20240326高德一面

项目

1. 用es做了哪些索引
2. 如何读取word文档的，有没有对数据做加工
3. es配置字段是怎么配置的，有没有区分文章标题和内容？比较简单，只使用了es id和文档内容
4. es是如何部署的，是不是单机
5. es是如何与mysql交互的，是如何更新的
6. es是如何使用的：采用easy-es与springboot结合，来操作es中的数据

八股

1. java抽象类和接口的区别

```
public interface Person{
  public static final int a=10;
  //JDK1.8
    default void sayHello(){
        System.out.println("Hello World");
    }
    public void say();
}
public abstract class Person{
  public abstract void say();
  public void eat(){};
}
```

相同点

（1）都不能被实例化 （2）接口的实现类或抽象类的子类都只有实现了接口或抽象类中的方法后才能实例化。(一个子类继承一个抽象类，则子类必须实现父类抽象方法，否则子类也必须定义为抽象类)

不同点

（1）接口只有定义，不能有方法的实现，java 1.8中可以定义default方法体，而抽象类可以有定义与实现，方法可在抽象类中实现。包含抽象方法的一定是抽象类，但是抽象类不一定含有抽象方法。

（2）实现接口的关键字为implements，继承抽象类的关键字为extends。一个类可以实现多个接口，但一个类只能继承一个抽象类。所以，使用接口可以间接地实现多重继承。

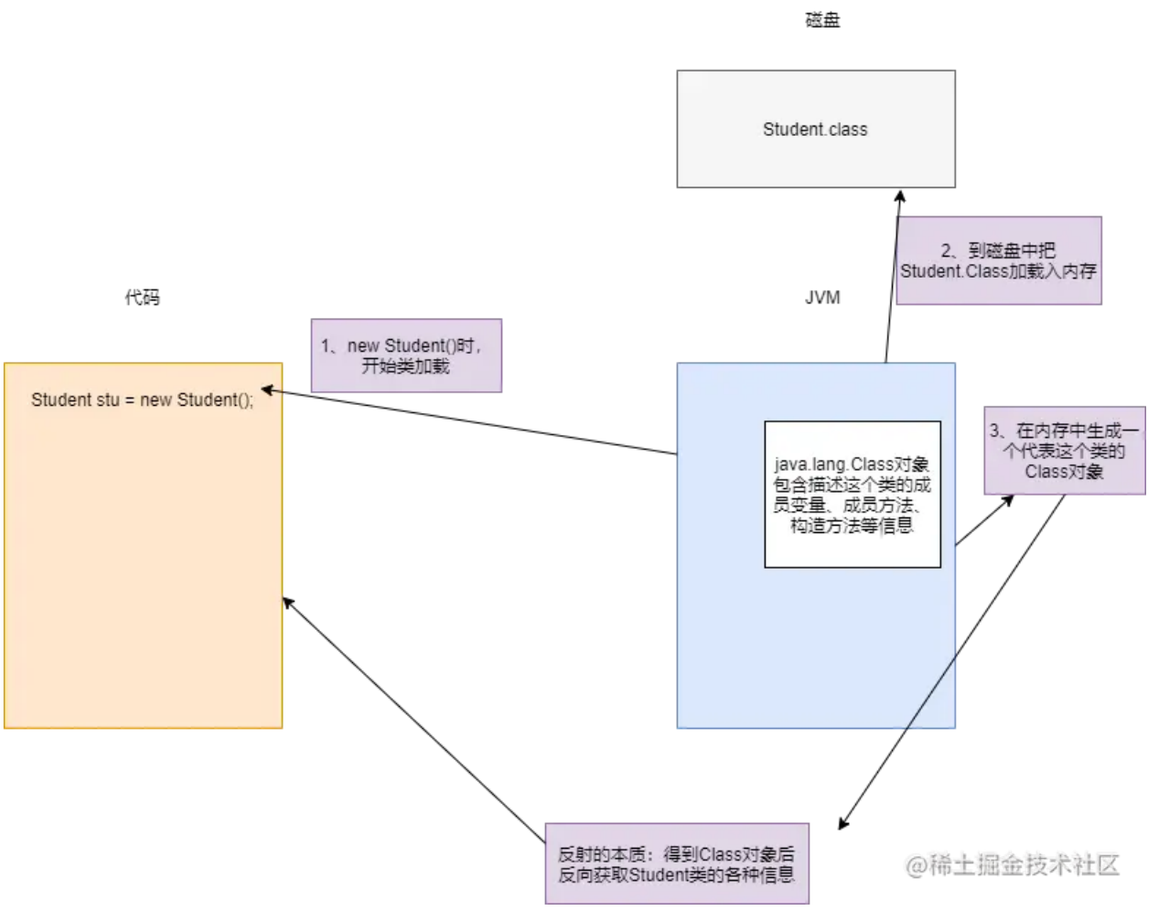
（3）接口强调特定功能的实现，而抽象类强调所属关系。一般来说，接口是某种规范的定义，抽象类是一类的功能的公共父类。 你可以简单理解抽象类是一些公共代码。接口是定义一类规范。

