

JAVA

1. JDK是什么？有哪些内容组成？

JDK是Java开发工具包

- JVM虚拟机：Java程序运行的地方
- 核心类库：Java已经写好的东西，我们可以直接用。
- 开发工具：javac、java、jdb、jhat...

2. JRE是什么？有哪些内容组成？

JRE是Java运行环境

JVM、核心类库、运行工具

3. JDK, JRE, JVM三者的包含关系

- JDK包含了JRE
- JRE包含了JVM

从内存的角度去解释：

基本数据类型：数据值是存储在自己的空间中

特点：赋值给其他变量，也是赋的真正的值。

引用数据类型：数据值是存储在其他空间中，
自己空间中存储的是地址值。

特点：赋值给其他变量，赋的地址值。

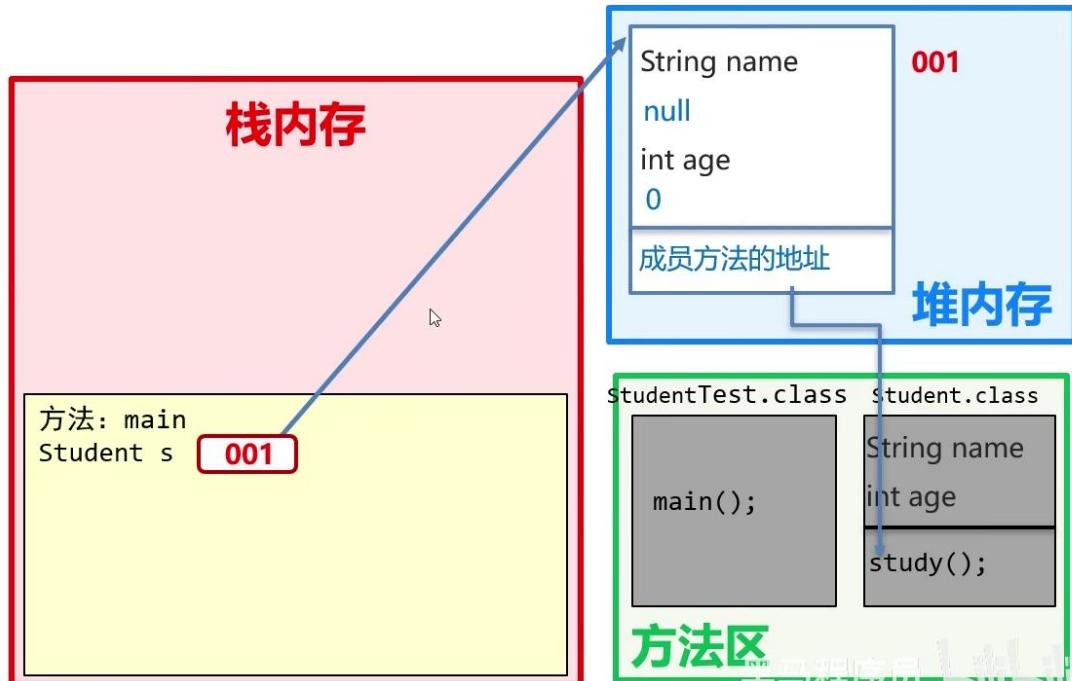
```
int[] arr1 = {1,2,3};  
int[] arr2 = arr1;
```

封装

封装

对象代表什么，就得封装对应的数据，并提供数据对应的行为

-
- 1. 加载class文件
 - 2. 申明局部变量
 - 3. 在堆内存中开辟一个空间
 - 4. 默认初始化
 - 5. 显示初始化
 - 6. 构造方法初始化
 - 7. 将堆内存中的地址值赋值给左边的局部变量



```
Java
Student student = new Student();
//student 中存的是堆内存中对象的地址
student = null;
//将 student 存地址赋为空，断开了 student 与堆内存的连接
```

String

this的作用：区分局部变量和成员变量

this的本质：代表方法调用者的地址值

StringBuilder, 打印对象不是地址值而是属性值

1. String

表示字符串的类，定义了很多操作字符串的方法

2. StringBuilder

一个可变的操作字符串的容器。

可以高效的拼接字符串，还可以将容器里面的内容反转。

3. StringJoiner

JDK8出现的一个可变的操作字符串的容器，可以高效，方便的拼接字符串。

在拼接的时候，可以指定间隔符号，开始符号，结束符号。

- 问题：下列代码的运行结果是？

```
public class Test3 {  
    public static void main(String[] args) {  
        String s1 = "abc"; 记录串池中的地址值  
        String s2 = "ab";  
        String s3 = s2 + "c"; 新new出来的对象  
        System.out.println(s1 == s3);  
    }  
}
```

字符串拼接的时候，如果有变量：

JDK8以前：系统底层会自动创建一个StringBuilder对象，然后再调用其append方法完成拼接。

拼接后，再调用其toString方法转换为String类型，而toString方法的底层是直接new了一个字符串对象。

JDK8版本：系统会预估要字符串拼接之后的总大小，把要拼接的内容都放在数组中，此时也是产生一个新的字符串。

字符串原理小结

扩展底层原理1：字符串存储的内存原理

- 直接赋值会复用字符串常量池中的
- new出来不会复用，而是开辟一个新的空间

扩展底层原理5：StringBuilder源码分析

- 默认创建一个长度为16的字节数组
- 添加的内容长度小于16，直接存
- 添加的内容大于16会扩容（原来的容量*2+2）
- 如果扩容之后还不够，以实际长度为准

扩展底层原理2：==号比较的到底是什么？

- 基本数据类型比较数据值
- 引用数据类型比较地址值

扩展底层原理3：字符串拼接的底层原理

- 如果没有变量参与，都是字符串直接相加，编译之后就是拼接之后的结果，会复用串池中的字符串。
- 如果有变量参与，会创建新的字符串，浪费内存。

扩展底层原理4：StringBuilder提高效率原理图

- 所有要拼接的内容都会往StringBuilder中放，不会创建很多无用的空间，节约内存

static

static内存图

```
public class Student {  
    String name;  
    int age;  
    static String teacherName;  
    public void show() {  
        System.out.println(name + "..." +  
                           age + "..." + TeacherName);  
    }  
}
```

```
public class Test1Static {  
    public static void main(String[] args) {  
        Student.teacherName = "阿玮老师";  
        Student s1 = new Student();  
        s1.name = "张三";  
        s1.age = 23;  
        s1.show();  
  
        Student s2 = new Student();  
        s2.show();  
    }  
}
```

栈内存



堆内存



工具类

帮助我们做-

01

类名见名知意

02

私有化构造方法

03

方法定义为静态

- 静态方法**只能**访问静态变量和静态方法
- 非静态方法**可以**访问静态变量或者静态方法，**也可以**访问非静态的成员变量和非静态的成员方法
- 静态方法中是没有this关键字

总结：静态方法中，只能访问静态。

非静态可以访问所有。

静态中没有this关键字

继承

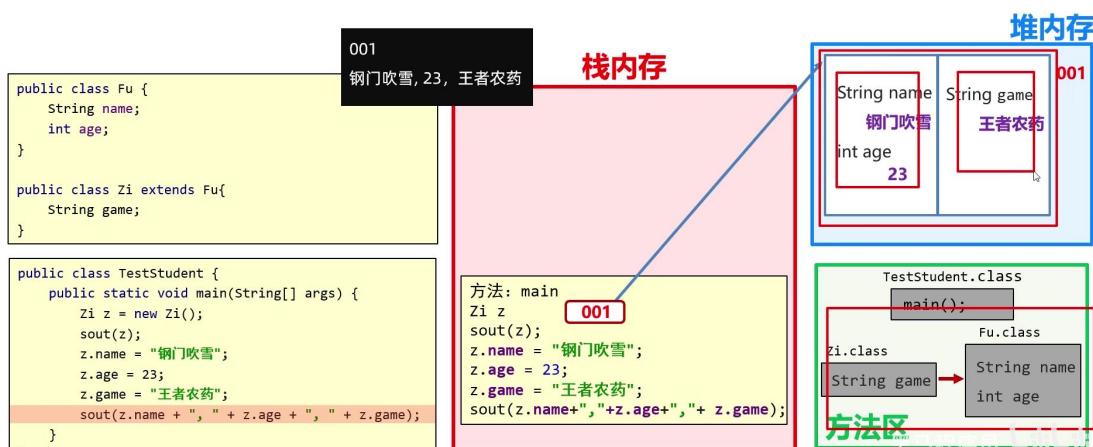
当类与类之间，存在相同（共性）的内容，并满足子类是父类中的一种，就可以考虑使用继承，来优化代码

- Java只支持**单继承**，不支持**多继承**，但支持**多层继承**。

多层继承：子类 A 继承父类 B，父类B 可以继承父类 C

每一个类都直接或者间接的继承于Object

继承的内存图



提高效率



子类到底能继承父类中的哪些内容？



1. 继承中成员变量访问特点：就近原则。

先在局部位置找，本类成员位置找，父类成员位置找，逐级往上。

2. 如果出现了重名的成员变量怎么办？

```

System.out.println(name);
System.out.println(this.name);
System.out.println(super.name);

```

从局部位置开始往上找

从本类成员位置开始往上找

从父类成员位置开始往上找

方法的重写

当父类的方法不能满足子类现在的需求时，需要进行方法重写

书写格式

在继承体系中，子类出现了和父类中一模一样的方法声明，我们就称子类这个方法是重写的方法。

@Override重写注解

1. @Override是放在重写后的办法上，校验子类重写时语法是否正确。

方法重写注意事项和要求

1. 重写方法的名称、形参列表必须与父类中的一致。
2. 子类重写父类方法时，访问权限子类必须大于等于父类（暂时了解：空着不写 < protected < public）
3. 子类重写父类方法时，返回值类型子类必须小于等于父类
- 4. 建议：重写的方法尽量和父类保持一致。**
5. 只有被添加到虚方法表中的方法才能被重写

1. 继承中成员方法的访问特点：

this调用：就近原则。

super调用：直接找父类。

2. 什么是方法重写？

在继承体系中，子类出现了和父类中一模一样的方法声明，我们就称子类的这个方法是重写的方法。

3. 方法重写建议加上哪个注解，有什么好处？

@Override注解可以校验重写是否正确，同时可读性好。

4. 重写方法有哪些基本要求？

- 子类重写的方法尽量跟父类中的方法保持一致。
- 只有虚方法表里面的方法可以被重写

5. 方法重写的本质？

覆盖虚方法表中的方法

继承中：构造方法的访问特点

- 父类中的构造方法不会被子类继承。
- 子类中所有的构造方法默认先访问父类中的无参构造，再执行自己。

为什么？

- 子类在初始化的时候，有可能会使用到父类中的数据，如果父类没有完成初始化，子类将无法使用父类的数据。
- 子类初始化之前，一定要调用父类构造方法先完成父类数据空间的初始化。

怎么调用父类构造方法的？

- 子类构造方法的第一行语句默认都是：super()，不写也存在，且必须在第一行。
- 如果想调用父类有参构造，必须手动写super进行调用。

this, super

this、super使用总结

- **this**: 理解为一个变量，表示当前方法调用者的地址值；
- **super**: 代表父类存储空间。

| 关键字 | 访问成员变量 | 访问成员方法 | 访问构造方法 |
|-------|------------------------|-----------------------------|------------------------|
| this | this.成员变量 访问本类成员变量 | this.成员方法(...) 访问本类成员方法 | this(...) 访问本类构造方法 |
| super | super.成员变量 访问父类成员变量 | super.成员方法(...) 访问父类成员方法 | super(...) 访问父类构造方法 |

多态

1. 什么是多态?

