

一、API文档

1.JDK API

- 什么是JDK API?
 - Application Programming Interface 应用程序接口
 - 已经写好的，可供直接调用的类。
 - 比如: 字符串操作、IO流、集合操作、线程、网络编程Socket等等
- JDK常用包

常用包	主要功能
java.lang	Java程序基础类，如:String、线程Thread等，不用通过import关键字导包
java.util	工具类，如集合Collection、随机数Random、时间Date等
java.io	输入输出IO流
java.math	数学运算操作
java.net	网络编程，如Socket
java.sql	操作数据库，如:JDBC
java.text	文本操作等

- API官方文档
不认识的类就去查

2.文档注释规范


- 文档注释一般加载类和方法的开头，用于说明作者、时间、版本等信息。
- 以 `/**` 开头

```
/**
 * zhe shi yi ge zi ding yi de Person lei
 *
 * @author Le
 * @since jdk2.1
 */
public class MyPerson {

    String name;
    int age;
    double salary;

    /**
     * @return int
     * @param i
     */
    public int show(int i) {
        System.out.println("show...");
    }
}
```

```
        return i;
    }
}
```

- 文档注释可以使用javadoc命名生成一个html文档
- image-20220222170812130
 - 生成结果如下

二、常用类

1.String类

1.1 String对象不能改变

- String 的底层是字符数组

```
private final char value[]; //定义了一个字符数组
```

- String类是final修饰的，不能被继承，不能改变，但引用可以重新赋值

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
```

- String采用编码方式是Unicode

1.2 String常量池

- 享元模式: (英语: Flyweight Pattern) 是一种软件[设计模式](#)。它使用共享物件，用来尽可能减少内存使用量以及分享资讯给尽可能多的相似物件；它适合用于只是因重复而导致使用无法令人接受的大量内存的大量物件。通常物件中的部分状态是可以分享。常见做法是把它们放在外部数据结构（常量池），当需要使用时再将它们传递给享元(字符串的直接量)。
- Java为了提高性能，静态字符串在常量池中创建，并且是同一对象，使用时直接拿取。
- 对于重复的字符串直接量，JVM会去常量池查找，如果有则返回，无则创建。
 - 测试:常量池

```
//在常量池中查找，没有创建一个"Hello";
String s = "Hello";
//在常量池中查询，不会创建新对象
String s1 = "Hello";
System.out.println(s == s1); //true
String s2 = new String("Hello");
System.out.println(s1 == s2); //false
```

- 扩展：常量池创建了几个对象的问题
 - String s = new String(String:"a");
 - 两个或1个，如果常量池中有a则返回，如果没有则创建一个a，加上new的对象是两个。
 - 案例:

```
//扩展 下列代码创建了几个对象？
String s3 = "Java" + "Hello";//创建了 0个或1个 对象
String s4 = "JavaHello";
//常量拼接不会创建新对象 “Java” + “Hello” <==> “JavaHello”
System.out.println("常量拼接:" + (s3 == s4));

String s5 = "Hello";//0或1
//1个(JavaHello)或3个("Hello","Java","JavaHello")
String s6 = "Java" + s5;
System.out.println("变量拼接常量:" + (s6 == s4));

String s7 = new String("world" + s5 + "Java");//2个或5个
//5个对象("world","Hello","Java","worldHelloJava",new)
//2个对象("worldHelloJava",new)
```

1.3 构造方法

<code>String()</code>	初始化一个新创建的 <code>String</code> 对象，它表示一个空字符序列。
<code>String(byte[] bytes)</code>	构造一个新的 <code>String</code> ，方法是使用平台的默认字符集解码字节的指定数组。
<code>String(byte[] bytes, int offset, int length)</code>	已过时。 该方法无法将字节正确转换为字符。从 <code>JDK 1.1</code> 起，完成该转换的首选方法是通过 <code>String</code> 构造方法，该方法接受一个字符集名称或使用平台的默认字符集。
<code>String(byte[] bytes, int offset, int length, String charsetName)</code>	构造一个新的 <code>String</code> ，方法是使用指定的字符集解码字节的指定子数组。
<code>String(byte[] bytes, int offset, int length, int count)</code>	已过时。 该方法无法将字节正确转换为字符。从 <code>JDK 1.1</code> 开始，完成该转换的首选方法是通过 <code>String</code> 构造方法，它接受一个字符集名称，或者使用平台默认的字符集。
<code>String(byte[] bytes, int offset, int length, String charsetName)</code>	构造一个新的 <code>String</code> ，方法是使用指定的字符集解码字节的指定子数组。
<code>String(char[] value)</code>	分配一个新的 <code>String</code> ，它表示当前字符数组参数中包含的字符序列。
<code>String(char[] value, int offset, int count)</code>	分配一个新的 <code>String</code> ，它包含来自该字符数组参数的一个子数组的字符。
<code>String(int[] codePoints, int offset, int count)</code>	分配一个新的 <code>String</code> ，它包含该 <code>Unicode</code> 代码点数组参数的一个子数组的字符。
<code>String(String original)</code>	初始化一个新创建的 <code>String</code> 对象，表示一个与该参数相同的字符序列；换句话说，新创建的字符串是该参数字符串的一个副本。
<code>String(StringBuffer buffer)</code>	分配一个新的字符串，它包含当前包含在字符串缓冲区参数中的字符序列。
<code>String(StringBuilder builder)</code>	分配一个新的字符串，它包含当前包含在字符串生成器参数中的字符序列。

- 测试:

```
//1无参构造
String s = new String();
System.out.println(s);//空字符序列
//2.byte[]
/*
GBK编码：是指中国的中文字符，其它它包含了简体中文与繁体中文字符，另外还有一种字
符“gb2312”，这种字符仅能存储简体中文字符。

UTF-8编码：它是一种全国家通过的一种编码，如果你的网站涉及到多个国家的语言，那么建议你选择
UTF-8编码。
*/
String s1 = new String(new byte[]{'1','2','3','4','5'},1,4,"utf8");
System.out.println(s1);
//3.char[]
String s2 = new String(new char[]{'H','E','L','L','O'});
System.out.println(s2);
//4.String
String s3 = new String("b");
System.out.println(s3);
//5.StringBuffer
String s4 = new String(new StringBuffer("abc"));
```

```
System.out.println(s4);
//6.StringBuilder
String s5 = new String(new StringBuilder("abc"));
System.out.println(s5);
```

1.4 String常用方法

1.4.1 String的判断方法

`boolean equals(Object obj)`: 比较字符串的内容是否相同
`boolean equalsIgnoreCase(String str)`: 比较字符串的内容是否相同,忽略大小写
`boolean startsWith(String str)`: 判断字符串对象是否以指定的`str`开头
`boolean endsWith(String str)`: 判断字符串对象是否以指定的`str`结尾

- 测试:

```
String s = "abc";
String s1 = "Abc";
String s2 = new String("abc");
//1. equals方法
System.out.println(s == s2); //f 比较的地址值
System.out.println(s.equals(s2)); //t 比较的是内容
//2. equalsIgnoreCase 不区分大小写比较
System.out.println(s.equalsIgnoreCase(s1)); //t
//3. startsWith 判断以**字符串开头
System.out.println(s.startsWith("ab")); //t
//4. endsWith 以**结尾
System.out.println(s.endsWith("c")); //t
System.out.println(s.endsWith("d")); //f
```

1.4.2 String的截取

`int length()`: 获取字符串的长度, 其实也就是字符个数
`char charAt(int index)`: 获取指定索引处的字符
`int indexOf(String str)`: 获取`str`在字符串对象中第一次出现的索引
`String substring(int start)`: 从`start`开始截取字符串
`String substring(int start, int end)`: 从`start`开始, 到`end`结束截取字符串。前包后不包

- 测试:

```
public static void main(String[] args) {
    //int[] i = new int[5];
    //System.out.println(i.length);
    String s = "Hello";
    //1.length 获取字符串长度
    //源码: return value.length;
    System.out.println(s.length());
    //2.charAt
    //return value[index];
    System.out.println(s.charAt(0)); //H
    //3.indexOf 通过首次出现的字符串的索引
    System.out.println(s.indexOf("e1")); //0
    /*
        4.subString(int,int)前包后不包的
    */
}
```

```

System.out.println("前:" + s.length());
System.out.println(s.substring(1)); //ello
System.out.println("后:" + s.length());
System.out.println(s.substring(1,3)); //el
}

```

1.4.3 String的转换

char[] `toCharArray()`: 把字符串转换为字符数组
 String `toLowerCase()`: 把字符串转换为小写字符串
 String `toUpperCase()`: 把字符串转换为大写字符串

- 测试:

```

public static void main(String[] args) {
    String s = "Hello world";
    //1 toCharArray 字符串--->字符数组
    char[] c = s.toCharArray();
    System.out.println(Arrays.toString(c));
    //字符数组--->字符串
    c = new char[]{'H','e','l'};
    String s1 = new String(c);
    System.out.println(s1);
    //2.toUpperCase()全部变大写
    System.out.println(s.toUpperCase());
    //3.toLowerCase()全部变小写
    System.out.println(s.toLowerCase());
}

```

1.4.4 其他方法

String `trim()`: 去除字符串两端空格
 String[] `split(String str)`: 按照指定符号分割字符串

- 测试:

```

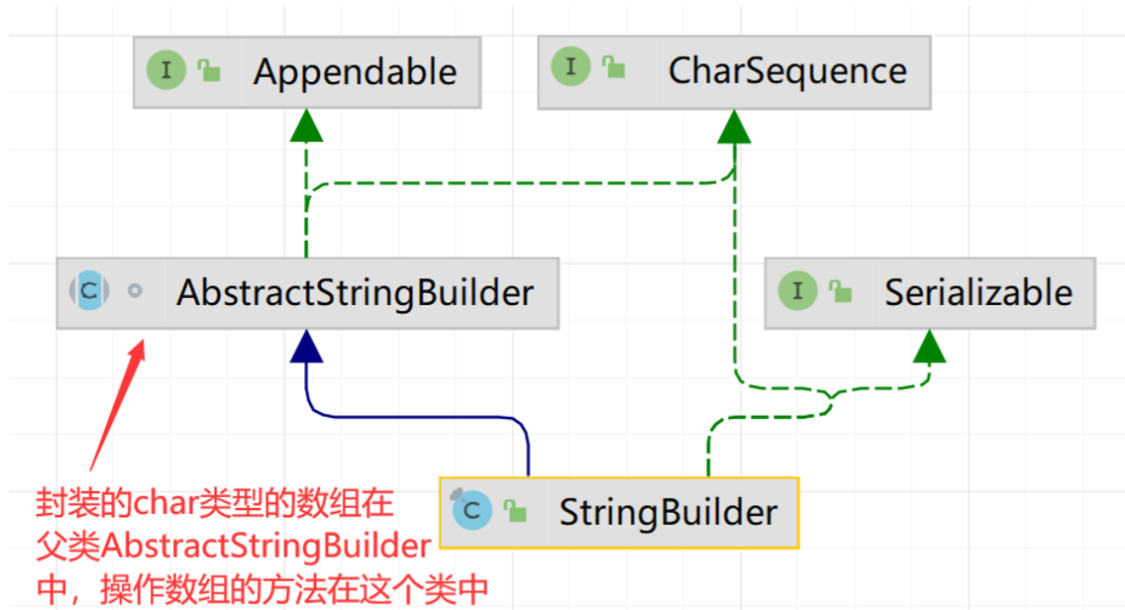
public static void main(String[] args) {
    //1.trim 去除字符串两端的空格
    String s = " Hello world ";
    System.out.println(s.trim()); // "Hello world"
    //2.split 分割字符串
    System.out.println(s.length());
    System.out.println(Arrays.toString(s.split("o")));
    //[, Hello, world]
    System.out.println(s.length());
    //案例: 拿去baidu
    String s1 = "www.baidu.com";
    //1. 获取两个的索引
    int start = s1.indexOf(".");
    int end = s1.lastIndexOf(".");
    //2. 根据点截取字符串
    String result = s1.substring(start+1, end);
    System.out.println(result);
}

```

1.5 StringBuilder

1.5.1 可变字符串

- StringBuilder是一个线程非安全可变字符串序列。



- StringBuilder底层在其父类(AbstractStringBuilder)中封装了一个char类型的数组
- StringBuilder创建的初始容量是16

```
public StringBuilder() {  
    super(capacity: 16);  
}
```

初始容量为16

1.5.2 构造方法

StringBuilder()

构造一个不带任何字符串的字符串生成器，其初始容量为 16 个字符。

StringBuilder(int capacity)

构造一个不带任何字符串的字符串生成器，其初始容量由 capacity 参数指定。

StringBuilder(String str)

构造一个字符串生成器，并初始化为指定的字符串内容。

- 测试

```

public static void main(String[] args) {
    //1 构造方法
    StringBuilder sb = new StringBuilder();
    System.out.println("无参构造:" + sb);
    System.out.println("初始容量:" + sb.capacity()); //16
    StringBuilder sb1 = new StringBuilder(32);
    System.out.println("自定义容量:" + sb1.capacity()); //32
    StringBuilder sb2 = new StringBuilder("Hello");
    System.out.println("Hello容量:" + sb2.capacity()); //21
    System.out.println("Hello长度:" + sb2.length()); //5
    System.out.println(sb2); //hello
}

```

1.5.3 常用方法

- **append(任意内容): 追加字符串**

- 追加扩容问题:

- 扩容实现原理: 数组的Arrays.copyOf方法

