基础知识

java 介绍

Java 是由 Sun 公司 1995 年 5 月推出的开源高级编程语言。它通过 JVM 虚拟机可以使程序运行在各种平台上，因此 java 常常用于 web 后端和 Android 的开发。

JDK 与 JRE

JRE 包含 JVM 和 Java 基础类库（运行环境），JDK 包含 JRE 和各种开发工具（编译环境）。jdk1.8 版本较受欢迎，因此通常使用此版本作为 java 的基础环境。

起步工作

Jdk 安装与配置

Java 命令

java 自带命令

jdk 提供了许多编译环境的命令，我们可以使用它们来编译、执行我们写好的程序：

.properties

```
# 编译
javac d:\dir\test.java

# 反编译
javap d:\dir\test.class

# 打jar包(打war包，只需将jar改为war即可)
jar cf test.jar  d:\dir\test1.class d:\dir\test2.class

# 运行java包
java -jar d:\dir\test.jar

# 解压jar
jar xvf d:\dir\test.jar

// 生成.h方法声明文件(c)
javah -jni com.demo.Test
```

jar 打包时指定配置：

Java

```
// 根目录配置文件内容：iniFile（:后留有空格）
Manifest-Version: 1.0
Main-Class: TestMain

// 打包，.表示全部
jar -cvmf test.jar iniFile -C boot .

```

执行系统命令

在 JAVA 程序中可以执行系统的 shell 命令：

Java

```
public static void runShell(String shell) throws IOException {

    // 命令执行
    Process process = Runtime.getRuntime().exec(shell);

    // 获取结果
    InputStream input = process.getInputStream();
    byte[] bytes = new byte[2048];
    StringBuilder builder = new StringBuilder();
    while (true){
        int length = input.read(bytes);
        if (length <= 0) break;
        builder.append(new String(bytes,0,length));
    }
    // 打印结果
    System.out.println(builder.toString());

}
```

常见用途：

- 构建python脚本运行命令，运行python脚本
- 使用命令启动一个程序

语法基础

面向对象、程序入口

java的入口方法main

Java

```
// 类的包路径
package com.good.util.code;

public class TestMain {

    public static void main(String[] args) {
        // 此处运行程序的代码
    }

}
```

java是一个面向对象的编程语言。什么是对象？万物皆对象。

在java中存在类和对象2个概念，即类是对象的抽象，对象是类的实例化。

举个例子："人"是一个类，"张三"则是一个具体的对象。在java中可以如下方式表示人：

Java

```
public class Person {

    // 属性：名字
    private String name;

    // 属性：手机号
    private String phone;

    // 属性：年龄
    private int age;

    // 方法：查询名字
    public String getName() {
        return name;
    }

    // 方法：设置名字
    public void setName(String name) {
        this.name = name;
    }

    // 方法：查询手机号
    public String getPhone() {
        return phone;
    }

    // 方法：设置手机号
```

```java
    public void setPhone(String phone) {
        this.phone = phone;
    }

    // 方法：查询年龄
    public int getAge() {
        return age;
    }

    // 方法：设置年龄
    public void setAge(int age) {
        this.age = age;
    }

}
```

现在我们在main函数中创建人的对象：

Java

```java
public class TestMain {

    public static void main(String[] args) {
        // 创建张三
        Person zhangSan = new Person();
        zhangSan.setName("张三");
        zhangSan.setAge(25);
        zhangSan.setPhone("13099998888");

        // 创建李四
        Person liSi = new Person();
        liSi.setName("李四");

        // 打印张三的手机号
        System.out.println(zhangSan.getPhone());
    }

}
```

