一、java基础：面向对象

1.1、面向对象的三大特征

- 1) 封装性，把相关的数据封装成一个“类”组件
- 2) 继承性，是子类自动共享父类属性和方法，这是类之间的一种关系
- 3) 多态性，增强软件的灵活性和重用性

1.1.1、类

- 1) Java语言最基本单位就是类，类似于类型。
- 2) 类是一类事物的抽象。
- 3) 可以理解为模板或者设计图纸。

1.1.2、对象

每个对象具有三个特点：对象的状态，对象的行为和对象的标识。

- 1) 对象的状态用来描述对象的基本特征。
- 2) 对象的行为用来描述对象的功能。
- 3) 对象的标识是指对象在内存中都有一个唯一的地址值用来和其他对象区分开来。
- 4) 类是一类事物的抽象，对象是具体的实现。

1.1.3、类和对象关系

- 1) 计算机语言来怎么描述现实世界中的事物的? 属性 + 行为
- 2) 那怎么通过java语言来描述呢?通过类来描述一类事物,把事物的属性当做成员变量,把行为当做方法

1.1.4、创建对象时内存发生了什么?

> ①、在栈内存中开辟一块空间，存放引用类型p，并把p压入栈低；先进后出 ②、在堆内存中开辟一块空间，存放phone对象，new出来的对象 ③、完成对象的初识化，并赋予默认值 ④、把初始化完毕的对象赋予唯一的地址值 ⑤、把地址值交给引用变量类型p存放

1.2、封装

1.2.1、封装是隐藏对象的属性和实现细节,仅仅对外提供公共的访问方式,比如类和方法

好处:

- 1) 提高安全性
- 2) 提高重用性

1.2.2、封装的使用:

①、通过private关键字(权限修饰符)来修饰成员变量/成员方法

被修饰的成员就实现了私有化,访问权限只能在本类中访问

②、如何访问私有资源?

关于成员变量:

- setXxx – 对外提供公共的设置值方式
- getXxx – 对外提供公共的获取值方式

关于成员方法:

- 把私有方法放在公共方法里供外界调用即可

1.2.3、创建对象流程(Class c = new Class())，使用new关键字

> - 1. 把Class.class文件加载进内存
> - 2. 在栈内存中，开辟空间，存放引用变量c
> - 3. 在堆内存中，开辟空间，存放Class对象
> - 4. 对成员变量进行默认的初始化
> - 5. 对成员变量进行显示初始化
> - 6. 执行构造方法（如果有构造代码块，就先执行构造代码块再执行构造方法）
> - 7. 堆内存完成
> - 8. 把堆内存的地址值赋值给变量c ， c就是一个引用变量，引用了Class对象的地址值

1.2.4、匿名对象

没有名字的对象，是对象的简化表示形式。

使用场景：

当被调用的对象只调用一次时（多次会创建多个对象浪费内存）

1.3、构造方法

1.3.1、概念

> - 构造方法是一种特殊的方法,它是一个与类同名且没有返回值类型的方法;
> - 对象创建就是通过构造方法完成的,主要功能是完成对象的创建或者对象的初始化;
> - 当类创建对象(实例化)时,会自动调用构造方法;
> - 构造方法与普通方法一样也可以重载，但是不能被继承和重写

1.3.2、形式

与类同名,且没有返回值类型,可以含参也可以不含参

```
修饰符   方法名   ([参数列表]){
•      注意:方法名与类名一样
•      代码。。。
}
```

TIPS:关于构造函数怎么记忆

特点:方法名与类名相同,且没有返回值类型

执行时机:创建对象时立即执行

默认会创建无参构造,但是,如果自定了含参构造,默认的无参构造会被覆盖,注意要手动添加哦

构造代码块与局部代码块

构造代码块特点

- 1) 位置: 在类的内部,在方法的外部
- 2) 作用: 用于抽取构造方法中的共性代码
- 3) 执行时机: 每次调用构造方法前都会调用构造代码块
- 4) 注意事项: 构造代码块优先宇构造方法加载

局部代码块

- 1) 位置: 在方法里面的代码块
- 2) 作用: 通常用于控制变量的作用范围,出了花括号就失效
- 3) 注意事项: 变量的作用范围越小越好,成员变量会存在线程安全的问题

变量

概念

可以改变的数，称为变量。在Java语言中，所有的**变量在使用前必须声明。**

一般通过"变量类型 变量名 = 变量值 ;"这三部分来描述一个变量。如:int a = 3 ;

变量的使用原则：就近原则,即尽量控制变量的使用范围到最小

局部变量

位置: 定义在方法里或者局部代码块中

注意: 必须手动初始化来分配内存.如:int i = 5;或者int i; i = 5;

作用域: 也就是方法里或者局部代码块中,方法运行完内存就释放了

成员变量

位置: 定义在类里方法外

注意: 不用初始化,也会自动被初始化成默认值

作用域: 整个类中,类消失了,变量才会释放

继承

概念

1. 继承是面向对象最显著的一个特征；
2. 继承是从已有的类中派生出新的类,新的类能吸收已有类的数据属性和行为,并扩展新的能力；
3. Java继承是会用已存在的类的定义作为基础建立新类的技术；
4. 新类的定义可以增加新的数据或者新的功能,也可以使用父类的功能,但不能选择性的继承父类(超类/基类)；
5. 这种继承使得复用以前的代码非常容易,能够大大的缩短开发的周期,降低开发费用；

特点

- 子类拥有父类对象的所有属性和方法(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无法是访问的，只是拥有；
- 子类可以拥有自己的属性和方法；即子类可以对父类进行拓展
- 子类不满意父类的方法，可以重写父类中的方法
- 构造方法无法继承

super

可以通过这个关键字使用父类的内容,Super代表的是父类的一个引用对象

注意:在构造方法里,默认有super(); 且出现的调用位置在第一行

重写@Override

1. 继承以后,子类就拥有了父类的功能
2. 在子类中,可以添加子类特有的功能,也可以修改父类的原有功能
3. 子类中方法的签名与父类完全一样时,会发生覆盖/复写的现象
4. 格式要求:方法的返回值 方法名 参数列表 要完全一致,就方法体是重写的

TIPS: 父类的私有方法不能被重写,子类在重写父类方法时,修饰符

子类重写父类方法时,子类修饰符要大于等于父类修饰符的权限

this和super的区别

this代表本类对象的引用

```
class Father3{ this.xxx } //this -- Father3 this = new Father3();
```

super代表父类对象的引用

```
class Father3{ super.xxx } //this -- Father3 super = new Father3();
```

就相当于创建了一个父类对象

> 1. this可以在两个变量名相同时,用于区分成员变量和局部变量
> 2. this 可以在本类的构造方法之间调用,位置必须是第一条语句,注意,不能相互调用,会死循环
> 3. super是发生了继承关系以后,子类如果想用父类的功能,可以通过super调用
> 4. 如果发生了重写,还想用父类的功能,需要使用super来调用
> 5. Super在调用父类构造方法时,必须出现在子类构造方法的第一条语句,而且如果父类中没有提供无参构造,子类可以通过super来调用父类其他的含参构造

重载Overload 与重写Override的区别

> 1. 重载: 是指在一个类中的现象,是指**一个类中有很多同名的方法,但是方法的参数列表不同**
> 2. 重写: 是指发生了继承关系以后(两个类),子类去修改父类原有的功能,**子类中有一个方法签名(返回值类型 方法名(参数列表))和父类的一模一样**
> 3. 重载的意义: 是为了方便外界对方法进行调用,什么样的参数程序都可以找到对应的方法来执行,体现的是程序的灵活性
> 4. 重写的意义:是在不修改源码的前提下,进行功能的修改和拓展(OCP原则:面向修改关闭,面向拓展开放)
> 5. 重写要求方法的修饰符: 子类权限 >= 父类的权限

static

特点

1. 可以修饰成员变量与成员方法
2. 随着类的加载而加载，优先于对象加载
3. 只加载一次，就会一直存在，不再开辟新空间, 直到类消失才一起消失
4. 静态资源也叫做类资源,全局唯一,被全局所有对象共享
5. 可以直接被类名调用
6. 静态资源只能调用静态资源，非静态可以随意调用
7. static不能和this或者super共用，因为有static时可能还没有对象

静态代码块、构造代码块、局部代码块

静态代码块格式

```
static {}
```

`静态资源随着类的加载而加载，并且只被加载一次，一般用于项目的初始化`

特点: 被static修饰,位置在类里方法外

三种代码块的比较

1) 静态代码块：在类加载时就加载，并且只被加载一次，一般用于项目的初始化

2) 构造代码块：在创建对象时会自动调用，每次创建对象都会被调用,提取构造共性

3) 局部代码块：方法里的代码块,限制局部变量的范围

总结

①、代码之间的执行顺序:

`静态代码块-->构造代码块-->构造方法-->局部代码块`

②、为什么是这样的顺序呢?

1. 静态代码块要优先于对象进行加载
2. 是随着类的加载而加载到内存中的
3. 只加载一次,并且一直存在,直到类消失,它才会消失

③、**每个元素的作用:**

1. 静态代码块:专门用来完成一些需要第一时间加载并且只加载一次的资源
2. 构造代码块:创建对象时才会触发,用来提取构造方法中的共性内容
3. 构造方法:创建对象时调用,用来创建对象,在构造代码块执行后执行
4.)局部代码块:调用所在的方法时才会调用,用来控制变量的作用范围

final

1) 是java提供的一个关键字

2) final是最终的意思

3) final可以修饰类，方法，成员变量

初衷：java出现继承后，子类可以更改父类的功能，当父类功能不许子类改变时,可以利用final关键字修饰父类。

特点

> 1. 被final修饰的类，不能被继承，也就是没有子类,它自己就是最终类
> 2. 被final修饰的方法，不能被重写
> 3. 被final修饰的变量是个常量，值不能被改变，但可以被继承
> 4. 常量的定义形式：final 数据类型 常量名 = 值

多态

`优势:可以把不同的子类对象都当作父类来看，可以屏蔽不同子类对象之间的差异，写出通用的代码，做出通用的编程，统一调用标准。`

多态可以让我们不用关心某个对象到底具体是什么类型,就可以使用该对象的某些方法，提高了程序的可扩展性和可维护性

特点

> 1. 多态的前提1：是继承和重写
> 2. `父类引用指向子类对象`,如：Animal a = new Cat();
> 3. 多态中，编译看左边，运行看右边
> 4. 父类提供的功能才能用,子类特有的功能用不了

重写

子类重写父类中的方法--对父类中的代码功能修改

重写:`方法签名保持一致(返回值类型 方法名(参数列表) )` && 子类权限修饰符 >= 父类权限修饰符

使用方式

> 1. 成员变量: 使用的是父类的
> 2. 成员方法: 使用的是父类的声明，子类的实现，重写则使用子类的
> 3. 静态成员: 随着类的加载而加载，谁调用就返回谁的
> 4. `注意!!!静态资源属于类,不存在重写现象,只是两个类中有同样声明的方法而已,不属于重写`

静态变量和实例变量的区别

在语法定义上的区别：**静态变量前要加static关键字，而实例变量前则不加。**

在程序运行时的区别：**实例变量属于某个对象的属性，必须创建了实例对象，其中的实例变量才会被分配空间，才能使用这个实例变量。**

解释：静态变量不属于某个实例对象，而是属于类，所以也称为类变量，只要程序加载了类的字节码，不用创建任何实例对象，静态变量就会被分配空间，静态变量就可以被使用了。总之，实例变量必须创建对象后才可以通过这个对象来使用，静态变量则可以直接使用类名来引用。

向上转型和向下转型

在JAVA中，继承是一个重要的特征，通过extends关键字，子类可以复用父类的功能，如果父类不能满足当前子类的需求，则子类可以重写父类中的方法来加以扩展。在应用中就存在着两种转型方式，分别是：**向上转型和向下转型。**

比如：父类Parent,子类Child

向上转型：父类的引用指向子类对象Parent p=new Child();

- 说明：向上转型时，子类对象当成父类对象，只能调用父类的功能，如果子类重写了父类的方法就根据这个引用指向调用子类重写方法。

向下转型(较少)：子类的引用的指向子类对象，过程中必须要采取到强制转型。

```
Parent p = new Child();//向上转型，此时，p是Parent类型

Child c = (Child)p;//此时，把Parent类型的p转成小类型Child

//其实，相当于创建了一个子类对象一样，可以用父类的，也可以用自己的
```

- 说明：向下转型时，是为了方便使用子类的特殊方法，也就是说当子类方法做了功能拓展，就可以直接使用子类功能。

异常

异常的继承结构(体现多态特点)

> **Throwable : 顶级父类**
>
> **--Error : 系统错误,无法修复**
>
> **--Exception : 可以修复的错误**
>
> **-- RunTimeException**
>
> **-- ClassCastException**
>
> **-- ClassNotFoundException**

异常处理

当程序中遇到了异常,通常有两种处理方式: `捕获或者向上抛出`

当一个方法抛出异常,调用位置可以不做处理继续向上抛出,也可以捕获处理异常

1) 捕获方式:

try{

// 可能会出现异常的代码

}catch(异常类型 异常名){

//如果捕获到异常的对应解决方案

}

2) 抛出方式:

在会发生异常的方法上添加代码: `throws` 异常类型

如果方法抛出异常,那么谁调用,谁需要解决(继续抛出/捕获解决)

例如:

```
public static void main(String[] args) throws Exception{}//最好不要抛给main，它之后没人解决
```

权限修饰符速查表

修饰符	同类	同包	子类	不同包(无关类)
public	YES	YES	YES	YES
protected	YES	YES	YES	NO
default	YES	YES	NO	NO
private	YES	NO	NO	NO

TIPS:default是表示不写修饰符,默认,如果写default单词来修饰会报错

抽象类

特点

1. abstract 可以修饰方法或者类
2. 被abstarct修饰的类叫做抽象类，被abstract修饰的方法叫做抽象方法
3. **抽象类可以有成员常量和成员变量**
4. **抽象类中可以没有抽象方法**，可以都是普通方法：如果不想让外界创建本类对象,可以把普通类声明成抽象类
5. **如果类中有抽象方法,那么该类必须定义为一个抽象类**
6. **子类继承了抽象类以后,要么还是一个抽象类,要么就把父类的所有抽象方法都重写**
7. **抽象类中可以有构造方法；父类的构造方法要优先于子类执行；抽象类不可以创建对象(实例化)；抽象类中存在的构造方法不是为了创建本类对象时调用； 而是为了创建子类对象时使用**

抽象方法要求子类继承后必须重写。

那么，abstract关键字不可以和哪些关键字一起使用呢？以下关键字，在抽象类中。用是可以用的，只是没有意义了。

> **private**：被私有化后，子类无法重写，与abstract相违背。
>
> **static**：静态优先于对象存在。而abstract是对象间的关系，存在加载顺序问题。
>
> **final**：被final修饰后，无法重写，与abstract相违背。

接口

- Java里面由于不允许多重继承,所以如果要实现多个类的功能,则可以通过实现多个接口来实现,Java接口和Java抽象类代表的就是抽象类型,就是我们需要提出的抽象层的具体表现
- OOP面向对象编程,如果要提高程序的复用率,增加程序的可维护性,可扩展性,就必须是面向接口编程,面向抽象的变成,正确的使用接口/抽象类这些抽象类型作为java结构层次上的顶层.

