### 1. 简单说下什么是跨平台

术语：操作系统指令集、屏蔽系统之间的差异

由于各种操作系统所支持的指令集不是完全一致，所以在操作系统之上加个虚拟机可以来提供统一接口，屏蔽系统之间的差异。

### 2. Java有几种基本数据类型

有八种基本数据类型。

| **数据类型** | **字节** | **默认值** |
| --- | --- | --- |
| byte | 1 | 0 |
| short | 2 | 0 |
| int | 4 | 0 |
| long | 8 | 0 |
| float | 4 | 0.0f |
| double | 8 | 0.0d |
| char | 2 | '\u0000' |
| boolean | 4 | false |

各自占用几字节也记一下。

### 3. 面向对象特征

面向对象的编程语言有封装、继承 、抽象、多态等4个主要的特征。

1. 封装： 把描述一个对象的属性和行为的代码封装在一个模块中，也就是一个类中，属性用变量定义，行为用方法进行定义，方法可以直接访问同一个对象中的属性。
2. 抽象： 把现实生活中的对象抽象为类。分为过程抽象和数据抽象

* 数据抽象 -->鸟有翅膀,羽毛等(类的属性)
* 过程抽象 -->鸟会飞,会叫(类的方法)

1. 继承：子类继承父类的特征和行为。子类可以有父类的方法，属性（非private）。子类也可以对父类进行扩展，也可以重写父类的方法。缺点就是提高代码之间的耦合性。
2. 多态： 多态是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定(比如：向上转型，只有运行才能确定其对象属性)。方法覆盖和重载体现了多态性。

### 4. 为什么要有包装类型

术语：让基本类型也具有对象的特征

| **基本类型** | **包装器类型** |
| --- | --- |
| boolean | Boolean |
| char | Character |
| int | Integer |
| byte | Byte |
| short | Short |
| long | Long |
| float | Float |
| double | Double |

为了让基本类型也具有对象的特征，就出现了包装类型（如我们在使用集合类型Collection时就一定要使用包装类型而非基本类型）因为容器都是装object的，这是就需要这些基本类型的包装器类了。

自动装箱：new Integer(6);，底层调用:Integer.valueOf(6)

自动拆箱: int i = new Integer(6);，底层调用i.intValue();方法实现。

Integer i  = 6;  
Integer j = 6;  
System.out.println(i==j);

答案在下面这段代码中找：

public static Integer valueOf(int i) {  
    if (i >= IntegerCache.low && i <= IntegerCache.high)  
        return IntegerCache.cache[i + (-IntegerCache.low)];  
    return new Integer(i);  
}

##### 二者的区别：

1. 声明方式不同：基本类型不使用new关键字，而包装类型需要使用new关键字来在**堆中分配存储空间**；
2. 存储方式及位置不同：基本类型是直接将变量值存储在栈中，而包装类型是将对象放在堆中，然后通过引用来使用；
3. 初始值不同：基本类型的初始值如int为0，boolean为false，而包装类型的初始值为null；
4. 使用方式不同：基本类型直接赋值直接使用就好，而包装类型在集合如Collection、Map时会使用到。

### 5. ==和equals区别

* ==较的是两个引用在内存中指向的是不是同一对象（即同一内存空间），也就是说在内存空间中的存储位置是否一致。如果两个对象的引用相同时（指向同一对象时），“==”操作符返回true，否则返回flase。
* equals用来比较**某些特征**是否一样。我们平时用的String类等的equals方法都是重写后的，实现比较两个对象的内容是否相等。

我们来看看String重写的equals方法：

它不止判断了内存地址，还增加了字符串是否相同的比较。

public boolean equals(Object anObject) {  
    //判断内存地址是否相同  
    if (this == anObject) {  
        return true;  
    }  
    // 判断参数类型是否是String类型  
    if (anObject instanceof String) {  
        // 强转  
        String anotherString = (String)anObject;  
        int n = value.length;  
        // 判断两个字符串长度是否相等  
        if (n == anotherString.value.length) {  
            char v1[] = value;  
            char v2[] = anotherString.value;  
            int i = 0;  
            // 一一比较 字符是否相同  
            while (n-- != 0) {  
                if (v1[i] != v2[i])  
                    return false;  
                i++;  
            }  
            return true;  
        }  
    }  
    return false;  
}

### 6. String、StringBuffer和StringBuilder区别

java中String、StringBuffer、StringBuilder是编程中经常使用的字符串类，他们之间的区别也是经常在面试中会问到的问题。现在总结一下，看看他们的不同与相同。

##### 1. 数据可变和不可变

1. String底层使用一个不可变的字符数组private final char value[];所以它内容不可变。
2. StringBuffer和StringBuilder都继承了AbstractStringBuilder底层使用的是可变字符数组：char[] value;

##### 2. 线程安全

* StringBuilder是线程不安全的，效率较高；而StringBuffer是线程安全的，效率较低。

通过他们的append()方法来看，StringBuffer是有同步锁，而StringBuilder没有：

@Override  
public synchronized StringBuffer append(Object obj) {  
    toStringCache = null;  
    super.append(String.valueOf(obj));  
    return this;  
}

@Override  
public StringBuilder append(String str) {  
    super.append(str);  
    return this;  
}

##### 3. 相同点

StringBuilder与StringBuffer有公共父类AbstractStringBuilder。

最后，操作可变字符串速度：StringBuilder > StringBuffer > String，这个答案就显得不足为奇了。

### 7. 讲一下Java中的集合

1. Collection下：List系(有序、元素允许重复)和Set系(无序、元素不重复)

set根据equals和hashcode判断，一个对象要存储在Set中，必须重写equals和hashCode方法

1. Map下：HashMap线程不同步；TreeMap线程同步
2. Collection系列和Map系列：Map是对Collection的补充，两个没什么关系

### 8. ArrayList和LinkedList区别？

之前专门有写过ArrayList和LinkedList源码的文章。

1. ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。
2. 对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。
3. 对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

[面试必会之ArrayList源码分析](http://mp.weixin.qq.com/s?__biz=MzI4Njc5NjM1NQ==&mid=2247487595&idx=1&sn=66ddcba9714ab9c38d76db14f5c22558&chksm=ebd62f47dca1a651a2eca99cb583c02ae8d6dadf50623d27122aea7761e058d8dc5b2e5cacc6&scene=21#wechat_redirect)

[面试必会之LinkedList源码分析](http://mp.weixin.qq.com/s?__biz=MzI4Njc5NjM1NQ==&mid=2247487599&idx=1&sn=7b7b1694929079f3d30a380853b5eb8c&chksm=ebd62f43dca1a655f651eda28672df5ae05b3738eed1a4747b99146ee94c3f556c1c03ad8980&scene=21#wechat_redirect)

### 9. ConcurrentModificationException异常出现的原因

public class Test {  
    public static void main(String[] args)  {  
        ArrayList<Integer> list = new ArrayList<Integer>();  
        list.add(2);  
        Iterator<Integer> iterator = list.iterator();  
        while(iterator.hasNext()){  
            Integer integer = iterator.next();  
            if(integer==2)  
                list.remove(integer);  
        }  
    }  
}

执行上段代码是有问题的，会抛出ConcurrentModificationException异常。

**原因**：调用list.remove()方法导致modCount和expectedModCount的值不一致。

final void checkForComodification() {  
    if (modCount != expectedModCount)  
    throw new ConcurrentModificationException();  
}

**解决办法**：在迭代器中如果要删除元素的话，需要调用Iterator类的remove方法。

public class Test {  
    public static void main(String[] args)  {  
        ArrayList<Integer> list = new ArrayList<Integer>();  
        list.add(2);  
        Iterator<Integer> iterator = list.iterator();  
        while(iterator.hasNext()){  
            Integer integer = iterator.next();  
            if(integer==2)  
                iterator.remove();   //注意这个地方  
        }  
    }  
}

