* technique is not required by the
32. \* Java™ programming language.)
33. \*
34. \* @return  a hash code value for this object.
35. \* @see     java.lang.Object#equals(java.lang.Object)
36. \* @see     java.lang.System#identityHashCode
37. \*/
38. **public** native **int** hashCode();

* hashCode()的意义：hashCode提高搜索效率，时间复杂度O(1),hashCode()的意义：在散列存储中定位存储位置
* 如果两个对象hashCode相同，不一定就相同，只是会存储在同一个桶中，这就是HashMap查找系统元素的逻辑，现在比较hashcode，再比较equals。
* Java对于equals方法和hashcode方法这样规定：如果两个对象相同，那么他们的hashcode值一定要相同。也告诉我们重写equals方法，一定要重写hashcode方法，hashcode要与类中的成员属性相关联，对象相同->成员属性相同->hashcode值一定相同。
* Java对于equals方法和hashcode方法这样规定：如果两个对象的hashCode相同，他们并不一定相同，也就是equals不一定相等。

## 面试题

### Override和Overload区别，分别用在什么场景？

### 若HashCode方法永远返回1会产生什么结果？

### 谈谈hashCode()与equals()

### Object类有哪些方法

### Object.Wait方法能不能被重写，wait能不能被中断？

### Object中哪些方法能被重写

### HashCode方法有什么用

### HashSet元素无序、不可重复，如何元素不可重复？

如果采用equals方法，每增加一个元素就检查一次，如果集合中元素有100万，最坏比较100万次，复杂度O(n)，效率极低，因此Java采用哈希表的原理。

当Set接收一个元素时根据对象的内存地址算出hashCode值，判断属于哪个桶，当两个对象hashCode值相等，只能表明这两个对象存放同一个桶，如果他们equals不等，就会采用链表存储不同对象。

问题：若两个对象equals相等，但不在一个桶，也就是hashCode不相等，那么这两个对象在存放Set时根本没有机会进行比较，会被认为是不同的对象。所以Java对于equals和hashCode方法这样规定：

1. 如果两个对象相同，那么他们hashCode值一定相同。就是说我们重写equals方法，一定要重写hashCode方法
2. 如果两个对象hashCode相同，他们并不一定相同，即equals不一定返回True，只能说这两个对象放在同一个桶中。

### HashSet方法里的hashcode存在哪

### 一个对象的Id=1，通过setId(2)改为2，HashSet里面有几个对象？

### String是可变的吗？

Final数组不可变，final的char数组

### String s=”abc”,String s=new(“abc”)区别

常量池，String的intern()方法

### StringBuffer和StringBuilder区别

### Java的引用类型

《深入理解JVM虚拟机》中

# 基础-反射

## 对象引用类型

* REF: 《深入理解JVM虚拟机》
* Final Reference 强引用
  + 强引用可以直接访问对象
  + 强引用指向的对象任何时候都不会被GC回收
  + 强引用可能会导致内存泄漏
* Soft Reference软引用
  + 软引用对象内存充足时不回收，内存不足时才回收，可用来实现内存敏感的缓存
  + 软引用和引用队列关联（ReferenceQueue），如果该引用对象被回收，JVM会把对象加入此队列。
* Weak Reference弱引用
  + 弱引用强度比软引用强度更弱，被弱引用关联的对象只能生存到下次垃圾回收之前。当GC工作时，无论当前内存是否足够，都会回收该对象。
  + 一旦一个弱引用被GC回收，就会加入到一个注册引用队列中。
* Phantom Reference虚引用
  + 虚引用被称为幽灵引用或者幻影引用，是一种最弱的引用关系。
  + 虚引用随时都有可能被回收；
  + 当视图通过get获取虚引用时，总是失败；
  + 虚引用必须和引用队列一起使用，作用是跟踪整个垃圾回收过程；

## 面试题

#### Java的反射如何实现？

# 基础-集合

### 面试题

#### LinkedHashMap的底层实现

### 快速失败(fail-fast)和安全失败(fail-safe)

## 面试题

### 定时器Timer如何实现？

### ArrayList和LinkedList区别？

### 线程安全和非线程安全的集合有哪些，底层实现？

### 描述HashMap的get方法实现

### HashMap数组的第一个元素存的是什么？

### 遍历HashMap的三种方式？

### HashMap如果只有一个写其他全是读会出什么问题？

### HashMap和HashTable的区别

### HashTable的null的处理是怎样的

### 两个Integer a,b ，swap(a,b)交换实现？

### 两个int a,b , swap(a,b)交换实现？(张勇)

方法1：位运算，公式：（a^b）^a=b

1. swap(**int** a, **int** b) {
2. a = a ^ b;
3. b = a ^ b;
4. a = a ^ b;
5. System.out.println("a:" + a);
6. System.out.println("b:" + b);
7. }

方法2：

1. swap(**int** a, **int** b) {
2. a = a + b;
3. b = a - b;
4. a = a - b;
5. System.out.println("a:" + a);
6. System.out.println("b:" + b);

}

### String s=new String(“abc”)分别在堆栈上新建了哪些对象？

### 集合中如何使用HashCode？

#### Ref: <https://blog.csdn.net/baidu_31657889/article/details/52298367>

#### Ref: <https://blog.csdn.net/zhangyuan19880606/article/details/51240372>

#### Error和RuntimeException的区别

# 锁

## 面试题

### Java有哪些锁？Sychronized、Lock、ReentrantLock、读写锁？

### ReentrantLock与Synchronized的区别？

### 什么是乐观锁、悲观锁？项目中使用场景？

### 可重入的读写锁，可重入是如何实现？

### Synchronized和Lock区别，可重入锁和非可重入锁的区别？

### 可重入锁中对应的wait和notify

### Synchronized加在方法上用的什么锁？

### 可重入锁中的lock和tryLock的区别？

### 读写锁？

### Static 加锁

# IO

## 面试题

### IO会阻塞吗？ReadLine是不是阻塞的？

## 面试题

### HashMap底层实现

# 线程

## 线程创建

## 线程状态

## 面试题

### 线程有几种状态？如何转换，通过什么方法转换？

### 创建线程的几种方法？

### 哪些方法能使线程从运行态变为阻塞态

### 一个线程连续调用start两次会出现什么情况（如何设计start处理这种情况，提示：结合线程状态机制）

### 线程的阻塞方式

### 线程如何退出结束

### 为什么线程运行要执行start，而不直接run

直接run是作为普通方法执行，start才作为线程来运行。

### Sleep和Wait区别

### 进程和线程的区别

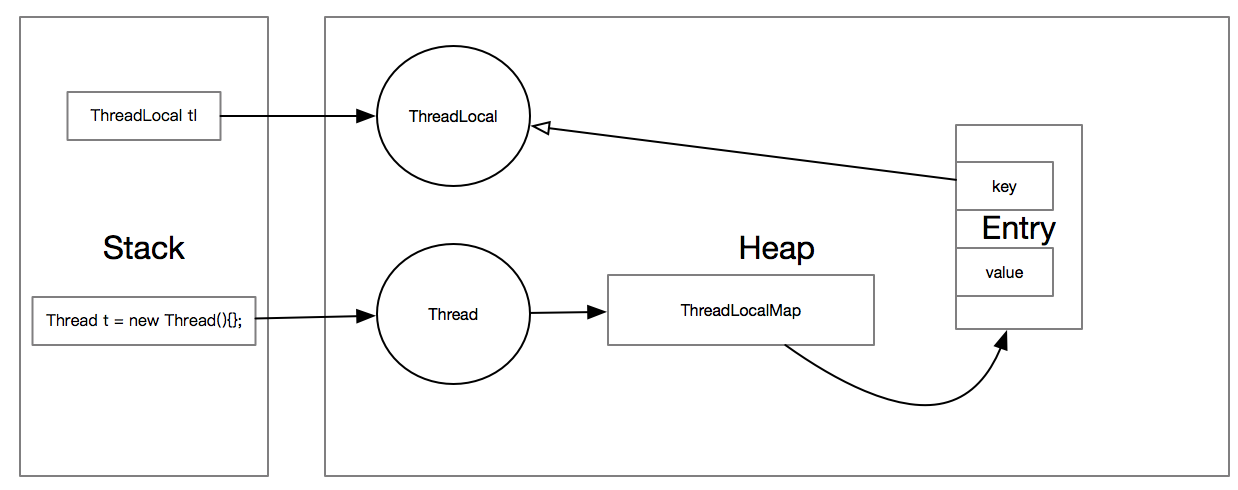
进程可以看作是程序执行时的实例，是一个分配了系统资源的独立实体。每个进程都在各自独立的地址空间执行，一个进程无法访问另一个进程的变量和数据结构。如果一个进程想要访问其他进程的资源，就必须使用进程间通信机制，包括管道、文件、套接字以及其他形式。