```

private void ensureCapacityInternal(int minimumCapacity) {
    // overflow-conscious code
    if (minimumCapacity - value.length > 0) {
        value = Arrays.copyOf(value,
            newCapacity(minimumCapacity));
    }
}

```

- 扩容算法: 源数组容量乘2加2

```

private int newCapacity(int minCapacity) {
    // overflow-conscious code
    int newCapacity = (value.length << 1) + 2;
    if (newCapacity - minCapacity < 0) {
        newCapacity = minCapacity;
    }
    return (newCapacity <= 0 || MAX_ARRAY_SIZE - newCapacity < 0)
        ? hugeCapacity(minCapacity)
        : newCapacity;
}

```

- 测试:

```

public static void main(String[] args) {
    //1.append方法
    StringBuilder sb1 = new StringBuilder(); //初始容量16
    sb1.append("1111111111111111"); //追加的字符串长度:17 超出了初始容量
    System.out.println("sb1的容量:" + sb1.capacity()); //34 17*2+2
    sb1.append("1111111111111111"); //18 再次追加超出
    System.out.println("sb1的内容:" + sb1);
    System.out.println("sb1的容量:" + sb1.capacity()); //70 34*2+2
    System.out.println("sb1的长度:" + sb1.length()); //35
}

```

- insert(int,任意类型)方法: 插入
- delete(int start,int end): 删除
- replace(int start,int end,String s):替换
- reverse():字符串的反转

◦ 测试:

```
public static void main(String[] args) {
    //1 insert方法
    StringBuffer sb1 = new StringBuffer("Heo");
    sb1.insert(2, "ll");//offset从这个位置处开始!
    System.out.println(sb1);
    System.out.println(sb1.length());//5
    System.out.println(sb1.capacity());//19
    //2 delete
    //sb1.delete(2,4);//前包后不包
    //System.out.println(sb1);
    //3 replace
    sb1.replace(2,4, "ww");
    System.out.println(sb1);
    //4 reverse
    System.out.println(sb1.reverse());
}
```

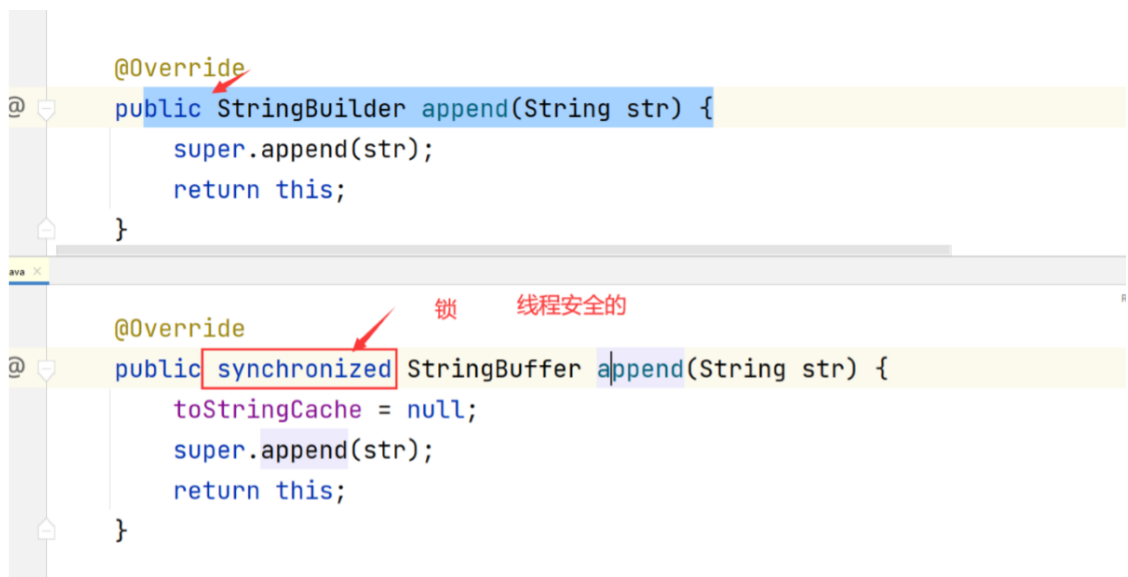
• 方法的返回值问题

◦ 因为这些方法的返回值语句是return this;可以连续调用

```
public static void main(String[] args) {
    StringBuilder sb = new StringBuilder();
    //因为这些方法的返回值语句是return this;可以连续调用
    sb.append("Hello")
      .append(" ")
      .append("world")
      .reverse()
      .delete(5,6);
    System.out.println(sb);
}
```

1.5.4 StringBuilder和StringBuffer区别

- StringBuffer是线程安全的



- StringBuilder是线程非安全的，因此运行速率较快。

2.正则表达式

2.1 基本正则表达式

- 正则表达式:一组特定的字符，用来描述文本规则的工具。
- 本质：就是字符串

[a-z]: 一个任意a~z的字符
[0-9]: 一个任意0~9的字符

2.1.1 正则表达式规则

- 字符集合

基本正则表达式	说明
[abc]	a、b、c中任意一个字符
[^abc]	除了a、b、c的任意一个字符
[a-z]	a到z中任意一个字符
[a-zA-Z0-9]	a至z、A至Z、0至9中任意一个字符
[a-z&[^bc]]	a至z中除了xy的任意一个字符

- 预定义字符集

正则表达式	说明
.	任意一个字符
\d	任意一个数字字符, 等价于[0-9]
\D	非数字字符
\w	任意一个单词字符,等价于[a-zA-Z0-9_]
\W	非单词字符
\s	任意一个空白字符,
\S	非空白字符

2.1.2 预定义字符集

- 测试:

```
public static void main(String[] args) {
    String s1 = "[a-z]";
    String s2 = "[abc]";
    String s3 = "[^abc]";
    String s4 = "[a-zA-Z0-9]";
    String s5 = "[[a-z]&&[^xy]]";
    //预定义字符集
    String s6 = ".";
    String s7 = "\\d";//[0-9]
    String s8 = "\\w";//[a-zA-Z0-9_]
    String s9 = "\\W";//[^a-zA-Z0-9^_]
    String s10 = "\\s";//空白字符
    //matches 判断字符串是否匹配正则 参数:正则表达式
    System.out.println("0".matches(s1));
    System.out.println("a".matches(s2));
    System.out.println("d".matches(s3));
    System.out.println("G".matches(s4));
    System.out.println("b".matches(s5));
    System.out.println("$".matches(s6));
    System.out.println("_".matches(s7));
    System.out.println("_".matches(s8));
    System.out.println("(" .matches(s9));
    System.out.println(" ".matches(s10));
}
```

2.1.3 数量词

- 量词:表示有几个字符

正则表达式	说明
[表达式]?	表示0个或1个[表达式] 如: [0-9]?表示0个或1个0-9的任意字符
[表达式]*	表示0个或任意多个[表达式]
[表达式]+	表示1个或任意多个[表达式]
[表达式]{n}	表示n个[表达式]
[表达式]{n,}	表示至少n个[表达式]
[表达式]{m,n}	表示m个到n个[表达式]

- 测试:

```
public static void main(String[] args) {
    String s1 = "[a-z]+";
    String s2 = "[abc]?";
    String s3 = "[^abc]*";
    String s4 = "[a-zA-Z0-9]+";
    String s5 = "[[a-z]&&[abc]]{6}";
    //预定义字符集
    String s6 = ".+";
    String s7 = "\\d?";//[0-9]
    String s8 = "\\w+";//[a-zA-Z0-9_]
    String s9 = "\\w{2,}";//[a-zA-Z0-9^_]
    String s10 = "\\s{2,4}";//空白字符
    //matches 判断字符串是否匹配正则 参数:正则表达式
    System.out.println("").matches(s1);
    System.out.println("").matches(s2);
    System.out.println("ddd".matches(s3));
    System.out.println("GYU".matches(s4));
    System.out.println("qwerty".matches(s5));
    System.out.println("$!@#%&".matches(s6));
    System.out.println("").matches(s7);
    System.out.println("_____".matches(s8));
    System.out.println("(").matches(s9);
    System.out.println("    ".matches(s10));
}
```

2.1.4 分组和起止符

() :表示分组,意思括起来的内容是一个整体
 ^ :开始标识
 \$:结束标识

- 测试:

```
public static void main(String[] args) {
    String s = "^(\+86|0086)?\s?\d{11}$";
    System.out.println("86 15682961080".matches(s));//f
    System.out.println("+8615682961080".matches(s));//t
}
```

2.2 String的正则API

- matches方法:一个字符串和正则表达式是否匹配

```
String s2 = "^((\\+86|0086)?\\s?\\d{11})$";
System.out.println("0086 16532659865".matches(s2));
```

- split方法:以正则表达式分割字符串
- replaceAll:以正则替换字符串

测试:

```
public static void main(String[] args) {
    String s1 = "www.23baidu.98com";
    String[] s1Arr = s1.split("\\.\\d{2}");//".23"
    System.out.println(Arrays.toString(s1Arr));
    String s2 = "abc123fgh456yiu798";//把所有的数字替换为中文"数"
    s2 = s2.replaceAll("\\d", "数");
    System.out.println(s2);
}
```

3.Object类

3.1 Object

- java.lang.Object位于java继承关系的顶端。
- Object类是所有类的父类
- Object类的引用可以指向任意对象。

```
Object person = new Person();
```

3.2 toString方法

- 当我们直接打印对象时，结果为对象地址值。

```
//1540e19d散列码
//cn.tedu.day01.Person 全类名
//源码:
getClass().getName() + "@" + Integer.toHexString(hashCode());
```

- 重写父类Object的toString方法可以将对象以字符串的形式输出

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

- 在定义实体类时强烈建议给每一个对象都重写toString方法

3.3 equals方法

- 在没有重写equals方法的情况下，它俩比较的都是地址值。
- 一般情况下，在每一个类中都应重写父类Object的equals方法，使比较更有意义

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Person person = (Person) o;
    return age == person.age && Objects.equals(name, person.name);
}
```

- 重写equals后，对象比较的是它的成员变量值。

3.4 equals和==区别

- 如果没有重写equals方法，则==和equals没有区别。都比较的是地址信息。
- 如果重写equals，则比较的是两个对象内容(成员变量的值)。

3.5 实体类创建

- 创建时应进行如下操作:
 - 私有化属性
 - 无参构造
 - 有参构造
 - set和get方法
 - toString方法
 - equals和hashCode方法
- 如下:

```
import java.util.Objects;

public class User {

    private int id;
    private String username;
    private String password;

    public User() {
    }

    public User(int id, String username, String password) {
        this.id = id;
        this.username = username;
        this.password = password;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getUsername() {
```

```

        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", username='" + username + '\'' +
            ", password='" + password + '\'' +
            '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return id == user.id && Objects.equals(username, user.username) &&
Objects.equals(password, user.password);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, username, password);
    }
}

```

4.包装类

4.1 概述

- 包装类就是基本类型转化成为对象的类(java.lang.Number类下的子类提供了6种基本类型的包装类)。
- 每一个基本类型都有一个与之对应的包装类(wrapper)。

基本类型	对应包装类型
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character --直接父类为Object
boolean	java.lang.Boolean -- 直接父类为Object

4.2 转换关系(重)

- 包装类 ---> 基本类型
- 基本类型 ---> 包装类
- 基本类型 ---> String
- String ---> 基本类型
 - 测试

```
public static void main(String[] args) {
    //包装类 -> 基本类型
    Integer i = new Integer(0);
    int i1 = i.intValue();
    //基本类型->包装类
    int i2 = 123;
    Integer integer = Integer.valueOf(i2);
    //基本类型->String
    int i3 = 456;
    String string = String.valueOf(i3);
    //String -> 基本类型
    String s = "1.0";
    double d = Double.parseDouble(s);
}
```

4.3 构造方法

- 构造方法中可以传入String值

构造方法摘要

`Integer`(int value)

构造一个新分配的 `Integer` 对象，它表示指定的 `int` 值。

`Integer`(String s)

构造一个新分配的 `Integer` 对象，它表示 `String` 参数所指示的 `int` 值。

- 注意:

- 如果传入的字符串表示的值不匹配基本类型，则转化时会报NumberFormatException数字格式化异常

```
Integer i4 = new Integer("zhang3");
System.out.println(i4);
```

4.4 自动装箱和拆箱

- jdk5.0后加入了autoboxing(自动装箱和拆箱)功能。
- 自动装箱拆箱就是编译器在生成字节码时自动完成了包装类和基本类型间的相互转换的方法调用
 - 装箱: 基本类型--->包装类
 - 拆箱: 包装类--->基本类型
- 测试:

```
public static void main(String[] args) {
    Integer i = 100; //Integer.valueOf(100) 自动装箱
    Integer i2 = 200; //自动装箱
    //int i3 = i.intValue();
    //int i4 = i2.intValue();
    //int i5 = i3 + i4;
    //Integer i6 = Integer.valueOf(i5);
    Integer i3 = i + i2; //自动先拆箱后自动装箱
}
```

5.时间日期类

5.1 Date类

- Java中的时间是用一个固定时间点的毫秒数表示
- 纪元(epoch): 1970年1月1日 00:00:00
- 常见的时区:
 - GMT:格林威治时间
 - CST:中国时间
 - UTC:全球标准时间
 - HST:夏威夷标准时间

