1、面向对象的三个基本特征?

面向对象的三个基本特征是：封装、继承和多态。

继承：让某个类型的对象获得另一个类型的对象的属性的方法。继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。

封装：隐藏部分对象的属性和实现细节，对数据的访问只能通过外公开的接口。通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

多态：对于同一个行为，不同的子类对象具有不同的表现形式。多态存在的3个条件：1）继承；2）重写；3）父类引用指向子类对象。

举个简单的例子：英雄联盟里面我们按下 Q 键这个动作：

- 对于亚索，就是斩钢闪
- 对于提莫，就是致盲吹箭
- 对于剑圣，就是阿尔法突袭

同一个事件发生在不同的对象上会产生不同的结果。

我再举一个简单的例子帮助大家理解，这个例子可能不是完全准确，但是我认为是有利于理解的。

```
public class Animal { // 动物
    public void sleep() {
        System.out.println("躺着睡");
    }
}
class Horse extends Animal { // 马 是一种动物
    public void sleep() {
        System.out.println("站着睡");
    }
}
class Cat extends Animal { // 猫 是一种动物
    private int age;
    public int getAge() {
        return age + 1;
    }
    @Override
    public void sleep() {
        System.out.println("四脚朝天的睡");
    }
}
```

在这个例子中：

House 和 Cat 都是 Animal，所以他们都继承了 Animal，同时也从 Animal 继承了 sleep 这个行为。

但是针对 sleep 这个行为，House 和 Cat 进行了重写，有了不同的表现形式（实现），这个我们称为多态。

在 Cat 里，将 age 属性定义为 private，外界无法直接访问，要获取 Cat 的 age 信息只能通过 getAge 方法，从而对外隐藏了 age 属性，这个就叫做封装。当然，这边 age 只是个例子，实际使用中可能是一个复杂很多的对象。

2、访问修饰符public，private，protected，以及不写时的区别?

修饰符	当前类	同 包	子 类	其他包
public	√	√	√	√
protected	√	√	√	×
不写	√	√	×	×
private	√	×	×	×

3、下面两个代码块能正常编译和执行吗?

```
// 代码块1
short s1 = 1; s1 = s1 + 1;
// 代码块2
short s1 = 1; s1 += 1;
```

代码块1编译报错，错误原因是：不兼容的类型: 从int转换到short可能会有损失”。

代码块2正常编译和执行。

我们将代码块2进行编译，字节码如下：

```
public class com.joonwhee.open.demo.Convert {
  public com.joonwhee.open.demo.Convert();
    Code:
       0: aload_0
       1: invokespecial #1 // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: iconst_1 // 将int类型值1入（操作数）栈
       1: istore_1 // 将栈顶int类型值保存到局部变量1中
       2: iload_1 // 从局部变量1中装载int类型值入栈
       3: iconst_1 // 将int类型值1入栈
       4: iadd // 将栈顶两int类型数相加，结果入栈
       5: i2s // 将栈顶int类型值截断成short类型值，后带符号扩展成int类型值入栈。
       6: istore_1 // 将栈顶int类型值保存到局部变量1中
       7: return
}
```

可以看到字节码中包含了 i2s 指令，该指令用于将 int 转成 short。i2s 是 int to short 的缩写。

其实，s1 += 1 相当于 s1 = (short)(s1 + 1)，有兴趣的可以自己编译下这两行代码的字节码，你会发现是一摸一样的。

说好的 Java 基础题，怎么又开始变态起来了？？？

4、基础考察，指出下题的输出结果

```
public static void main(String[] args) {
    Integer a = 128, b = 128, c = 127, d = 127;
    System.out.println(a == b);
    System.out.println(c == d);
}
```

答案是：false，true。

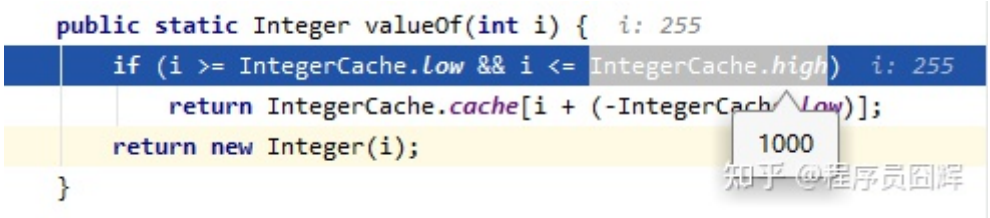
执行 Integer a = 128，相当于执行：Integer a = Integer.valueOf(128)，基本类型自动转换为包装类的过程称为自动装箱（autoboxing）。

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

在 Integer 中引入了 IntegerCache 来缓存一定范围的值，IntegerCache 默认情况下范围为：-128~127。

本题中的 127 命中了 IntegerCache，所以 c 和 d 是相同对象，而 128 则没有命中，所以 a 和 b 是不同对象。

但是这个缓存范围时可以修改的，可能有些人不知道。可以通过JVM启动参数：-XX:AutoBoxCacheMax= 来修改上限值，如下图所示：



5、用最有效率的方法计算2乘以8?

2 << 3。

进阶：通常情况下，可以认为位运算是性能最高的。但是，其实编译器现在已经“非常聪明了”，很多指令编译器都能自己做优化。所以在实际实用中，我们无需特意去追求实用位运算，这样不仅会导致代码可读性很差，而且某些自作聪明的优化反而会误导编译器，使得编译器无法进行更好的优化。

这可能就是所谓的“猪队友”吧。

6、&和&&的区别?

&&：逻辑与运算符。当运算符左右两边的表达式都为 true，才返回 true。同时具有短路性，如果第一个表达式为 false，则直接返回 false。

&：逻辑与运算符、按位与运算符。

按位与运算符：用于二进制的计算，只有对应的两个二进位均为1时，结果位才为1 ，否则为0。

逻辑与运算符：& 在用于逻辑与时，和 && 的区别是不具有短路性。所在通常使用逻辑与运算符都会使用 &&，而 & 更多的适用于位运算。

7、String 是 Java 基本数据类型吗?

答：不是。Java 中的基本数据类型只有8个：byte、short、int、long、float、double、char、boolean；除了基本类型（primitive type），剩下的都是引用类型（reference type）。

基本数据类型：数据直接存储在栈上

引用数据类型区别：数据存储在堆上，栈上只存储引用地址

8、String 类可以继承吗?

不行。String 类使用 final 修饰，无法被继承。

9、String和StringBuilder、StringBuffer的区别?

String：String 的值被创建后不能修改，任何对 String 的修改都会引发新的 String 对象的生成。

StringBuffer：跟 String 类似，但是值可以被修改，使用 synchronized 来保证线程安全。

StringBuilder：StringBuffer 的非线程安全版本，没有使用 synchronized，具有更高的性能，推荐优先使用。

10、String s = new String("xyz") 创建了几个字符串对象?

一个或两个。如果字符串常量池已经有“xyz”，则是一个；否则，两个。

当字符创常量池没有 “xyz”，此时会创建如下两个对象：

一个是字符串字面量 "xyz" 所对应的、驻留（intern）在一个全局共享的字符串常量池中的实例，此时该实例也是在堆中，字符串常量池只放引用。

另一个是通过 new String() 创建并初始化的，内容与"xyz"相同的实例，也是在堆中。

11、String s = "xyz" 和 String s = new String("xyz") 区别?