### 10. HashMap和HashTable、ConcurrentHashMap区别？

相同点:

1. HashMap和Hashtable都实现了Map接口
2. 都可以存储key-value数据

不同点：

1. HashMap可以把null作为key或value，HashTable不可以
2. HashMap线程不安全，效率高。HashTable线程安全，效率低。
3. HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。

什么是fail-fast?  
就是最快的时间能把错误抛出而不是让程序执行。

##### 10.2 如何保证线程安全又效率高？

Java 5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。

ConcurrentHashMap将整个Map分为N个segment(类似HashTable)，可以提供相同的线程安全，但是效率提升N倍，默认N为16。

##### 10.3 我们能否让HashMap同步？

HashMap可以通过下面的语句进行同步：  
Map m = Collections.synchronizeMap(hashMap);

### 11. 拷贝文件的工具类使用字节流还是字符流

答案：字节流

##### 11.1 什么是字节流，什么是字符流？

字节流：传递的是字节（二进制），

字符流：传递的是字符

##### 11.2 答案

我们并不支持下载的文件有没有包含字节流(图片、影像、音源)，所以考虑到通用性，我们会用字节流。

### 12. 线程创建方式

这个之前自己做过总结，也算比较全面。

#### 方法一：继承Thread类，作为线程对象存在（继承Thread对象）

public class CreatThreadDemo1 extends Thread{  
    /\*\*  
     \* 构造方法： 继承父类方法的Thread(String name)；方法  
     \* @param name  
     \*/  
    public CreatThreadDemo1(String name){  
        super(name);  
    }  
  
    @Override  
    public void run() {  
        while (!interrupted()){  
            System.out.println(getName()+'线程执行了...');  
            try {  
                Thread.sleep(200);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
  
    public static void main(String[] args) {  
        CreatThreadDemo1 d1 = new CreatThreadDemo1('first');  
        CreatThreadDemo1 d2 = new CreatThreadDemo1('second');  
  
        d1.start();  
        d2.start();  
  
        d1.interrupt();  //中断第一个线程  
    }  
}

常规方法，不多做介绍了，interrupted方法，是来判断该线程是否被中断。（终止线程不允许用stop方法，该方法不会施放占用的资源。所以我们在设计程序的时候，要按照中断线程的思维去设计，就像上面的代码一样）。

###### 让线程等待的方法

* Thread.sleep(200); //线程休息2ms
* Object.wait()； //让线程进入等待，直到调用Object的notify或者notifyAll时，线程停止休眠

#### 方法二：实现runnable接口，作为线程任务存在

public class CreatThreadDemo2 implements Runnable {  
    @Override  
    public void run() {  
        while (true){  
            System.out.println('线程执行了...');  
        }  
    }  
  
    public static void main(String[] args) {  
        //将线程任务传给线程对象  
        Thread thread = new Thread(new CreatThreadDemo2());  
        //启动线程  
        thread.start();  
    }  
}

Runnable 只是来修饰线程所执行的任务，它不是一个线程对象。想要启动Runnable对象，必须将它放到一个线程对象里。

#### 方法三：匿名内部类创建线程对象

public class CreatThreadDemo3 extends Thread{  
    public static void main(String[] args) {  
        //创建无参线程对象  
        new Thread(){  
            @Override  
            public void run() {  
                System.out.println('线程执行了...');  
            }  
        }.start();  
       //创建带线程任务的线程对象  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                System.out.println('线程执行了...');  
            }  
        }).start();  
        //创建带线程任务并且重写run方法的线程对象  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                System.out.println('runnable run 线程执行了...');  
            }  
        }){  
            @Override  
            public void run() {  
                System.out.println('override run 线程执行了...');  
            }  
        }.start();  
    }  
  
}

创建带线程任务并且重写run方法的线程对象中，为什么只运行了Thread的run方法。我们看看Thread类的源码，

，我们可以看到Thread实现了Runnable接口，而Runnable接口里有一个run方法。  
所以，我们最终调用的重写的方法应该是Thread类的run方法。而不是Runnable接口的run方法。

#### 方法四：创建带返回值的线程

public class CreatThreadDemo4 implements Callable {  
    public static void main(String[] args) throws ExecutionException, InterruptedException {  
        CreatThreadDemo4 demo4 = new CreatThreadDemo4();  
  
        FutureTask<Integer> task = new FutureTask<Integer>(demo4); //FutureTask最终实现的是runnable接口  
  
        Thread thread = new Thread(task);  
  
        thread.start();  
  
        System.out.println('我可以在这里做点别的业务逻辑...因为FutureTask是提前完成任务');  
        //拿出线程执行的返回值  
        Integer result = task.get();  
        System.out.println('线程中运算的结果为:'+result);  
    }  
  
    //重写Callable接口的call方法  
    @Override  
    public Object call() throws Exception {  
        int result = 1;  
        System.out.println('业务逻辑计算中...');  
        Thread.sleep(3000);  
        return result;  
    }  
}

Callable接口介绍：

public interface Callable<V> {  
    /\*\*  
     \* Computes a result, or throws an exception if unable to do so.  
     \*  
     \* @return computed result  
     \* @throws Exception if unable to compute a result  
     \*/  
    V call() throws Exception;  
}

返回指定泛型的call方法。然后调用FutureTask对象的get方法得道call方法的返回值。

#### 方法五：定时器Timer

public class CreatThreadDemo5 {  
  
    public static void main(String[] args) {  
        Timer timer = new Timer();  
  
        timer.schedule(new TimerTask() {  
            @Override  
            public void run() {  
                System.out.println('定时器线程执行了...');  
            }  
        },0,1000);   //延迟0，周期1s  
  
    }  
}

#### 方法六：线程池创建线程

public class CreatThreadDemo6 {  
    public static void main(String[] args) {  
        //创建一个具有10个线程的线程池  
        ExecutorService threadPool = Executors.newFixedThreadPool(10);  
        long threadpoolUseTime = System.currentTimeMillis();  
        for (int i = 0;i<10;i++){  
            threadPool.execute(new Runnable() {  
                @Override  
                public void run() {  
                    System.out.println(Thread.currentThread().getName()+'线程执行了...');  
                }  
            });  
        }  
        long threadpoolUseTime1 = System.currentTimeMillis();  
        System.out.println('多线程用时'+(threadpoolUseTime1-threadpoolUseTime));  
        //销毁线程池  
        threadPool.shutdown();  
        threadpoolUseTime = System.currentTimeMillis();  
    }  
  
}

#### 方法七：利用java8新特性 stream 实现并发

lambda表达式不懂的，可以看看我的java8新特性文章：

java8-lambda：

https://www.jianshu.com/p/3a08dc78a05f

java8-stream：

https://www.jianshu.com/p/ea16d6712a00

public class CreatThreadDemo7 {  
    public static void main(String[] args) {  
        List<Integer> values = Arrays.asList(10,20,30,40);  
        //parallel 平行的，并行的  
        int result = values.parallelStream().mapToInt(p -> p\*2).sum();  
        System.out.println(result);  
        //怎么证明它是并发处理呢  
        values.parallelStream().forEach(p-> System.out.println(p));  
    }  
}

输出：

200  
40  
10  
20  
30

怎么证明它是并发处理呢,他们并不是按照顺序输出的 。

/\*\*

\* linkedList实现map 原理: 数组+链表的方式实现

\* 实体基于Entry<K,V> 存储

\* 1. 核心知识

\* (1).两个对象进行比较时,如果hashCode()值相同,那么 他们的值不一定相同

\* (2).两个对象进行比较时,如果equals()比较为true,那么他们的hashCode()值一定相同

\* 2. 哈希(hash)算法(散列算法): 利用当前对象的hashCode()值与某个数取模a, 则值得范围一定在(0,a)之间

\* 例如: Object[] object = new Object[10];

\* 对象A的hashCode() 值为2 , 那么 int hash = 2 % object.length = 2;

\* 此时就可以把当前元素存放于object[2]的位置, 方式,即可确定元素存放在数组的下标位置,

\* 在获取元素的时候,也可以用该hash定位数组位置,查找快

\* 3. 通过hash算法即可确认元素存放的数组下标

\* 4. hash算法的冲突问题?

