****

实践教学

****

兰州理工大学

软件学院

2016 年秋季学期

生产实习报告

专业班级：软件工程 2 班

姓　　名：毛烨辉

学　　号：13270212

指导教师：张秋余

兰州理工大学生产实习报告

一、 JDK 主要包括内容

1) Java 虚拟机：负责解析和执行 Java 程序。Java 虚拟机可运行在各种平台上
2) JDK 类库(API)： 提供最基础的 Java 类及各种实用类。java。lang， java。io， java。util， javax。swing 和 java。sql 包中的类都位于 JDK 类库中。
3) 开发工具： 这些开发工具都是可执行程序，主要包括：
   - javac。exe 编译工具；
   - javadoc。exe 生成 JavaDoc 文档的工具
   - jar。exe 打包工具

二、 java 的特点:

1) 面向对象:万物皆对象
2) 简单性:(相对应 C 和 C++来讲)java 不需要程序员操作内存
3) 跨平台:一次编译，到处运行 (JVM)
4) 多线程:其他语言的多线程一般是系统级别的，java 的多线程是语言级别的
5) 动态编译:(相对应 C 或者 C++来说)，在一个项目中，如果需要修改一个 java 文件的话，那么就修改这一个文件就行了，其他的 java 文件不需要动。

三、 垃圾回收机制(gc)

java 不需要程序员释放内存，就是因为 java 中有 gc

垃圾:没有任何作用的/不再使用的内存空间

对象创建，需要占用内存空间，在一个程序运行过程中要创建无数个对象，如果对象永久占用内存，那么内存很快会消费光，导致后续对象无法创建或者出现内存不足的错误。因此必须采取一定措施及时回收那些无用对象的内存，这样保证内存可以被重复利用。

C++等程序由程序员显示进行内存释放，这样有可能：

1) 忘记内存释放，导致没有意义的内存占用；
2) 释放核心类库占用内存，倒致系统崩溃；

四、 Java 中垃圾回收处理特点：

1) 由虚拟机通过垃圾回收器线程自动完成；
2) 只有当对象不再被使用，它的内存才有可能被回收；如果虚拟机认为系统不需要额外的内存，即便对象不再使用，内存也不会回收；
3) 程序无法显示迫使垃圾回收器立即执行垃圾回收，可以通过 java。lang。System。gc()或 java。lang。Runtime。gc()建议虚拟机回收对象
4) 垃圾回收器线程在释放无用对象占用内存之前会先行调用该对象的 finalize()方法

2016 年 7 月 26 日 于上海杰普

一、 基本类型：

Java 语言把数据类型分为**基本类型**和**引用类型**

- 基本类型:float double byte short int long char boolean
- 引用类型:类类型 接口类型 数组类型

二、 理解对象

面向对象的开发方法把软件系统看成各种对象的集合，对象就是最小的子系统，一组相关的对象能够组合成更复杂的子系统。面向对象的开发方法将软件系统看成各种对象的集合，接近人的自然思维方式。

三、 java 中变量的介绍

1) 成员变量：在类中声明，它的作用域是整个类；成员变量又叫做属性/实例变量
2) 局部变量：在一个方法的内部或方法的一个代码块的内部声明。如果在一个方法内部声明，它的作用域是整个方法；如果在一个方法的某个代码块的内部声明，它的作用域是这个代码块。代码块是指位于一对大括号"{}"以内的代码。
3) 方法参数：方法或者构造方法的参数，它的作用域是整个方法或者构造方法。
4) 异常处理参数：和方法参数很相似，差别在于前者是传递参数给异常处理代码块，而后者是传递参数给方法或者构造方法。异常处理参数是指 catch(Exception e)语句中的异常参数"e"，它的作用域是紧跟着 catch(Exception e)语句后的代码块。

四、 类型转换

1) 隐式转换:自动转换

基本类型:精度小可以自动转换为精度大的

```
byte b = 1;
int a = b;
```

引用类型:子类类类型可以自动转换为父类类型

```
Student s = new Student();
Object o = s;
```

2) 显式转换:强制类型转换

基本类型:精度大的可以强制类型转换为精度小的，但是可能损失精度

```
int a = 1;
byte b = (byte)a;
```

引用类型:父类类型可以强制类型转换转换为子类类型，但是可能出现类型转换错误

```
Object 类 是 Student 类的父类
//这个是正确的
Object o = new Student();
Student s = (Student)o;
//这个会报错
Object o1 = new Object();
Student s1 = (Student) o1;
```

2016 年 7 月 29 日 于上海杰普

一、 **Array**

1. 数组是指一组数据的集合，数组中的每个数据称为元素。在 Java 中，数组也是 Java 对象。数组中的元素可以是任意类型(包括基本类型和引用类)，但同一个数组里只能存放类型相同的元素。创建数组大致包括如下步骤：
2. 声明一个数组类型的引用变量，简称为数组变量；
3. 用 new 语句构造数组的实例。new 语句为数组分配内存，并且为数组中的每个元素赋予默认值；
4. 初始化，即为数组的每个元素设置合适的初始值。

二、 **声明数组：**

1) 一个存放同一类型数据的集合
   a) 即可以是基本类型，也可以是对象类型；
   b) 数组中的每个数据为元素；
2) 数组是一个对象，成员是数组长度和数组中的元素;
3) 声明了一个数组变量并不是创建了一个对象;

三、 **初始化数组对象**

数组中的每个元素都有一个索引，或者称为下标。数组中的第一个元素的索引为 0，第二个元素的索引为 1，依次类推。

通过索引可以访问数组中的元素或者给数组中元素内容赋值。

1) 声明、创建、初始化分开：

```
int[] iArray;
iArray = new int[2];
iArray[0] = 0;
iArray[1] = 1;
```

2) 声明、创建的同时并初始化数组：

```
int[] iArray = {0, 1};
int[] iArray = new int[]{0, 1};
Student sArray[] = new Student[] { new Student(“George”, “Male”, 20), new Student()};
Student[] stArray = { new Student(), new Student()} ;
```

四、 **多维数组：**

```
String[][] room = new String[2][];
room[0] = new Stirng[]{"Mike","Jane","Duke",null};
room[1] = new String[]{"Mary",null,"kevin"};
```

等价于：

```
String[][] room = { {"Mike","Jane","Duke",null},
                    {"Mary",null,"kevin"},
                    {null,"Tom","Jerry",null,"Rose"}
                  };
```