两个语句都会先去字符串常量池中检查是否已经存在 “xyz”，如果有则直接使用，如果没有则会在常量池中创建 “xyz” 对象。

另外，String s = new String("xyz") 还会通过 new String() 在堆里创建一个内容与 "xyz" 相同的对象实例。

所以前者其实理解为被后者的所包含。

12、== 和 equals 的区别是什么?

==：运算符，用于比较基础类型变量和引用类型变量。

对于基础类型变量，比较的变量保存的值是否相同，类型不一定要相同。

```
short s1 = 1; long l1 = 1;
 // 结果：true。类型不同，但是值相同
System.out.println(s1 == l1);
```

对于引用类型变量，比较的是两个对象的地址是否相同。

```
Integer i1 = new Integer(1);
Integer i2 = new Integer(1);
// 结果：false。通过new创建，在内存中指向两个不同的对象
System.out.println(i1 == i2);
```

equals：Object 类中定义的方法，通常用于比较两个对象的值是否相等。

equals 在 Object 方法中其实等同于 ==，但是在实际的使用中，equals 通常被重写用于比较两个对象的值是否相同。

```
Integer i1 = new Integer(1);
Integer i2 = new Integer(1);
// 结果：true。两个不同的对象，但是具有相同的值
System.out.println(i1.equals(i2));

// Integer的equals重写方法
public boolean equals(Object obj) {
    if (obj instanceof Integer) {
        // 比较对象中保存的值是否相同
        return value == ((Integer)obj).intValue();
    }
    return false;
}
```

13、两个对象的 hashCode() 相同，则 equals() 也一定为 true，对吗?

不对。hashCode() 和 equals() 之间的关系如下：

当有 a.equals(b) == true 时，则 a.hashCode() == b.hashCode() 必然成立，

反过来，当 a.hashCode() == b.hashCode() 时，a.equals(b) 不一定为 true。

14、什么是反射

反射是指在运行状态中，对于任意一个类都能够知道这个类所有的属性和方法；并且对于任意一个对象，都能够调用它的任意一个方法；这种动态获取信息以及动态调用对象方法的功能称为反射机制。

15、深拷贝和浅拷贝区别是什么?

数据分为基本数据类型和引用数据类型。基本数据类型：数据直接存储在栈中；引用数据类型：存储在栈中的是对象的引用地址，真实的对象数据存放在堆内存里。

浅拷贝：对于基础数据类型：直接复制数据值；对于引用数据类型：只是复制了对象的引用地址，新旧对象指向同一个内存地址，修改其中一个对象的值，另一个对象的值随之改变。

深拷贝：对于基础数据类型：直接复制数据值；对于引用数据类型：开辟新的内存空间，在新的内存空间里复制一个一模一样的对象，新老对象不共享内存，修改其中一个对象的值，不会影响另一个对象。

深拷贝相比于浅拷贝速度较慢并且花销较大。

16、并发和并行有什么区别?

并发：两个或多个事件在同一时间间隔发生。

并行：两个或者多个事件在同一时刻发生。

并行是真正意义上，同一时刻做多件事情，而并发在同一时刻只会做一件事件，只是可以将时间切碎，交替做多件事情。

网上有个例子挺形象的：

你吃饭吃到一半，电话来了，你一直到吃完了以后才去接，这就说明你不支持并发也不支持并行。

你吃饭吃到一半，电话来了，你停了下来接了电话，接完后继续吃饭，这说明你支持并发。

你吃饭吃到一半，电话来了，你一边打电话一边吃饭，这说明你支持并行。

17、构造器是否可被 重写?

Constructor 不能被 override（重写），但是可以 overload（重载），所以你可以看到一个类中有多个构造函数的情况。

18、当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递?

值传递。Java 中只有值传递，对于对象参数，值的内容是对象的引用。

19、Java 静态变量和成员变量的区别。

```
public class Demo {
    /**
     * 静态变量：又称类变量，static修饰
     */
    public static String STATIC_VARIABLE = "静态变量";
    /**
     * 实例变量：又称成员变量，没有static修饰
     */
    public String INSTANCE_VARIABLE = "实例变量";
}
```

成员变量存在于堆内存中。静态变量存在于方法区中。

成员变量与对象共存亡，随着对象创建而存在，随着对象被回收而释放。静态变量与类共存亡，随着类的加载而存在，随着类的消失而消失。

成员变量所属于对象，所以也称为实例变量。静态变量所属于类，所以也称为类变量。

成员变量只能被对象所调用 。静态变量可以被对象调用，也可以被类名调用。

20、是否可以从一个静态（static）方法内部发出对非静态（non-static）方法的调用?

区分两种情况，发出调用时是否显示创建了对象实例。

1）没有显示创建对象实例：不可以发起调用，非静态方法只能被对象所调用，静态方法可以通过对象调用，也可以通过类名调用，所以静态方法被调用时，可能还没有创建任何实例对象。因此通过静态方法内部发出对非静态方法的调用，此时可能无法知道非静态方法属于哪个对象。

```
public class Demo {
    public static void staticMethod() {
        // 直接调用非静态方法：编译报错
        instanceMethod();
    }
    public void instanceMethod() {
        System.out.println("非静态方法");
    }
}
```

2）显示创建对象实例：可以发起调用，在静态方法中显示的创建对象实例，则可以正常的调用。

```
public class Demo {
    public static void staticMethod() {
        // 先创建实例对象，再调用非静态方法：成功执行
        Demo demo = new Demo();
        demo.instanceMethod();
    }
    public void instanceMethod() {
        System.out.println("非静态方法");
    }
}
```

21、初始化考察，请指出下面程序的运行结果。

```
public class InitialTest {
    public static void main(String[] args) {
        A ab = new B();
        ab = new B();
    }
}
class A {
    static { // 父类静态代码块
        System.out.print("A");
    }
    public A() { // 父类构造器
        System.out.print("a");
    }
}
class B extends A {
```

```
    static { // 子类静态代码块
        System.out.print("B");
    }
    public B() { // 子类构造器
        System.out.print("b");
    }
}
```

执行结果：ABabab，两个考察点：

1）静态变量只会初始化（执行）一次。

2）当有父类时，完整的初始化顺序为：父类静态变量（静态代码块）->子类静态变量（静态代码块）->父类非静态变量（非静态代码块）->父类构造器 ->子类非静态变量（非静态代码块）->子类构造器 。

关于初始化，这题算入门题，我之前还写过一道有（fei）点（chang）意（bian）思（tai）的进阶题目，有兴趣的可以看看：一道有意思的"初始化"面试题

22、重载（Overload）和重写（Override）的区别?

方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性，而后者实现的是运行时的多态性。

重载：一个类中有多个同名的方法，但是具有有不同的参数列表（参数类型不同、参数个数不同或者二者都不同）。

重写：发生在子类与父类之间，子类对父类的方法进行重写，参数都不能改变，返回值类型可以不相同，但是必须是父类返回值的派生类。即外壳不变，核心重写！重写的好处在于子类可以根据需要，定义特定于自己的行为。

23、为什么不能根据返回类型来区分重载?