对象的多种形态。

2. 多态的前提?

- 有继承/实现关系
- 有父类引用指向子类对象
- 有方法的重写

3. 多态的好处?

使用父类型作为参数，可以接收所有子类对象，体现多态的扩展性与便利。

- 有父类引用指向子类对象

```
Fu f = new Zi () ;
```

```
Animal a = new Dog();  
//调用成员变量：编译看左边，运行也看左边  
//编译看左边：javac编译代码的时候，会看左边的父类中有没有这个变量，如果有，编译成功，如果没有编译失败。  
//运行也看左边：java运行代码的时候，实际获取的就是左边父类中成员变量的值  
System.out.println(a.name); //动物  
  
//调用成员方法：编译看左边，运行看右边  
//编译看左边：javac编译代码的时候，会看左边的父类中有没有这个方法，如果有，编译成功，如果没有编译失败。  
//运行看右边：java运行代码的时候，实际上运行的是子类中的方法。  
a.show(); //Dog --- show方法
```

1. 多态的优势

方法中，使用父类型作为参数，可以接收所有子类对象

2. 多态的弊端是什么？

不能使用子类的特有功能

3. 引用数据类型的类型转换，有几种方式？

自动类型转换、强制类型转换

```
Person p = new Student();
```

4. 强制类型转换能解决什么问题？

```
Student s = (Student)p;
```

- 可以转换成真正的子类类型，从而调用子类独有功能。
- 转换类型与真实对象类型不一致会报错
- 转换的时候用instanceof关键字进行判断

包，final

1. 包的作用？

包就是文件夹，用来管理各种不同功能的Java类

2. 包名书写的规则？

公司域名反写 + 包的作用，需要全部英文小写，见名知意

3. 什么是全类名？

包名 + 类名

4. 什么时候需要导包？什么时候不需要导包？

- 使用同一个包中的类时，不需要导包。
- 使用java.lang包中的类时，不需要导包。
- 其他情况都需要导包
- 如果同时使用两个包中的同名类，需要用全类名。

final

方法

表明该方法是最终方法，不能被重写

类

表明该类是最终类，不能被继承

变量

叫做常量，只能被赋值一次

常量

实际开发中，常量一般作为系统的配置信息，方便维护，提高可读性。

常量的命名规范：

- 单个单词：全部大写
- 多个单词：全部大写，单词之间用下划线隔开

细节：

final修饰的变量是基本类型：那么变量存储的数据值不能发生改变。

final修饰的变量是引用类型：那么变量存储的地址值不能发生改变，对象内部的可以改变。

1. 代码块的分类：

局部代码块，构造代码块，静态代码块

2. 局部代码块的作用

提前结束变量的生命周期（已淘汰）

3. 构造代码块的作用

抽取构造方法中的重复代码（不够灵活）

4. 静态代码块的作用

数据的初始化（重点）

抽象类

1. 抽象类的作用是什么样的？

抽取共性时，无法确定方法体，就把方法定义为抽象的。

强制让子类按照某种格式重写。

抽象方法所在的类，必须是抽象类。

2. 抽象类和抽象方法的格式？

public **abstract** 返回值类型 方法名(参数列表);

public **abstract** class 类名{}

3. 继承抽象类有哪些要注意？

- 要么重写抽象类中的所有抽象方法

- 要么是抽象类

接口

- 接口和接口的关系

继承关系，可以单继承，也可以多继承

接口中成员的特点

- 成员变量

只能是常量

默认修饰符: **public static final**

- 构造方法

没有

- 成员方法

只能是抽象方法

默认修饰符: **public abstract**

- **JDK7以前**: 接口中只能定义抽象方法。

- **JDK8的新特性**: 接口中可以定义有方法体的方法。

- **JDK9的新特性**: 接口中可以定义私有方法。 ↳

JDK8以后接口中新增的方法

- 允许在接口中定义默认方法，需要使用关键字 **default** 修饰

作用：解决接口升级的问题

接口中**默认方法**的定义格式：

- 格式: **public default 返回值类型 方法名(参数列表) { }**
- 范例: **public default void show() { }**

接口中默认方法的**注意事项**:

- 默认方法不是抽象方法，所以不强制被重写。但是如果被重写，重写的时候去掉**default**关键字
- **public**可以省略，**default**不能省略
- 如果实现了多个接口，多个接口中存在相同名字的默认方法，子类就必须对该方法进行重写



JDK8以后接口中新增的方法

- 允许在接口中定义静态方法，需要用static修饰

接口中**静态方法**的定义格式：

- 格式：public **static** 返回值类型 方法名(参数列表) { }
- 范例：public **static** void show() { }

接口中静态方法的**注意事项**：

- 静态方法只能通过接口名调用，不能通过实现类名或者对象名调用
- public可以省略，**static不能省略**

1. JDK7以前：接口中只能定义抽象方法。

2. JDK8：接口中可以定义有方法体的方法。 (默认、静态)

3. JDK9：接口中可以定义**私有方法**。

4. 私有方法分为两种：普通的私有方法，静态的私有方法

1. 接口代表规则，是行为的抽象。想要让哪个类拥有一个行为，就让这个类实现对应的接口就可以了。

2. 当一个方法的参数是接口时，可以传递接口**所有实现**类的对象，这种方式称之为**接口多态**。

1. 当一个接口中抽象方法过多，但是我只要使用其中一部分的时候，就可以适配器设计模式
2. 书写步骤：

编写中间类XXXAdapter 实现对应的接口

对接口中的抽象方法进行空实现

让真正的实现类继承中间类，并重写需要用的方法

为了避免其他类创建适配器类的对象，中间的适配器类用abstract进行修饰

内部类

1. 什么是内部类？

写在一个类里面的类就叫做内部类

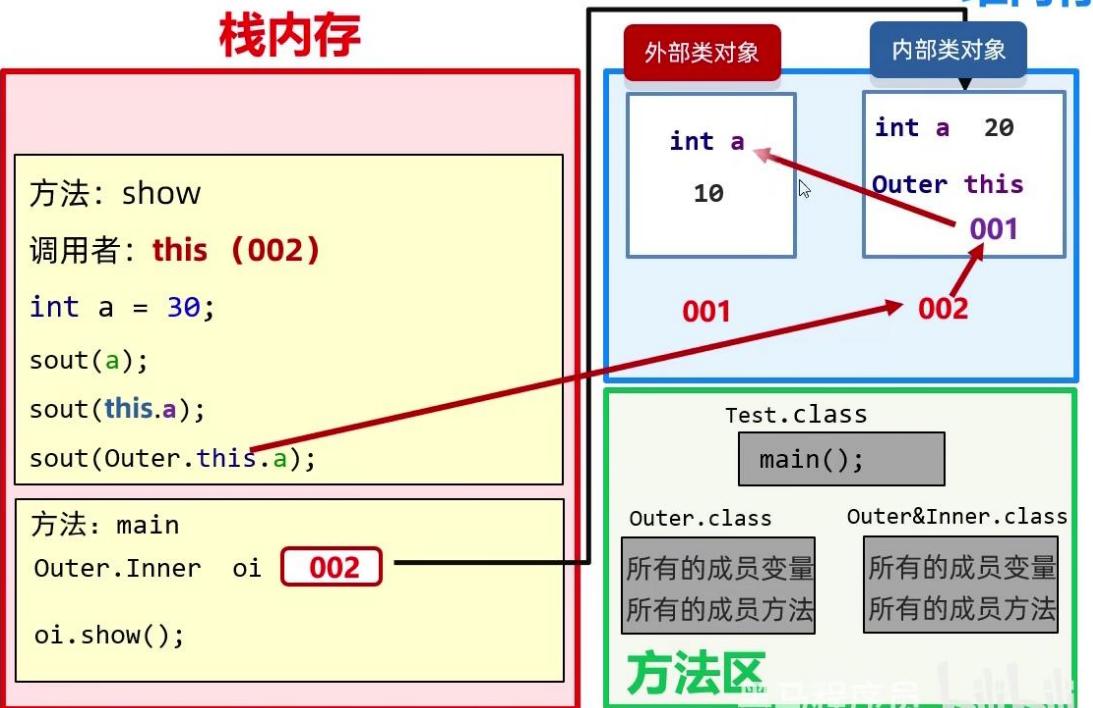
2. 什么时候用到内部类？

B类表示的事物是A类的一部分，且B单独存在没有意义。

比如：汽车的发动机，ArrayList的迭代器，人的心脏等等

堆内存

栈内存



1. 内部类的分类?

成员内部类，静态内部类，局部内部类，匿名内部类。

2. 什么是成员内部类?

写在成员位置的，属于外部类的成员。

3. 获取成员内部类对象的两种方式?

方式一：当成员内部类被private修饰时。

在外部类编写方法，对外提供内部类对象

方式二：当成员内部类被非私有修饰时，直接创建对象。

`Outer.Inner oi = new Outer().new Inner();`

4. 外部类成员变量和内部类成员变量重名时，在内部类如何访问?