2016 年 8 月 2 日 于上海杰普

一、 类中方法的定义

修饰符 返回类型 方法名（参数列表）异常抛出类型{...}

1) 必须有返回值，如果方法没有返回值，必须用 void 申明返回类型。
2) 构造器没有返回类型，构造器加上了返回类型就变成了一个普通方法的声明了。
3) 方法的修饰符可以同时有多个。

二、 参数传递分为两种：

1) 值传递:对于基本数据类型，参数通过值传递。(把实参的值复制一份再传给形参)
2) 引用传递:对于引用类型，参数通过引用(对象的引用)传递。(把实参引用中地址值复制一份再传给形参

三、 this 关键字

在方法调用、参数传递过程中，极有可能出现参数名称与实例变量名同时的情况。在一个方法内，可以定义和成员变量同名的局部变量或参数，此时成员变量被屏蔽。

四、 封装

1.对属性的封装

首先属性可以被一下修饰符修饰:

public protected "default" private(这四种修饰符可以修饰成员变量,也可以修饰方法)

```
public String name;
protected String name;
private String name;
default String name;
```

2.方法的封装:(指的是用方法来封装代码)

站在使用的者的角度去考虑，用户只关心方法的使用,不管里面到底怎么实现的细节.

从编程的角度去看的话:为了代码的重用.

五、 方法重载(overload)

有时侯，类的同一种功能有多种实现方式，换句话说，有很多相同名称的方法，参数不同。这给用户对这种功能的调用使用提供了很大的灵活性。

```
public void print(int i){
    System.out.println("i = "+i);
}
public void print(String s){
    System.out.println("s = "+s);
}
```

对于类的方法(包括从父类中继承的方法), 如果有两个方法的方法名相同，但参数不一致，那么可以说，一个方法是另一个方法的重载方法。这种现象叫重载。

2016 年 8 月 5 日 于上海杰普

内部类分为：成员内部类 静态内部类 局部内部类 匿名内部类

一、 静态内部类：(相对应类中的一个静态变量)

1) 静态内部类中访问不到外部类的非静态属性或者方法
2) 静态内部类的对象不需要依赖于外部类的对象

内部类 变量名字 = new 内部类();

```
public class A {
    public static class B{
    private int v;
    public void say(){
        System.out.println("hello");
    }
    }
    public static void main(String[] args){
        B b = new B();
    }
}
```

二、 成员内部类：(相当于类中的一个成员变量)

1) 成员内部类中不能有 static 的声明属性或者方法
2) 成员内部类可以由 public protected default private 修饰
3) 成员内部类是依赖于外部类的对象而存在的
4) 外部类.内部类 var = new 外部类().内部类();
5) Outer.InnerTool tool = new Outer().new InnerTool();

三、 局部内部类：(相当于一个方法中的局部变量)

1) 局部内部类不能用 public private 等修饰符修饰
2) 写在方法当中,而且只能在方法当中使用
3) 可以访问外层类的普通成员变量和静态成员变量以及普通方法和静态方法,也可以访问该内部类所在方法当中的局部变量,但是这个局部变量必须是 final 修饰；

四、 匿名内部类：(和局部内部类很相似)

1) 匿名内部类也是用的最多的内部类
2) 可以写成成员变量的形式,也可以写在方法当中,一般写在方法当中较多
3) 匿名内部类里可以访问外部类的普通属性和方法,已经静态属性和方法,如果要访问这个内部类所在方法中的局部变量,那么要求这个局部变量必须是 final 修饰的
4) 匿名内部类里面没有构造函数,因为这个类没有名字,所以在其他地方不能用

2016 年 8 月 9 日 于上海杰普

一、 异常的基本概念

1. 异常产生的条件
   a) 整数相除运算中，分母为 0;
   b) 通过一个没有指向任何具体对象的引用去访问对象的方法；
   c) 使用数组长度作为下标访问数组元素；
   d) 将一个引用强制转化成不相干的对象；
2. 异常会改变正常程序流程;异常产生后，正常的程序流程被打破了，要么程序中止，要么程序被转向异常处理的语句；
3. 当一个异常的事件发生后，该异常被虚拟机封装形成异常对象抛出。
4. 用来负责处理异常的代码被称为异常处理器
5. 通过异常处理器来捕获异常

二、 try...catch 语句

在 Java 语言中，用 try...catch 语句来捕获处理异常。格式如下：

```
try {
    可能会出现异常情况的代码；
} catch(异常类型  异常参数) {
    异常处理代码
} catch(异常类型  异常参数) {
    异常处理代码
}
```

三、 finally 语句:

任何情况下都必须执行的代码

四、 异常调用栈

异常处理时所经过的一系列方法调用过程被称为异常调用栈。

1) 异常的传播

哪个调用，哪个处理；

   a) 异常情况发生后，发生异常所在的方法可以处理；
   b) 异常所在的方法内部没有处理，该异常将被抛给该方法调用者，调用者可以处理；
   c) 如调用者没有处理，异常将被继续抛出；如一直没有对异常处理，异常将被抛至虚拟机；

2) 如果异常没有被捕获，那么异常将使你的程序将被停止。

   异常产生后，如果一直没有进行捕获处理，该异常被抛给虚拟机。程序将被终止。

3) 经常会使用的异常 API

   getMessage：获得具体的异常出错信息，可能为 null。

   printStatckTrace()：打印异常在传播过程中所经过的一系列方法的信息，简称异常处理方法调用栈信息；在程序调试阶段，此方法可用于跟踪错误。

2016 年 8 月 12 日 于上海杰普

什么是线程：

1) 进程是指运行中的应用程序，每一个进程都有自己独立的内存空间。一个应用程序可以同时启动多个进程。
2) 线程是指进程中的一个执行流程。一个进程可以由多个线程组件。即在一个进程中可以同时运行多个不同的线程，它们分别执行不同的任务，当进程内的多个线程同时运行时，这种运行方式称为并发运行。
3) 线程与进程的主要区别在于：每个进程都需要操作系统为其分配独立的内存地址空间，而同一进程中的所有线程在同一块地址空间中工作，这些线程可以共享同一块内存和系统资源。比如共享一个对象或者共享已经打开的一个文件。

java 中的线程

1) 在 java 虚拟机进程中，执行程序代码的任务是由线程来完成的。每当用 java 命令启动一个 Java 虚拟机进程时，Java 虚拟机都会创建一个主线程。该线程从程序入口 main()方法开始执行。
2) 计算机中机器指令的真正执行者是 CPU，线程必须获得 CPU 的使用权，才能执行一条指令。