如果我们有两个方法如下，当我们调用：test(1) 时，编译器无法确认要调用的是哪个。

```
// 方法1
int test(int a);
// 方法2
long test(int a);
```

方法的返回值只是作为方法运行之后的一个"状态"，但是并不是所有调用都关注返回值，所以不能将返回值作为重载的唯一区分条件。

24、抽象类（abstract class）和接口（interface）有什么区别?

抽象类只能单继承，接口可以多实现。

抽象类可以有构造方法，接口中不能有构造方法。

抽象类中可以有成员变量，接口中没有成员变量，只能有常量（默认就是 public static final）

抽象类中可以包含非抽象的方法，在 Java 7 之前接口中的所有方法都是抽象的，在 Java 8 之后，接口支持非抽象方法：default 方法、静态方法等。Java 9 支持私有方法、私有静态方法。

抽象类中的抽象方法类型可以是任意修饰符，Java 8 之前接口中的方法只能是 public 类型，Java 9 支持 private 类型。

设计思想的区别：

接口是自上而下的抽象过程，接口规范了某些行为，是对某一行为的抽象。我需要这个行为，我就去实现某个接口，但是具体这个行为怎么实现，完全由自己决定。

抽象类是自下而上的抽象过程，抽象类提供了通用实现，是对某一类事物的抽象。我们在写实现类的时候，发现某些实现类具有几乎相同的实现，因此我们将这些相同的实现抽取出来成为抽象类，然后如果有一些差异点，则可以提供抽象方法来支持自定义实现。

我在网上看到有个说法，挺形象的：

普通类像亲爹 ，他有啥都是你的。

抽象类像叔伯，有一部分会给你，还能指导你做事的方法。

接口像干爹，可以给你指引方法，但是做成啥样得你自己努力实现。

25、Error 和 Exception 有什么区别?

Error 和 Exception 都是 Throwable 的子类，用于表示程序出现了不正常的情况。区别在于：

Error 表示系统级的错误和程序不必处理的异常，是恢复不是不可能但很困难的情况下的一种严重问题，比如内存溢出，不可能指望程序能处理这样的情况。

Exception 表示需要捕捉或者需要程序进行处理的异常，是一种设计或实现问题，也就是说，它表示如果程序运行正常，从不会发生的情况。

26、Java 中的 final 关键字有哪些用法?

修饰类：该类不能再派生出新的子类，不能作为父类被继承。因此，一个类不能同时被声明为abstract 和 final。

修饰方法：该方法不能被子类重写。

修饰变量：该变量必须在声明时给定初值，而在以后只能读取，不可修改。 如果变量是对象，则指的是引用不可修改，但是对象的属性还是可以修改的。

```
public class FinalDemo {
    // 不可再修改该变量的值
    public static final int FINAL_VARIABLE = 0;
    // 不可再修改该变量的引用，但是可以直接修改属性值
    public static final User USER = new User();
    public static void main(String[] args) {
        // 输出：User(id=0, name=null, age=0)
        System.out.println(USER);
        // 直接修改属性值
        USER.setName("test");
        // 输出：User(id=0, name=test, age=0)
        System.out.println(USER);
    }
```

```
}
```

27、阐述 final、finally、finalize 的区别。

其实是三个完全不相关的东西，只是长的有点像。。

final 如上所示。

finally：finally 是对 Java 异常处理机制的最佳补充，通常配合 try、catch 使用，用于存放那些无论是否出现异常都一定会执行的代码。在实际使用中，通常用于释放锁、数据库连接等资源，把资源释放方法放到 finally 中，可以大大降低程序出错的几率。

finalize：Object 中的方法，在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。finalize()方法仅作为了解即可，在 Java 9 中该方法已经被标记为废弃，并添加新的 java.lang.ref.Cleaner，提供了更灵活和有效的方法来释放资源。这也侧面说明了，这个方法的设计是失败的，因此更加不能去使用它。

28、try、catch、finally 考察，请指出下面程序的运行结果。

```
public class TryDemo {
    public static void main(String[] args) {
        System.out.println(test());
    }
    public static int test() {
        try {
            return 1;
        } catch (Exception e) {
            return 2;
        } finally {
            System.out.print("3");
        }
    }
}
```

执行结果：31。

相信很多同学应该都做对了，try、catch。finally 的基础用法，在 return 前会先执行 finally 语句块，所以是先输出 finally 里的 3，再输出 return 的 1。

29、try、catch、finally 考察2，请指出下面程序的运行结果。

```
public class TryDemo {
    public static void main(String[] args) {
        System.out.println(test1());
    }
    public static int test1() {
        try {
            return 2;
        } finally {
            return 3;
        }
    }
}
```

执行结果：3。

这题有点先将，但也不难，try 返回前先执行 finally，结果 finally 里不按套路出牌，直接 return 了，自然也就走不到 try 里面的 return 了。

finally 里面使用 return 仅存在于面试题中，实际开发中千万不要这么用。

30、try、catch、finally 考察3，请指出下面程序的运行结果。

```
public class TryDemo {
    public static void main(String[] args) {
        System.out.println(test1());
    }
    public static int test1() {
        int i = 0;
        try {
            i = 2;
            return i;
        } finally {
            i = 3;
        }
    }
}
```

执行结果：2。

这边估计有不少同学会以为结果应该是 3，因为我们知道在 return 前会执行 finally，而 i 在 finally 中被修改为 3 了，那最终返回 i 不是应该为 3 吗？确实很容易这么想，我最初也是这么想的，当初的自己还是太年轻了啊。

这边的根本原因是，在执行 finally 之前，JVM 会先将 i 的结果暂存起来，然后 finally 执行完毕后，会返回之前暂存的结果，而不是返回 i，所以即使这边 i 已经被修改为 3，最终返回的还是之前暂存起来的结果 2。

这边其实根据字节码可以很容易看出来，在进入 finally 之前，JVM 会使用 iload、istore 两个指令，将结果暂存，在最终返回时在通过 iload、ireturn 指令返回暂存的结果。

为了避免气氛再次变态起来，我这边就不贴具体的字节码程序了，有兴趣的同学可以自己编译查看下。

31、JDK1.8之后有哪些新特性?

接口默认方法：Java 8允许我们给接口添加一个非抽象的方法实现，只需要使用 default关键字即可

Lambda 表达式和函数式接口：Lambda 表达式本质上是一段匿名内部类，也可以是一段可以传递的代码。Lambda 允许把函数作为一个方法的参数（函数作为参数传递到方法中），使用 Lambda 表达式使代码更加简洁，但是也不要滥用，否则会有可读性等问题，《Effective Java》作者 Josh Bloch 建议使用 Lambda 表达式最好不要超过3行。

Stream API：用函数式编程方式在集合类上进行复杂操作的工具，配合Lambda表达式可以方便的对集合进行处理。Java8 中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。使用Stream API 对集合数据进行操作，就类似于使用 SQL 执行的数据库查询。也可以使用 Stream API 来并行执行操作。简而言之，Stream API 提供了一种高效且易于使用的处理数据的方式。

方法引用：方法引用提供了非常有用的语法，可以直接引用已有Java类或对象（实例）的方法或构造器。与lambda联合使用，方法引用可以使语言的构造更紧凑简洁，减少冗余代码。