线程存在于进程中，共享进程的资源（包括它的对空间）。同一进程里的多个线程将共享同一个堆空间。这跟进程大不相同，一个进程不能直接访问另一个进程的内存。每个线程仍然会有自己的寄存器和栈，而其他线程可以读写堆内存。

### ThreadLocal的使用场景

### ThreadLocal会发生内存泄漏吗？如何解决？

* ThreadLocal实现原理：每个Thread维护一个ThreadLocalMap映射表，映射表的key是ThreadLocal实例，并且使用ThreadLocal的弱引用，value是具体需要存储的对象。
* Java引用类型知识点：弱引用，每次都被GC回收！
* 如果ThreadLocal对象是局部变量，比如某方法体中声明，方法体结束后，没有变量引用该实例，当发生GC时，该实例被回收，ThreadLocalMap映射表中的key变为null，外部将无法获取key对应的value，如果当前线程一直存活，那么就会有一条强引用链：Thread实例—>ThreadLocalMap->Entry->value，导致value对应的Object一直无法被回收，产生内存泄漏！



* ThreadLocal的get/set/remove方法都实现了对所有key为null的value的清除，但仍可能发生泄漏：可能使用了ThreadLocal.get/set/方法后发生GC，之后不调用get/set/remove方法，为null的value就不会被清除。
* **解决方法一：每次使用完ThreadLocal，都调用它的remove()方法清除数据**
* **解决方法二：将ThreadLocal变量定义为private static，一直存在ThreadLocal的强引用，保证任何时候通过ThreadLocal的弱引用访问到Entry的value，进而清除！**

## 线程状态

# 线程池

## 面试题

### Java线程池哪些，如何实现？源码？

### 说说线程池，线程池中的一个线程执行完任务后处理哪种状态

### Thread类的interrupt，interrupted，isInterrupted方法的区别

### Spring线程池如何实现，源码？

### Java线程池中基于缓存和基于定长的两种线程池，当请求太多时分别如何处理？定长的队列，如果队列满了？交换进磁盘？基于缓存的线程池解决方法呢？

### 项目中用的线程池，工作原理，任务拒绝策略

# 网络编程

## IO

## BIO

## NIO

### 原生NIO在JDK1.7版本存在epoll bug

导致Selector空轮询，最终导致CPU100%。官方声称在JDK1.6版本的update18修复，但是直到JDK1.7版本问题仍旧存在，只不过该Bug发生概率降低了一些而已，并没有根本解决。

## AIO

## Http协议

## TCP协议

## 面试题

### http协议的格式，get和post区别

### TCP与UDP的区别

### TCP三次握手流程

### 什么是NIO

### Socket中的accept什么用

### 两个不同IP地址的计算机之间如何通信

### 描述Http请求过程

# 并发编程

## 并发&并行

## 线程间通信

## 面试题

### Volatile是什么？它的用途是什么？

### Volatile与Synchronized区别

### ConcurrentHashMap是如何加锁的？是不是分段越多越好？

### ConcurrentHashMap是如何加锁的？如果刚计算完一段，此段发生变化，该如何处理？

### 如何解决并发问题

### 原子类有哪些？如何实现？

### CountDownLatch实现原理及适用场景？

* 创建CountDownLatch实例：new CountDownLatch(3)，将count参数赋值给内部计数器state

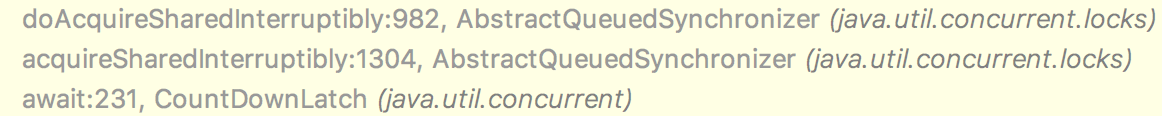
1. /\*\*
2. \* Constructs a {@code CountDownLatch} initialized with the given count.
3. \*
4. \* @param count the number of times {@link #countDown} must be invoked
5. \*        before threads can pass through {@link #await}
6. \* @throws IllegalArgumentException if {@code count} is negative
7. \*/
8. **public** CountDownLatch(**int** count) {
9. **if** (count < 0) **throw** **new** IllegalArgumentException("count < 0");
10. **this**.sync = **new** Sync(count);
11. }
12. /\*\*
13. \* Synchronization control For CountDownLatch.
14. \* Uses AQS state to represent count.
15. \*/
16. **private** **static** **final** **class** Sync **extends** AbstractQueuedSynchronizer {
17. **private** **static** **final** **long** serialVersionUID = 4982264981922014374L;
19. Sync(**int** count) {
20. setState(count);
21. }
23. **int** getCount() {
24. **return** getState();
25. }
27. **protected** **int** tryAcquireShared(**int** acquires) {
28. **return** (getState() == 0) ? 1 : -1;
29. }
31. **protected** **boolean** tryReleaseShared(**int** releases) {
32. // Decrement count; signal when transition to zero
33. **for** (;;) {
34. **int** c = getState();
35. **if** (c == 0)
36. **return** **false**;
37. **int** nextc = c-1;
38. **if** (compareAndSetState(c, nextc))
39. **return** nextc == 0;
40. }
41. }
42. }
43. /\*\*
44. \* The synchronization state.
45. \*/
46. **private** **volatile** **int** state;
48. /\*\*
49. \* Returns the current value of synchronization state.
50. \* This operation has memory semantics of a {@code volatile} read.
51. \* @return current state value
52. \*/
53. **protected** **final** **int** getState() {
54. **return** state;
55. }
57. /\*\*
58. \* Sets the value of synchronization state.
59. \* This operation has memory semantics of a {@code volatile} write.
60. \* @param newState the new state value
61. \*/
62. **protected** **final** **void** setState(**int** newState) {
63. state = newState;
64. }

* await()方法

调用await()方法阻塞当前线程，并将当前线程封装加入到等待队列中，直到state等于零或当前线程被中断；方法栈调用如下：

await()源码：

1. /\*\*
2. \* Causes the current thread to wait until the latch has counted down to
3. \* zero, unless the thread is {@linkplain Thread#interrupt interrupted}.
4. \*
5. \* <p>If the current count is zero then this method returns immediately.
6. \*
7. \* <p>If the current count is greater than zero then the current
8. \* thread becomes disabled for thread scheduling purposes and lies
9. \* dormant until one of two things happen:
10. \* <ul>
11. \* <li>The count reaches zero due to invocations of the
12. \* {@link #countDown} method; or
13. \* <li>Some other thread {@linkplain Thread#interrupt interrupts}
14. \* the current thread.
15. \* </ul>
16. \*
17. \* <p>If the current thread:
18. \* <ul>
19. \* <li>has its interrupted status set on entry to this method; or
20. \* <li>is {@linkplain Thread#interrupt interrupted} while waiting,
21. \* </ul>
22. \* then {@link InterruptedException} is thrown and the current thread's
23. \* interrupted status is cleared.
24. \*
25. \* @throws InterruptedException if the current thread is interrupted
26. \*         while waiting
27. \*/
28. **public** **void** await() **throws** InterruptedException {
29. sync.acquireSharedInterruptibly(1);
30. }



入队逻辑：

1. /\*\*
2. \* Acquires in shared interruptible mode.
3. \* @param arg the acquire argument
4. \*/
5. **private** **void** doAcquireSharedInterruptibly(**int** arg)
6. **throws** InterruptedException {
7. **final** Node node = addWaiter(Node.SHARED);
8. **boolean** failed = **true**;
9. **try** {
10. **for** (;;) {
11. **final** Node p = node.predecessor();
12. **if** (p == head) {
13. **int** r = tryAcquireShared(arg);
14. **if** (r >= 0) {
15. setHeadAndPropagate(node, r);
16. p.next = **null**; // help GC
17. failed = **false**;
18. **return**;
19. }
20. }
21. **if** (shouldParkAfterFailedAcquire(p, node) &&
22. parkAndCheckInterrupt())
23. **throw** **new** InterruptedException();
24. }
25. } **finally** {
26. **if** (failed)
27. cancelAcquire(node);
28. }
29. }