线程的创建和启动

1) 扩展 java.lang.Thread 类;
2) 实现 Runnable 接口;

线程状态

1) 新建状态(New)：用 new 语句创建的线程对象处于新建状态， 此时它和其他 Java 对象一样；仅在堆区中被分配了内存；
2) 就绪状态(Runnable)：当一个线程对象创建后，其他线程调用它的 start()方法， 该线程就进入就绪状态，处于这个状态的线程位于可运行池中， 等待获得 CPU 的使用权。
3) 运行状态(Running)：处于这个状态的线程占用 CPU，执行程序代码。在并发运行环境中， 如果计算机只有一个 CPU, 那么任何时刻只会有一个线程处于这个状态。如果计算机有多个 CPU, 那么同一时刻可以让几个线程占用不同的 CPU，使它们都处于运行状态。只有处于就绪状态的线程才有机会转到运行状态。
4) 阻塞状态(Blocked)指线程因为某些原因放弃 CPU， 暂时停止运行。当线程处于阻塞状态时，Java 虚拟机不会给线程分配 CPU，直到线程重新进入就绪状态，它才有机会转到运行状态。
5) 死亡状态(Dead)：当线程退出 run()方法时，就进入死亡状态，该线程结束生命周期。线程有可能是正常执行完 run()方法退出，也有可能是遇到异常而退出。不管该线程正常结束还是异常结束，都不会对其他线程造成影响。

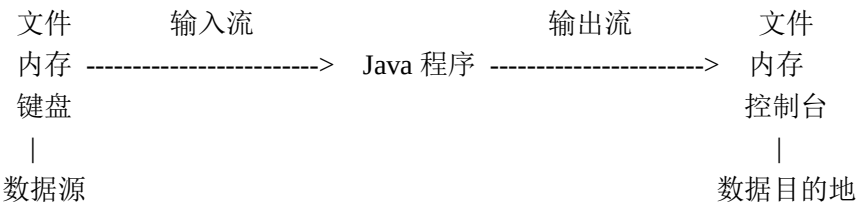
线程调度

1) 分时调度模型：让所有线程轮流获得 CPU 的使用权，并且平均分配每个线程占用 CPU 的时间片。
2) 抢占式调度模型：优先让可运行池中优先级高的线程较多可能占用 CPU(概率高)，如果可运行池中线程的优先级相同，那么就随机选择一个线程，使其占用 CPU。处于可运行状态的线程会一直运行，直至它不得不放弃 CPU。Java 虚拟机采用这种。

2016 年 8 月 16 日 于上海杰普

流的概念

程序的主要任务是操纵数据。在 Java 中，把一组有序的数据序列称为流。根据操作的方向，可以把流分为输入流和输出流两种。程序从输入流读取数据，向输出流写出数据。



Java I/O 系统负责处理程序的输入和输出，I/O 类库位于 java.io 包中，它对各种常见的输入流和输出流进行了抽象。

常用到的字节输入流和输出流

- in:输入流
  - ByteArrayInputStream//读取 byte 类型的数组中的数据
  - FileInputStream//从文件中读取数据；
  - PipedInputStream//(管道流)连接一个 PipedOutputStream;
  - ObjectInputStream/对象输入流；
  - StringBufferInputStream//可以读取一个字符串,在 API 中已经过时
- out: 输出流
  - ByteArrayOutputStream:
  - FileOutputStream
  - PipedoutputStream:
  - ObjectoutputStream

BufferedInputStream 类

1) BufferedInputStream 类覆盖了被过滤的输入流的读数据行为，利用缓冲区来提高读数据的效率。BufferedInputStream 类先把一批数据读入到缓冲区，接下来 read()方法只需要从缓冲区内获取数据，就能减少物理性读取数据的次数。
2) BufferedInputStream(InputStream in)——参数 in 指定需要被过滤的输入流。
3) BufferedInputStream(InputStream in, int size)——参数 in 指定需要被过滤的输入流。参数 size 指定缓冲区的大小，以字节为单位。

DataInputStream 类

1) DataInputStream 实现了 DataInput 接口，用于读取基本类型数据，如 int, float, long, double 和 boolean 等。
2) readByte()——从输入流中读取 1 个字节，指它转换为 byte 类型的数据；
3) readLong()——从输入流中读取 8 个字节，指它转换为 long 类型的数据；
4) readFloat()——从输入流中读取 4 个字节，指它转换为 float 类型的数据；
5) readUTF()—— 从输入流中读取 1 到 3 个字节，指它转换为 UTF-8 字符编码的字符串；

管道输入类：PipedInputStream 类

管道输入流从一个管理输出流中读取数据。通常由一个线程向管理输出流写数据，由另一个线程从管理输入流中读取数据，两个线程可以用管理来通信。

2016 年 8 月 19 日 于上海杰普

Java 8 允许我们给接口添加一个非抽象的方法实现，只需要使用 default 关键字即可，这个特征又叫做扩展方法

```
public interface Formula {
    double calculate(int a);
    default double sqrt(int a){
        return Math.sqrt(a);
    }
}
main:
Formula f = new Formula() {
    @Override
    public double calculate(int a) {
        return a+1;
    }
};
System.out.println(f.calculate(4));
System.out.println(f.sqrt(8));
```

注意:现在接口还可以存在静态方法，

可以使用 接口名.静态方法名 的形式直接调用

Lambda 表达式

```
public class LambdaTest1 {
    public static void main(String[] args) {
        List<String> list = Arrays.asList ("hello","tom","apple","bbc");
            Collections.sort(list, new Comparator<String>(){
            @Override
            public int compare(String o1, String o2) {
                return -o1.compareTo(o2);
            }
        });
        //使用 Lambda 表达式
        Collections.sort(list,(String s1,String s2)->{
            return s1.compareTo(s2);
        });
        Collections.sort(list,(s1,s2)->s1.compareTo(s2));

        System.out.println(list);
    }
}
```