日期时间API：Java 8 引入了新的日期时间API改进了日期时间的管理。

Optional 类：著名的 NullPointerException 是引起系统失败最常见的原因。很久以前 Google Guava 项目引入了 Optional 作为解决空指针异常的一种方式，不赞成代码被 null 检查的代码污染，期望程序员写整洁的代码。受Google Guava的鼓励，Optional 现在是Java 8库的一部分。

新工具：新的编译工具，如：Nashorn引擎 jjs、 类依赖分析器 jdeps。

50、wait() 和 sleep() 方法的区别

来源不同：sleep() 来自 Thread 类，wait() 来自 Object 类。

对于同步锁的影响不同：sleep() 不会该表同步锁的行为，如果当前线程持有同步锁，那么 sleep 是不会让线程释放同步锁的。wait() 会释放同步锁，让其他线程进入 synchronized 代码块执行。

使用范围不同：sleep() 可以在任何地方使用。wait() 只能在同步控制方法或者同步控制块里面使用，否则会抛 IllegalMonitorStateException。

恢复方式不同：两者会暂停当前线程，但是在恢复上不太一样。sleep() 在时间到了之后会重新恢复；wait() 则需要其他线程调用同一对象的 notify()/nofityAll() 才能重新恢复。

51、线程的 sleep() 方法和 yield() 方法有什么区别?

线程执行 sleep() 方法后进入超时等待（TIMED_WAITING）状态，而执行 yield() 方法后进入就绪（READY）状态。

sleep() 方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程运行的机会；yield() 方法只会给相同优先级或更高优先级的线程以运行的机会。

52、线程的 join() 方法是干啥用的?

用于等待当前线程终止。如果一个线程A执行了 threadB.join() 语句，其含义是：当前线程A等待 threadB 线程终止之后才从 threadB.join() 返回继续往下执行自己的代码。

53、编写多线程程序有几种实现方式?

通常来说，可以认为有三种方式：1）继承 Thread 类；2）实现 Runnable 接口；3）实现 Callable 接口。

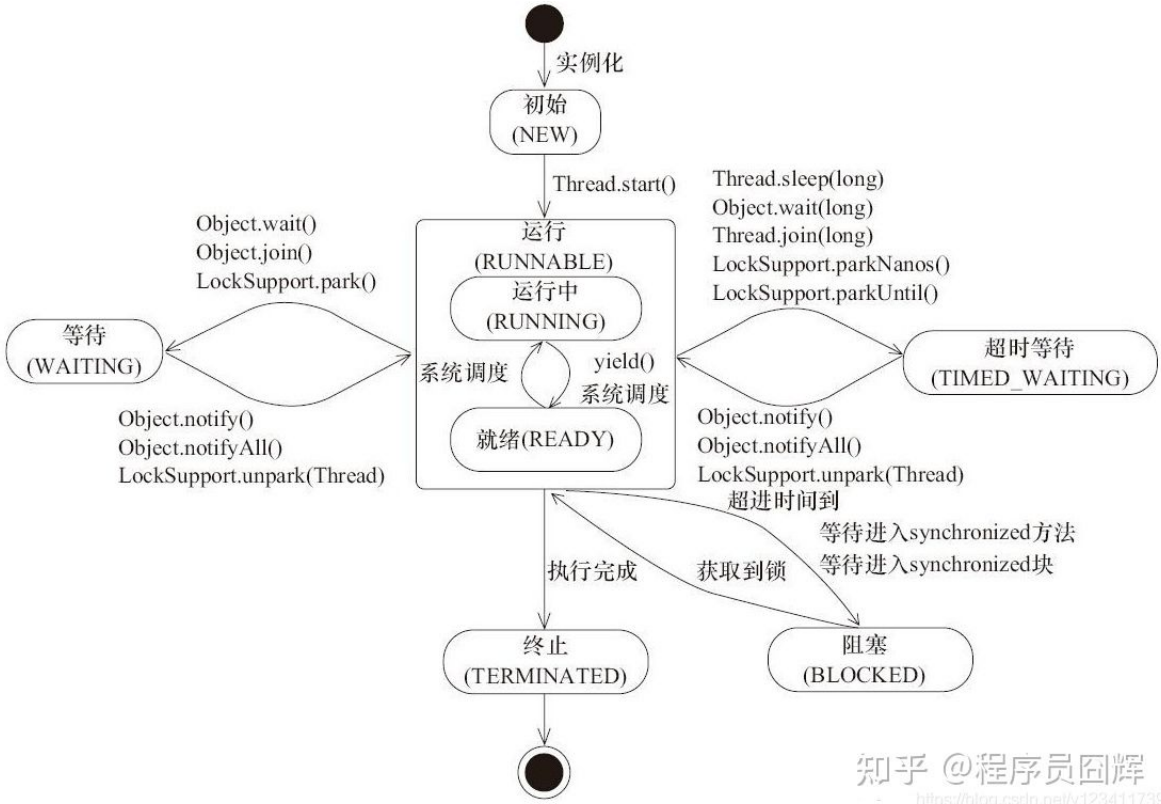
其中，Thread 其实也是实现了 Runable 接口。Runnable 和 Callable 的主要区别在于是否有返回值。

54、Thread 调用 start() 方法和调用 run() 方法的区别

run()：普通的方法调用，在主线程中执行，不会新建一个线程来执行。

start()：新启动一个线程，这时此线程处于就绪（可运行）状态，并没有运行，一旦得到 CPU 时间片，就开始执行 run() 方法。

55、线程的状态流转



一个线程可以处于以下状态之一：

NEW：新建但是尚未启动的线程处于此状态，没有调用 start() 方法。

RUNNABLE：包含就绪（READY）和运行中（RUNNING）两种状态。线程调用 start() 方法会会进入就绪（READY）状态，等待获取 CPU 时间片。如果成功获取到 CPU 时间片，则会进入运行中（RUNNING）状态。

BLOCKED：线程在进入同步方法/同步块（synchronized）时被阻塞，等待同步锁的线程处于此状态。

WAITING：无限期等待另一个线程执行特定操作的线程处于此状态，需要被显示的唤醒，否则会一直等待下去。例如对于 Object.wait()，需要等待另一个线程执行 Object.notify() 或 Object.notifyAll()；对于 Thread.join()，则需要等待指定的线程终止。

TIMED_WAITING：在指定的时间内等待另一个线程执行某项操作的线程处于此状态。跟 WAITING 类似，区别在于该状态有超时时间参数，在超时时间到了后会自动唤醒，避免了无期限的等待。

TERMINATED：执行完毕已经退出的线程处于此状态。

线程在给定的时间点只能处于一种状态。这些状态是虚拟机状态，不反映任何操作系统线程状态。

56、synchronized 和 Lock 的区别

1）Lock 是一个接口；synchronized 是 Java 中的关键字，synchronized 是内置的语言实现；

2）Lock 在发生异常时，如果没有主动通过 unLock() 去释放锁，很可能会造成死锁现象，因此使用 Lock 时需要在 finally 块中释放锁；synchronized 不需要手动获取锁和释放锁，在发生异常时，会自动释放锁，因此不会导致死锁现象发生；

3）Lock 的使用更加灵活，可以有响应中断、有超时时间等；而 synchronized 却不行，使用 synchronized 时，等待的线程会一直等待下去，直到获取到锁；