\* 当多个元素通过hash算法 , 都算出相同的hash值得时候

\* 例如: Object[] object = new Object[10];

\* 对象A的hashCode() 值为2 , 那么 int hash = 2 % object.length = 2;

\* 对象B的hashCode() 值为2 , 那么 int hash = 2 % object.length = 2;

\* 那么此时, 就出现了A 与B 的hash冲突,

\* 为了解决: 使用链表技术:

\* 5. 添加entry的时候解决hash相同时, 元数是在链表进行修改,还是 再链表后面继续添加元素?

\* 利用知识点1: Entry<K,V>

\* 如果key的hash相同, 则判定是在数组的哪个下标

\* 如果当前要添加元素的key的equals()方法与当前数组下标所在链表遍历下的entry的key的值相同, 则进行修改操作

\* 否则: 知识hash相同(确保了数组下标位置,找到是哪个链表),在当前链表下继续添加该entry即可

\* 6.获取元素entry的key对应的value值?

\* (1) 通过key计算hash, 获取object[hash] 对应的链表

\* (2) 遍历链表实体entry, 若entry的key的equals() 与当前要查找的key值相同, 则返回查找key对应的value值

\* @author

\*/

public class LinkedListHashMap {

private LinkedList<Entry>[] tables;

@SuppressWarnings("unchecked")

public LinkedListHashMap() {

// 初始20个大小

tables = new LinkedList[20];

}

/\*\*

\* 存放元素

\* @param key

\* @param value

\*/

public void put(Object key, Object value) {

// 创建实体

Entry newEntry = new Entry(key,value);

// 计算hash

int hash = key.hashCode() % tables.length;

// 获取 链表

LinkedList<Entry> linkedList = tables[hash];

if (linkedList == null) {

// 第一个元素,新建链表

linkedList = new LinkedList<Entry>();

linkedList.add(newEntry);

tables[hash] = linkedList;

} else {

// hash是否冲突

boolean flag = true;

for (Entry entry : linkedList) {

if (entry.key.equals(key)) {

//修改

entry.value = value;

flag = false;

break;

}

}

// key的hash相同,key的值不同,添加到链表后面

if (flag)

linkedList.add(newEntry);

}

}

/\*\*获取元素值\*/

public Object get(Object key) {

// 校验key

// key的hash

int hash = key.hashCode() % tables.length;

LinkedList<Entry> linkedList = tables[hash];

for (Entry entry : linkedList) {

if (entry.key.equals(key)) {

return entry.value;

}

}

return null;

}

class Entry {

public Entry(Object key, Object value) {

this.key = key;

this.value = value;

}

Object key;

Object value;

}

public static void main(String[] args) {

LinkedListHashMap map = new LinkedListHashMap();

map.put("a", "bb");

System.out.println(map.get("a"));

map.put("a", "cc");

System.out.println(map.get("a"));

map.put("b", "ddd");

System.out.println(map.get("b"));

}

过滤器和拦截器非常相似，但是它们有很大的区别

最简单明了的区别就是过滤器可以修改request，而拦截器不能

过滤器需要在servlet容器中实现，拦截器可以适用于javaEE，javaSE等各种环境

拦截器可以调用IOC容器中的各种依赖，而过滤器不能

过滤器只能在请求的前后使用，而拦截器可以详细到每个方法

@PathVariable主要是用来处理URl路径的问题，利用@PathVariable可以是实现在URL路径中实现参数的隐藏效果。

@RequestBody主要是用来处理请求类型转换问题，例如可以把json字符串通过配置自动转换为对象之类的。

@ResponseBody主要是用来处理返回类型转换问题，例如可以把json字符串通过配置自动转换为对象之类的。

JDK 和 JRE 有什么区别?

JRE： Java Runtime Environment  
JDK：Java Development Kit  
JRE顾名思义是java运行时环境，包含了java虚拟机，java基础类库。是使用[java语言](https://www.baidu.com/s?wd=java%E8%AF%AD%E8%A8%80&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)编写的程序运行所需要的软件环境，是提供给想运行java程序的用户使用的。  
JDK顾名思义是java开发工具包，是程序员使用[java语言](https://www.baidu.com/s?wd=java%E8%AF%AD%E8%A8%80&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)编写java程序所需的开发工具包，是提供给程序员使用的。JDK包含了JRE，同时还包含了编译java源码的编译器javac，还包含了很多java程序调试和分析的工具：jconsole，jvisualvm等工具软件，还包含了java程序编写所需的文档和demo例子程序。

如果你需要运行java程序，只需安装JRE就可以了。如果你需要编写java程序，需要安装JDK。  
JRE根据不同[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)（如：windows，linux等）和不同JRE提供商（IBM,ORACLE等）有很多版本，最常用的是Oracle公司收购SUN公司的JRE版本。

### 13. 如何将字符串反转?

使用StringBuffer或者StringBuilder中的reverse()方法

### 14. String类常用的方法有哪些？

indexOf() 返回指定字符得索引

charAt() 返回指定索引处得字符

repalce() 字符串替换

trim() 去除字符串两端的空白

split() 分割字符串 返回分割后的字符串数组

getBytes() 返回字符串的byte类型数组

length() 返回字符串的长度

toLowerCase() 字符串转小写

toUpperCase() 字符串转大写

substring() 截取字符串

equals() 字符串比较

### 15. 抽象类必须要有抽象方法吗？

答案是：不必须

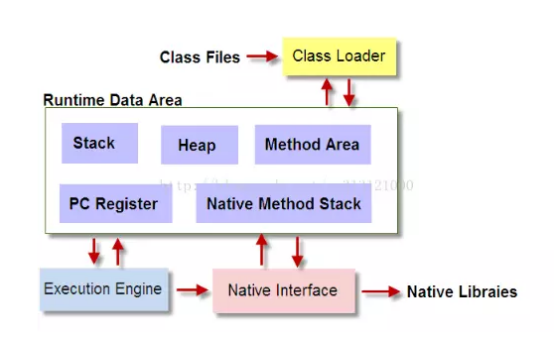
这道题考察的是抽象类的知识：

1. 抽象类必须有关键字abstract来修饰。
2. 抽象类可以不含有抽象方法
3. 如果一个类包含抽象方法，则该类必须是抽象类

### 16普通类和抽象类有哪些区别？

* 抽象类不能被实例化
* 抽象类可以有抽象方法，抽象方法只需申明，无需实现
* 含有抽象方法的类必须申明为抽象类
* 抽象的子类必须实现抽象类中所有抽象方法，否则这个子类也是抽象类
* 抽象方法不能被声明为静态
* 抽象方法不能用private修饰
* 抽象方法不能用final修饰

[jvm主要组成部分及其作用](https://www.cnblogs.com/rong0912/p/12047674.html)



1.类加载器（Class Loader）：加载类文件到内存。Class loader只管加载，只要符合文件结构就加载，至于能否运行，它不负责，那是有Exectution Engine 负责的。

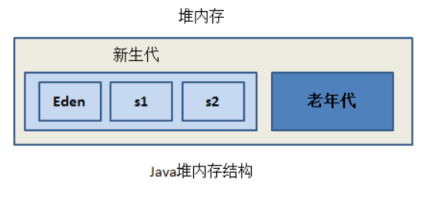
2.执行引擎（Execution Engine）：也叫解释器，负责解释命令，交由操作系统执行。

