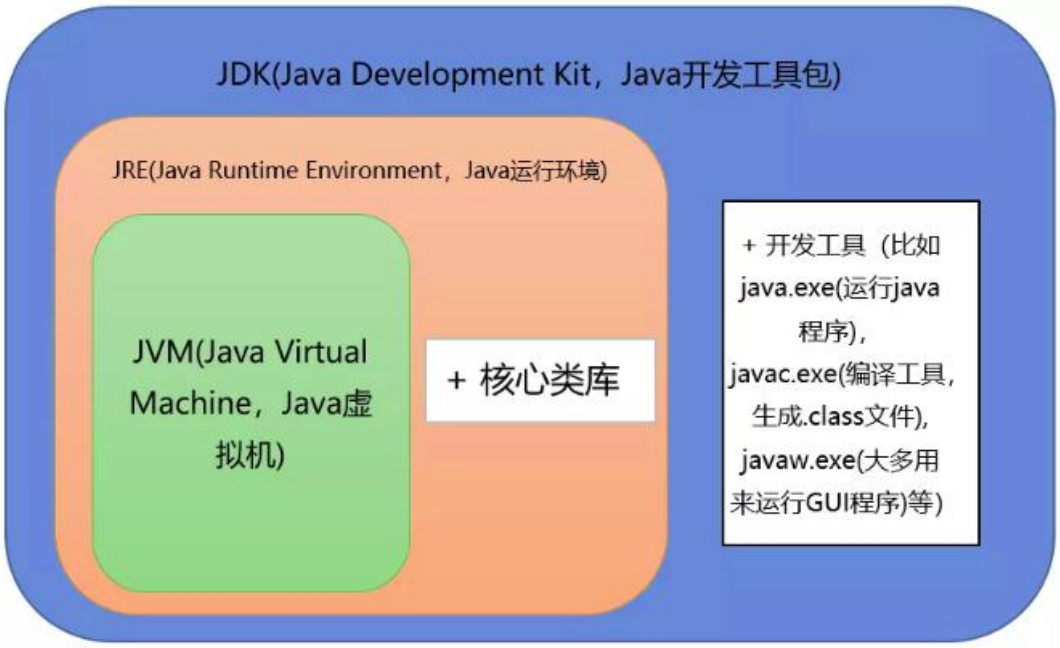
1、jvm、jre、jdk 关系图

JVM&JRE&JDK关系图



Jvm + 核心类库 == Jre　　Jre + 开发工具 == Jdk

2、跨平台性

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。

3、访问修饰符

分类

private : 在同一类内可见。使用对象：变量、方法。注意：不能修饰类（外部类）
default (即缺省，什么也不写，不使用任何关键字) : 在同一包内可见，不使用任何修饰符。使用对象：类、接口、变量、方法。
protected : 对同一包内的类和所有子类可见。使用对象：变量、方法。注意：不能修饰类（外部类）。
public : 对所有类可见。使用对象：类、接口、变量、方法

访问修饰符图

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

所以 protected 比什么都不写要更开放一点，因为它开放给子类，而 default 仅仅开放给同包。

4、super 和 this

3、引用父类构造函数

super（参数）：调用父类中的某一个构造函数（应该为构造函数中的第一条语句）。

this（参数）：调用本类中另一种形式的构造函数（应该为构造函数中的第一条语句）。

this与super的区别

super: 它引用当前对象的直接父类中的成员（用来访问直接父类中被隐藏的父类中成员数据或函数，基类与派生类中有相同成员定义时如：super.变量名 super.成员函数据名（实参）

this：它代表当前对象名（在程序中易产生二义性之处，应使用this来指明当前对象；如果函数的形参与类中的成员数据同名，这时需用this来指明成员变量名）

super()和this()类似,区别是，super()在子类中调用父类的构造方法，this()在本类内调用本类的其它构造方法。

super()和this()均需放在构造方法内第一行。

尽管可以用this调用一个构造器，但却不能调用两个。

this和super不能同时出现在一个构造函数里面，因为this必然会调用其它的构造函数，其它的构造函数必然也会有super语句的存在，所以在同一个构造函数里面有相同的语句，就失去了语句的意义，编译器也不会通过。

this()和super()都指的是对象，所以，均不可以在static环境中使用。包括：static变量,static方法，static语句块。

从本质上讲，this是一个指向本对象的指针, 然而super是一个Java关键字。

一般而言父类都要声明无参构造方法，因为子类执行构造方法时，都会默认执行 super()

5、java 中如何跳出多重循环

在Java中，要想跳出多重循环，可以在外面的循环语句前定义一个标号，然后在里层循环体的代码中使用带有标号的break 语句，即可跳出外层循环。例如：

```
public static void main(String[] args) {
    ok:
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            System.out.println("i=" + i + ",j=" + j);
            if (j == 5) {
                break ok;
            }

        }
    }
}
```

6、多态

一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

父类或接口定义的引用变量可以指向子类或具体实现类的实例对象。提高了程序的拓展性。

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）。

方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多态性（也称为后绑定）。

Java 实现多态有三个必要条件：继承、重写、向上转型。

继承：在多态中必须存在有继承关系的子类和父类。

重写：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法。

向上转型：在多态中需要将子类的引用赋给父类对象，只有这样该引用才能够具备技能调用父类的方法和子类的方法。

重写与重载

构造器（constructor）是否可被重写（override）

构造器不能被继承，因此不能被重写，但可以被重载。

重载（Overload）和重写（Override）的区别。重载的方法能否根据返回类型进行区分？

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：发生在同一个类中，方法名相同参数列表不同（参数类型不同、个数不同、顺序不同），与方法返回值和访问修饰符无关，即重载的方法不能根据返回类型进行区分

重写：发生在父子类中，方法名、参数列表必须相同，返回值小于等于父类，抛出的异常小于等于父类，访问修饰符大于等于父类（里氏代换原则）；如果父类方法访问修饰符为private则子类中就不是重写。

7、面向对象的 六原则一法则是什么？

1 单一职责原则：一个类只做它该做的事情，实现高内聚原则

2 开闭原则：对扩展开放，对修改关闭

3 依赖倒转原则：面向接口编程，声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型

4 里氏替换原则：任何时候都可以用子类型来替换掉父类型

5 接口隔离原则：接口要小而专，不要大而全

6 合成聚合复用原则：多用组合少用继承实现代码复用（将狗继承宠物改成狗类有一个宠物成员变量）

迪米特法则：低耦合原则

8、抽象类和接口的对比

抽象类是用来捕捉子类的通用特性的。接口是抽象方法的集合。从设计层面来说，抽象类是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

接口和抽象类各有优缺点，在接口和抽象类的选择上，必须遵守这样一个原则：

行为模型应该总是通过接口而不是抽象类定义，所以通常是优先选用接口，尽量少用抽象类。

选择抽象类的时候通常是如下情况：需要定义子类的行为，又要为子类提供通用的功能。

9、成员变量和局部变量的对比

成员变量：方法外部，类内部定义的变量
局部变量：类的方法中的变量。

作用域

成员变量：针对整个类有效。
局部变量：只在某个范围内有效。(一般指的就是方法,语句体内)

存储位置

成员变量：随着对象的创建而存在，随着对象的消失而消失，存储在堆内存中。
局部变量：在方法被调用，或者语句被执行的时候存在，存储在栈内存中。当方法调用完，或者语句结束后，就自动释放。

生命周期

成员变量：随着对象的创建而存在，随着对象的消失而消失
局部变量：当方法调用完，或者语句结束后，就自动释放。

初始值

成员变量：有默认初始值。

局部变量：没有默认初始值，使用前必须赋值。

使用原则

在使用变量时需要遵循的原则为：就近原则
首先在局部范围找，有就使用；接着在成员位置找。

10、构造方法的特性

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

11、在一个静态方法里调用非静态成员为什么是非法的

因为静态方法为类共享，非静态成员为对象专有

12、内部类

我认为两种：
成员内部类（定义在类的成员变量位置，其中用 static 修饰的叫静态内部类）；
局部内部类（定义在成员方法局部变量内部，含匿名内部类(new 类名/接口)）；

静态内部类

定义在类内部的静态类，就是静态内部类。

```
public class Outer {

    private static int radius = 1;

    static class StaticInner {
        public void visit() {
            System.out.println("visit outer static variable:" + radius);
        }
    }
}
```

静态内部类可以访问外部类所有的静态变量，而不可访问外部类的非静态变量；静态内部类的创建方式， `new 外部类.静态内部类()` ，如下：

```
Outer.StaticInner inner = new Outer.StaticInner();
inner.visit();
```

成员内部类

定义在类内部，成员位置上的非静态类，就是成员内部类。

```
public class Outer {

    private static  int radius = 1;
    private int count =2;

     class Inner {
        public void visit() {
            System.out.println("visit outer static variable:" + radius);
            System.out.println("visit outer variable:" + count);
        }
    }
}
```

成员内部类可以访问外部类所有的变量和方法，包括静态和非静态，私有和公有。成员内部类依赖于外部类的实例，它的创建方式 `外部类实例.new 内部类()` ，如下：

```
Outer outer = new Outer();
Outer.Inner inner = outer.new Inner();
inner.visit();
```

局部内部类

定义在方法中的内部类，就是局部内部类。

```
public class Outer {

    private  int out_a = 1;
    private static int STATIC_b = 2;

    public void testFunctionClass(){
        int inner_c =3;
        class Inner {
            private void fun(){
                System.out.println(out_a);
                System.out.println(STATIC_b);
                System.out.println(inner_c);
            }
        }
        Inner inner = new Inner();
        inner.fun();
    }
    public static void testStaticFunctionClass(){
        int d =3;
        class Inner {
            private void fun(){
                // System.out.println(out_a); 编译错误，定义在静态方法中的局部
                System.out.println(STATIC_b);
                System.out.println(d);
            }
        }
        Inner inner = new Inner();
        inner.fun();
    }
}
```

匿名内部类

匿名内部类就是没有名字的内部类，日常开发中使用的比较多。

```
public class Outer {

    private void test(final int i) {
        new Service() {
            public void method() {
                for (int j = 0; j < i; j++) {
                    System.out.println("匿名内部类" );
                }
            }
        }.method();
    }
 }
 //匿名内部类必须继承或实现一个已有的接口
 interface Service{
    void method();
}
```