`System.out.println(Outer.this.变量名);`

1. 什么是静态内部类？

静态内部类是一种特殊的成员内部类。

2. 直接创建静态内部类对象的方式？

Outer.Inner oi = new Outer.Inner();

3. 如何调用静态内部类中的方法？

非静态方法：先创建对象，用对象调用

静态方法：外部类名.内部类名.方法名();

1. 什么是匿名内部类？

隐藏了名字的内部类，可以写在成员位置，也可以写在局部位置。

2. 匿名内部类的格式？

```
new 类名或者接口名() {  
    重写方法;  
};
```

3. 格式的细节

包含了继承或实现，方法重写，创建对象。

整体就是一个类的子类对象或者接口的实现类对象

4. 使用场景

当方法的参数是接口或者类时，

以接口为例，可以传递这个接口的实现类对象，

如果实现类只要使用一次，就可以用匿名内部类简化代码。

API

1. System: 也是一个工具类，提供了一些与系统相关的方法
2. 时间原点：1970年1月1日 0:0:0， 我国在东八区，有8小时时差。
3. 1秒 = 1000 毫秒
4. 常见方法如下：
5. **exit**: 停止虚拟机
6. **currentTimeMillis**: 获取当前时间的毫秒值
7. **arraycopy**: 拷贝数组

| Runtime | |
|--|---------------------------|
| 方法名 | 说明 |
| public static Runtime getRuntime() | 当前系统的运行环境对象 |
| public void exit(int status) | 停止虚拟机 |
| public int availableProcessors() | 获得CPU的线程数 |
| public long maxMemory() | JVM能从系统中获取总内存大小 (单位byte) |
| public long totalMemory() | JVM已经从系统中获取总内存大小 (单位byte) |
| public long freeMemory() | JVM剩余内存大小 (单位byte) |
| public Process exec(String command) | 运行cmd命令 |

1. Object是Java中的顶级父类。

所有的类都直接或间接的继承于Object类。

2. `toString()`: 一般会重写，打印对象时打印属性

3. `equals()`: 比较对象时会重写，比较对象属性值是否相同

4. `clone()`: 默认浅克隆。

如果需要深克隆需要重写方法或者使用第三方工具类。

1. Objects是一个对象工具类，提供了一些操作对象的方法

2. `equals(对象1, 对象2)`: 先做非空判断，比较两个对象

3. `isNull(对象)`: 判断对象是否为空



4. `nonNull(对象)`: 判断对象是否不是空

1. BigInteger表示一个大整数。

2. 如何获取BigInteger的对象？

```
BigInteger b1 = BigInteger.valueOf(0.1);  
BigInteger b1 = new BigInteger("整数");
```

3. 常见操作

| | |
|--------------------------|------------------------------|
| 加: add | 减: subtract |
| 乘: multiply | 除: divide、divideAndRemainder |
| 比较: equals、max、min | |
| 次幂: pow | |
| 转成整数: intValue、longValue | |

1. BigDecimal的作用是什么？

- 表示较大的小数和解决小数运算精度失真问题。

2. BigDecimal的对象如何获取？

- BigDecimal bd1 = new BigDecimal("较大的小数");
- BigDecimal bd2 = BigDecimal.valueOf(0.1);

3. 常见操作

| | |
|--|--|
| 加: add | |
| 减: subtract | |
| 乘: multiply | |
| 除: divide (四舍五入: RoundingMode.HALF_UP) | |

字符类(只匹配一个字符)

| | |
|------------------|--------------------------|
| [abc] | 只能是a, b, 或c |
| [^abc] | 除了a, b, c之外的任何字符 |
| [a-zA-Z] | a到z A到Z, 包括 (范围) |
| [a-d[m-p]] | a到d, 或m到p |
| [a-z&&[def]] | d, e, 或f(交集) |
| [a-zA-Z&&[^bc]] | a到z, 除了b和c (等同于[ad-z]) |
| [a-zA-Z&&[^m-p]] | a到z, 除了m到p (等同于[a-lq-z]) |

预定义字符(只匹配一个字符)

| | |
|----|------------------------|
| . | 任何字符 |
| \d | 一个数字: [0-9] |
| \D | 非数字: [^0-9] |
| \s | 一个空白字符: [\t\n\x0B\f\r] |
| \S | 非空白字符: [^\s] |
| \w | [a-zA-Z_0-9] 英文、数字、下划线 |
| \W | [^\w] 一个非单词字符 |

数量词

| | |
|--------|--------------|
| X? | X, 一次或0次 |
| X* | X, 零次或多次 |
| X+ | X, 一次或多次 |
| X{n} | X, 正好n次 |
| X{n,} | X, 至少n次 |
| X{n,m} | X, 至少n但不超过m次 |

1. 正则表达式中分组有两种：

捕获分组、非捕获分组

2. 捕获分组（默认）：

可以获取每组中的内容反复使用。

3. 组号的特点：

从1开始，连续不间断

以左括号为基准，最左边的是第一组

4. 非捕获分组：

分组之后不需要再用本组数据，仅仅把数据括起来，不占组号。

(?:) (?=) (?!)

都是非捕获分组

捕获分组和非捕获分组

捕获分组：

后续还要继续使用本组的数据。

正则内部使用：\组号

正则外部使用：\$组号

非捕获分组：

分组之后不再用本组数据，仅仅是把数据括起来。

| 符号 | 含义 | 举例 |
|----------|---------------|------------------|
| (? : 正则) | 获取所有 | Java(?:8 11 17) |
| (?= 正则) | 获取前面部分 | Java(?=8 11 17) |
| (?! 正则) | 获取不是指定内容的前面部分 | Java(?:!8 11 17) |



1、如何创建日期对象？

- **Date date = new Date();**
- **Date date = new Date(指定毫秒值);**

2、如何修改时间对象中的毫秒值

- **setTime(毫秒值);**

3、如何获取时间对象中的毫秒值

- **getTime();**

1. SimpleDateFormat的两个作用

格式化

解析

2. 如何指定格式

yyyy年MM月dd日 HH: mm: ss



1. Calendar表示什么？

表示一个时间的日历对象

2. 如何获取对象

通过getInstance方法获取对象

3. 常见方法：

setXxx：修改

getXxx：获取

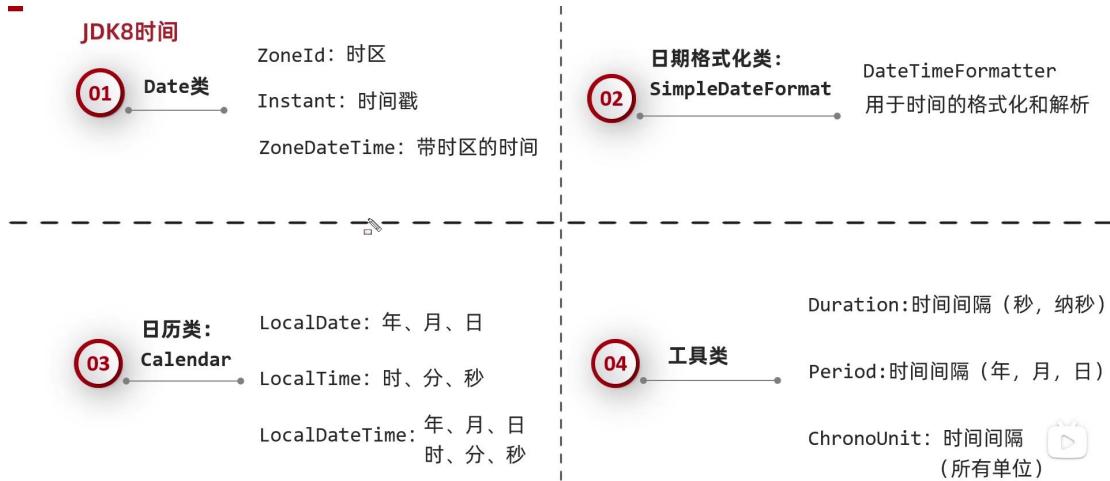


add：在原有的基础上进行增加或者减少

4. 细节点：

日历类中月份的范围：0~11

日历类中星期的特点：星期日是一周中的第一天



LocalDate、LocalTime、LocalDateTime

| 方法名 | 说明 |
|--------------------|----------------------|
| static XXX now() | 获取当前时间的对象 |
| static XXX of(...) | 获取指定时间的对象 |
| get开头的方法 | 获取日历中的年、月、日、时、分、秒等信息 |
| isBefore, isAfter | 比较两个 LocalDate |
| with开头的 | 修改时间系列的方法 |
| minus开头的 | 减少时间系列的方法 |
| plus开头的 | 增加时间系列的方法 |

把对象进行拆箱，变成基本数据类型

把得到的结果再次进行装箱（变会包装类）

1. 什么是包装类?

基本数据类型对应的对象

2. JDK5以后对包装类新增了什么特性?

自动装箱、自动拆箱

3. 我们以后如何获取包装类对象?

不需要new, 不需要调用方法, 直接赋值即可

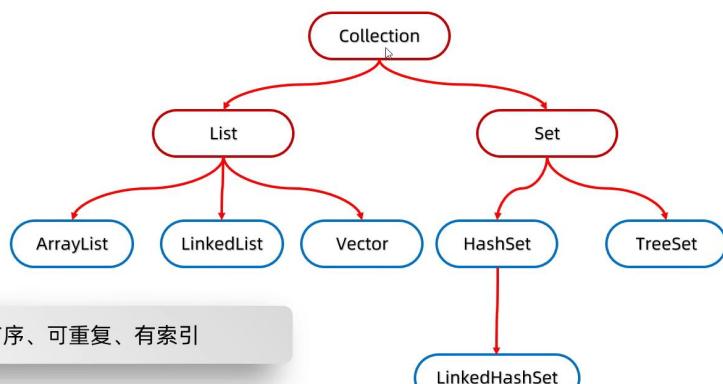
```
Integer i = 10;
```

```
Integer i1 = 10;  
Integer i2 = 10;  
Integer i3 = i1 + i2;
```

集合

collection-list

集合体系结构



```
//1.添加元素
```

```
//细节1: 如果我们要往List系列集合中添加数据, 那么方法永远返回true, 因为List系列的是允许元素重复的。  
//细节2: 如果我们要往Set系列集合中添加数据, 如果当前要添加元素不存在, 方法返回true, 表示添加成功。  
//如果当前要添加的元素已经存在, 方法返回false, 表示添加失败。  
//因为Set系列的集合不允许重复。
```

迭代器遍历

迭代器在Java中的类是**Iterator**, 迭代器是集合专用的遍历方式。

Collection集合获取迭代器

| 方法名称 | 说明 |
|---|-----------------------|
| <code>Iterator<E> iterator()</code> | 返回迭代器对象, 默认指向当前集合的0索引 |