3.本地库接口（Native Interface）：本地接口的作用是融合不同的语言为java所用

4.运行时数据区（Runtime Data Area）：



（1）**堆**。堆是java对象的存储区域，任何用new字段分配的java对象实例和数组，都被分配在堆上，java堆可用-Xms和-Xmx进行内存控制，jdk1.7以后，运行时常量池从方法区移到了堆上。



　　　　　　　　新生代：老年代 = 1：2

　　　　　　　　Eden：s1: s2=8:1:1

（2）**方法区**：用于存储已被虚拟机加载的类信息，常量，静态变量，即时编译器编译后的代码等数据。

**误区：方法区不等于永生代**

　　很多人原因把方法区称作“永久代”（Permanent Generation），本质上两者并不等价，只是HotSpot虚拟机垃圾回收器团队把GC分代收集扩展到了方法区，或者说是用来永久代来实现方法区而已，这样能省去专门为方法区编写内存管理的代码，但是在Jdk8也移除了“永久代”，使用Native Memory来实现方法区。

（3）**虚拟机栈：**虚拟机栈中执行每个方法的时候，都会创建一个栈桢用于存储局部变量表，操作数栈，动态链接，方法出口等信息。

（4）**本地方法栈：**与虚拟机发挥的作用相似，相比于虚拟机栈为Java方法服务，本地方法栈为虚拟机使用的Native方法服务，执行每个本地方法的时候，都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。

（5）**程序计数器。**指示Java虚拟机下一条需要执行的字节码指令。

说一下 jvm 运行时数据区?

不同虚拟机的运行时数据区可能略微有所不同，但都会遵从 Java 虚拟机规范，Java虚拟机规范规定的区域分为以下5个部分：

·Java 虚拟机栈（Java Virtual Machine Stacks）：用于存储局部变量表、操作数栈、动态链接、方法出口等信息；

·Java 堆（Java Heap）：Java虚拟机中内存最大的一块，是被所有线程共享的，几乎所有的对象实例都在这里分配内存；

·程序计数器（Program Counter Register）：当前线程所执行的字节码的行号指示器，字节码解析器的工作是通过改变这个计数器的值，来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能，都需要依赖这个计数器来完成；

·方法区（Methed Area）：用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等数据。

·本地方法栈（Native Method Stack）：与虚拟机栈的作用是一样的，只不过虚拟机栈是服务Java方法的，而本地方法栈是为虚拟机调用Native方法服务的；

说一下堆栈的区别?

栈：存放基本类型的变量和对象的引用变量，但对象本身不存放在栈中，而是存放在堆（new 出来的对象）或者常量池中(对象可能在常量池里)（字符串常量对象存放在常量池中。

第一点：栈内存就好像一个矿泉水瓶，像里面放入东西，那么先放入的沉入底部，所以它的特点是：先进后出，后进先出

第二点：存取速度比堆要快，仅次于寄存器，栈数据可以共享，但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性

堆：存放所有new出来的对象和数组

第一点：堆其实可以类似的看做是管道，或者说是平时去排队买票的的情况差不多，所以堆内存的特点就是：先进先出，后进后出，也就是你先排队，好，你先买票。

第二点：堆可以动态地分配内存大小，生存期也不必事先告诉编译器，因为它是在运行时动态分配内存的，但缺点是，由于要在运行时动态分配内存，存取速度较慢。

区别如下：

1.堆内存用来存放由new创建的对象和数组。   
2.栈内存用来存放方法或者局部变量等   
3.堆是先进先出，后进后出   
4.栈是后进先出，先进后出

队列和栈是什么?有什么区别?

1：队列和栈是两种不同的数据结构。它们有以下区别：

（1）操作的名称不同。队列的插入称为入队，队列的删除称为出队。栈的插入称为进栈，栈的删除称为出栈。

（2）可操作的方式不同。队列是在队尾入队，队头出队，即两边都可操作。而栈的进栈和出栈都是在栈顶进行的，无法对栈底直接进行操作。

（3）操作的方法不同。队列是先进先出（FIFO），即队列的修改是依先进先出的原则进行的。新来的成员总是加入队尾（不能从中间插入），每次离开的成员总是队列头上（不允许中途离队）。而栈为后进先出（LIFO）,即每次删除（出栈）的总是当前栈中最新的元素，即最后插入（进栈）的元素，而最先插入的被放在栈的底部，要到最后才能删除。

什么是双亲委派模型? ----了解就可以了

在说双亲委派模型之前首先得先了解一下类加载阶段。

**类的加载阶段**

类加载阶段分为**加载、连接、初始化**三个阶段，而加载阶段需要通过类的全限定名来获取定义了此类的二进制字节流。

**Java特意把这一步抽出来用类加载器来实现**。把这一步骤抽离出来使得应用程序可以按需自定义类加载器。并且得益于类加载器，OSGI、热部署等领域才得以在JAVA中得到应用。

在Java中**任意一个类都是由这个类本身和加载这个类的类加载器来确定这个类在JVM中的唯一性**。也就是你用你A类加载器加载的com.aa.ClassA和你A类加载器加载的com.aa.ClassA它们是不同的，也就是用instanceof这种对比都是不同的。所以即使都来自于同一个class文件但是由不同类加载器加载的那就是两个独立的类。

类加载器除了能用来加载类，还能用来作为类的层次划分。Java自身提供了3种类加载器

1、启动类加载器(Bootstrap ClassLoader),它是属于虚拟机自身的一部分，用C++实现的，主要负责加载

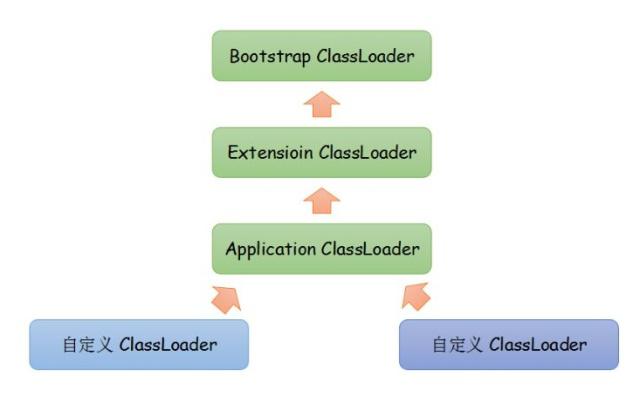
<JAVA\_HOME>\lib目录中或被-Xbootclasspath指定的路径中的并且文件名是被虚拟机识别的文件。它等于是所有类加载器的爸爸。

2、扩展类加载器(Extension ClassLoader),它是Java实现的，独立于虚拟机，主要负责加载<JAVA\_HOME>\lib\ext目录中或被java.ext.dirs系统变量所指定的路径的类库。

3、应用程序类加载器(Application ClassLoader),它是Java实现的，独立于虚拟机。主要负责加载用户类路径(classPath)上的类库，如果我们没有实现自定义的类加载器那这玩意就是我们程序中的默认加载器。

**双亲委派模型**

知道上面这几个概念就能来看看双亲委派模型了。



双亲委派的意思是如果一个类加载器需要加载类，那么首先它会把这个类请求委派给父类加载器去完成，每一层都是如此。一直递归到顶层，当父加载器无法完成这个请求时，子类才会尝试去加载。这里的双亲其实就指的是父类，没有mother。父类也不是我们平日所说的那种继承关系，只是调用逻辑是这样。



**双亲委派模型不是一种强制性约束，也就是你不这么做也不会报错怎样的，它是一种JAVA设计者推荐使用类加载器的方式**。

双亲委派有啥好处呢？它使得类有了层次的划分。就拿java.lang.Object来说，你加载它经过一层层委托最终是由Bootstrap ClassLoader来加载的，也就是最终都是由Bootstrap ClassLoader去找<JAVA\_HOME>\lib中rt.jar里面的java.lang.Object加载到JVM中。