控制台打印出张三的手机号

```
13099998888

进程已结束，退出代码为 0
```

控制台输出

通常我们不使用debug时，会使用控制台输出来观察我们的程序运行到什么位置，一般的控制台输出语法如下：

Java

```java
System.out.println("hello java");
```

彩色控制台输出

Java

```java
/*
  字体颜色：0 默认、30黑色、31红色、32绿色、33黄色、34蓝色、35紫色、36青色、37灰色
  背景颜色：1 默认、40黑色、41红色、42绿色、43黄色、44蓝色、45紫色、46青色、47灰色
  样式数值：1加粗、2默认、3斜体、4下划线、(90-97)各种颜色字体高亮
*/
System.out.println("\33[字体颜色;背景颜色;样式值m文本内容\33[m");
```

重写输出语句：定义两个int变量并打印，在打印前插入一个任意方法，使程序打印出新的值：

Java

```java
import java.io.PrintStream;

public class Demo {
    public static void main(String[] args) {
        int a = 10;
        int b = 10;
        // 需要在method方法被调用之后，仅打印出 a = 100,b = 200,请写出method方法的代码
        method2(a,b);
        System.out.println("a = " + a);
```

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

    public static void method1(int a , int b){
        System.out.println("a = " + 100);
        System.out.println("b = " + 200);
        //终止虚拟机，退出Java程序
        System.exit(0);
    }

    public static void method2(int a , int b){
        //方法二：重写输出语句
        PrintStream ps = new PrintStream(System.out){
            @Override
            public void println(String str) {
                if(str == null){
                    throw new NullPointerException();
                }
                if(str.startsWith("a")){
                    super.println(str + 0);
                }
                if(str.startsWith("b")){
                    super.println("b=" + 200);
                }
            }
        };
        System.setOut(ps);

    }
}
```

文件操作

文件生成

如果服务器上存在指定文件就不需要生成了，有时我们导出数据时需要生成不存在的文件，简单的一种文件生成如下：

Java

```
public void createFile(){
    String fileName = getServerPath()+"123.txt"; //这里直接在服务器生成
    File file = new File(excelPath+downloadName);
    file.createNewFile();
}
```

由于这里只举例，因此只生成一个空文件。

文件读取

将文件内容读取到字节数组：

Java

```
public byte[] getDataByFile(File file){
     FileInputStream input = new FileInputStream(file);
     byte[] bytes = new byte[input.available()];
     input.read(bytes);
     input.close();
     return bytes;
}
```

将文件内容直接读为文本：

Java

```
InputStream input = new FileInputStream(file);
int c;
ByteArrayOutputStream output = new ByteArrayOutputStream();
while((c = input.read ()) != -1){
  output.write(c);
}
String info = output .toString();
```

使用 commons 工具类写文件

引入依赖

XML

```
<dependency>
```

```
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.6</version>
</dependency>
```

写文件

```java
File file = new File("C:\\test\\a.json");
FileUtils.writeStringToFile(file,builder.toString(),"UTF-8");
```

使用 fastjson 读文件

使用 fastJson，引入 maven 依赖：

```xml
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.76</version>
</dependency>
```

读取文件：

```java
public static ArrayList readJsonFile(String path) throws IOException {
    ObjectMapper mapper = new ObjectMapper();
    return mapper.readValue(new File(path),ArrayList.class);
}
```

注意：要根据数据内容格式来选取 map 或 list。

静态代码

静态成员变量、静态代码块、静态方法会在类或子类加载过程中首先执行。（程序启动后没有自动执行是因为未触发类加载）

静态成员：会被所有类实例共享。

```java
public class Demo{
  // 静态常量
  private final static String value1 = 1;

  // 静态变量
  private static String value2 = 2;
}
```

静态代码块：在类加载时触发一些逻辑

```java
public class Demo{

    private static String value = 1;

    static{
       value = 2;
    }
}
```

子类进行类加载时，也会触发父类静态代码块的执行。

原生方法 Native

java 可以通过 native 使用本地非 java 方法，主要步骤如下：

- 使用 System.loadLibrary()将包含本地方法实现的动态文件加载进内存。
- 调用本地方法时，JVM 在加载的动态文件中定位并链接该方法然后执行。

加载本地方法文件