接口的特点

- [x] **接口中都是抽象方法**(默认拼接public abstract)，**没有成员变量，有静态常量**
- [ ] 通过interface关键字来定义接口；通过implements让子类来实现接口
- [ ] 可以理解成,接口是一个特殊的抽象类(接口里的方法都是抽象方法)
- [ ] 接口突破了java单继承的局限性
- [x] **接口中没有构造方法**
- [x] **接口和类之间可以多实现,接口与接口之间可以多继承**

- [ ] 接口是对外暴露的规则,是一套开发规范；接口提高了程序的功能拓展,降低了耦合性
- [x] **JDK1.8之前，接口里面不可以有普通方法和成员变量；JDK1.8之后可以有普通方法，通过加default和static修饰方法实现**

```
public interface interf {
    int AGE = 10;//默认拼接public static final
    void eat();//默认拼接public abstract
    void play();
    /**jdk1.8后可以有普通方法,加default*/
    default void run(){
        System.out.println("1");
    }
    /**加static*/
    static void study(){
        System.out.println("2");
    }
}
```

接口的使用

问题:子类创建对象时,默认会调用父类的构造方法，目前接口实现类的父级是一个接口,而接口没有构造方法，那实现类构造方法中的super()调用的是谁呢?

结论:如果一个类没有明确指定父类,那么默认继承顶级父类Object，所以super()会自动调用Object类中的无参构造

接口里是没有构造方法的。在创建实现类的对象时默认的super()，是调用的默认Object的无参构造。

```
interface A{
    void eat();
}
class B implements A{
    public B(){
        spuer();//调用Object类的无参构造
    }
    @Override
    public void eat(){
        //具体实现
    }
}
```

总结

1.类与类的关系

> - --`继承关系,只支持单继承`
> - --比如,A是子类 B是父类,A具备B所有的功能(除了父类的私有资源和构造方法)
> - --子类如果要修改原有功能,需要重写(方法签名与父类一致 + 权限修饰符>=父类修饰符)

2.类和接口的关系

> - --`**实现关系.可以单实现,也可以多实现**`
> - --class A implements B,C{}
> - --其中A是实现类,B和C是接口,A拥有BC接口的所有功能,只是需要进行方法的重写,否则A就是抽象类

3.接口与接口的关系

> - --`**是继承关系,可以单继承,也可以多继承**`
> - --interface A extends B,C{}
> - --其中ABC都是接口,A是子接口,具有BC接口的所有功能(抽象方法)
> - --class X implements A{}
> - --X实现类需要重写ABC接口的所有方法,否则就是抽象类
> - --class A extends B implements C,D{}
> - --其中A是实现类,也是B的子类,同时拥有CD接口的所有功能
> - --这时A需要重写CD接口里的所有抽象方法

4.抽象类与接口的区别

1. 抽象类可以有构造方法，接口中不能有构造方法。
2. 抽象类中可以有普通成员变量，接口中没有普通成员变量
3. 抽象类中可以包含非抽象的普通方法，接口中的所有方法必须都是抽象的，不能有非抽象的普通方法。
4. 抽象类中的抽象方法的访问类型可以是public，protected和默认类型，（虽然eclipse下不报错，但应该也不行，但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型）。
5. **JDK1.8之前，接口里面不可以有普通方法和成员变量；JDK1.8之后可以有普通方法，通过加default和static修饰方法实现**；抽象类可以有普通方法
6. 抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public static final类型，并且默认即为public static final类型。
7. 一个类可以实现多个接口，但只能继承一个抽象类。

内部类

概述

如果一个类存在的意义就是为另一个类使用，可以把这个类放入另一个类的内部。就是把类定义在类的内部的情况就可以形成内部类的形式。

！特点!

> - 1) 内部类可以直接访问外部类中的成员，包括私有成员
> - 2) 外部类要访问内部类的成员，必须要建立内部类的对象
> - 3) 在成员位置的内部类是成员内部类
> - 4) 在局部位置的内部类是局部内部类

！！总结！！

<常规内部类>

1. 创建外部类
2. 创建成员内部类--类的特殊成员，内部类根据位置不同,分为两种:成员内部类(类里方法外)/局部内部类(方法里)

3. 创建内部类对象,使用内部类的资源：外部类名.内部类名 变量名 = 外部类对象.内部类对象，例如 **Outer.Inner oi = new Outer().new Inner();**
4. 外部类中可以使用内部类资源吗? 外部类想要使用内部类资源,必须先创建内部类对象,通过内部类对象来访问
5. 内部类可以使用外部类的资源吗?--可以!!!私有成员也可以!!!

```
public class Inner1 {
    public static void main(String[] args) {

        //三、.创建内部类对象,使用内部类的资源
        //外部类名.内部类名 变量名 = 外部类对象.内部类对象
        Outer.Inner oi = new Outer().new Inner();
        oi.get();
        oi.ok();
        oi.runs();
        //oi.go();私有的无法使用

        System.out.println(oi.count);

        //外部类和之前一样,创建对象直接调用
        new Outer().save();
    }
}

/**一、创建外部类*/
class Outer{

    String name;
    private int age;

    public void save() {
        System.out.println("Outer...save()");

        //五、外部类想要使用内部类资源,必须先创建内部类对象,通过内部类对象来访问
        Inner in = new Inner();
        System.out.println(in.count);
        in.get();
    }

    /**二、创建成员内部类--类的特殊成员*/
    class Inner{
        int count = 10;
        String name = "海绵宝宝";
        public void get() {
            System.out.println("Inner...get()");
            //四、内部类可以使用外部类的资源吗?--可以!!!私有成员也可以!!!
            System.out.println(name);
            System.out.println(age);
            System.out.println(Outer.this.name);//同名时调用方式，前提是不要私有化
            //save();
        }
        //创建内部类的private成员方法
        private void go(){
            System.out.println("私有成员方法");
        }
        //创建内部类的private成员方法
        protected void runs(){
```

```
            System.out.println("内部类的protected成员方法");
        }
        //创建default普通方法
        void ok(){
            System.out.println("内部类的default成员方法");
        }
    }
}
```

成员内部类被private修饰

成员内部类(类里方法外)被Private修饰以后,无法被外界直接创建创建对象使用，所以可以创建外部类对象,通过外部类对象间接访问内部类的资源

1. 创建外部类
2. 创建private成员内部类
3. 提供外部类公共的全局访问点,在方法内部创建内部类对象,调用内部类方法
4. 先创建外部类对象,调用全局访问点，间接访问私有内部类的资源(曲线访问，类似于单立设计)

```
public class Inner2 {
    public static void main(String[] args) {
        //四、调用全局访问点
        new Outer2().getInner2Eat();
    }
}

/**一、创建外部类*/
class Outer2{

    /**三、.提供外部类公共的全局访问点,在方法内部创建Inner2内部类对象,调用内部类方法*/
    void getInner2Eat() {
        //外部类可以访问内部类的私有成员,但要先创建对象
        Inner2 in = new Inner2();
        in.eat();
        in.go();
        in.ok();
        in.runs();
    }

    //二、创建成员内部类Inner2
    private class Inner2 {
        //3.创建内部类的public成员方法
        public void eat() {
            System.out.println("我是Inner2的eat()");
        }
        //创建内部类的private成员方法
        private void go(){
            System.out.println("私有成员方法");
        }
        //创建内部类的private成员方法
        protected void runs(){
            System.out.println("内部类的protected成员方法");
        }
        //创建default普通方法
        void ok(){
            System.out.println("内部类的default成员方法");
        }
```

```
    }
}
```

成员内部类被static修饰

静态资源访问时不需要创建对象,可以通过类名直接访问，访问静态类中的静态资源可以通过”. . . ”链式加载的方式访问

1. 创建外部类
2. 创建静态内部类:
3. 定义成员内部类中普通的成员方法
4. 内部类被static修饰—并不常用!浪费内存
5. 通过外部类的类名创建静态内部类对象 外部类名.内部类名 对象名 = new 外部类名.内部类名()
6. 匿名的内部类对象调用普通方法 外部类名.内部类名().普通方法名
7. 访问静态内部类中的静态资源--链式加载 外部类名.内部类名.静态方法名

```
public class Inner3 {
    public static void main(String[] args) {

        /* ******************如何访问内部类的show()?**********************/
        //④.创建内部类对象访问show()
        //方式一:按照之前的方式,创建内部类对象调用show()
        //Outer3.Inner3 oi = new Outer3().new Inner3();
        //oi.show();
        //方式二:创建匿名内部类对象访问show()
        //new Outer3().new Inner3().show();

        /* ***************现象:当内部类被static修饰以后,new Outer3()报错*****/
        //⑥.用static修饰内部类以后,上面的创建语句报错,注释掉
        //通过外部类的类名创建静态内部类对象
        Outer3.Inner3 oi = new Outer3.Inner3();
        oi.show();

        //⑦.匿名的内部类对象调用show()
        new Outer3.Inner3().show();

        //⑨.访问静态内部类中的静态资源--链式加载
        Outer3.Inner3.play();
    }
}

/**①.创建外部类Outer3*/
class Outer3{

    //②.创建成员内部类Inner3
    /**⑤.内部类被static修饰—并不常用!浪费内存!*/
    static class Inner3{
        //③.定义成员内部类中普通的成员方法
        public void show() {
            System.out.println("我是Inner3类的show()");
        }

        //⑧.定义成员内部类的静态成员方法(前提是这个内部类用static修饰)
        public static void play() {//Inner classes cannot have static
declarations
            System.out.println("我是Inner3的play()");
        }
```

```
    }
}
```

局部内部类(方法里)

如何使用内部类的资源呢? **注意**:直接调用外部类对象的show()是无法触发局部内部类功能的 需要再外部类中创建内部类对象并且进行调用,才能触发内部类的功能

1. 创建外部类
2. 创建成员方法
3. 创建局部内部类—-不太常用!!!
4. 创建局部内部类的普通属性与方法
5. 在外部类的成员方法里，直接创建内部类对象，当在外部类中创建局部内部类对象并且进行功能调用后,内部类的功能才能被调用
6. 创建外部类对象调用外部类的成员方法

```
public class Inner4 {
    public static void main(String[] args) {
        //⑤.创建外部类对象调用show()
        //Outer4 outer4 = new Outer4();
        //outer4.show();
        //⑦.当在外部类show()中创建局部内部类对象并且进行功能调用后,内部类的功能才能被调用
        new Outer4().show();
    }
}

/**①.创建外部类Outer4*/
class Outer4{

    /**②.创建外部类的成员方法*/
    public void show() {
        /*③.创建局部内部类Inner4--不太常用!!!
        位置:局部内部类的位置在方法里*/

        class Inner4 {
            //④.创建局部内部类的普通属性与方法
            String name;
            int age;
            public void eat() {
                System.out.println("我是Inner4的eat()");
            }
        }

        /*如何使用局部内部类的资源?*/
        //⑥.在show()里直接创建内部类对象
        Inner4 in = new Inner4();
        in.eat();
        System.out.println(in.name);
        System.out.println(in.age);
    }
}
```

匿名内部类

匿名内部类属于局部内部类,而且是没有名字的局部内部类,通常和匿名对象一起使用

- 优势：方便，优化程序设计结构
- 劣势：只能调用一个,且只能调用一次

`new 对象名() + {...}; 匿名对象 + 匿名内部类`

1. 创建接口，定义接口中的抽象方法
2. 创建抽象类和普通类
3. 由于只是用一次，可以直接 `new 接口名(){实现接口的具体代码}.要调用的实现方法名();`
4. 抽象类和普通类一样 `new 类名(){重写的代码}.重写后要调用的方法名();`

```
public class Inner5 {
    public static void main(String[] args) {

        /*接口可以创建对象吗?不可以!!!*/
        //new Inner1();//接口不能创建对象

        //③.  new Inner1(){...}; 匿名对象
        /*就相当于创建了一个接口的实现类 + 重写接口中的所有抽象方法*/
        new Inners1() {//a、接口的实现类

            @Override
            public void save() {//重写接口中的抽象方法1
                System.out.println("我是Inner1接口的save()");
            }

            @Override
            public void get() {//重写接口中的抽象方法2
                System.out.println("我是Inner1接口的get()");
            }
        }.get();//b、触发指定的重写后的方法,只能调用一个,且只能调用一次
        /*注意!!!匿名对象只干一件事!!!*/

        /*
**********************************************************************/

        //⑤. new Inner2(){...}; 匿名对象,相当于创建了抽象类的子类,必须重写所有抽象方法
        new Inners2() {
            @Override
            public void eat() {
                System.out.println("我是Inner2抽象类的eat()");
            }
        }.eat();

        /*
**********************************************************************/

        //⑦.普通类的匿名对象,没有强制要求产生匿名内部类的重写方法
        new Inners3(){
            @Override
            public void game() {
                System.out.println();
            }

            @Override
```

```
            public void sleep() {
                System.out.println();
            }
        };
        //如果使用对象,只需要干一件事--直接创建匿名对象,简单方便
        new Inners3().sleep();
        //如果使用同一个对象,要干很多事情--必须给对象起名字
        Inners3 in = new Inners3();
        in.game();
        in.sleep();
    }
}

/**
 * ①.创建接口
 */
interface Inners1 {

    /**
     * ②.定义接口中的抽象方法
     */
    void get();
    void save();
}

/**
 * ④.创建抽象类
 */
abstract class Inners2 {

    public abstract void eat();

    public void play() {
        System.out.println("我是Inner2抽象类的play()");
    }
}

/**
 * ⑥.创建普通类
 */
class Inners3 {
    public void game() {
        System.out.println("我是Inner3普通类的game()");
    }

    public void sleep() {
        System.out.println("我是Inner3普通类的sleep()");
    }
}
```

Java API

Object

Object类是所有Java类的祖先,也就是说我们所说的"顶级父类"。存在于java.lang.Object,这个包不需要我们手动导包。每个类都使用Object作为超类.所有对象(包括数组)都实现这个类的方法.在不明确给出超类的情下,Java会自动把Object类作为要定义类的超类。

常用方法

toString()

本方法用于返回对应对象的字符串表示.