2016 年 8 月 23 日 于上海杰普

网络通信

- IP 地址：
  1) IP 网络中每台主机都必须有一个惟一的 IP 地址；
  2) IP 地址是一个逻辑地址；
  3) 因特网上的 IP 地址具有全球唯一性；
  4) 32 位，4 个字节，常用点分十进制的格式表示，例如：192.168.0.16。
- 协议：
  1) 为进行网络中的数据交换（通信）而建立的规则、标准或约定；(=语义+语法+规则) ；
  2) 不同层具有各自不同的协议。
- 端口号：

端口使用一个 16 位的数字来表示，它的范围是 0--65535，1024 以下的端口号保留给预定义的服务。例如：http 使用 80 端口。

基于 TCP 的 Socket 编程步骤：

1) 服务器程序编写：
   a) 调用 ServerSocket(int port)创建一个服务器端套接字，并绑定到指定端口上；
   b) 调用 accept()，监听连接请求，如果客户端请求连接，则接受连接，返回通信套接字；
   c) 调用 Socket 类的 getOutputStream()和 getInputStream 获取输出流和输入流，开始网络数据的发送和接收；
   d) 最后关闭通信套接字。
2) 客户端程序编写：
   a) 调用 Socket()创建一个流套接字，并连接到服务器端；
   b) 调用 Socket 类的 getOutputStream()和 getInputStream 获取输出流和输入流，开始网络数据的发送和接收；
   c) 最后关闭通信套接字。

基于 UDP 的 Socket 编程步骤：

1）接收端程序编写：
   a) 调用 DatagramSocket(int port)创建一个数据报套接字，并绑定到指定端口上；
   b) 调用 DatagramPacket(byte[] buf, int length)，建立一个字节数组以接收 UDP 包 ；
   c) 调用 DatagramSocket 类的 receive()，接收 UDP 包；
   d) 最后关闭数据报套接字。

2）发送端程序编写：
   1) 调用 DatagramSocket()创建一个数据报套接字；
   2) 调用 DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)，建立要发送的 UDP 包；
   3) 调用 DatagramSocket 类的 send()，发送 UDP 包；
   4) 最后关闭数据报套接字。

2016 年 8 月 26 日 于上海杰普

继承关系　　继承指的是一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能的能力。在 Java 中继承关系通过关键字 extends 明确标识，在设计时一般没有争议性。在 UML 类图设计中，继承用一条带空心三角箭头的实线表示，从子类指向父类，或者子接口指向父接口。

实现关系　　实现指的是一个 class 类实现 interface 接口（可以是多个）的功能，实现是类与接口之间最常见的关系。在 Java 中此类关系通过关键字 implements 明确标识，在设计时一般没有争议性。在 UML 类图设计中，实现用一条带空心三角箭头的虚线表示，从类指向实现的接口。

依赖关系　　简单的理解，依赖就是一个类 A 使用到了另一个类 B，而这种使用关系是具有偶然性的、临时性的、非常弱的，但是类 B 的变化会影响到类 A。比如某人要过河，需要借用一条船，此时人与船之间的关系就是依赖。表现在代码层面，为类 B 作为参数被类 A 在某个 method 方法中使用。在 UML 类图设计中，依赖关系用由类 A 指向类 B 的带箭头虚线表示。

关联关系　关联体现的是两个类之间语义级别的一种强依赖关系，比如我和我的朋友，这种关系比依赖更强、不存在依赖关系的偶然性、关系也不是临时性的，一般是长期性的，而且双方的关系一般是平等的。关联可以是单向、双向的。表现在代码层面，为被关联类 B 以类的属性形式出现在关联类 A 中，也可能是关联类 A 引用了一个类型为被关联类 B 的全局变量。在 UML 类图设计中，关联关系用由关联类 A 指向被关联类 B 的带箭头实线表示，在关联的两端可以标注关联双方的角色和多重性标记。

聚合关系　　聚合是关联关系的一种特例，它体现的是整体与部分的关系，即 has-a 的关系。此时整体与部分之间是可分离的，它们可以具有各自的生命周期，部分可以属于多个整体对象，也可以为多个整体对象共享。比如计算机与 CPU、公司与员工的关系等，比如一个航母编队包括海空母舰、驱护舰艇、舰载飞机及核动力攻击潜艇等。表现在代码层面，和关联关系是一致的，只能从语义级别来区分。在 UML 类图设计中，聚合关系以空心菱形加实线箭头表示。

组合关系　　组合也是关联关系的一种特例，它体现的是一种 contains-a 的关系，这种关系比聚合更强，也称为强聚合。它同样体现整体与部分间的关系，但此时整体与部分是不可分的，整体的生命周期结束也就意味着部分的生命周期结束，比如人和人的大脑。表现在代码层面，和关联关系是一致的，只能从语义级别来区分。在 UML 类图设计中，组合关系以实心菱形加实线箭头表示。

总结　　对于继承、实现这两种关系没多少疑问，它们体现的是一种类和类、或者类与接口间的纵向关系。其他的四种关系体现的是类和类、或者类与接口间的引用、横向关系，是比较难区分的，有很多事物间的关系要想准确定位是很难的。前面也提到，这四种关系都是语义级别的，所以从代码层面并不能完全区分各种关系，但总的来说，后几种关系所表现的强弱程度依次为：组合>聚合>关联>依赖。

2016 年 8 月 30 日 于上海杰普

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。

使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。

项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。

设计模式的分类

1) 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。
2) 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。
3) 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。

工厂方法模式（Factory Method）

- 普通工厂模式，就是建立一个工厂类，对实现了同一接口的产品类进行实例的创建
- 多个工厂方法模式 是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。
- 静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。
- 抽象工厂模式（Abstract Factory）

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

单例模式（Singleton）

单例对象（Singleton）是一种常用的设计模式。在 Java 应用中，单例对象能保证在一个 JVM 中，该对象只有一个实例存在。这样的模式有几个好处：

1) 某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。
2) 省去了 new 操作符，降低了系统内存的使用频率，减轻 GC 压力。
3) 有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。

2016 年 9 月 2 日 于上海杰普

原型模式（Prototype）

原型模式虽然是创建型的模式，但是与工程模式没有关系，从名字即可看出，该模式的思想就是将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象。在 Java 中，复制对象是通过 clone()实现的，先创建一个原型类：

```
public class Prototype implements Cloneable {
    public Object clone() throws CloneNotSupportedException {
        Prototype proto = (Prototype) super.clone();
        return proto;
    }
}
```