5.1.1 构造方法

- Date对象封装了当前时间
- 构造方法:

```
public static void main(String[] args) {
    Date date = new Date();
    System.out.println(date); //打印当前时间
    //Sat Dec 25 19:18:01 CST 2021
    Date date1 = new Date(1000);
    System.out.println(date1);
    //Thu Jan 01 08:00:01 CST 1970
}
```


5.1.2 setTime和getTime

```
public static void main(String[] args) {
    Date date = new Date();
    date.setTime(1000 - (8*60*60*1000));
    System.out.println(date);
    System.out.println(date.getTime());

    //案例-打印明天的当前时间
    Date date1 = new Date();
    long time = date1.getTime();//获取当前毫秒数
    time += 24*60*60*1000;
    date1.setTime(time);
    System.out.println(date1);
}
```

5.2 SimpleDateFormat类

- 格式化和解析日期的类，可以对时间格式化或规范化

5.2.1 构造方法

`SimpleDateFormat()`

用默认的模式和默认语言环境的日期格式符号构造 `SimpleDateFormat`。

`SimpleDateFormat(String pattern)`

用给定的模式和默认语言环境的日期格式符号构造 `SimpleDateFormat`。

- 测试:

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH-mm-ss E");
```

5.2.2 日期匹配字符

字符	意义	举例
y	年	yyyy年---2013年; yy-13年
M	月	MM月-12月; M月---1月
d	日	dd日---06日; d日---6日
E	星期	E---星期日
H	小时(24小时制)	HH--15时
h	小时(12小时制)	hh(a上午、p下午)--1时
m	分钟	mm分
s	秒	ss秒

5.2.3 常用方法

`String format(Date date)`: 将给定的 `Date` 格式化为日期/时间字符串
`Date parse(String text)` 分析字符串的文本, 生成 `Date`

- 测试:

```
public static void main(String[] args) throws ParseException {  
  
    //format 格式化  
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH-mm-ss E");  
    Date date = new Date();  
    String s = sdf.format(date);  
    System.out.println(s); //2021-12-25 19:48:11  
  
    //解析  
    String s1 = "2021.5.12";  
    //参数的字符串格式必须匹配s1的规则  
    //如果不匹配则会抛出ParseException -- 解析异常  
    SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy.MM.dd");  
    Date date1 = sdf1.parse(s1);  
    System.out.println(date1); //Wed May 12 00:00:00 CST 2021  
}
```

5.3 Calendar类

- Calendar: 抽象类, 用于时间分量的计算。
- GregorianCalendar: 是抽象类Calendar的子类

5.3.1 构造方法

- GregorianCalendar提供了重载的构造方法

1. `GregorianCalendar()` 在具有默认语言环境的默认时区内使用当前时间构造一个默认的 `GregorianCalendar`。
2. `GregorianCalendar(int year, int month, int dayOfMonth)`
在具有默认语言环境的默认时区内构造一个带有给定日期设置的 `GregorianCalendar`。
3. `GregorianCalendar(int year, int month, int dayOfMonth, int hourOfDay, int minute)`
为具有默认语言环境的默认时区构造一个具有给定日期和时间设置的 `GregorianCalendar`。
- 4 `GregorianCalendar(int year, int month, int dayOfMonth, int hourOfDay, int minute, int second)`
为具有默认语言环境的默认时区构造一个具有给定日期和时间设置的 `GregorianCalendar`。

- 测试

```
Calendar calendar = Calendar.getInstance(); //获取calendar对象  
//输出Calendar对象的实际类型  
System.out.println(calendar.getClass().getName()); //GregorianCalendar  
//getTime返回对应的date对象  
System.out.println(calendar.getTime()); //Sat Dec 25 20:20:12 CST 2021  
//创建GregorianCalendar  
GregorianCalendar gc = new GregorianCalendar(2013, 10, 25);  
System.out.println(gc.getTime()); //Mon Nov 25 00:00:00 CST 2013
```

5.3.2 计算日期分量

- Calendar中提供了很多静态常量，用来描述时间分量
- 测试:

```
public static void main(String[] args) {  
  
    //获取时间分量  
    Calendar calendar = Calendar.getInstance();  
    calendar.set(Calendar.YEAR,2021);  
    calendar.set(Calendar.MONTH,11);//12月  
    calendar.set(Calendar.DATE,20);  
  
    System.out.println("时间为:" + calendar.getTime());  
  
    //1 一周中的星期几  
    int dayOfWeek = calendar.get(Calendar.DAY_OF_WEEK);//一周中的第几天  
    System.out.println(dayOfWeek);//4 星期三  
  
    //2 一年的第几天  
    int dayOfYear = calendar.get(Calendar.DAY_OF_YEAR);  
    System.out.println(dayOfYear);//20  
  
    //3.一月中第几周  
    int weekOfMonth = calendar.get(Calendar.WEEK_OF_MONTH);  
    System.out.println(weekOfMonth);//4  
  
    //4. 当前月中的第几个星期。  
    int dayOfWeekInMonth = calendar.get(Calendar.DAY_OF_WEEK_IN_MONTH);  
    System.out.println(dayOfWeekInMonth);//3  
}
```

5.3.3常用方法

- set(int fields,int amount):设置时间分量
- get(int fields,int amount):获取时间分量
- getActualMaximum(int fields):返回日历字段可能拥有的最大值
- add(int fields,int amount):为指定时间分量加上指定的值。
 - 测试:

```
public static void main(String[] args) {  
    //获取时间分量  
    Calendar calendar = Calendar.getInstance();  
    calendar.set(Calendar.YEAR,2021);  
    calendar.set(Calendar.MONTH,10);//11月  
    calendar.set(Calendar.DATE,20);  
    calendar.add(Calendar.YEAR,-1);  
    System.out.println("时间为:" + calendar.getTime());//2021.11.20  
  
    int days = calendar.getActualMaximum(Calendar.DAY_OF_YEAR);  
    System.out.println(days);  
  
    int dayofMonths = calendar.getActualMaximum(Calendar.DAY_OF_MONTH);  
    System.out.println(dayofMonths);  
}
```

}

1.Collection

	<code>addAll(Collection c)</code> 将指定 collection 中的所有元素都添加到此 collection 中（可选操作）。
void	<code>clear()</code> 移除此 collection 中的所有元素（可选操作）。
boolean	<code>contains(Object o)</code> 如果此 collection 包含指定的元素，则返回 true。
boolean	<code>containsAll(Collection<?> c)</code> 如果此 collection 包含指定 collection 中的所有元素，则返回 true。
boolean	<code>equals(Object o)</code> 比较此 collection 与指定对象是否相等。
int	<code>hashCode()</code> 返回此 collection 的哈希码值。
boolean	<code>isEmpty()</code> 如果此 collection 不包含元素，则返回 true。
<code>Iterator<E></code>	<code>iterator()</code> 返回在此 collection 的元素上进行迭代的迭代器。
boolean	<code>remove(Object o)</code> 从此 collection 中移除指定元素的单个实例，如果存在的话（可选操作）。
boolean	<code>removeAll(Collection<?> c)</code> 移除此 collection 中那些也包含在指定 collection 中的所有元素（可选操作）。
boolean	<code>retainAll(Collection<?> c)</code> 仅保留此 collection 中那些也包含在指定 collection 的元素（可选操作）。
int	<code>size()</code> 返回此 collection 中的元素数。
<code>Object[]</code>	<code>toArray()</code> 返回包含此 collection 中所有元素的数组。
<code><T> T[]</code>	<code>toArray(T[] a)</code> 返回包含此 collection 中所有元素的数组；返回数组的运行时类型与指定数组的运行时类型相同。

1.1 Collection接口

- 集合: 存储数据的容器(数据结构)
- Collection: 是一个接口，定义了操作集合相关方法
- Collection 下有两个子接口
 - List
 - Set

1.1.1 集合存储对象的引用

- 集合存储的是对象的引用地址值，而不是对象本身。存储原理参考对象数组。

1.1.2 常用方法

- 测试:

```
public static void main(String[] args) {
    Collection<String> collections = new ArrayList<String>();
    System.out.println(collections); //[]
    //1 add方法
```

```

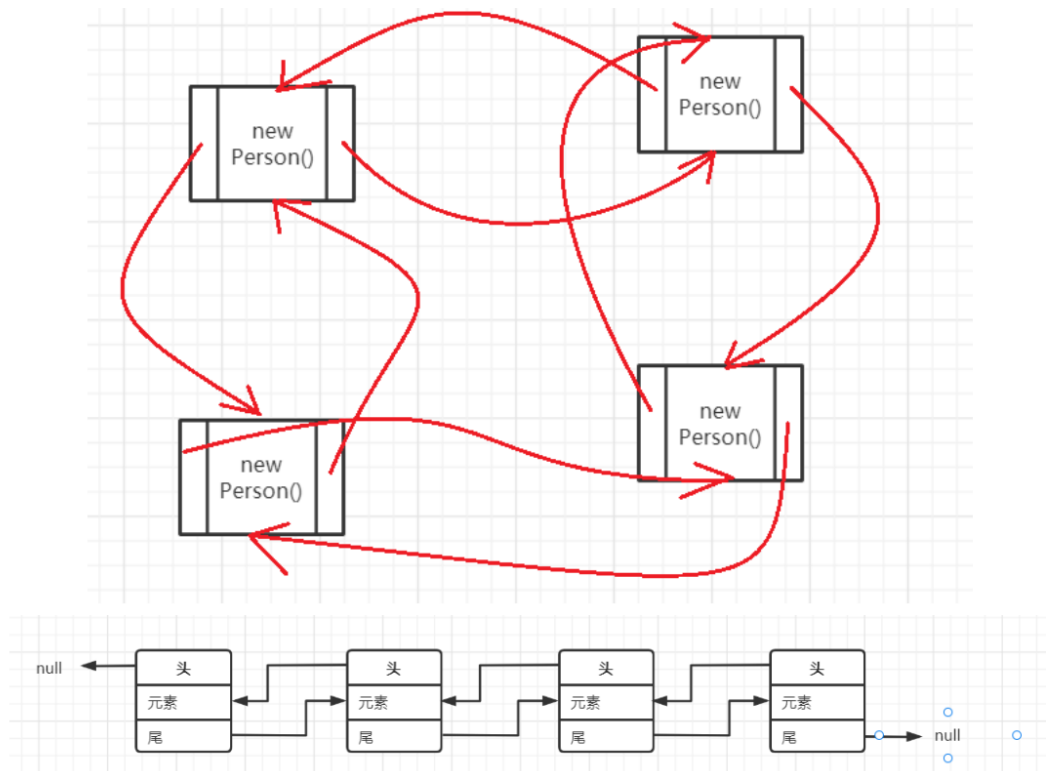
collections.add("java");
collections.add("c");
//2 addAll
Collection<String> collections2 = new ArrayList<String>();
collections2.add("php");
collections2.add("javascript");
collections2.add("c++");
collections.addAll(collections2);
System.out.println(collections);
//3 contains
boolean res = collections.contains("c");
System.out.println(res);
//4 containsAll
Collection<String> collections3 = new ArrayList<String>();
collections3.add("c");
collections3.add("phx");
System.out.println(collections.containsAll(collections3));
//5 clear
collections3.clear();
//6 isEmpty
System.out.println(collections3.isEmpty());
//7 remove
System.out.println(collections);//[ ]
collections.remove("java");
System.out.println(collections);
//8 retainAll
collections.retainAll(collections2);
System.out.println(collections);
//9 size
System.out.println(collections.size());
//10 toArray
Object[] objs =collections.toArray();
/*
    如果这里强制类型转化时，向下造型会发生ClassCastException
*/
System.out.println(Arrays.toString(objs));
}

```

2.List

2.1 List接口

- List: 有序(自然顺序)有重复的集合
- List常见实现类:
 - ArrayList: 数组实现 线程非安全效率高
 - LinkedList: 双向链表 不存在扩容问题



- Vector: 数组实现 线程安全, 效率低不建议使用
- Stack: Vector的子类 数组实现 遵循先进后出的原则

2.2 List使用

- 测试有序有重复

```
public static void main(String[] args) {
    List<Integer> lists = new ArrayList<Integer>();
    lists.add(23);
    lists.add(15);
    lists.add(64);
    System.out.println(lists); //23 15 64
    /*
        有序:指自然顺序 ; 存储元素的先后顺序 不是排序
    */
    lists.add(64);
    //可以有重复元素
    System.out.println(lists);
}
```

- ArrayList的初始容量和扩容机制
 - 初始容量:10