除了没有名字，匿名内部类还有以下特点：

- 匿名内部类必须继承一个抽象类或者实现一个接口。
- 匿名内部类不能定义任何静态成员和静态方法。
- 当所在的方法的形参需要被匿名内部类使用时，必须声明为 final。
- 匿名内部类不能是抽象的，它必须要实现继承的类或者实现的接口的所有抽象方法。

内部类的优点

我们为什么要使用内部类呢？因为它有以下优点：

- 一个内部类对象可以访问创建它的外部类对象的内容，包括私有数据！
- 内部类不为同一包的其他类所见，具有很好的封装性；
- 内部类有效实现了“多重继承”，优化 java 单继承的缺陷。
- 匿名内部类可以很方便的定义回调。

内部类有哪些应用场景

一些多算法场合
解决一些非面向对象的语句块。
适当使用内部类，使得代码更加灵活和富有扩展性。
当某个类除了它的外部类，不再被其他的类使用时。

局部内部类和匿名内部类访问局部变量的时候，为什么变量必须要加上final呢？它内部原理是什么呢？

先看这段代码：

```
public class Outer {

    void outMethod(){
        final int a =10;
        class Inner {
            void innerMethod(){
                System.out.println(a);
            }

        }
    }
}
```

以上例子，为什么要加final呢？是因为**生命周期不一致**， 局部变量直接存储在栈中，当方法执行结束后，非final的局部变量就被销毁。而局部内部类对局部变量的引用依然存在，如果局部内部类要调用局部变量时，就会出错。加了final，可以确保局部内部类使用的变量与外层的局部变量区分开，解决了这个问题。

内部类相关，看程序说出运行结果

```
public class Outer {
    private int age = 12;

    class Inner {
        private int age = 13;
        public void print() {
            int age = 14;
            System.out.println("局部变量：" + age);
            System.out.println("内部类变量：" + this.age);
            System.out.println("外部类变量：" + Outer.this.age);
        }
    }

    public static void main(String[] args) {
        Outer.Inner in = new Outer().new Inner();
        in.print();
    }

}
```

运行结果：

```
局部变量：14
内部类变量：13
外部类变量：12
```

13、==和 equals 的区别

==：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() 方法来两个对象的内容相等；若它们的内容相等，则返回 true (即，认为这两个对象相等)。

举个例子：

```
public class test1 {
    public static void main(String[] args) {
        String a = new String("ab"); // a 为一个引用
        String b = new String("ab"); // b为另一个引用,对象的内容一样
        String aa = "ab"; // 放在常量池中
        String bb = "ab"; // 从常量池中查找
        if (aa == bb) // true
            System.out.println("aa==bb");
        if (a == b) // false，非同一对象
            System.out.println("a==b");
        if (a.equals(b)) // true
            System.out.println("aEQb");
        if (42 == 42.0) { // true
            System.out.println("true");
        }
    }
}
```

说明：

> String中的equals方法是被重写过的，因为object的equals方法是比较的对象的内存地址，而String的equals方法比较的是对象的值。
>
> 当创建String类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个String对象。

上述例子中有一个注释写错了，String aa = “ab”,这条语句“ab”这个实例早已经存在，从常量池返回即可，无需重新放入常量池。

14、hashcode 和 equals

首先明白一点：

两个对象 equals 相等，那么他们的 hashcode 一定相等；

两个对象的 hashcode 相等，但他们的 equals 不一定相等；

前者证明了重写 equals 方法，那么一定要重写 hashcode 方法

后者证明了 HashMap 用 hash 值找到了对应桶 ,但是还是要用 equals 方法判断是否相等，hashcode 方法减少了 equals 方法的调用，因为 hashcode 不等，那么没必要比较 equals 了。

15、Java 中只有值传递

只有值传递，而之所以有引用传递的效果，只不过是因为你传递的值是引用的拷贝，指的还是同一个地址而已。

example 1

```
public static void main(String[] args) {
    int num1 = 10;
    int num2 = 20;

    swap(num1, num2);

    System.out.println("num1 = " + num1);
    System.out.println("num2 = " + num2);
}

public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;

    System.out.println("a = " + a);
    System.out.println("b = " + b);
}
```

结果：

```
a = 20
b = 10
num1 = 10
num2 = 20
```

如上，传递的是值的拷贝

通过上面例子，我们已经知道了一个方法不能修改一个基本数据类型的参数，而对象引用作为参数就不一样，请看 example2.

example 2

```
public static void main(String[] args) {
        int[] arr = { 1, 2, 3, 4, 5 };
        System.out.println(arr[0]);
        change(arr);
        System.out.println(arr[0]);
    }

    public static void change(int[] array) {
        // 将数组的第一个元素变为0
        array[0] = 0;
    }
```

结果：

```
1
0
```

解析：

array 被初始化 arr 的拷贝也就是一个对象的引用，也就是说 array 和 arr 指向的时同一个数组对象。因此，外部对引用对象的改变会反映到所对应的对象上。

通过 example2 我们已经看到，实现一个改变对象参数状态的方法并不是一件难事。理由很简单，方法得到的是对象引用的拷贝，对象引用及其他的拷贝同时引用同一个对象。

很多程序设计语言（特别是，C++和Pascal)提供了两种参数传递的方式：值调用和引用调用。有些程序员（甚至本书的作者）认为Java程序设计语言对对象采用的是引用调用，实际上，这种理解是不对的。由于这种误解具有一定的普遍性，所以下面给出一个反例来详细地阐述一下这个问题。

example 3

```
public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student s1 = new Student("小张");
        Student s2 = new Student("小李");
        Test.swap(s1, s2);
        System.out.println("s1:" + s1.getName());
        System.out.println("s2:" + s2.getName());
    }

    public static void swap(Student x, Student y) {
        Student temp = x;
        x = y;
        y = temp;
        System.out.println("x:" + x.getName());
        System.out.println("y:" + y.getName());
    }
}
```

结果：

```
x:小李
y:小张
s1:小张
s2:小李
```

也就是说你要看方法参数是否引用类型，如果是，将改变实参的值。

但是 java 传递的还是值的拷贝，只不过的引用值的拷贝而已，引用值的拷贝，但是引用指的还是一个地址啊。

16、反射

在日常的第三方应用开发过程中，经常会遇到某个类的某个成员变量、方法或是属性是私有的或是只对系统应用开放 ,这时候就可以**利用 Java 的反射机制通过反射来获取所需的私有成员或是方法**

1.定义

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方法和属性**；这种动态获取信息以及动态调用对象方法的功能称为 java 语言的反射机制。

2.反射机制的相关类

类名	用途
Class类	代表类的实体，在运行的Java应用程序中表示类和接口
Field类	代表类的成员变量（成员变量也称为类的属性）
Method类	代表类的方法
Constructor类	代表类的构造方法

后三种类是基于 class 类获得的。

A.Class 类

- **获得类相关的方法**

方法	用途
asSubclass(Class<U> clazz)	把传递的类的对象转换成代表其子类的对象
Cast	把对象转换成代表类或是接口的对象
getClassLoader()	获得类的加载器
getClasses()	返回一个数组，数组中包含该类中所有公共类和接口类的对象
getDeclaredClasses()	返回一个数组，数组中包含该类中所有类和接口类的对象
forName(String className)	根据类名返回类的对象
getName()	获得类的完整路径名字
newInstance()	创建类的实例
getPackage()	获得类的包
getSimpleName()	获得类的名字
getSuperclass()	获得当前类继承的父类的名字
getInterfaces()	获得当前类实现的类或是接口

获得 class 类的三种方法

Java获取反射的三种方法

1.通过new对象实现反射机制 2.通过路径实现反射机制 3.通过类名实现反射机制

```
public class Student {
    private int id;
    String name;
    protected boolean sex;
    public float score;
}
public class Get {
    //获取反射机制三种方式
    public static void main(String[] args) throws ClassNotFoundException
        //方式一(通过建立对象)
        Student stu = new Student();
        Class classobj1 = stu.getClass();
        System.out.println(classobj1.getName());
        //方式二 (所在通过路径-相对路径)
        Class classobj2 = Class.forName("fanshe.Student");
        System.out.println(classobj2.getName());
        //方式三 (通过类名)
        Class classobj3 = Student.class;
        System.out.println(classobj3.getName());
    }
}
```

```
// 创建对象
static void reflectNewInstance() {
    try {
        Class<?> classBook = Class.forName("F:\\面试突击\\test\\src\\反射\\Book.java");
        Object objectBook = classBook.newInstance();
        Book book = (Book) objectBook;
        book.setName("Android进阶之光");
        book.setAuthor("刘望舒");
        System.out.println(book.toString());
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

- **获得类中属性相关的方法**

方法	用途
getField(String name)	获得某个公有的属性对象
getFields()	获得所有公有的属性对象
getDeclaredField(String name)	获得某个属性对象
getDeclaredFields()	获得所有属性对象

带declared能获取私有方法，不带的只能获取公有方法

- **获得类中注解相关的方法**

方法	用途
getAnnotation(Class<A> annotationClass)	返回该类中与参数类型匹配的公有注解对象
getAnnotations()	返回该类所有的公有注解对象
getDeclaredAnnotation(Class<A> annotationClass)	返回该类中与参数类型匹配的所有注解对象
getDeclaredAnnotations()	返回该类所有的注解对象

```
// 反射私有属性
public static void reflectPrivateField() {
    try {
        Class<?> classBook = Class.forName("F:\\面试突击\\test\\src\\反射\\Book.java");
        Object objectBook = classBook.newInstance();
        Field fieldTag = classBook.getDeclaredField( name: "TAG");
        fieldTag.setAccessible(true);
        String tag = (String) fieldTag.get(objectBook);
        System.out.println(tag);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

- **获得类中构造器相关的方法**

方法	用途
getConstructor(Class...<?> parameterTypes)	获得该类中与参数类型匹配的公有构造方法
getConstructors()	获得该类的所有公有构造方法
getDeclaredConstructor(Class...<?> parameterTypes)	获得该类中与参数类型匹配的构造方法
getDeclaredConstructors()	获得该类所有构造方法