适配器模式（Adapter）

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

类的适配器模式：

核心思想就是：有一个 Source 类，拥有一个方法，待适配，目标接口是 Targetable，通过 Adapter 类，将 Source 的功能扩展到 Targetable 里。

例子：

```
    public class Source {
        public void method1() {
            System.out.println("this is original method!");
        }
    }
    public interface Targetable {
        /* 与原类中的方法相同 */
        public void method1();
        /* 新的方法 */
        public void method2();
    }
```

接口的适配器模式

接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。

2016 年 9 月 6 日 于上海杰普

一、 **JDBC 驱动：**

JDBC 驱动程序：Sun 公司所制定的 JDBC 接口实现类的集合，由数据库厂商提供，不同数据库其 JDBC 驱动程序是不同的；

注册驱动

```
DriverManager.registerDriver
                (new com.mysql.jdbc.Driver());
System.setProperty("jdbc.drivers", "com.mysql.jdbc.Driver");
Class.forName("com.mysql.jdbc.Driver");
Driver driver = new com.mysql.jdbc.Driver();
-Djdbc.drivers=com.mysql.jdbc.Driver
```

二、 **获得连接对象**

url 格式 jdbc:子协议:子名称://ip:端口/数据库?属性名 1=属性值 1&属性名 2=属性值 2.....

```
String url = "jdbc:mysql://localhost:3306/javademo?" +
  "user=root&password=root&useUnicode=true&characterEncoding=UTF8";
Connection conn = DriverManager.getConnection (url, user, password);
```

三、 **获得 statement 对象**

```
String sql = "select last_name,salary,dept_id from s_emp";
String sql = "insert into s_emp() values()"
String sql = "update s_emp set salary = 0 where id = 26";
String sql = "delete from s_emp where id = 27"
//实现和数据库交互，传输 SQL 语句到数据库并执行 SQL
ResultSet rs = stmt.executeQuery(sql);
int i = stmt.executeUpdate(sql);
boolean flag = stmt.execute(sql);
//实现和数据库交互，传输参数值到数据库并执行 SQL
 String sql = "select last_name,salary,dept_id from s_emp where id = ? and last_name=?"
 pstmt.setInt(1,26);
 pstmt.setString(2,"Winnie");
 ResultSet rs = pstmt.executeQuery();
 int i = pstmt.executeUpdate();
 boolean flag = pstmt.execute();
```

四、 **处理结果集**

1) 使用结果集(ResultSet)对象的访问方法获取数据;
   a) next():下一个记录
   b) first():第一个记录
   c) ast():最后一个记录
   d) previous():上一个记录
2) 通过字段名或索引取得数据
3) 结果集保持了一个指向了当前行的指针，初始化位置为第一个记录前。

2016 年 9 月 9 日于上海杰普

新建文件 a.txt，并用权限模式、权限值两种方式修改权限为 rwxrwx--x

touch a.txt　　chmod 771 a.txt

在用户的家目录中创建多级目录 a/b/c/d/e

mkdir -p a/b/c/d/e

显示家目录中所有以.txt 结尾的文件

ls *.txt

使用 touch 命令在主目录中建立文件 file1 和 file2

touch file1 file2

在主目录中创建子目录 dir1、dir2

mkdir dir1 dir2

将 file1 file2 复制到 dir1 中，查看主目录与 dir1 目录中有何变化

cp file1 file2 dir1

将文件 file1,file2 从主目录移动至 dir2 中，查看主目录与 dir2 有何变化

mv file1 file2 dir2

将 dir2 中的文件 file1 改名为 myfile1

mv dir2/file1 dir2/myfile1

请使用绝对路径删除文件 myfile1

rm /home/zky/dir2/myfile1

将 dir1 中文件拷入 dir2 子目录，注意有没有提示，考虑原因。

cp dir1/* dir2

删除 dir1 子目录

rm -r dir1

将目录 dir2 复制到 dir1

cp -r dir2 dir1

将目录 dir2 重命名为 dir3

grep:从管道或文件中搜寻所满足条件的行

wc:从文件或管道中统计行数、字符个数、单词个数

wc[options] filename(s)

-l 行数

-w 字数

-c 字符数

2016 年 9 月 13 日 于上海杰普

telnet ssh
exit：退出当前 shell logout：退出最终
用户管理的相关文件
/etc/passwd /etc/shadow /etc/group
passwd 修改当前用户密码
sudo passwd 当前用户 管理员修改用户命令
clear 清屏
pwd 显示当前路径
cd 切换目录
ls 查看目录
ls -a 显示隐藏文件，隐藏目录
ls -R 递归显示目录
打包
tar cvf ***.tar file1 file2 ....归档
tar cvfz ***.tar.gz file1 file2 .....归档并且 gz 压缩
tar cvfj ***.tar.bz2 file file2 ....归档并且 bz2 压缩
对应的解包
tar xvf ***.tar -C /***
解压到指定目录
tar xvfz ***.tar.gz
tar.xvfj ***.tar.bz2
链接文件
硬链接 类似于别名 建立多个名字防止删除
软链接 建立新文件指向原始文件 加上-s 原始文件删除 软链接则无效
apt-get upgrade 更新源的软件包
apt-get dist-upgrade 更新内核及软件包
apt-get update 更新源
apt-cache search ftpd 搜索软件包
apt-get remove *** 只卸载软件包
apt-get autoremove
ifconfig 查看本机网卡配置信息
netstat -rn 查看本机路由信息
traceroute www.baidu.com 查看到指定地点的路由信息
crontab 计划任务