* countdown()方法

抵用countDown()方法使state值减1，如果state等于零则唤醒等待队列中的线程。

源码：

1. /\*\*
2. \* Decrements the count of the latch, releasing all waiting threads if
3. \* the count reaches zero.
4. \*
5. \* <p>If the current count is greater than zero then it is decremented.
6. \* If the new count is zero then all waiting threads are re-enabled for
7. \* thread scheduling purposes.
8. \*
9. \* <p>If the current count equals zero then nothing happens.
10. \*/
11. **public** **void** countDown() {
12. sync.releaseShared(1);
13. }
14. /\*\*
15. \* Releases in shared mode.  Implemented by unblocking one or more
16. \* threads if {@link #tryReleaseShared} returns true.
17. \*
18. \* @param arg the release argument.  This value is conveyed to
19. \*        {@link #tryReleaseShared} but is otherwise uninterpreted
20. \*        and can represent anything you like.
21. \* @return the value returned from {@link #tryReleaseShared}
22. \*/
23. **public** **final** **boolean** releaseShared(**int** arg) {
24. **if** (tryReleaseShared(arg)) {
25. doReleaseShared();
26. **return** **true**;
27. }
28. **return** **false**;
29. }
30. /\*\*
31. \* Release action for shared mode -- signals successor and ensures
32. \* propagation. (Note: For exclusive mode, release just amounts
33. \* to calling unparkSuccessor of head if it needs signal.)
34. \*/
35. **private** **void** doReleaseShared() {
36. /\*
37. \* Ensure that a release propagates, even if there are other
38. \* in-progress acquires/releases.  This proceeds in the usual
39. \* way of trying to unparkSuccessor of head if it needs
40. \* signal. But if it does not, status is set to PROPAGATE to
41. \* ensure that upon release, propagation continues.
42. \* Additionally, we must loop in case a new node is added
43. \* while we are doing this. Also, unlike other uses of
44. \* unparkSuccessor, we need to know if CAS to reset status
45. \* fails, if so rechecking.
46. \*/
47. **for** (;;) {
48. Node h = head;
49. **if** (h != **null** && h != tail) {
50. **int** ws = h.waitStatus;
51. **if** (ws == Node.SIGNAL) {
52. **if** (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
53. **continue**;            // loop to recheck cases
54. unparkSuccessor(h);
55. }
56. **else** **if** (ws == 0 &&
57. !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
58. **continue**;                // loop on failed CAS
59. }
60. **if** (h == head)                   // loop if head changed
61. **break**;
62. }
63. }

* 使用场景

客户端一个请求，需要服务端多个信息查询任务，总的执行时间时所有任务执行的时间累加。如果使用CountDownLatch，把多个信息查询分成多个线程执行，总的执行时间为耗时最长的线程的执行时间，大大缩短整个请求的处理时间。

### AtomicInteger底层原理？

### CAS机制会出现什么问题？

### JCU包都哪些类

# 性能调优

## 常用分析工具

## 分析方法

# JVM

## 字节码

## 内存模型

## HSDB用法

* HSDB工具可以查看运行时内存内容
* REF: <https://caoliujia.com>
* REF: <http://rednaxelafx.iteye.com/blog/1847971>
* Mac环境，root权限
* sudo java –classpath “$JAVA\_HOME/lib/sa-jdi.jar” sum.jvm.hotspot.HSDB
* jps 获取线程ID

## 垃圾回收

## 类加载

## 面试题

### rt.jar被什么类加载器加载，什么时间加载？

### 自己写的类被什么类加载器加载，什么时间加载？

### 自己写的两个不同的类是被同一个类加载器加载的吗？为什么？

### Java的内存模型，方法区存什么？

### 引用计数器与GC Root可达性分析的区别？

### GC机制及算法

#### 双亲委派机制

### JVM命令有哪些

### 如何把java内存的数据全部dump出来

### 如何手动触发全量回收垃圾，如何立即触发垃圾回收？

### 如何确定JVM中哪些对象是可回收的？

* REF:《深入理解JVM虚拟机》

GC在对堆进行回收之前，会判断哪些对象是可继续存活，哪个对象是可回收的，（如何确定对象已死）

* 可达性分析算法，JAVA采用此种算法，这个算法过程通过一系列的称为“GC Roots”对象作为起点，从这些起点向下搜索，搜索整个引用链过程，如果一个对象和引用链没有任何关系时，表明这个对象是可以回收的。

可作为GC Roots的对象有：

* java虚拟机栈（栈帧中本地变量表）中引用的对象
* 本地方法栈中JNI（一般说的Native方法）引用的对象
* 方法区中类静态属性引用的对象
* 方法区中常量引用的对象
* 引用计数器算法，每个对象添加一个引用计数器，对象每增加一个引用，计数器+1
* **缺陷：很难解决对象循环引用的问题**

### 何时出现内存泄漏，内存泄漏会抛出哪些异常？

### 静态内部类加载到哪个区？

方法区

### class文件编译后加载到哪里？

框架部分

# Mybatis

## 面试题

### Mybatis如何映射表结构

# 设计模式

## 面试题

### 代理模式

### 适配器和代理模式的区别？

### 门面模式，类图（外观模式）

### 线程安全的单例模式

### 策略模式和工厂模式，项目中用到的

### 模板方法模式

### 开闭原则

# Spring

## 特性

### IOC

### AOP

面向切面编程，在我们的应用中，经常需要做一些事情，但是这些事情与核心业务无关，比如，要记录所有update方法的执行时间时间，操作人等等信息，记录到日志，Spring AOP可以在不修改update代码的前提下完成。

AOP通过动态代理实现，两种方式JDK动态代理和CGLIB动态代理。JDK动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是InvocationHandler接口和Proxy类。

如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

### 动态代理

JDK 动态代理类和委托类需要都实现同一个接口。也就是说只有实现了某个接口的类可以使用Java动态代理机制。但是，事实上使用中并不是遇到的所有类都会给你实现一个接口。因此，对于没有实现接口的类，就不能使用该机制。而CGLIB则可以实现对类的动态代理。

## 面试题

### SpringBean的生命周期

### BeanFactory和ApplicationContext有什么区别？

### Spring的监听器

### Spring Bean的几种配置方式，注入Bean的几种方式？

### AOP的底层实现，动态代理如何动态，假如有100个对象，如何动态的为这100个对象动态代理？

### Spring IOC如何实现？

### Spring AOP的实现原理

### Spring注解

### 动态代理有几种？JDK和cglib区别？

### Spring事务实现方式及底层原理

1. 编码方式
2. 声明式事务管理方式 一种是XML方式，一种是业务方法上@Transaction注解
3. 原理：
4. 划分处理单元—IOC- 由于spring解决的问题是对单个数据库进行局部事务处理的，具体的实现首相用spring中的IOC划分了事务处理单元。并且将对事务的各种配置放到了ioc容器中（设置事务管理器，设置事务的传播特性及隔离机制）
5. AOP拦截需要事务处理的类 Spring事务处理模块是通过AOP功能来实现声明式事务处理的，具体操作（比如事务实行的配置和读取，事务对象的抽象），用TransactionProxyFactoryBean接口来使用AOP功能，生成proxy代理对象，通过TransactionInterceptor完成对代理方法的拦截，将事务处理的功能编织到拦截的方法中。读取ioc容器事务配置属性，转化为spring事务处理需要的内部数据结构（TransactionAttributeSourceAdvisor），转化为TransactionAttribute表示的数据对象。
6. 实现事务（生成、提交、回滚、挂起） spring委托给具体的事务处理器实现。实现了一个抽象和适配。适配的具体事务处理器：DataSource数据源支持、hibernate数据源事务处理支持、JDO数据源事务处理支持，JPA、JTA数据源事务处理支持。这些支持都是通过设计PlatformTransactionManager、AbstractPlatforTransaction一系列事务处理的支持。 为常用数据源支持提供了一系列的TransactionManager。
7. 结合 PlatformTransactionManager实现了TransactionInterception接口，让其与TransactionProxyFactoryBean结合起来，形成一个Spring声明式事务处理的设计体系。