- **获得类中方法相关的方法**

方法	用途
getMethod(String name, Class...<?> parameterTypes)	获得该类某个公有的方法
getMethods()	获得该类所有公有的方法
getDeclaredMethod(String name, Class...<?> parameterTypes)	获得该类某个方法
getDeclaredMethods()	获得该类所有方法

```
// 反射私有的构造方法
public static void reflectPrivateConstructor() {
    try {
        Class<?> classBook = Class.forName("F:\\面试突击\\test\\src\\反射\\Book.java");
        Constructor<?> declaredConstructorBook = classBook.getDeclaredConstructor(String.class,String.class);
        declaredConstructorBook.setAccessible(true);
        Object objectBook = declaredConstructorBook.newInstance( ...initargs: "Android开发艺术探索","任玉刚");
        Book book = (Book) objectBook;
        System.out.println(book.toString());
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
 // 反射私有方法
 public static void reflectPrivateMethod() {
     try {
         Class<?> classBook = Class.forName("F:\\面试突击\\test\\src\\反射\\Book.java");
         Method methodBook = classBook.getDeclaredMethod( name: "declaredMethod",int.class);
         methodBook.setAccessible(true);
         Object objectBook = classBook.newInstance();
         String string = (String) methodBook.invoke(objectBook, ...args: 0);
         System.out.println(string);
     } catch (Exception ex) {
         ex.printStackTrace();
     }
 }
```

- **类中其他重要的方法**

方法	用途
isAnnotation()	如果是注解类型则返回true
isAnnotationPresent(Class<? extends Annotation> annotationClass)	如果是指定类型注解类型则返回true
isAnonymousClass()	如果是匿名类则返回true
isArray()	如果是一个数组类则返回true
isEnum()	如果是枚举类则返回true
isInstance(Object obj)	如果obj是该类的实例则返回true
isInterface()	如果是接口类则返回true
isLocalClass()	如果是局部类则返回true
isMemberClass()	如果是内部类则返回true

B. Field 类

方法	用途
equals(Object obj)	属性与obj相等则返回true
get(Object obj)	获得obj中对应的属性值
set(Object obj, Object value)	设置obj中对应属性值

```
// 反射私有属性
public static void reflectPrivateField() {
    try {
        Class<?> classBook = Class.forName("F:\\面试突击\\test\\src\\反射\\Book.java");
        Object objectBook = classBook.newInstance();
        Field fieldTag = classBook.getDeclaredField( name: "TAG");
        fieldTag.setAccessible(true);
        String tag = (String) fieldTag.get(objectBook);
        System.out.println(tag);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

C. Method 类

方法	用途
invoke(Object obj, Object... args)	传递object对象及参数调用该对象对应的方法

```java
// 反射私有方法
public static void reflectPrivateMethod() {
    try {
        Class<?> classBook = Class.forName("F:\\面试突击\\test\\src\\反射\\Book.java");
        Method methodBook = classBook.getDeclaredMethod( name: "declaredMethod",int.class);
        methodBook.setAccessible(true);
        Object objectBook = classBook.newInstance();
        String string = (String) methodBook.invoke(objectBook, ...args: 0);
        System.out.println(string);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

```java
public class Book{
    private final static String TAG = "BookTag";

    private String name;
    private String author;

    @Override
    public String toString() {...}

    public Book() {}

    private Book(String name, String author) {...}

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getAuthor() { return author; }

    public void setAuthor(String author) { this.author = author; }

    private String declaredMethod(int index) {
        String string = null;
        switch (index) {
            case 0:
                string = "I am declaredMethod 1 !";
                break;
            case 1:
                string = "I am declaredMethod 2 !";
                break;
            default:
                string = "I am declaredMethod 1 !";
        }

        return string;
    }
}
```

D.Constructor 类

方法	用途
newInstance(Object... initargs)	根据传递的参数创建类的对象

```java
// 反射私有的构造方法
public static void reflectPrivateConstructor() {
    try {
        Class<?> classBook = Class.forName("F:\\面试突击\\test\\src\\反射\\Book.java");
        Constructor<?> declaredConstructorBook = classBook.getDeclaredConstructor(String.class,String.class);
        declaredConstructorBook.setAccessible(true);
        Object objectBook = declaredConstructorBook.newInstance( ...initargs: "Android开发艺术探索","任玉刚");
        Book book = (Book) objectBook;
        System.out.println(book.toString());
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

17、String、StringBuffer、StringBuilder

String 不是基本数据类型，只是因为觉得字符用数组太过繁琐，而设定的引用数据类型，底层仍是字符数组。

1) String不可变但不代表引用不可以变

```
String str = "Hello";
str = str + " World";
System.out.println("str=" + str);
结果：
str=Hello World
```

解析：

实际上，原来String的内容是不变的，只是str由原来指向"Hello"的内存地址转为指向"Hello World"的内存地址而已，也就是说多开辟了一块内存区域给"Hello World"字符串。

2) 通过反射是可以修改所谓的“不可变”对象

```
// 创建字符串"Hello World", 并赋给引用s
String s = "Hello World";

System.out.println("s = " + s); // Hello World

// 获取String类中的value字段
Field valueFieldOfString = String.class.getDeclaredField("value");

// 改变value属性的访问权限
valueFieldOfString.setAccessible(true);

// 获取s对象上的value属性的值
char[] value = (char[]) valueFieldOfString.get(s);

// 改变value所引用的数组中的第5个字符
value[5] = '_';

System.out.println("s = " + s); // Hello_World
结果：

s = Hello World
s = Hello_World
```

解析：

用反射可以访问私有成员， 然后反射出String对象中的value属性， 进而改变通过获得的value引用改变数组的结构。但是一般我们不会这么做，这里只是简单提一下有这个东西。

String s = new String(“xyz”);创建了几个字符串对象

两个对象，一个是静态区的"xyz"，一个是用new创建在堆上的对象。

```
String str1 = "hello"; //str1指向静态区
String str2 = new String("hello"); //str2指向堆上的对象
String str3 = "hello";
String str4 = new String("hello");
System.out.println(str1.equals(str2)); //true
System.out.println(str2.equals(str4)); //true
System.out.println(str1 == str3); //true
System.out.println(str1 == str2); //false
System.out.println(str2 == str4); //false
System.out.println(str2 == "hello"); //false
str2 = str1;
System.out.println(str2 == "hello"); //true
```

如何将字符串反转？

使用 StringBuilder 或者 stringBuffer 的 reverse() 方法。

示例代码：

```
// StringBuffer reverse
StringBuffer stringBuffer = new StringBuffer();
stringBuffer. append("abcdefg");
System. out. println(stringBuffer. reverse()); // gfedcba
// StringBuilder reverse
StringBuilder stringBuilder = new StringBuilder();
stringBuilder. append("abcdefg");
System. out. println(stringBuilder. reverse()); // gfedcba
```

数组有没有 length()方法？String 有没有 length()方法

数组没有 length()方法，有 length 的属性。String 有 length()方法。JavaScript中，获得字符串的长度是通过 length 属性得到的，这一点容易和 Java 混淆。

这也能说明 String 不可变的特点，如果可变的话，那么就会有 reverse 方法了，因为 reverse 方法就是在自身的基础上翻转元素顺序

String和StringBuffer、StringBuilder的区别是什么？String为什么是不可变的

可变性

String类中使用字符数组保存字符串，private final char value[]，所以string对象是不可变的。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，这两种对象都是可变的。

线程安全性

String中的对象是不可变的，也就可以理解为常量，线程安全。AbstractStringBuilder是StringBuilder与StringBuffer的公共父类，定义了一些字符串的基本操作，如expandCapacity、append、insert、indexOf等公共方法。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

如果要操作少量的数据用 = 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

18、int 和 Integer

Integer a= 127 与 Integer b = 127相等吗

对于对象引用类型：==比较的是对象的内存地址。
对于基本数据类型：==比较的是值。

如果整型字面量的值在-128到127之间，那么自动装箱时不会new新的Integer对象，而是直接引用常量池中的Integer对象，超过范围 a1==b1的结果是false

```
public static void main(String[] args) {
    Integer a = new Integer(3);
    Integer b = 3; // 将3自动装箱成Integer类型
    int c = 3;
    System.out.println(a == b); // false 两个引用没有引用同一对象
    System.out.println(a == c); // true a自动拆箱成int类型再和c比较
    System.out.println(b == c); // true

    Integer a1 = 128;
    Integer b1 = 128;
    System.out.println(a1 == b1); // false

    Integer a2 = 127;
    Integer b2 = 127;
    System.out.println(a2 == b2); // true
}
```

Integer 对象也会有字符串常量池的概念，但是只能装内容为-128 到 127 的对象。

Integer.valueOf 和 Integer.parseInt 和 new

Integer 区别及注意事项

1.System.out.println(127==127); //true , int type compare
2.System.out.println(128==128); //true , int type compare
3.System.out.println(new Integer(127) == new Integer(127)); //false, object compare
4.System.out.println(Integer.parseInt("128")==Integer.parseInt("128")); //true, int type compare
5.System.out.println(Integer.valueOf("127")==Integer.valueOf("127")); //true ,object compare, because IntegerCache return a same object
6.System.out.println(Integer.valueOf("128")==Integer.valueOf("128")); //false ,object compare, because number beyond the IntegerCache
7.System.out.println(Integer.parseInt("128")==Integer.valueOf("128")); //true , int type compare

int 整型常量比较时，== 是值比较，所以 1,2 返回 true。1，2 是值比较。

new Integer() 每次构造一个新的 Integer 对象，所以 3 返回 false。3 是对象比较。

Integer.parseInt 每次构造一个 int 常量，所以 4 返回 true。4 是值比较。

Integer.valueOf 返回一个 Integer 对象，默认在-128~127 之间时返回缓存中的已有对象（如果存在的话），所以 5 返回 true，6 返回 false。5，6 是对象比较。
第 7 个比较特殊，是 int 和 Integer 之间的比较，结果是值比较，**引用数据类型会自动拆箱，但是这句话只针对 Integer 和 int，因为 String 没有基本数据类型。**

19、线程安全和线程不安全的单例模式？