```java
static{
    System.loadLibrary("dll文件名");
```

```
}
```

调用本地方法

```java
public class ComNative{

    // 此方法在类中起始位置
    private static native void registerNatives();

    static {
        registerNatives();
    }

    // 调用本地方法run
    public static native void run();

}
```

Java 对象

Enum 枚举

静态常量类:

```java
public class NumEnum {
     public static final String AAA = "AAA";
     public static final Integer BBB = "BBB";
     public static final Integer CCC = "CCC";
}
```

使用 enum 关键字定义枚举类，其成员可以理解为枚举类的多个对象：

```java
// 此类只存在3个对象：USER1,USER2,USER3（以此可以想到使用枚举实现单例模式）
public enum NumEnum {
    USER1,USER2,USER3;
}
```

枚举参数值：

```java
 // 枚举在
 public enum NumEnum {
     // 枚举值
     USER1(1,"张三"),
     USER2(2,"李四");

     // 属性值
     Integer id;
     String name;

     // 构造方法
     NumEnum(Integer id, String name) {
         this.id = id;
         this.name = name;
     }

     public Integer getId(){
         return id;
     }

     public String getName() {
         return name;
     }

 }
```

遍历枚举值

```java
```

```java
for (NumEnum item : NumEnum.values()) {
    System.out.println(item.id);
    System.out.println(item.name);
}
```

Exception 异常

java 中一般使用 try 捕获并由 catch 进行异常处理：

Java

```java
// try可以捕获异常，允许多层嵌套
try{
    // 可能会出现异常的代码
} catch(Exception e){
    // 处理异常，也可以继续抛出
}finally{
    // 此处代码不管是否异常都会执行的
}

// java1.7开始，try(初始化流){} 会自动关闭流
```

如果我们不想处理异常，可以使用 throw/throws 关键字抛出：

Java

```java
// throw 直接抛出异常对象
throw new RuntimeException("运行时异常"); // 运行时异常可以不处理

// 使用throws 抛出异常
public static void run() throws Exception {
    throw new Exception("需处理异常");
}
```

Throwable 、Exception、Error

Java

```java
// Throwable 是Exception和Error的父类，它们都需要抛出或处理
Throwable item = new Throwable("错误、异常");
Error error = new Error("java错误");
Exception exception = new Exception("java异常");
```

POJO 与 JavaBean

POJO

pojo 是简单的 Java 对象，即不实现接口、不继承类、不添加注解的简单类。

JavaBean

javabean 是可序列化的 pojo，具有 getter、setter 方法，不过继承了 Serializable 接口的 Javabean 才是真正意义上的 JavaBean。因为 bean 表示一个可重用对象。

DTO： 数据传输对象

其实体类一般以【业务领域名称 DTO】命名，仅在接受请求参数时使用。

Java

```java
@RestController
public class UserController {

    @PutMapping("user")
    public void updateUser(UserDTO userDTO){ }

}
```

可以继承、组合其他 DTO、VO、BO 对象。

PO： 数据对象(do、po、entity)

实体类属性必须和数据库字段保持一致，其实体类一般以【表名 PO】命名，用于接收查询结果，即 dao 层返回类型。

Java

```java
@Service
public class UserServiceImpl {

    public UserBO getUserInfo(){
```

```
        UserPO userPo = userDao.selectUser(1); // PO
        return userBo;
    }

}
```

VO： 展示对象

其实体类一般以【网页名VO】命名，响应时使用，即封装返回的数据。

Java

```java
@RestController
public class UserController {

    @GetMapping("user")
    public UserVO selectUser(){
        return new UserVO();
    }

}
```

BO： 业务对象

service、manager、dao层使用，一般为业务层返回类型。业务层调用多个dao层接口，会产生多个do对象，此时使用bo对它们封装，最终返回。

Java