如果想要查看对象的属性值,需要重写toString(),否则会使用Object的默认实现,打印对象的地址值

```
@Override
public String toString() {
    return "Person{" +
            "name='" + name + '\'' +
            ", age=" + age +
            '}';
}
```

hashcode()

本方法用于返回对应对象的哈希码值

TIPS:哈希码是一种算法,使不同的对象有不同的哈希码值,但是也有相同的情况,我们称之为"哈希碰撞"

```
@Override
public int hashCode() {
    return Objects.hash(name, age);
}
```

equals()

本方法用于指示其他某个对象是否与当前对象"相等"

如果要判断两个对象间的所有属性值相同,比较完的结果返回true,需要重写equlas(),否则会使用Object类的默认实现==比较

```
@Override
public boolean equals(Object obj) {
        //判断,如果是学生,咱俩比较,如果连学生都不是,就返回false--匹配种类
        if(obj instanceof Student) {//同一个类型的对象间比较
            //把obj强转成Student类型,因为想用子类的特有属性,如果不转,只能用父类的属性
            Student argsStudent = (Student)obj;
            //判断:如果当前对象this和参数对象argsStudent,他们的属性&属性值完全一样,就返回
true
            if(this.name == argsStudent.name && this.age == argsStudent.age) {
                return true;
            }
        }
        return false;
    }
```

String

特点

String是一个**封装char[]数组的对象,字符串不可变**；通过底层实现可以看出:

`String 类中使用final关键字修饰字符数组来保存字符串,是常量，所以线程安全，也解决了线程同步问题，所以 String 对象是不可变的。在 Java 9 之后，String 类的实现改用 byte 数组存储字符串:private final byte[] value`

String str = "abc"; 等效于：char data[] = {'a', 'b', 'c'};

常用方法

> - length()-查看字符串的长度
> - charAt()—定位某个字符,返回它的位置
> - lastIndexOf()-某个字符最后一次出现的位置
> - substring()-截取子串,如果参数有两个左闭右开[1,5)
> - equals()-判断两个串是否相等,注意String重写了Object的此方法,所以内容相同就返回true
> - startsWith()-判断是不是以参数开头
> - endsWith()--判断是不是以参数结尾
> - split()-以指定字符分割
> - trim()-去掉首尾两端的空格
> - getBytes()-把串转换成数组
> - toUpperCase()-变成全大写
> - toLowerCase()-变成全小写
> - String.valueOf(10)-把int类型的10转换成String类型

String/StringBuilder/StringBuffer区别

```
1. 在线程安全上 ：
   --String 中的对象是不可变的，也就可以理解为常量，线程安全。
   --StringBuffer是旧版本就提供的，StringBuffer对方法加了同步锁或者对调用的方法加了同步锁，
     所以是线程安全的线程安全的。@since JDK1.0
   --StringBuilder是jdk1.5后产生,StringBuilder并没有对方法进行加同步锁,所以是非线程安全
的。@since      JDK1.5

2. 在执行效率上，
    --StringBuilder > StringBuffer > String
    --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 StringBuilder 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。

3.源码体现上
    --String 类中使用 final 关键字修饰字符数组来保存字符串，所以 String 对象是不可变的。
    --在 Java 9 之后，String 类的实现改用 byte 数组存储字符串:private final byte[]
value
    --而 StringBuilder 与 StringBuffer 都继承自AbstractStringBuilder类，在
AbstractStringBuilder类也是使用字符数组保存字符串变的。但是没有用 final 关键字修饰，所以这
两种对象都是可变的
    --StringBuilder 与 StringBuffer 的构造方法都是调用父类构造方法也就是
AbstractStringBuilder 实现的，

   对于三者使用的总结：
   ①.操作少量的数据: 适用 String
```

```
②.单线程操作字符串缓冲区下操作大量数据: 适用 StringBuilder
③.多线程操作字符串缓冲区下操作大量数据: 适用 StringBuffer
```

正则表达式

正确的字符串格式规则。

常用来判断用户输入的内容是否符合格式的要求，注意是严格区分大小写的。

Matches(正则) : 当前字符串能否匹配正则表达式**

replaceAll(正则,子串) : 替换子串**

split(正则) : 拆分字符串**

常用正则表达式

一、校验数字的表达式

- 数字：^[0-9]*$
- n位的数字：^\d{n}$
- 至少n位的数字：^\d{n,}$
- m-n位的数字：^\d{m,n}$
- 零和非零开头的数字：^(0|[1-9][0-9]*)$
- 非零开头的最多带两位小数的数字：^([1-9][0-9]*)+(\.[0-9]{1,2})?$
- 带1-2位小数的正数或负数：^(\-)?\d+(\.\d{1,2})$
- 正数、负数、和小数：^(\-|\+)?\d+(\.\d+)?$
- 有两位小数的正实数：^[0-9]+(\.[0-9]{2})?$
- 有1~3位小数的正实数：^[0-9]+(\.[0-9]{1,3})?$
- 非零的正整数：^[1-9]\d*$ 或 ^([1-9][0-9]*){1,3}$ 或 ^\+?[1-9][0-9]*$
- 非零的负整数：^\-[1-9][]0-9"*$ 或 ^-[1-9]\d*$
- 非负整数：^\d+$ 或 ^[1-9]\d*|0$
- 非正整数：^-[1-9]\d*|0$ 或 ^((-\d+)|(0+))$
- 非负浮点数：^\d+(\.\d+)?$ 或 ^[1-9]\d*\.\d*|0\.\d*[1-9]\d*|0?\.0+|0$
- 非正浮点数：^((-\d+(\.\d+)?)|(0+(\.0+)?))$ 或 ^(-([1-9]\d*\.\d*|0\.\d*[1-9]\d*))|0?\.0+|0$
- 正浮点数：^[1-9]\d*\.\d*|0\.\d*[1-9]\d*$ 或 ^(([0-9]+\.[0-9]*[1-9][0-9]*)|([0-9]*[1-9][0-9]*\.[0-9]+)|([0-9]*[1-9][0-9]*))$
- 负浮点数：^-([1-9]\d*\.\d*|0\.\d*[1-9]\d*)$ 或 ^(-(([0-9]+\.[0-9]*[1-9][0-9]*)|([0-9]*[1-9][0-9]*\.[0-9]+)|([0-9]*[1-9][0-9]*)))$
- 浮点数：^(-?\d+)(\.\d+)?$ 或 ^-?([1-9]\d*\.\d*|0\.\d*[1-9]\d*|0?\.0+|0)$

校验字符的表达式

- 汉字：^[\u4e00-\u9fa5]{0,}$
- 英文和数字：^[A-Za-z0-9]+$ 或 ^[A-Za-z0-9]{4,40}$
- 长度为3-20的所有字符：^.{3,20}$
- 由26个英文字母组成的字符串：^[A-Za-z]+$
- 由26个大写英文字母组成的字符串：^[A-Z]+$
- 由26个小写英文字母组成的字符串：^[a-z]+$
- 由数字和26个英文字母组成的字符串：^[A-Za-z0-9]+$
- 由数字、26个英文字母或者下划线组成的字符串：^\w+$ 或 ^\w{3,20}$
- 中文、英文、数字包括下划线：^[\u4E00-\u9FA5A-Za-z0-9_]+$
- 中文、英文、数字但不包括下划线等符号：^[\u4E00-\u9FA5A-Za-z0-9]+$ 或 ^[\u4E00-\u9FA5A-Za-z0-9]{2,20}$
- 可以输入含有^%&',;=?$\"等字符：[^%&',;=?$\x22]+
- 禁止输入含有~的字符：[^~\x22]+

三、特殊需求表达式

- Email地址：^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$
- 域名：[a-zA-Z0-9][-a-zA-Z0-9]{0,62}(\.[a-zA-Z0-9][-a-zA-Z0-9]{0,62})+\.?
- InternetURL：[a-zA-z]+://[^\s]* 或 ^http://([\w-]+\.)+[\w-]+(/[\w-./?%&=]*)?$
- 手机号码：^(13[0-9]|14[5|7]|15[0|1|2|3|4|5|6|7|8|9]|18[0|1|2|3|5|6|7|8|9])\d{8}$
- 电话号码("XXX-XXXXXXX"、"XXXX-XXXXXXXX"、"XXX-XXXXXXX"、"XXX-XXXXXXXX"、"XXXXXXX"和"XXXXXXXX)：^(\(\d{3,4}-)|\d{3.4}-)?\d{7,8}$
- 国内电话号码(0511-4405222、021-87888822)：\d{3}-\d{8}|\d{4}-\d{7}
- 电话号码正则表达式（支持手机号码，3-4位区号，7-8位直播号码，1 - 4位分机号）：((\d{11})|^((\d{7,8})|(\d{4}|\d{3})-(\d{7,8})|(\d{4}|\d{3})-(\d{7,8})-(\d{4}|\d{3}|\d{2}|\d{1})|(\d{7,8})-(\d{4}|\d{3}|\d{2}|\d{1}))$)
- 身份证号(15位、18位数字)，最后一位是校验位，可能为数字或字符X：(^\d{15}$)|(^\d{18}$)|(^\d{17}(\d|X|x)$)
- 帐号是否合法(字母开头，允许5-16字节，允许字母数字下划线)：^[a-zA-Z][a-zA-Z0-9_]{4,15}$
- 密码(以字母开头，长度在6~18之间，只能包含字母、数字和下划线)：^[a-zA-Z]\w{5,17}$
- 强密码(必须包含大小写字母和数字的组合，不能使用特殊字符，长度在 8-10 之间)：^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])[a-zA-Z0-9]{8,10}$
- 强密码(必须包含大小写字母和数字的组合，可以使用特殊字符，长度在8-10之间)：^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,10}$
- 日期格式：^\d{4}-\d{1,2}-\d{1,2}
- 一年的12个月(01～09和1～12)：^(0?[1-9]|1[0-2])$
- 一个月的31天(01～09和1～31)：^((0?[1-9])|((1|2)[0-9])|30|31)$
- 钱的输入格式：
    1. 有四种钱的表示形式我们可以接受:"10000.00" 和 "10,000.00", 和没有 "分" 的 "10000" 和 "10,000"：^[1-9][0-9]*$
    2. 这表示任意一个不以0开头的数字,但是,这也意味着一个字符"0"不通过,所以我们采用下面的形式：^(0|[1-9][0-9]*)$
    3. 一个0或者一个不以0开头的数字.我们还可以允许开头有一个负号：^(0|-?[1-9][0-9]*)$
    4. 这表示一个0或者一个可能为负的开头不为0的数字.让用户以0开头好了.把负号的也去掉,因为钱总不能是负的吧。下面我们要加的是说明可能的小数部分：^[0-9]+(.[0-9]+)?$
    5. 必须说明的是,小数点后面至少应该有1位数,所以"10."是不通过的,但是 "10" 和 "10.2" 是通过的：^[0-9]+(.[0-9]{2})?$
    6. 这样我们规定小数点后面必须有两位,如果你认为太苛刻了,可以这样：^[0-9]+(.[0-9]{1,2})?$
    7. 这样就允许用户只写一位小数.下面我们该考虑数字中的逗号了,我们可以这样：^[0-9]{1,3}(,[0-9]{3})*(.[0-9]{1,2})?$
    8. 1到3个数字,后面跟着任意个 逗号+3个数字,逗号成为可选,而不是必须：^([0-9]+|[0-9]{1,3}(,[0-9]{3})*)(.[0-9]{1,2})?$
    9. 备注：这就是最终结果了,别忘了"+"可以用"*"替代如果你觉得空字符串也可以接受的话(奇怪,为什么?)最后,别忘了在用函数时去掉去掉那个反斜杠,一般的错误都在这里
- xml文件：^([a-zA-Z]+-?)+[a-zA-Z0-9]+\\.[x|X][m|M][l|L]$
- 中文字符的正则表达式：[\u4e00-\u9fa5]
- 双字节字符：[^\x00-\xff] (包括汉字在内，可以用来计算字符串的长度(一个双字节字符长度计2，ASCII字符计1))