```
 public class Singleton {
      private Singleton(){}  // 构造方法声明为私有，所以只能生成一个对象
      private static Singleton instance = new Singleton();
      public static Singleton getInstance(){
           return instance;
      }
}
public class Singleton {
      private static Singleton instance = null;
      private Singleton() {}
      public static synchronized Singleton getInstance(){
           if (instance == null) instance  =  new Singleton();
           return instance;
      }
}
```

单例设计模式：保证类在内存中只有一个对象。

单例设计模式分为两种，一类是饿汉式，上来就 new 对象，用空间换时间，二类是懒汉式，先判断是否有单例对象，如果没有再 new 对象，判断需要时间，所以用的是时间换空间。

明显是饿汉式更好，但是懒汉式可能会用于面试。

在多线程访问时，饿汉式不会创建多个对象，而懒汉式可能会创建多个对象，因为饿汉式创建对象的方法私有，而懒汉式创建对象时的方法不私有，线程可能多次访问（待考证）。

在 jdk 手册中，如果一个类没有看到构造方法，说明该类的构造方法很有可能被私有化了。说明该类可能是单例设计模式下的类，不允许实例化对象，只允许用类名调用返回。

20、日志优先级

一般而言都是 INFO，可以打印警告和错误．用到 DEBUG 就比较细了．

log4j定义了8个级别的log（除去OFF和ALL，可以说分为6个级别），优先级从高到低依次为：OFF、FATAL、ERROR、WARN、INFO、DEBUG、TRACE、 ALL。

ALL 最低等级的，用于打开所有日志记录。

TRACE designates finer-grained informational events than the DEBUG.Since:1.2.12，很低的日志级别，一般不会使用。

DEBUG 指出细粒度信息事件对调试应用程序是非常有帮助的，主要用于开发过程中打印一些运行信息。

INFO 消息在粗粒度级别上突出强调应用程序的运行过程。打印一些你感兴趣的或者重要的信息，这个可以用于生产环境中输出程序运行的一些重要信息，但是不能滥用，避免打印过多的日志。

WARN 表明会出现潜在错误的情形，有些信息不是错误信息，但是也要给程序员的一些提示。

ERROR 指出虽然发生错误事件，但仍然不影响系统的继续运行。打印错误和异常信息，如果不想输出太多的日志，可以使用这个级别。

FATAL 指出每个严重的错误事件将会导致应用程序的退出。这个级别比较高了。重大错误，这种级别你可以直接停止程序了。

OFF 最高等级的，用于关闭所有日志记录。

21、java 定义二维数组的方法：

1、动态初始化
数据类型 数组名 [ ][ ] = new 数据类型[m][n]
数据类型 [ ][ ] 数组名 = new 数据类型[m][n]
数据类型 [ ] 数组名 [ ] = new 数据类型[m][n]
举例：int [ ][ ] arr=new int [5][3]; 也可以理解为“5 行 3 例”
2、 静态初始化
数据类型 [ ][ ] 数组名 = {{元素 1,元素 2....},{元素 1,元素 2....},{元素 1,元素 2....}.....};
举例：int [ ][ ] arr={{22,15,32,20,18},{12,21,25,19,33},{14,58,34,24,66},}

22、正则表达式的() [] {}有不同的意思。

1. () 是为了**提取匹配的字符串**。表达式中有几个()就有几个相应的匹配字符串。
例: (\s*)表示连续空格的字符串。(0-9) 匹配 '0-9′ 本身。
2. []是定义**匹配的字符范围**。
例:[\s*]表示空格或者*号。[0-9]* 匹配数字（注意后面有 *，可以为空）[0-9]+ 匹配数字（注意后面有 + ,不可以为空）比如 [a-zA-Z0-9] 表示相应位置的字符要匹配英文字符和数字。

3. {}一般用来表示**匹配的长度**

例: {1-9} 写法错误。比如 \s{3} 表示匹配三个空格，\s{1,3}表示匹配一到三个空格。

[0-9]{0,9} 表示长度为 0 到 9 的数字字符串

23、nextInt 和 nextLine 引发的问题

nextInt：此方法**只读取整型数值**，并且在读取输入后**把光标留在本行**

next：**读取输入直到遇见空格**。此方法不能读取被空格分隔开的内容，并且在读取输入后把光标留在本行

nextLine：读取包括空格在内的输入，而且还会读取行尾的换行字符\n，读取完成后**光标被放在下一行**

如果用了 nextInt 后，又用 nextLine 时，nextLine 会将 nextInt 后的换行符给接收，那么这样的话，接收的数组接受的数据会少了最后一个，因为前面接收了一个换行符。所以要先吧这个换行符给接收：sc.nextLine；

```
public class Test {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        sc.nextLine(); // 读掉数值后面的换行符
        String[] arr = new String[n];
        for (int i = 0; i < n; i++) {
            arr[i] =  sc.nextLine();
        }

        System.out.println(Arrays.toString(arr));
    }
}
```

同理，如果用 hasNextLine 作为循环跳出条件，那么循环会永远成立，跳不出循环，因为每一行后面都会有一个换行符.

24、BigInteger、BigDecimal

A:BigInteger 的概述

- 可以让超过 Integer 范围内的数据进行运算

B:构造方法

- public BigInteger(String val)

C:成员方法

- public BigInteger add(BigInteger val)
- public BigInteger subtract(BigInteger val)
- public BigInteger multiply(BigInteger val)
- public BigInteger divide(BigInteger val)
- public BigInteger[] divideAndRemainder(BigInteger val)

A:BigDecimal 的概述

- 由于在运算的时候，float 类型和 double 很容易丢失精度。
- 所以，为了能精确的表示、计算浮点数，Java 提供了 BigDecimal。
- 不可变的、任意精度的有符号十进制数。

B:构造方法

- public BigDecimal(String val)

C:成员方法

- public BigDecimal add(BigDecimal augend)
- public BigDecimal subtract(BigDecimal subtrahend)
- public BigDecimal multiply(BigDecimal multiplicand)
- public BigDecimal divide(BigDecimal divisor)

25、多线程

进程线程间通讯原理

1、进程是什么？

是具有一定独立功能的程序、它是**系统进行资源分配和调度的一个独立单位**，重点在系统调度和单独的单位，也就是说进程是可以独立运行的一段程序。

2、线程又是什么？

线程进程的一个实体，是 CPU 调度和分派的基本单位，他是比进程更小的能独立运行的基本单位，线程自己**基本上不拥有系统资源**。在运行时，只是暂用一些计数器、寄存器和栈。**是程序执行的最小单位**。

从三个角度来剖析二者之间的区别

1、调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位。

2、并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可以并发执行。

3、拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源。

1. 线程间通讯原理

1、共享变量

最常见的就是共享一个对象，如银行存款取款共享一个 Acount 对象：

银行存取款实例：

```
package com.xujishou;

public class Acount {

    private int money;

    public Acount(int money) {
        this.money = money;
    }

    public synchronized void getMoney(int money) {
        // 注意这个地方必须用while循环，因为即便再存入钱也有可能比取的要少
        while (this.money < money) {
            System.out.println("取款: " + money + " 余额: " + this.money
                    + " 余额不足，正在等待存款.....");
            try {
                wait();
            } catch (Exception e) {
            }
        }
        this.money = this.money - money;
        System.out.println("取出: " + money + " 还剩余: " + this.money);

    }

    public synchronized void setMoney(int money) {

        try {
            Thread.sleep(10);
        } catch (Exception e) {
        }
        this.money = this.money + money;
        System.out.println("新存入: " + money + " 共计: " + this.money);
        notify();
    }

    public static void main(String args[]) {
        Acount Acount = new Acount(0);
        Bank b = new Bank(Acount);
        Consumer c = new Consumer(Acount);
        new Thread(b).start();
        new Thread(c).start();
    }
}
```

```java
// 存款类
class Bank implements Runnable {
    Acount Acount;

    public Bank(Acount Acount) {
        this.Acount = Acount;
    }

    public void run() {
        while (true) {
            int temp = (int) (Math.random() * 1000);
            Acount.setMoney(temp);
        }
    }

}

// 取款类
class Consumer implements Runnable {
    Acount Acount;

    public Consumer(Acount Acount) {
        this.Acount = Acount;
    }

    public void run() {
        while (true) {
            int temp = (int) (Math.random() * 1000);
            Acount.getMoney(temp);
        }
    }
}
```

只要存的钱大于要取的钱，就一直取随机数量，每隔 10 秒钟存一次随机数量的钱，所以这个程序不会停止。

2、wait/notify 机制

为了实现线程通信，我们可以使用 Object 类提供的 wait()、notify()、notifyAll()三个方法，这三个方法在使用 synchronized 时使用：

定义 Account 类

```
public class Account {
    private String accountNo;
    private double balance;
    //标识账户中是否有存款的旗标
    private boolean flag=false;

    public Account() {
        super();
    }

    public Account(String accountNo, double balance) {
        super();
        this.accountNo = accountNo;
        this.balance = balance;
    }


    public synchronized void draw (double drawAmount){

        try {
              if(!flag){
              this.wait();
             }else {
                 //取钱
                 System.out.println(Thread.currentThread().getName()+" 取钱:"+drawAmount);
                 balance=balance-drawAmount;
                 System.out.println("余额 : "+balance);
                 //将标识账户是否已有存款的标志设为false
                 flag=false;
                 //唤醒其它线程
                 this.notifyAll();
             }
            } catch (Exception e) {
                e.printStackTrace();
        }
    }

   public synchronized void deposit(double depositAmount){
      try {
              if(flag){
                this.wait();
              }
              else{
                  System.out.println(Thread.currentThread().getName()+"存钱"+depositAmount);
                  balance=balance+depositAmount;
                  System.out.println("账户余额为: "+balance);
                  flag=true;
                  //唤醒其它线程
                  this.notifyAll();
              }
        } catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
        }
   }

}
```

定义取钱线程类

取钱线程类：

```java
public class DrawThread implements Runnable {

    private Account account;
    private double drawAmount;


    public DrawThread(Account account, double drawAmount) {
        super();
        this.account = account;
        this.drawAmount = drawAmount;
    }

    public void run() {
        for(int i=0;i<100;i++){
            account.draw(drawAmount);
        }
    }
}
```

定义存钱线程类

存钱线程类：

```java
public class depositThread implements Runnable{
    private Account account;
    private double depositAmount;

    public depositThread(Account account, double depositAmount) {
        super();
        this.account = account;
        this.depositAmount = depositAmount;
    }


    public void run() {
    for(int i=0;i<100;i++){
        account.deposit(depositAmount);
      }
    }