```
private static final int DEFAULT_CAPACITY = 10;
```

- 扩容机制: 原来容量的1.5倍

```
int newCapacity = oldCapacity + (oldCapacity >> 1);
```

- ArrayList底层是数组实现, 线性结构, 更适合查询

2.3 ArrayList和LinkedList区别(面试)

1、数据结构不同

ArrayList是**Array**(动态数组)的数据结构，**LinkedList**是**Link**(链表)的数据结构。

2、效率不同

当随机访问**List**（**get**和**set**操作）时，**ArrayList**比**LinkedList**的效率更高，因为**LinkedList**是线性的数据存储方式，所以需要移动指针从前往后依次查找。

当对数据进行增加和删除的操作(**add**和**remove**操作)时，**LinkedList**比**ArrayList**的效率更高，因为**ArrayList**是数组，所以在其中进行增删操作时，会对操作点之后所有数据的下标索引造成影响，需要进行数据的移动。

3、自由性不同

ArrayList自由性较低，因为它需要手动的设置固定大小的容量，但是它的使用比较方便，只需要创建，然后添加数据，通过调用下标进行使用；而**LinkedList**自由性较高，能够动态的随数据量的变化而变化，但是它不便于使用。

4、主要控件开销不同

ArrayList主要控件开销在于需要在**list**列表预留一定空间；而**LinkedList**主要控件开销在于需要存储结点信息以及结点指针信息。

2.4 截取子串subList

- **subList(int fromIndex, int toIndex)**: 索引为前包后不包

```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<Integer>();
    for (int i = 0; i < 10; i++) {
        list.add((int)(Math.random()*100));
    }
    System.out.println("List中的元素为: " + list);
    //使用subList截取子list//索引为3-7且不包含7
    List<Integer> subList = list.subList(3,7);
    System.out.println("截取子List为:" + subList);
    //对子list元素*10
    for (int i = 0; i < subList.size(); i++) {
        subList.set(i,subList.get(i) * 10);
    }
    System.out.println("运算后的子list为:" + subList);
    //打印原list
    System.out.println("List的元素为:" + list);
}
```

◦ 结论:

- 截取字符串操作后，原list也会随之而改变
- List和截取后的subList公用一个数据空间资源

2.5 list与数组之间的转换

- 集合转化为数组的两种方式:

- **Object[] toArray()**
- **T[] toArray(T[] a)**

- 测试:

```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<Integer>();
    for (int i = 0; i < 10; i++) {
        list.add((int)(Math.random()*100));
    }
    System.out.println("List中的元素为: " + list);
    //1.集合转换为数组的第一种方式
    Object[] objects = list.toArray();
    System.out.println(Arrays.toString(objects));
    //会产生类型转换异常--不建议使用
    Integer[] integers = (Integer[]) objects;
    System.out.println(Arrays.toString(integers));
    //原因一看就知道了，不能将Object[] 转化为String[].转化的话只能是取出每一个元素再转化，

    //1.集合转换为数组的第二种方式--建议使用
    Integer[] integers2 = list.toArray(new Integer[list.size()]);
    System.out.println(Arrays.toString(integers2));
}
```

- 数组转化为集合

```
Integer[] integers3 = {1,2,3};
List<Integer> list2 = Arrays.asList(integers3);
System.out.println(list2);
//转化后不能对list进行操作，如果操作就会抛出异常
list2.add(4);
```

```
Exception in thread "main" java.lang.UnsupportedOperationException Create breakpoint
at java.util.AbstractList.add(AbstractList.java:148)
```

3.Set

3.1 Set接口

- Set: 无序无重复的集合
- Set的常见实现类:
 - HashSet: 底层使用HashMap操作数据
 - TreeSet: 底层封装了TreeMap, 对元素可以进行排序, 树结构实现
 - LinkedHashSet:具有自然顺序的操作

3.2 Set测试

- 测试无序无重复集合

```
public static void main(String[] args) {
    Set<Integer> sets = new HashSet<Integer>();
    sets.add(1);
    sets.add(-1);
    sets.add(24);
    sets.add(13);
    sets.add(13);
    sets.add(2);
    sets.add(3);
    System.out.println(sets);//[ -1, 1, 2, 3, 24, 13]
}
```

- 结果为[-1, 1, 2, 3, 24, 13], 去除重复元素, 且不是按照添加元素顺序输出
- 测试TreeSet

```
public static void main(String[] args) {
    Set<Integer> sets = new TreeSet<Integer>();
    sets.add(1);
    sets.add(-1);
    sets.add(24);
    sets.add(13);
    sets.add(13);
    sets.add(2);
    sets.add(3);
    System.out.println(sets);//[ -1, 1, 2, 3, 13, 24]
}
```

- 结果为[-1, 1, 2, 3, 13, 24], 排序后打印
- 测试LinkedHashSet

```
public static void main(String[] args) {
    Set<Integer> sets = new TreeSet<Integer>();
    sets.add(1);
    sets.add(-1);
    sets.add(24);
    sets.add(13);
    sets.add(13);
    sets.add(2);
    sets.add(3);
    System.out.println(sets);//[ 1, -1, 24, 13, 2, 3]
}
```

- 结果为[1, -1, 24, 13, 2, 3], 按照插入顺序打印

4.集合排序问题

4.1 Collections

- Collections是为集合提供的工具类
- 集合排序的方法为sort(List list),作用是对集合中的元素排序

```

public static void main(String[] args) {
    List<Integer> list = new ArrayList<Integer>();
    for (int i = 0; i < 10; i++) {
        list.add((int)(Math.random()*100));
    }
    System.out.println("排序前:" + list);
    Collections.sort(list);
    System.out.println("排序后:" + list);
}

```

4.2 对象排序问题

- 使用Collections方法排序的集合元素必须实现Comparable。
- 如果元素为对象，那么可以通过实现Comparable的compareTo方法为对象添加比较逻辑
- 案例-根据学生分数进行排序
 - 编写实体类Student

```

public class Student implements Comparable{
    private String name;
    private Integer score;

    public String getName() {
        return name;
    }

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

    public Integer getScore() {
        return score;
    }

    public void setScore(Integer score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return "Student{" +
            "name='" + name + '\'' +
            ", score=" + score +
            '}';
    }

    public Student(String name, Integer score) {
        this.name = name;
        this.score = score;
    }

    @Override
    public int compareTo(Object o) {
        Student student = (Student) o;
        /*

```

```

        差值为负则从小到大
        差值为正则从大到小
        */
        return this.score - student.score;
    }
}

```

- 编写测试类

```

List<Student> list = new ArrayList<Student>();
list.add(new Student("zhang3",32));
list.add(new Student("li4",35));
list.add(new Student("wang5",98));
list.add(new Student("zhao6",85));
System.out.println("排序前:" + list);
Collections.sort(list);
System.out.println("排序后:" + list);

```

- 接口回调问题

- 如果需要临时变换比较规则，则采用接口回调方式

```

Collections.sort(list, new Comparator<Student>() {
    @Override
    public int compare(Student o1, Student o2) {
        return o1.getScore() - o2.getScore();
    }
});

```

5.Queue

5.1 队列Queue

- 遵循先进先出的原则，只能从一端添加(offer)元素，从另一端取出(poll)元素



- 常用方法:

E element()	检索，但是不移除此队列的头。
boolean offer(E o)	如果可能，将指定的元素插入此队列。
E peek()	检索，但是不移除此队列的头，如果此队列为空，则返回 null。
E poll()	检索并移除此队列的头，如果此队列为空，则返回 null。
E remove()	检索并移除此队列的头。

- 测试

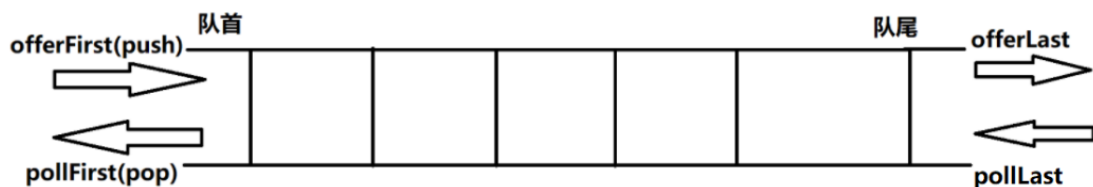
```

public static void main(String[] args) {
    Queue<String> queue = new LinkedList<String>();
    queue.offer("a");
    queue.offer("b");
    queue.offer("c");
    System.out.println("原队列:" + queue);
    System.out.println("返回队首元素:" + queue.peek());
    System.out.println("取出队首元素:" + queue.poll());
    System.out.println("取出队首元素后的队列:" + queue);
    queue.add("d");
    System.out.println(queue);
}

```

5.2 双端队列Deque

- Queue的子接口，定义了双端队列，可以从两端入队和出队
- 如果将双端队列限制为一端存取，则可以实现栈的数据结构。栈遵循先进后出的原则



- 测试

```

public static void main(String[] args) {
    //双端队列
    Deque<String> deque = new LinkedList<String>();
    deque.offerFirst("a");
    deque.offerFirst("b");
    deque.offerFirst("c");
    deque.offerLast("d");
    deque.offerLast("e");
    deque.offerLast("f");
    System.out.println("从队首取出元素:" + deque.pollFirst());
    System.out.println("从队尾取出元素:" + deque.pollLast());
    //限制从一端存取--栈
    Deque<String> stack = new LinkedList<String>();
    stack.push("a");
    stack.push("b");
    stack.push("c");
    System.out.println("拿出元素:" + stack.pop());
}

```

6.Iterator

- 迭代器Iterator，用于遍历集合元素
- 常用方法:
 - boolean hasNext();判断集合是否还有其他元素可以遍历
 - E next();返回迭代的下一个元素

- remove: 删除由next()迭代过的元素

```
//创建迭代器对象
Iterator<String> iterator = stack.iterator();
//判断是否由下一个元素
while(iterator.hasNext()) {
    //获取集合中的元素
    String str = iterator.next();
    System.out.println(str);
    //判断是否为a吗, 如果是a则删除
    if (str == "a") {
        //注意! 一定要用迭代器定义的remove方法, 不能使用集合的remove方法, 否则会抛出异常。
        iterator.remove();
    }
}
System.out.println(stack);
```

7.Map

- Map集合是以(K-V)键值对的形式存储数据, 无序, key可以看为Value的索引, 且key是唯一的。
- 常见实现类:
 - **HashMap**: 底层使用位桶、链表、红黑树实现(红黑树的加入是jdk1.8后, 当链表长度超过阈值(8)时, 使用红黑树), 大大减少了查找时间
 - **TreeMap**: 底层位红黑树实现
 - **LinkedHashMap**: 按插入的顺序排列。HashMap和双向链表合二为一就是LinkedHashMap
 - **Hashtable**: 和HashMap存储原理相似, 但方法都是synchronized, 因此此时线程安全的
- Map接口中的方法:

boolean	containsValue (Object value)	如果此映射为指定值映射一个或多个键, 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)	返回此映射中映射到指定键的值。
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相关联 (可选操作)。
void	putAll (Map<? extends K, ? extends V> t)	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)	如果存在此键的映射关系, 则将其从映射中移除 (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。

7.1 常用方法

- 测试:

```
public static void main(String[] args) {
    Map<Integer,String> maps = new HashMap<Integer,String>();
    //1. put 加入元素(Key,value)
    maps.put(1,"a");
    maps.put(2,"b");
    maps.put(3,"c");
    maps.put(4,"d");
    System.out.println(maps);
    //2. size 集合中k-v队的数量
    System.out.println(maps.size());
    //3. isEmpty 判断是否位空
    System.out.println(maps.isEmpty());
    //4. putAll 添加集合
    Map<Integer,String> maps2 = new HashMap<Integer,String>();
    maps2.put(5,"e");
    maps2.put(4,"f");//key是唯一的, 因此当存入key一致时则覆盖原有的value
    maps.putAll(maps2);
    System.out.println("添加后的集合:" + maps);
    //5. get 根据key返回value
    System.out.println("key为4的value为:" + maps.get(4));
    //6. containsKey containsValue 是否包含key或value
    System.out.println(maps.containsKey(6));
    System.out.println(maps.containsValue("a"));
    //7.keySet 返回所有键组成集合set
    Set<Integer> integers = maps.keySet();
    System.out.println("所有键组成的Set集合:" + integers);
    //8.values 返回所有值组成的集合Collection
    Collection<String> values = maps.values();
    System.out.println("所有值组成的Collection集合:" + values);
    //9.entrySet 返回所有键及所有值组成的集合Set
    /*
        注意! :返回Set集合的泛型为Entry对象,
        */
    Set<Map.Entry<Integer, String>> entries = maps.entrySet();
    System.out.println("返回所有键值对:" + entries);
    //可以使用增强for循环取遍历
    for (Map.Entry<Integer,String> e:entries) {
        System.out.println("key:" + e.getKey() + " value: " +e.getValue());
    }
    //10.clear 清除所有的键值对
    maps.clear();
    System.out.println("清楚后的map: " +maps);
}
```

7.2 HashMap底层原理

- 逻辑图


```

        if (getName() != null ? !getName().equals(person.getName()) :
person.getName() != null) return false;
        return getAge() != null ? getAge().equals(person.getAge()) :
person.getAge() == null;
    }

    @Override
    public int hashCode() {
        int result;
        long temp;
        result = getName() != null ? getName().hashCode() : 0;
        result = 31 * result + (getAge() != null ? getAge().hashCode() : 0);
        temp = Double.doubleToLongBits(getSalary());
        result = 31 * result + (int) (temp ^ (temp >>> 32));
        return result;
    }

```

- Hash冲突：两个不同对象，hashCode的返回值结果一致。

7.3 HashMap优化

- 容量(Capacity): 初始容量为16

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

- 最大容量

