# 面试题重点

一般面试题会对你的项目涉及的技术进行3层的挖掘，

比如一个ajax，第一层：在项目中怎样运用ajax达到什么作用。第二层：为什么用ajax，讲一讲ajax的基础和原理。第三层：ajax是怎么实现的，比如说他做到了局部刷新，会遇到xxx问题，ajax底层又是怎么解决这些问题的？

比如线程池：第一层：项目中你会遇到xxx（线程过多）问题，那么解决xxx问题用到什么技术？第二层：为什么用线程池，怎么使用这项技术？第三层：线程池的底层如何实现，怎么确保线程安全等问题？

比如hashmap：第一层：在哪些情况下需要用到hashmap？第二层：怎么用hashmap，基于什么原理实现（看java源码）？第三层：hashmap的底层怎么实现？(看相关文献)

## 并发

线程池技术

进程线程区别

## Java基础

JAVA特性

抽象类和接口的区别

重载和覆盖、重写的区别

多继承的实现

数据类型和类的区别

### JAVA8的新特性

接口可以有默认实现

HashMap的优化（冲突链表长度大于8转化为红黑树）

Lambda表达式

### JAVA1.5新特性

静态导包(用来导入类中的静态资源)

### JAVA特性

封装、继承、多态、跨平台、没有指针、只有值传递没有引用传递

封装把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法，如果属性不想被外界访问，我们大可不必提供方法给外界访问。但是如果一个类没有提供给外界访问的方法，那么这个类也没有什么意义了。

继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承我们能够非常方便地复用以前的代码。

*关于继承如下 3 点请记住：*

*子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，只是拥有。*

*子类可以拥有自己属性和方法，即子类可以对父类进行扩展。*

*子类可以用自己的方式实现父类的方法。*

在Java中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口（实现接口并覆盖接口中同一方法）。

### JAVA引用

* 引用数据类型为java两大数据类型之一
* 引用数据型在被传值时，首先要在**栈**上给其引用**（句柄）**分配一块内存，而对象的具体信息都存储在堆内存上，然后由栈上面的引用指向堆中对象的地址。
* 引用数据类型包括：类、接口类型、数组类型、枚举类型、注解类型，字符串型；
* java另一大数据类型为基本数据类型，其包括包括数值型，字符型和布尔型。
* 基本数据类型在被创建时，在栈上给其划分一块内存，将数值直接存储在栈上；
* 就是这个数据的别名，就像一个人的外号一样，你处理这个外号，就是对个人本身的处理 相当于C里面的引用，即&，存有某个类的实例地址

### JAVA引用和C++指针的区别

**1.类型：**引用其值为地址的数据元素，java封装了的地址，可以转成字符串查看，长度可以不必关心。C++指针是一个装地址的变量,长度一般是计算机字长，可以认为是个int。

**2.所占内存：** 引用声明时没有实体，不占空间。C++指针如果声明后会用到才会赋值，如果用不到不会分配内存。

**3.类型转换：**引用的类型转换，也可能不成功，抛异，或者IDE就不能通过。C++指针只是个内存地址，指向那里，对程序来说还都是一个地址，但可能所指的地址不是程序想要的！

**4.初始值：**引用初始值为java关键字null。C++指针是INT，如不初始化指针，那他的值就不是固定的了。

**5.计算：**引用是不可以计算的。C++指针是INT，所以他可以计算，所以经常用指针来代替数组下标。

**6.控制：**引用不可以计算，所以他只能在自己程序里，可以被控制。C++指针是内存地址，也可以计算，所以他有可能指向了一个不归自己程序使用的内存地址，对于其他程序来说是很危险的，对自己程序来说也是不容易被控制的。

**7.内存泄露：**JAVA引用不会产生内存泄露。C++指针是容易产生内存泄露的，所以程序员要小心使用，及时回收。

**8.作为参数：**JAVA的方法参数只是传值，引用做为参数使用时候，回给函数内引用的值的COPY，所以在函数内交换两个引用参数是没意义的，因为 函数只交换的参数的COPY值，但在函数内改变一个引用参数的属性是有意义的，因为引用参数的COPY所引用的对象是和引用参数是同一个对象。 C++指针做为参数给函数使用，实际上就是他所指的地址在被函数操作，所以函数内用指针参数的操作都将直接作用到指针所指向的地址（变量，对象，函数，等 等）。

**9.操作符和声明：**java里用"."。C++指针用"->" 用"\*"声明指针。（似乎这条没什么用！）

**10.本质：**java中的引用和C++中的指针本质上都是想通过一个叫做引用或者指针的东西，找到要操作的目标（变量 对象等），方便在程序里操作。所不同的是JAVA的办法更安全，方便些，但没有了C++的灵活，高效。

java中的引用和C++中的指针，他们最好不要混在一起，也不要认为是一回事，只不过他们的作用有些相似，总叫人不自觉的对比。但在我们写 JAVA程序的时候最好忘记C++的指针，写C++程序的时候，也不要老惦记着JAVA的引用。只要记得 引用无大小，安全，不可计算，强转时要小心就好了。

### 在 Java 中定义一个不做事且没有参数的构造方法的作用

Java 程序在执行子类的构造方法之前，如果没有用 super() 来调用父类特定的构造方法，则会调用父类中“没有参数的构造方法”。**因此，**如果父类中只定义了有参数的构造方法，而在子类的构造方法中又没有用 super() 来调用父类中特定的构造方法，则编译时将发生错误，因为 Java 程序在父类中找不到没有参数的构造方法可供执行。解决办法是在父类里加上一个不做事且没有参数的构造方法。

### Hashcode、equals、==

hashCode（）与equals（）

* 如果两个对象相等，则hashcode一定也是相同的
* 两个对象相等,对两个equals方法返回true
* 两个对象有相同的hashcode值，它们也不一定是相等的
* 综上，equals方法被覆盖过，则hashCode方法也必须被覆盖
* hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写hashCode()，则该class的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）。

\*重写HashCode()原因：例如HashSet中如果发现有相同 hashcode 值的对象，这时会调用 equals（）方法来检查 hashcode 相等的对象是否真的相同。如果两者相同，HashSet 就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。