- 空白行的正则表达式：**\n\s*\r (可以用来删除空白行)**
- HTML标记的正则表达式：**<(\S*?)[^>]*>.*?|<.*? /> (首尾空白字符的正则表达式：^\s*|\s*$或(^\s*)|(\s*$) (可以用来删除行首行尾的空白字符(包括空格、制表符、换页符等等)，非常有用的表达式)**
- 腾讯QQ号：**[1-9][0-9]{4,} (腾讯QQ号从10000开始)**
- 中国邮政编码：**[1-9]\d{5}(?!\d) (中国邮政编码为6位数字)**
- IPv4地址：**((2(5[0-5]|[0-4]\d))|[0-1]?\d{1,2})(\.((2(5[0-5]|[0-4]\d))|[0-1]?\d{1,2})){3}**

包装类

把基本类型进行包装，提供更加完善的功能。

基本类型是没有任何功能的,只是一个变量,记录值,而包装类可以有更加丰富的功能

基本数据类型	包装数据类型	二进制位数
byte	Byte	8
short	Short	16
int	Integer	32
char	Character	16
float	Float	32
double	Double	64
boolean	Boolean	1
long	Long	64

```
public class Box {
    public static void main(String[] args) {

        //1.定义包装类型的数据
        //之前的方式:创建包装类型的两种方式(区别):
        //①Integer引用类型变量a中储存在堆中开辟的空间的地址，地址中储存100
        Integer i11 = new Integer(127);
        //②直接调用Integer的属性valueOf,将int类型的10，隐式加包为Integer类型的10
        //再将转化后的10储存在Integer引用类型变量a中
        Integer i22 = Integer.valueOf(127);

        //Integer有一个高效的效果(-128~127)
        Integer a = new Integer(100);
        Integer b = new Integer(100);
        Integer c = Integer.valueOf(100);
        Integer d = Integer.valueOf(100);
        Integer e = Integer.valueOf(1000);
        Integer f = Integer.valueOf(1000);

        //false,a/b只是存储了地址值
        System.out.println(a == b);
        //ture
        System.out.println(c == d);
        //false,有加包范围
        System.out.println(e == f);
    }
}
```

自动装箱和自动拆箱

自动装箱：把基本数据类型用他们对应的引用类型包装起来的过程

自动拆箱：把包装类型转换成基本数据类型的过程

==和equals的区别

- **1.当使用==比较时，如果相比较的两个变量是引用类型，那么比较的是两者的物理地值（内存地址），如果相比较的两个变量都是数值类型，那么比较的是具体数值是否相等。**
- **2.当使用equals()方法进行比较时，比较的结果实际上取决于equals()方法的具体实现，初始行为还是是使用==比较对象的内存地址**

众所周知，任何类都继承自Object类，因此所有的类均具有Object类的特性，比如String、integer等，他们在自己的类中重写了equals()方法，此时他们进行的是数值的比较，而在Object类的默认实现中，equals()方法的底层是通过==来实现的。

BigDecimal/BigInteger

- 使用BigDecimal:常用来解决精确的浮点数运算
- BigInteger： 常用来解决超大的整数运算

```
//1.最好不要使用double作为构造函数的参数,不然还会产生不精确的现象,有坑!!!!
//2.最好使用重载的,参数类型是String的构造函数,double转String,直接拼个空串就可以
   BigDecimal bd1 = new BigDecimal(a + "");
```

> - Add(BigDecimal bd) : 做加法运算
> - Subtract(BigDecimal bd) : 做减法运算
> - Multiply(BigDecimal bd) : 做乘法运算
> - Divide(BigDecimal bd) : 做除法运算,除不尽时会抛异常
> - **Divide(BigDecimal bd,保留位数,舍入方式) : 除不尽时使用**
> - setScale(**保留位数**,舍入方式) : 同上
> - pow(int n) : 求数据的几次幂

舍入方式解析

ROUND_HALF_UP 四舍五入,五入 如:4.4结果是4; 4.5结果是5

ROUND_HALF_DOWN 五舍六入,五不入 如:4.5结果是4; 4.6结果是5

ROUND_HALF_EVEN 公平舍入(银行常用)

比如:在5和6之间,靠近5就舍弃成5,靠近6就进位成6,如果是5.5,就找偶数,变成6

ROUND_UP 直接进位,不算0.1还是0.9,都进位

ROUND_DOWN 直接舍弃,不算0.1还是0.9,都舍弃

ROUND_CEILING(天花板)** 向上取整,取实际值的大值

朝正无穷方向round 如果为正数，行为和round_up一样，如果为负数，行为和round_down一样

ROUND_FLOOR(地板)** 向下取整,取实际值的小值

朝负无穷方向round 如果为正数，行为和round_down一样，如果为负数，行为和round_up一样

Java IO

字节流：针对二进制文件，可以是任何文件

`InputStream`

- --FileInputStream
- --BufferedInputStream
- --ObjectInputStream

`OutputStream`

- --FileOutputStream
- --BufferedOutputStream
- --ObjectOutputStream

字符流:常用于纯文本文件,读写容易出现乱码的现象,在读写时,最好指定编码集为UTF-8

`Reader`

- --FileReader
- --BufferedReader
- --InputStreamReader

`Writer`

- --FileWriter
- --BufferedWriter
- --OutputStreamWriter
- --PrintWriter一行行写出

常用IO总结:

最好可以默写出来

①.字节输入流:`InputStream`--抽象父类，不可以创建对象，学习其公共方法，此抽象类是表示字节输入流的所有类的超类/抽象类,不可创建对象。查看api手册学习方法。 1、*FileInputStream*--操作文件的字节输入流 如:FileInputStream in = new FileInputStream("D:\ready\1.txt"); 2、BufferedInputStream--缓冲(高效)字节输入流 如：BufferedInputStream in = new BufferedInputStream(new FileInputStream("D:\ready\1.txt"));

②.字符输入流:`Reader`--抽象父类，不可以创建对象，学习其公共方法 1、*FileReader*--操作文件的字符输出流 如：FileReader in = new FileReader("D:\ready\1.txt"); 2、*BufferedReader*--缓冲(高效)字符输入流 如：BufferedReader in = new BufferedReader(new FileReader("D\ready\1.txt"));

③.字节输出流:`OutputStream`--抽象父类，不可以创建对象，学习其公共方法 1、*FileOutputStream*--操作文件的字节输出流 如：FileOutputStream out = new FileOutputStream("D:\ready\out.txt"); 2、*BufferedOutputStream*--缓冲(高效)字节输出流 如：BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream("D:\ready\out.txt"));

④.字符输出流:`Writer`--抽象父类，不可以创建对象，学习其公共方法 1、*FileWriter*--操作文件的字符输出流 如：FileWriter out = new FileWriter("D:\ready\out.txt",true); 2、*BufferedWriter*--缓冲(高效)字符输出流 如：BufferedWriter out = new BufferedWriter(new FileWriter("D://ready//out.txt",true));

```
File file = new File(pathname);//封装一个磁盘路径字符串,对这个路径可以执行一次操作，可以封装文件路径、文件夹路径、不存在的路径

file.length();//获取指定文件的字节量
file.exists();//判断指定文件是否存在
file.isFile();//判断指定内容是否为文件
```

```
file.isDirectory();//判断指定内容是否为文件夹
file.getName();//获取指定文件的名字
file.getParent();//获取指定文件的上级,父级目录
file.getAbsolutePath();//获取指定内容的绝对路径
file.createNewFile();//在windows中创建不存在的文件,XXX.txt被自动创建成功
file.mkdir();//创建不存在的单级文件夹,m文件夹自动创建成功
file.mkdirs();//创建不存在的多级文件夹,多级文件夹会被自动创建成功
file.delete();//删除文件或者删除空的文件夹
file.list();//查看文件夹中的所有文件的名称,返回值类型是String[]
file.listFiles();//列出文件夹中所有的文件夹和文件对象,返回值是File[],每个数组元素都是file对象,操作性强
```

常用方法

> **写入常用:对象名.read();方法，读取到为-1时，则表示没有数据了。还有它重载后的方法。**

```
int read(char[] cbuf) //将字符读入数组
abstract int read(char[] cbuf, int off, int len)// 将字符读入数组的某一部分
int read(CharBuffer target) //试图将字符读入指定的字符缓冲区
abstract void close() //关闭该流并释放与之关联的所有资源
```

> **写出常用:对象名.write();方法,还有重载后的方法。**

```
Void close() //关闭此输出流并释放与此流相关的所有系统资源
Void flush() //刷新此输出流并强制写出所有缓冲的输出字节
Void write(byte[ ] b) //将b.length个字节从指定的byte数组写入此输出流
Void write(byte[ ] b,int off ,int len) //将指定byte数组中从偏移量off开始的len个字节写入输出流
Abstract void write(int b) //将指定的字节写入此输出流
```

序列化和反序列化

序列化概念:

是指把程序中的java对象,永久保存在磁盘中,相当于是**写出**的过程,序列化就是将一个对象转换成字节序列，方便存储和传输。不会对静态变量进行序列化，因为序列化只是保存对象的状态，静态变量属于类的状态。序列化(Serialization)是将对象的状态信息转换为可以存储或传输形式的过程.在序列化期间,对象将其当前状态写入到临时或持久性存储区.以后可以通过从存储区中读取或者帆序列化对象的状态,重新创建该对象.

序列化

利用ObjectOutputStream,把对象的信息,按照固定的格式转成一串字节值输出并持久保存到磁盘

方向是 out --> ObjectOutputStream

反序列化

反序列化:是指把已经序列化在文件中保存的数据,读取/恢复到java程序中的过程,

方向是 in --> ObjectInputStream

特点/应用场景

- 1.`需要序列化的文件必须实现Serializable接口,用来启用序列化功能`

```
public class ClassName implements Serializable{}
```

- 2.`不需要序列化的数据可以修饰成static`,原因:static资源属于类资源,不随着对象被序列化输出
- 3.`每一个被序列化的文件都有一个唯一的UId`,如果没有添加此id,编译器会自动根据类的定义信息计算产生一个

```
private static final long serialVersionUID = 1L;
```

- 4.在反序列化时,如果和序列化的版本号不一致,无法完成反序列化
- 5.常用与服务器之间的数据传输,序列化成文件,反序列化读取数据
- 6.常用使用套接字流在主机之间传递对象
- 7.`不需要序列化的数据也可以被修饰成transient(临时的)`,只在程序运行期间在内存中存在,不会被序列化持久保存

泛型

概念

泛型是（Generics）JDK1.5 的一个新特性，其实就是一个『语法糖』，本质上就是编译器为了提供更好的可读性而提供的一种小手段，小技巧，虚拟机层面是不存在所谓『泛型』的概念的。

`泛型可以在接口 类 方法上使用`

作用/优点

- 通过泛型的语法定义<>,模拟数组的数据类型检查，**约束集合元素的类型**,编译器可以在编译期提供一定的类型安全检查
- 这样可以**避免程序运行时才暴露BUG**,代码的通用性也会更强
- 泛型可以提升程序代码的可读性,但是它只是一个『语法糖』(编译后这样的部分会被删除,不出现在最终的源码中),所以不会影响JVM后续运行时的性能.

常用名称

> - E - Element (在集合中使用,因为集合中存放的是元素)
> - T - Type (Java类)
> - K - Key (键)
> - V - Value (值)
> - N - Number (数值类型)
> - ? - 表示不确定的java类型

Collection

概述

是用来存放对象的**数据结构,而且长度可变**,可以存放不同类型的对象,并且还提供了一组操作成批对象的方法

List接口：数据是有下标的,所以数据是有序的,可以存重复值**

- `ArrayList`:**底层是数组，查询快,增删数据效率会低**，适合查询较多的场景；内部数组默认的初始容量是10,如果不够会以1.5倍的容量增长；

> 常用方法：

```
list.contains(300);//判断集合是否包含元素300
list.get(0);//获取集合中指定下标位置上的元素
list.indexOf(200);//判断集合中指定元素第一次出现的下标
list.lastIndexOf(200);//判断集合中指定元素最后一次出现的下标
list.isEmpty();//判断集合是否为空
list.remove(1);//移除集合中指定下标对应着的元素,移除成功,返回被移除的元素
list.set(2, "77");//更改集合中对应下标上元素的值
```

> 四种迭代方式：ArrayList<? > list = new ArrayList<>();

```
方式1
//开始:0    结束:最大下标(集合长度-1)   变化:++
for (int i = 0; i < list.size(); i++) {
//根据下标获取对应下标位置上的元素
    System.out.print(list.get(i));
}
//逆序遍历
for (int i = list.size() - 1; i >= 0; i--){
    System.out.println(list.get(i));
}
```

```
方式2:增强for循环,普通for循环遍历效率低,可以通过foreach提高遍历效率
//好处:语法简洁效率高 坏处:不能按照下标来处理数据
//格式:for(A B : C){循环体} C是要遍历的数据 A和B是遍历得到的单个数据的类型 和 名字
for (? i : list) {
    //打印每次循环到的集合元素
    System.out.print(i);
}
```

```
方式3:Iterator
/*1.获取迭代器  2.判断是否还有元素(一般用来做循环条件)  3.获取当前遍历到的元素*/
//获取集合用来迭代的迭代器,此迭代器是继承自Collection接口中的
Iterator<?> iterator = list.iterator();
//通过迭代器来判断集合中是否还有元素,如果有,继续迭代,如果没有,结束循环
while(iterator.hasNext()) {
    //获取当前遍历到的集合元素
    //String num = it.next();
    //打印当前遍历到的集合元素
    System.out.print(iterator.next());
}
```

```
方式4:ListIterator
System.out.println("方式4");
//获取集合用来迭代的迭代器,此迭代器是List接口中的迭代器
ListIterator<?> listIterator = list.listIterator();
//通过迭代器来判断集合中是否还有元素,如果有继续迭代,如果没有,结束循环
```

```
while(listIterator.hasNext()) {
    //获取当前遍历到的集合元素
    //String s2 = it2.next();
    //打印当前遍历到的集合元素
    System.out.print(listIterator.next());
}
//逆序遍历
while (it2.hasPrevious()){
    System.out.println(it2.previous());
}
```