```
static final int MAXIMUM_CAPACITY = 1 << 30;
```

- 元素个数

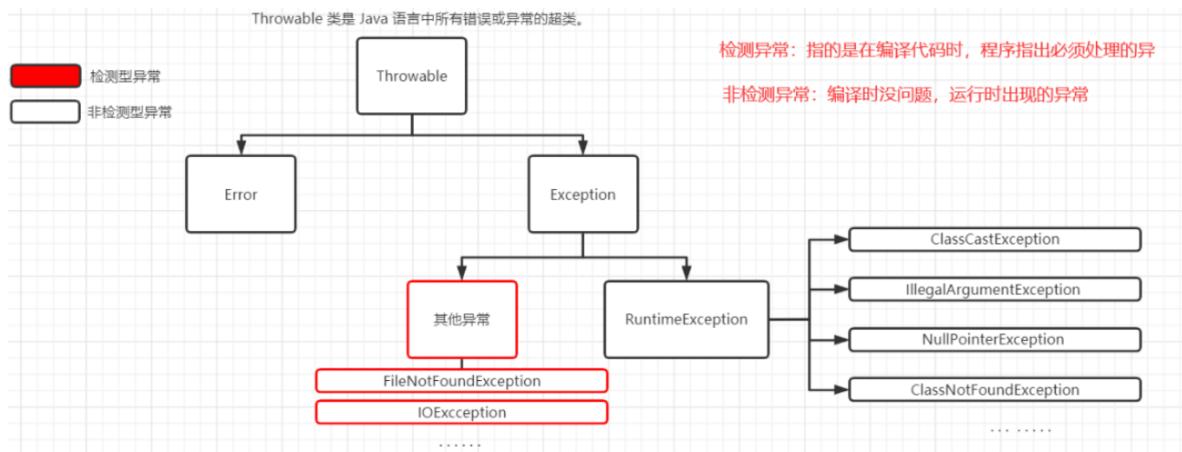
```
size()
```

- 默认加载因子：size/capacity的结果值，
 - 如果大于0.75，则重新散列(rehashing)

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

- 优化：
 - 加载因子较小时，查找性能会提高，但是会浪费桶容量
 - 0.75是相对平衡状态
 - 因此通过构造方法指定合理的容量，减少重新散列，提高性能。

四、异常处理机制



1 异常概述

- 异常的父类为Throwable, 有两个派生子类Error和Exception
 - Error: 指Java运行环境出现的错误, 一般程序不允许出现Error错误, 如: JVM的资源耗尽等
 - Exception: 网络故障、文件损坏、用户输入非法等导致的异常, 其子类RuntimeException尤为重要。
- 异常又可分为检测型异常和非检测型异常
 - 检测型异常: 编译代码时, 程序指出的必须处理的异常, 如: InterruptedException、IOException等。
 - 非检测异常: 运行时才会出现的异常, 如: ArrayIndexOutOfBoundsException、ClassCastException、NullPointerException等。

2 异常处理方式

- 异常处理的目的: 当异常出现, 程序就会终止。交给异常处理代码处理(抛出或抓取), 其他代码正常运行
- 异常处理的方式有两种, 抛出或抓取
 - throws: 用于方法参数后, 对方法中发生异常的代码进行异常抛出

```
public void testTryCatch() throws InterruptedException {
    Thread.sleep(1000);
}
```

- try-catch语句: 用于在异常代码之内。try块对异常代码抓取, catch块对异常代码进行处理

```
public void testTryCatch() {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace(); //打印堆栈信息
    }
}
```

- 若代码有多个异常, 则可以一一处理或直接处理父类异常

```
public void testTryCatch() {
    try {
```

```

        Thread.sleep(1000);
        InputStream is = new FileInputStream("demo.txt");
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
或
public void testTryCatch() {
    try {
        Thread.sleep(1000); // 打断性异常
        InputStream is = new FileInputStream("demo.txt"); // IO异常
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

- finally块：一般跟在catch之后，表示异常处理块的最终出口，无论有没有抛出异常，都会执行finally中的代码
 - 通常用来进行资源释放等工作

```

public void testTryCatch() {
    try {
        Thread.sleep(1000);
        InputStream is = new FileInputStream("demo.txt");
    } catch (Exception e) {
        e.printStackTrace();
        System.out.println("异常出现了, QAQ");
    } finally {
        System.out.println("资源释放...");
    }
}

```

- throw关键字：用于程序员主动抛出异常

```

throw new RuntimeException(); // 抛出新异常

```

3 常见方法

- printStackTrace(): 打印异常事件发生时的堆栈信息
- getMessage(): 打印有关异常事件的信息。
- getCause(): 获取异常出现的原因

4 自定义异常

- 自定义异常：继承Exception

```
public class MyException extends Exception{

    public MyException() {
        System.out.println("自定义异常。。。");
        getMessage();
    }

    @Override
    public String getMessage() {
        return "自定义信息....";
    }
}
```

- 测试:

```
package com.hqyj.text;

public class TestMyException {

    public static void main(String[] args) throws MyException {
        throw new MyException();
    }
}
```

五、IO流

1 File

- java.io.File表示文件(目录)，操作本地的文件和目录。
- File对象只能操作文件的基本信息(名称、大小等)。不能对文件内容进行访问。

1.1 创建File对象

- 相对路劲和绝对路劲
 - 相对路劲：相对于当前书写目录的位置

```
../demo/a.txt//上级目录的a.txt
```

- 绝对路劲:指文件在硬盘上真正存在的路径

```
D:\IdeaProjects\javase
```

- 构造方法:

 image-20220224112258925

- 测试

```

在hqyj下新建text\\demo\\a.txt
//1.注意: 如果使用相对路劲时 java.io默认的相对路劲为module所在的目录
File file = new File("02-javase-oop\\src\\com\\hqyj\\text\\demo\\a.txt");
System.out.println(file.isFile()); //判断是否为文件
System.out.println(file.getPath());
//2.
File parent = new File("02-javase-oop\\src\\com\\hqyj\\text\\demo");
File file2 = new File(parent, "a.txt");
System.out.println(file2.isFile());

```

1.2 常用方法

```

public boolean canRead(); 测试应用程序是否可以读取此抽象路径名表示的文件。
public boolean exists(); 测试此抽象路径名表示的文件或目录是否存在。
public boolean isFile(); 测试此抽象路径名表示的文件是否是一个标准文件。
public long lastModified(); 返回此抽象路径名表示的文件最后一次被修改的时间。
public String getName(); 返回由此抽象路径名表示的文件或目录的名称。
public boolean canWrite(); 测试应用程序是否可以修改此抽象路径名表示的文件。
public boolean isDirectory(); 测试此抽象路径名表示的文件是否是一个目录。
public boolean isHidden(); 测试此抽象路径名指定的文件是否是一个隐藏文件。
public long length(); 返回由此抽象路径名表示的文件的长度。
public String getPath(); 将此抽象路径名转换为一个路径名字符串。

public boolean createNewFile() throws IOException; 当且仅当不存在具有此抽象路径名指定的名称的文件时，原子地创建由此抽象路径名指定的一个新的空文件。
public boolean delete(); 删除此抽象路径名表示的文件或目录。
public boolean mkdir(); 创建此抽象路径名指定的目录。
public boolean mkdirs(); 创建此抽象路径名指定的目录，包括创建必需但不存在的父目录。

```

- 测试:

```

package com.hqyj.text;

import java.io.File;
import java.io.IOException;

public class TestFile {

    public static void main(String[] args) throws IOException {
        File file1 = new File("D:\\IdeaProjects\\javase\\02-javase-
oop\\src\\com\\hqyj\\text\\demo", "a.txt");
        System.out.println("是否存在:" + file1.exists());
        System.out.println("是否能读:" + file1.canRead());
        System.out.println("是否能写:" + file1.canWrite());
        System.out.println("是否为文件:" + file1.isFile());
        System.out.println("路劲:" + file1.getPath());
        System.out.println("文件名:" + file1.getName());
        System.out.println("是否为目录:" + file1.isDirectory());
        System.out.println("是否为隐藏文件:" + file1.isHidden());
        System.out.println("最后一次修改时间为:" + file1.lastModified());
        System.out.println("长度为:" + file1.length());

        File parent = new File("02-javase-oop\\src\\com\\hqyj\\text\\demo");
        File file2 = new File(parent, "demo2");
        file2.mkdir();
    }
}

```

```

        File file3 = new File(parent, "demo3\\c.txt");
        file3.mkdirs();

        File file4 = new File(parent, "d.txt");
        file4.createNewFile();
        //file4.delete();
    }
}

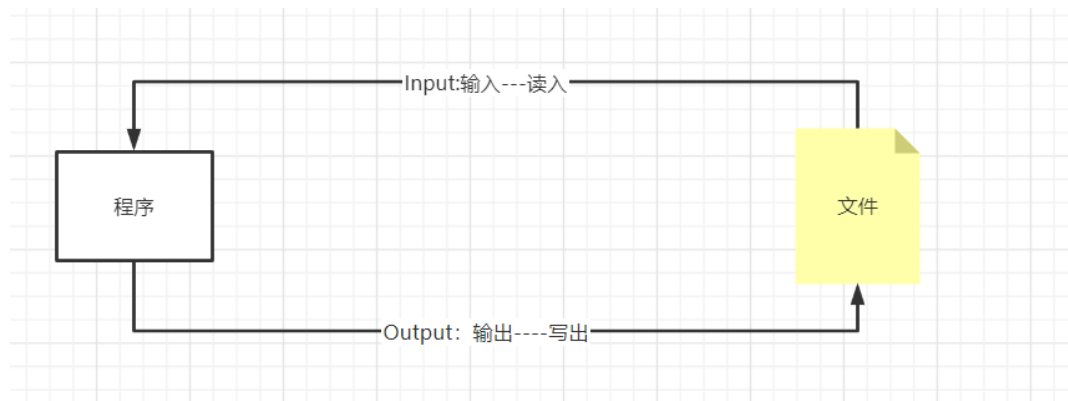
```

- 目录为:

 image-20220224141819752

2 IO流概述

- IO:指输入输出
 - Input: 输入, 从外界到程序, 读取数据
 - Output: 输出, 从程序到外界, 写出数据



- 流的分类:
 - 字节流和字符流
 - 字节流: 传输数据的最小单位是字节
 - 字符流: 传输数据的最小单位是字符
 - 节点流(低级流)和处理流(高级流或过滤流)
 - 节点流: 可以向一个特定的地方读写数据
 - 处理流: 封装了节点流, 通过封装的流的功能调用实现数据读写。
- 流的链接: 一个流对象经过其他流的多次包装, 成为流的链接。

3 字节流

- 字节流有两个抽象类: InputStream和OutputStream
- IS(InputStream)的常用方法

方法摘要	
int available()	返回此输入流方法的下一个调用方可以不受阻塞地从此输入流读取（或跳过）的字节数。
void close()	关闭此输入流并释放与该流关联的所有系统资源。 finally
void mark(int readlimit)	在此输入流中标记当前的位置。
boolean markSupported()	测试此输入流是否支持 mark 和 reset 方法。
abstract int read()	从输入流读取下一个数据字节。
int read(byte[] b)	从输入流中读取一定数量的字节并将其存储在缓冲区数组 b 中。
int read(byte[] b, int off, int len)	将输入流中最多 len 个数据字节读入字节数组。
void reset()	将此流重新定位到对此输入流最后调用 mark 方法时的位置。
long skip(long n)	跳过和放弃此输入流中的 n 个数据字节。

- OS(OutputStream)的常用方法

方法摘要	
void close()	关闭此输出流并释放与此流有关的所有系统资源。
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 个字节从指定的字节数组写入此输出流。
void write(byte[] b, int off, int len)	将指定字节数组中从偏移量 off 开始的 len 个字节写入此输出流。
abstract void write(int b)	将指定的字节写入此输出流。

3.1 FIS和FOS

- FIS: FileInputStream, 文件输入流

FileInputStream(File file)

通过打开一个到实际文件的连接来创建一个 **FileInputStream**，该文件通过文件系统中的 **File** 对象 **file** 指定。

FileInputStream(String name)

通过打开一个到实际文件的连接来创建一个 **FileInputStream**，该文件通过文件系统中的路径名 **name** 指定。

- FOS: FileOutputStream, 文件输出流

FileOutputStream(File file)

创建一个向指定 **File** 对象表示的文件中写入数据的文件输出流。

FileOutputStream(File file, boolean append)

创建一个向指定 **File** 对象表示的文件中写入数据的文件输出流。

FileOutputStream(String name)

创建一个向具有指定名称的文件中写入数据的输出文件流。

FileOutputStream(String name, boolean append)

创建一个向具有指定 **name** 的文件中写入数据的输出文件流。

- 测试:

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

    //写出数据到文件
    //0.创建File对象
    File file = new File("demo\\demo.txt");
    //1.创建文件输出流
```

```

FileOutputStream fos = new FileOutputStream(file,true);
//2. 写出一组字节
fos.write("helloWorld".getBytes());
fos.close();//关闭fos

//读取数据到控制台
FileInputStream fis = new FileInputStream(file);
int d = -1;//当读取到文件末尾时为1
while((d = fis.read())!= -1) {
    System.out.print((char) d + " ");
}
fis.close();//关闭fis
}

```

- 案例使用FIS和FOS实现文件的复制

```

package com.hqyj.text;