这样如果有不法分子自己造了个java.lang.Object,里面嵌了不好的代码，如果我们是按照双亲委派模型来实现的话，最终加载到JVM中的只会是我们rt.jar里面的东西，也就是这些核心的基础类代码得到了保护。因为这个机制使得系统中只会出现一个java.lang.Object。不会乱套了。你想想如果我们JVM里面有两个Object,那岂不是天下大乱了。

因此既然推荐使用这种模型当然是有道理了。

但是人生不如意事十之八九，**有些情况不得不违反这个约束**，例如JDBC。

你先得知道SPI(Service Provider Interface)，这玩意和API不一样，它是面向拓展的，也就是我定义了这个SPI，具体如何实现由扩展者实现。我就是定了个规矩。

JDBC就是如此，在rt里面定义了这个SPI，那mysql有mysql的jdbc实现，oracle有oracle的jdbc实现，反正我java不管你内部如何实现的，反正你们都得统一按我这个来，这样我们java开发者才能容易的调用数据库操作。

所以因为这样那就不得不违反这个约束啊，Bootstrap ClassLoader就得委托子类来加载数据库厂商们提供的具体实现。因为它的手只能摸到<JAVA\_HOME>\lib中，其他的它无能为力。这就违反了自下而上的委托机制了。

Java就搞了个线程上下文类加载器，通过setContextClassLoader()默认情况就是应用程序类加载器然后Thread.current.currentThread().getContextClassLoader()获得类加载器来加载。

怎么判断对象是否可以被回收?

常见的两种判断的算法：

引用计数算法

可达性分析算法（Java使用的这一种）

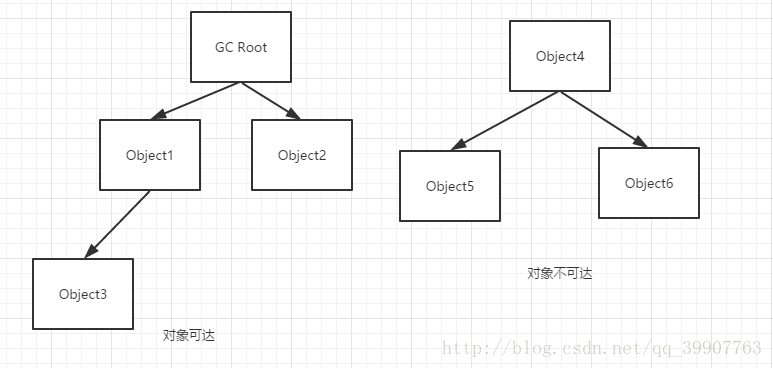
引用计数算法

引用计数算法是在对象中加入一个计数器，当对象被引用，计数器+1，当引用失效，计数器-1

这种算法实现简单，效率高，但是有一个严重的问题会导致内存泄漏，那就是对象之间循环引用，比如说A对象持有B对象的引用，B对象持有A对象的引用，那么A和B的计数器值永远>=1，也就是说这两个对象永远不会被回收

可达性分析算法

Java中定义了一些起始点，称为GC Root，当有对象引用它的时候，就把对象挂载在它下面，形成一个树状结构，当一个对象处于一个这样的树里时，就认为此对象是可达的，反之是不可达，如下图



如图 Object1,Object2,Object3是对象可达的，Object4,Object5,Object6是不可达的

那么有哪些可以作为GC Root呢？

静态属性（被static修饰的属性）

常量（被static final修饰的属性）

虚拟机栈（本地变量表）中引用的对象

本地方法栈中引用的对象

说一下类加载的执行过程?

一般主要包括两个过程

**编译**

**运行**

编译，即把我们写好的java文件，通过javac命令编译成字节码，也就是我们常说的.class文件。

运行，则是把编译声称的.class文件交给Java虚拟机(JVM)执行。

而我们所说的类加载过程即是指JVM虚拟机把.class文件中类信息加载进内存，并进行解析生成对应的class对象的过程。

举个通俗点的例子来说，JVM在执行某段代码时，遇到了class A， 然而此时内存中并没有class A的相关信息，于是JVM就会到相应的class文件中去寻找class A的类信息，并加载进内存中，这就是我们所说的类加载过程。

由此可见，JVM不是一开始就把所有的类都加载进内存中，而是只有第一次遇到某个需要运行的类时才会加载，且只加载一次。

类加载  
类加载的过程主要分为三个部分：

**加载**

**链接**

**初始化**

而链接又可以细分为三个小部分：

**验证**

**准备**

**解析**

**加载**

简单来说，加载指的是把class字节码文件从各个来源通过类加载器装载入内存中。

这里有两个重点：

字节码来源。一般的加载来源包括从本地路径下编译生成的.class文件，从jar包中的.class文件，从远程网络，以及动态代理实时编译

类加载器。一般包括启动类加载器，扩展类加载器，应用类加载器，以及用户的自定义类加载器。

注：为什么会有自定义类加载器？

一方面是由于java代码很容易被反编译，如果需要对自己的代码加密的话，可以对编译后的代码进行加密，然后再通过实现自己的自定义类加载器进行解密，最后再加载。

另一方面也有可能从非标准的来源加载代码，比如从网络来源，那就需要自己实现一个类加载器，从指定源进行加载。

**验证**  
主要是为了保证加载进来的字节流符合虚拟机规范，不会造成安全错误。

包括对于文件格式的验证，比如常量中是否有不被支持的常量？文件中是否有不规范的或者附加的其他信息？

对于元数据的验证，比如该类是否继承了被final修饰的类？类中的字段，方法是否与父类冲突？是否出现了不合理的重载？

对于字节码的验证，保证程序语义的合理性，比如要保证类型转换的合理性。

对于符号引用的验证，比如校验符号引用中通过全限定名是否能够找到对应的类？校验符号引用中的访问性（private，public等）是否可被当前类访问？

**准备**  
主要是为类变量（注意，不是实例变量）分配内存，并且赋予初值。

特别需要注意，初值，不是代码中具体写的初始化的值，而是Java虚拟机根据不同变量类型的默认初始值。

比如8种基本类型的初值，默认为0；引用类型的初值则为null；常量的初值即为代码中设置的值，final static tmp = 456， 那么该阶段tmp的初值就是456

**解析**  
将常量池内的符号引用替换为直接引用的过程。

两个重点：

符号引用。即一个字符串，但是这个字符串给出了一些能够唯一性识别一个方法，一个变量，一个类的相关信息。

直接引用。可以理解为一个内存地址，或者一个偏移量。比如类方法，类变量的直接引用是指向方法区的指针；而实例方法，实例变量的直接引用则是从实例的头指针开始算起到这个实例变量位置的偏移量

举个例子来说，现在调用方法hello()，这个方法的地址是1234567，那么hello就是符号引用，1234567就是直接引用。

在解析阶段，虚拟机会把所有的类名，方法名，字段名这些符号引用替换为具体的内存地址或偏移量，也就是直接引用。

**初始化**  
这个阶段主要是对类变量初始化，是执行类构造器的过程。

换句话说，只对static修饰的变量或语句进行初始化。

如果初始化一个类的时候，其父类尚未初始化，则优先初始化其父类。

如果同时包含多个静态变量和静态代码块，则按照自上而下的顺序依次执行。

Redis

Redis是什么？有哪些应用场景？

Redis是一个开源的 key—value型 单线程 数据库，支持string、list、set、zset和hash类型数据。

使用场景：

1.配合关系型数据库做高速缓存

* 缓存高频次访问的数据，降低数据库io
* 分布式架构，做session共享

2.可以持久化特定数据。

* 利用zset类型可以存储排行榜
* 利用list的自然时间排序存储最新n个数据

**适用场景：**

1. **数据高并发的读写**
2. **海量数据的读写**
3. **对扩展性要求高的数据**

不适场景：

1. 需要事务支持（非关系型数据库）
2. 基于sql结构化查询储存，关系复杂

MySql

一张自增表里面总共有 7 条数据，删除了最后 2 条数据，重启 mysql 数据库，又插入了一条数据，此时 id 是几?