（4）接口成员变量默认为public static final，必须赋初值，不能被修改；其所有的成员方法都是public、abstract的。抽象类中成员变量默认default，可在子类中被重新定义，也可被重新赋值；抽象类中的抽象方法的修饰符只能为public或者protected，默认为public，不能被private、static、synchronized和native等修饰，必须以分号结尾，不带花括号。

2. java反射的作用和使用场景

https://www.cnblogs.com/zimug/p/16586952.html

2.1 原理



java执行编译的时候将java文件编译成字节码class文件,类加载器在类加载阶段将class文件加载到内存，并实例化一个java.lang.Class的对象。比如：对于Student类在加载阶段：

- 在内存(方法区或叫代码区)中实例化一个Class对象，注意是Class对象不是Student对象
- 一个Class类（字节码文件）对应一个Class对象
- 该Class对象保存了Student类的基础信息，比如这个Student类有几个字段（Filed）？有几个构造方法（Constructor）？有几个方法（Method）？有哪些注解（Annotation）？等信息。

有了上面的关于Student类的基本信息对象（java.lang.Class对象）,在运行期就可以根据这些信息来实例化Student类的对象。

- 在运行期你可以直接new一个Student对象
- 也可以使用反射的方法构造一个Student对象

但是无论你new多少个Student对象，不论你反射构建多少个Student对象，保存Student类信息的java.lang.Class对象都只有一个。

2.2 作用

java的反射机制（**在运行期动态的改变程序的调用行为的方法**）的作用：

- 在程序运行期动态的根据 `package名.类名` 实例化类对象
- 在程序运行期动态获取类对象的信息，包括对象的成本变量和方法
- 在程序运行期动态使用对象的成员变量属性

- 在程序运行期动态调用对象的方法（私有方法也可以调用）

eg：

```
package com.zimug.java.reflection;

public class Student {
    public String nickName;
    private Integer age;

    public void dinner(){
        System.out.println("吃晚餐！");
    }

    private void sleep(int minutes){
        System.out.println("睡" + minutes + "分钟");
    }
}

//获取Student类信息
Class cls = Class.forName("com.zimug.java.reflection.Student");
//对象实例化
Object obj = cls.getDeclaredConstructor().newInstance();
//根据方法名获取并执行方法
Method dinnerMethod = cls.getDeclaredMethod("dinner");
dinnerMethod.invoke(obj);  //打印：吃晚餐！
```

通过上面的代码我们看到，com.zimug.java.reflection.Student类名和dinner方法名是字符串。既然是字符串我们就可以通过配置文件，或数据库、或什么其他的灵活配置方法来执行这段程序了。这就是反射最基础的使用方式。

2.3 优缺点

- 优点：自由，使用灵活，不受类的访问权限限制。可以根据指定类名、方法名来实现方法调用，非常适合实现业务的灵活配置。
- 缺点：
  - 也正因为反射不受类的访问权限限制，其安全性低，很大部分的java安全问题都是反射导致的。
  - 相对于正常的对象的访问调用，反射因为存在类和方法的实例化过程，性能也相对较低
  - 破坏java类封装性，类的信息隐藏性和边界被破坏

2.4 应用场景

基于JDK的动态代理就是基于反射实现的。

```
public class DebugInvocationHandler implements InvocationHandler
    /**
```

```
     * 代理类中的真实对象
     */
    private final Object target;

    public DebugInvocationHandler(Object target) {
        this.target = target;
    }


    public Object invoke(Object proxy, Method method, Object[] a
        System.out.println("before method " + method.getName());
        Object result = method.invoke(target, args);
        System.out.println("after method " + method.getName());
        return result;
    }
}
```