==与equals的区别

==是判断两个变量或实例是不是指向同一个内存空间 equals是判断两个变量或实例所指向的内存空间的值是不是相同

==是指对内存地址进行比较 equals()是对字符串的内容进行比较

==指引用是否相同 equals()指的是值是否相同

== : 它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象(基本数据类型==比较的是值，引用数据类型==比较的是内存地址)。

equals() : 它的作用也是判断两个对象是否相等。但它一般有两种使用情况：

情况1：类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时，等价于通过“==”比较这两个对象。

情况2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来比较两个对象的内容是否相等；若它们的内容相等，则返回 true (即，认为这两个对象相等)。

### String、StringBuffer和StringBuilder的区别与联系

**可变性**

String 类中使用 final 关键字字符数组保存字符串，private　final　char　value[]，所以 String 对象是不可变的。而StringBuilder 与 StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中也是使用字符数组保存字符串char[]value 但是没有用 final 关键字修饰，所以这两种对象都是可变的。

**线程安全性**

StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder 并没有对方法进行加同步锁，所以是非线程安全的。

**性能**

每次对 String 类型进行改变的时候，都会生成一个新的 String 对象，然后将指针指向新的 String 对象。StringBuffer 每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用 **StirngBuilder 相比使用 StringBuffer 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。**

**对于三者使用的总结：**

操作少量的数据 = String

单线程操作字符串缓冲区下操作大量数据 = StringBuilder

多线程操作字符串缓冲区下操作大量数据 = StringBuffer

### final 关键字的一些总结

final关键字主要用在三个地方：变量、方法、类。

1. **对于一个final变量**，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；**如果是引用类型的变量**，则在对其初始化之后便不能再让其指向另一个对象。
2. **当用final修饰一个类时**，表明这个类不能被继承。final类中的所有成员方法都会被隐式地指定为final方法。
3. 使用**final方法**的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；第二个原因是效率。在早期的Java实现版本中，会将final方法转为内嵌调用。但是如果方法过于庞大，可能看不到内嵌调用带来的任何性能提升**（现在的Java版本已经不需要使用final方法进行这些优化了）。类中所有的private方法都隐式地指定为final。**

### static 关键字

static 关键字主要有以下四种使用场景：

1. **修饰成员变量和成员方法(常用):** 被 static 修饰的成员属于类，不属于单个这个类的某个对象，被类中所有对象共享，可以并且建议通过类名调用，由于静态方法可以不通过对象进行调用，因此在静态方法里，不能调用其他非静态变量，也不可以访问非静态变量成员。被static 声明的成员变量属于静态成员变量，静态变量 存放在 Java 内存区域的方法区。调用格式：类名.静态变量名 类名.静态方法名()
2. **静态代码块:** 静态代码块定义在类中方法外, 静态代码块在非静态代码块之前执行(静态代码块—>非静态代码块—>构造方法)。 该类不管创建多少对象，静态代码块只执行一次.
3. **静态内部类（static修饰类的话只能修饰内部类）：** 静态内部类与非静态内部类之间存在一个最大的区别: 非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围类，但是静态内部类却没有。没有这个引用就意味着：1. 它的创建是不需要依赖外围类的创建。2. 它不能使用任何外围类的非static成员变量和方法。
4. **静态导包(用来导入类中的静态资源，1.5之后的新特性，应用例如Math包导入可直接调用max()函数):** 格式为：import static 这两个关键字连用可以指定导入某个类中的指定静态资源，并且不需要使用类名调用类中静态成员，可以直接使用类中静态成员变量和成员方法。

### 自动装箱与拆箱

**装箱**：将基本类型用它们对应的引用类型包装起来；

**拆箱**：将包装类型转换为基本数据类型；

### 反射机制

参考博文：https://blog.csdn.net/sinat\_38259539/article/details/71799078

* **什么是反射**

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

要想解剖一个类,必须先要获取到该类的字节码文件对象。而解剖使用的就是Class类中的方法.所以先要获取到每一个字节码文件对应的Class类型的对象.

反射就是把java类中的各种成分映射成一个个的Java对象

* **反射机制优缺点**

优点： 运行期类型的判断，动态加载类，提高代码灵活度。

缺点： 性能瓶颈：反射相当于一系列解释操作，通知 JVM 要做的事情，性能比直接的java代码要慢很多。

* **反射的应用场景**

反射是框架设计的灵魂。

在我们平时的项目开发过程中，基本上很少会直接使用到反射机制，但这不能说明反射机制没有用，实际上有很多设计、开发都与反射机制有关，例如**模块化的开发**，通过反射去调用对应的字节码；**动态代理设计模式**也采用了反射机制，还有我们日常使用的 Spring／Hibernate 等框架也大量使用到了反射机制。

**举例**：①我们在使用JDBC连接数据库时使用Class.forName()通过反射加载数据库的驱动程序；②Spring框架也用到很多反射机制，最经典的就是xml的配置模式。Spring 通过 XML 配置模式装载 Bean 的过程：1) 将程序内所有 XML 或 Properties 配置文件加载入内存中; 2)Java类里面解析xml或properties里面的内容，得到对应实体类的字节码字符串以及相关的属性信息; 3)使用反射机制，根据这个字符串获得某个类的Class实例; 4)动态配置实例的属性

* **代码层面**

**//获取字段、调用字段**

**/\***

**\* 获取成员变量并调用：**

**\***

**\* 1.批量的**

**\* 1).Field[] getFields():获取所有的"公有字段"**

**\* 2).Field[] getDeclaredFields():获取所有字段，包括：私有、受保护、默认、公有；**

**\* 2.获取单个的：**

**\* 1).public Field getField(String fieldName):获取某个"公有的"字段；**

**\* 2).public Field getDeclaredField(String fieldName):获取某个字段(可以是私有的)**

**\***

**\* 设置字段的值：**

**\* Field --> public void set(Object obj,Object value):**

**\* 参数说明：**

**\* 1.obj:要设置的字段所在的对象；**

**\* 2.value:要为字段设置的值；**

**\***

**\*/**

**public class Fields {**

**public static void main(String[] args) throws Exception {**

Class stuClass = Class.forName("fanshe.field.Student");