表的类型是InnoDB，如果新增一条记录（不重启mysql的情况下），这条记录的id是8；但是如果重启（文中提到的）MySQL的话，这条记录的ID是6。因为InnoDB表只把自增主键的最大ID记录到内存中，所以重启数据库或者对表OPTIMIZE操作，都会使最大ID丢失。

但是，如果我们使用表的类型是MylSAM，那么这条记录的ID就是8。因为MylSAM表会把自增主键的最大ID记录到数据文件里面，重启MYSQL后，自增主键的最大ID也不会丢失。

注：如果在这7条记录里面删除的是中间的几个记录（比如删除的是3,4两条记录），重启MySQL数据库后，insert一条记录后，ID都是8。因为内存或者数据库文件存储都是自增主键最大ID

# Mysql char 和 varchar 的区别

char:固定长度

varchar:可变长度

1.char和varchar最大的区别就在于char不管实际value都会占用n个字符的空间，而varchar只会占用实际字符应该占用的字节空间

2.超过char和varchar的n设置后，字符串会被截断。

3.char的上限为255字节，varchar的上限65535字节，text的上限为65535字节。

4.char在存储的时候会截断尾部的空格，varchar和text不会。

# 数据库建立索引为什么为加快查询速度？

DB在执行一条SQL语句的时候，默认的方式是根据搜索条件进行全表扫描，遇到匹配条件的就加入搜索结果集合。如果对某一字段添加索引，查询时就会先去索引列表中一次定位到特定值的行数，大大减少遍历匹配的行数，索引能增加查询的速度。

    --有索引的话，首先去索引列表中查询，而索引列表是B类树的数据结构，定位到特定值的行就会非常快，所以其查询速度就很快。

表结构中字段是否添加索引判断依据是什么？

    --字段是否是查询条件或者是排序条件。适合于查询多而添删改少的表。

是否将所有的字段都添加索引，来加快查询？

    --不行的

1、  索引会占用存储空间，索引越多，使用的存储空间越多

2、  插入数据，存储索引也会消耗时间，索引越多，插入数据的速度越慢

**一般你们会在什么情况下加索引?**

（1）主键自动建立唯一索引

（2）频繁作为查询条件的字段应该创建索引

（3）查询中与其他表关联的字段，外键关系建立索引

（4）单键/组合索引的选择问题，组合索引的性价比更高

（5）查询中排序的字段，排序字段若通过索引去访问将大大提高排序速度

（6）查询中统计或者分组字段。

（7）过滤条件好的字段选择一段选择加索引

MyISAM引擎和InnoDB引擎的区别?

MyISAM：支持全文索引;不支持事务;它是表级锁;会保存表的具体行数.  
　　InnoDB：5.6以后才有全文索引;支持事务；它是行级锁;不会保存表的具体行数.

事务及其四大特性？

### （1）原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚

### （2） 一致性（Consistency）

事务开始到结束的时间段内，数据都必须保持一致状态。

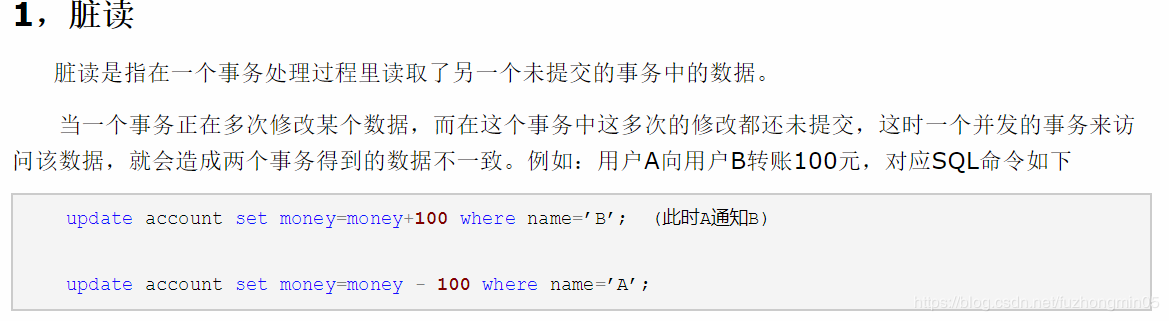
拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

### （3）隔离性（Isolation）

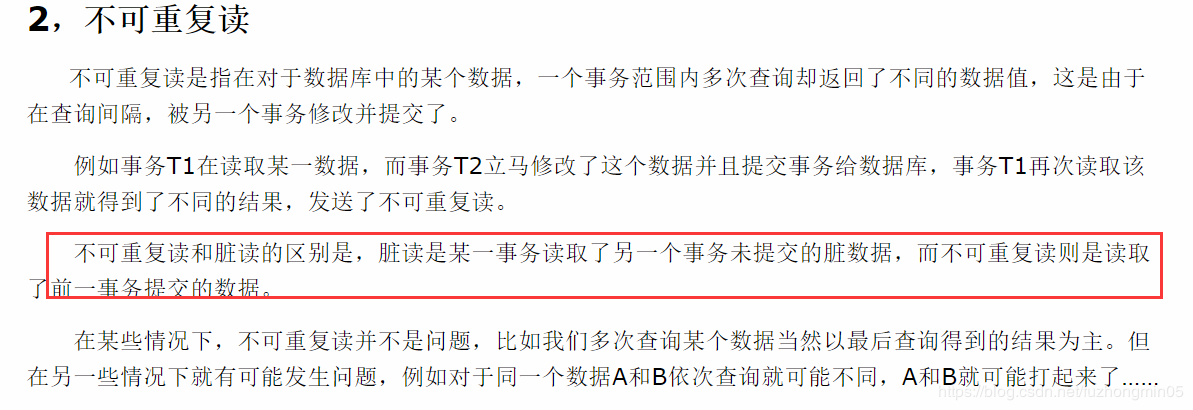
隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

### （4）持久性（Durability）

事务完成后，它对于数据的修改是永久性的，即使出现系统故障也能够保持



（事务A读取了事务B已经修改但尚未提交的数据。若事务B回滚数据，事务A的数据存在不一致性的问题）



3、幻读：一个事务在前后两次查询同一个范围的时候，后一次查询看到了前一次查询没有看到的行。如果事务中都使用快照读，那么就不会产生幻读现象，但是快照读和当前读混用就会产生幻读。关于快照读与当前读可参见[《InnoDB对MVCC的实现》](https://blog.csdn.net/fuzhongmin05/article/details/91351933)。

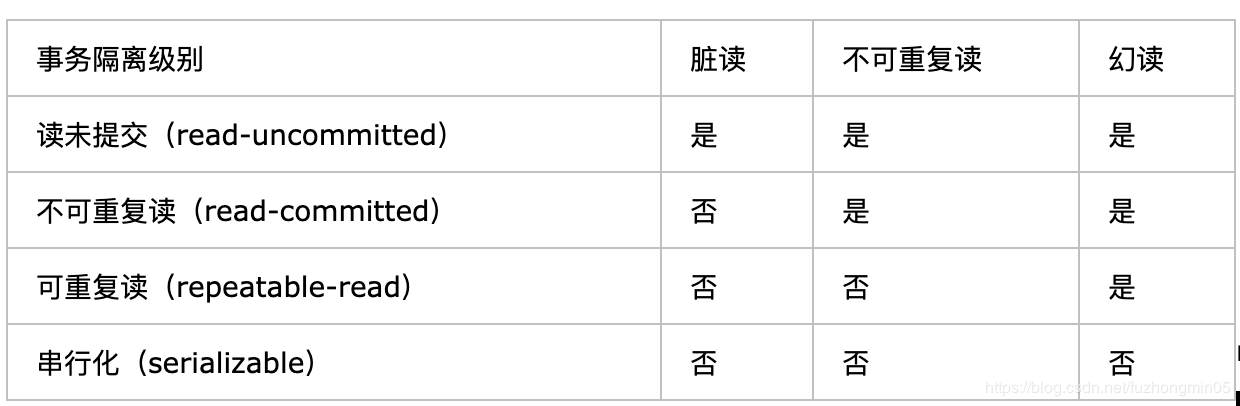
幻读和不可重复读都是读取了另一条已经提交的事务（这点就与脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