3. java进程模型

从JVM内存区域来回答？

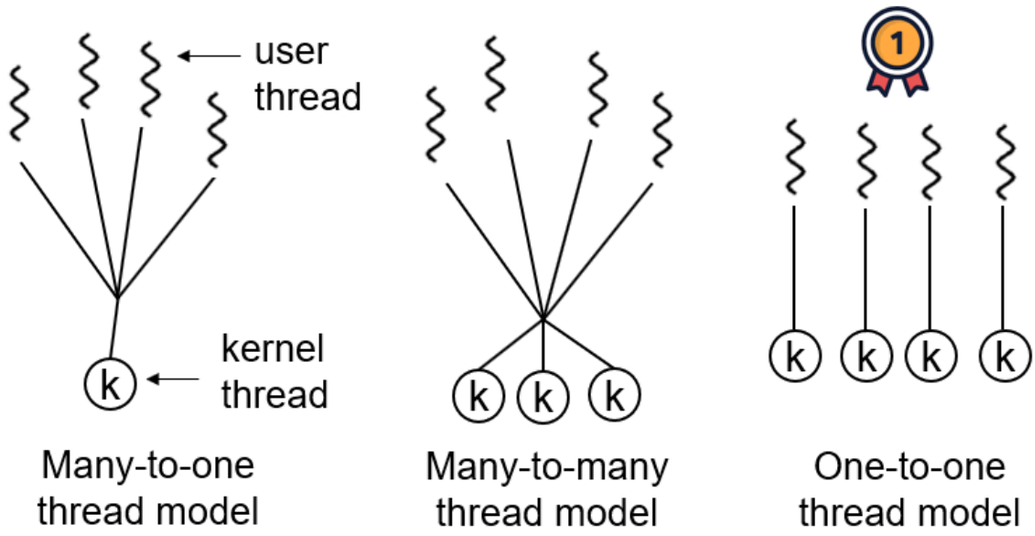
进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。在 Java 中，当我们启动 main 函数时其实就是启动了一个 JVM 的进程，而 main 函数所在的线程就是这个进程中的一个线程，也称主线程。

线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享进程的**堆**和**方法区**资源，但每个线程有自己的**程序计数器**、**虚拟机栈**和**本地方法栈**，所以系统在产生一个线程，或是在各个线程之间做切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

现在的 Java 线程的本质其实就是操作系统的线程。

线程模型是用户线程和内核线程之间的关联方式，常见的线程模型有这三种：

1. 一对一（一个用户线程对应一个内核线程）
2. 多对一（多个用户线程映射到一个内核线程）
3. 多对多（多个用户线程映射到多个内核线程）



在 Windows 和 Linux 等主流操作系统中，Java 线程采用的是一对一的线程模型，也就是一个 Java 线程对应一个系统内核线程。Solaris 系统是一个特例（Solaris 系统本身就支持多对多的线程模型），HotSpot VM 在 Solaris 上支持多对多和一对一。

4. 多线程通信和互斥的方法

多线程通信（Thread Communication）和互斥（Mutual Exclusion）是并发编程中两个非常重要的概念，它们在多线程环境下处理线程间的关系和同步问题时扮演着不同的角色。下面是它们之间的主要区别：

多线程通信

多线程通信是指线程之间为了协调工作而进行的信息交换。线程通信通常发生在以下几种情况：

1. **生产者-消费者问题**：生产者线程生成数据，消费者线程消费数据。它们需要通信以确保生产者不会在缓冲区满时添加数据，消费者不会在缓冲区空时尝试取出数据。
2. **读者-写者问题**：多个读者线程可以同时读取共享资源，但写者线程在写入共享资源时需要独占访问。读者和写者需要通信以协调对共享资源的访问。
3. **任务同步**：在一个复杂的任务中，不同的线程可能需要等待其他线程完成特定的工作才能继续执行。

线程通信的常见机制包括等待/通知（wait/notify）模式、条件同步（Condition Objects）、信号量（Semaphores）和管道（Pipes）等。