2016 年 9 月 16 日 于上海杰普

Samba 是一个能让 Linux 与 Windows 计算机相互共享资源的软件。
NFS 是分布式计算系统的一个组成部分，可实现在异种网络上共享和 装配远程文件系统。
NFC 协议本身没有网络传输功能，而是基于远程过程调用(Remote Procedure Call,RPC)协议实现的。
NFS 安装与启动
sudo apt-get install nfs-common nfs-kernel-server
sudo service nfs-kernel-server restart
编辑配置文件
sudo vi /etc/export
e.g. /var/ftp/yum 192.168.0.0/24(ro) 192.168.1.0/24(ro)
查看 NFS 服务器上所有共享目录
showmount –e ip_address
查看服务器上哪些共享目录被挂载
showmount –d ip_address
挂载
e.g. sudo mount –t nfs 172.16.0.189:/home/test /mnt
启动时挂载
sudo vi /etc/fstab
e.g. 192.168.0.200:/share /share nfs hard,intr 0 0
虚拟磁盘管理
sudo apt-get install qemu-utils
创建虚拟磁盘
sudo qemu-img create –f qcow2 /bigdata/vmdk/vmfs/server.qcow2 20G
Libvirt 库是一种实现 Linux 虚拟化功能的 Linux API(支持 KVM, Xen)
创建虚拟网卡
virsh iface-bridge eth0 virbr0
KVM 上网的两种模式：
1) NAT，支持主机与虚拟机互访，也支持虚拟机访问互联网
2) Bridge 方式，可以使用虚拟机为网络中具有独立 IP 的主机
克隆虚拟机
e.g. sudo virt-clone –o server –n server_new –f /bigdata/vmdk/vmfs/server_new.qcow2 –force
启动虚拟机
virsh start server_new
关闭虚拟机
Virsh shutdown server_new
删除虚拟机
virsh undefined server_new

2016 年 9 月 20 日 于上海杰普

MySQL 登陆方式

mysql –u 用户名 –p 数据库名 –h 服务端 ip –P 端口

查看数据库名

show databases

使用数据库

use database_name

查看表

show tables

查看表结构

desc table_name

显示某个表的索引

show index from table_name

查看数据库运行信息

show status

列级约束：写在列的后面，针对某个列的约束

Create table student(id number primary key,name varchar(10));

表级约束：写在建表语句的后边，针对某个列的约束

Create table student(id number , name varchar(10),primary key(id));

主键的作用： 强制某个或者多个列唯一并且非空（一个表只能有一个主键!）

联合主键：强制表示两个列联合唯一

```
create table orderline ( order_id integer,
line_id integer,
name varchar(20),
order_date date,
primary key(order_id integer, line_id)
)
```

什么是外键？

1) 外键是用于约束数据的一致性。
2) 外键必须匹配主表的主键列或者唯一列。
3) 外键可以为空。

MySQL 只有 InnoDB 存储引擎才可以试用外键约束

使用 innodb 引擎创建数据库表

```
create table orderline ( id integer primary key,
name varchar(10))
engine=innodb;
```

2016 年 9 月 23 日 于上海杰普

建立表的外键

```
create table suborder1 ( id integer primary key,
name varchar(10),
oid integer,
foreign key(oid) references orderline(id))
engine=innodb;
```

条件查询 where 子句

```
select id,last_name,first_name from s_emp
where salary < 5000
```

order by 排序

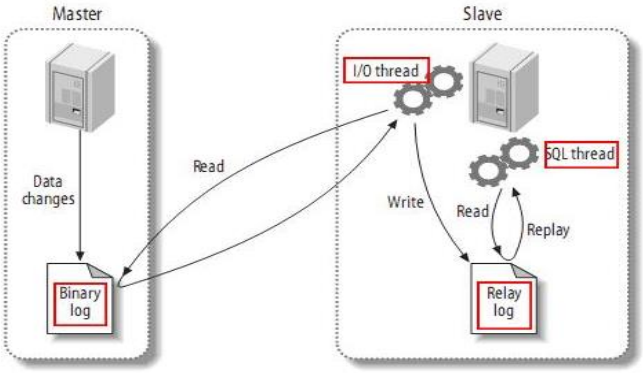
```
select id,last_name,salary from s_emp
order by salary asc/desc;
```

组函数

```
select max(salary),min(salary),avg(salary) from s_emp;
select count(salary),stddev(salary), sum(salary) from s_emp;
```

Mysql 主从复制的原理：

Mysql 的 replication 是一个异步的复制过程，从一个 Mysql instance(Master)复制到另一个 Mysql instance(slave)。在 Master 与 Slave 之间的实现整个复制过程主要由三个线程来完成，其中两个线程(sql 线程和 IO 线程)在 Slave 端，另外一个线程(IO 线程)在 Master 端。



Mysql 复制应用类型：

1. 数据分布
2. 负载平衡
3. 读写分离
4. 高可用性和容错性

Mysql 集群的主主复制：

1. 在 slave 节点授权账号
2. 在 master 节点进行 slave 配置，将原来的 slave 当做 master 进行连接

2016 年 9 月 27 日 于上海杰普

XML 有什么作用,为什么要学习 XML？

1. 第一个需求: 数据传输需要一定的格式
   1) 数据的可读性
   2) 将来的扩展
   3) 将来的维护

XML 现在已经是业内传输数据的标准,它和语言无关

2. 第二个需求: 配置文件,之前使用的.properties 资源文件中描述的信息不丰富
3. 第三个需求: 保存数据,充当小型的数据库。保存数据一般是使用数据库保存,或者使用一般的文件保存,这个时候也可以选择 XML 文件,因为 XML 可以描述复杂的数据关系。从普通文件中读取数据的速度肯定是比从数据库中读取数据的速度快,只不过这样不是很安全而已

文档声明

XML 文件的后缀为.xml

XML 文本要求文件有一个头部声明: 用来告诉解析器一些相关信息

<?xml version="1.0" encoding="UTF-8"?>

表明当前文件是 xml 文件,XML 版本为 1.0,文件内容使用的字符编码为 UTF-8

注意:要小心在系统中这个 xml 文件保存时候的编码,是否和文件内容中设置的编码一致

元素

每个 XML 文档必须有且只有一个根元素。

根元素是一个完全包括文档中其他所有元素的元素。

根元素的起始标记要放在所有其他元素的起始标记之前。

根元素的结束标记要放在所有其他元素的结束标记之后。

属性

```
<student id="100">
    <name>TOM</name>
</student>
```

属性值用双引号（"）或单引号（'）分隔

一个元素可以有多个属性，它的基本格式为：

<元素名 属性名="属性值">

特定的属性名称在同一个元素标记中只能出现一次

属性值不能包括<, >, &

实体 entity

xml 文件中有些特殊的字符是不能直接表示出来的,例如:大于号,小于号,单引号、双引号等等预定义字符实体,由 XML 规范预先定义好了

<	<
>	>
&	&
"	"
'	'