Iterator中的常用方法

| 方法名称 | 说明 |
|--------------------------------|-------------------------------------|
| <code>boolean hasNext()</code> | 判断当前位置是否有元素, 有元素返回true, 没有元素返回false |
| <code>E next()</code> | 获取当前位置的元素, 并将迭代器对象移向下一个位置。 |

1. 迭代器在遍历集合的时候是不依赖索引的

2. 迭代器需要掌握三个方法:

```
Iterator<String> it = list.iterator();
while(it.hasNext()){
    String str = it.next();
    System.out.println(str);
}
```

3. 迭代器的四个细节:

- 如果当前位置没有元素, 还要强行获取, 会报NoSuchElementException
- 迭代器遍历完毕, 指针不会复位
- 循环中只能用一次next方法
- 迭代器遍历时, 不能用集合的方法进行增加或者删除

修改增强 for 中的变量, 不会修改原本的数据

1. Collection是单列集合的顶层接口, 所有方法被List和Set系列集合共享

2. 常见成员方法:

`add、clear、remove、contains、isEmpty、size`

3. 三种通用的遍历方式:

- 迭代器: 在遍历的过程中需要删除元素, 请使用迭代器。
- 增强for、Lambda:

仅仅想遍历, 那么使用增强for或Lambda表达式。

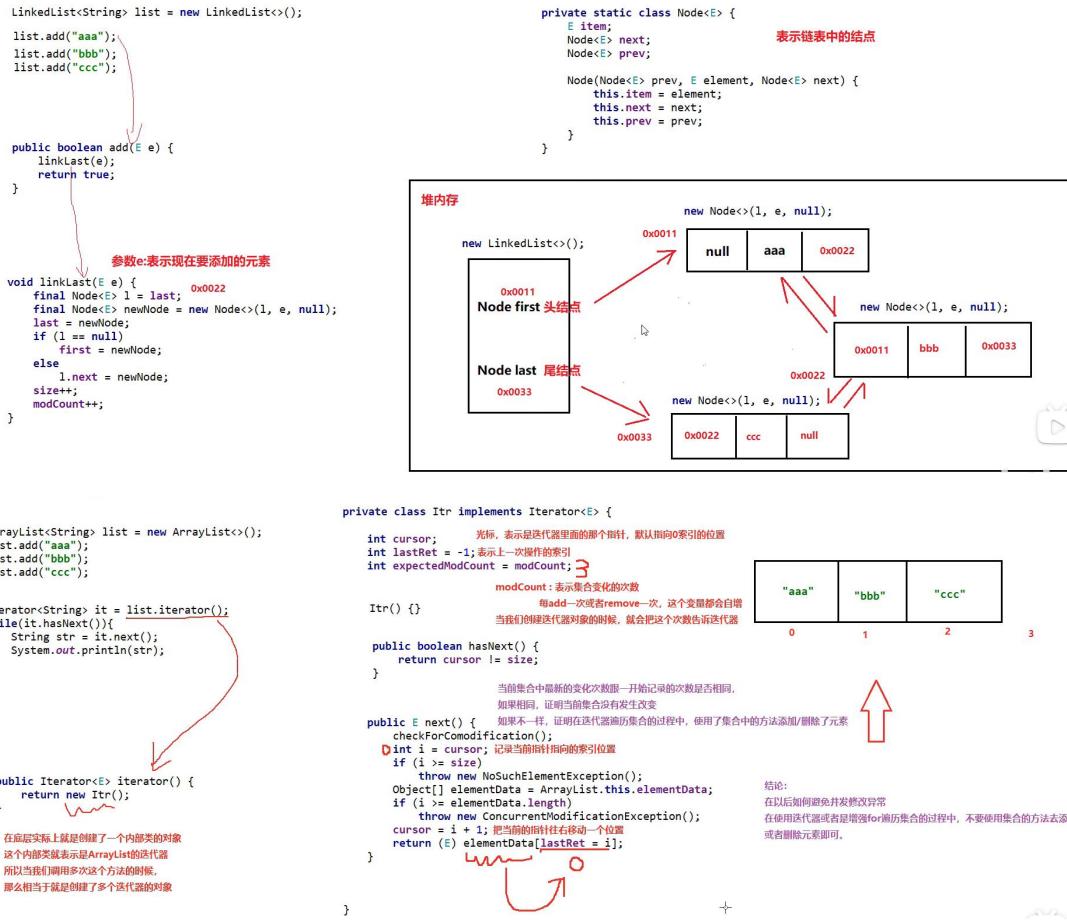
ArrayList集合底层原理

- ① 利用空参创建的集合，在底层创建一个默认长度为0的数组
- ② 添加第一个元素时，底层会创建一个新的长度为10的数组
- ③ 存满时，会扩容1.5倍
- ④ 如果一次添加多个元素，1.5倍还放不下，则新创建数组的长度以实际为准

底层原理

The screenshot shows the Java code for the `ArrayList` class. It highlights the `add()` method and the `grow()` helper method. Red annotations explain the parameters and logic:

- ArrayList Constructor:** `ArrayList<String> list = new ArrayList<>();` 默认初始长度: 0
- 添加第一个元素:** `list.add("aaa");`
- add(E e) Method:** `public boolean add(E e) { modCount++; add(e, elementData, size); return true; }`
 - 参数一: 当前要添加的元素
 - 参数二: 集合底层的数组名字
 - 参数三: 集合的长度/当前元素应存入的位置
- grow() Helper Method:** `private Object[] grow(int minCapacity) { int oldCapacity = elementData.length; // 记录原来的容量 if (oldCapacity > 0 || elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) { int newCapacity = ArraysSupport.newLength(oldCapacity, minCapacity - oldCapacity, oldCapacity >> 1); return elementData = Arrays.copyOf(elementData, newCapacity); } else { return elementData = new Object[Math.max(DEFAULT_CAPACITY, minCapacity)]; } }`
 - 第一次添加数据的时候，会执行到else这里
- newLength() Method:** `public static int newLength(int oldLength, int minGrowth, int prefGrowth) { int prefLength = oldLength + Math.max(minGrowth, prefGrowth); if (0 < prefLength && prefLength <= SOFT_MAX_ARRAY_LENGTH) { return prefLength; } else { return hugeLength(oldLength, minGrowth); } }`
- add(E e, Object[] elementData, int s) Method:** `private void add(E e, Object[] elementData, int s) { if (s == elementData.length) elementData = grow(); // grow()表示数组扩容 elementData[s] = e; size = s + 1; }`
- grow() Helper Method:** `private Object[] grow() { return grow(size + 1); } // 把现有的个数+1`
- newLength() Method (Bottom):** `public static int newLength(int oldLength, int minGrowth, int prefGrowth) { int prefLength = oldLength + Math.max(minGrowth, prefGrowth); if (0 < prefLength && prefLength <= SOFT_MAX_ARRAY_LENGTH) { return prefLength; } else { return hugeLength(oldLength, minGrowth); } }`
 - 第一种情况：如果一次添加一个元素，那么第二个参数一定是1，表示此时数组只要扩容1个单位就可以了。
 - 第二种情况：如果一次添加多个元素，假设100，那么第二个参数是100，表示此时数组需要扩容100个单位才可以



泛型

1. 什么是泛型？

JDK5引入的特性，可以在编译阶段约束操作的数据类型，并进行检查

2. 泛型的好处？

- 统一数据类型
- 把运行时期的问题提前到了编译期间，避免了强制类型转换可能出现的异常，因为在编译阶段类型就能确定下来。

3. 泛型的细节？

- 泛型中不能写基本数据类型
- 指定泛型的具体类型后，传递数据时，可以传入该类型和他的子类类型
- 如果不写泛型，类型默认是Object

4. 哪里定义泛型？

- 泛型类：在类名后面定义泛型，创建该类对象的时候，确定类型
- 泛型方法：在修饰符后面定义方法，调用该方法的时候，确定类型
- 泛型接口：在接口名后面定义泛型，实现类确定类型，实现类延续泛型

5. 泛型的继承和通配符

- 泛型不具备继承性，但是数据具备继承性
- 泛型的通配符：?
 - ? extend E
 - ? super E

6. 使用场景

- 定义类、方法、接口的时候，如果类型不确定，就可以定义泛型
- 如果类型不确定，但是能知道是哪个继承体系中的，可以使用泛型的通配符

二叉树

① 左左

一次右旋

② 左右

先局部左旋，再整体右旋

③ 右右

一次左旋

④ 右左

先局部右旋，再整体左旋

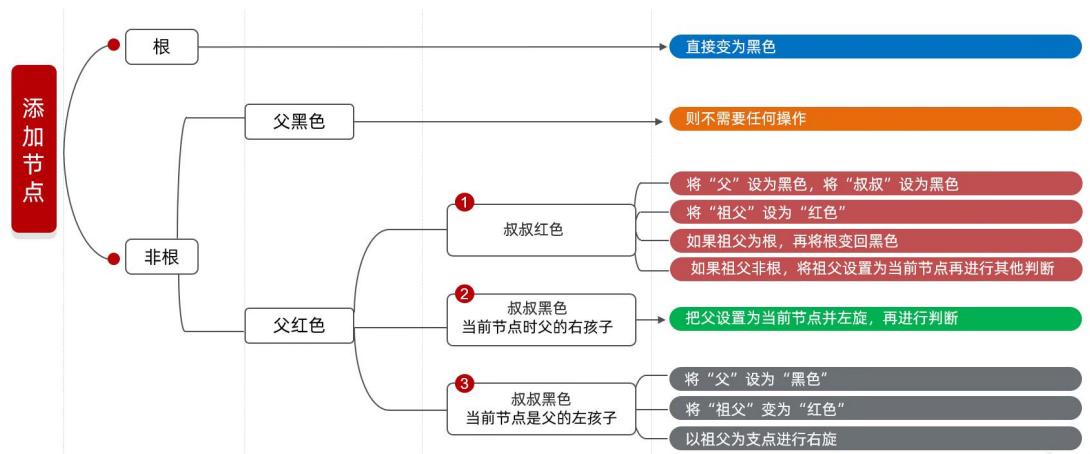
规则如下：

- ① 每一个节点或是红色的，或者是黑色的。
- ② 根节点必须是黑色
- ③ 如果一个节点没有子节点或者父节点，则该节点相应的指针属性值为Nil，这些Nil视为叶节点，每个叶节点(Nil)是黑色的；
- ④ 不能出现两个红色节点相连的情况
- ⑤ 对每一个节点，从该节点到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点；