```java
@Service
public class UserServiceImpl {

    public UserBO getUserInfo(){
        UserBO userBo = new UserBO();
        UserPO userPo = userDao.selectUser(1);
        userBo.setUserDO(userPo);
        return userBo;
    }

}
```

Query： 查询对象

查询时的sql参数封装，dao层入参

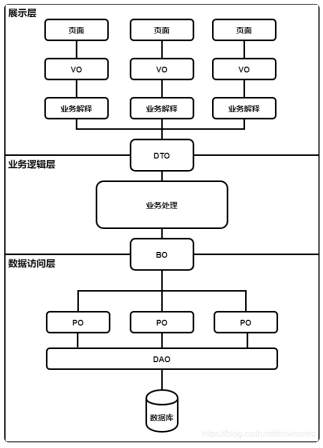
Java

```java
@Mapper
public class UserMapper {

    void selectUser(UserQuery);

}
```

使用场景如图：



序列化-反序列化

// 暂无

常用方法

数字交换

现存在数字变量a,b交换ab的值：

Java

```
// 方法一：中间量
int c = a;
a = b;
b = c;

// 方法二：计算法
a = a + b; // (a+b)
b = a - b; // (a+b)-b = a
a = a - b; // (a+b)-a = b

// 方法三：异或运算（1位二进制加法）,效率高于方法二 设a=2，b=3
a = a ^ b; // 0001 = 0010 ^ 0011
b = a ^ b; // 0010 = 0001 ^ 0011
a = a ^ b; // 0011 = 0001 ^ 0010
// 规律：两个数字异或的结果，在异或其中一个数字时，会得到另一个数字的值
```

日期方法

获取月末日期

Java

```
/**
 * 使用查日历的方式，获取月最后一天
 * @param year 年份
 * @param month 月份
 * @return 返回月份最后一天日期
 */
public static String  getMonthLastDay(int year,int month){
    Calendar calendar = Calendar.getInstance();
    calendar.set(Calendar.YEAR, year);
    calendar.set(Calendar.MONTH, month - 1);
    // 如果不将日期设置为1日，那么当系统当前日期为31日、查询的小于31天的月份时会出现问题,因此可以设置为1-28之间任意数字
    calendar.set(Calendar.DAY_OF_MONTH, 1);
    calendar.set(Calendar.DATE, calendar.getActualMaximum(Calendar.DATE));
    DateFormat format = new SimpleDateFormat("yyyyMMdd");
    return format.format(calendar.getTime());
}
```

获取近几个月中第一个月的日期

Java

```
/**
 * 获取前几个月的首月日期,1则代表本月
 * @param year 年份
 * @param month 月份
 * @param monthNum 月的个数
 * @return 年-月
 */
public static String getAgoMonthsFristMonthDate(String year,String month,int monthNum){
    int y = Integer.parseInt(year);
    int m = Integer.parseInt(month);
    y = y - monthNum / 12;
    monthNum = monthNum % 12;
    m = m - monthNum + 1;
    if (m <= 0){
        m = m + 12;
        y--;
    }
    return y + "-" + (m < 10 ? "0" + m : m);
}
```

获取当月序时进度

Java

```java
private static String getChronologicalProgress(String year,String month) {
    SimpleDateFormat format=new SimpleDateFormat("yyyyMM");
    Date date = new Date();
    try {
        date = format.parse(year+month);
    } catch (ParseException e) {
        e.printStackTrace();
    }
    Calendar calendar = Calendar.getInstance();
    calendar.setTime(date);
    // 查询目标月最后一天日期
    calendar.set(Calendar.DAY_OF_MONTH,calendar.getActualMaximum(Calendar.DAY_OF_MONTH));
    Date monthEndDate = calendar.getTime();
    // 一年起始日期
    calendar.set(Calendar.DAY_OF_YEAR,1);
    Date yearStartDare = calendar.getTime();
    // 一年的天数
    int dayByYear = calendar.getActualMaximum(Calendar.DAY_OF_YEAR);
    long distanceDay = (monthEndDate.getTime() - yearStartDare.getTime()) /(1000 * 60 * 60 * 24)+1;
    return ((double) distanceDay/dayByYear * 100)+"%";
}
```

获取两个日期相隔天数