//获取所有公有的字段,变成了对象

Field[] fieldArray = stuClass.getFields();

for(Field f : fieldArray){ //遍历

System.out.println(f);

}

//获取所有的字段(包括私有、受保护、默认的)

fieldArray = stuClass.getDeclaredFields();

for(Field f : fieldArray){

System.out.println(f);

}

//获取公有字段\*\*并调用

Field f = stuClass.getField("name");

System.out.println(f);

//获取对象

Object obj = stuClass.getConstructor().newInstance(); //产生Student对象--》Student stu = new Student();

f.set(obj, "刘德华"); //为Student对象中的name属性赋值--》stu.name = "刘德华"

//验证

Student stu = (Student)obj;

System.out.println("验证姓名：" + stu.name);

//获取私有字段\*\*\*\*并调用

f = stuClass.getDeclaredField("phoneNum");

System.out.println(f);

f.setAccessible(true); //暴力反射，解除私有限定

f.set(obj, "18888889999");

System.out.println("验证电话：" + stu);

**}**

**}**

**由此可见**

**调用字段时：需要传递两个参数：**

**//获取成员方法并调用**

**/\***

**\* 获取成员方法并调用：**

**\***

**\* 1.批量的：**

**\* public Method[] getMethods():获取所有"公有方法"；（包含了父类的方法也包含Object类）**

**\* public Method[] getDeclaredMethods():获取所有的成员方法，包括私有的(不包括继承的)**

**\* 2.获取单个的：**

**\* public Method getMethod(String name,Class<?>... parameterTypes):**

**\* 参数：**

**\* name : 方法名；**

**\* Class ... : 形参的Class类型对象**

**\* public Method getDeclaredMethod(String name,Class<?>... parameterTypes)**

**\***

**\* 调用方法：**

**\* Method --> public Object invoke(Object obj,Object... args):**

**\* 参数说明：**

**\* obj : 要调用方法的对象；**

**\* args:调用方式时所传递的实参；**

**):**

**\*/**

**//获取构造方法并使用**

/\*

\* 通过Class对象可以获取某个类中的：构造方法、成员变量、成员方法；并访问成员；

\*

\* 1.获取构造方法：

\* 1).批量的方法：

\* public Constructor[] getConstructors()：所有"公有的"构造方法

public Constructor[] getDeclaredConstructors()：获取所有的构造方法(包括私有、受保护、默认、公有)

\* 2).获取单个的方法，并调用：

\* public Constructor getConstructor(Class... parameterTypes):获取单个的"公有的"构造方法：

\* public Constructor getDeclaredConstructor(Class... parameterTypes):获取"某个构造方法"可以是私有的，或受保护、默认、公有；

\*

\* 调用构造方法：

\* Constructor-->newInstance(Object... initargs)

\*/

**//**反射main方法

//1、获取Student对象的字节码

Class clazz = Class.forName("fanshe.main.Student");

//2、获取main方法

Method methodMain = clazz.getMethod("main", String[].class);//第一个参数：方法名称，第二个参数：方法形参的类型，

//3、调用main方法

// methodMain.invoke(null, new String[]{"a","b","c"});

//第一个参数，对象类型，因为方法是static静态的，所以为null可以，第二个参数是String数组，这里要注意在jdk1.4时是数组，jdk1.5之后是可变参数

//这里拆的时候将 new String[]{"a","b","c"} 拆成3个对象。。。所以需要将它强转。

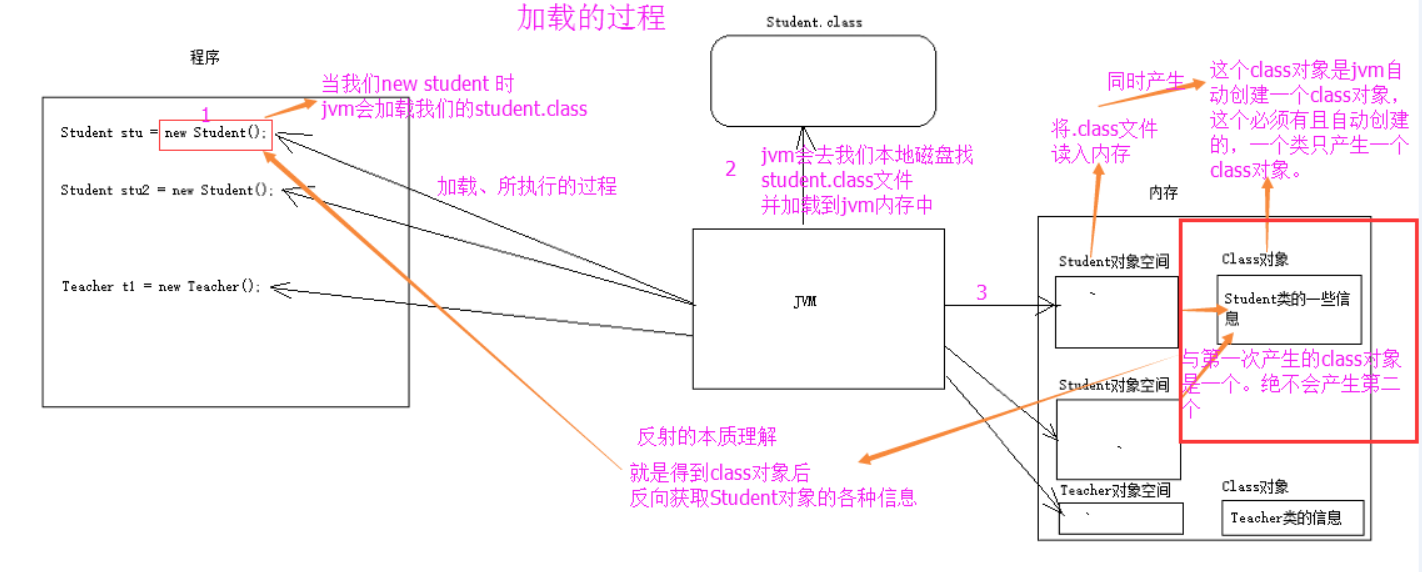
methodMain.invoke(null, (Object)new String[]{"a","b","c"});//方式一