}
```

最后我们测试一下这个取钱和存钱的操作！

```java
public  class TestDraw {

    public static void main(String[] args) {
        //创建一个账户
        Account account=new Account();
        new Thread(new DrawThread(account, 800),"取钱者").start();
        new Thread(new depositThread(account, 800),"存款者甲").start();
        new Thread(new depositThread(account, 800),"存款者乙").start();
        new Thread(new depositThread(account, 800),"存款者丙").start();

    }

}
```

其实也用到了共享变量对象 account，只是利用 wait，notifyAll 使其交叉运行，使得不能连续存钱和取钱。上面也用到了 wait 和 notify，所以二者常常配合使用。

```
存款者甲存钱800.0
账户余额为：800.0
取钱者 取钱:800.0
余额：0.0
存款者丙存钱800.0
账户余额为：800.0
取钱者 取钱:800.0
余额：0.0
存款者甲存钱800.0
账户余额为：800.0
取钱者 取钱:800.0
余额：0.0
存款者丙存钱800.0
账户余额为：800.0
取钱者 取钱:800.0
余额：0.0
存款者甲存钱800.0
账户余额为：800.0
取钱者 取钱:800.0
余额：0.0
存款者丙存钱800.0
账户余额为：800.0
取钱者 取钱:800.0
余额：0.0
存款者甲存钱800.0
账户余额为：800.0
取钱者 取钱:800.0
余额：0.0
存款者丙存钱800.0
账户余额为：800.0
取钱者 取钱:800.0
余额：0.0
存款者甲存钱800.0
账户余额为：800.0
取钱者 取钱:800.0
余额：0.0
```

3、Lock/Condition 机制

如何程序不使用 synchronized 关键字来保持同步，而是直接适用 Lock 对像来保持同步，则系统中不存在隐式的同步监视器对象，也就不能使用 wait()、notify()、notifyAll()来协调线程的运行.

当使用 LOCK 对象保持同步时，JAVA 为我们提供了 Condition 类来协调线程的运行。

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Account {

    //显示定义Lock对象
    private final Lock lock=new ReentrantLock();
    //获得指定Lock对象对应的条件变量
    private final  Condition con=lock.newCondition();

    private String accountNo;
    private double balance;
    //标识账户中是否有存款的旗标
    private boolean flag=false;

    public Account() {
        super();
    }

    public Account(String accountNo, double balance) {
        super();
        this.accountNo = accountNo;
        this.balance = balance;
    }
```

```
public void draw (double drawAmount){

    //加锁
    lock.lock();
    try {
        if(!flag){
//          this.wait();
          con.await();
        }else {
            //取钱
            System.out.println(Thread.currentThread().getName()+" 取钱:"+drawAmount);
            balance=balance-drawAmount;
            System.out.println("余额："+balance);
            //将标识账户是否已有存款的标志设为false
            flag=false;
            //唤醒其它线程
//          this.notifyAll();
            con.signalAll();
        }
      } catch (Exception e) {
        e.printStackTrace();
    }
      finally{
        lock.unlock();
      }
  }
```

```
public void deposit(double depositAmount){
    //加锁
    lock.lock();
    try {
        if(flag){
//          this.wait();
          con.await();
        }
        else{
            System.out.println(Thread.currentThread().getName()+"存钱"+depositAmount);
            balance=balance+depositAmount;
            System.out.println("账户余额为："+balance);
            flag=true;
            //唤醒其它线程
//          this.notifyAll();
            con.signalAll();
        }
    } catch (Exception e) {
        // TODO: handle exception
        e.printStackTrace();
    }finally{
        lock.unlock();
    }
  }

}
```

4、管道

管道流是 JAVA 中线程通讯的常用方式之一，基本流程如下：

1) 创建管道输出流 PipedOutputStream pos 和管道输入流 PipedInputStream pis

2) 将 pos 和 pis 匹配，pos.connect(pis);

3) 将 pos 赋给信息输入线程，pis 赋给信息获取线程，就可以实现线程间的通讯了

```java
public class testPipeConnection {

    public static void main(String[] args) {
        /**
         * 创建管道输出流
         */
        PipedOutputStream pos = new PipedOutputStream();
        /**
         * 创建管道输入流
         */
        PipedInputStream pis = new PipedInputStream();
        try {
            /**
             * 将管道输入流与输出流连接 此过程也可通过重载的构造函数来实现
             */
            pos.connect(pis);
        } catch (IOException e) {
            e.printStackTrace();
        }
        /**
         * 创建生产者线程
         */
        Producer p = new Producer(pos);
        /**
         * 创建消费者线程
         */
        Consumer1 c1 = new Consumer1(pis);
        /**
         * 启动线程
         */
        p.start();
        c1.start();
    }
}
```

```java
/**
 * 生产者线程(与一个管道输入流相关联)
 *
 */
class Producer extends Thread {
    private PipedOutputStream pos;

    public Producer(PipedOutputStream pos) {
        this.pos = pos;
    }

    public void run() {
        int i = 0;
        try {
            while(true)
            {                    3s写入一个数字
            this.sleep(3000);
            pos.write(i);
            i++;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```java
/**
 * 消费者线程(与一个管道输入流相关联)
 *
 */
class Consumer1 extends Thread {
    private PipedInputStream pis;

    public Consumer1(PipedInputStream pis) {
        this.pis = pis;
    }

    public void run() {
        try {                                                    只要有写，就直接读出来
            while(true)
            {
            System.out.println("consumer1:"+pis.read());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

输出结果：

```
consumer1:0
consumer1:1
consumer1:2
consumer1:3
......
```

管道流虽然使用起来方便，但是也有一些缺点

1）管道流只能在两个线程之间传递数据

线程 consumer1 和 consumer2 同时从 pis 中 read 数据，当线程 producer 往管道流中写入一段数据后，**每一个时刻只有一个线程能获取到数据**，并不是两个线程都能获取到 producer 发送来的数据，因此一个管道流只能用于两个线程间的通讯。不仅仅是管道流，其他 IO 方式都是一对一传输。

2）管道流**只能实现单向发送**，如果要两个线程之间互通讯，则需要两个管道流

Java 提供管道功能，实现管道通信的类有两组：PipedInputStream 和 PipedOutputStream 或者是 PipedReader 和 PipedWriter。管道通信主要用于不同线程间的通信。

一个 PipedInputStream 实例对象必须和一个 PipedOutputStream 实例对象进行连接而产生一个通信管道。PipedOutputStream 向管道中写入数据，PipedIntputStream 读取 PipedOutputStream 向管道中写入的数据。一个线程的 PipedInputStream 对象能够从另外一个线程的 PipedOutputStream 对象中读取数据。

2.进程间通讯原理

（1）管道（Pipe）：

管道可用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信。一般用于线程间通讯。

Linux 管道的实现机制

从本质上说，管道也是一种文件，但它又和一般的文件有所不同，管道可以克服使用文件进行通信的两个问题，具体表现为：·

1）限制管道的大小。实际上，管道是一个固定大小的缓冲区。在 Linux 中，该缓冲区的大小为 1 页，即 4K 字节，使得它的大小不象文件那样不加检验地增长。使用单个固定缓冲区也会带来问题，比如在写管道时可能变满，当这种情况发生时，随后对管道的 write()调用将默认地被阻塞，等待某些数据被读取，以便腾出足够的空间供 write()调用写。

2）读取进程也可能工作得比写进程快。当所有当前进程数据已被读取时，管道变空。当这种情况发生时，一个随后的 read()调用将默认地被阻塞，等待某些数据被写入，这解决了 read()调用返回文件结束的问题

注意：

从管道读数据是一次性操作，数据一旦被读，它就从管道中被抛弃，释放空间以便写更多的数据。

管道的结构

在 Linux 中，管道的实现并没有使用专门的数据结构，而是借助了文件系统的 file 结构和 VFS 的索引节点 inode。

过将两个 file 结构指向同一个临时的 VFS 索引节点，而这个 VFS 引节点又指向一个物理页面而实现的。

（2）命名管道（named pipe）：

命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。命名管道在文件系统中有对应的文件名。命名管道通过命令 mkfifo 或系统调用 mkfifo 来创建。

1）首先要创建一个管道文件，这一点 Java 做不到，我们要借助 C/C++中的 mkfifo()函数来实现。

以下代码创建并打开两个管道，一个用于读取输入，一个用于输出：

```
char name1[1024];
char name2[1024];
sprintf(name1,"./process_in");
sprintf(name2,"./process_out");
if(mkfifo(name1,0666 )<0)
{
 printf("\n error when mkfifo");
}
if(mkfifo(name2,0666 )<0)
{
 printf("\n error when mkfifo");
}

int fd1, fd2;
if ((fd1 = open (name1, O_RDWR ))<0)
   {
      perror("Could not open named pipe.");
   }
if ((fd2 = open (name2, O_RDWR ))<0)
   {
      perror("Could not open named pipe.");
   }
```

这是查看当前目录，可以看到两个 Pipe 文件，它和普通文件不同，具有 p 属性，表明是一个管道文件。

```
prw-rw-r-- 1 ligh ligh    0 Dec 23 10:21 process_in
prw-rw-r-- 1 ligh ligh    0 Dec 23 10:21 process_out
drwxrwxrwx 2 ligh ligh 4096 Dec 17 18:59
-rwxrw-rw- 1 ligh ligh   45 Nov 22 14:08 start.sh
```

2）Java 中，使用文件读写的方式打开这两个文件，即可进行读写。

```
String namedPipe1=workdir+"/process_in";
String namedPipe2=workdir+"/process_out";

File pipe1 = new File(namedPipe1);
File pipe2 = new File(namedPipe2);