方式三使用的是父接口中的Iterator;方式四使用的是子接口中的ListIterator

- `LinkedList`：**底层是链表，查询慢,增删数据效率会高**，适合增删操作较多的场景； **注意:LinkedList查询慢是指数据量大时,查询中间要慢,首位操作还是比较快的**

常用方法:

```
void addFirst(E e); //将指定元素插入此列表的开头
void addLast(E e);// 将指定元素添加到此列表的结尾

E getFirst(); //返回此列表的第一个元素
E getLast(); //返回此列表的最后一个元素
E removeFirst();//移除并返回此列表的第一个元素
E removeLast(); //移除并返回此列表的最后一个元素
E element(); //获取但不移除此列表的头（第一个元素）

boolean offer(E e); //将指定元素添加到此列表的末尾（最后一个元素）
boolean offerFirst(E e); //在此列表的开头插入指定的元素
boolean offerLast(E e); //在此列表末尾插入指定的元素

E peek(); //获取但不移除此列表的头（第一个元素）
E peekFirst(); //获取但不移除此列表的第一个元素；如果此列表为空，则返回 null
E peekLast(); //获取但不移除此列表的最后一个元素；如果此列表为空，则返回 null
E poll();//获取并移除此列表的头（第一个元素）
E pollFirst(); //获取并移除此列表的第一个元素；如果此列表为空，则返回 null
E pollLast(); //获取并移除此列表的最后一个元素；如果此列表为空，则返回 null
```

> 迭代方式同ArrayList

- `Vector`:**List的古老实现类，底层用数组存储，线程安全**

Set接口：数据是没有下标的,所以数据是无序的,不可以存重复的值,一般用于去重**

- `HashSet`：**底层是哈希表，包装了HashMap，相当于向HashSet中存入数据时，会把数据作为K，存入内部的HashMap中。当然K仍然不许重复。**

!HashSet如何查重!

> **当你把对象加入HashSet时，HashSet 会先计算对象的HashCode值来判断对象加入的位置，同时也会与其他加入的对象的hashcode值作比较，如果没有相符的hashcode，HashSet会假设 对象没有重复出现。但是如果发现有相同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就不会让加入操作成功。**

！！==和equals区别！！

- 对于基本类型来说，**== 比较的是值是否相等**；
- 对于引用类型来说，== 比较的是两个引用是否指向同一个对象地址（两者在内存中存放的地址（堆内存地址）是否指向同一个地方）；
- 对于引用类型（包括包装类型）来说，**equals 如果没有被重写，对比它们的地址是否相等；如果equals()方法被重写（例如 String），则比较的是地址里的内容。**

> set常用方法：

```
Set<String> set = new HashSet<String>();
set.add(null);//向set集合添加数据null
set.contains("小兔纸");//false,判断set集合中是否包含指定元素"小兔纸"
set.equals("牛气冲天");//false,判断set集合对象与指定元素是否相等
set.hashCode();//1961052313,获取当前set集合对象的哈希码
set.isEmpty();//false,判断当前集合是否为空
set.remove("null");//false,移除指定元素,没有"null"元素,所以返回false
set.remove(null);//true,成功移除指定元素null,所以返回true

set.size();//2,获取当前set集合的元素个数,类似数组长度
Object[] array = set.toArray();//把集合中的元素放入数组中

//4.集合间的操作
set.addAll(set2);//true,把集合set2中的元素添加到set集合中,成功返回true
set.containsAll(set2);//true,判断set集合中是否包含set2集合中的所有元素,如果包含返回true
set.removeAll(set2);//ture,移除set集合中属于set2集合的所有元素
set.containsAll(set2);//false,判断set集合中是否包含set2集合中的所有元素,不包含返回false
set.retainAll(set2);//retainAll()方法是取两个集合直接的公共部分,谁调用,影响谁
```

> **迭代和ArrayList一样**

- `TreeSet`：**底层是TreeMap,也是红黑树的形式,便于查找数据**

Map接口：键值对的方式存数据 <Key,Value>，也叫做哈希表、散列表. 常用于键值对结构的数据.其中键不能重复,值可以重复

1. Map可以根据键来提取对应的值
2. Map的键不允许重复,如果重复,对应的值会被覆盖
3. Map存放的都是无序的数据
4. Map的初始容量是16,默认的加载因子是0.75

- `HashMap`：

JDK1.8之前，底层的实现方式是**数组+链表**；JDK1.8之后是**数组+链表+红黑树**(为了解决二叉查找树的缺陷，因为二叉查找树可能会退化成线性结构)

> - HashMap底层是一个Entry[ ]数组,当存放数据时,会根据hash算法来计算数据的存放位置
> - 算法:hash(key)%n , n就是数组的长度,其实也就是集合的容量
> - 当计算的位置没有数据的时候,会直接存放数据
> - 当计算的位置,有数据时,会发生**hash冲突/hash碰撞**,解决的办法就是采用链表的结构,在对应的数据位置存放链表的头节点,对于这个链表来说,每次新加的节点会从头部位置开始加入,也就是说,数组中的永远是新节点.

HashMap扩容

加载因子:

```
static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

前面的讲述已经发现，当你空间只有仅仅为10的时候是很容易造成2个对象的hashcode 所对应的地址是一个位置的情况。这样就造成 2个 对象会形成散列桶（链表）。这时就有一个加载因子的参数，值默认为0.75，如果你hashmap的 空间有 100那么当你插入了75个元素的时候 hashmap就需要扩容了，不然的话会形成很长的散列桶结构，对于查询和插入都会增加时间，因为它要一个一个的equals比较。但又不能让加载因子很小，如0.01，这样显然是不合适的，频繁扩容会大大消耗你的内存。这时就存在着一个平衡，jdk中默认是0.75，当然负载因子可以根据自己的实际情况进行调整。

> 常用方法:

```
void clear();  //从此映射中移除所有映射关系（可选操作）。
boolean containsKey(Object key);  //如果此映射包含指定键的映射关系，则返回 true。
boolean containsValue(Object value);  //如果此映射将一个或多个键映射到指定值，则返回
true。
Set<Map.Entry<K,V>> entrySet();  //返回此映射中包含的映射关系的 Set 视图。
boolean equals(Object o);  //比较指定的对象与此映射是否相等。
V get(Object key);  //返回指定键所映射的值；如果此映射不包含该键的映射关系，则返回 null。
int hashCode();  //返回此映射的哈希码值。
boolean isEmpty();  //如果此映射未包含键-值映射关系，则返回 true。
Set<K> keySet();  //返回此映射中包含的键的 Set 视图。
V put(K key, V value); //将指定的值与此映射中的指定键关联（可选操作）。
void putAll(Map<? extends K,? extends V> m);//从指定映射中将所有映射关系复制到此映射中
(可选操作)。
V remove(Object key);  //如果存在一个键的映射关系，则将其从此映射中移除（可选操作）。
int size();  //返回此映射中的键-值映射关系数。
Collection<V> values();  //返回此映射中包含的值的 Collection 视图。
```

> **迭代和ArrayList一样**

!!!!为什么重写equals()就必须重写HashCode()!!!!

equals与hashcode间的关系是这样的:

1、如果两个对象相同（即用equals比较返回true），那么它们的hashCode值一定相同;

2、如果两个对象的hashCode相同，它们并不一定相同:Hash碰撞(即用equals比较返回false)

> 即: (1)当obj1.equals(obj2)为true时，obj1.hashCode() == obj2.hashCode()必须为true
>
> (2)当obj1.hashCode() == obj2.hashCode()为false时，obj1.equals(obj2)必须false
>
> (3)当obj1.hashCode() == obj2.hashCode()为true石，obj1.equals(obj2)不一定为true

这一块内容详解看 https://www.cnblogs.com/skywang12345/p/3324958.html

Java高并发

进程概念

进程就是正在运行的程序,它代表了程序所占用的内存区域

特点

- **独立性**

进程是系统中独立存在的实体,它可以拥有自己独立的资源,每个进程都拥有自己私有的地址空间,在没有经过进程本身允许的情况下,一个用户进程不可以直接访问其他进程的地址空间

- **动态性**

`进程与程序的区别在于,程序只是一个静态的指令集合,而进程是一个正在系统中活动的指令集合`,程序加入了时间的概念以后,称为进程,具有自己的生命周期和各种不同的状态。

- **并发性**

多个进程可以在单个处理器CPU上并发执行,多个进程之间不会互相影响.

并行和并发

并发：多个进程抢占公共资源

并行：多个CPU同时处理不同的进程

并行：指在高并发的情景中,尽可能的保证程序的可用性,减少系统不能提供服务的时间

线程概念

线程是操作系统OS能够**进行运算调度的最小单位**,它被包含在进程之中,是进程中的实际运作单位. 一个进程可以开启多个线程,其中有一个主线程来调用本进程中的其他线程

`同一时刻,只有一个线程在执行`

进程/线程区别

一个操作系统中可以有多个进程,一个进程中可以包含一个线程(单线程程序),也可以包含多个线程(多线程程序)，但是主线程只有一个。

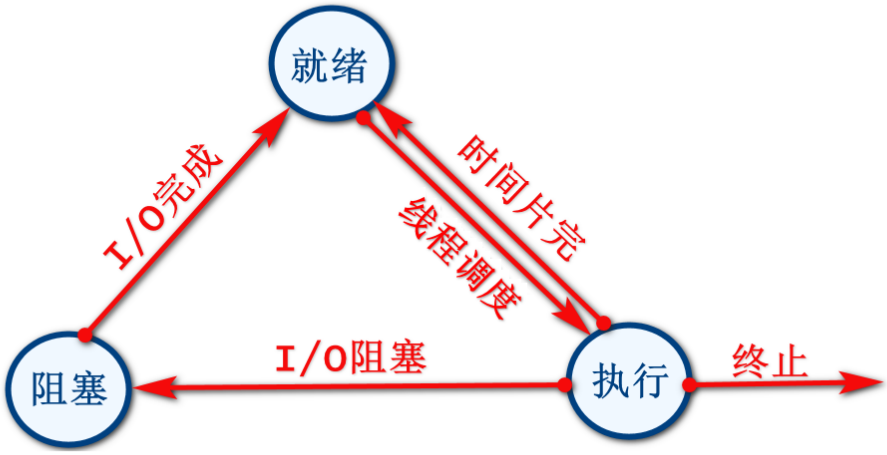
每个线程在共享同一个进程中的内存的同时,又有自己独立的内存空间

线程状态

三态模型

`就绪(可运行)状态`：线程已经准备好运行，只要获得CPU，就可立即执行 `执行(运行)状态`：线程已经获得CPU，其程序正在运行的状态 `阻塞状态`：正在运行的线程由于某些事件（I/O请求等）暂时无法执行的状态，即线程执行阻塞

线程的三态模型



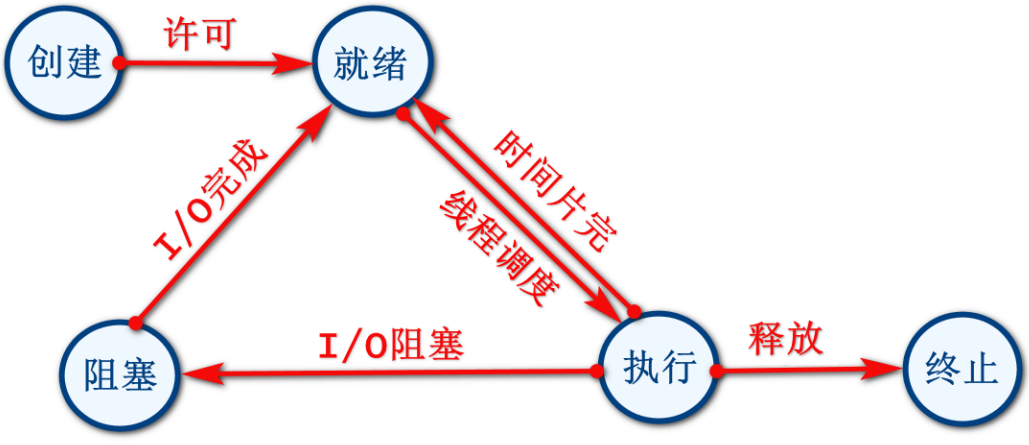
> 就绪 → 执行:为就绪线程分配CPU即可变为执行状态(想要执行，只有通过就绪) 执行 → 就绪:正在执行的线程由于时间片用完被剥夺CPU暂停执行,就变为就绪状态 执行 → 阻塞:由于发生某事件,使正在执行的线程受阻,无法执行,则由执行变为阻塞 (例如线程正在访问临界资源,而资源正在被其他线程访问) 反之,如果获得了之前需要的资源,则由阻塞变为就绪状态,等待分配CPU再次执行

五态模型

在三态模型基础上再加两种状态

- `创建状态`:线程的创建比较复杂,需要先申请PCB,然后为该线程运行分配必须的资源,并将该线程转为就绪状态插入到就绪队列中
- `终止状态`:等待OS进行善后处理,最后将PCB清零,并将PCB返回给系统

线程的五态模型



线程的生命周期

线程生命周期,主要有五种状态:

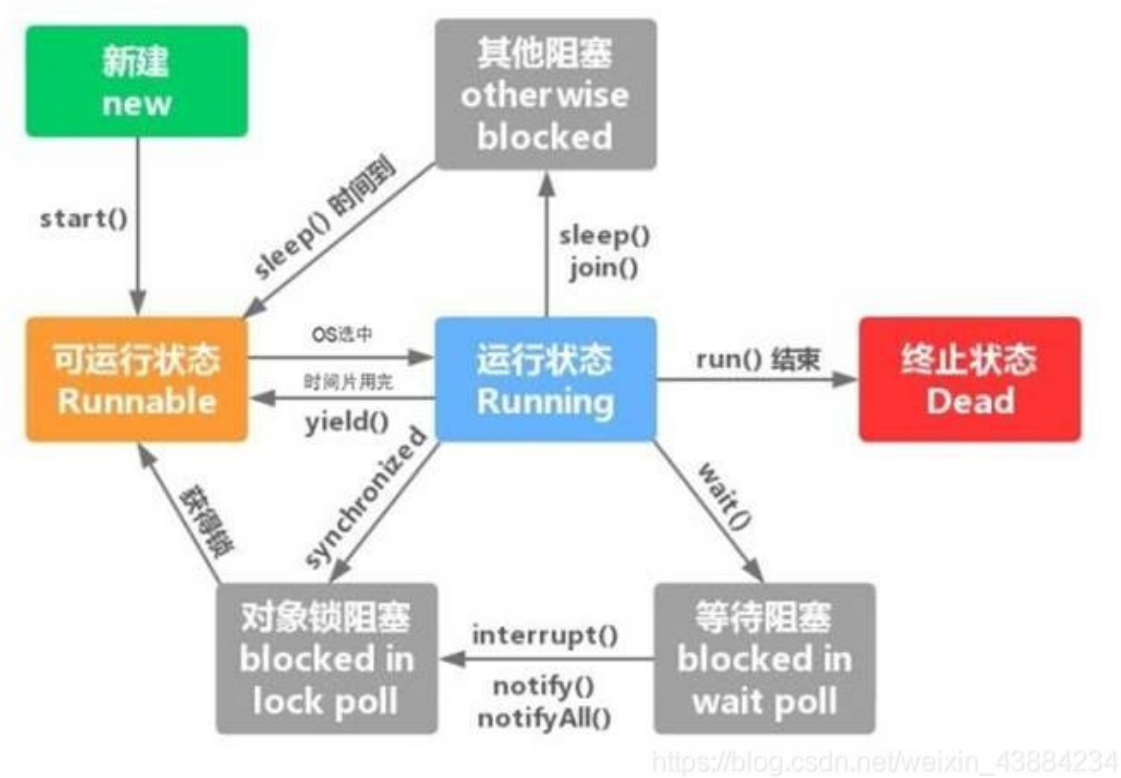
- 新建状态(New): 当线程对象创建后就进入了新建状态.如:Thread t = new MyThread();
- 就绪状态(Runnable):当调用线程对象的start()方法,线程即为进入就绪状态.

> 处于就绪(可运行)状态的线程,只是说明线程已经做好准备,随时等待CPU调度执行,并不是执行了t.start()此线程立即就会执行

- 运行状态(Running):当CPU调度了处于就绪状态的线程时,此线程才是真正的执行,即进入到运行状态

> `就绪状态是进入运行状态的唯一入口`,也就是线程想要进入运行状态状态执行,先得处于就绪状态

- 阻塞状态(Blocked):处于运状态中的线程由于某种原因,暂时放弃对CPU的使用权,停止执行,此时进入阻塞状态,直到其进入就绪状态才有机会被CPU选中再次执行. 根据阻塞状态产生的原因不同,阻塞状态又可以细分成三种: 等待阻塞:运行状态中的线程执行wait()方法,本线程进入到等待阻塞状态 同步阻塞:线程在获取synchronized同步锁失败(因为锁被其他线程占用),它会进入同步阻塞状态 其他阻塞:调用线程的sleep()或者join()或发出了I/O请求时,线程会进入到阻塞状态.当sleep()状态超时.join()等待线程终止或者超时或者I/O处理完毕时线程重新转入就绪状态
- 死亡状态(Dead):线程执行完了或者因异常退出了run()方法,该线程结束生命周期



线程创建

一、继承Thread类

1. **优点**: 编写简单,如果需要访问当前线程,无需使用Thread.currentThread()方法,直接使用this即可获得当前线程
2. **缺点**: 自定义的线程类已继承了Thread类,所以后续无法再继承其他的类

构造方法

> Thread() 分配新的Thread对象 Thread(String name) 分配新的Thread对象 Thread(Runnable target) 分配新的Thread对象 Thread(Runnable target,String name) 分配新的Thread对象

普通方法

> static Thread currentThread( )------返回对当前正在执行的线程对象的引用 long getId()------返回该线程的标识 String getName()------返回该线程的名称 void run()------如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法 static void sleep(long millions)------在指定的毫秒数内让当前正在执行的线程休眠(暂停执行) void start()------使该线程开始执行:Java虚拟机调用该线程的run()