//methodMain.invoke(null, new Object[]{new String[]{"a","b","c"}});//方式二

//反射方法的其它使用之---通过反射运行配置文件内容

//反射方法的其它使用之---通过反射越过泛型检查

### 类加载过程以及反射原理



### 什么是JDK?什么是JRE？什么是JVM？三者之间的联系与区别

JDK: 顾名思义它是给开发者提供的开发工具箱,是给程序开发者用的。它除了包括完整的JRE（Java Runtime Environment），Java运行环境，还包含了其他供开发者使用的工具包。

JRE: 普通用户而只需要安装JRE（Java Runtime Environment）来运行Java程序。而程序开发者必须安装JDK来编译、调试程序。

JVM： 当我们运行一个程序时，JVM负责将字节码转换为特定机器代码，JVM提供了内存管理/垃圾回收和安全机制等。这种独立于硬件和操作系统，正是java程序可以一次编写多处执行的原因。

**区别与联系：**

1、JDK用于开发，JRE用于运行java程序 ；

2、JDK和JRE中都包含JVM ；

3、JVM是java编程语言的核心并且具有平台独立性。

### JAVA源代码到执行的过程，字节码

Java源代码---->编译器---->jvm可执行的Java字节码(即虚拟指令)---->jvm---->jvm中解释器----->机器可执行的二进制机器码---->程序运行。

**采用字节码的好处：**

在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以Java程序运行时比较高效，而且，由于字节码并不专对一种特定的机器，因此，Java程序无须重新编译便可在多种不同的计算机上运行。

### 接口和抽象类的区别是什么？

1、接口的方法默认是public，所有方法在接口中不能有实现，抽象类可以有非抽象的方法

2、接口中的实例变量默认是final类型的，而抽象类中则不一定

3、一个类可以实现多个接口，但最多只能实现一个抽象类

**4、一个类实现接口的话要实现接口的所有方法，而抽象类不一定**

5、接口不能用new实例化，但可以声明，但是必须引用一个实现该接口的对象 从设计层面来说，抽象是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

注意：Java8 后接口可以有默认实现( default )。

### Java的public、protected、default、private



默认的类型大多是default

### 成员变量与局部变量的区别有那些？

1、**语法形式：**成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被public,private,static等修饰符所修饰，而局部变量不能被访问控制修饰符及static所修饰；但是，成员变量和局部变量都能被final所修饰；

2、**变量在内存中的存储方式：**成员变量是对象的一部分，而对象存在于堆内存，局部变量存在于栈内存

3、**变量在内存中的生存时间：**成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动消失。

4、成员变量如果没有被赋初值，则会自动以类型的默认值而赋值（一种情况例外被final修饰但没有被static修饰的成员变量必须显示地赋值）；而局部变量则不会自动赋值。

### 重载和重写的区别

**重载：** 发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同，发生在编译时。

**重写：** 发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类；如果父类方法访问修饰符为private则子类就不能重写该方法。

### 字符型常量和字符串常量的区别

1、形式上: 字符常量是单引号引起的一个字符 字符串常量是双引号引起的若干个字符

2、含义上: 字符常量相当于一个整形值(ASCII值),可以参加表达式运算 字符串常量代表一个地址值(该字符串在内存中存放位置)

3、占内存大小 字符常量只占一个字节 字符串常量占若干个字节(至少一个字符结束标志)

### List,Set,Map三者的区别？

**List(对付顺序的好帮手)：** List接口存储一组不唯一（可以有多个元素引用相同的对象），有序的对象

**Set(注重独一无二的性质):** 不允许重复的集合。不会有多个元素引用相同的对象。

**Map(用Key来搜索的专家):** 使用键值对存储。Map会维护与Key有关联的值。两个Key可以引用相同的对象，但Key不能重复，典型的Key是String类型，但也可以是任何对象。

### Arraylist 与 LinkedList 区别?（数组和链表的区别？）

**是否线程安全：**ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全。

**底层数据结构：**Arraylist 底层使用的是Object数组；LinkedList 底层使用的是双向链表数据结构（JDK1.6之前为循环链表，JDK1.7取消了循环）。

**插入和删除是否受元素位置的影响：**① ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。② LinkedList 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是近似 O（1）而数组为近似 O（n）。

**是否支持快速随机访问：**LinkedList 不支持高效的随机元素访问，而 ArrayList 支持。

**内存空间占用：**ArrayList的空 间浪费主要体现在在list列表的结尾会预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数据）。

**list 的遍历方式选择：**

实现了RandomAccess接口的list，优先选择普通for循环 ，其次foreach,

未实现RandomAccess接口的ist， 优先选择iterator遍历（foreach遍历底层也是通过iterator实现的），大size的数据，千万不要使用普通for循环

### ArrayList 与 Vector 区别呢?为什么要用Arraylist取代Vector呢？

**Vector类的所有方法都是同步的。**可以由两个线程安全地访问一个Vector对象、但是一个线程访问Vector的话代码要在同步操作上耗费大量的时间。

**Arraylist不是同步的**，所以在不需要保证线程安全时时建议使用Arraylist。

### ArrayList扩容 重点是grow()和Array.copyOf()

[通过源码一步一步分析ArrayList 扩容机制](https://github.com/Snailclimb/JavaGuide/blob/master/docs/java/collection/ArrayList-Grow.md)

以无参数构造方法创建 ArrayList 时，实际上初始化赋值的是一个空数组。当真正对数组进行添加元素操作时，才真正分配容量。即向数组中添加第一个元素时，数组容量扩为10。

Add()->ensureCapacityInternal()->ensureExplicitCapacity(minCapacity)-> grow(minCapacity); //grow()时可以扩大size，同时执行Arrays.copyOf()来实现扩容。