BufferedReader reader = new BufferedReader(new FileReader(pipe2));
//DataInputStream reader = new DataInputStream(new FileInputStream(pipe2));
BufferedWriter writer = new BufferedWriter(new FileWriter(pipe1));

writer.write("test");

        char [] buf = new char[8];
reader.read(buf);
```

从管道1写数据，从管道2读数据

（3）信号（Signal）：

信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送 信号给进程本身；linux 除了支持 Unix 早期信号语义函数 sigal 外，还支持语义符合 Posix.1 标准的信号函数 sigaction（实际上，该函数是基于 BSD 的，BSD 为了实现可靠信号机制，又能够统一对外接口，用 sigaction 函数重新实现了 signal 函数）。

（4）消息（Message）队列：

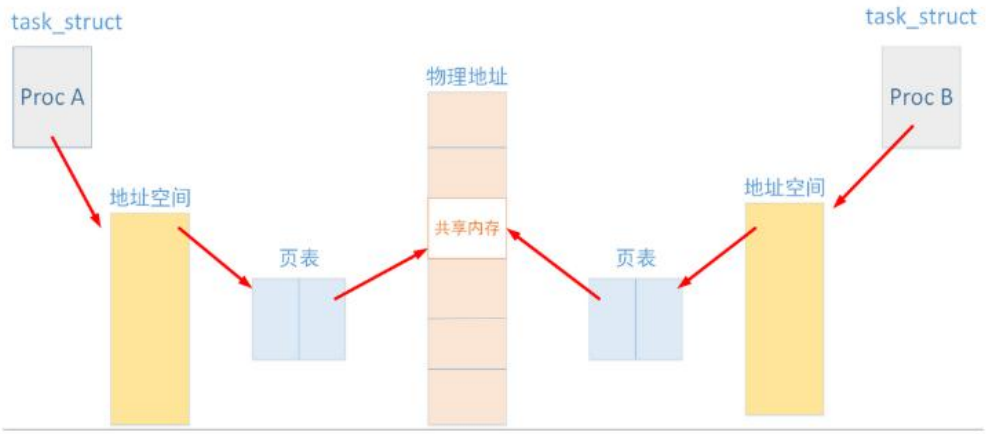
消息队列是消息的链接表，包括 Posix 消息队列 system V 消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺

（5）共享内存：

使得多个进程可以访问同一块内存空间，是最快的可用 IPC 形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。

在 Linux 中，每个进程都有属于自己的进程控制块（PCB）和地址空间（Addr Space），并且都有一个与之对应的页表，负责将进程的虚拟地址与物理地址进行映射，通过内存管理单元（MMU）进行管理。两个不同的虚拟地址通过页表映射到物理空间的同一区域，它们所指向的这块区域即共享内存。

共享内存的通信原理示意图：



对于上图我的理解是：当两个进程通过页表将虚拟地址映射到物理地址时，在物理地址中有一块共同的内存区，即共享内存，这块内存可以被两个进程同时看到。这样当一个进程进行写操作，另一个进程读操作就可以实现进程间通信。但是，我们要确保一个进程在写的时候不能被读，因此我们使用信号量来实现同步与互斥。

对于一个共享内存，实现采用的是引用计数的原理，当进程脱离共享存储区后，计数器减一，挂架成功时，计数器加一，只有当计数器变为零时，才能被删除。当进程终止时，它所附加的共享存储区都会自动脱离。

（6）内存映射（mapped memory）：

内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它。

（7）信号量（semaphore）：

主要作为进程间以及同一进程不同线程之间的同步手段。

（8）套接口（Socket）：

更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由 Unix 系统的 BSD 分支开发出来的，但现在一般可以移植到其它类 Unix 系统上：Linux 和 System V 的变种都支持套接字。

服务端代码：

```
public class TcpServer {
    public static void main(String[] args) throws Exception {
        ServerSocket server = new ServerSocket( port: 9091);
        try {
            Socket client = server.accept();
            try {                                                    获取输入流缓冲
                BufferedReader input =  new BufferedReader(new InputStreamReader(client.getInputStream()));
                boolean flag = true;
                int count = 1;                                                    socket建立输入流连接，可以获取数据了

                while (flag) {
                    System.out.println("客户端要开始说话了，这是第" + count + "次");
                    count++;

                    String line = input.readLine();    从输入流中一行一行获取数据
                    System.out.println("客户端说" + line);

                    if (line.equals("exit")) {
                        flag = false;
                        System.out.println("客户端不想玩了！");
                    } else {
                        System.out.println("客户端说：  " + line);
                    }

                }
            } finally {
                client.close();
            }                       关闭socket

        } finally {
            server.close();
        }                   关闭服务端socket
    }
}
```

客户端代码

多线程创建方法

一般来说是有五种方法创建线程：

1、直接实现 Runnable 接口，重写 run 方法，称为 MyThread 类

new Thread (new MyThread) .start()

2、继承 Thread 类，直接重写 run 方法

new Thread ( ) ，其中还是有一个 target 的 Runnable 对象

3、匿名内部类

```
new Thread( new Runnable() {
@Override
      public void run() {
     System.out.println("匿名内部类的方式创建线程");
      }
}).start();
```

4、实现 Callable 接口，重写其中的 run 方法，但是却要这样：