```
public class DemoThread extends Thread{
    public static void main(String[] args){
        //如果只是调用两个线程的run(),那么会按顺序先执行完一个线程,再执行另一个线程,不会有多线程的效果
        ////run()只是一个普通方法执行的效果,也就是单线程顺序执行的效果,没有多线程的线现象
        MyThread Mythread = new MyThread("线程名字1");//此时对应的状态就是新建状态
        MyThread Mythread1 = new MyThread("线程名字2");
        MyThread Mythread2 = new MyThread("线程名字2");
        //只有调用start()才会使线程从新建状态变成可运行状态,才把线程加入就绪队列
        Mythread.run();
        Mythread1.run();
        Mythread2.run();
    }
}

class MyThread{
    /**最后:为了修改线程名称,不再使用系统分配的默认名称,需要提供含参构造,并在构造中调用父类给线程起名的构造函数*/
    public MyThread() {
        super();
    }
    public MyThread(String s) {
        super(s);
    }
    @Override
    public void run(){
        //具体实现，填写业务需求
    }
}
```

二、实现Runnable接口(推荐)

1. **优点**: 自定义的线程类只是实现了Runnable接口或Callable接口,后续还可以继承其他类,在这种方式下,多个线程可以共享同一个target对象,所以非常适合多个相同线程来处理同一份资源的情况,从而可以将CPU、代码、还有数据分开(解耦),形成清晰的模型,较好地体现了面向对象的思想
2. **缺点**: 编程稍微复杂,如想访问当前线程,则需使用Thread.currentThread()方法

继承Thread和实现Runnable接口的区别:

- a.`实现Runnable接口避免多继承局限`
- b.实现Runnable()可以更好的体现共享的概念

```
public class Thread2 {
    public static void main(String[] args) {

        MyRunnable target = new MyRunnable();
        //把接口的实现类和Thread类绑定:接口没有start()方法
        Thread thread1 = new Thread(target);
```

```
        //以多线程编程的方式启动,需要创建多个线程对象并启动
        //使用指定的构造函数修改线程的名称--使用Thread类的含参构造
        Thread thread2 = new Thread(target,"杰克");
        Thread thread3 = new Thread(target,"露丝");
        thread1.start();
        thread2.start();
        thread3.start();
    }
}

class MyRunnable implements Runnable{
    @Override
    public void run(){
        //具体实现，填写业务需求
    }
}
```

线程同步和安全问题

判断程序有没有可能出现线程安全问题,主要有以下三个条件:

`在多线程程序中 + 有共享数据 + 多条语句操作共享数据`

同步和异步概念

- 同步：体现了排队的效果，同一时刻只能有一个线程独占资源，其他没有权利的线程排队。坏处就是效率会降低，不过保证了安全。
- 异步：体现了多线程抢占资源的效果，线程间互相不等待，互相抢占资源。坏处就是有安全隐患,效率要高一些。

线程同步前提

- 前提1：**同步需要两个或者两个以上的线程**(单线程无需考虑多线程安全问题)
- 前提2：**多个线程间必须使用同一个锁**(我上锁后其他人也能看到这个锁,不然我的锁锁不住其他人,就没有了上锁的效果)

特点

1. synchronized同步关键字可以用来**修饰方法**,称为`同步方法`,使用的锁对象是this(**因为同步代码块可以保证同一个时刻只有一个线程进入,但同步方法不可以保证同一时刻只能有一个线程调用,所以使用本类代指对象this来确保同步**)
2. synchronized同步关键字可以用来**修饰代码块**,称为`同步代码块`,使用的锁对象可以任意
3. 同步的缺点是会降低程序的执行效率，但我们为了保证线程的安全,只要控制好加锁的范围，有些性能是必须要牺牲的

```
 /** 把有可能出现问题的代码包起来，一次只让一个线程执行。
 * 通过synchronized关键字实现线程同步。
 * 当多个对象操作共享数据时，可以使用同步锁解决线程安全问题。
 */
 synchronized (对象){需要同步的代码;}

public class TestDemo {
    public static void main(String[] args) {
```

```
        //创建接口实现类对象作为业务目标对象
        Demo target = new Demo();

        //将目标对象交由Thread线程对象
        Thread t1 = new Thread(target);
        Thread t2 = new Thread(target);
        Thread t3 = new Thread(target);
        Thread t4 = new Thread(target);

        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}

/**3.同步锁:相当于给容易出现问题的代码加了一把锁,从可能出现问题的代码开始,到问题代码结束
 *    位置:不能太大,也不能太小,太大,干啥都得排队,导致程序的效率太低,太小,没锁住*/
class Demo implements Runnable{

    //public synchronized void run() {被synchronized修饰的方法是同步方法

    @Override
    public void run() {//被synchronized修饰的方法是同步方法

        /*synchronized(锁对象){}--同步代码块:是指同一时间这一资源会被一个线程独享,
            大家使用的时候,都得排队,同步效果*/
        /*锁对象的要求:多线程之间必须使用同一把锁,同步代码块的方式,关于锁对象可以任意定义*/
        synchronized (Demo.class) {
            //可能会线程异步的代码
        }
    }
}
```

synchronized的特性(了解底层实现)

- 1.1 `原子性`

  所谓原子性就是指一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

- 1.2 `可见性`

  可见性是指多个线程访问一个资源时，该资源的状态、值信息等对于其他线程都是可见的。

- 1.3 `有序性`

  有序性值程序执行的顺序按照代码先后执行。

- 1.4 `可重入性`

  synchronized和ReentrantLock都是可重入锁。当一个线程试图操作一个由其他线程持有的对象锁的临界资源时，

  将会处于阻塞状态，但当一个线程再次请求自己持有对象锁的临界资源时，这种情况属于重入锁。

  通俗一点讲就是说一个线程拥有了锁仍然还可以重复申请锁。

乐观锁和悲观锁

- 悲观锁：

悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁,就像synchronized,不管三七二十一，直接上了锁就操作资源了。

- 乐观锁：

乐观锁认为竞争不总是会发生，因此它不需要持有锁，将"比较-替换"这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

两种常见的锁

synchronized 互斥锁（悲观锁，有罪假设）

采用synchronized修饰符实现的同步机制叫做互斥锁机制，它所获得的锁叫做互斥锁。 每个对象都有一个monitor(锁标记)，当线程拥有这个锁标记时才能访问这个资源，没有锁标记便进入锁池。任何一个对象系统都会为其创建一个互斥锁，这个锁是为了分配给线程的，防止打断原子操作。每个对象的锁只能分配给一个线程，因此叫做互斥锁。

ReentrantLock 排他锁（悲观锁，有罪假设）

ReentrantLock是排他锁，排他锁在同一时刻仅有一个线程可以进行访问，实际上独占锁是一种相对比较保守的锁策略，在这种情况下任何"读/读"、"读/写"、"写/写"操作都不能同时发生，这在一定程度上降低了吞吐量。然而读操作之间不存在数据竞争问题，如果"读/读"操作能够以共享锁的方式进行，那会进一步提升性能。

ReentrantReadWriteLock 读写锁（乐观锁，无罪假设）

- 因此引入了ReentrantReadWriteLock，顾名思义，ReentrantReadWriteLock是Reentrant（可重入）Read（读）Write（写）Lock（锁），我们下面称它为读写锁。 读写锁内部又分为读锁和写锁，`读锁可以在没有写锁的时候被多个线程同时持有，写锁是独占的`。 读锁和写锁分离从而提升程序性能，读写锁主要应用于读多写少的场景。
- 与互斥锁相比，**读-写锁允许对共享数据进行更高级别的并发访问。**虽然一次只有一个线程（writer 线程）可以修改共享数据，但在许多情况下，任何数量的线程可以同时读取共享数据（reader 线程）从理论上讲，与互斥锁定相比，使用读-写锁允许的并发性增强将带来更大的性能提高。

```
public class ReentrantReadWriteLocks {
    public static void main(String[] args) {
        My2 target = new My2();

        Thread t = new Thread(target, "1号窗口：");
        Thread t2 = new Thread(target, "2号窗口：");
        Thread t3 = new Thread(target, "3号窗口：");
        Thread t4 = new Thread(target, "4号窗口：");

        t.start();
        t2.start();
        t3.start();
        t4.start();
    }
}
class My2 implements Runnable {

    int sum = 100;
    /**1.定义可重入读写锁对象,静态保证全局唯一*/
    static ReentrantReadWriteLock lock = new ReentrantReadWriteLock(true);
```