2016 年 9 月 30 日 于上海杰普

狭义的 Hadoop：核心项目------->Common、 HDFS、 MapReduce

广义的 Hadoop：

核心项目+其他项目------->Avro、ZooKeeper、Hive、Pig、HBase 等

以上述为基础，面向具体领域或应用的项目------->Mahout 、X-Rime、Crossbow、Ivory 等

数据交换、工作流等外围支撑系统------->Chukwa、Flume、Sqoop、Oozie、Karmasphere

1. Hadoop Common

为 Hadoop 其他项目提供一些常用工具，如系统配置工具 Configuration、远程过程调用 RPC、序列化机制和 Hadoop 抽象文件系统 FileSystem 等

2. HDFS

分布式文件系统，运行于大型商用机集群，是 Hadoop 体系中海量数据存储管理的基础

3. MapReduce

分布式数据处理模型和执行环境，是 Hadoop 体系中海量数据处理的基础

4. Avro

一种支持高效、跨语言的 RPC 以及永久存储数据的数据序列化系统

5. ZooKeeper

一个分布式的服务框架，解决了分布式计算中的一致性问题。

可用于解决分布式应用中遇到的数据管理问题，如统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等，常作为其他 Hadoop 相关项目的主要组件。

6. Hive

最早由 Facebook 设计，建立在 Hadoop 基础上的 数据仓库框架。Hive 管理 HDFS 中存储 的数据，并提供基于 SQL 查询语言以查询数据。

7. Pig

一种数据流语言和运行环境，用以检索非常大的数据集。Pig 本身运行在 HDFS 和 MapReduce 集群上。

8. Sqoop

SQL-to-Hadoop 的缩写，主要作用是在结构化数据存储和 Hadoop 之间进行数据转换。

是一种在数据库和 HDFS 之间高效传输数据的工具。

配置 ssh 能够本机无密码登录

1. 切换目录

$cd ~/.ssh

2. 生成公钥（id_dsa.pub）和私钥文件(id_dsa)

$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa

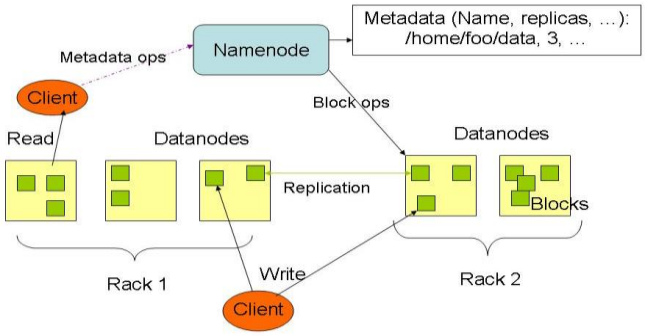
3. 将公钥拷贝到要免登陆的机器上

$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys 或$ ssh-copy-id -i localhost

2016 年 10 月 11 日 于上海杰普

HDFS 中的文件被划分为多个数据块，作为独立的存储单元存放在不同的节点。其中几个主要概念如下：

- 数据块：一个存储单元即一个数据块，默认大小为 128MB。
- 命名节点(NameNode)：HDFS 集群中两大类节点之一，用于管理文件系统的命名空间，维护着文件系统树和整棵树内所有的文件和目录。其中，关于文件与数据块的关系信息以两类文件持久保存在本地磁盘：命名空间镜像文件(fsimage)和编辑日志文件(edits)；关于数据块与节点的关系信息并未持久保存，而是系统启动时 DataNode 重建。
- 数据节点(DataNode)： HDFS 集群中两大类节点之一，用于根据客户端或 NameNode 的调度存储和检索数据块，并定期向 NameNode 发送其所存储块的列表。
- 辅助命名节点（SecondaryNameNode）：该节点与 NameNode 职责并不相同，其主要作用是定期通过 NameNode 中的编辑文件(Edits)合并命名空间镜像文件(fsimage)
- 客户端（Client）通过与 NameNode 和 DataNode 交互来访问整个文件系统。



HDFS 保证可靠性的措施：

1) 冗余备份

每个文件存储成一系列数据块（Block）。为了容错，文件的所有数据块都会有副本（副本数量即复制因子，可配置）

2) 副本存放

采用机架感知（Rack-aware）的策略来改进数据的可靠性、可用性和网络带宽的利用率

3) 心跳检测

NameNode 周期性地从集群中的每个 DataNode 接受心跳包和块报告，收到心跳包说明该 DataNode 工作正常

4) 安全模式

系统启动时，NameNode 会进入一个安全模式。此时不会出现数据块的写操作

5) 数据完整性检测

HDFS 客户端软件实现了对 HDFS 文件内容的校验和（Checksum）检查

2016 年 10 月 14 日 于上海杰普

java 访问 HDFS 主要编程步骤：

1) 构建 Configuration 对象，读取并解析相关配置文件
   Configuration conf=new Configuration();)
2) 获取特定文件系统实例 fs（以 HDFS 文件系统实例）
   FileSystem fs=FileSystem.get(new URI("hdfs://172.16.0.101:9000"),conf,"niuhy");
3) 通过文件系统实例 fs 进行文件操作(以删除文件实例)
   fs.delete(new Path("/user/niuhy/someWords.txt"));

HDFS 写入文件剖析：

1. 客户端通过对象 DistributedFileSystem 的 create 方法创建文件
2. DistributedFileSystem 创建一个 RPC 调用，申请在文件系统的命名空间中创建一个新的文件，此时文件中还没有相应的数据块。客户端开始使用 FSDataOutputStream 对象写入数据。
3. DFSOutputStream 将数据分成数据包，写入 data queue。data queue 由 DataStreamer 读取，并通知 namenode 分配 DataNode，用来存储数据块(每块默认复制 3 块)。分配的 DataNode 放在一个 pipeline 里。Data Streamer 将数据块写入 pipeline 中的第一个数据节点。第一个数据节点将数据块发送给第二个数据节点。第二个数据节点将数据发送给第三个数据节点。
4. DFSOutputStream 为发出去的数据块保存了 ack queue，等待 pipeline 中的数据节点告知数据已经写入成功。如果 DataNode 在写入的过程中失败，关闭 pipeline，将 ack queue 中的数据块放入 data queue 的开始，当前的 Block 在已经写入的 DataNode 中被 NameNode 赋予新的标示，则错误节点重启后能够察觉其数据块是过时的，会被删除。NameNode 则被通知此 Block 是复制块数不足，将来会再创建第三份备份。
5. 当客户端结束写入数据，则调用 stream 的 close 函数。此操作将所有的数据块写入 pipeline 中的数据节点，并等待 ack queue 返回成功。
6. 最后通知 NameNode 写入完毕。