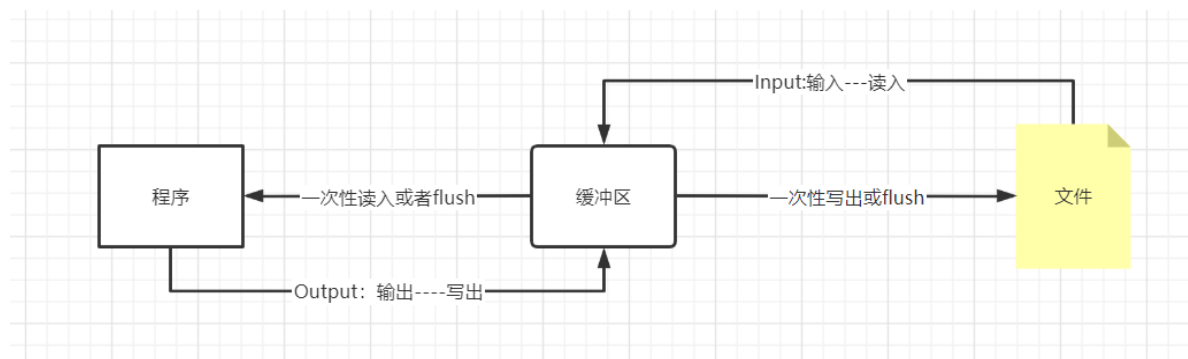
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;

public class FileCopy1 {
    public static void main(String[] args) throws Exception {

        //写出数据到文件
        //0.创建File对象
        File file1 = new File("demo\\demo.txt");
        File file2 = new File("demo\\demo_copy.txt");
        //1.创建文件输出输入流
        FileOutputStream fos = new FileOutputStream(file2);
        FileInputStream fis = new FileInputStream(file1);
        int d = -1;//当读取到文件末尾时为1
        byte[] buf = new byte[1024];
        while((d = fis.read(buf))!= -1) {
            fos.write(buf,0,d);
        }
        fis.close();
    }
}

```

3.2 BIS和BOS



- 缓冲区的优势：减少了读写次数

- flush(): 清空缓冲区, 将缓冲区中的数据全部写出
- BIS: BufferedInputStream 缓冲输入流

BufferedInputStream(InputStream in)

创建 BufferedInputStream 并保存其参数, 即输入流 in, 以便将来使用。

BufferedInputStream(InputStream in, int size)

创建具有指定缓冲区大小的 BufferedInputStream, 并保存其参数, 即输入流 in, 以便将来使用。

- BOS: BufferedOutputStream 缓冲输出流

BufferedOutputStream(OutputStream out)

创建一个新的缓冲输出流, 以将数据写入指定的基础输出流。

BufferedOutputStream(OutputStream out, int size)

创建一个新的缓冲输出流, 以将具有指定缓冲区大小的数据写入指定的基础输出流。

- 测试:

```
package com.hqyj.text;

import java.io.*;

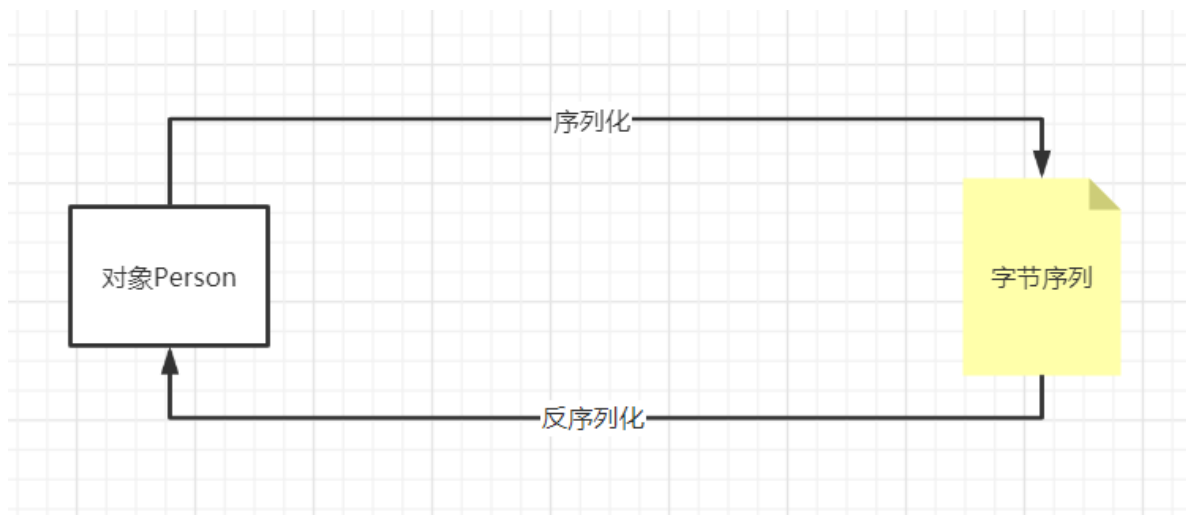
public class TestBISBOS {

    public static void main(String[] args) throws Exception {
        //0.创建File对象
        File file = new File("demo\\demo_bos.txt");

        //1.创建FOS
        FileOutputStream fos = new FileOutputStream(file);
        //2.创建BOS
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        //先写入缓冲区, 再一次性写出到文件
        bos.write("how are you ?".getBytes());
        bos.close();
        //1.创建FIS
        FileInputStream fis = new FileInputStream(file);
        //2.创建BIS
        BufferedInputStream bis = new BufferedInputStream(fis);
        int d = -1;
        while ((d = bis.read()) != -1) {
            System.out.print((char) d + " ");
        }
        bis.close();
    }
}
```

4 对象流

4.1 序列化和反序列化



- 序列化：对象转化为字节序列
- 反序列化：字节序列转化为对象

4.2 OIS和OOS

- OIS: `ObjectInputStream`, 对象输入流

```
ObjectInputStream(InputStream in)
    创建从指定 InputStream 读取的 ObjectInputStream。
```

- OOS: `ObjectOutputStream`, 对象输出流

```
ObjectOutputStream(OutputStream out)
    创建写入指定 OutputStream 的 ObjectOutputStream。
```

- 常用方法

- 序列化

```
void writeObject(Object o);
```

- 反序列化

```
Object readObject();
```

- 对象序列化的注意事项:

- 必须实现`Serializable`接口, 该接口没有方法
- 建议为对象添加类版本号, 避免版本升级后造成的不兼容问题
- `transient`关键字: 修饰属性可以对没必要的属性值忽略, 从而对对象序列化后的字节序列“瘦身”

```
public class Person implements Serializable {

    private static final long serialVersionUID = 4502478679183016996L;
    private String name;
    private transient Integer age;
    private transient double salary;

    public Person(String name, Integer age, double salary) {
```

```

        this.name = name;
        this.age = age;
        salary = salary;
    }

    public String getName() {
        return name;
    }

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

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        salary = salary;
    }
}

```

- 测试:

```

package com.hqyj.text;

import java.io.*;

public class TestBISBOS {

    public static void main(String[] args) throws Exception {
        //0.创建File对象
        File file = new File("demo\\demo_bos.txt");

        //序列化

        //1.创建FOS
        FileOutputStream fos = new FileOutputStream(file);
        //2.创建BOS
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        //先写入缓冲区，再一次性写出到文件
        bos.write("how are you ?".getBytes());
        bos.close();

        //反序列化

        //1.创建FIS
    }
}

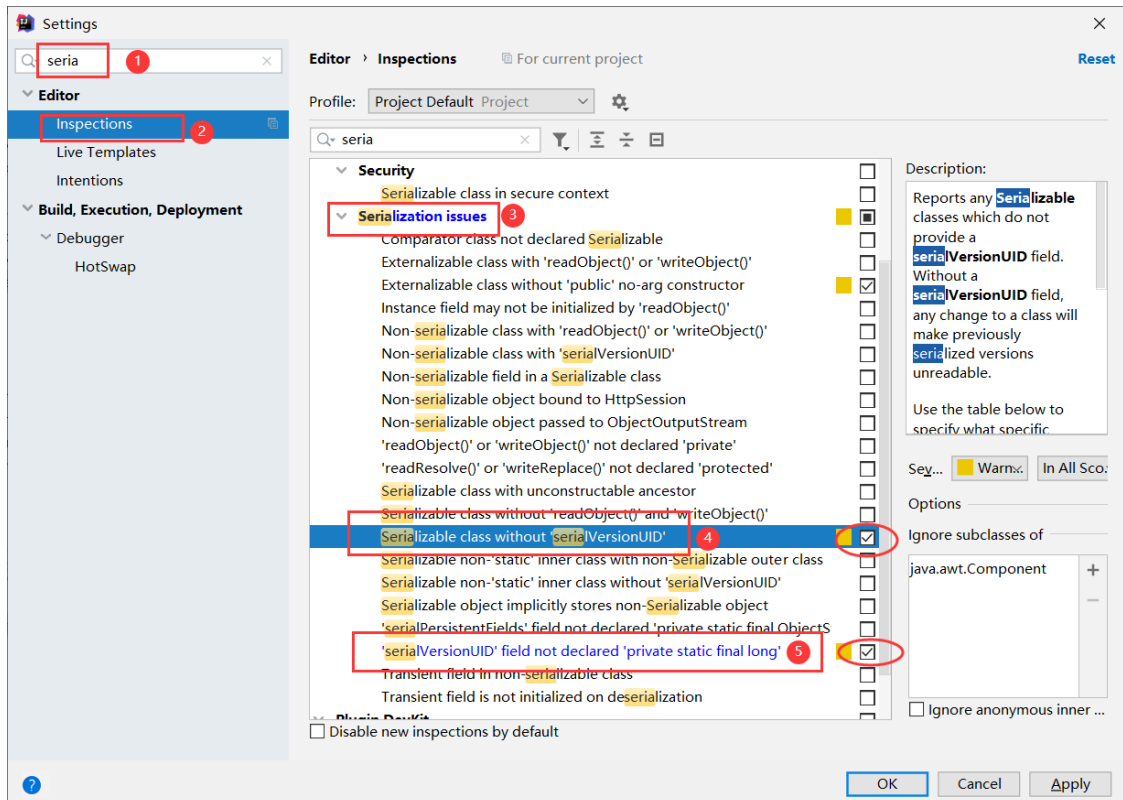
```

```

FileInputStream fis = new FileInputStream(file);
//2.创建BIS
BufferedInputStream bis = new BufferedInputStream(fis);
int d= -1;
while ((d = bis.read()) != -1) {
    system.out.print((char) d + " ");
}
bis.close();
}
}

```

- IDEA添加serialVersionUID



5 字符流

- 字符流有两个抽象类 Reader和Writer
- 字符流的底层是字节流，每次处理一个字符(char)
- 常见字符集：
 - UTF-8: 是针对Unicode的一种可变长度字符编码。
 - GBK: 汉字编码字符集。
- Reader常用方法
- Writer常用方法

5.1 ISR和OSW

- ISR: InputStreamReader, 字符输入流

```
InputStreamReader(InputStream in)
    创建一个使用默认字符集的 InputStreamReader。
InputStreamReader(InputStream in, String charsetName)
    创建使用指定字符集的 InputStreamReader。
```

- OSW: OutputStreamWriter, 字符输出流

```
OutputStreamWriter(OutputStream out)
    创建使用默认字符编码的 OutputStreamWriter。
OutputStreamWriter(OutputStream out, String charsetName)
    创建使用指定字符集的 OutputStreamWriter。
```

- 测试

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

    FileOutputStream fos = new FileOutputStream("demo\\a.txt");
    //1.写
    OutputStreamWriter osw = new OutputStreamWriter(fos,"UTF-8");
    osw.write("大家好! ");
    osw.close();

    //2.读
    FileInputStream fis = new FileInputStream("demo\\a.txt");
    InputStreamReader isr = new InputStreamReader(fis,"UTF-8");
    int c = -1;
    while ((c = isr.read()) != -1) {
        System.out.println((char) c);
    }

    isr.close();
}
```

5.2 PW和BR

- PW: PrintWriter, 输出打印流

```
--构造方法
PrintWriter(File file)
    使用指定文件创建不具有自动行刷新的新 PrintWriter。
PrintWriter(File file, String csn)
    创建具有指定文件和字符集且不帶自动刷行新的新 PrintWriter。
PrintWriter(OutputStream out)
    根据现有的 OutputStream 创建不帶自动行刷新的新 PrintWriter。
PrintWriter(OutputStream out, boolean autoFlush)
    通过现有的 OutputStream 创建新的 PrintWriter。
PrintWriter(String fileName)
    创建具有指定文件名称且不帶自动行刷新的新 PrintWriter。
PrintWriter(String fileName, String csn)
    创建具有指定文件名称和字符集且不帶自动行刷新的新 PrintWriter。
PrintWriter(Writer out)
    创建不帶自动行刷新的新 PrintWriter。
PrintWriter(Writer out, boolean autoFlush)
    创建新 PrintWriter。
```

--常用方法

PW提供了丰富的print方法和println方法的重载方法

- 自动行刷新:

```
public static void main(String[] args) throws Exception {
    FileOutputStream fos = new FileOutputStream("demo\\pw.txt");
    OutputStreamWriter osw = new OutputStreamWriter(fos);
    PrintWriter pw = new PrintWriter(osw,true);
    pw.println("大家好! hello");//当autoFlush为true时会立即写出
    pw.println("bye");
    pw.println("bye");
    pw.close();
}
```