# SpringMVC

## Spring MVC的启动流程

## Spring MVC 处理请求流程



# SpringBoot

# MQ

## MQ架构

## MQ原理

## 面试题

### RabbitMQ消息队列丢失消息，重复消费问题

### qmq的消息实现机制（去哪网封装的消息队列）

# 缓存

## 面试题

### CPU能不能直接和二级缓存打交道？

### 本地缓存过期策略怎么设置，一致性怎么保证？

### Memcached和Redis的区别

### 用HashMap实现Redis会出现什么问题？（死锁？死循环？）

可使用ConcurrentHashMap

### Redis数据类型

### Redis集群宕机，数据迁移问题

### Redis能把内存空间交换到磁盘吗？

### Redis缓存是分布式存储吗？不同服务器上的数据是否重复？GuavaCache呢？是否重复？

不同的机器存的数据不同。

### 缓存穿透

* REF: http://ifeve.com/concurrency-cache-cross/
* 描述：查询缓存中不存在的key，导致直接查询DB。如有恶意攻击，会对DB造成很大压力，甚至宕机。
* 解决方法：自定义空值，表示不存在的key对应的值，例如：key=”$$”或者key=”&&”，在逻辑中当获取”&&”或者”&&”时，进行对应的逻辑，避免读取数据库；如下：

1. String ticket = RedisUtils.get("XX\_TICKET");
2. **if**(ticket != null && "$$".equals(ticket)){
3. System.out.println("用户没有优惠券");
4. }

### 缓存并发

* REF: http://ifeve.com/concurrency-cache-cross/
* 描述：指缓存失效，导致多个请求同时查询DB，同时设置缓存，如果并发过大，造成DB压力过大，甚至宕机。针对同一个Key。
* 解决方法：对缓存查询加锁，如果key不存在，加锁->查询DB并放入缓存->解锁；其他线程发现有锁就等待，等解锁后返回数据或者查询DB。

### 缓存失效

* REF: http://ifeve.com/concurrency-cache-cross/
* 描述：一般设置缓存失效时间可能是1分钟、5分钟，在高并发的情况下，会引起同时新增大量失效时间相同的key，而这些key都是同一时间点存放，这将导致在某一时刻大量缓存失效，使得大量请求转向DB，造成DB压力过大。针对大量key。
* 解决方法：分散失效时间，在原有失效时间基础上再加随机时间，避免集体失效。

### 缓存雪崩

* 描述，高并发情况下，大量缓存穿透会引起缓存雪崩

### 缓存一致性问题

* 实时同步、强一致性：同步更新，代码封装
* 准实时同步：MQ
* 缓存失效机制：弱一致性
* 异步作业调度：最终一致性

# Memcached

# Redis

* REF: <https://blog.csdn.net/sunhuiliang85>

## 数据类型

## 队列

## 发布订阅

## Cluster

## 一致性哈希算法

* <https://blog.csdn.net/xlgen157387/article/details/79544524>

## Redis持久化-AOF

* REF: <https://blog.csdn.net/sunhuiliang85/article/details/75453231>

## Redis持久化-RDB

* REF: <https://blog.csdn.net/sunhuiliang85/article/details/75329757>

# RPC

## Dubbo

## Hession

## 面试题

### 类序列化的版本号的用途，如果没有指定版本号，系统会怎么处理？如果加了字段会怎样？

### 说说RPC

# Netty

* REF: <https://www.toutiao.com/a6540511780894933507/>

## Netty简介

## Netty优势

1. API使用简单，开发门槛低；
2. 功能强大，预置了多种编解码功能，支持多种主流协议；
3. 定制能力强，可以通过ChannelHandler对通信框架进行灵活的扩展；
4. 性能高，通过与其他业界主流的NIO框架比，Netty的综合性能最优；
5. 成熟、稳定、Netty修复了已经发现的所有JDK NIO BUG，业务开发人员不需要再为NOI的Bug烦恼；
6. 社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会被加入；
7. 经历了大规模的商业应用考验，质量已经得到验证。在互联网、大数据、网络游戏、企业应用、电信软件等众多行业得到成功商用，证明了它可以完全满足不同行业的商业应用。

## Netty使用场景

1. 构建高性能、低时延的各种Java中间件，例如MQ、分布式服务框架、ESB消息总线等，Netty主要作为基础通信框架提供高性能、低时延的通信服务；
2. 公有或者私有协议栈的基础通信框架，例如可以基于Netty构建异步、高性能的WebSocket协议栈；
3. 各领域应用，例如大数据、游戏等，Netty作为高性能的通信框架用于内部各个模块的数据分发、传输和汇总等，实现模块之间高性能通信。

## Netty线程模型

1. Netty使用EventLoop来处理连接上的读写事件，而一个连接上的所有请求都保证在一个EventLoop中被处理；
2. 一个EventLoop中只有一个Thread，所以也就实现了一个连接上的所有事件只会在一个线程中被执行。
3. 一个EventLoopGroup包含多个EventLoop，可以一个EventLoop当做是Reactor线程模型中的一个线程，而一个EventLoopGroup类似于一个ExecutorService

## Netty的零拷贝

“零拷贝”是指计算机操作的过程中，CPU不需要为数据在内存之间的拷贝消耗资源。通常是指计算机在网络上发送文件时，不需要将文件内容拷贝到用户空间（User Space）而直接在内核空间（Kernal Space）中传输到网络的方式。

## Netty内部执行流程

1. Netty的接收和发送ByteBuffer采用Direct Buffers，使用堆外直接内存进行Socket读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存（Heap Buffers）进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中，然后才写入Socket中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。
2. Netty提供了组合Buffer对象，可以聚合多个ByteBuffer对象，用户可以像操作一个Buffer那样方便的对组合Buffer进行操作，避免了传统内存拷贝的方式将几个小Buffer合并成一个大的Buffer。
3. Netty的文件传输采用了TransferTo方法，它可以直接将文件缓冲区的数据发送到目标Channel，避免了传统通过循环write方式导致的内存拷贝问题。

## Netty重连实现

1. 心跳机制检测连接存活
2. 启动时连接重试
3. 运行中连接断开时重试

# 高并发

## 面试题

### Nginx的请求转发算法，如何配置根据权重转发

#### 数据库读写分离何时强制读主库，读哪个从库是根据什么决定，Mysql从库同步方式是什么？

### Web的Http请求如何响应时间变长导致处理请求变少，该如何处理？

使用队列，将处理不了的请求入队异步处理，web如何实现队列？

# 分布式

## CAP理论

## Base理论

## Paxos理论

## 分布式事务

### 什么是TCC

TCC(Try-Confirm-Cancel)

开源实现 https://github.com/changmingxie/tcc-transaction

## 分布式锁

## 分布式存储

## 面试题

### 分布式高并发系统接口如何保证幂等性？

接口的幂等性保证多次请求产生的最终结果是一致的 ，即“处理过程”不会重复执行，无论请求次数多少，仅执行一次。

对于单台服务器：

对于分布式系统，需要多服务器共同提供幂等性，关键是提供全局服务，通过检查数据状态位是否为终态或者判断全局是否有响应的可执行标识来判断，进行幂等设计。例如使用防重表设置数据有效期，当有接口执行时插入表或存到缓存中，再次执行时校验是否存在或者校验状态位。

### 分布式锁实现方式

### 分布式事务的实现方式

### 5台服务器如何选举Leader（选举算法）

# 架构

## 面试题

### 微服务的理解

### 什么是SOA，与微服务的区别

# Linux

1字节：1Byte

1位：1bit

1Byte = 8bit

1KB = 1024B = B

1MB = 1024KB = 1024 \* 1024 B = \* B = B

1GB = 1024MB = 1024 \* B = B

4GB = B

## X86的控制寄存器

* REF: <https://blog.csdn.net/wyt4455/article/details/8691500>

四个32位的控制寄存器，分别是CR0、CR1、CR2、CR3。

* CR2和CR3

CR2和CR3用于分页控制。CR3包含存放页目录表页面的物理地址，因此CR3也被称为PDBR。待续…..