红黑树增删改查的性能都很好

数据结构(红黑树)添加节点规则

——红黑树在添加节点的时候，添加的节点默认是红色的



collection-Set

1. Set系列集合的特点

- 无序、不重复、无索引
- Set集合的方法上基本上与Collection的API一致

2. Set集合的实现类特点

- HashSet : 无序、不重复、无索引
- LinkedHashSet: 有序、不重复、无索引
- TreeSet: 可排序、不重复、无索引

哈希值

- 根据hashCode方法算出来的int类型的整数
- 该方法定义在Object类中，所有对象都可以调用，默认使用地址值进行计算
- 一般情况下，会重写hashCode方法，利用对象内部的属性值计算哈希值

对象的哈希值特点

- 如果没有重写hashCode方法，不同对象计算出的哈希值是不同的
- 如果已经重写hashCode方法，不同的对象只要属性值相同，计算出的哈希值就是一样的
- 在小部分情况下，不同的属性值或者不同的地址值计算出来的哈希值也有可能一样。（哈希碰撞）

HashSet JDK8以后底层原理



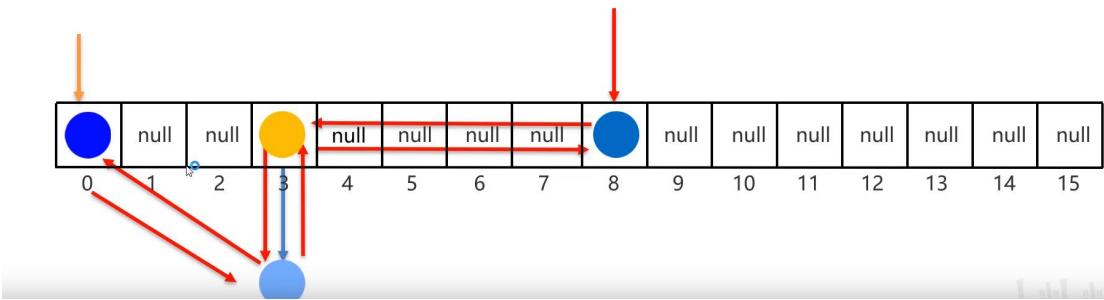
扩容倍数为 2 倍

JDK8以后，当链表长度**超过8**，而且数组长度**大于等于64**时，自动转换为红黑树

如果集合中存储的是自定义对象，必须要重写**hashCode**和**equals**方法

LinkedHashSet底层原理

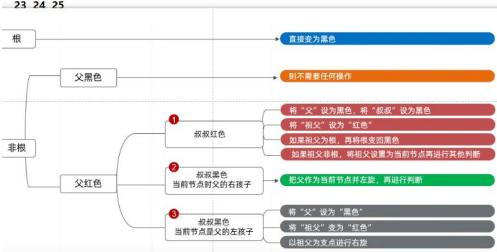
- **有序**、不重复、无索引。
- 这里的有序指的是保证存储和取出的元素顺序一致
- **原理**：底层数据结构是依然哈希表，只是每个元素又额外的多了一个双链表的机制记录存储的顺序。



```
//1. 创建三个学生对象
Student s1 = new Student("zhangsan",23);
Student s2 = new Student("lisi",24);
Student s3 = new Student("wangwu",25);
s4 = new Student("zhaoiu",26);

//2. 创建集合对象
TreeSet<Student> ts = new TreeSet<>();

//3. 添加元素
ts.add(s3);
ts.add(s2);
ts.add(s1);
ts.add(s4);
//4. 打印集合
System.out.println(ts);
```



排序规则：
@Override
public int compareTo(Student o) {
 //指定排序的规则
 //只看年龄，我想要按照年龄的升序进行排序
 return this.getAge() - o.getAge();
}

this: 表示当前要添加的元素
o: 表示已经在红黑树存在的元素

返回值：
负数：认为要添加的元素是小的，存左边
正数：认为要添加的元素是大的，存右边
0： 认为要添加的元素已经存在，舍弃

叶子节点
Nil

1. TreeSet集合的特点是怎么样的？

- 可排序、不重复、无索引
- 底层基于红黑树实现排序，增删改查性能较好

2. TreeSet集合自定义排序规则有几种方式

- 方式一：JavaBean类实现Comparable接口，指定比较规则
- 方式二：创建集合时，自定义Comparator比较器对象，指定比较规则

3. 方法返回值的特点

- 负数：表示当前要添加的元素是小的，存左边
- 正数：表示当前要添加的元素是大的，存右边
- 0：表示当前要添加的元素已经存在，舍弃

1. 如果想要集合中的元素可重复
 - 用ArrayList集合，基于数组的。 (用的最多)
2. 如果想要集合中的元素可重复，而且当前的增删操作明显多于查询
 - 用LinkedList集合，基于链表的。
3. 如果想对集合中的元素去重
 - 用HashSet集合，基于哈希表的。 (用的最多)
4. 如果想对集合中的元素去重，而且保证存取顺序
 - 用LinkedHashSet集合，基于哈希表和双链表，效率低于HashSet。
5. 如果想对集合中的元素进行排序
 - 用TreeSet集合，基于红黑树。后续也可以用List集合实现排序。

Map

TreeMap

- TreeMap跟TreeSet底层原理一样，都是红黑树结构的。
- 由键决定特性：不重复、无索引、可排序
- 可排序：对键进行排序。
- 注意：默认按照键的从小到大进行排序，也可以自己规定键的排序规则

代码书写两种排序规则

- 实现Comparable接口，指定比较规则。
- 创建集合时传递Comparator比较器对象，指定比较规则。

YamlMyHashMap.java

```

63 //参数一：键的哈希值
64 //参数二：键
65 //参数三：值
66 //参数四：如果键重复了是否保留
67 //      true，表示老元素的值将保留，不会覆盖
68 //      false，表示老元素的值不保留，会进行覆盖
69 final V putVal(Im<hash, K key, V value, boolean onlyIfAbsent,boolean evict) {
70     Node<K,V>[] tab;
71     Node<K,V> p;
72     //临时的第三方变量，用来记录键值对对象的地址值
73     int n;
74     //表示当前数组的长度
75     int l;
76     //表示索引
77     int i;
78     //把哈希表中数组的地址值，赋值给局部变量tab
79     tab = table;
80
81     if (tab == null || (n = tab.length) == 0){
82         //1.如果当前是第一次添加数据，底层会创建一个默认长度为16，加载因子为0.75的数组
83         //2.如果不是第一次添加数据，会看数组中的元素是否达到了扩容的条件
84         //如果没有达到扩容条件，底层不会做任何操作
85         //如果达到了扩容条件，底层会把数组扩容为原先的两倍，并把数据全部转移到新的哈希表中
86         tab = resize();
87         //表示数组的长度赋值给n
88         n = tab.length;
89     }
90
91     //拿着数组的长度跟键的哈希值进行计算，计算出当前键值对对象，在数组中应存入的位置
92     i = (n - 1) & hash://Index
93     //获取数组中对应元素的数据
94     p = tab[i];
95
96     if (p == null){
97         //底层会创建一个键值对对象，直接放到数组当中
98
99
100
101
102
}

```

MyHashMap.java

```

//等号的左边：数组中键值对的哈希值
//等号的右边：当前要添加键值对的哈希值
//如果键不一样，则返回false
boolean bl = p.hash == hash;

if (bl && ((k == p.key) == key || (key != null && key.equals(k)))) {
    e = p;
} else if (p instanceof TreeNode) {
    //判断数组中获取出来的键值对是不是红黑树中的节点
    //如果是，则调用方法putTreeVal，把当前的节点按照红黑树的规则添加到树当中。
    ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
} else {
    //如果从数组中获取出来的键值对不是红黑树中的节点
    //表示此时下面挂的是链表
    for (int binCount = 0; ; ++binCount) {
        if ((e = p.next) == null) {
            //此时就会创建一个新的节点，挂在下面形成链表
            p.next = newNode(hash, key, value, null);
            //判断当前键值对的长度是否超过8，如果超过，就会调用方法treeifyBin
            //判断数组的长度是否大于等于64
            //如果同时满足这两个条件，就会把这个链表转成红黑树
            if (binCount >= TREEIFY_THRESHOLD - 1)
                treeifyBin(tab, hash);
            break;
        }
        if (e.hash == hash && ((k == e.key) == key || (key != null && key.equals(k))))
            break;
        p = e;
    }
}

if (e != null) {
    V oldValue = e.value;
    if (!onlyIfAbsent || oldValue == null) {
        e.value = value;
    }
    afterNodeAccess(e);
    return oldValue;
}

//如果e为null，表示当前不需要覆盖任何元素
//如果e不为null，表示当前的键是一样的，值会被覆盖
//e:0x0044 ddd 555
//要添加的元素： 0x0055 ddd 555
if (e.hash == hash && ((k == e.key) == key || (key != null && key.equals(k)))) {
    break;
}

p = e;
}

if (e != null) {
    V oldValue = e.value;
    if (!onlyIfAbsent || oldValue == null) {
        //等号的右边：当前要添加的值
        //等号的左边：0x0044的值
        e.value = value;
    }
    afterNodeAccess(e);
    return oldValue;
}

//threshold: 记录的就是数组的长度 * 0.75，哈希表的扩容时机 16 * 0.75 = 12
if (++size > threshold){
    resize();
}

//表示当前没有覆盖任何元素，返回null
return null;
}