**详细说明：**

* 当我们要 add 进第1个元素到 ArrayList 时，elementData.length 为0 （因为还是一个空的 list），因为执行了 ensureCapacityInternal() 方法 ，所以 minCapacity 此时为10。此时，minCapacity - elementData.length > 0 成立，所以会进入 grow(minCapacity) 方法。
* 当add第2个元素时，minCapacity 为2，此时e lementData.length(容量)在添加第一个元素后扩容成 10 了。此时，minCapacity - elementData.length > 0 不成立，所以不会进入 （执行）grow(minCapacity) 方法。
* 添加第3、4···到第10个元素时，依然不会执行grow方法，数组容量都为10。直到添加第11个元素，minCapacity(为11)比elementData.length（为10）要大。进入grow方法进行扩容。

int newCapacity = oldCapacity + (oldCapacity >> 1),所以 ArrayList 每次扩容之后容量都会变为原来的 1.5 倍！（JDK1.6版本以后） JDk1.6版本时，扩容之后容量为 1.5 倍+1！

当新容量大于 MAX\_ARRAY\_SIZE,进入(执行) hugeCapacity() 方法来比较 minCapacity 和 MAX\_ARRAY\_SIZE，如果minCapacity大于最大容量，则新容量则为Integer.MAX\_VALUE，否则，新容量大小则为 MAX\_ARRAY\_SIZE 即为 Integer.MAX\_VALUE – 8

**两个方法：System.arraycopy() 和 Arrays.copyOf()**

System.arraycopy()实现自己拷贝自己，Arrays.copyOf()实现把A拷贝到B并扩容。

System.arraycopy(elementData, index, elementData, index + 1, size - index); //elementData:源数组;index:源数组中的起始位置;elementData：目标数组；index + 1：目标数组中的起始位置； size - index：要复制的数组元素的数量。

Arrays.copyOf(elementData, size); //elementData：要复制的数组；size：要复制的长度

联系和区别：

联系：copyOf() 内部实际调用了 System.arraycopy() 方法

区别：arraycopy() 需要目标数组，将原数组拷贝到你自己定义的数组里或者原数组，而且可以选择拷贝的起点和长度以及放入新数组中的位置 copyOf() 是系统自动在内部新建一个数组，并返回该数组。

#### 优化：

**ensureCapacity方法**

**在 add 大量元素之前用 ensureCapacity 方法，以减少增量重新分配的次数**

final int N = 10000000;

list.ensureCapacity(N);

for (int i = 0; i < N; i++) {

list.add(i);

}

### Java中的length、length()、size()

* java 中的 length 属性是针对数组说的,比如说你声明了一个数组,想知道这个数组的长度则用到了 length 这个属性.
* java 中的 length() 方法是针对字符串说的,如果想看这个字符串的长度则用到 length() 这个方法.
* java 中的 size() 方法是针对泛型集合说的,如果想看这个泛型有多少个元素,就调用此方法来查看!

### HashMap 和 Hashtable 的区别

**线程是否安全：** HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；

**效率：** 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；（说起来是不是HashMap也被TreeMap淘汰了？）

**对Null key 和Null value的支持：** HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛出 NullPointerException。

**初始容量大小和每次扩充容量大小的不同 ：** ①创建时如果不指定容量初始值，Hashtable 默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap 默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为2的幂次方大小（HashMap 中的tableSizeFor()方法保证）。也就是说 HashMap 总是使用2的幂作为哈希表的大小, (位操作更快：hash值太大，要先取余缩小范围。取余(%)操作中如果除数是2的幂次则等价于与其除数减一的与(&)操作（**也就是说 hash%length==hash&(length-1)的前提是 length 是2的 n 次方；**）。” 并且 采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是2的幂次方。)

**底层数据结构：** JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

### HashMap 和 HashSet区别

HashSet 底层就是基于 HashMap 实现的。（HashSet 的源码非常非常少，因为除了 clone() 、writeObject()、readObject()是 HashSet 自己不得不实现之外，其他方法都是直接调用 HashMap 中的方法。



### Hash使用红黑树底层

TreeMap、TreeSet以及JDK1.8之后的HashMap底层都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

### 为何用红黑树而不是平衡二叉树

### 并发环境推荐用ConcurrentHashMap取代HashMap

主要原因在于 并发下的Rehash 会造成元素之间会形成一个循环链表。不过，jdk 1.8 后解决了这个问题，但是还是不建议在多线程下使用 HashMap,因为多线程下使用 HashMap 还是会存在其他问题比如数据丢失。

**ConcurrentHashMap 和 Hashtable 的区别**

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。

**底层数据结构：**

JDK1.7的 ConcurrentHashMap 底层采用 分段的**数组+链表** 实现，JDK1.8 采用的数据结构跟HashMap1.8的结构一样，**数组+链表/红黑二叉树**。

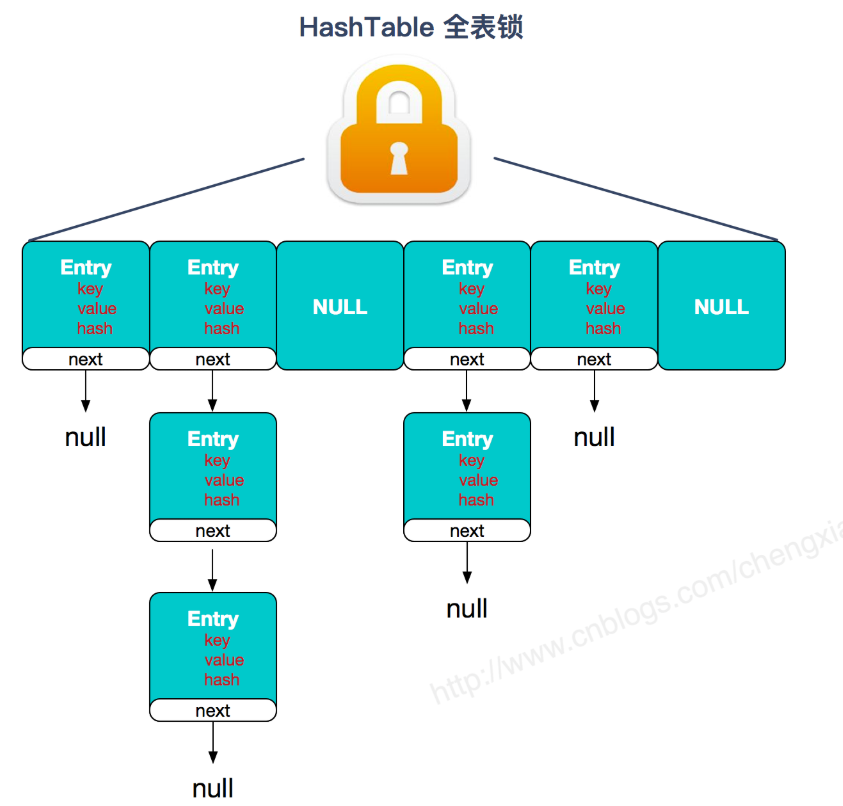
Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 **数组+链表** 的形式