## 内存管理

## 用户空间与内核空间

* REF: <https://blog.csdn.net/lizuobin2/article/details/51791863>

Linux中，将程序的运行空间分为内核空间和用户空间，即内核态和用户态，在逻辑上他们之间是相互隔离的，因此用户程序不能访问内核数据，也无法使用内核函数。当用户进程必须访问内核或者使用某个内核函数时，就使用系统调用（SCI），在Linux中，系统调用是用户空间访问内核空间的唯一途径。

## API

应用程序接口（Application Programming Interface），是在用户空间直接使用的函数接口。是一些预定义的函数，比如read(),malloc(),free(),abs()函数等，这些函数都具有一定的功能，跟内核没有必然的联系。提供应用程序与开发人员访问系统例程的能力，开发人员无需访问源码，无需理解内部工作机制。

## 系统调用(SCI)

系统调用是一种特殊接口，通过该接口，用户可以访问内核空间。系统调用规定了用户进程进入内核的具体位置。

系统调用是用户进程进入内核空间的接口层，它本身并非内核函数，但它是由内核函数实现，进入内核后，不同的系统调用会找到各自对应的内核函数，这些内核函数被称为系统调用的“服务例程”。比如系统调用getpid实际是调用内核函数sys\_getpid()，或者说系统调用getpid是 服务例程sys\_getpid()的“封装例程”，调用流程是：用户进程->系统调用->内核函数->返回用户空间。

Linux系统调用（SCI，System Call Interface）的实现机制实际上是一个多路汇集以及分解的过程，该汇集点就是0x80中断这个入口点（X86系统结构）。也就是说，所有系统调用都从用户空间中汇聚到0x80中断点，同时保存具体的系统调用号。当0x80中断处理程序运行时，将根据系统调用号对不同的系统调用分别处理（调用不同的内核函数处理）。

为了通过系统调用号来调用不同的内核服务例程，**系统必须维护一个系统调用表**，这个表实质上就是系统调用号与内核服务函数的对照表。Linux是用数组**sys\_call\_tabl**来作为这个表，在这个表的每个表项存放着对应内核服务器服务例程的指针，而该表项的下标就是该内核服务例程的系统调用号。Linux规定，在I386体系中，系统调用号由处理器的寄存器eax来传递。

系统调用的接口的主要任务是把进程从用户态切换到内核态。在具有保护机制的计算机系统中，用户必须通过软件中断或者陷阱，才能使进程从用户态切换到内核态。如图-系统调用过程



图-系统调用过程

系统调用的执行需要一个用户空间到内核空间的状态转换，不同平台具有不同的指令可以完成这种转换，这种这令也称作操作系统陷入（operating system trap）指令。

System\_call函数到系统调用服务例程，所有的系统调用都会统一跳转到这个地址进而执行system\_call函数，到2.6.23版为止，内核提供的系统调用已经达到325个，那么system\_call函数又如何派发他们到各自的服务例程呢？软中断指令 int 0x80 执行时，系统调用号会放入eax寄存器，同时sys\_call\_table每一项占用4个字节。System\_call函数可以读取eax寄存器获得当前系统调用的系统调用号，将其乘以4生成偏移地址，然后以sys\_call\_table为基址，基址加上偏移地址所指向的内容即是应该执行的系统用的服务例程的地址。

### 面试题

#### 用户空间（进程）是否有高端内存概念？

用户进程没有高端内存概念。只有在内核空间才存在高端内存。32位内核下，用户进程最多只有可以访问3G物理内存，而内核进程可以访问所有物理内存。

#### 64位内核中有高端内存吗？

目前现实中，64位Linux内核不存在高端内存，因为64位内核可以支持超过512GB内存，若物理内存超过内核地址空间范围，就会存在高端内存。

#### 用户进程能访问多少物理内存？内核代码能访问多少物理内存？

32位系统，用户进程最大可以访问3GB，内核代码可以访问所有物理内存。

64位系统，用户进程最大可以访问超过512GB，内核代码可访问所有。

#### 高端内存和物理地址、逻辑地址、线性地址的关系？

高端内存只和逻辑地址有关系，和物理地址、线性地址没有直接关系。

#### 为什么不把所有的地址空间都分配给内核？

若把所有地址空间都给内核，用户进程无法使用内存，无法保证内核使用内存和用户进程不起冲突。

## SCI与API 区别

API是函数定义，规定了函数的功能，跟内核无直接关系。而SCI是通过中断向内核发起请求，实现内核提供的某些服务。

程序员调用的是API函数，API函数通过系统调用共同完成函数功能。因此API是提供给应用的接口，一组函数，是与程序员进行直接交互。

系统调用则不与程序员交互，而是根据API函数，通过软中断机制向内核提交请求，从而获取内核服务的接口。

一些API所提供的功能涉及到与内核空间进行交互，那么这些API内部会封装系统调用，如果不涉及到内核空间则不会封装系统调用。也就是说，API和系统并没有严格的一一对应关系，一个API可能恰好只对应0个或者一个或者多个系统调用；一个系统调用可能被多个API调用。

对于程序员来说：系统调用和API都是函数；系统调用在内核完成，API则是在函数库中实现。

**API是用户程序直接可使用的函数接口，如果每个操作系统都有属于自己的API，那么应用程序的可移植性将会很差。基于POSIX（Portable Operating System Interface）标准的API有用很好的可移植性，它定义了一套POSIX兼容标准，这使得这个标准实现的API可以在各种版本的UNIX中使用。现如今，它也可以在除UNIX之外的操作系统中使用，比如Linux，WindowsNT等。**

## 系统命令

每一个系统命令其实就是一个可执行的程序，这些可执行程序的实现调用了某些系统调用。并且，这些可执行程序又分为普通用户和管理员使用的命令。普通用户可用的命令位于/bin目录下，管理员命令在/sbin目录下。

## 内核函数

内核函数和用户空间中的函数并无两样，只不过内核函数只在内核中实现。虽然系统调用是用户进程进入内核的唯一途径，但系统调用并不真正实现其功能，而是通过内核函数完成，也就是说，用户进程通过某个系统调用进入内核后，会直接去执行这个系统调用对应的内核函数。这个内核函数也称为系统调用的服务例程。

由于内核函数是在内核中实现的，它必须符合内核编程的规则，比如函数名以sys\_开始，函数定义时加asmlinkage标识符等。

## read()系统

* REF: <https://www.ibm.com/developerworks/cn/linux/l-cn-read/index.html>

## write()系统

## DMA

### DMA原理

* REF:<https://zh.wikipedia.org/wiki/%E7%9B%B4%E6%8E%A5%E8%A8%98%E6%86%B6%E9%AB%94%E5%AD%98%E5%8F%96>
* REF: <https://baike.baidu.com/item/DMA/2385376>

直接内存访问（Direct Memory Access，DMA）是计算机科学中的一种内存访问技术。它允许某些电脑内部的硬件子系统（电脑外设），可以独立地直接读写系统内存，而不需要中央处理器（CPU）接入处理。在同等程度的处理器负担下，DMA是一种快速的数据传送的方式。很多硬件的系统会使用DMA，包含硬盘控制器、绘图显卡、网卡和声卡。

原理：DMA传输将数据从一个地址空间复制到另外一个地址空间。当CPU初始化这个传输动作，传输动作本书是有DMA控制器来执行和完成。整个过程并没有让CPU工作拖延，可以继续处理其他工作。

在实现DMA传输时，有DMA控制器直接掌管总线，因此，存在着一个总线控制权转移问题。即DMA传输前，CPU要把总线控制权交给DMA控制器，而在结束DMA传输后，DMA控制器应立即把总线控制权再交回给CPU。一个完整的DMA传输过程必须经过DMA请求、DMA响应、DMA传输、DMA结束4个步骤。

* **请求**：CPU对DMA控制器初始化，并向I/O接口发出操作指令，I/O接口提出DMA请求；
* **响应**：DMA控制器对DMA请求判别优先级及屏蔽，向总线裁决逻辑提出总线请求。当CPU执行完当前总线周期即可释放总线控制权。此时，总线裁决逻辑输出总线应答，表示DMA已经响应，通过DMA控制器通知I/O接口开始DMA传输；
* **传输**：DMA控制器获取总线控制权后，CPU即刻挂起或者只执行内部操作，由DMA控制器输出读写命令，直接控制RAM与I/O接口进行DMA传输。在DMA控制器的控制下，在存储器和外部设备之间直接进行数据传送，在传送过程中不需要CPU的参与。开始需要提供要传送的数据的起始位置和数据长度。
* **结束**：当完成数据传送之后，DMA控制器即释放总线控制权，并向I/O接口发出结束信号。当I/O接口收到结束信号后，一方面停止I/O设备的工作，另一方面向CPU提出中断请求，使CPU从不介入状态解脱，并执行一段检查本次DMA传输操作不正确性的代码。最后，带着本次操作结果及状态继续执行原来的程序。

由此可见，DMA传输方式无需CPU直接控制传输，也没有中断处理方式那样保留现场和恢复现场的过程，通过硬件为RAM与I/O设备开辟一条直接传送数据的通路，使CPU得效率大为提高。