```
    @Override
    public void run() {
        while (true) {
            // t t2 t3 t4都要开门，t有钥匙，进来了出去后，t2再开门干活再锁门
            // synchronized (this) {
            //2.在操作共享资源前上写锁
            lock.writeLock().lock();
            // sum=1时 t t2 t3 t4都进来
            try {
                if (sum > 0) {
                    try {
                        // t t2 t3 t4都睡了
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    // t醒了 sum=1时，sum-- = 1,sum=0
                    // t2醒了 sum=0时，sum-- = 0,sum=-1
                    // t3醒了 sum=-1时，sum-- = -1,sum=-2
                    // t4醒了 sum=-2时，sum-- = -2,sum=-3
                    System.out.println(Thread.currentThread().getName() + sum-
-);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }finally {
                lock.writeLock().unlock();//防止死锁，会自动释放，不释放就独占报错了
            }
        }
    }
}
```

需要注意的是，用synchronized修饰的方法或者语句块在代码执行完之后锁会自动释放，而用**Lock需要我们手动释放锁**，所以为了保证锁最终被释放(发生异常情况)，要把互斥区放在try内，释放锁放在finally内！

三、实现Callable接口

①、核心方法叫call()方法，可以定义返回值

②、可以抛出异常

```
public class CreateThread3 implements Callable<String> {
    @Override
    public String call() {
        //具体业务

        return "sale out";//有返回值
    }

    public static void main(String[] args) throws Exception{
        Callable<String> callable  = new CreateThread3();
        FutureTask<String> futureTask = new FutureTask<>(callable);

        Thread myThread = new Thread(futureTask);
        Thread myThread2 = new Thread(futureTask);
        Thread myThread3 = new Thread(futureTask);

        myThread.start();
        myThread2.start();
        myThread3.start();
    }
}
```

四、多线程创建

其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。线程池一般不去关闭，关闭使用shotdown()方法

为什么要用线程池： 合理利用线程池能够带来三个好处

```
1.降低资源消耗。减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。
2.提高响应速度
3.提高线程的可管理性 线程池的核心思想：线程复用。同一个线程可以被重复使用
```

- execute(Runnable任务对象) 把任务丢到线程池
- newFixedThreadPool(int nThreads) 最多n个线程的线程池
- newCachedThreadPool() 足够多的线程,使任务不必等待
- newSingleThreadExecutor() 只有一个线程的线程池

①、自动创建线程池

线程池`ExecutorService`:用来存储线程的池子,把新建线程/启动线程/销毁线程的任务都交给池来管理
`Executors`用来辅助创建线程池对象,newFixedThreadPool()创建具有参数个数的线程数的线程池

TIPS

《阿里巴巴Java开发手册》：

Executors返回的线程池对象的弊端如下： 1） FixedThreadPool 和 SingleThreadPool： 允许的请求队列长度为 Integer.MAX_VALUE，可能会堆积大量的请求，从而导致OOM。 2） CachedThreadPool： 允许的创建线程数量为 Integer.MAX_VALUE，可能会创建大量的线程，从而导致 OOM**

```
public class ThreadPool {
    public void autoThreadPool(){
        Demo target = new Demo();

        //创建有5条线程的线程池
        ExecutorService pool = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 5; i++) {
            /*execute()让线程池中的线程来执行任务,每次调用都会启动一个线程*/
            //execute()的参数就是执行的业务,也就是实现类的目标对象
            pool.execute(target);
            //pool.shutdown();
        }
    }
}
```

②、自动创建线程池

创建线程池的7个参数

> - 1、corePoolSize线程池的核心线程数
> - 2、maximumPoolSize能容纳的最大线程数
> - 3、keepAliveTime空闲线程存活时间
> - 4、unit 存活的时间单位
> - 5、workQueue 存放提交但未执行任务的队列
> - 6、threadFactory 创建线程的工厂类
> - 7、handler 等待队列满后的拒绝策略

工作队列

新任务被提交后，会先进入到此工作队列中，任务调度时再从队列中取出任务。jdk中提供了四种工作队列： `①ArrayBlockingQueue` 基于数组的有界阻塞队列，按FIFO排序。新任务进来后，会放到该队列的队尾，有界的数组可以防止资源耗尽问题。 当线程池中线程数量达到corePoolSize后，再有新任务进来，则会将任务放入该队列的队尾，等待被调度。 如果队列已经是满的，则创建一个新线程，如果线程数量已经达到maxPoolSize，则会执行拒绝策略。

`②LinkedBlockingQueue` 基于链表的无界阻塞队列（其实最大容量为Integer.MAX），按照FIFO排序。由于该队列的近似无界性，当线程池中线程数量达到corePoolSize后，再有新任务进来，会一直存入该队列，而不会去创建新线程直到maxPoolSize，因此使用该工作队列时，参数maxPoolSize其实是不起作用的。

`③SynchronousQueue` 一个不缓存任务的阻塞队列，生产者放入一个任务必须等到消费者取出这个任务。也就是说新任务进来时，不会缓存，而是直接被调度执行该任务，如果没有可用线程，则创建新线程，如果线程数量达到maxPoolSize，则执行拒绝策略。

`④PriorityBlockingQueue` 具有优先级的无界阻塞队列，优先级通过参数Comparator实现。

handler 拒绝策略

当工作队列中的任务已到达最大限制，并且线程池中的线程数量也达到最大限制，这时如果有新任务提交进来，该如何处理呢。这里的拒绝策略，就是解决这个问题的，jdk中提供了4中拒绝策略：

①CallerRunsPolicy：该策略下，在调用者线程中直接执行被拒绝任务的run方法，除非线程池已经shutdown，则直接抛弃任务； ②AbortPolicy：该策略下，直接丢弃任务，并抛出RejectedExecutionException异常。 ③DiscardPolicy：该策略下，直接丢弃任务，什么都不做。 ④DiscardOldestPolicy：该策略下，抛弃进入队列最早的那个任务，然后尝试把这次拒绝的任务放入队列

```
    public void manualThreadPool(){
        TicketSync target = new TicketSync();

        ExecutorService threadPool = new ThreadPoolExecutor(2,5,
                1L, TimeUnit.SECONDS,
                new LinkedBlockingQueue<>(3),
                Executors.defaultThreadFactory(),
                new ThreadPoolExecutor.AbortPolicy());
        threadPool.execute(target);
    }
```

ThreadLocal

ThreadLocal对外提供的方法只有三个**get()、set(T)、remove()。**

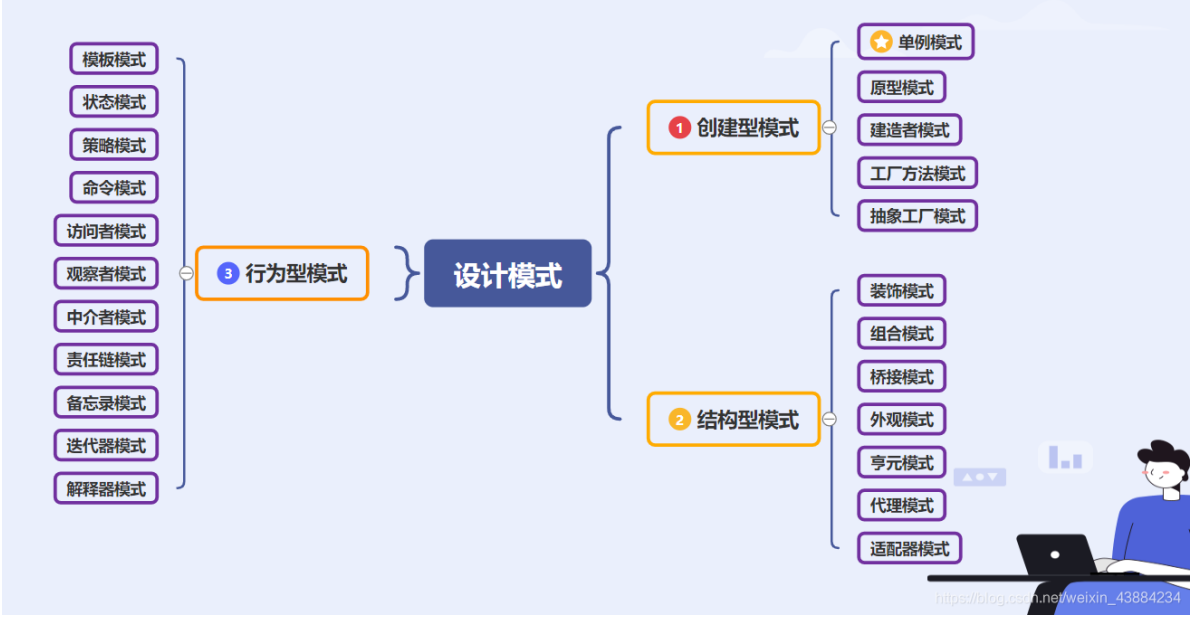
ThreadLocal是为了线程隔离，为了变量只能被固定的线程使用 用ThreadLocalMap是来存entry,因为**key为弱引用，value为强引用，会存在内存泄漏问题**：ThreadLocal在保存的时候会把自己当做Key存在ThreadLocalMap中，正常情况应该是key和value都应该被外界强引用才对，但是现在key被设计成WeakReference弱引用了。这就导致了一个问题，ThreadLocal在没有外部强引用时，发生GC时会被回收，如果创建ThreadLocal的线程一直持续运行，那么这个Entry对象中的value就有可能一直得不到回收，发生内存泄露。就比如线程池里面的线程，线程都是复用的，那么之前的线程实例处理完之后，出于复用的目的线程依然存活，所以，ThreadLocal设定的value值被持有，导致内存泄露。

解决方案：在代码最后使用remove()方法

```
    public void threadLocalPool() throws InterruptedException {
        //新建一个ThreadLocal
        final ThreadLocal<String> threadLocal = new ThreadLocal<>();
        threadLocal.set("main-thread : Hello");

        Thread thread = new Thread(() -> {
            // 获取不到主线程设置的值，所以为null
            System.out.println(threadLocal.get());
            threadLocal.set("sub-thread : World");
            System.out.println(threadLocal.get());
        });
        // 启动子线程
        thread.start();
        // 让子线程先执行完成，再继续执行主线
        thread.join();
        // 获取到的是主线程设置的值，而不是子线程设置的
        System.out.println(threadLocal.get());
        threadLocal.remove();
        System.out.println(threadLocal.get());
    }
```

设计模式



单立设计模式

概念

单例模式有很多好处，比如可节约系统内存空间，控制资源的使用。

其中单例模式最重要的：`确保对象只有一个`。简单来说，保证一个类在内存中的对象就一个

步骤

底层的实现思路一共分为了3个步骤:

- ①、对本类构造方法私有化,防止外部调用构造方法创建对象
- ②、创建全局唯一的对象,也做私有化处理
- ③、通过自定义的公共方法设置全局访问点，将创建好的对象返回(类似封装属性后的getXxx())

饿汉式(代码详解)

```
public class SingleTon {
    public static void main(String[] args) {

        //通过类名直接调用返回对象的方法
        MySingle single = MySingle.getSingle();
        MySingle single2 = MySingle.getSingle();

        //测试获取到的这两个引用类型变量是否相等
        //true,==比较的是地址值
        System.out.println(single == single2);
        System.out.println(single);
        System.out.println(single2);
    }
}

/**0.创建自己的单例程序*/
class MySingle{
```

```
    /**①、提供构造方法,并将构造方法私有化(构造方法私有化的目的:不让外界随意实例化/new本类对象)*/
    private MySingle() {}

    /**②、在类的内部,创建本类对象,并且把对象私有化
          (本资源也用static进行修饰,因为静态资源getSingle()只能调用静态资源)*/
    private static MySingle single = new MySingle(){};

    /*
     * 也就是以公共的方式向外界提供本类对象
     * 思考:对象创建好并且用公共的方式返回,那么此公共方法外界该如何调用呢?
     * 之前我们都是在外部创建好本类,通过对象进行调用的,但是现在外界无法创建本类对象
     * 解决方案:我们可以利用之前学习的静态的概念,被static关键字修饰的资源可以通过类名直接调用
     */

    /**③、对外提供一个全局访问点(用static关键字来修饰本方法,后续可以通过类名调用)*/
    public static MySingle getSingle() {
        //④、把内部创建好的对象返回到调用位置
        return single;
    }
}
```

！！！懒汉式(代码详解)！！！

饿汉式与懒汉式的区别:

- 饿汉式是不管你用不用这个类的对象,先帮你创建一个
- 懒汉式是先不给你创建这个类的对象,等你需要时再帮你创建--**延迟加载**的思想

```
public class SingleTons {
    public static void main(String[] args) {

        MySingle2 my1 = MySingle2.getMySingle2();
        MySingle2 my2 = MySingle2.getMySingle2();

        System.out.println(my1 == my2);
        System.out.println(my1);
        System.out.println(my2);
    }
}

/**创建单例类*/
class MySingle2 {

    /**①、私有化构造方法--为了防止外界调用此构造方法创建对象*/
    private MySingle2(){}

    /**②、在类的内部创建好引用类型成员变量  --注意私有化 -- 延迟加载
          本处引用类型变量single也需要用static修饰,因为静态资源只能调用静态资源*/
    private static MySingle2 single;

    //static Object obj = new Object();

    /*
     * 问题:程序中有共享资源single,并且有多条语句(3句)操作了共享资源
     * 此时single共享资源一定会存在多线程数据安全问题
```

```
     * 解决方案:加同步锁-同步代码块-同步方法[由于方法中的所有代码都被同步了,所以可以直接变成同步方法]
     * 锁的位置:操作共享资源的多条语句
     * 锁对象:static中不能使用this & 外部创建了obj,注意是static的
     */

    /**③、对外提供全局访问点，对外提供公共的getXxx(),返回本类对象
     * 注意要使用static来修饰本公共方法,方便后续通过类名之间调用*/
    public static synchronized MySingle2 getMySingle2() {