**实现线程安全的方式（重要）：**

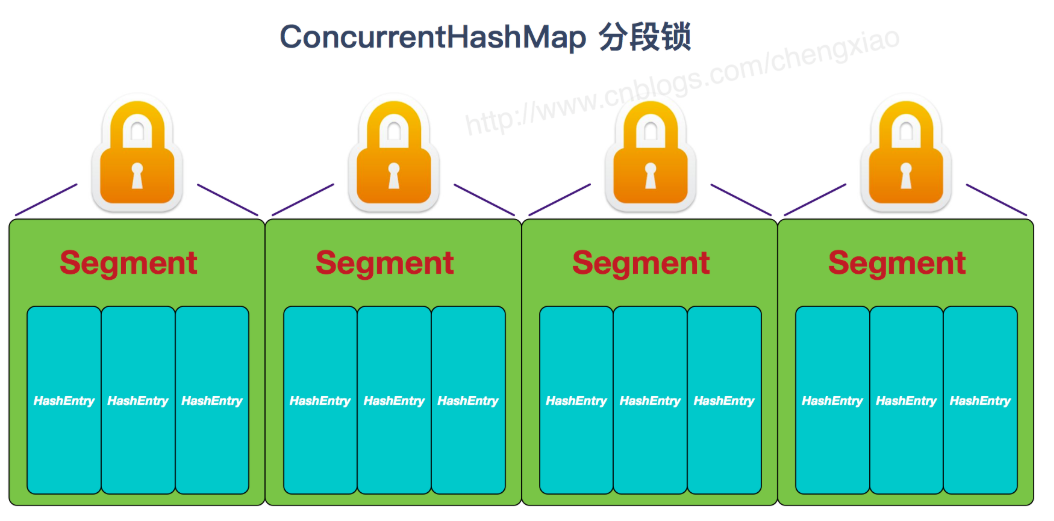
① 在JDK1.7的时候，ConcurrentHashMap（分段锁） **对整个桶数组进行了分割分段(Segment)，**每一把锁只锁容器其中一部分数据，**多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。** 到了 JDK1.8 的时候已经摒弃了Segment的概念，而是**直接用 Node 数组+链表+红黑树**的数据结构来实现，**并发控制使用 synchronized 和 CAS 来操作**。（JDK1.6以后 对 synchronized锁做了很多优化） **整个看起来就像是优化过且线程安全的 HashMap**，虽然在JDK1.8中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；

② Hashtable(同一把锁) :**使用 synchronized 来保证线程安全，效率非常低下**。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

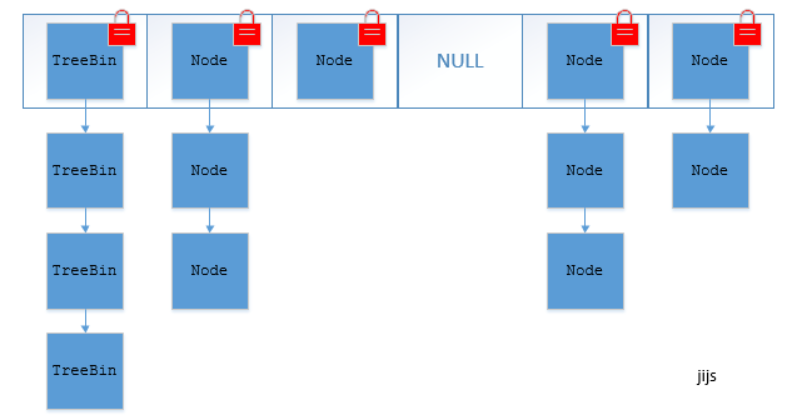
图片对比：



Jdk1.7:



Jdk1.8（TreeBin: 红黑二叉树节点 Node: 链表节点）:



### ConcurrentHashMap线程安全的具体实现方式/底层具体实现

JDK1.7（分段锁）：

ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。

Segment 实现了 ReentrantLock,所以 Segment 是一种可重入锁，扮演锁的角色。HashEntry 用于存储键值对数据。

一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和HashMap类似，是一种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，每个 Segment 守护着一个HashEntry数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment的锁。

JDK1.8（CAS和synchronized）：

ConcurrentHashMap取消了Segment分段锁，采用CAS和synchronized来保证并发安全。数据结构跟HashMap1.8的结构类似，数组+链表/红黑二叉树。Java 8在链表长度超过一定阈值（8）时将链表（寻址时间复杂度为O(N)）转换为红黑树（寻址时间复杂度为O(long(N))）

synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲突，就不会产生并发，效率又提升N倍。

**jdk1.8对ConcurrentHashMap做了哪些优化？为什么这样优化？**

插入的时候如果数组元素使用了红黑树，取消了分段锁设计，synchronize替代了Lock锁；避免冲突严重时链表多长，提高查询效率，时间复杂度从O(N)提高到O(logN)