### DMA传输方式

DMA技术的出现，使得外围设备可以通过DMA控制器直接访问内存，与此同时，CPU可以继续执行程序，那么DMA控制器与CPU怎样分时使用内存呢？通常采用三种方法：停止CPU访问内存；周期挪用；DMA与CPU交替访问内存。

* **停止CPU访问内存**

当外围设备要求传送一批数据时，由DMA控制器发一个停止信号给CPU，要求CPU放弃对地址总线、数据总线和有关控制总线的使用权．DMA控制器获得总线控制权以后，开始进行数据传送．在一批数据传送完毕后，DMA控制器通知CPU可以使用内存，并把总线控制权交还给CPU；

**优点:** 控制简单，它适用于数据传输率很高的设备进行成组传送。

**缺点:** 在DMA控制器访内阶段，内存的效能没有充分发挥，相当一部分内存工作周期是空闲的。这是因为，外围设备传送两个数据之间的间隔一般总是大于内存存储周期，即使高速I/O设备也是如此。例如，软盘读出一个8位二进制数大约需要32us，而半导体内存的存储周期小于0.5us，因此许多空闲的存储周期不能被CPU利用．

* 周期挪用

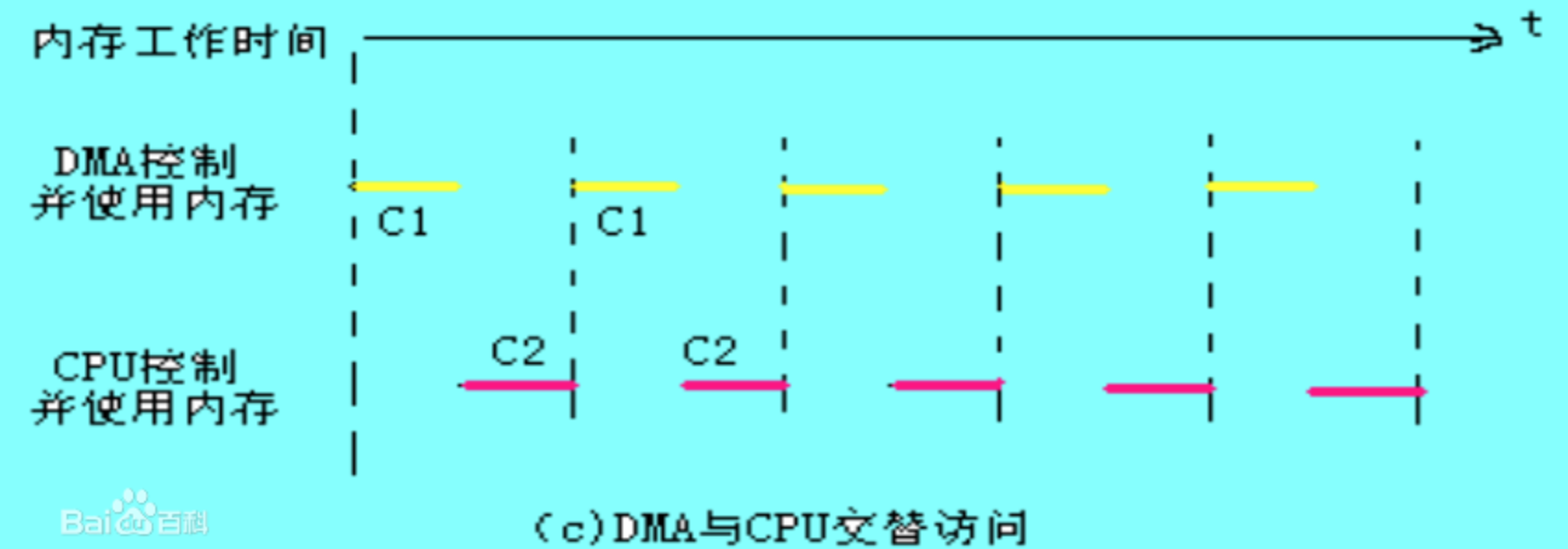
当I/O设备没有DMA请求时，CPU按程序要求访问内存；一旦I/O设备有DMA请求，则由I/O设备挪用一个或几个内存周期。

I/O设备要求DMA传送时可能遇到两种情况：

1. 此时CPU不需要访内，如CPU正在执行乘法指令。由于乘法指令执行时间较长，此时I/O访内与CPU访内没有冲突，即I/O设备挪用一二个内存周期对CPU执行程序没有任何影响。
2. I/O设备要求访内时CPU也要求访内，这就产生了访内冲突，在这种情况下I/O设备访内优先，因为I/O访内有时间要求，前一个I/O数据必须在下一个访问请求到来之前存取完毕。显然，在这种情况下I/O 设备挪用一二个内存周期，意味着CPU延缓了对指令的执行，或者更明确地说，在CPU执行访内指令的过程中插入DMA请求，挪用了一二个内存周期。 与停止CPU访内的DMA方法比较，周期挪用的方法既实现了I/O传送，又较好地发挥了内存和CPU的效率，是一种广泛采用的方法。但是I/O设备每一次周期挪用都有申请总线控制权、建立线控制权和归还总线控制权的过程，所以传送一个字对内存来说要占用一个周期，但对DMA控制器来说一般要2—5个内存周期(视逻辑线路的延迟而定)。因此，周期挪用的方法适用于I/O设备读写周期大于内存存储周期的情况。

* DMA与CPU交替访问内存

如果CPU的工作周期比内存存取周期长很多，此时采用交替访内的方法可以使DMA传送和CPU同时发挥最高的效率。



此图是DMA与CPU交替访内的详细时间图．假设CPU工作周期为1.2us，内存存 取周期小于0.6us，那么一个CPU周期可分为C1和C2两个分周期，其中C1专供DMA控制器访内，C2专供CPU访内。

这种方式不需要总线使用权的申请、建立和归还过程，总线使用权是通过C1和C2分时制的。CPU和DMA控制器各自有自己的访内地址寄存器、数据寄存器和读/写信号等控制寄存器。在C1周期中，如果DMA控制器有访内请求，可将地址、数据等信号送到总线上。在C2周期中，如CPU有访内请求，同样传送地址、数据等信号。事实上，对于总线，这是用C1，C2控制的一个多路[转换器](https://baike.baidu.com/item/%E8%BD%AC%E6%8D%A2%E5%99%A8)，这种总线控制权的转移几乎不需要什么时间，所以对DMA传送来讲效率是很高的。

这种传送方式又称为“透明的DMA”方式，其来由是这种DMA传送对CPU来说，如同透明的玻璃一般，没有任何感觉或影响。在透明的DMA方式下工作，CPU既不停止主程序的运行，也不进入等待状态，是一种高效率的工作方式。当然，相应的硬件逻辑也就更加复杂。

## RDMA

远程直接内存访问（Remote Direct Memory Access，RDMA）是一种直接存储器访问技术，它将数据直接从一台计算机的内存传输到另一台计算机，无需双方操作系统接入。这允许高通量、低延迟的网络通信，尤其适合在大规模并行计算机集群中使用。

## 进程间通信(IPC)

### 进程介绍

* REF: <https://zh.wikipedia.org/wiki/%E8%A1%8C%E7%A8%8B%E9%96%93%E9%80%9A%E8%A8%8A>

进程间通信(IPC，*Inter-Process Communication*)，指至少两个进程或者线程间传送数据或信号的一些技术或方法。

进程是计算机系统分配资源的最小单位（严格来说是线程）。每个进程都有自己的一部分独立的系统资源，彼此是隔离的。为了能使不同的进程互相访问资源并进行协调工作，才有了进程间通信。通常，使用进程间通信的两个应用可以被分为客户端和服务端（Client/Server,C/S网络架构），客户端进程请求数据，服务端响应客户端的数据请求。有一些应用本身既是服务又是客户端，这是分布式计算中，时常可以见到。这些进程可以运行在同一计算机上或网络连接的不同计算机上。

使用IPC的理由：

* 信息共享：Web服务器，通过网页浏览器使用进程间通信来共享Web文件（网页等）和多媒体；
* 加速：维基百科使用通过进程间通信进行交流的多服务器来满足用户的请求；
* 模块化
* 私有权分离

主要的IPC方法：

* 文件
* 信号（signal）
* Berkeley套接字
* 消息队列
* 管道（pipe）
* 命名管道（fifo）
* 信号量
* 共享内存（最快）
* Message Passing，JAVA RMI等
* Memory-mapped file

### 信号量

* REF: <https://blog.csdn.net/ljianhui/article/details/10243617>
* 工作原理：信号量只能进行两种操作P和V，