- BR: BufferedReader: 缓冲字符输入流

-- 构造方法

BufferedReader(Reader in)

创建一个使用默认大小输入缓冲区的缓冲字符输入流。

BufferedReader(Reader in, int sz)

创建一个使用指定大小输入缓冲区的缓冲字符输入流。

-- 常用方法

String readLine()

连续读取一行字符串，直到读取到换行符为止，返回的字符串中不包含该换行符

- 测试:

```
FileInputStream fis = new FileInputStream("dmeo\\pw.txt");
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader br = new BufferedReader(isr);
String line = null;
while((line = br.readLine()) != null) {
    System.out.println(line);
}
br.close();
```

六、多线程

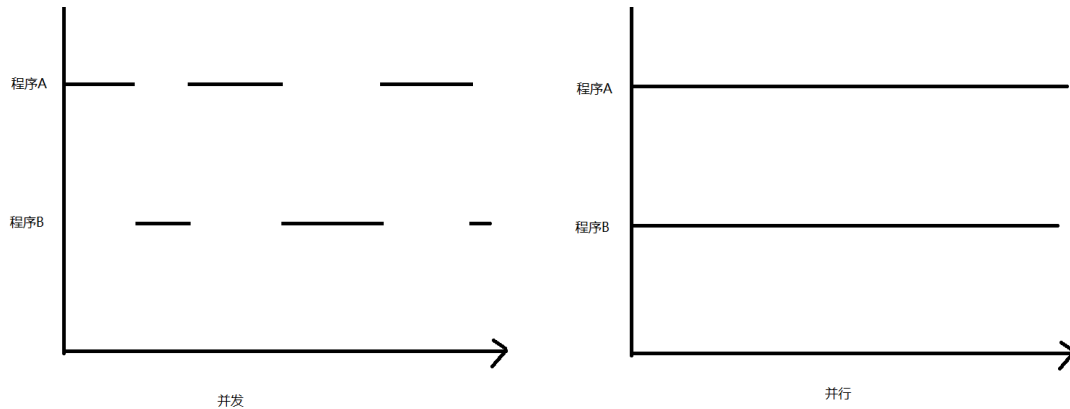
1 线程和进程

1.1 进程和线程概述

- 进程 (Process)：是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是[操作系统](#)结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。
- 线程：**线程**（英语：thread）是[操作系统](#)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。一条线程指的是[进程](#)中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

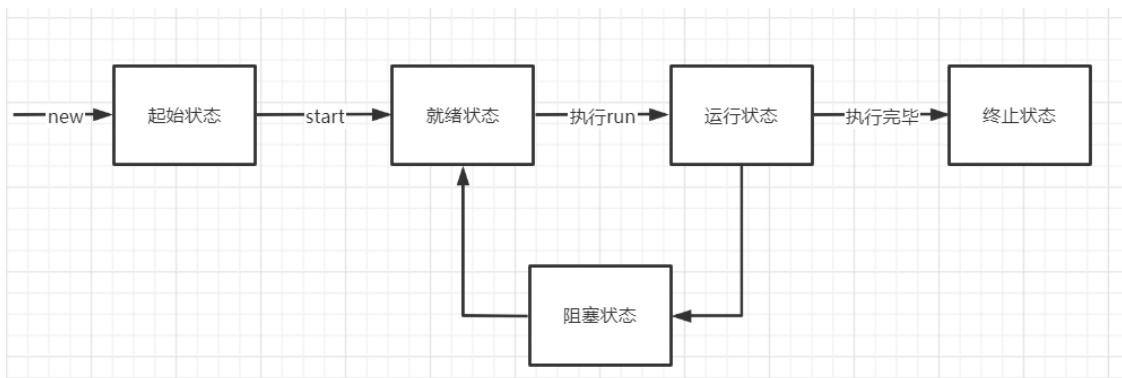
1.2 并发和并行

- 并发:指在同一时刻只能有一条指令执行,但多个进程指令被快速的轮换执行,使得在宏观上具有多个进程同时执行的效果,但在微观上并不是同时执行的,只是把时间分成若干段,使多个进程快速交替的执行。
- 并行:真正意义上的同时发生



2 线程的生命周期

- 线程的5种状态
 - 起始: 当使用new关键字创建线程对象时
 - 就绪: 调用start方法启动线程时, 指能运行但还没有运行的过程, 会等待CPU为其分配时间片。
 - 运行: 执行run方法时
 - 阻塞: 当线程中断、主动出让、等待、睡眠时, 会重新进入就绪状态
 - 结束: 指run方法执行完毕时



3 线程创建

3.1 继承Thread类

- 通过继承Thread类重写run()方法实现

```
public class TestThread1 extends Thread{

    @Override
    public void run() {
        System.out.println(this.getName() + "...running....");
    }

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

```

        new TestThread1().start();
        new TestThread1().start();
        new TestThread1().start();
        new TestThread1().start();
    }
}

```

3.2 实现Runnable接口

- 通过实现Runnable接口重写run()方法

```

package com.hqyj.text;

public class TestThread2 implements Runnable{

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() +
"...running...");
    }

    public static void main(String[] args) {
        new TestThread1().start();
        new TestThread1().start();
        new TestThread1().start();
        new TestThread1().start();
    }
}

```

- 好处是可以实现多个接口
- 也可使用匿名内部类的方式

```

public static void main(String[] args) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName() +
"..running...");
        }
    }).start();
}

```

3.3 继承Callable接口

- 思考：怎么让线程计算0-100的值？
- 实现Callable接口，可以定义返回指定的泛型，依赖FutureTask类

```

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

public class TestThread2 implements Callable<Integer> {

```



```

@Override
public Integer call() throws Exception {
    int sum = 0;
    for (int i = 1; i <= 100; i++) {
        sum += i;
    }
    return sum;
}

public static void main(String[] args) throws Exception {
    FutureTask futureTask = new FutureTask<Integer>(new TestThread2());
    new Thread(futureTask).start();
    System.out.println("结果为:" + futureTask.get());
}
}

```

3.4 线程池的创建

- 工厂类Executors
 - 常用方法:

```

static ExecutorService newCachedThreadPool()
    创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。

static ExecutorService newFixedThreadPool(int nThreads)
    创建一个可重用固定线程集合的线程池，以共享的无界队列方式来运行这些线程。

static ScheduledExecutorService newScheduledThreadPool(int
    corePoolSize)
    创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

static ExecutorService newSingleThreadExecutor()
    创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。

```

- 测试:

```

public static void main(String[] args) {
    ExecutorService threadPool = Executors.newFixedThreadPool(3);
    for (int i = 0; i < 100; i++) {
        threadPool.execute(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + " is
running");
            }
        });
    }
}
}

```

4 线程的基本API

4.1 获取线程信息

- 常用方法

```
static Thread currentThread()
    返回对当前正在执行的线程对象的引用。
long getId()
    返回该线程的标识符。
String getName()
    返回该线程的名称。
-- 优先级
int getPriority() 返回线程的优先级。
    优先级: 可以通过提高线程的优先级来尽可能的让线程获取时间片的几率增加。
    等级: 1-10, 1最小, 10最大
    int MAX_PRIORITY
        线程可以具有的最高优先级。
    int MIN_PRIORITY
        线程可以具有的最低优先级。
    int NORM_PRIORITY
        分配给线程的默认优先级。

void setPriority(int newPriority)
    更改线程的优先级。
Thread.State getState()
    返回该线程的状态。
boolean isAlive()
    测试线程是否处于活动状态。
-- 守护线程
void setDaemon(boolean on)
    将该线程标记为守护线程或用户线程。
boolean isDaemon()
    守护线程: 当进程中只剩下守护线程时, 所有的守护线程强制终止, 如GC
    测试该线程是否为守护线程。

boolean isInterrupted()
    测试线程是否已经中断。
```

- 测试常用方法

```
public static void main(String[] args) {
    Thread t1 = new Thread(new Runnable() {
        @Override
        public void run() {
            System.out.println("当前线程的引用为:" + Thread.currentThread()); // 获取
线程信息
            System.out.println("当前线程的名字为:" +
Thread.currentThread().getName()); // 获取线程名字
            System.out.println("当前线程的标识为:" +
Thread.currentThread().getId()); // 获取线程标识
            System.out.println("当前线程的优先级为:" +
Thread.currentThread().getPriority()); // 获取线程优先级
        }
    });
    t1.start();
    System.out.println("当前线程是否处于活动状态:" + t1.isAlive());
    System.out.println("当前线程是否为守护线程:" + t1.isDaemon());
    System.out.println("当前线程是否处于中断:" + t1.isInterrupted());
}
```

- 测试守护线程

```
package com.hqyj.text;

public class TestDaemon {

    public static void main(String[] args) {

        Thread t_Daemon = new Thread(new Runnable() {
            @Override
            public void run() {
                while(true) {
                    System.out.println("守护线程" +
Thread.currentThread().getName());
                    try {
                        Thread.sleep(500);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        });
        t_Daemon.setDaemon(true); //设置守护线程
        t_Daemon.setName("Daemon_t");
        t_Daemon.start();
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("main结束。观察守护线程是否结束。。。");
    }
}
```

4.2 线程常用方法

- sleep方法

```
static void sleep(long millis)
在指定的毫秒数内让当前正在执行的线程休眠（暂停执行）。
```

```
for (int i = 0; i < 100; i++) {
    System.out.println(i);
    if(i == 50) {
        try {
            Thread.sleep(3000); //线程被打断3s
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- sleep睡眠完成后进去就绪状态

- 需要处理InterruptedException
- yield方法

```
static void yield()
```

暂停当前正在执行的线程对象，并执行其他线程。

- 使用yield()的目的是让相同优先级的线程之间能适当的轮转执行。
但是，实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。
 - yield()从未导致线程转到等待/睡眠/阻塞状态。在大多数情况下，yield()将导致线程从运行状态转到可运行状态，但有可能没有效果。
- join方法

```
void join()
```

等待该线程终止。

```
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(new Runnable() {
        @Override
        public void run() {
            for (int i = 0; i < 500; i++) {
                System.out.println(i + " " +
Thread.currentThread().getName());
            }
        }
    });
    t1.start();

    for (int i = 0; i < 500; i++) {
        t1.join(); //main在执行的时候等待t1执行完毕
        System.out.println(i + " " + Thread.currentThread().getName());
    }
}
```

5 线程同步

- 案例-抢购问题

```
package com.hqyj.edu.thread;

/**
 * 5个人同时抢购，不同的线程同时操作同一资源，数据紊乱
 */
public class RushToBuy implements Runnable {

    private Integer stock = 100; //库存

    private boolean flag = true; //定义标志位，确定当前是否还有库存

    public static void main(String[] args) {
        RushToBuy rushToBuy = new RushToBuy();
```

```

        new Thread(rushToBuy, "A").start();
        new Thread(rushToBuy, "B").start();
        new Thread(rushToBuy, "C").start();
        new Thread(rushToBuy, "D").start();
        new Thread(rushToBuy, "E").start();
    }

    @Override
    public void run() {
        while(flag) {
            //买
            buy();
        }
    }

    private void buy() {
        if (stock <= 0) {
            flag = false;
            return;
        }

        System.out.println(Thread.currentThread().getName() + "库存剩余:" +
(-- stock));
    }
}

```

- 结果出现紊乱
- 线程同步：同一个资源，多个人想使用，解决方法：排队，
- 多个需要同时访问 此对象的线程进入这个对象的等待池 形成队列, 等待前面线程使用完毕，下一个线程再使用
- 每个对象都有把锁，当获取对象时，独占资源，其他线程必须等待，使用结束后才释放
 - 一个线程持有锁会导致其他所有需要此锁的线程挂起
 - 在多线程竞争下，加锁，释放锁会导致比较多的上下文切换和调度延时，引起性能问题
 - 如果一个优先级高的线程等待一个优先级低的线程释放锁，会导致优先级倒置，引起性能问题

5.1 同步方法

- **synchronized** 修饰的方法
- 测试：修改RushToBuy中的代码如下：

```

@Override
public synchronized void run() {
    while(flag) {
        //买
        buy();
    }
}