```
//FutureTask 继承自 RunnableFuture，而后者继承自 Runnable
FutureTask<Integer> futureTask = new FutureTask<>(new C());
// 传递的仍是一个 Runnable 对象
Thread thread2 = new Thread(futureTask);
thread2.start();
```

5、采用线程池，就是存储上述方式创建的线程。

综上：其实可以总结为一个方法，就是最终都要经过 Thread 这个类，传递给 Thread 类一个 Runnable 对象 ,然后调用 Thread 的 start 方法 ,有一个除外 ,无需传递 Runnable 对象，因为继承 Thread 类，自身携带 target Runnable 对象，其余的在构造函数时会调用 init 函数，该函数会将传入的 Runnabe 对象覆盖 target 对象。

关于你明明重写的是 run 方法，为什么调用的确实 start 方法。

（1）start()方法是来自线程的，真正的实现了多线程的运行，而不用等到 run()方法执行完毕之后再执行下面的代码段：

通过调用 Thread 类的 **start()方法来启动一个线程，这时此线程是处于就绪状态**，并没有运行，**其实就是将该线程对象放入了线程组中 group 中。**

然后通过此 Thread 类调用方法 run()来完成其运行操作的，这里方法 run()称为线程体，它包含了要执行的这个线程的内容，**Run 方法运行结束，此线程终止**，而 CPU 再运行其它线程。

（2）而 run()方法只是类中我们自己定义的方法，是一种普通的方法，顺序执行，只有在执行完这一段代码之后才会继续执行下去，而**如果直接用 Run 方法，这只是调用一个方法而已，程序中依然只有主线程--这一个线程**，其程序执行路径还是只有一条，这样就没有达到写线程的目的。

用户态和内核态的理解和区别

首先明白的是这两种状态都是针对进程来说的，而进程是操作系统创建的，而每个进程所能做的工作都不一样，为了将这些工作分轻重缓急、重要程度，给进程建立了特权级的概念，最关键的工作交给特权级最高的线程去做，这样可以做到集中管理，减少有限资源的访问和冲突。而 inter x86 的 cpu 一共有 4 个级别 0-3，其中 0 的特权级别最高。而用户态的进程就是特权级别为 3，内核态的进程就是特权级别为 0.

一般而言，自己刚开始创建的进程都会处于用户态，但如果要执行文件操作，网络数据发送等操作必须调用 write\send 等系统调用，那么这个时候需要将进程从用户态切换到内核态，因为用户态的进程的特权不允许访问本该内核态才能访问的代码和数据。当完成系统调用后，又会切换为用户态。

总结一下：

内核态：运行操作系统程序，操作硬件

用户态：运行用户程序

差别在于：

处于用户态执行时，进程所能访问的内存空间和对象受到限制，其所处于占有的处理器是可被抢占的

处于内核态执行时，则能访问所有的内存空间和对象，且所占有的处理器是不允许被抢占的。

26.输入输出流

1. 字符流和字节流的对比

继承关系如下：

字节流：inputStream --> FileInputStream --> BufferInputStream、DataInputStream
outputStream --> FileOutputStream --> BufferOutputStream、DataOutputSteam

字符流：inputStreamReader--> FileReader --> BufferFileReader
InputSteamWriter --> FileWriter、Printwriter --> BufferFileWriter

FileInputStream 在读取文件的时候，就是一个一个 byte 地读取，
DataInputStream 则是在 FileInputStream 的一个轻量级的包装类，
BufferInputStream 则是自带缓冲区，默认缓冲区大小为 8X1024
通俗地讲就是：
FileInputStream 在读取文件的时候，一滴一滴地把水从一个缸复制到另外一个缸
DataInputStream 则是一瓢一瓢地把水从一个缸复制到另外一个缸
BufferInputStream 则是一桶一桶地把水从一个缸复制到另外一个缸。
后两者，虽然底层封装的都是 File 流，但是明显 buffer 更厉害。

一句话概括 Buffer 流的精髓：
读数据的时候不需要频繁跟磁盘交互，先从 buffer 读，没有数据了，再从磁盘里拿；
写数据的时候不需要频繁和磁盘交互，先往 buffer 里面写，写满了，再刷到磁盘。
而且二者的 buffer 是共享的，这就导致了 buffer 里面都是热点数据，大大减少了 IO 操作。
注意，buffer 里面存的都是原始的 File 输入输出流。

FileInputStream 与 FileReader 区别：

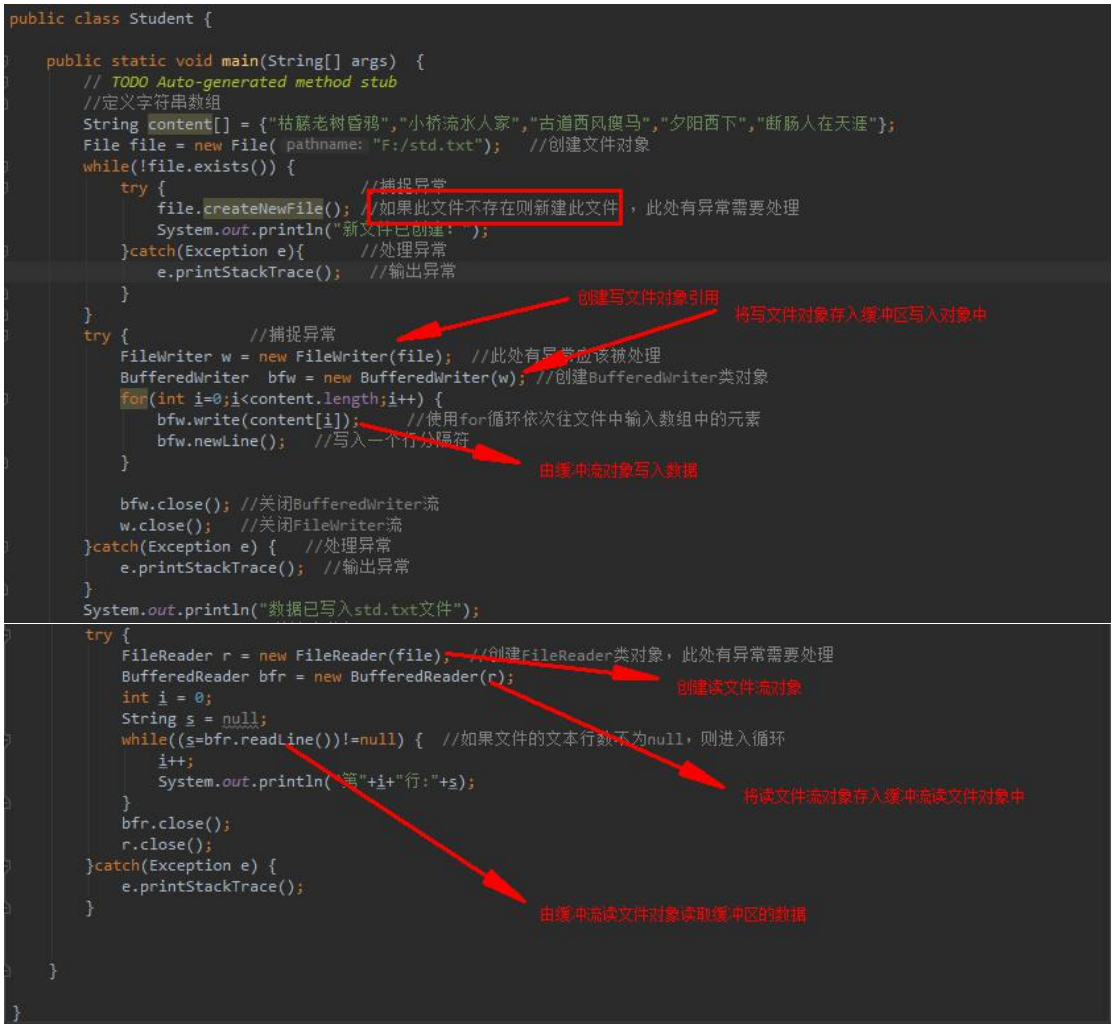
FileInputStream 是字节流，FileReader 是字符流，用字节流读取中文的时候，可能会出现乱码，而用字符流则不会出现乱码，而且用字符流读取的速度比字节流要快；

2.带缓冲的输入输出流（字符流流为例）

BufferedReader 和 BufferedWriter 类

写入文件过程：字符数据→BufferedWriter→FileWriter→文件

从文件中读出过程：文件→FileReader→BufferedReader→目的地



27、锁的分类介绍（指的都是锁的思想）

Java 中就两种加锁的方式：
一种是使用 synchronized 关键字，另一种是使用 Lock 接口的实现类
前者就相当于自动挡，可以满足一切驾驶需求；但是你如果想要更高级的操作，比如玩漂移和各种高级的骚操作，那么就需要手动挡，也就是后者；
而 Synchronized 在经过每个版本的各种优化后，效率也变得很高了，只是使用起来没有 Lock 接口的实现类那么方便

1. 悲观锁和乐观锁

乐观锁：就是很乐观，每次去拿数据的时候都认为别人不会修改，如果想要更新数据，就会在更新之前检查在读取至更新这段时间别人有没有修改过这个数据 ,如果修改过 ,则重新读取，再次尝试更新，循环往复，直到更新成功，**当然也允许失败的线程放弃更新操作。这就是 CAS**

（compare and swap）的概念，是实现乐观锁的基础，**所以乐观锁是没有上锁的，所以乐观锁，也称为无锁编程。**

乐观锁允许多个线程同时读取（因为根本没有加锁操作），如果更新数据的话，有且仅有一个线程成功更新数据，并导致其他线程需要回滚重试，CAS 利用 CPU 指令，从硬件层面保证了原子性，以达到类似于锁的效果。

悲观锁：就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次都在拿数据的时候上锁。想获取数据的线程，必须等到悲观锁释放，再去获得锁，然后再获取数据

悲观锁阻塞事务，乐观锁回滚重试

适用场景：乐观锁适用于写比较少的情况下，即冲突真的很少发生的场景，这样可以省去锁的开销，极大系统的整个吞吐量。如果经常发生冲突，应用就会不断重试，这样反而降低了性能，用悲观锁更好。

2. 偏向锁、轻量级锁(自旋锁)、重量级锁(Sychronized 为例)

前面讲了 Synchronized 锁已经经过了升级优化 :就是不是直接上来就是悲观的重量级锁，而是从偏向锁 --> 轻量级锁 --> 重量级锁的转变。

也就是说使用 sychronized 关键字同步代码块的时候，上的是偏向锁，字面意思是**“偏向第一个获取它的线程的锁”。线程执行完同步代码块之后，并不会制动释放偏向锁**，当有线程第二次想进入同步代码块时，线程会判断持有锁的线程是否就是自己（持有锁的线程 ID 在锁的对象头里存储），如果是则正常往下执行。由于之前没有释放，这里不需要重新加锁，如果从头到尾都是一个线程在使用锁，很明显，偏向锁几乎没有额外的开销，这也说明适用 synchronized 关键字的锁是可冲入锁，

但是如果有第二个线程加入锁竞争，**偏向锁会转换为轻量级锁，也就是自旋锁。没有抢到锁的线程会进行自旋操作**，即在一个循环中不停判断是否可以获取锁，就是通过 CAS 操作修改对象头里的锁标志位，将锁的类型变为轻量级锁，如果抢到了锁，就把当前锁的所有者信息改为自己；

假如我们获取到锁的线程操作时间很长，那么其他线程的自旋时间较长，会发生忙等现象，如果忙等时间不长，可以用短时间的忙等获取线程在用户态和内核态之间切换的开销。

但是，**如果 JVM 中的 PC 计数器记录的自旋次数超过最大自旋次数，那么轻量级锁会升级为重量级锁**，依然是通过 CAS 修改锁标志位，表示达到了重量级锁级别，如果后续线程尝试获取锁，**发现被暂用的锁是重量级锁，那么直接将自己挂起，不会进行自旋操作。**

在 JVM 张，synchronized 锁只能逐渐升级，而不允许降级。其中轻量级锁可以认为是加锁版的乐观锁。

3. 可重入锁（递归锁）

可以重新进入的锁，**即允许同一个线程多次获取同一把锁**，java 中应该都是可重入锁，不可重入锁意义不大。

4. 公平锁和非公平锁

如果多个线程申请一把公平锁，那么锁的想成释放锁时，**先申请的先得到，很公平，如果是非公平锁，后申请的线程可能先获得锁**，是随机还是其他方式，取决于实现算法。
对于 ReentrantLock 类来说，可以通过构造函数指定公平还是非公平。

```
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

而对于 synchronized 锁而言，只能是非公平锁。
如果没有特殊要求，优先使用非公平锁，因为其吞吐量更大。

5. 可中断锁

可以相应中断的锁，java 中并没有提供任何可以直接中断线程的方法，只提供了中断机制，即 Thread.interrupt()方法，但是不能直接终止线程，线程会自行选择在合适的时间点以自己的方式相应中断。
Synchronized 是不可中断锁， 而 Lock 的实现类都是可中断锁，只要实现 Lock 接口的 Interruptibly 方法就好

```
public interface Lock {

    void lock();

    void lockInterruptibly() throws InterruptedException;

    boolean tryLock();

    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;


    void unlock();

    Condition newCondition();
}
```

6. 读写锁

读写锁其实是一对锁，一个读锁（共享读锁）和一个写锁（互斥锁、排它锁）
如果我读取值是为了更新他（sql 中的 for update 就是这个意思），那么加锁的时候直接加写锁，持有写锁时，别的线程，不论是读还是写，都需要等待；如果读取数据仅仅是为了前端展示，那么加锁时就明确加一个读锁，其他线程如果也要加读锁，不需要等待，可以直接获取（**读锁计数器加 1**）。
虽然读写锁感觉像乐观锁，但是读写锁是悲观锁策略，**因为读写锁没有再更新前判断值有没有被修改过，而是在加锁前决定应用写锁还是读锁。**乐观锁特指无锁编程。

7. synchronized 详解

https://blog.csdn.net/zhengwangzw/article/details/105141484

Synchronized 有三种作用范围：

1、在静态方法上加锁
2、在非静态方法上加锁
3、在代码块上加锁

注意：锁只能加在对象上。

作用范围	锁对象
非静态方法	当前对象 => this
静态方法	类对象 => SynchronizedSample.class （一切皆对象，这个是类对象）
代码块	指定对象 => lock （以上面的代码为例）

```
public class SynchronizedSample {

    private final Object lock = new Object();

    private static int money = 0;
        //非静态方法
    public synchronized void noStaticMethod(){
        money++;
    }
        //静态方法
    public static synchronized void staticMethod(){
        money++;
    }

    public void codeBlock(){
        //代码块
        synchronized (lock){
            money++;
        }
    }
}
```

synchronized 在代码块上加锁是通过 monitorenter 和 monitorexit 指令实现，在静态方法和非静态方法上加锁是在方法的 flags 中加入 ACC_SYNCHRONIZED 。JVM 运行方法时检查方法的 flags，遇到同步标识开始启动前面的加锁流程，在方法内部遇到 monitorenter 指令开始加锁。

JVM 怎么通过 sychronized 在对象上加锁，保证多线程访问竟态资源安全。

总的来说：

1、jdk6 之前，都是重量级锁

2、Jdk6 之后，引入了偏向锁和轻量级锁，

(1) 原因 1、因为 Sun 程序员发现大部分程序大多数时间都不会发生多个线程同时访问竞态资源的情况，每次线程都加锁解锁，每次这么搞都要操作系统在用户态和内核态之间来回切（加锁后，线程处于内核态，不允许抢占该线程占有的资源），太耗性能了。线程获取资源的时候，会发生系统调用，从用户态转化为内核态，假如加了重量锁，线程会重返用户态，来回切换会消耗性能，而假设加了轻量锁，线程会先保持内核态，当自旋操作达到一定数量后，将轻量锁标志位置为重量锁标志位，然后再重返用户态。