### 四种隔离级别

标准SQL规范中，定义了四个事务隔离级别：

* Read uncommitted (读未提交)：最低级别，以上问题均无法解决。
* Read committed (读已提交)：读已提交，可避免脏读情况发生。
* Repeatable Read（可重复读）：确保事务可以多次从一个字段中读取相同的值，在此事务持续期间，禁止其他事务对此字段的更新，可以避免脏读和不可重复读，仍会出现幻读问题。
* Serializable （串行化）：最严格的事务隔离级别，要求所有事务被串行执行，不能并发执行，可避免脏读、不可重复读、幻读情况的发生。

这四种隔离级别，分别有可能产生问题总结如下：



为什么一般是在I/O操作都时候，要用多线程呢(面试高频题，必背)？

"在磁盘上数据是分磁道、分簇存储的，而数据往往并不是连续排列在同一磁道上，所以磁头在读取数据时往往需要在磁道之间反复移动，因此这里就有一个寻道耗时！另外，盘面旋转将请求数据所在扇区移至读写头下方也是需要时间，这里还存在一个旋转耗时！"

"那么，在这一时间段（即"I/O等待"）内，线程是在“阻塞”着等待磁盘，此时操作系统可以将那个空闲的CPU核心用于服务其他线程。因此在I/O操作的情况下，使用多线程，效率会更高！"

redis 线程模型

Redis的线程IO模型是单线程的，使用多路线程I/O复用模型来提升处理效率。

单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU的情况。I/O多路复用，非阻塞IO，采用轮询机制依次处理就绪的流，避免了大量的无用操作。

redis 持久机制

Redis常见的持久化机制主要有两种RDB和AOF。

RDB是快照的方式，也是默认的持久化机制，就是每隔一段时间将数据集以快照的方式写入到二进制文件dump.rdb中。

AOF(Append-only file)是会记录每一步写操作，最大限度的保证了数据的安全性。文件往往比RDB的大，运行效率也偏低，但是数据一致性更好。

redis 数据类型

Redis支持丰富的数据类型，如字符串（String），哈希（Hash），列表（List），集合（Set），有序集合（ZSet）与范围查询，Bitmaps，Hyperloglogs 和地理空间（Geospatial）索引半径查询

redis 淘汰策略有哪些?(了解)

Redis主要有六种淘汰策略，分两大类：设置过期时间的数据和没有设置的。

volatile-lru：从已设置过期时间的数据集中挑选最近最少使用的数据淘汰；

volatile-ttl：从已设置过期时间的数据集中挑选将要过期的数据淘汰；

volatile-random：从已设置过期时间的数据集中任意选择数据淘汰；

allkeys-lru：从数据集中挑选最近最少使用的数据淘汰；

allkeys-random：从数据集中任意选择数据淘汰；

no-enviction（驱逐）：禁止驱逐数据；

redis常用工具类

Jedis：提供了较全面的Redis命令的支持，是Redis的Java实现客户端。

Redisson：实现了分布式和可扩展的Java数据结构，如分布式锁。

Lettuce：主要用于分布式缓存框架，线程安全同步，异步和响应，支持集群。

说一下 mysql 的行锁和表锁?

## 行锁

第一步：连接[数据库](https://www.2cto.com/database/)，这里为了方便区分命名为Transaction-A，设置autocommit为零，表示需手动提交事务。

第二步：Transaction-A，执行update修改id为1的命令。

第三步：新增一个连接，命名为Transaction-B，能正常修改id为2的数据。再执行修改id为1的数据命令时，却发现该命令一直处理阻塞等待中。

第四步：Transaction-A，执行commit命令。Transaction-B，修改id为1的命令自动执行，等待37.51秒。

总结：**多个事务操作同一行数据时，后来的事务处于阻塞等待状态。这样可以避免了脏读等数据一致性的问题。后来的事务可以操作其他行数据，解决了表锁高并发性能低的问题**。

**（InnoDB的行锁是针对索引加的锁，不是针对记录加的锁。并且该索引不能失效，否则都会从行锁升级为表锁**。）

**行锁的劣势：开销大；加锁慢；会出现死锁**

**行锁的优势：锁的粒度小，发生锁冲突的概率低；处理并发的能力强**

## 表锁

**表锁的优势：开销小；加锁快；无死锁**

**表锁的劣势：锁粒度大，发生锁冲突的概率高，并发处理能力低**

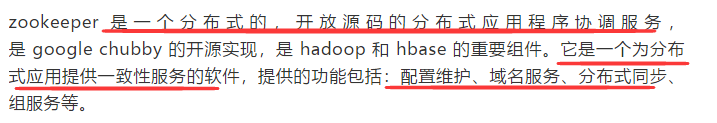
.说一下乐观锁和悲观锁?

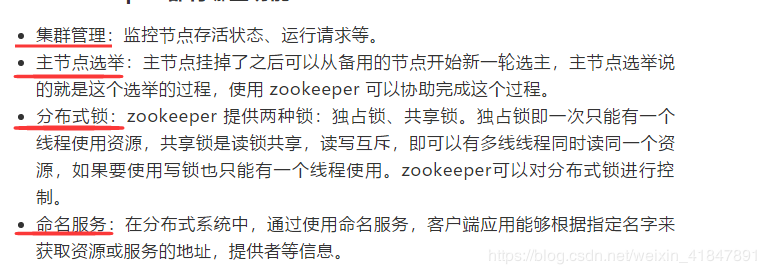
**每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，这样可以提高吞吐量**

**每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁**

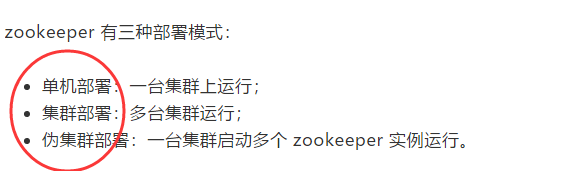
Zookeeper

zookeeper 是什么?

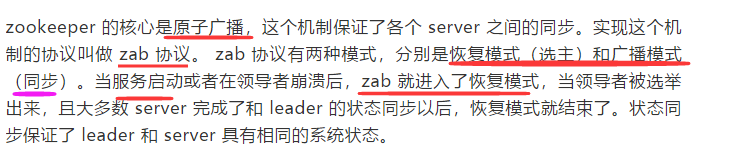
zookeeper 都有哪些功能?