### Volatile作用：保证内存可见性；底层原理

底层原理：主内存和工作内存，读写内存屏障，happen-before

**多核CPU下，可见性怎么保证？**总线嗅探技术

### 集合框架底层数据结构总结

1. List

**Arraylist：** Object数组

**Vector：** Object数组

**LinkedList：** 双向链表(JDK1.6之前为循环链表，JDK1.7取消了循环) 详细可阅读JDK1.7-LinkedList循环链表优化

2. Set

**HashSet（无序，唯一）:** 基于 HashMap 实现的，底层采用 HashMap 来保存元素

**LinkedHashSet：** LinkedHashSet 继承与 HashSet，并且其内部是通过 LinkedHashMap 来实现的。有点类似于我们之前说的LinkedHashMap 其内部是基于 Hashmap 实现一样，不过还是有一点点区别的。

**TreeSet（有序，唯一）：** 红黑树(自平衡的排序二叉树。)

3. Map

**HashMap：** JDK1.8之前HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）。JDK1.8以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间

**LinkedHashMap：** LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外，LinkedHashMap 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。详细可以查看：《LinkedHashMap 源码详细分析（JDK1.8）》

**Hashtable：** 数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的

**TreeMap：** 红黑树（自平衡的排序二叉树）

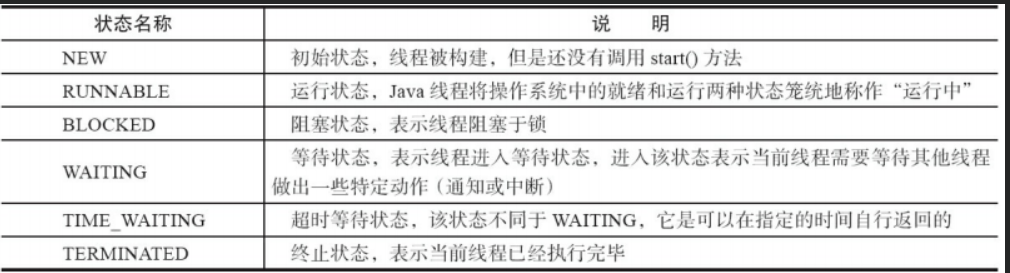
**集合的选用**

主要根据集合的特点来选用

比如我们需要根据键值获取到元素值时就选用Map接口下的集合，需要排序时选择TreeMap,不需要排序时就选择HashMap,需要保证线程安全就选用ConcurrentHashMap.

当我们只需要存放元素值时，就选择实现Collection接口的集合，需要保证元素唯一时选择实现Set接口的集合比如TreeSet或HashSet，不需要就选择实现List接口的比如ArrayList或LinkedList，然后再根据实现这些接口的集合的特点来选用。

### 线程的生命周期



线程创建之后它将处于 NEW（新建） 状态，调用 start() 方法后开始运行，线程这时候处于 READY（可运行） 状态。可运行状态的线程获得了 CPU 时间片（timeslice）后就处于 RUNNING（运行） 状态。

当线程执行 wait()方法之后，线程进入 \*\*WAITING（等待）\*\*状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 TIME\_WAITING(超时等待) 状态相当于在等待状态的基础上增加了超时限制，比如通过 sleep（long millis）方法或 wait（long millis）方法可以将 Java 线程置于 TIMED WAITING 状态。当超时时间到达后 Java 线程将会返回到 RUNNABLE 状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到 BLOCKED（阻塞） 状态。线程在执行 Runnable 的run()方法之后将会进入到 TERMINATED（终止） 状态。

### 实现多线程的3种方式

**1）继承Thread类**

MyThread.java

public class MyThread extends Thread {

@Override

public void run() {

super.run();

System.out.println("MyThread");

}

}

Run.java

public class Run {

public static void main(String[] args) {

MyThread mythread = new MyThread();

mythread.start();

System.out.println("运行结束");

}

}

**2）实现Runnable接口**

推荐实现Runnable接口方式开发多线程，因为Java单继承但是可以实现多个接口。

MyRunnable.java

public class MyRunnable implements Runnable {

@Override

public void run() {

System.out.println("MyRunnable");

}

}

Run.java

public class Run {

public static void main(String[] args) {

Runnable runnable=new MyRunnable();

Thread thread=new Thread(runnable);

thread.start();

System.out.println("运行结束！");

}

}

**3）继承Callable接口**

用于获取线程执行返回值

第一步：创建Callable子类的实例化对象

第二步：创建FutureTask对象，并将Callable对象传入FutureTask的构造方法中

（注意：FutureTask实现了Runnable接口和Future接口）

第三步：实例化Thread对象，并在构造方法中传入FurureTask对象

第四步：启动线程

public class Main {

public static void main(String[] args) {

//执行方法1

ExecutorService pool = Executors.newCachedThreadPool();

Future future = pool.submit(new TestCallAble());

try {

System.out.println("result = " + future.get());

} catch (Exception e) {

e.printStackTrace();

}

//执行方法2

FutureTask<String> task = new FutureTask<String>(new TestCallAble());

Thread thread = new Thread(task);

thread.start();

try {

System.out.println("result = " + task.get());

} catch (Exception e) {

e.printStackTrace();

}

}

private static class TestCallAble implements Callable {

@Override

public String call() throws Exception {

Thread.sleep(3000);

return "finish";

}

}

}

**4）使用线程池**

**在《阿里巴巴Java开发手册》“并发处理”这一章节，明确指出线程资源必须通过线程池提供，不允许在应用中自行显示创建线程：**

使用线程池的好处是减少在创建和销毁线程上所消耗的时间以及系统资源开销，解决资源不足的问题。如果不使用线程池，有可能会造成系统创建大量同类线程而导致消耗完内存或者“过度切换”的问题。

**另外《阿里巴巴Java开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险**

Executors 返回线程池对象的弊端如下：

FixedThreadPool 和 SingleThreadExecutor ： 允许请求的队列长度为 Integer.MAX\_VALUE,可能堆积大量的请求，从而导致OOM。

CachedThreadPool 和 ScheduledThreadPool ： 允许创建的线程数量为 Integer.MAX\_VALUE ，可能会创建大量线程，从而导致OOM。

### sleep()和wait()的区别

* 两者最主要的区别在于：**sleep方法没有释放锁，而wait方法释放了锁** 。
* 两者都可以暂停线程的执行。
* Wait通常被用于线程间交互/通信，sleep通常被用于暂停执行。
* wait()方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的notify()或者notifyAll()方法。sleep()方法执行完成后，线程会自动苏醒。