4）在性能上，随着近些年 synchronized 的不断优化，Lock 和 synchronized 在性能上已经没有很明显的差距了，所以性能不应该成为我们选择两者的主要原因。官方推荐尽量使用 synchronized，除非 synchronized 无法满足需求时，则可以使用 Lock。

57、synchronized 各种加锁场景的作用范围

1.作用于非静态方法，锁住的是对象实例（this），每一个对象实例有一个锁。

```
public synchronized void method() {}
```

2.作用于静态方法，锁住的是类的Class对象，因为Class的相关数据存储在永久代元空间，元空间是全局共享的，因此静态方法锁相当于类的一个全局锁，会锁所有调用该方法的线程。

```
public static synchronized void method() {}
```

3.作用于 Lock.class，锁住的是 Lock 的Class对象，也是全局只有一个。

```
synchronized (Lock.class) {}
```

4.作用于 this，锁住的是对象实例，每一个对象实例有一个锁。

```
synchronized (this) {}
```

5.作用于静态成员变量，锁住的是该静态成员变量对象，由于是静态变量，因此全局只有一个。

```
public static Object monitor = new Object();
synchronized (monitor) {}
```

58、如何检测死锁?

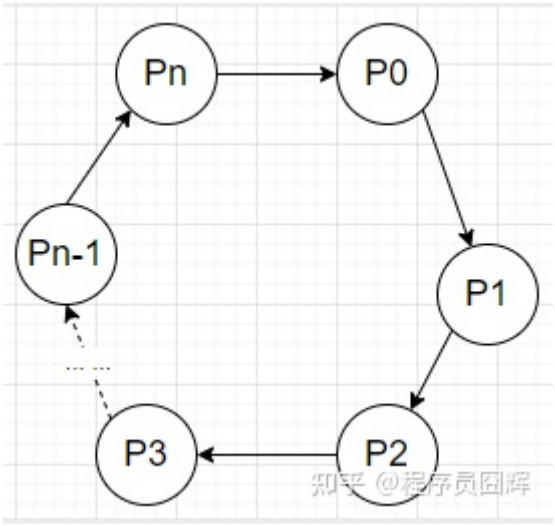
死锁的四个必要条件：

1）互斥条件：进程对所分配到的资源进行排他性控制，即在一段时间内某资源仅为一个进程所占有。此时若有其他进程请求该资源，则请求进程只能等待。

2）请求和保持条件：进程已经获得了至少一个资源，但又对其他资源发出请求，而该资源已被其他进程占有，此时该进程的请求被阻塞，但又对自己获得的资源保持不放。

3）不可剥夺条件：进程已获得的资源在未使用完毕之前，不可被其他进程强行剥夺，只能由自己释放。

4）环路等待条件：存在一种进程资源的循环等待链，链中每一个进程已获得的资源同时被 链中下一个进程所请求。即存在一个处于等待状态的进程集合{Pl, P2, ..., pn}，其中 Pi 等待的资源被 P(i+1) 占有（i=0, 1, ..., n-1），Pn 等待的资源被 P0占 有，如下图所示。



59、怎么预防死锁?

预防死锁的方式就是打破四个必要条件中的任意一个即可。

1）打破互斥条件：在系统里取消互斥。若资源不被一个进程独占使用，那么死锁是肯定不会发生的。但一般来说在所列的四个条件中，“互斥”条件是无法破坏的。因此，在死锁预防里主要是破坏其他几个必要条件，而不去涉及破坏“互斥”条件。。

2）打破请求和保持条件：1）采用资源预先分配策略，即进程运行前申请全部资源，满足则运行，不然就等待。 2）每个进程提出新的资源申请前，必须先释放它先前所占有的资源。

3）打破不可剥夺条件：当进程占有某些资源后又进一步申请其他资源而无法满足，则该进程必须释放它原来占有的资源。

4）打破环路等待条件：实现资源有序分配策略，将系统的所有资源统一编号，所有进程只能采用按序号递增的形式申请资源。

60、为什么要使用线程池？直接new个线程不是很舒服?

如果我们在方法中直接new一个线程来处理，当这个方法被调用频繁时就会创建很多线程，不仅会消耗系统资源，还会降低系统的稳定性，一不小心把系统搞崩了，就可以直接去财务那结帐了。

如果我们合理的使用线程池，则可以避免把系统搞崩的窘境。总得来说，使用线程池可以带来以下几个好处：

- 降低资源消耗。通过重复利用已创建的线程，降低线程创建和销毁造成的消耗。
- 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。
- 增加线程的可管理型。线程是稀缺资源，使用线程池可以进行统一分配，调优和监控。

61、线程池的核心属性有哪些?