(2) 原因 2、同一个线程加锁解锁的重复率很高，这也是偏向锁的由来。

偏向锁适合低并发时，同一个线程总是获得该资源，为避免重复的加锁解锁操作，而保持的一直占用，当高并发时，用的都是轻量级锁，因为线程执行的时间可能很短亦或是很少发生线

程争抢 ,所以加轻量级锁就好 ,使进程保持短暂的内核态比用户态和内核态切换消耗的性能要少，当线程执行时间比较长或者并发度很高时，那么使用重量级锁较好。

总结：

并发度极低：使用偏向锁，避免频繁加锁解锁

并发度中等，线程执行时间较短：使用轻量级锁，使进程保持短暂的内核态，避免频繁的用户态和内核态切换。

并发度高或线程执行时间较长：使用重量级锁，防止大量线程执行较长时间的轻量级锁的自旋操作。

加锁具象化理解：

在一个类中，全局变量可以看做是竟态资源，如 StringBuffer 当中的 value 数组， 而对其中的 append 方法加 sychronized 关键字，实例化一个 StringBuffer 对象，在多个线程中调用 StringBuffer 对象的 append 方法，锁住的就是当前对象，不允许多个线程同时调用 append 方法，那么也就不会同时操作 value 数组。也就是说，**sychronized 锁住的其实是操作资源的代码块，而不是真的资源。**

```
public static void main(String[] args) throws InterruptedException {
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < 10; i++){
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int j = 0; j < 1000; j++){
                    stringBuilder.append("a");
                }      多个线程只用到了一个对象，所以只有一个对象头，只有一个mark word
            }
        }).start();
    }

    Thread.sleep(100);
    System.out.println(stringBuilder.length());
}
```

8.java 对象头

Java 对象头又是什么呢？我们以 Hotspot 虚拟机为例，Hopspot 对象头主要包括两部分数据：Mark Word（标记字段） 和 class Pointer（类型指针）。

Mark Word：默认存储对象的 HashCode，分代年龄和锁标志位信息。这些信息都是与对象自身定义无关的数据 ,所以 Mark Word 被设计成一个非固定的数据结构以便在极小的空间内存存储尽量多的数据。它会根据对象的状态复用自己的存储空间，也就是说在运行期间 Mark Word 里存储的数据会随着锁标志位的变化而变化。

class Point：对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

32 位虚拟机对象头分配情况

锁状态	25 bit		4 bit	1 bit	2 bit
	23 bit	2 bit		是否偏向锁	锁标志位
无状态	标记对象的 hashcode		分代年龄	0	01
偏向锁	线程ID	epoch	分代年龄	1	01
轻量级锁	指向栈中锁记录指针				00
重量级锁	指向重量级锁的指针				10
GC标记	无				11

无状态也就是无锁的时候，对象头开辟 25bit 的空间用来存储对象的 hashcode ，4bit 用于存放分代年龄，1bit 用来存放是否偏向锁的标识位，2bit 用来存放锁标识位为 01

偏向锁 中划分更细，还是开辟 25bit 的空间，其中 23bit 用来存放线程 ID，2bit 用来存放 epoch，4bit 存放分代年龄，1bit 存放是否偏向锁标识， 0 表示无锁，1 表示偏向锁，锁的标识位还是 01

轻量级锁中直接开辟 30bit 的空间存放指向栈中锁记录的指针，2bit 存放锁的标志位，其标志位为 00

重量级锁中和轻量级锁一样，30bit 的空间用来存放指向重量级锁的指针，2bit 存放锁的标识位，为 11

GC 标记开辟 30bit 的内存空间却没有占用，2bit 空间存放锁标志位为 11。

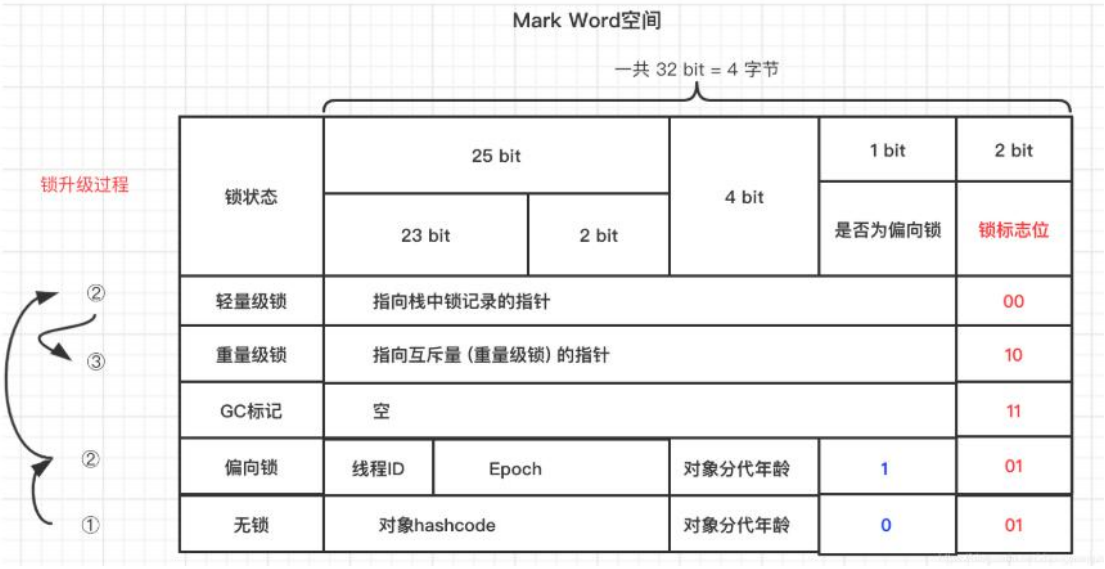
其中无锁和偏向锁的锁标志位都是 01，只是在前面的 1bit 区分了这是无锁状态还是偏向锁状态。

首先，java 对象包含三个部分：

java 对象头：

对象头分为二个部分，Mard Word 和 Klass Word，👇列出了详细说明：

对象头结构	存储信息-说明
Mard Word	存储对象的hashCode、锁信息或分代年龄或GC标志等信息
Klass Word	存储指向对象所属类（元数据）的指针，JVM通过这个确定这个对象属于哪个类



以上是 32 位 JVM 背景下的对象头中的 mark word ,锁标志位代表该对象是否获得该资源，获的该资源加的锁是什么，01 是初始化时的标志位，表示无锁，年龄为 1 时，表示获得该资源，加了偏向锁，00 是轻量级锁，10 是重量级锁，11 表示该对象要被回收。

下面是简化后的 Mark Word

bitfields				tag bits
hash	age	0		01
ptr to lock record				00
ptr to heavyweight monitor				10
				11
thread id	epoch	age	1	01

state
unlocked
lightweight locked
inflated
marked for gc
biasable

对象实例数据：

成员变量等

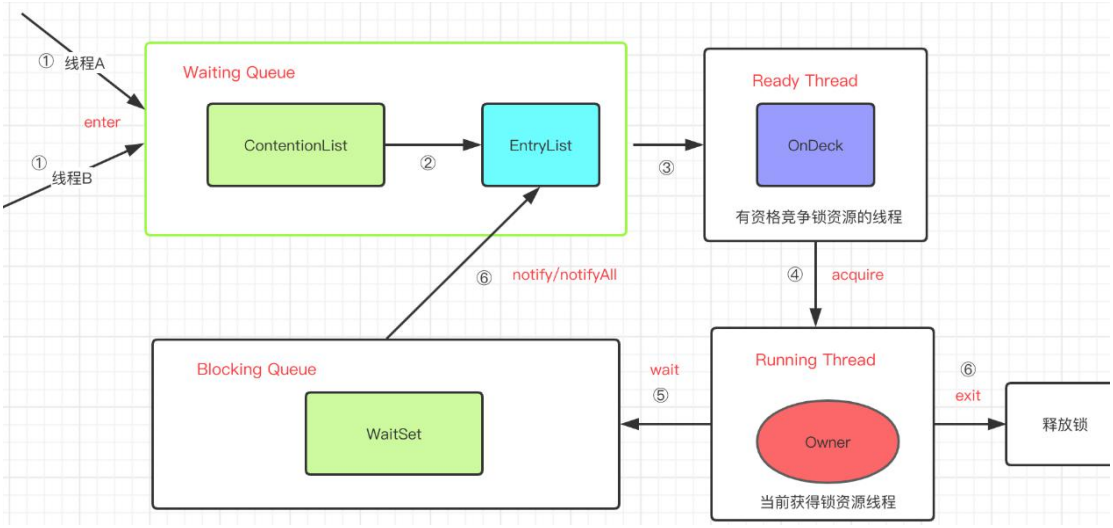
对齐填充：

JVM 要求对象占用的空间必须是 8 的倍数，方便内存分配（以字节为最小单位分配），因此这部分就是用于填满不够的空间凑数用的。

9.Monitor

每个对象都有一个与之关联的 Monitor 对象

```
//👇图详细介绍重要变量的作用
ObjectMonitor() {
    _header       = NULL;
    _count        = 0;    // 重入次数
    _waiters      = 0,    // 等待线程数
    _recursions   = 0;
    _object       = NULL;
    _owner        = NULL;  // 当前持有锁的线程
    _WaitSet      = NULL;  // 调用了 wait 方法的线程被阻塞 放置在这里
    _WaitSetLock  = 0 ;
    _Responsible  = NULL ;
    _succ         = NULL ;
    _cxq          = NULL ;
    FreeNext      = NULL ;
    _EntryList    = NULL ; // 等待锁 处于block的线程 有资格成为候选资源的线程
    _SpinFreq     = 0 ;
    _SpinClock    = 0 ;
    OwnerIsThread = 0 ;
  }
```



总结

Java 中的各种锁基本都是悲观锁，乐观锁的话只有 java.uti.concurrent.atomic 下面的原子类是通过乐观锁实现的。

例如：

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}
```

可以看到，在一个循环里不断 CAS，知道成功为止。

26、