互斥

互斥是指在任何时刻，只允许一个线程访问共享资源，以防止数据竞争和确保操作的原子性。互斥的目的是保护共享资源，确保线程安全。

互斥的常见机制包括：

1. **同步方法或同步块**：在Java中，通过`synchronized`关键字来实现。
2. **锁**：如`ReentrantLock`，提供了更复杂的锁机制，允许更细粒度的控制。
3. **读写锁**：如`ReadWriteLock`，允许多个线程并发地读取，但写入时需要独占。

区别

- **目的不同**：多线程通信的目的是为了协调线程之间的工作，确保它们能够正确地协作完成任务。而互斥的目的是为了防止多个线程同时访问同一资源，以保证数据的一致性和线程的安全性。
- **实现机制不同**：多线程通信通常涉及到线程间的等待和通知，而互斥则涉及到对共享资源的加锁和解锁。
- **使用场景不同**：当需要处理生产者-消费者、读者-写者等模式时，多线程通信是必要的。而当需要保护共享资源不被多个线程同时修改时，互斥是必要的。

总的来说，多线程通信和互斥是并发编程中处理线程间关系的不同方面。它们可以单独使用，也可以结合使用，以设计出高效、可靠的多线程程序。

5. 进程间通信和互斥的方法

进程间通信（Inter-Process Communication, IPC）和互斥（Mutual Exclusion）是操作系统中用于管理并发进程的两个不同的概念。它们在多进程环境下解决不同的问题，并具有不同的目的和实现机制。

进程间通信（IPC）

进程间通信是指在不同进程之间传输数据或信号的机制。由于进程在不同的内存空间中运行，它们不能直接访问对方的变量或数据结构。IPC允许进程共享信息、同步状态或协调工作。IPC的常见方法包括：

1. **管道（Pipes）**：允许具有亲缘关系的进程（如父子进程）进行通信。
2. **命名管道（Named Pipes）**：类似于管道，但可以在不具有亲缘关系的进程之间通信。
3. **消息队列（Message Queues）**：允许进程发送和接收消息。
4. **信号（Signals）**：用于通知接收进程某个事件已经发生。
5. **共享内存（Shared Memory）**：允许多个进程访问同一块内存区域。
6. **信号量（Semaphores）**：用于控制对共享资源的访问。
7. **套接字（Sockets）**：允许不同主机上的进程进行网络通信。

互斥（Mutual Exclusion）

互斥是指在多进程环境中，确保一次只有一个进程能够访问共享资源的机制。互斥用于防止多个进程同时修改同一资源，从而避免数据不一致和竞态条件。互斥的常见实现包括：

1. **互斥锁（Mutexes）**：一种锁机制，确保同一时间只有一个进程可以进入临界区。
2. **信号量（Semaphores）**：可以用来实现互斥，通过控制对共享资源的访问权限。
3. **读写锁（Read-Write Locks）**：允许多个读进程并发访问资源，但写进程需要独占访问。

区别

- **目的不同**：IPC的目的是实现进程间的数据交换和协作，而互斥的目的是同步进程对共享资源的访问，确保数据的一致性和完整性。
- **使用场景不同**：IPC用于解决进程间需要交换信息的问题，而互斥用于解决多个进程可能同时访问同一资源的问题。

- **机制不同**：IPC机制侧重于数据的传输和接收，而互斥机制侧重于控制对共享资源的并发访问。
- **复杂性不同**：IPC可能涉及复杂的数据结构和通信协议，而互斥通常涉及锁和同步原语。

在设计并发系统时，IPC和互斥都是不可或缺的组成部分。它们可以单独使用，也可以结合使用，以确保进程间的有效通信和安全的数据访问。

6. 程序OOM如何排查

https://www.cnblogs.com/jiang-xiao-bei/p/18031815

6.1 堆内存OOM（也叫堆内存溢出）

发生在JVM试图分配对象空间时，却发现剩余的堆内存不足以存储新对象。例如我们执行下面的代码，就可以模拟出堆内存OOM的场景：

```
// 创建大量对象导致堆内存溢出
public class HeapOOM {
    static class OOMObject {
        // 假设这里有一些属性
    }

    public static void main(String[] args) {
        List<OOMObject> list = new ArrayList<>();

        while (true) {
            list.add(new OOMObject()); // 不断创建对象并添加到lis
t中
        }
    }
}
```