threadFactory（线程工厂）：用于创建工作线程的工厂。

corePoolSize（核心线程数）：当线程池运行的线程少于 corePoolSize 时，将创建一个新线程来处理请求，即使其他工作线程处于空闲状态。

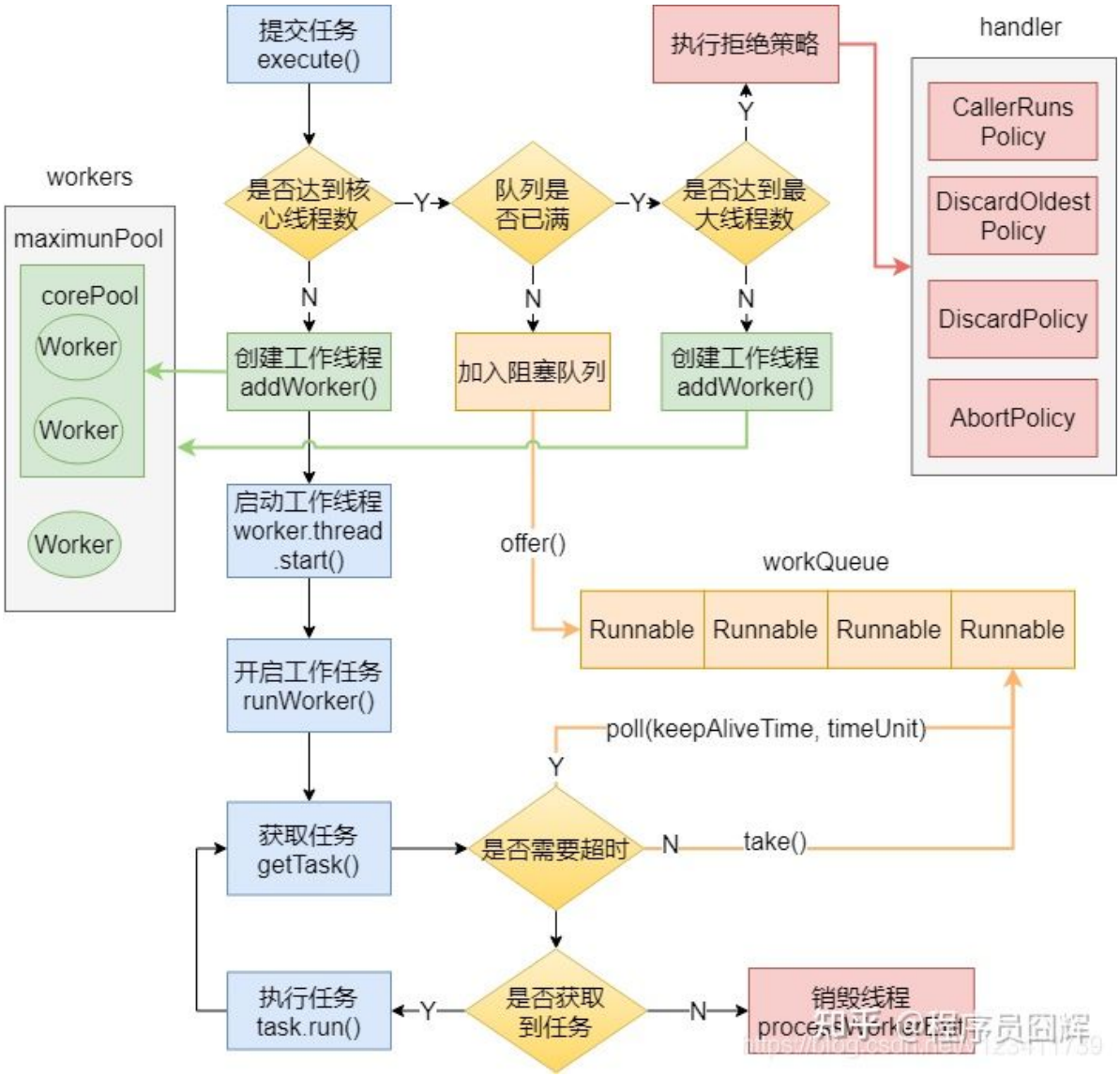
workQueue（队列）：用于保留任务并移交给工作线程的阻塞队列。

maximumPoolSize（最大线程数）：线程池允许开启的最大线程数。

handler（拒绝策略）：往线程池添加任务时，将在下面两种情况触发拒绝策略：1）线程池运行状态不是 RUNNING；2）线程池已经达到最大线程数，并且阻塞队列已满时。

keepAliveTime（保持存活时间）：如果线程池当前线程数超过 corePoolSize，则多余的线程空闲时间超过 keepAliveTime 时会被终止。

62、说下线程池的运作流程。



63、线程池有哪些拒绝策略?

AbortPolicy：中止策略。默认的拒绝策略，直接抛出 RejectedExecutionException。调用者可以捕获这个异常，然后根据需求编写自己的处理代码。

DiscardPolicy：抛弃策略。什么都不做，直接抛弃被拒绝的任务。

DiscardOldestPolicy：抛弃最老策略。抛弃阻塞队列中最老的任务，相当于就是队列中下一个将要被执行的任务，然后重新提交被拒绝的任务。如果阻塞队列是一个优先队列，那么“抛弃最旧的”策略将导致抛弃优先级最高的任务，因此最好不要将该策略和优先级队列放在一起使用。

CallerRunsPolicy：调用者运行策略。在调用者线程中执行该任务。该策略实现了一种调节机制，该策略既不会抛弃任务，也不会抛出异常，而是将任务回退到调用者（调用线程池执行任务的主线程），由于执行任务需要一定时间，因此主线程至少在一段时间内不能提交任务，从而使得线程池有时间来处理完正在执行的任务。

70、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来搜索的专业户）: 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。

71、ArrayList 和 LinkedList 的区别。

ArrayList 底层基于动态数组实现，LinkedList 底层基于链表实现。

对于按 index 索引数据（get/set方法）：ArrayList 通过 index 直接定位到数组对应位置的节点，而 LinkedList需要从头结点或尾节点开始遍历，直到寻找到目标节点，因此在效率上 ArrayList 优于 LinkedList。

对于随机插入和删除：ArrayList 需要移动目标节点后面的节点（使用System.arraycopy 方法移动节点），而 LinkedList 只需修改目标节点前后节点的 next 或 prev 属性即可，因此在效率上 LinkedList 优于 ArrayList。

对于顺序插入和删除：由于 ArrayList 不需要移动节点，因此在效率上比 LinkedList 更好。这也是为什么在实际使用中 ArrayList 更多，因为大部分情况下我们的使用都是顺序插入。

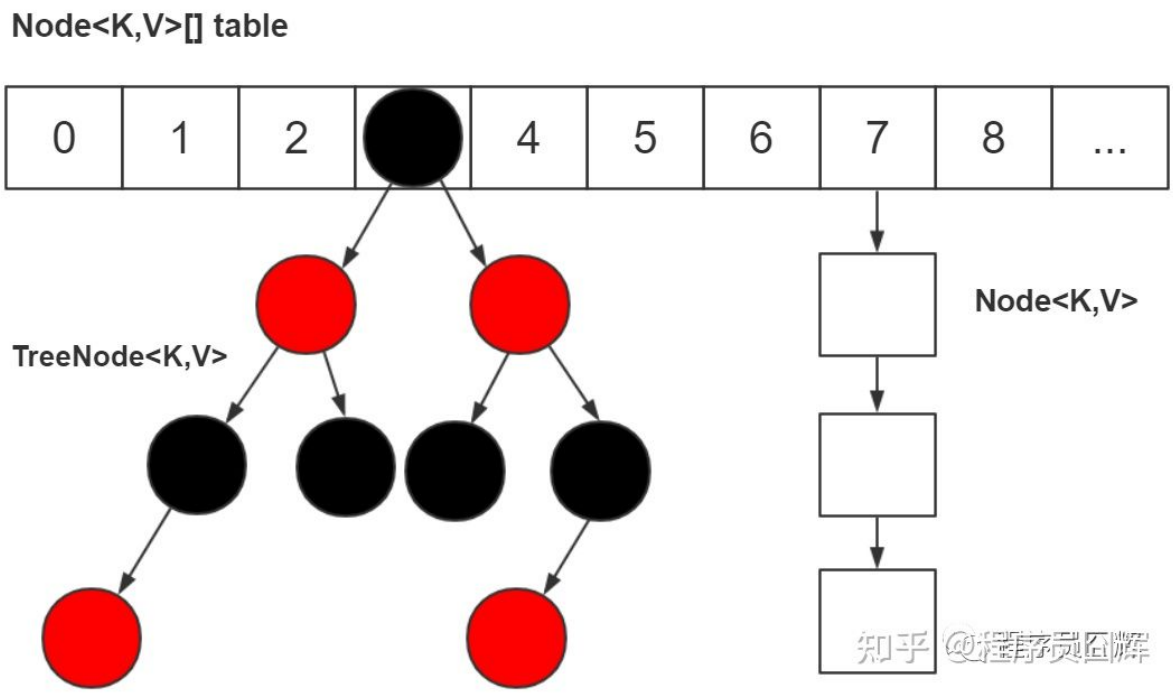
72、ArrayList 和 Vector 的区别。

Vector 和 ArrayList 几乎一致，唯一的区别是 Vector 在方法上使用了 synchronized 来保证线程安全，因此在性能上 ArrayList 具有更好的表现。

有类似关系的还有：StringBuilder 和 StringBuffer、HashMap 和 Hashtable。

73、介绍下 HashMap 的底层数据结构

我们现在用的都是 JDK 1.8，底层是由“数组+链表+红黑树”组成，如下图，而在 JDK 1.8 之前是由“数组+链表”组成。



74、为什么要改成“数组+链表+红黑树”?