```

- 可以保证数据的正确性，但当一个线程进入该方法时，该线程就获得了此锁，其他线程必须等待该线程执行完方法释放锁。

- 当这个方法比较庞大时，而要加锁的代码(修改)只有一部分，其他代码(只读)也被锁住，而造成效率问题。
- 考虑使用同步代码块解决问题

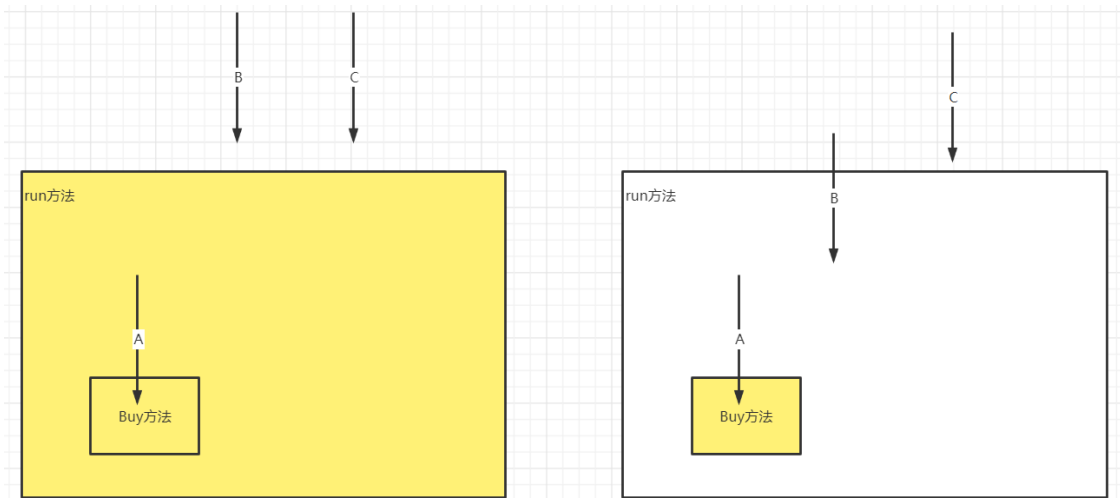
5.2 同步代码块

```
synchronized(obj){}
```

- Obj: 称为同步监视器
 - Obj 可以是任何对象，但是推荐使用共享资源作为同步监视器
 - 同步方法中无需指定同步监视器，因为同步方法的同步监视器就是this，就是这个对象本身
- 测试：修改RushToBuy中的代码如下：

```
@Override
public void run() {
    while(flag) {
        synchronized(this) { //也可以使用RushToBuy.class
            //买
            buy();
        }
    }
}
```

- 同步代码块是指定锁住固定的东西，在方法里其他的内容不受影响。



5.3 死锁

- 死锁是指在一组进程中的各个进程均占有不会释放的资源，但因互相申请被其他进程所站用不会释放的资源而处于的一种永久等待状态。
- 死锁的四个必要条件：
 - 互斥条件(Mutual exclusion): 资源不能被共享，只能由一个进程使用。
 - 请求与保持条件(Hold and wait): 已经得到资源的进程可以再次申请新的资源。
 - 非剥夺条件(No pre-emption): 已经分配的资源不能从相应的进程中被强制地剥夺。
 - 循环等待条件(Circular wait): 系统中若干进程组成环路，该环路中每个进程都在等待相邻进程正占用的资源。
- 解决死锁：如果打破上述任何一个条件，便可让死锁消失。

```
package com.hqyj.11s.thread;
```

```

import java.util.LinkedHashSet;

/**
 * @ClassName DeadLock
 * @Description TODO
 * @Author lijian
 * @Date 2022/5/12 19:15
 *
 * 死锁：
 * 不同的线程相互持有锁，而不愿释放锁，互相等待的状态
 *
 * 解决死锁的方式：
 * 死锁产生的4个必要条件，打破其中一个
 * - 互斥条件(Mutual exclusion)：资源不能被共享，只能由一个进程使用。
 * - 请求与保持条件(Hold and wait)：已经得到资源的进程可以再次申请新的资源。
 * - 非剥夺条件(No pre-emption)：已经分配的资源不能从相应的进程中被强制地剥夺。
 * - 循环等待条件(Circular wait)：系统中若干进程组成环路，该环路中每个进程都在等待相邻进程正占用的资源。
 *
 * 在实际开发中，要避免死锁的出现
 */
public class DeadLock {

    public static String obj1 = "obj1";
    public static String obj2 = "obj2";

    public static void main(String[] args) {
        LockA lockA = new LockA();
        LockB lockB = new LockB();
        new Thread(lockA).start();
        new Thread(lockB).start();
    }
}

class LockA implements Runnable{

    @Override
    public void run() {
        System.out.println("LockA开始执行");
        while (true) {
            synchronized (DeadLock.obj1) {
                System.out.println("LockA锁住了obj1");
                try {
                    Thread.sleep(3000);
                    synchronized (DeadLock.obj2) {
                        Thread.sleep(10000000);
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

class LockB implements Runnable{

```

```

@Override
public void run() {
    System.out.println("LockB开始执行");
    while (true) {
        synchronized (DeadLock.obj2) {
            System.out.println("LockB锁住了obj2");
            try {
                Thread.sleep(3000);
                synchronized (DeadLock.obj1) {
                    Thread.sleep(100000000); //只要拿到就永远锁住
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
}
}

```

5.4 Lock锁

- 通过显示定义同步锁对象（Lock）来实现同步
- ReentrantLock类实现了Lock接口，它拥有与synchronized相同的并发性和内存语义，在实现线程安全的控制中，比较常用的是ReentrantLock,可以显式加锁、释放锁。
- 测试:修改修改RushToBuy中的代码如下：

```

private final ReentrantLock lock = new ReentrantLock(); //定义lock锁

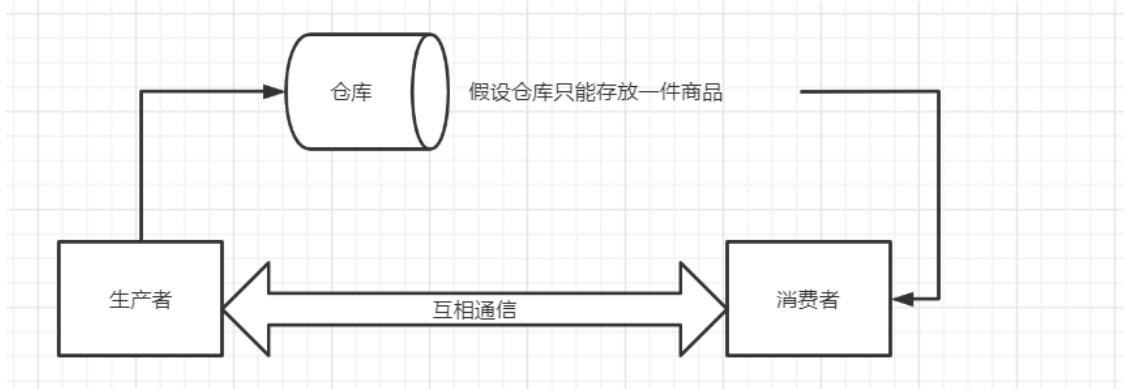
@Override
public void run() {
    while(flag) {
        try {
            //进入加锁状态
            lock.lock();
            buy();
        }
        finally {
            //解锁
            lock.unlock();
        }
    }
}
}

```

- Lock是显式锁（手动开启和关闭锁，别忘记关闭锁）synchronized是隐式锁，出了作用域自动释放
- Lock只有代码块锁，synchronized有代码块锁和方法锁
- 使用Lock锁，JVM将花费较少的时间来调度线程，性能更好。并且具有更好的扩展性（提供更多的子类）

6.线程通信

- 生产者消费者模型



- 对于生产者,没有生产产品之前,要通知消费者等待。而生产了产品之后,又需要马上通知消费者消费
- 对于消费者,在消费之后,要通知生产者已经结束消费,需要生产新的产品 以供消费

6.1 线程通信API

- Object类中提供了线程通信的方法

```
void notify()
    唤醒在此对象监视器上等待的单个线程。
void notifyAll()
    唤醒在此对象监视器上等待的所有线程。
void wait()
    导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。
void wait(long timeout)
    导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量。
```

- 注意：
 - 调用wait和notify线程必须拥有相同的对象锁
 - 该API中的方法必须在同步方法或同步代码块中。
 - 否则会报异常IllegalMonitorStateException

```
Exception in thread "main" java.lang.IllegalMonitorStateException
    at java.lang.Object.wait(Native Method)
    at java.lang.Object.wait(Object.java:502)
    at com.hqyj.api.collections.TestThread.main(TestThread.java:8)
```

6.2 线程通信

- 测试 线程A和线程B交替执行

```
Object lock = new Object(); //创建共享锁

Thread A = new Thread(() -> {
    synchronized (lock) {
        System.out.println("A 1"); //A先启动获取锁
        try {
            lock.wait(); //调用wait方法释放锁
        } catch (InterruptedException e) {}
    }
});
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("A 2");
        System.out.println("A 3");
    }
});

Thread B = new Thread(() -> {
    synchronized (lock) {
        System.out.println("B 1");
        System.out.println("B 2");
        System.out.println("B 3");
        lock.notify();
    }
});

A.start();
B.start();
```

七、反射

1.反射概述

1.1 为什么要用反射？

- java文件在运行时会创建一个Class对象
- Class对象存储在方法区，存储类的所有信息，是类的一面镜子
- 可以通过Class对象获取类的所有信息，反射在手，天下我有

1.2 反射是什么？

- 动态加载和静态加载
 - 动态加载:发生在运行期
 - 静态加载:发生在编译期
- 概念：
 - 《Using Java Reflection》文章: 反射是一个可以获取Java类、属性的工具，因为他是动态加载的。
 - 《A Button is a Bean》中提到:反射是为了能把一个类的属性可视化展示给用户。
 - 简单说: 反射是可以动态获取一个类的所有信息及动态调用类中定义的方法的机制。

1.3 反射的应用

- 反射可以使Java具备动态性语言的特征，很强大！
- 反射是框架的灵魂！
- 反射可以使我们很流畅的阅读源码！
- 反射可以让我变更强，自己编写框架！

2.三种创建方式

2.1 类名.class

```
//1.类名.class
Class<Person> personClass = Person.class;
System.out.println(personClass);
```

2.2 Class.forName()

```
//2.Class.forName
Class personClass2 = Class.forName("com.hqyj.demo.Person");
System.out.println(personClass2);
```

2.3 对象.getClass()

```
//3.getClass()方法
Person p = new Person();
Class personClass3 = p.getClass();
System.out.println(personClass3);
```

2.4 三种创建方式的区别

```
public class CreateReflect02 {

    //创建反射的三种方式
    @Test
    public void test01() {
        Class clz1 = Student.class;
    }

    @Test
    public void test02() throws Exception{
        Class clz2 = Class.forName("com.hqyj.demo.Student");
    }

    @Test
    public void test03() {
        Student student = new Student();
        Class clz3 = student.getClass();
    }

    @Test
    public void test04() throws Exception{
        Class clz1 = Student.class;
        System.out.println("-----");
        Class clz2 = Class.forName("com.hqyj.demo.Student");
        System.out.println("-----");
        Student student = new Student();
        Class clz3 = student.getClass();
    }

    @Test
    public void test05() throws Exception{
        Student student = new Student();
        Class clz3 = student.getClass();
        System.out.println("-----");
    }
}
```

```

        Class clz1 = Student.class;
        System.out.println("-----");
        Class clz2 = Class.forName("com.hqyj.demo.Student");
    }
}

```

- 总结:
 - 类名.class: 类加载前(还没有加载的内存前), 不做类的初始化工作
 - Class.forName: 进内存, 做类的静态初始化工作, 需要抛出异常
 - 对象.getClass(): 进内存对类做静态、非静态初始化;

3.类的加载

- 加载:
 - 将class文件字节码内容加载到内存中, 并将这些静态数据转换成方法区的运行时数据结构, 然后生成一个代表这个类的java.lang.Class对象。
- 链接: 将Java类的二进制代码合并到JVM的运行状态之中的过程。
 - 验证: 确保加载的类信息符合JVM规范, 没有安全方面的问题
 - 准备: 正式为类变量 (static) 分配内存并设置类变量默认初始值的阶段, 这些内存都将在方法区中进行分配。
 - 解析: 虚拟机常量池内的符号引用 (常量名) 替换为直接引用 (地址) 的过程。
- 初始化:
 - 执行类构造器()方法的过程。类构造器()方法是由编译期自动收集类中所有类变量的赋值动作和静态代码块中的语句合并产生的。(类构造器是构造类信息的, 不是构造该类对象的构造器)。
 - 当初始化一个类的时候, 如果发现其父类还没有进行初始化, 则需要先触发其父类的初始化。

4.案例

- 需求: 创建一个类, 在不改变任何代码的情况下, 动态加载生成一个类对象
 - 设计: 创建配置文件, 编写一个类, 修改配置文件后, 动态打印不同对象
 - 类CreateClass01:

```

package com.hqyj.demo03;

import java.io.InputStream;
import java.util.Properties;

public class CreateClass01 {

    public static void main(String[] args) throws Exception {
        //1 加载配置文件
        Properties properties = new Properties();
        ClassLoader classLoader = CreateClass01.class.getClassLoader();
        InputStream inputStream =
        classLoader.getResourceAsStream("demo.properties");
        properties.load(inputStream);

        //2. 获取配置文件内容
    }
}

```

```
String className = properties.getProperty("classname");

Class cls = Class.forName(className);
System.out.println(cls);

}

}
```

- 配置文件 demo.properties

```
classname=com.hqyj.demo.Student
```

八、设计模式

1.设计模式概述及分类

- 设计模式分为三大类：
 - 创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。
 - 结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
 - 行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

2.单例模式

- 思想：只提供一个静态实例

2.1 实现单例

```
public class SingleObject {

    //创建 singleObject 的一个对象
    private static SingleObject instance = new SingleObject();

    //让构造函数为 private，这样该类就不会被实例化
    private SingleObject(){}

    //获取唯一可用的对象
    public static SingleObject getInstance(){
        return instance;
    }

    public void showMessage(){
        System.out.println("Hello world!");
    }
}

测试：
```

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

```
//不合法的构造函数
//编译时错误: 构造函数 SingleObject() 是不可见的
//SingleObject object = new SingleObject();

//获取唯一可用的对象
SingleObject object = SingleObject.getInstance();

//显示消息
object.showMessage();
    }
}
```