P(sv)：if sv>0 ，sv-1 ; if sv=0 进程挂起；

V(sv)：如果有其他进程因等待sv挂起，让这些挂起的进程恢复执行，如果没有，sv+1;

举例说明，有两个进程P1、P2，共享信号量sv，如果有P1执行P操作，得到信号量，并可以进入临界区，使得sv-1。P2试图执行P1操作时被挂起，等待第一个进程离开临界区并执行V操作释放信号量，此时P2从挂起状态恢复并进入临界区。

信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行等待（P）和发送（V）操作。我们通常通过信号量来解决多个进程对同一资源的访问竞争问题，使在任意时刻只能有一个执行线程访问代码的临界区，也可以说他是协调进程间对同一资源的访问权，用于同步进程。

### 共享内存

* REF: <https://blog.csdn.net/ljianhui/article/details/10253345>

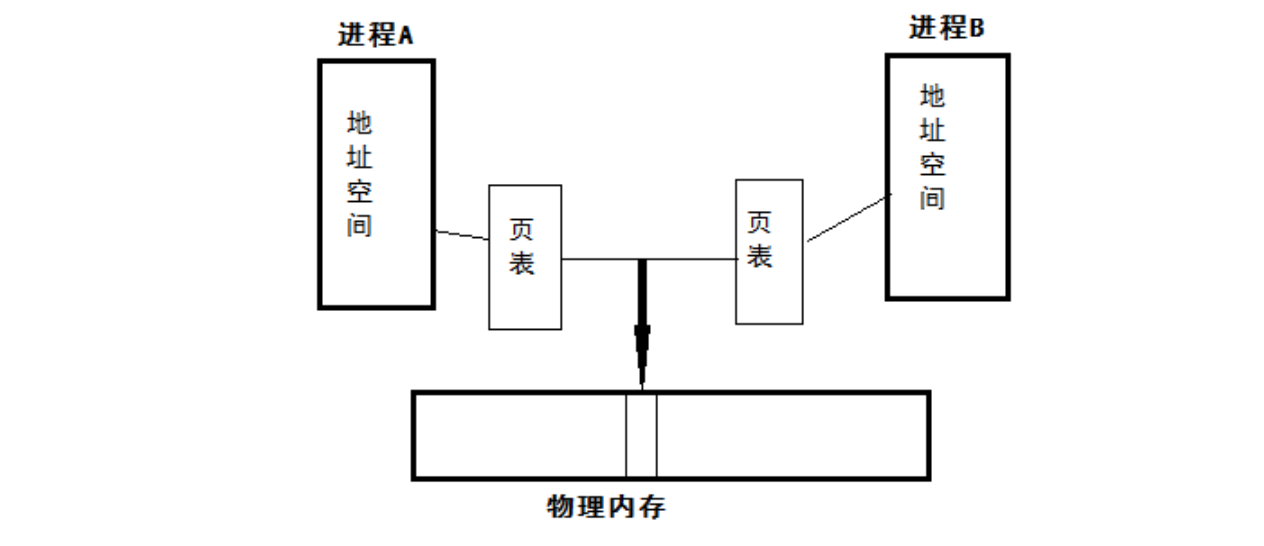
共享内存是进程通信的主要方式，也是最快的IPC形式。使用共享内存，不同进程可以对同一块内存进行读写。由于所有进程对共享内存的访问就像访问自己的内存空间一样。

**优点：**共享内存不需要进行额外系统调用或内核操作，同时还避免了多余的内存拷贝，所以，共享内存的IPC方式效率最高、速度最快的一种方式。

**缺点：缺乏同步机制，**内核不提供任何对共享内存访问的同步机制，比如同时对共享内存的相同地址进行写操作，后写的数据会覆盖之前的数据。所以使用共享内存一般还需要其他IPC机制（如信号量）进行读写同步与互斥。

共享内存原理：

Linux内存管理机制是以页为单位，一般每个page大小为4K。而程序本身的虚拟地址空间是线性的，所以内核管理了进程从虚拟地址空间到起对应的页的映射。创建共享内存空间后，内核将不同进程虚拟地址的映射到同一个页面：所以在不同进程中，对共享内存所在的内存地址的访问最终都被映射到同一页面。下图演示了共享内存的工作机制：



mmap()系统调用使得进程之间通过映射同一个普通文件实现共享内存。详细解释：<https://www.ibm.com/developerworks/cn/linux/l-ipc/part5/index1.html>

### 消息队列

* REF: <https://blog.csdn.net/ljianhui/article/details/10287879>
* 什么是消息队列

消息队列提供了一种从一个进程到另一个进程发送数据块的方法。每个数据块都被认为含有一个类型，

### 匿名管道(pipe)

* REF: <https://blog.csdn.net/ljianhui/article/details/10168031>

### 命名管道(FIFO)

* REF: <https://blog.csdn.net/ljianhui/article/details/10202699>

## 进程和线程简介

* REF:《Linux进程内核》

### 进程（Process）

进程是Unix操作系统最基本的抽象概念之一。进程是正在执行的程序，同时也是操作系统进行资源管理的最小单位，进程需要管理打开的文件、挂起的信号、内核内部数据、处理器状态等。

### 线程（Thread）

是进程中正在执行的程序片段，是操作系统进行调度的最小单位，一个线程指的是进程中一个单一顺序的控制流。Linux中线程只是一个特殊的进程，并没有对进程和线程进行专门的区分。

### 区别

进程提供了两种虚拟抽象：虚拟处理器和虚拟内存。虚拟处理器让进程觉得自己独占处理器，虚拟内存让程序觉得自己独占系统所有内存。

Linux系统中通常通过fork()创建一个新的进程，通过exec函数族来载入新的程序执行（一般在fork了一个新进程之后调用），通过exit()推出执行。

* Linux进程的实现
* 创建进程
* 进程终结 通常进程的终结发生在exit()系统调用时，进程可能显示的调用这个系统，也可能隐式地调用。当进程收到一个终结信号或者发生进程无法处理也不能忽略的异常时，集成会被动地终结。终结进程的大部分工作是在do\_exit()中完成，do\_exit()会释放进程占用的资源。包括：

1. 释放地址空间结构mm\_struct（如果没有共享，就会彻底释放该地址空间对象）
2. 将使用的文件描述符、文件系统的引用计数减1
3. 向父进程发送信号，并给当前的进程的子进程寻找新的父进程
4. 将进程描述符task\_struct的exit\_state 设为EXIT\_ZOMBIE（僵尸进程），称为僵尸进程。

* 僵尸进程与孤儿进程 如果父进程先于子进程退出，子进程会成为孤儿进程。此时必须给子进程找到新的进程作为父进程，否则当没有父进程的子进程退出时，因为没有父进程收尸，子进程会永远作为僵尸进程存在于系统当中，浪费资源。Linux内核对此的处理方法是在当前线程组中为子进程寻找一个线程作为父亲，如果不行（比如当前线程没有其他线程），就让init做父进程。Init进程会例行的调用wait()来清除僵尸子进程。
* Linux线程的实现

线程是现代操作系统的常见一个抽象概念，线程之间共享内存地址空间、打开的文件等资源、并且可以并发。

Linux的线程实现机制非常特殊，对Linux内核来说，不存在线程的概念，Linux使用标准进程来实现线程，每一个线程都有一个自己独占task\_struct，线程只是和其他进程共享地址空间等资源的进程而已。

* 创建线程： 线程创建的方式和内存是一样的，区别在于给clone()系统调用传递的参数不一样，创建线程时，clone的参数规定了哪些资源是共享的。

1. clone(CLONE\_VM | CLONE\_FS | CLONE\_FILES | CLONE\_SIGHAND, 0);

* 内核线程：内核通过内核线程在后台执行各种操作，内核线程指的是只在于内核空间的标准进程（线程），内核线程（进程）和普通线程之间的最大区别是内核线程没有独立的地址空间（task\_struct的mm域为null），他们只在内核空间运行，不会切换到用户空间。实际上内核线程会使用上一个运行的进程的地址空间，因为用户空间和内核空间是分开的，所以不会产生冲突。内核线程和普通进程一样可以被调度和抢占。只有内核线程可以创建新的内核线程，内核线程通常是在系统启动时被创建的。