主要是为了提升在 hash 冲突严重时（链表过长）的查找性能，使用链表的查找性能是 O(n)，而使用红黑树是 O(logn)。

75、那在什么时候用链表？什么时候用红黑树?

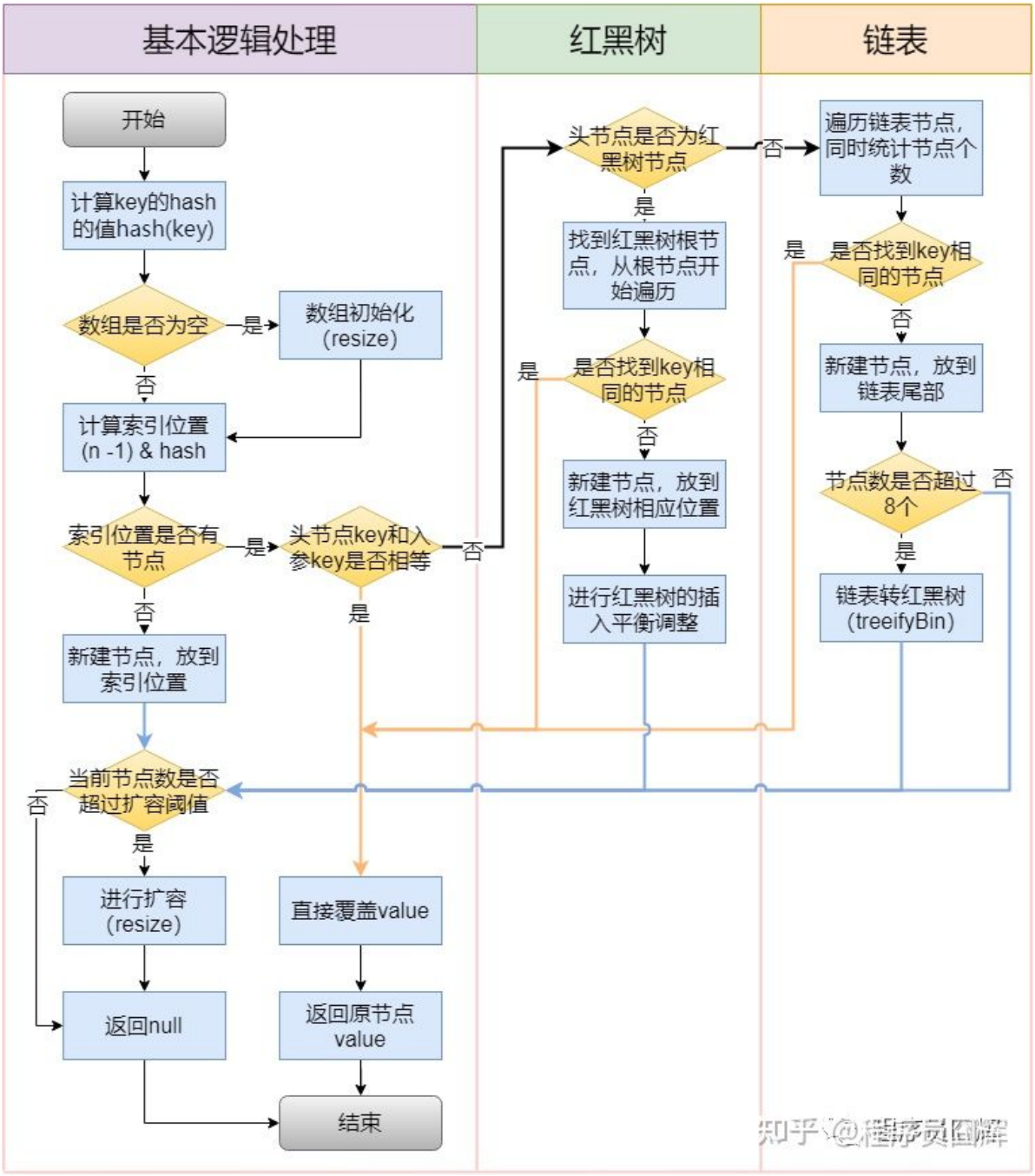
对于插入，默认情况下是使用链表节点。当同一个索引位置的节点在新增后超过8个（阈值8）：如果此时数组长度大于等于 64，则会触发链表节点转红黑树节点（treeifyBin）；而如果数组长度小于64，则不会触发链表转红黑树，而是会进行扩容，因为此时的数据量还比较小。

对于移除，当同一个索引位置的节点在移除后达到 6 个，并且该索引位置的节点为红黑树节点，会触发红黑树节点转链表节点（untreeify）。

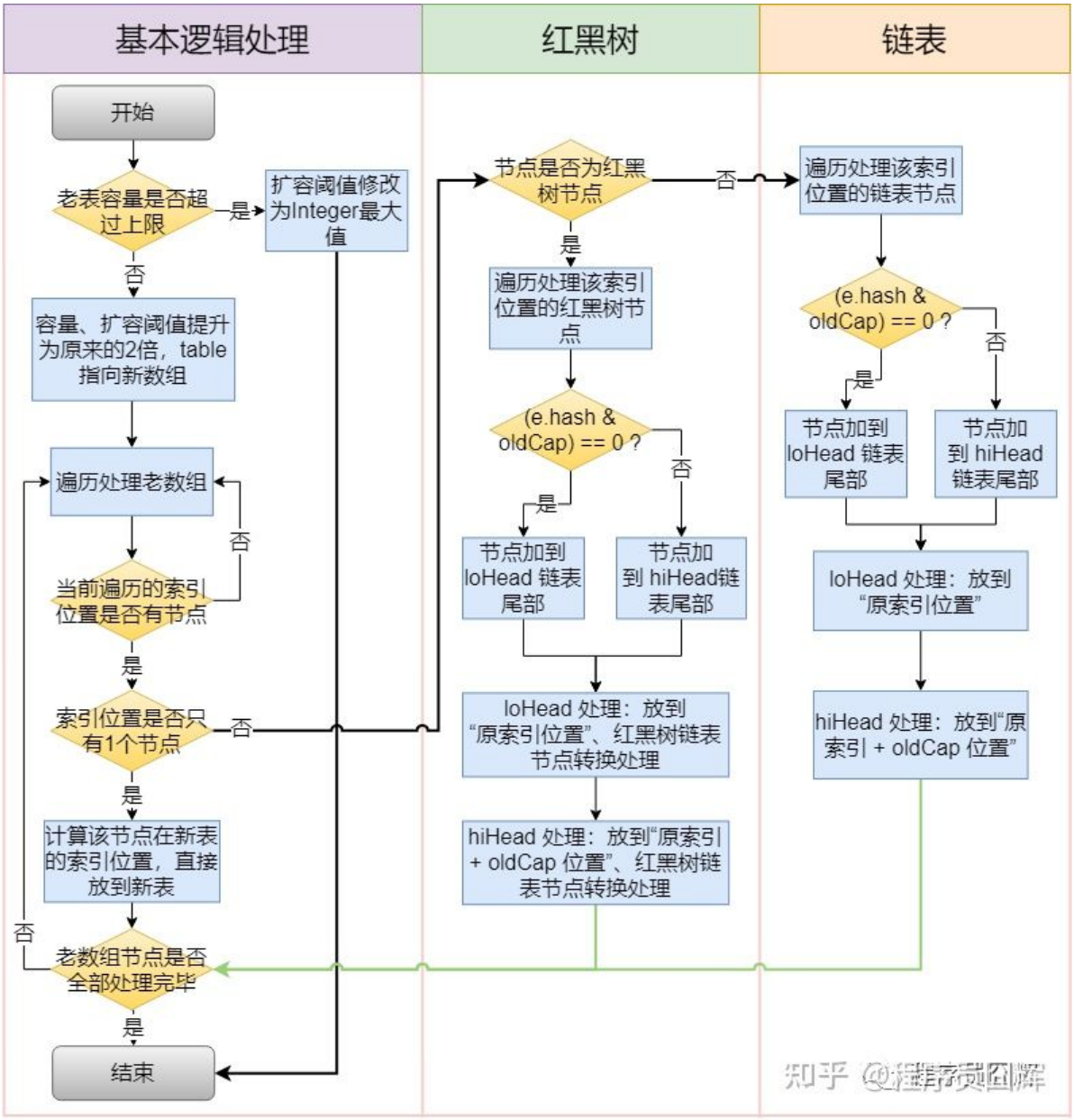
76、HashMap 的默认初始容量是多少？HashMap 的容量有什么限制吗?