```java
public static int getDays(Date date1, Date date2){
    return (date2.getTime() - date1.getTime()) / (1000 * 24 * 3600);
}
```

格式化方法

小于 10 的整数补 0

```java
// 方法1
month = "" + month  / 10 + month  % 10
// 方法2
month = month.length == 1? "0" + month : month
```

保留两位小数

如果将结果转换为 double，末尾为 0 将会被丢弃。

```java
// 方法一：（"#.00"会导致0时为.00 ）
String res = new DecimalFormat("0.00").format(value);   // 末尾留0
String res = new DecimalFormat("#.##").format(value);   // 末尾为0将舍弃，变成1位

// 方法2：
String res = String.format("%.2f", value);              // 不能使用整数
```

日期格式化

```java
SimpleDateFormat format=new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

// Date转String
String dateStr = format.format(new Date());

// String转Date
Date date = format.parse("2021-10-10 23:40:34");
```

条件语句

字符串比较

```java
// 如果item可能为null，则方式2就可能出现空指针异常（因为null没有equals方法）
boolean equal = "文本".equals(item);
boolean equal = item.equals("文本");
```

字符串包含

```java
```

```java
boolean contains = "12345".contains("23");
```

浮点型比较

```java
double d1 = 1.0;
double d2 = 1.0;

// d1 == d2 ,true

Double do1 = 1.0;
Double do2 = 1.0;

// do1 == do2 ,false : Double在 == 和 != 比较时不会自动拆箱
```

正则表达式

正则表达式用于快速文本处理，并非java专属。

```java
// 文本是否与表达式一致
boolean match = Pattern.compile("表达式").matcher("文本").matches();

// 符合表达式的文本是否存在
boolean isExists = Pattern.matches("表达式", "文本");
boolean isExists = "文本".matches("表达式");

// 获取匹配结果
String match = Pattern.compile("表达式").matcher("文本").group(0);
```

语法优化

过多的if有时会让代码变得冗长且难以维护，我们可以通过对其简化来使其变得简洁而优雅。

现在有如下代码：

```java
public static void consumer(String value){
    if("100".equals(value)){
        System.out.println("张三");
    } else if("200".equals(value)){
        System.out.println("李四");
    } else if("300".equals(value)){
        System.out.println("王五");
    } else if("400".equals(value)) {
        System.out.println("赵六");
    }
}
```

switch 优化 IF

语句少的合适，但每个if有大量语句依然难以阅读：

```java
public static void consumer(String value){
    switch (value){
        case "100": System.out.println("张三"); break;
        case "200": System.out.println("李四"); break;
        case "300": System.out.println("王五"); break;
        case "400": System.out.println("赵六"); break;
    }
}
```

Map 优化 IF

简单优化：

```java
public static void consumer(String value){
    Map<String,Object> map = new HashMap<>();
    map.put("100","张三");
    map.put("200","李四");
    map.put("300","王五");
    map.put("400","赵六");
    System.out.println(map.get(value));
}
```

使用函数式接口优化方法：

```java
public static void consumer(String value){

    Function<String,String> function1 = (v) -> "张三";
    Function<String,String> function2 = (v) -> "李四";
    Function<String,String> function3 = (v) -> "王五";
    Function<String,String> function4 = (v) -> "赵六";

    Map<String,Function<String,String>> map = new HashMap<>();
    map.put("100",function1);
    map.put("200",function2);
    map.put("300",function3);
    map.put("400",function4);
    System.out.println(map.get(value).apply("0"));
}
```