## zookeeper 有几种部署模式



## zookeeper 怎么保证主从节点的状态同步



## 集群中为什么要有主节点

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，

其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，所以就需要主节点。

## 群中有 3 台服务器，其中一个节点宕机，这个时候 zookeeper 还可以使用吗？

可以继续使用，单数服务器只要没超过一半的服务器宕机就可以继续使用。

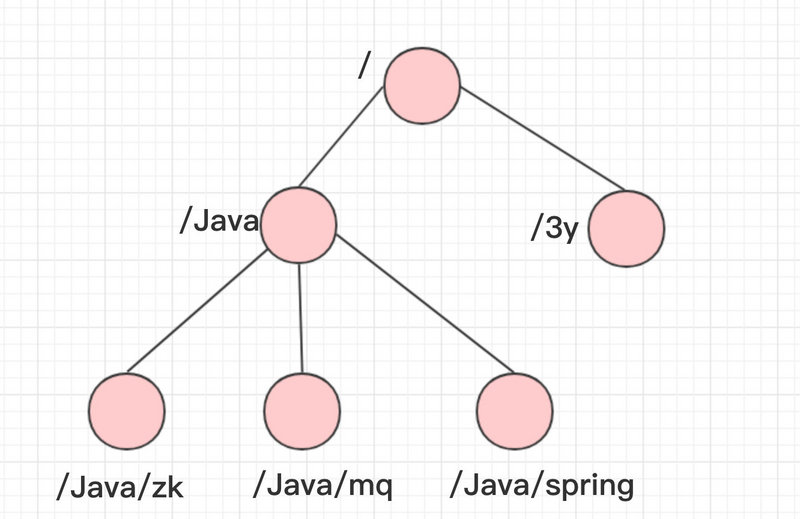
## 说一下 zookeeper 的通知机制

客户端端会对某个 znode 建立一个 watcher 事件，

当该 znode 发生变化时，这些客户端会收到 zookeeper 的通知，

然后客户端可以根据 znode 变化来做出业务上的改变。

ZooKeeper的数据结构，跟Unix文件系统非常类似，可以看做是一颗**树**，每个节点叫做**ZNode**。每一个节点可以通过**路径**来标识，结构图如下：



## ****kafka****

## kafka 可以脱离 zookeeper 单独使用吗？为什么？

kafka 不能脱离 zookeeper 单独使用，

因为 kafka 使用 zookeeper 管理和协调 kafka 的节点服务器。

## kafka 有几种数据保留的策略

kafka 有两种数据保存策略：

按照过期时间保留

按照存储的消息大小保留

## kafka 同时设置了 7 天和 10G 清除数据，到第五天的时候消息达到了 10G，这个时候 kafka 将如何处理？

这个时候 kafka 会执行数据清除工作，时间和大小不论那个满足条件，都会清空数据。

## 什么情况会导致 kafka 运行变慢

cpu 性能瓶颈

磁盘读写瓶颈

网络瓶颈

## 使用 kafka 集群需要注意什么？

集群的数量不是越多越好，最好不要超过 7 个，

因为节点越多，消息复制需要的时间就越长，整个群组的吞吐量就越低。

集群数量最好是单数，因为超过一半故障集群就不能用了，设置为单数容错率更高。

Mybatis

## mybatis 中 #{}和 ${}的区别是什么？

**#{}是预编译处理，${}是字符串替换；**

**Mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；**

**Mybatis在处理${}时，就是把sql中的${}替换成变量的值；**

**使用#{}可以有效的防止SQL注入，提高系统安全性**

## mybatis 有几种分页方式

数组分页

sql分页

拦截器分页

RowBounds分页

**SS**

## mybatis 是否支持延迟加载？延迟加载的原理是什么？

**mybatis支持延迟加载**

**适用场景**

**一对一，多对一 立即加载**

**一对多，多对多 延迟加载**

**说一下 mybatis 的一级缓存和二级缓存?**

**详细（**<https://blog.csdn.net/weixin_41847891/article/details/100623375>**）**

**一级缓存也叫sqlSession级别的缓存 ，也就是在同一个sqlSession内执行两次多次相同结果的查询语句，只会在第一次时发出sql查询数据库的数据，然后之后每次从一级缓存中查询数据返回**

### 二级缓存是mapper级别的缓存，也就是多个sqlSession之间可以实现数据的共享。

**二级缓存默认是不开启，所以在使用二级缓存时需要在做以下配置：**

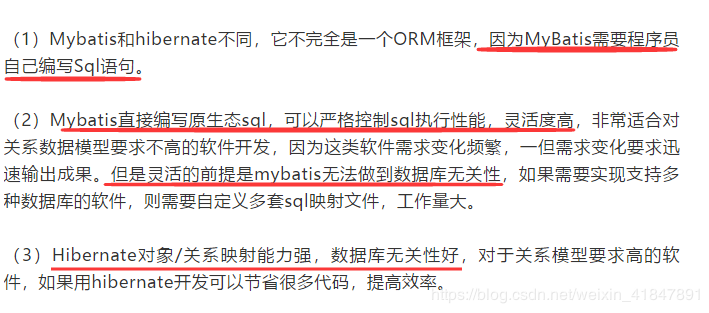
在mybatis-config.xml文件中开启二级缓存https://img-blog.csdnimg.cn/20190415202249402.png

在mapper.xml文件中开启二级缓存https://img-blog.csdnimg.cn/2019041520234484.png

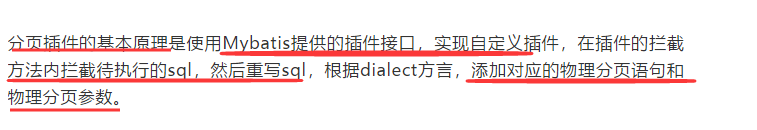
让entity实现Serializable接口https://img-blog.csdnimg.cn/20190415202510858.png

### 注意：1 当执行insert、delete、update时清除二级缓存，因为一级缓存都不存在了，何来二级缓存。2 当没有session.close()时，没有二级缓存，因为压根没有写入

**mybatis 和 hibernate 的区别有哪些?**



## mybatis 分页插件的实现原理是什么



## 抽象类和接口的区别

下面比较一下两者的语法区别：

1.抽象类可以有构造方法，接口中不能有构造方法。

2.抽象类中可以有普通成员变量，接口中没有普通成员变量

3.抽象类中可以包含非抽象的普通方法，接口中的所有方法必须都是抽象的，不能有非抽象的普通方法。

4. 抽象类中的抽象方法的访问类型可以是public，protected和（默认类型,虽然

eclipse下不报错，但应该也不行），但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型。

5. 抽象类中可以包含静态方法，接口中不能包含静态方法

6. 抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public static final类型，并且默认即为public static final类型。

7. 一个类可以实现多个接口，但只能继承一个抽象类。

下面接着再说说两者在应用上的区别：

接口更多的是在系统架构设计方法发挥作用，主要用于定义模块之间的通信契约。而抽象类在代码实现方面发挥作用，可以实现代码的重用，例如，模板方法设计模式是抽象类的一个典型应用，假设某个项目的所有Servlet类都要用相同的方式进行权限判断、记录访问日志和处理异常，那么就可以定义一个抽象的基类，让所有的Servlet都继承这个抽象基类，在抽象基类的service方法中完成权限判断、记录访问日志和处理异常的代码，在各个子类中只是完成各自的业务逻辑代码，伪代码如下：

public abstract class BaseServlet extends HttpServlet{

public final void service(HttpServletRequest request, HttpServletResponse response) throws IOExcetion,ServletException    {

                    记录访问日志

                    进行权限判断

if(具有权限){

         try{

                   doService(request,response);

}

         catch(Excetpion e)        {

                            记录异常信息

         }

}

           }

protected abstract void doService(HttpServletRequest request, HttpServletResponse response) throws IOExcetion,ServletException;

//注意访问权限定义成protected，显得既专业，又严谨，因为它是专门给子类用的

}

public class MyServlet1 extends BaseServlet

{

protected void doService(HttpServletRequest request, HttpServletResponse response) throws IOExcetion,ServletException

           {

                    本Servlet只处理的具体业务逻辑代码

           }

}

父类方法中间的某段代码不确定，留给子类干，就用模板方法设计模式。

备注：这道题的思路是先从总体解释抽象类和接口的基本概念，然后再比较两者的语法细节，最后再说两者的应用区别。比较两者语法细节区别的条理是：先从一个类中的构造方法、普通成员变量和方法（包括抽象方法），静态变量和方法，继承性等6个方面逐一去比较回答，接着从第三者继承的角度的回答，特别是最后用了一个典型的例子来展现自己深厚的技术功底。