默认初始容量是16。HashMap 的容量必须是2的N次方，HashMap 会根据我们传入的容量计算一个大于等于该容量的最小的2的N次方，例如传 9，容量为16。

77、HashMap 的插入流程是怎么样的?



78、HashMap 的扩容（resize）流程是怎么样的?



79、除了 HashMap，还用过哪些 Map，在使用时怎么选择?

类	介绍	使用场景
Hashtable	早期的线程安全 Map，直接通过在方法加 synchronized 实现线程安全	现在理论上不会使用
Cocurrent HashMap	线程安全的 Map，通过 synchronized + CAS 实现线程安全	需要保证线程安全
LinkedHashMap	能记录访问顺序或插入顺序的 Map，通过 head、tail 属性维护有序双向链表，通过 Entry 的 after、before 属性维护节点的顺序	需要记录访问顺序或插入顺序
TreeMap	通过实现 Comparator 实现自定义顺序的 Map，如果没有指定 Comparator 则会按 key 的升序排序，key 如果没有实现 Comparable 接口，则会抛异常	需要自定义排序
HashMap	最通用的 Map，非线程安全、无序	无特殊需求都可使用

80、HashMap 和Hashtable 的区别?

HashMap 允许 key 和 value 为 null，Hashtable 不允许。

HashMap 的默认初始容量为 16，Hashtable 为 11。

HashMap 的扩容为原来的 2 倍，Hashtable 的扩容为原来的 2 倍加 1。

HashMap 是非线程安全的，Hashtable是线程安全的。

HashMap 的 hash 值重新计算过，Hashtable 直接使用 hashCode。

HashMap 去掉了 Hashtable 中的 contains 方法。

HashMap 继承自 AbstractMap 类，Hashtable 继承自 Dictionary 类。

90、Java 内存结构（运行时数据区）

程序计数器：线程私有。一块较小的内存空间，可以看作当前线程所执行的字节码的行号指示器。如果线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Native方法，这个计数器值则为空。

Java虚拟机栈：线程私有。它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法在执行的同时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。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

本地方法栈：线程私有。本地方法栈与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的Native方法服务。

Java堆：线程共享。对大多数应用来说，Java堆是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

方法区：与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息（构造方法、接口定义）、常量、静态变量、即时编译器编译后的代码（字节码）等数据。方法区是JVM规范中定义的一个概念，具体放在哪里，不同的实现可以放在不同的地方。

运行时常量池：运行时常量池是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

```
String str = new String("hello");
```

上面的语句中变量 str 放在栈上，用 new 创建出来的字符串对象放在堆上，而"hello"这个字面量是放在堆中。

91、什么是双亲委派模型?

如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

92、Java虚拟机中有哪些类加载器?

启动类加载器（Bootstrap ClassLoader）：

这个类加载器负责将存放在<JAVA_HOME>\lib目录中的，或者被-Xbootclasspath参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如rt.jar，名字不符合的类库即使放在lib目录中也不会被加载）类库加载到虚拟机内存中。

扩展类加载器（Extension ClassLoader）：

这个加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载<JAVA_HOME>\lib\ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。

应用程序类加载器（Application ClassLoader）：

这个类加载器由sun.misc.Launcher$AppClassLoader实现。由于这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值，所以一般也称它为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

自定义类加载器：

用户自定义的类加载器。

93、类加载的过程

类加载的过程包括：加载、验证、准备、解析、初始化，其中验证、准备、解析统称为连接。

加载：通过一个类的全限定名来获取定义此类的二进制字节流，在内存中生成一个代表这个类的java.lang.Class对象。

验证：确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

准备：为静态变量分配内存并设置静态变量初始值，这里所说的初始值“通常情况”下是数据类型的零值。

解析：将常量池内的符号引用替换为直接引用。

初始化：到了初始化阶段，才真正开始执行类中定义的 Java 初始化程序代码。主要是静态变量赋值动作和静态语句块（static{}）中的语句。

94、介绍下垃圾收集机制（在什么时候，对什么，做了什么）？

在什么时候?

在触发GC的时候，具体如下，这里只说常见的 Young GC 和 Full GC。

触发Young GC：当新生代中的 Eden 区没有足够空间进行分配时会触发Young GC。

触发Full GC：

- 当准备要触发一次Young GC时，如果发现统计数据说之前Young GC的平均晋升大小比目前老年代剩余的空间大，则不会触发Young GC而是转为触发Full GC。（通常情况）
- 如果有永久代的话，在永久代需要分配空间但已经没有足够空间时，也要触发一次Full GC。
- System.gc()默认也是触发Full GC。
- heap dump带GC默认也是触发Full GC。
- CMS GC时出现Concurrent Mode Failure会导致一次Full GC的产生。

对什么?

对那些JVM认为已经“死掉”的对象。即从GC Root开始搜索，搜索不到的，并且经过一次筛选标记没有复活的对象。

做了什么?

对这些JVM认为已经“死掉”的对象进行垃圾收集，新生代使用复制算法，老年代使用标记-清除和标记-整理算法。

95、GC Root有哪些?

在Java语言中，可作为GC Roots的对象包括下面几种：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象。
- 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。
- 本地方法栈中JNI（即一般说的Native方法）引用的对象。

96、垃圾收集有哪些算法，各自的特点?

标记 - 清除算法

首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。它的主要不足有两个：一个是效率问题，标记和清除两个过程的效率都不高；另一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

复制算法

为了解决效率问题，一种称为“复制”（Copying）的收集算法出现了，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为了原来的一半，未免太高了一点。

标记 - 整理算法

复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

分代收集算法

当前商业虚拟机的垃圾收集都采用“分代收集”（Generational Collection）算法，这种算法并没有什么新的思想，只是根据对象存活周期的不同将内存划分为几块。

一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。

在老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用标记—清理或者标记—整理算法来进行回收。