## 面试题

### Linux系统日志在哪里？

### 如何查看网络进程？

### 用awk统计一个ip文件中的top10

# Zookeeper

## 面试题

### Zookeeper的实现机制，有缓存，如何存储注册服务？

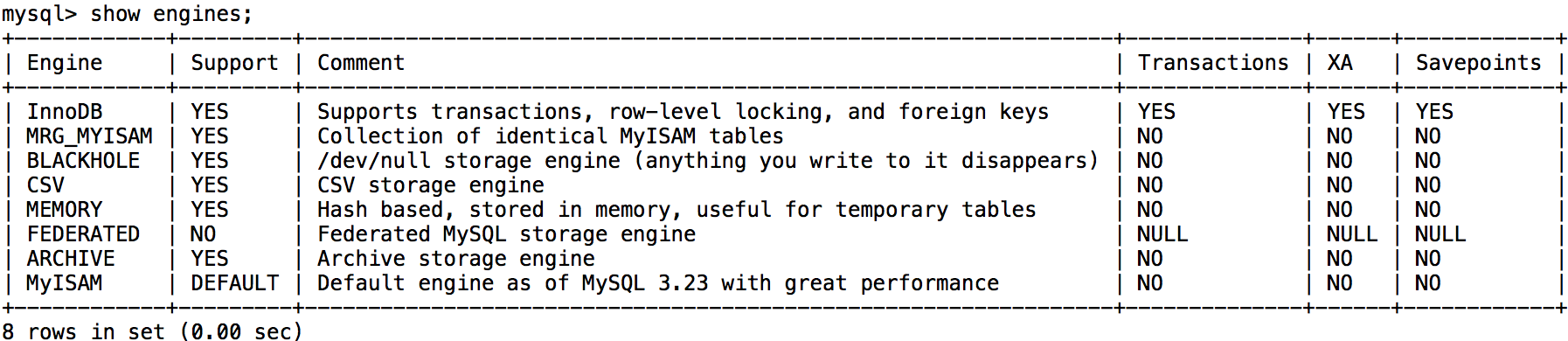
### Zookeeper中事务、节点、服务方挂了如何通知消费方？

# 数据库

举例：本地mysql数据库名称：kanglei，数据库表：user

## 查看Mysql支持的存储引擎

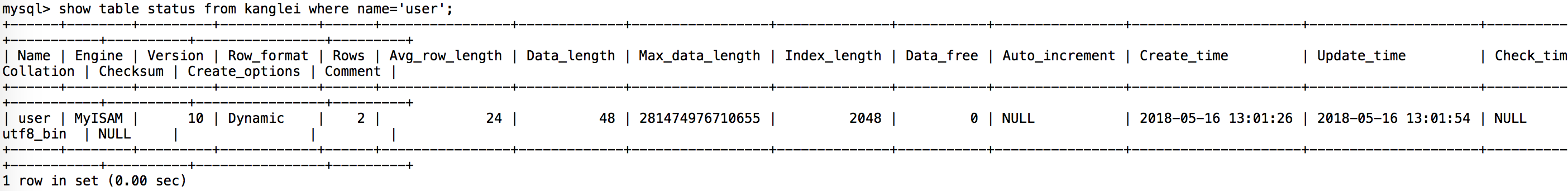
命令行：show engines;



## 查看表使用的存储引擎

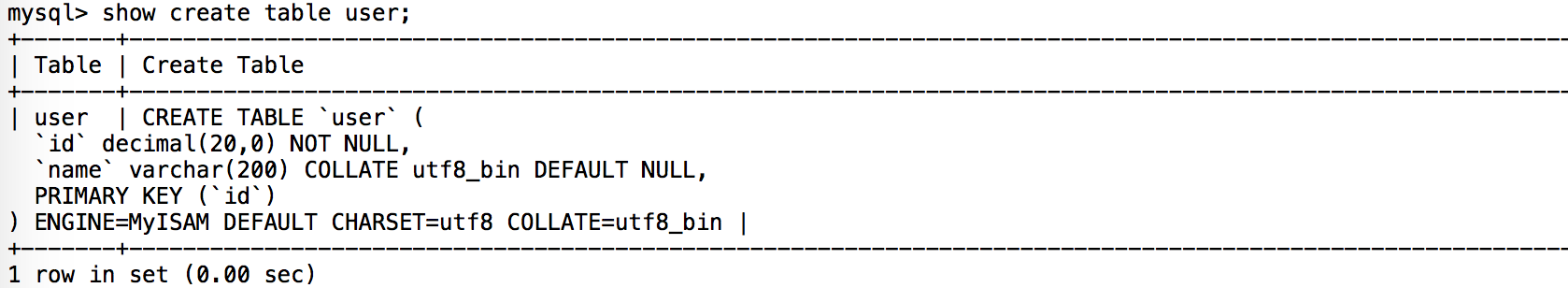
* show table status from [db\_name] where name=[table\_name]

show table status from kanglei where name=’user’;



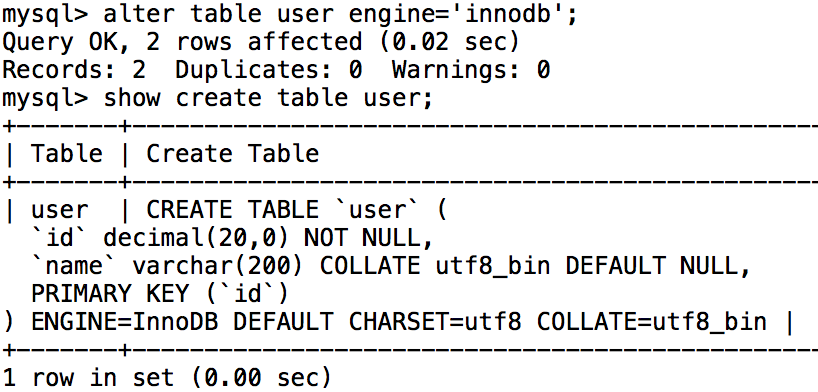
* show create table [table\_name]

show create table user;



## 修改表引擎

alter table user engine=’innodb’;



## 索引如何实现

## 分库

### 为什么分库

当单表数据量大于500w的时候，查询性能会降低，这是需要对其进行分库分表。

### 分库策略

#### 垂直拆分

根据业务模块划分不同的数据库，例如：订单/促销/会员/商品/库存/价格等等，分而治之，分摊压力。但如果订单业务剧增，仍会造成订单库数据量剧增的问题。由于数据库TPS&QPS有限，业务剧增的情况下，单一业务库迟早到达瓶颈，因而需要进一步拆拆拆！

#### 水平拆分

当单表数据量过大时，对数据库进行水平拆分。水平拆分是指将数据行按照一定的规律分布到不同的数据表中。降低单表数据量，提高查询性能。但本质上拆出的表还在同一个数据库中，其仍然受限于库级别的IO/TPS，需要分库，将拆出的表分布在分库中，以此减轻单库压力，提升总的TPS

* **常见水平切分算法：范围、哈希Hash、取模、类别、某个字段值等规则**

#### 水平+垂直

## 分表

### 分表策略

## 面试题

### 请描述基因分库法

### 外置索引提高查找效率

### Mysql的行级锁在哪个位置实现

### Mysql的性能优化方式？

### Myisam和innodb的区别，各自优点？

Innodb是行级锁，Myisam是表级别锁？

### Mongodb和Hbase的区别

### 左连接和右连接，内连接？

### 数据库的三大范式

### Mysql的binlog

### Mysql是如何实现事务

### 数据库引擎认识几种，innodb和myisam的区别，项目中使用哪种？

### 事务的隔离级别？Mysql的默认隔离级别，其他级别有哪些？

### Mysql的引擎选择，读多写少选择哪个引擎

### 假如要统计多个表，用什么引擎？

### Innodb对一行数据的读写会加锁吗？

不加锁，实际读的是副本。

### 对表做统计时可直接查看schema info信息，即：查看表的系统信息。

### SQL语句的执行顺序，select where group by order by

### Select xx from xx where xx and xx order by xx limit xx 如何优化？

使用explain

### 主从复制

### 求表的size，或者做数据统计可用什么存储引擎？

# 业务场景

## 秒杀

## 抢购

# 源码分析

## HashMap源码

# 工程

## maven

## jenkins

## git

## gradle

## 面试题

### 解释Maven的生命周期，每个生命周期具体任务是？

### Git rebase命令

# 附录

## 代码文档化：http://www.planetb.ca/syntax-highlight-word/

## 清英博客：<http://ifeve.com/concurrency-cache-cross/>

## 左耳听风：<https://www.coolshell.cn/>

## 《深入理解JVM虚拟机》

## 《Linux进程内核》