1. **获取Heap Dump文件**

通过分析JVM生成的堆转储（Heap Dump）文件来进行。这个过程涉及到获取堆转储文件、使用分析工具进行深入分析和解读分析结果。

在Java应用启动命令中加入以下JVM参数，以确保在发生OOM时能**自动生成**堆转储文件：

```
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/path/to/hea
pdump.hprof
```

这些参数的作用是：

- `XX:+HeapDumpOnOutOfMemoryError`：指示JVM在遇到OOM错误时生成堆转储文件。
- `XX:HeapDumpPath`：指定堆转储文件的存储路径，可以自定义路径和文件名。

或者你可以**在应用运行期间手动生成**，使用`jmap`命令生成Heap Dump文件：

```
jmap -dump:live,format=b,file=/path/to/heapdump.hprof <pid>
```

其中`<pid>`是Java进程的ID，`/path/to/heapdump.hprof`是你希望保存Heap Dump文件的位置。

2. **使用Heap Dump分析工具**

有了Heap Dump文件后，你需要使用专门的工具来进行分析。以下是一些常用的分析工具：

- **Eclipse Memory Analyzer (MAT)**：非常强大的内存分析工具，能帮助识别内存泄漏和查看内存消耗情况。
- **VisualVM**：提供了一个可视化界面，可以用来分析Heap Dump文件。
- **JVisualVM**：随JDK一起提供的工具，也支持加载Heap Dump文件进行分析。

3. **分析Heap Dump文件**

使用MAT（Eclipse Memory Analyzer）作为示例，分析流程如下：

- **打开Heap Dump文件**：启动MAT并打开Heap Dump文件（.hprof）。
- **运行Leak Suspects Report**：MAT可以自动生成一个内存泄漏报告（Leak Suspects Report），这个报告会指出可能的内存泄漏路径。
- **分析Dominators Tree**：这个视图显示了占用最多内存的对象及其引用。通过它，你可以找到最大的内存消耗者。
- **查看Histogram**：对象Histogram列出了所有对象的实例数和总大小，帮助你识别哪种类型的对象占用了最多的内存。
- **检查GC Roots**：为了确定对象为什么没有被垃圾回收，可以查看对象到GC Roots的引用链。
- **分析引用链**：通过分析对象的引用链，你可以确定是什么持有了这些对象的引用，导致它们无法被回收。

6.2 元空间（MetaSpace）OOM

元空间是方法区在`HotSpot JVM`中的实现，方法区主要用于存储类的信息、常量池、方法数据、方法代码等。方法区逻辑上属于堆的一部分，但是为了与堆进行区分，通常又叫“非堆”。

JVM 在启动后或者某个时间点开始，**MetaSpace 的已使用大小在持续增长，同时每次 GC 也无法释放，调大 MetaSpace 空间也无法彻底解决**。

元空间OOM的核心原因：生成了大量动态类

比如：

1. 使用大量动态生成类的框架（如某些ORM框架、动态代理技术、热部署工具等）
2. 程序代码中大量使用反射，反射在大量使用时，因为使用缓存的原因，会导致ClassLoader和它引用的Class等对象不能被回收

例如下面的生成大量动态代理类的代码示例，则会导致元空间的OOM

```
// 使用CGLIB动态生成大量类导致元空间溢出
public class MetaspaceOOM {
    public static void main(String[] args) {
        while (true) {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(OOMObject.class);
            enhancer.setUseCache(false);
            enhancer.setCallback(new MethodInterceptor() {
                public Object intercept(Object obj, Method me
thod, Object[] args, MethodProxy proxy) throws Throwable {
                    return proxy.invokeSuper(obj, args);
                }
            });
            enhancer.create(); // 动态生成类并加载
        }
    }

    static class OOMObject {
        // 这里可以是一些业务方法
    }
}
```

减少程序中反射的大量使用

6.3 堆外内存OOM

Java对外内存（Direct Memory）OOM指的是Java直接使用的非堆内存（off-heap memory）耗尽导致的OutOfMemoryError。这部分内存主要用于Java NIO库，允许Java程序以更接近操作系统的方式管理内存，常用于高性能缓存、大型数据处理等场景