```

```
41 5. 添加元素
42  public V put(K key, V value) {
43     return put(key, value, true);
44 }
45
46 参数一：键
47 参数二：值
48 参数三：当键重复的时候，是否需要覆盖值
49     true：覆盖
50     false：不覆盖
51
52 private V put(K key, V value, boolean replaceOld) {
53     //获取根节点的地址值，赋值给局部变量
54     Entry<K,V> t = root;
55     //判断根节点是否为null
56     //如果为null，表示当前是第一次添加，会把当前要添加的元素，当做根节点
57     //如果不为null，表示当前不是第一次添加，跳过这个判断继续执行下面的代码
58     if (t == null) {
59         //方法的底层，会创建一个Entry对象，把他当做根节点
60         addEntryToEmptyMap(key, value);
61         //表示此时没有覆盖任何的元素
62         return null;
63     }
64     //表示两个元素的键比较之后的结果
65     int cmp;
66     //表示当前要添加节点的父节点
67     Entry<K,V> parent;
68
69     //表示当前的比较规则
70     //如果是我们是采取默认的自然排序，那么此时comparator记录的是null，cpr记录的也是null
71     //如果我们是采取比较云排序方式，那么此时comparator记录的是就是比较器
72     Comparator<? super K> cpr = comparator;
73     //表示判断当前是否有比较器对象
74     //如果传递了比较器对象，就执行if里面的代码，此时以比较器的规则为准
75     //如果没有传递比较器对象，就执行else里面的代码，此时以自然排序的规则为准
76     if (cpr != null) {
77         do {
78             parent = t;
79             cmp = cpr.compare(key, t.key);
80             if (cmp < 0)
81                 t = t.left;
82             else if (cmp > 0)
83                 t = t.right;
84             else {
85                 V oldValue = t.value;
86                 if (replaceOld || oldValue == null) {
87                     t.value = value;
88                 }
89                 return oldValue;
90             }
91         } while (t != null);
92     } else {
93         //把键进行强转，强转成Comparable类型的
94         //要求：键必须要实现Comparable接口，如果没有实现这个接口
95         //此时在强转的时候，就会报错。
```

```
92 } else {
93     //把键进行强转，强转成Comparable类型的
94     //要求：键必须要实现Comparable接口，如果没有实现这个接口
95     //此时在强转的时候，就会报错。
96     Comparable<? super K> k = (Comparable<? super K>) key;
97     do {
98         //把根节点当做当前节点的父节点
99         parent = t;
100        //调用compareTo方法，比较根节点和当前要添加节点的大小关系
101        cmp = k.compareTo(t.key);
102
103        if (cmp < 0)
104            //如果比较的结果为负数
105            //那么继续到根节点的左边去找
106            t = t.left;
107        else if (cmp > 0)
108            //如果比较的结果为正数
109            //那么继续到根节点的右边去找
110            t = t.right;
111        else {
112            //如果比较的结果为0，会覆盖
113            V oldValue = t.value;
114            if (replaceOld || oldValue == null) {
115                t.value = value;
116            }
117            return oldValue;
118        }
119    } while (t != null);
120
121    //就会把当前节点按照指定的规则进行添加
122    addEntry(key, value, parent, cmp < 0);
123    return null;
124}
125
126
127
128    private void addEntry(K key, V value, Entry<K, V> parent, boolean addToLeft) {
129        Entry<K,V> e = new Entry<>(key, value, parent);
130        if (addToLeft)
131            parent.left = e;
132        else
133            parent.right = e;
134        //添加完毕之后，需要按照红黑树的规则进行调整
135        fixAfterInsertion(e);
136        size++;
137        modCount++;
138    }
139
140
141
142    private void fixAfterInsertion(Entry<K,V> x) {
143        //因为红黑树的节点默认就是红色的
144        x.color = RED;
145
146        //按照红黑规则进行调整
```

```

146 //按照红黑规则进行调整
147
148 //parentOf:获取x的父节点
149 //parentOf(parentOf(x)):获取x的爷爷节点
150 //leftOf:获取左子节点
151 while (x != null && x != root && x.parent.color == RED) {
152
153     //判断当前节点的父节点是爷爷节点的左子节点还是右子节点
154     //目的:为了获取当前节点的叔叔节点
155     if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {
156         //表示当前节点的父节点是爷爷节点的左子节点
157         //那么下面就可以用rightOf获取到当前节点的叔叔节点
158         Entry<K,V> y = rightOf(parentOf(parentOf(x)));
159         if (colorOf(y) == RED) {
160             //叔叔节点为红色的处理方案
161
162             //把父节点设置为黑色
163             setColor(parentOf(x), BLACK);
164             //把叔叔节点设置为黑色
165             setColor(y, BLACK);
166             //把爷爷节点设置为红色
167             setColor(parentOf(parentOf(x)), RED);
168
169             //把爷爷节点设置为当前节点
170             x = parentOf(parentOf(x));
171         } else {
172             //叔叔节点为黑色的处理方案
173
174             //表示判断当前节点是否为父节点的右子节点
175             if (x == rightOf(parentOf(x))) {
176
177                 //表示当前节点是父节点的右子节点
178                 x = parentOf(x);
179                 //左旋
180                 rotateLeft(x);
181
182             }
183             setColor(parentOf(x), BLACK);
184             setColor(parentOf(parentOf(x)), RED);
185             rotateRight(parentOf(parentOf(x)));
186
187         }
188     } else {
189         //表示当前节点的父节点是爷爷节点的右子节点
190         //那么下面就可以用leftOf获取到当前节点的叔叔节点
191         Entry<K,V> y = leftOf(parentOf(parentOf(x)));
192         if (colorOf(y) == RED) {
193             //把父节点设置为黑色
194             setColor(parentOf(x), BLACK);
195             //把叔叔节点设置为黑色
196             setColor(y, BLACK);
197             //把爷爷节点设置为红色
198             setColor(parentOf(parentOf(x)), RED);
199             x = parentOf(parentOf(x));
200         } else {
201             if (x == leftOf(parentOf(x))) {
202                 x = parentOf(x);
203             }
204         }
205     }
206 }

```

6.课堂思考问题：

6.1TreeMap 添加元素的时候，键是否需要重写 hashCode 和 equals 方法？

此时是不需要重写的。

6.2HashMap 是哈希表结构的，JDK8 开始由数组，链表，红黑树组成的。

既然有红黑树，HashMap 的键是否需要实现 Comparable 接口或者传递比较器对象呢？

不需要的。

因为在 HashMap 的底层，默认是利用哈希值的大小关系来创建红黑树的

6.3TreeMap 和 HashMap 谁的效率更高？

如果是最坏情况，添加了 8 个元素，这 8 个元素形成了链表，此时 TreeMap 的效率要更高

但是这种情况出现的几率非常的少。

一般而言，还是 `HashMap` 的效率要更高。

6.4 你觉得在 `Map` 集合中，`java` 会提供一个如果键重复了，不会覆盖的 `put` 方法呢？

此时 `putIfAbsent` 本身不重要。

传递一个思想：

代码中的逻辑都有两面性，如果我们只知道了其中的 A 面，而且代码中还发现了有变量可以控制两面性的发生。

那么该逻辑一定会有 B 面。

习惯：

`boolean` 类型的变量控制，一般只有 AB 两面，因为 `boolean` 只有两个值

`int` 类型的变量控制，一般至少有三面，因为 `int` 可以取多个值。

6.5 三种双列集合，以后如何选择？

`HashMap` `LinkedHashMap` `TreeMap`

默认：`HashMap`（效率最高）

如果要保证存取有序：`LinkedHashMap`

如果要进行排序：`TreeMap`

1. 可变参数本质上就是一个数组

2. 作用：在形参中接收多个数据

3. 格式：数据类型...参数名称

举例：int...a

4. 注意事项：

- 形参列表中可变参数只能有一个
- 可变参数必须放在形参列表的最后面



可变参数

Collections常用的API

| 方法名称 | 说明 |
|--|------------------|
| public static <T> boolean addAll (Collection<T> c, T... elements) | 批量添加元素 |
| public static void shuffle (List<?> list) | 打乱List集合元素的顺序 |
| public static <T> void sort (List<T> list) | 排序 |
| public static <T> void sort (List<T> list, Comparator<T> c) | 根据指定的规则进行排序 |
| public static <T> int binarySearch (List<T> list, T key) | 以二分查找法查找元素 |
| public static <T> void copy (List<T> dest, List<T> src) | 拷贝集合中的元素 |
| public static <T> int fill (List<T> list, T obj) | 使用指定的元素填充集合 |
| public static <T> void max/min (Collection<T> coll) | 根据默认的自然排序获取最大/小值 |
| public static <T> void swap (List<?> list, int i, int j) | 交换集合中指定位置的元素 |

1. 不可变集合的特点?

- 定义完成后不可以修改，或者添加、删除

2. 如何创建不可变集合?

- List、Set、Map接口中，都存在of方法可以创建不可变集合

3. 三种方式的细节

- List: 直接用
- Set: 元素不能重复
- Map: 元素不能重复、键值对数量最多是10个。

超过10个用ofEntries方法

stream

Stream流的中间方法

| 名称 | 说明 |
|--|----------------------------|
| Stream<T> filter (Predicate<? super T> predicate) | 过滤 |
| Stream<T> limit (long maxSize) | 获取前几个元素 |
| Stream<T> skip (long n) | 跳过前几个元素 |
| Stream<T> distinct() | 元素去重，依赖(hashCode和equals方法) |
| static <T> Stream<T> concat (Stream a, Stream b) | 合并a和b两个流为一个流 |
| Stream<R> map (Function<T , R> mapper) | 转换流中的数据类型 |

注意1： 中间方法，返回新的Stream流，原来的Stream流只能使用一次，建议使用链式编程

注意2： 修改Stream流中的数据，不会影响原来集合或者数组中的数据

Stream流的终结方法

| 名称 | 说明 |
|---------------------------------------|---------------|
| void forEach (Consumer action) | 遍历 |
| long count() | 统计 |
| toArray() | 收集流中的数据，放到数组中 |
| collect(Collector collector) | 收集流中的数据，放到集合中 |

1. Stream流的作用

结合了Lambda表达式，简化集合、数组的操作

2. Stream的使用步骤

- 获取Stream流对象
- 使用中间方法处理数据
- 使用终结方法处理数据

3. 如何获取Stream流对象

- 单列集合：Collection中的默认方法stream
- 双列集合：不能直接获取
- 数组：Arrays工具类型中的静态方法stream
- 一堆零散的数据：Stream接口中的静态方法of

4. 常见方法

中间方法： filter, limit, skip, distinct, concat, map

终结方法： forEach, count, collect

方法引用

1. 什么是方法引用?

把已经存在的方法拿过来用，当做函数式接口中抽象方法的方法体

2. :: 是什么符号?

方法引用符

3. 方法引用时要注意什么?