### 为什么我们调用start()方法时会执行run()方法，为什么我们不能直接调用run()方法？

new一个Thread，线程进入了新建状态;调用start()方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。 start()会执行线程的相应准备工作，然后自动执行run()方法的内容，这是真正的多线程工作。 而直接执行run()方法，会把run方法当成一个main线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。

总结： 调用start方法方可启动线程并使线程进入就绪状态，而run方法只是thread的一个普通方法调用，还是在主线程里执行。

### Exception和Error的区别：异常能被程序本身可以处理，错误是无法处理。

**Error（错误）:是程序无法处理的错误**，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。**例如**，Java虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。

这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如Java虚拟机运行错误（Virtual MachineError）、类定义错误（NoClassDefFoundError）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之 外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。

**Exception（异常）:是程序本身可以处理的异常。**Exception 类有一个重要的子类 RuntimeException。RuntimeException 异常由Java虚拟机抛出。NullPointerException（要访问的变量没有引用任何对象时，抛出该异常）、ArithmeticException（算术运算异常，一个整数除以0时，抛出该异常）和 ArrayIndexOutOfBoundsException （下标越界异常）。

在以下4种特殊情况下，finally块不会被执行：

1、在finally语句块第一行发生了异常。 因为在其他行，finally块还是会得到执行

2、在前面的代码中用了System.exit(int)已退出程序。 exit是带参函数 ；若该语句在异常语句之后，finally会执行

3、程序所在的线程死亡。

4、关闭CPU。

### 一个类的构造方法的作用和特性

主要作用是完成对类对象的初始化工作。因为一个类即使没有声明构造方法也会有默认的不带参数的构造方法。

特性：

1. 名字与类名相同。
2. 没有返回值，但不能用void声明构造函数。
3. 生成类的对象时自动执行，无需调用。

### 静态方法和实例方法有何不同

1、在外部调用静态方法时，可以使用"类名.方法名"的方式，也可以使用"对象名.方法名"的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。

**2、静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此限制。**

## J2EE

Servlet

Spring AOP 动态代理-JDK和CGLib、

特点

1、降低模块之间的耦合度

2、使系统容易扩展

3、更好的代码复用。

SpringMVC

Mybatis

### Spring的生命周期

### Spring的单例是怎么实现的？(单例注册表)

## JVM

JVM实现原理

GC机制、常用算法、项目中运用过没

分代、内存区

堆、栈的区别

如何实现oom或sof

### 引用和对象

<https://www.cnblogs.com/dolphin0520/p/3592498.html>

Person person;

person = new Person("张三");

person = new Person("李四");

其中,person就是一个引用。可以理解成指针，但有区别。

**强引用**（只要某个对象有强引用与之关联，JVM就不会回收）：

Object[] objArr = new Object[1000]; //方法调用完或显示置null就可以回收

**软引用**（只有在内存不足的时候JVM才会回收该对象）：

SoftReference<String> sr = new SoftReference<String>(new String("hello"));

**弱引用**（当JVM进行垃圾回收时，无论内存是否充足，都会回收）：

WeakReference<String> sr = new WeakReference<String>(new String("hello"));

**虚引用**（如果一个对象与虚引用关联，则跟没有引用与之关联一样）。

虚引用必须和引用队列关联使用，当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会把这个虚引用加入到与之 关联的引用队列中。

软引用适合用来进行缓存，当内存不够时能让JVM回收内存，弱引用能用来在回调函数中防止内存泄露。

### 说说GC的过程

young gc和full gc的触发条件和回收过程以及对象创建的过程

### CMS GC有什么问题？

并发清除算法，浮动垃圾，短暂停顿

### 怎么避免产生浮动垃圾？

记得有个VM参数设置可以让扫描新生代之前进行一次young gc，但是因为gc是虚拟机自动调度的，所以不保证一定执行。但是还有参数可以让虚拟机强制执行一次young gc

**强制young gc会有什么问题？**(STW停顿时间变长)

### G1回收过程

young gc、并发阶段、混合阶段、full gc，说了Remember Set

Remember Set底层是怎么实现的？

## Mysql、并发、锁

### 乐观锁和悲观锁的区别

悲观锁假定会发生冲突，访问的时候都要先获得锁，保证同一个时刻只有线程获得锁，读读也会阻塞；乐观锁假设不会发生冲突，只有在提交操作的时候检查是否有冲突

这两种锁在Java和MySQL分别是怎么实现的？（Java乐观锁通过CAS实现，悲观锁通过synchronize实现。mysql乐观锁通过MVCC，也就是版本实现，悲观锁可以通过select... for update加上排它锁

### （MySQL）使用什么存储引擎，为什么使用InnnoDB？

支持事务、聚簇索引、MVCC

### 订单表有做拆分么，怎么拆的？

垂直拆分和水平拆分

### 分库分表后怎么解决读写压力？

一主多从、多主多从

拆分后主键怎么保证惟一？(UUID、Snowflake算法)

Snowflake生成的ID是全局递增唯一么？(不是，只是全局唯一，单机递增)

怎么实现全局递增的唯一ID？(讲了TDDL的一次取一批ID，然后再本地慢慢分配的做法)

### Mysql的索引结构说下

说了B+树，B+树可以对叶子结点顺序查找，因为叶子结点存放了数据结点且有序

主键索引和普通索引的区别？(主键索引的叶子结点存放了整行记录，普通索引的叶子结点存放了主键ID，查询的时候需要做一次回表查询)一定要回表查询么？(不一定，当查询的字段刚好是索引的字段或者索引的一部分，就可以不用回表，这也是索引覆盖的原理)

## 分布式

### RpcContext是在什么维度传递的？(线程)

### Dubbo的远程调用怎么实现的？

读取配置、拼装url、创建Invoker、服务导出、服务注册以及消费者通过动态代理、filter、获取Invoker列表、负载均衡等过程

## 参考资料

<https://github.com/Snailclimb/JavaGuide/blob/master/README.md#%E5%9F%BA%E7%A1%80>