例如下面的代码，如何堆外内存太小，就会导致堆外内存的OOM：

```
// 分配大量直接内存导致OOM
import java.nio.ByteBuffer;

public class DirectMemoryOOM {
    private static final int ONE_MB = 1024 * 1024;

    public static void main(String[] args) {
        int count = 1;

        try {
            while (true) {
                ByteBuffer byteBuffer = ByteBuffer.allocateDi
rect(ONE_MB);
                count++;
            }
        } catch (Exception e) {
```

```
                System.out.println("Exception: instance created "
+ count);
                throw e;
            }
        }
}
```

堆外内存的原因

- **分配过量的直接内存**：程序中大量使用DirectByteBuffer等直接内存分配方式，而没有相应的释放机制，导致内存迅速耗尽，常见于NIO、Netty等相关组件。
- **内存泄露**：如果分配的直接内存没有被及时释放（例如，ByteBuffer未被回收），就可能发生内存泄露。
- **JVM对外内存限制设置不当**：通过`XX:MaxDirectMemorySize`参数控制堆外内存大小，如果设置过小，可能无法满足应用需求。

堆外内存OOM的解决方案

- **合理设置对外内存大小**：根据应用的实际需求调整`XX:MaxDirectMemorySize`参数，给予足够的直接内存空间。
- **优化内存使用**：减少不必要的直接内存分配，重用DirectByteBuffer等资源。
- **内存泄露排查**：使用工具（如VisualVM、JProfiler等）定位和解决内存泄露问题。
- **代码优化**：确保使用完直接内存后显式调用`sun.misc.Cleaner.clean()`或通过其他机制释放内存。

6.4 StackOverFlow VS OutofMemory

stackoverflow，是线程运行时报的错，表示当前线程使用的栈内存已经超过最大值了。一般是是由于递归调用（一个线程内的方法递归调用），或者申请的局部变量太大，产生了超过栈内存最大值的数据。

outofmemory，是数据创建前报的错，表示当前剩余的内存已经不够了，不能创建新的数据了。

引发 StackOverFlowError 的常见原因有以下几种：

- 无限递归循环调用（最常见）。
- 执行了大量方法，导致线程栈空间耗尽。
- 方法内声明了海量的局部变量。
- native 代码有栈上分配的逻辑，并且要求的内存还不小，比如 java.net.SocketInputStream.read0 会在栈上要求分配一个 64KB 的缓存（64位 Linux）。

常见的解决方法包括以下几种：

- 修复引发无限递归调用的异常代码， 通过程序抛出的异常堆栈，找出不断重复的代码行，按图索骥，修复无限递归 Bug。
- 排查是否存在类之间的循环依赖。
- 排查是否存在在一个类中对当前类进行实例化，并作为该类的实例变量。

- 通过 JVM 启动参数 -Xss 增加线程栈内存空间， 某些正常使用场景需要执行大量方法或包含大量局部变量，这时可以适当地提高线程栈空间限制，例如通过配置 -Xss2m 将线程栈空间调整为 2 mb。

7. 机器CPU占用率高如何排查

https://juejin.cn/post/6957903936657293319#heading-24

第一步：使用top命令，然后按shift+p按照CPU排序找到占用CPU过高的进程的id

第二步：使用

```
top -H -p [进程id]
```

找到进程中消耗资源最高的线程的id

第三步：使用

```
echo 'obase=16;[线程id]' | bc或者printf "%x\n" [线程id]
```

将线程id转换为16进制（字母要小写）

bc是linux的计算器命令

第四步：执行

```
jstack [进程id] |grep -A 10 [线程id的16进制]”
```

查看线程状态信息，从堆栈中可以找到导致cpu飙升的原因

8. 海量数据求最大的10个

首先读入前10个数来创建大小为10的最小堆，然后遍历后续的数字，并于堆顶（最小）数字进行比较。如果比最小的数小，则继续读取后续数字；如果比堆顶数字大，则替换堆顶元素并重新调整堆为最小堆。整个过程直至10亿个数全部遍历完为止。然后按照中序遍历的方式输出当前堆中的所有10个数字。这个方法使用的内存是可控的，只有10个数字所需的内存即可。

9. 单链表环判断

10. Python是否支持多线程

算法

判断回文串