- 需要有函数式接口
- 被引用方法必须已经存在
- 被引用方法的形参和返回值需要跟抽象方法保持一致
- 被引用方法的功能要满足当前的需求

方法引用（类名引用成员方法）

格式

类名::成员方法

需求：

集合里面一些字符串，要求变成大写后进行输出

方法引用的规则：

1. 需要有函数式接口
2. 被引用的方法必须已经存在
3. 被引用方法的形参，需要跟抽象方法的第二个形参到最后一个形参保持一致，返回值需要保持一致。
4. 被引用方法的功能需要满足当前的需求

抽象方法形参的详解：

第一个参数：表示被引用方法的调用者，决定了可以引用哪些类中的方法

在 Stream 流当中，第一个参数一般都表示流里面的每一个数据。

假设流里面的数据是字符串，那么使用这种方式进行方法引用，只能引用 String 这个类中的方法

第二个参数到最后一个参数：跟被引用方法的形参保持一致，如果没有第二个参数，说明被引用的方法需要是无参的成员方法

*/

```
//1. 创建集合对象
ArrayList<String> list = new ArrayList<>();
//2. 添加数据
Collections.addAll(list, ...elements: "aaa", "bbb", "ccc", "ddd");
//3. 变成大写后进行输出
//map(String::toUpperCase)
//拿着流里面的每一个数据，去调用String类中的toUpperCase方法，方法的返回值就是转换之后的结果。
list.stream().map(String::toUpperCase).forEach(s -> System.out.println(s));
```

1. 引用静态方法

类名: : 静态方法

2. 引用成员方法

对象: : 成员方法

this: : 成员方法

super: : 成员方法

3. 引用构造方法

类名: : new

4. 使用类名引用成员方法

类名: : 成员方法

不能引用所有类中的成员方法
如果抽象方法的第一个参数是A类型的
只能引用A类中的方法

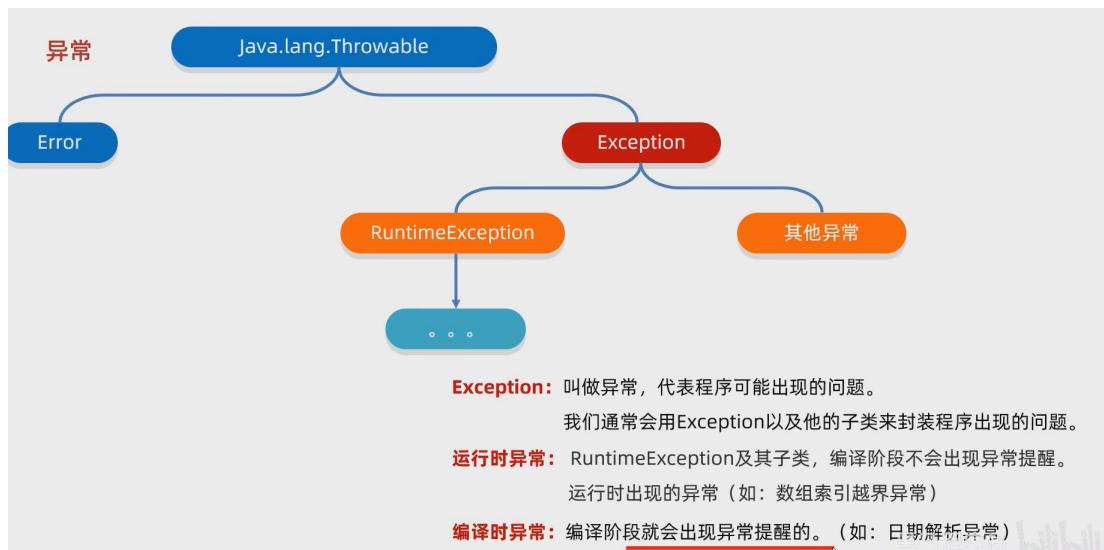
5. 引用数组的构造方法

数据类型[]::new

异常

Error: 代表的系统级别错误（属于严重问题）系统一旦出现问题，sun 公司就会把这些错误封装成 Error 对象

Error 是 sun 公司自己用的，不是给程序员用的。



```

int[] arr = {1, 2, 3, 4, 5, 6};
try{
    //可能出现异常的代码;
    System.out.println(arr[10]); //此处出现了异常，程序就会在这里创建一个ArrayIndexOutOfBoundsException对象
                                //new ArrayIndexOutOfBoundsException();
    //拿着这个对象到catch的小括号中对比，看括号中的变量是否可以接收这个对象
    //如果能被接收，就表示该异常就被捕获（抓住），执行catch里面对应的代码
    //当catch里面所有的代码执行完毕，继续执行try...catch体系下面的其他代码
}catch(ArrayIndexOutOfBoundsException e){
    //如果出现了ArrayIndexOutOfBoundsException异常，我该如何处理
    System.out.println("索引越界了");
}

System.out.println("看看我执行了吗？");

```

灵魂一问：如果try中没有遇到问题，怎么执行？

答：

会把try里面所有的代码全部执行完毕。
不会执行catch里面的代码

灵魂二问：如果try中可能会遇到多个问题，怎么执行？

答：

会写多个catch与之对应，
父类异常需要写在下面

灵魂三问：如果try中遇到的问题没有被捕获，怎么执行？

答：

相当于try...catch白写了，
当前异常会交给虚拟机处理

灵魂四问：如果try中遇到了问题，那么try下面的其他代码还会执行吗？

答：

不会执行了。try中遇到问题，直接跳转到对应的catch
如果没有对应的catch与之匹配，则交给虚拟机处理

public String getMessage()
public String toString()

返回此 throwable 的详细消息字符串
返回此可抛出的简短描述

public void printStackTrace()

在底层是利用System.err.println进行输出
把异常的错误信息以红色字体输出在控制台
细节：仅仅是打印信息，不会停止程序运行

抛出处理

throws

注意：写在方法定义处，表示声明一个异常
告诉调用者，使用本方法可能会有哪些异常

throw

注意：写在方法内，结束方法
手动抛出异常对象，交给调用者
方法中下面的代码不再执行了

```
public void 方法() throws 异常类名1, 异常类名2...{  
    ...  
}
```

```
public void 方法(){  
    throw new NullPointerException();  
}
```

- 编译时异常：必须要写。
- 运行时异常：可以不写。

1. 虚拟机默认处理异常的方式

把异常信息以红色字体打印在控制台，并结束程序

2. 捕获：try...catch

一般用在调用处，能让代码继续往下运行。

3. 抛出：throw throws

在方法中，出现异常了。

方法就没有继续运行下去的意义了，采取抛出处理。

让该方法结束运行并告诉调用者出现了问题。

自定义异常

① 定义异常类

② 写继承关系

③ 空参构造

④ 带参构造

意义：就是为了让控制台的报错信息更加的见名之意

file

File的常见成员方法（获取并遍历）

| 方法名称 | 说明 |
|---------------------------|--------------|
| public File[] listFiles() | 获取当前该路径下所有内容 |



- 当调用者File表示的路径不存在时，返回null
- 当调用者File表示的路径是文件时，返回null
- 当调用者File表示的路径是一个空文件夹时，返回一个长度为0的数组
- 当调用者File表示的路径是一个有内容的文件夹时，将里面所有文件和文件夹的路径放在File数组中返回
- 当调用者File表示的路径是一个有隐藏文件的文件夹时，将里面所有文件和文件夹的路径放在File数组中返回，包含隐藏文件
- 当调用者File表示的路径是需要权限才能访问的文件夹时，返回null

IO 流

1. 什么是IO流?

存储和读取数据的解决方案

I: input O: output

流：像水流一样传输数据

2. IO流的作用?

用于读写数据（本地文件，网络）

3. IO流按照流向可以分类哪两种流?

输出流：程序  文件

输入流：文件  程序

4. IO流按照操作文件的类型可以分类哪两种流?

字节流：可以操作所有类型的文件

字符流：只能操作纯文本文件

5. 什么是纯文本文件?

用windows系统自带的记事本打开并且能读懂的文件

txt文件，md文件，xml文件，lrc文件等

字节流

FileOutputStream书写细节

① 创建字节输出流对象

细节1：参数是字符串表示的路径或者File对象都是可以的

细节2：如果文件不存在会创建一个新的文件，但是要保证父级路径是存在的。

细节3：如果文件已经存在，则会清空文件

② 写数据

细节：write方法的参数是整数，但是实际上写到本地文件中的是整数在ASCII上对应的字符

③ 释放资源

细节：每次使用完流之后都要释放资源

字符集

1. 在计算机中，任意数据都是以二进制的形式来存储的
2. 计算机中最小的存储单元是一个字节
3. ASCII字符集中，一个英文占一个字节
4. 简体中文版Windows，默认使用GBK字符集
5. GBK字符集完全兼容ASCII字符集

一个英文占一个字节，二进制第一位是0

一个中文占两个字节，二进制高位字节的第一位是1

. Unicode字符集的UTF-8编码格式

一个英文占一个字节，二进制第一位是0，转成十进制是正数

一个中文占三个字节，二进制第一位是1，第一个字节转成十进制是负数

字符流

字符流原理解析

① 创建字符输入流对象

底层：关联文件，并创建缓冲区（长度为8192的字节数组）

② 读取数据

底层：1. 判断缓冲区中是否有数据可以读取

2. 缓冲区没有数据：就从文件中获取数据，装到缓冲区中，每次尽可能装满缓冲区

如果文件中也没有数据了，返回-1

3. 缓冲区有数据：就从缓冲区中读取。

空参的read方法：一次读取一个字节，遇到中文一次读多个字节，把字节解码并转成十进制返回

有参的read方法：把读取字节，解码，强转三步合并了，强转之后的字符放到数组中

缓冲流

1. 缓冲流有几种？

- 字节缓冲输入流：**BufferedInputStream**
- 字节缓冲输出流：**BufferedOutputStream**
- 字符缓冲输入流：**BufferedReader**
- 字符缓冲输出流：**BufferedWriter**

2. 缓冲流为什么能提高性能

- 缓冲流自带长度为8192的缓冲区
- 可以显著提高字节流的读写性能
- 对于字符流提升不明显，对于字符缓冲流而言关键点是两个特有的方法

3. 字符缓冲流两个特有的方法是什么？

- 字符缓冲输入流**BufferedReader**: **readLine ()**
- 字符缓冲输出流**BufferedWriter**: **newLine ()**

序列化流

序列化流/反序列化流的细节汇总

① 使用序列化流将对象写到文件时，需要让Javabean类实现Serializable接口。

否则，会出现NotSerializableException异常

② 序列化流写到文件中的数据是不能修改的，一旦修改就无法再次读回来了

③ 序列化对象后，修改了Javabean类，再次反序列化，会不会有问题？

会出问题，会抛出**InvalidClassException**异常

解决方案：给Javabean类添加serialVersionUID (序列号、版本号)