        /* 同步代码块:静态区域内不能使用this关键字,因为静态资源优先于对象this加载
         * 因为静态资源属于类资源,随着类的加载而加载,而this指的是本类对象*/

        //synchronized (this) {//报错
        //synchronized (obj) {

        /*④、当用户调用此方法时,才说明用到对象了,那么再将本类对象返回
        如果调用此方法时为null,说明之前没有new过,保存的是默认值null*/
        if(single == null) {
            single = new MySingle2(){};
        }
        //将本类对象返回
        return single;
        //}
    }
}
```

懒汉式优化(经典):双重锁校验

```
class DoubleLock implements Serializable {

    /**
     * 双重锁校验的单例
     */
    private static final long serialVersionUID = -2661668900873276710L;
    /** volatile防止指令重排序，内存可见
       (缓存中的变化及时刷到主存，并且其他的内存失效，必须从主存获取)*/
    public static volatile DoubleLock doubleLock = null;

    private DoubleLock(){
        //构造器必须私有   不然直接new就可以创建
    }

    public static DoubleLock getInstance(){
        //第一次判断，假设会有好多线程，如果doubleLock没有被实例化，那么就会到下一步获取锁，只有一个能获取到，
        //如果已经实例化，那么直接返回了，减少除了初始化时之外的所有锁获取等待过程
        if(doubleLock == null){
            synchronized (DoubleLock.class){
                //第二次判断是因为假设有两个线程A、B,两个同时通过了第一个if，然后A获取了锁，进入,然后判断doubleLock是null，
                //他就实例化了doubleLock，然后他出了锁，
                //这时候线程B经过等待A释放的锁，B获取锁了，如果没有第二个判断，那么他还是会去new DoubleLock()，
                // 再创建一个实例，所以为了防止这种情况，需要第二次判断
                if(doubleLock == null){
                    //下面这句代码其实分为三步：
```

```
                    //1.开辟内存分配给这个对象
                    //2.初始化对象
                    //3.将内存地址赋给虚拟机栈内存中的doubleLock变量
                    //注意上面这三步，第2步和第3步的顺序是随机的，这是计算机指令重排序的问题
                    //假设有两个线程，其中一个线程执行下面这行代码，如果第三步先执行了，就会把没有初始化的内存赋值给doubleLock
                    //然后恰好这时候有另一个线程执行了第一个判断if(doubleLock == null)，然后就会发现doubleLock指向了一个内存地址
                    //这另一个线程就直接返回了这个没有初始化的内存，所以要防止第2步和第3步重排序
                    doubleLock = new DoubleLock();
                }
            }
        }
        return doubleLock;
    }
}
```

拓展：volatile关键字

被volatile修饰的变量能够保证每个线程能够获取该变量的最新值，从而避免出现数据脏读的现象。

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

①、保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

- （1）当写一个volatile变量时，JMM会把该线程本地内存中的变量强制刷新到主内存中去；
- （2）这个写会操作会导致其他线程中的volatile变量缓存无效。

②、禁止进行指令重排序。

重排序是指编译器和处理器为了优化程序性能而对指令序列进行排序的一种手段。重排序需要遵守一定规则：

- （1）**重排序操作不会对存在数据依赖关系的操作进行重排序。**

  比如：a=1;b=a; 这个指令序列，由于第二个操作依赖于第一个操作，所以在编译时和处理器运行时这两个操作不会被重排序。

- （2）**重排序是为了优化性能，但是不管怎么重排序，单线程下程序的执行结果不能被改变**

  比如：a=1;b=2;c=a+b这三个操作，第一步（a=1)和第二步(b=2)由于不存在数据依赖关系，所以可能会发生重排序，但是c=a+b这个操作是不会被重排序的，因为需要保证最终的结果一定是c=a+b=3。重排序在单线程下一定能保证结果的正确性，但是在多线程环境下，可能发生重排序，影响结果

volatile原理

volatile可以保证线程可见性且提供了一定的有序性，但是无法保证原子性。

在JVM底层volatile是采用"内存屏障"来实现的。观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，**加入volatile关键字时，会多出一个lock前缀指令，lock前缀指令实际上相当于一个内存屏障（也成内存栅栏）**，内存屏障会提供3个功能;

- （1）它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面;即在执行到内存屏障这句指令时，在它前面的操作已经全部完成;
- （2）它会强制将对缓存的修改操作立即写入主存;

- （3）如果是写操作，它会导致其他CPU中对应的缓存行无效。

总结

- a.当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；
- b.在进行指令优化时，不能将对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。即执行到volatile变量时，其前面的所有语句都执行完，后面所有语句都未执行。且前面语句的结果对volatile变量及其后面语句可见。

Java注解与反射

注解优势：

框架可以根据注解去自动生成很多代码，从而减少代码量，程序更易读

分类

JDK注解

JDK注解的注解，就5个：

1. `@Override` :用来标识重写方法
2. `@Deprecated` 标记就表明这个方法已经过时了，但我就要用，别提示我过期
3. `@SuppressWarnings("deprecation")` 忽略警告
4. `@SafeVarargs` jdk1.7出现，堆污染，不常用
5. `@FunctionalInterface` jdk1.8出现，配合函数式编程拉姆达表达式，不常用

元注解

定义注解的注解，就5个：

1. `@Target` :注解用在哪里：类上、方法上、属性上(里面的值可以多个)
2. `@Retention` :注解的生命周期：源文件中、.class字节码文件中、运行中
3. `@Inherited` :允许子注解继承
4. `@Documented` :生成javadoc时会包含注解，不常用
5. `@Repeatable` :注解为可重复类型注解，可以在同一个地方多次使用，不常用

描述注解存在的位置

`@Target ElementType.class`

1. `ElementType.TYPE` 应用于类的元素
2. `ElementType.METHOD` 应用于方法级
3. `ElementType.FIELD` 应用于字段或属性(成员变量)
4. `ElementType.ANNOTATION_TYPE` 应用于注释类型
5. `ElementType.CONSTRUCTOR` 应用于构造函数
6. `ElementType.LOCAL_VARIABLE` 应用于局部变量
7. `ElementType.PACKAGE` 应用于包声明
8. `ElementType.PARAMETER` 应用于方法的参数

描述注解保留时间长短

`@Retention RetentionPolicy.class`

定义了该注解被保留的时间长短，某些注解仅出现在源代码中，而被编译器丢弃；而另一些却被编译在class文件中； 编译在class文件中的注解可能会被虚拟机忽略，而另一些在class被装载时将被读取。为何要分有没有呢？没有时，反射就拿不到，从而就无法去识别处理。

- `SOURCE` 在源文件中有效（即源文件保留）
- `CLASS` 在class文件中有效（即class保留）
- `RUNTIME` 在运行时有效（即运行时保留）

自定义注解

注解@Tests可以有多个位置,多个值,可以使用"{ , }"的格式,@Target中维护的是ElementType[]数组

```
@Target({ElementType.METHOD,ElementType.FIELD,ElementType.TYPE})
@Retention(RetentionPolicy.SOURCE)
@interface Tests{
    /** 自定义注解还可以添加功能--给注解添加属性
     *注意int age();不是方法的定义,而是给自定义注解中定义age属性的语法
     *如果为了使用注解时方便,还可以给age属性指定默认值,这样就可以直接使用,格式:"int age()
default 0;"
     */
    //int age();//报错,要么自己赋值，要么赋予默认值

    int age() default 0;
    //String sex();


    /** 注解中还可以添加功能--可以定义特殊属性value
     *  特殊属性的定义方式与别的属性相同,主要是使用方式不同
     *  使用此注解给属性赋值的时候可以不用写成"@Tests(value = "apple")",
     *  格式可以简化成"@Tests("apple")",直接写值（没有冲突时）
     *  但是在自定义注解类中的赋予默认值不能简写,如果自定义了默认值,使用时可以不用赋值直接使用
属性的默认值
     */
    String value() default "lemon";
}

@Test//使用方式
public ClassName{

}
```

反射

JVM内存中的两大对象

字节码对象:

每个类在加载(将类读到内存)时都会创建一个字节码对象,且这个对象在一个JVM内存中是唯一的.此对象中存储的是类的结构信息.

类的实例对象

- 1) 客观事务在内存中的呈现(堆内存中的一块区域)
- 2) 类的实例对象在同一个JVM内存中可以有多份.

Java创建对象方式

- 1）通过new关键字创建
- 2）通过反射创建(首先要先获取字节码对象)

Java中对象的作用?

- 1) 存储数据(变量:类变量,实例变量,参数变量,局部变量)

a) Pojo (普通的java对象)

b) Vo (值对象)

- 2) 执行业务逻辑(方法):各司其职,各尽所能.

  a) Controller --前台--网页

  b) Service -- 业务--校验

  c) Dao -- 数据--数据库

  建议:面向对象设计时不要设计一个大而全的对象.

类加载过程(TODO)

反射前提：获取解码对象

> 1. Class.forName("类的全路径");
> 2. 类名.class
> 3. 对象.getClass();

常用方法

```
获取包名 类名
clazz.getPackage().getName()//包名
clazz.getSimpleName()//类名
clazz.getName()//完整类名

获取成员变量定义信息
getFields()//获取所有公开的成员变量,包括继承变量
getDeclaredFields()//暴力获取本类定义的成员变量,包括私有,但不包括继承的变量
getField(变量名)
getDeclaredField(变量名)

获取构造方法定义信息
getConstructor(参数类型列表)//获取公开的构造方法
getConstructors()//获取所有的公开的构造方法
getDeclaredConstructors()//暴力获取所有的构造方法,包括私有
getDeclaredConstructor(int.class,String.class)

获取方法定义信息
getMethods()//获取所有可见的方法,包括继承的方法
getMethod(方法名,参数类型列表)
getDeclaredMethods()//暴力获取本类定义的的方法,包括私有,不包括继承的方法
```

```
getDeclaredMethod(方法名,int.class,String.class)

反射新建实例
clazz.newInstance();//执行无参构造创建对象
clazz.newInstance(666,"海绵宝宝");//执行含参构造创建对象
clazz.getConstructor(int.class,String.class)//获取构造方法

反射调用成员变量
clazz.getDeclaredField(变量名);//暴力获取变量
clazz.setAccessible(true);//使私有成员允许访问
f.set(实例,值);//为指定实例的变量赋值,静态变量,第一参数给null
f.get(实例);//访问指定实例变量的值,静态变量,第一参数给null

反射调用成员方法
Method m = Clazz.getDeclaredMethod(方法名,参数类型列表);
m.setAccessible(true);//使私有方法允许被调用
m.invoke(实例,参数数据);//让指定实例来执行该方法
```

Socket网络编程

服务器端：ServerSocket

> 注意:
>
> - 1.使用ServerSocket需要导包java.net.ServerSocket;
> - 2.操作会抛出异常
> - 3.指定的端口号范围是:0-65535,而0-1024是系统端口号,不能指定

```
public class Server {
    public static void main(String[] args) throws Exception {
        final int frequency = 5;

        //①.启动服务器,指定端口号为1230,等待客户端的连接
        ServerSocket serverSocket = new ServerSocket(1230);
        System.out.println("服务器已开启！等待客户端发送数据");
        //②.等待接收客户端的连接请求,并建立数据通信通道(等待连接和建立通道)
        Socket socket = serverSocket.accept();

        System.out.print("已接收，请输入想回复的数据：");
        //③.获取到读取流,接收并读取客户端发来的数据(接收并读取)
        InputStream in = socket.getInputStream();

        //通过循环挨个读取显示读到的内容
        for(int i = 0; i < frequency; i++) {
            //int b = in.read();//此方法读取的结果是把字符转成数字
            //为了直接显示读取到的字符,需要强制类型转换(大转小,int转char)
            //不满足5个字符read()会死等，必须输入5个字符才可以
            char data = (char) in.read();
            //print()同行输出,注意细节哦
            System.out.print(data);
        }

        //⑤.给客户端发送数据
        OutputStream out = socket.getOutputStream();
        String input = new Scanner(System.in).nextLine();
        out.write(input.getBytes());
```

```
        //刷新数据
        out.flush();

        //④.释放资源
        /*注意关流的顺序,后出现的先关闭*/
        in.close();
        serverSocket.close();
    }
}
```

flush和close区别

`close()`关闭流对象，但是先刷新一次缓冲区，关闭之后，流对象不可以继续再使用了。

`flush()`仅仅是刷新缓冲区(一般写字符时要用,因为字符是先进入的缓冲区)，流对象还可以继续使用

客户端：Socket

注意:

> 1. 1.使用Socket需要导包java.net.Socket
> 2. 2.此操作会抛出异常
> 3. 3.如果使用的是本机的IP,地址是固定值,用来测试时使用127.0.0.1

```
public class Client {
    public static void main(String[] args) throws Exception {
        final int frequency = 5;

        //①.指定要连接的服务器,需要同时指定服务器的IP & Port
        Socket socket = new Socket("127.0.0.1",1230);
        System.out.println("客户端已启动，与服务器连接成功！请发送数据");

        //②.给服务器端发送数据
        OutputStream out = socket.getOutputStream();
        //把要输出的数据字符串转变成byte[]的形式进行输出
        String input = new Scanner(System.in).nextLine();
        out.write(input.getBytes());
        //把流里的数据刷新给服务器
        out.flush();

        //④.读取从服务器端返回的数据
        InputStream in = socket.getInputStream();
        for (int i = 0; i < frequency; i++) {

            //为了显示字符而不是数字,强制类型转换成char
            char c = (char) in.read();
            //不换行展示获取到的数据
            System.out.print(c);
        }
        //③.释放资源
        out.close();
        socket.close();
    }
}
```