④ 如果一个对象中的某个成员变量的值不想被序列化，又该如何实现呢？

解决方案：给该成员变量加transient关键字修饰，**该关键字标记的成员变量不参与序列化过程**

打印流

1. 打印流有几种？各有什么特点？

- 有字节打印流和字符打印流两种
- 打印流不操作数据源，只能操作目的地
- 字节打印流：默认自动刷新，特有的println自动换行
- 字符打印流：自动刷新需要开启，特有的println自动换行

解压流

```
//定义一个方法用来解压
public static void unzip(File src,File dest) throws IOException {
    //解压的本质：把压缩包里面的每一个文件或者文件夹读取出来，按照层级拷贝到目的地当中
    //创建一个解压缩流用来读取压缩包中的数据
    ZipInputStream zip = new ZipInputStream(new FileInputStream(src));
    //要先获取到压缩包里面的每一个zipentry对象
    //表示当前在压缩包中获取到的文件或者文件夹
    ZipEntry entry;
    while((entry = zip.getNextEntry()) != null){
        System.out.println(entry);
        if(entry.isDirectory()){
            //文件夹：需要在目的地dest处创建一个同样的文件夹
            File file = new File(dest,entry.toString());
            file.mkdirs();
        }else{
            //文件：需要读取到压缩包中的文件，并把他存放到目的地dest文件夹中（按照层级目录进行存放）
            FileOutputStream fos = new FileOutputStream(new File(dest,entry.toString()));
            int b;
            while((b = zip.read()) != -1){
                //写到目的地
                fos.write(b);
            }
            fos.close();
            //表示在压缩包中的一个文件处理完毕了。
            zip.closeEntry();
        }
    }
}
```

压缩流

```
/*
 * 作用: 压缩
 * 参数一: 表示要压缩的文件
 * 参数二: 表示压缩包的位置
 */
public static void toZip(File src, File dest) throws IOException {
    //1.创建压缩流关联压缩包
    ZipOutputStream zos = new ZipOutputStream(new FileOutputStream(new File(dest, child: "a.zip")));
    //2.创建ZipEntry对象, 表示压缩包里面的每一个文件和文件夹
    ZipEntry entry = new ZipEntry( name: "a.txt");
    //3.把ZipEntry对象放到压缩包当中
    zos.putNextEntry(entry);
    //4.把src文件中的数据写到压缩包当中
    FileInputStream fis = new FileInputStream(src);
    int b;
    while((b = fis.read()) != -1){
        zos.write(b);
    }
    zos.closeEntry();
    zos.close();
}
```

```
//1.创建File对象表示要压缩的文件夹
File src = new File( pathname: "D:\\aaa");
//2.创建File对象表示压缩包放在哪里 (压缩包的父级路径)
File destParent = src.getParentFile(); //D:\\
//3.创建File对象表示压缩包的路径
File dest = new File(destParent, child: src.getName() + ".zip");
//4.创建压缩流关联压缩包
ZipOutputStream zos = new ZipOutputStream(new FileOutputStream(dest));
//5.获取src里面的每一个文件, 变成ZipEntry对象, 放入到压缩包当中

//6.释放资源
zos.close();
```

```

/*
 * 作用：获取src里面的每一个文件，变成ZipEntry对象，放入到压缩包当中
 * 参数一：数据源
 * 参数二：压缩流
 * 参数三：压缩包内部的路径
 */
public static void toZip(File src, ZipOutputStream zos, String name) throws IOException {
    //1.进入src文件夹
    File[] files = src.listFiles();
    //2.遍历数组
    for (File file : files) {
        if(file.isFile()){
            //3.判断-文件，变成ZipEntry对象，放入到压缩包当中
            ZipEntry entry = new ZipEntry( name: name + "\\" + file.getName());
            zos.putNextEntry(entry);
            //读取文件中的数据，写到压缩包
            FileInputStream fis = new FileInputStream(file);
            int b;
            while((b = fis.read()) != -1){
                zos.write(b);
            }
            fis.close();
            zos.closeEntry();
        }else{
            //4.判断-文件夹，递归
            toZip(file,zos, name: name + "\\" + file.getName());
        }
    }
}

```

| | |
|--|----------------|
| FileUtils类 | |
| static void copyFile(File srcFile, File destFile) | 复制文件 |
| static void copyDirectory(File srcDir, File destDir) | 复制文件夹 |
| static void copyDirectoryToDirectory(File srcDir, File destDir) | 复制文件夹 |
| static void deleteDirectory(File directory) | 删除文件夹 |
| static void cleanDirectory(File directory) | 清空文件夹 |
| static String readFileToString(File file, Charset encoding) | 读取文件中的数据变成成字符串 |
| static void write(File file, CharSequence data, String encoding) | 写出数据 |
| | |
| IOUtils类 | |
| public static int copy(InputStream input, OutputStream output) | 复制文件 |
| public static int copyLarge(Reader input, Writer output) | 复制大文件 |
| public static String readLines(Reader input) | 读取数据 |
| public static void write(String data, OutputStream output) | 写出数据 |

Commons-io

多线程

1. 并发：在同一时刻，有多个指令在单个CPU上**交替**执行



2. 并行：在同一时刻，有多个指令在多个CPU上**同时**执行

常见的成员方法

| 方法名称 | 说明 |
|--|----------------------|
| <code>String getName()</code> | 返回此线程的名称 |
| <code>void setName(String name)</code> | 设置线程的名字（构造方法也可以设置名字） |
| <code>static Thread currentThread()</code> | 获取当前线程的对象 |
| <code>static void sleep(long time)</code> | 让线程休眠指定的时间，单位为毫秒 |
| <code>setPriority(int newPriority)</code> ↳ <code>final int getPriority()</code> | 设置线程的优先级 获取线程的优先级 |
| <code>final void setDaemon(boolean on)</code> | 设置为守护线程 |
| <code>public static void yield()</code> | 出让线程/礼让线程 |
| <code>public static void join()</code> | 插入线程/插队线程 |

线程的状态

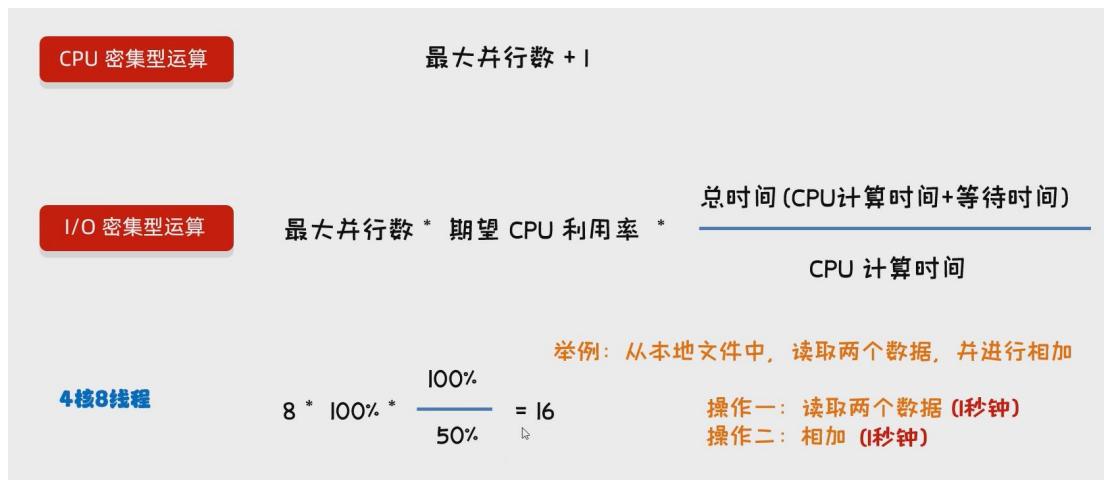


① 创建一个空的池子

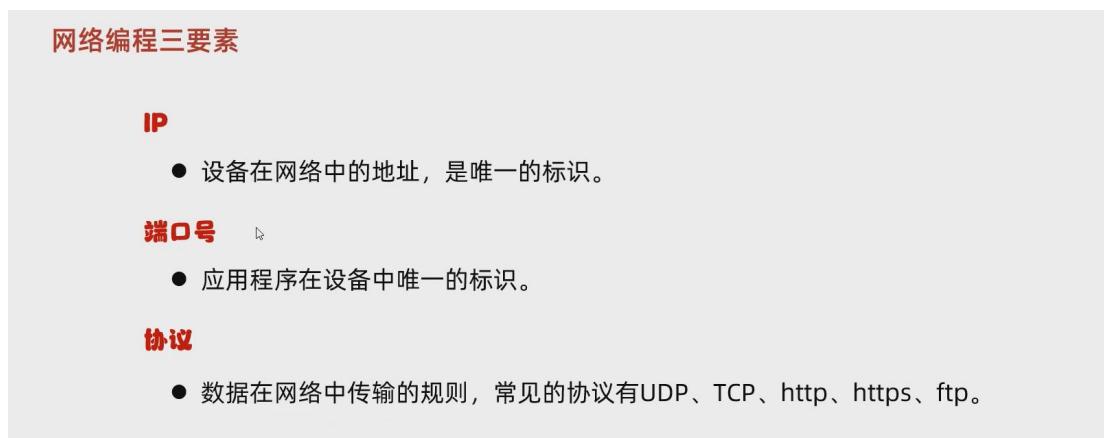
② 有任务提交时，线程池会创建线程去执行任务，执行完毕归还线程

不断的提交任务，会有以下三个临界点：

- ① 当核心线程满时，再提交任务就会排队
- ② 当核心线程满，队伍满时，会创建临时线程
- ③ 当核心线程满，队伍满，临时线程满时，会触发任务拒绝策略



网络编程



1. IP的作用

设备在网络中的地址，是唯一的标识

2. IPv4有什么特点

目前的主流方案

最多只有 2^{32} 次方个ip，目前已经用完了

3. IPv6有什么特点

为了解决IPv4不够用而出现的

最多有 2^{128} 次方个ip

可以为地球上的每一粒沙子都设定ip

端口号

应用程序在设备中唯一的标识。

端口号：由两个字节表示的整数，取值范围：0~65535

其中0~1023之间的端口号用于一些知名的网络服务或者应用。

我们自己使用1024以上的端口号就可以了。

注意：一个端口号只能被一个应用程序使用。

四次挥手)

确保连接断开，且数据处理完毕