HDFS 读取文件剖析：

1. 客户端(client)用 FileSystem(HDFS 对应于 DistributedFileSystem)实例的 open()函数打开文件
2. FileSystem 用 RPC 调用 NameNode，得到文件的 Blocks 信息，对于每一个 Block，NameNode 返回保存 Block 的 DataNode 的地址，通过 FSDataInputStream 封装。FSDataInputStream 中进一步封装 DFSInputStream 对象，该对象主要用于与 NameNode 和 DataNode 通信。
3. 客户端调用 FSDataInputStream 的 read()函数开始读取数据。
4. DFSInputStream 连接保存此文件第一个 Block 的最近的 DataNode，数据从 DataNode 读到客户端。在读取数据的过程中，如果客户端在与 DataNode 通信出现错误，则尝试连接包含此 Block 的下一个 DataNode。失败的 DataNode 将被记录，以后不再连接。
5. 当此 Block 读取完毕时，DFSInputStream 关闭和此 DataNode 的连接，然后连接此文件下一个 Block 的最近的 DataNode。
6. 当客户端读取完毕数据的时候，调用 FSDataInputStream 的 close 函数。

2016 年 10 月 18 日 于上海杰普

什么是 MapReduce

1) 面向大规模数据并行处理
2) 基于集群的高性能并行计算平台
3) 并行程序设计模型和方法
4) 并行程序开发和运行框架

MapReduce 编程模型

1. 各个 map 函数对所划分的数据并行处理，从不同的输入数据产生不同的中间结果输出
2. 各个 reduce 也各自并行计算，各自负责处理不同的中间结果数据集合
3. 进行 reduce 处理之前,必须等到所有的 map 函数做完，因此,在进入 reduce 前需要有一个同步障(barrier);这个阶段也负责对 map 的中间结果数据进行收集整理(aggregation & shuffle)处理,以便 reduce 更有效地计算最终结果
4. 最终汇总所有 reduce 的输出结果即可获得最终结果

MapReduce 并行处理流程

1) MapReduce 库先把 user program 的输入文件划分为 M 份（M 为用户定义），如图左方所示分成了 split0~4；然后使用 fork 将用户进程拷贝到集群内其它机器上。
2) user program 的副本中有一个称为 master，其余称为 worker，master 是负责调度的，为空闲 worker 分配作业（Map 作业或者 Reduce 作业），worker 的数量也是可以由用户指定的。
3) 被分配了 Map 作业的 worker，开始读取对应分片的输入数据，Map 作业数量是由 M 决定的，和 split 一一对应；Map 作业从输入数据中抽取出键值对，每一个键值对都作为参数传递给 map 函数，map 函数产生的中间键值对被缓存在内存中。
4) 缓存的中间键值对会被定期写入本地磁盘，而且被分为 R 个区，R 的大小是由用户定义的，将来每个区会对应一个 Reduce 作业；这些中间键值对的位置会被通报给 master，master 负责将信息转发给 Reduce worker。
5) master 通知分配了 Reduce 作业的 worker 它负责的分区在什么位置（肯定不止一个地方，每个 Map 作业产生的中间键值对都可能映射到所有 R 个不同分区），当 Reduce worker 把所有它负责的中间键值对都读过来后，先对它们进行排序，使得相同键的键值对聚集在一起。因为不同的键可能会映射到同一个分区也就是同一个 Reduce 作业（谁让分区少呢），所以排序是必须的。
6) reduce worker 遍历排序后的中间键值对，对于每个唯一的键，都将键与关联的值传递给 reduce 函数，reduce 函数产生的输出会添加到这个分区的输出文件中。

当所有的 Map 和 Reduce 作业都完成了，master 唤醒正版的 user program，MapReduce 函数调用返回 user program 的代码

所有执行完毕后，MapReduce 输出放在了 R 个分区的输出文件中（分别对应一个 Reduce 作业）。用户通常并不需要合并这 R 个文件，而是将其作为输入交给另一个 MapReduce 程序处理。整个过程中，输入数据是来自底层分布式文件系统（GFS）的，中间数据是放在本地文件系统的，最终输出数据是写入底层分布式文件系统（GFS）的。

2016 年 10 月 21 日 于上海杰普

序列化的概念

序列化：将结构化对象转化为字节流以便在网络上传输或写入磁盘持久存储的过程

反序列化：将字节流转回结构化对象的逆过程

用途

1. 进程间通信
2. 永久存储

序列化格式的特点

1. 紧凑------>充分利用网络带宽资源和存储空间
2. 快速------>减少序列化和反序列化的性能开销
3. 可扩展------>兼容老格式的消息
4. 互操作------>支持不同的语言

序列化实现

Java 自己的序列化机制----->java.io.Serializable

Hadoop 自己的序列化机制----->org.apache.hadoop.io.Writable

默认的 MapReduce 程序

```
public class MinimalMapReduce {
    public static void main(String[] args) throws Exception{
//          构建新的作业
            Configuration conf=new Configuration();
            Job job = Job.getInstance(conf, "MinimalMapReduce");
            job.setJarByClass(MinimalMapReduce.class);
//          设置输入输出路径
            FileInputFormat.addInputPath(job, new Path(args[0]));
            FileOutputFormat.setOutputPath(job, new Path(args[1]));
//          提交作业运行
            System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

使用自带的 Mapper

ChainMapper:方便用户编写链式 Map 任务，即 Map 阶段包含多个 Mapper

InverseMapper:一个能交换 key 和 value 的 Mapper

RegexMapper:检查输入是否匹配某正则表达式，输出匹配字符串和计数器（1）

TockenCounterMapper:将输入分解为独立的单词，输出个单词和计数器（1）

使用自带的 Reducer

ChainReducer:方便用户编写链式 Reduce 任务，即 Reduce 阶段包含多个 Reducer

IntSumReducer/LongSumReducer:对各 key 的所有整型值求和

2016 年 10 月 25 日 于上海杰普

Combiner：

每一个 map 都可能会产生大量的本地输出，Combiner 的作用就是对 map 端的输出先做一次合并，以减少在 map 和 reduce 节点之间的数据传输量，提高网络 IO 性能，是 MapReduce 的一种优化手段之一。

Partitioner：

Partitioner 的作用是对 Mapper 产生的中间结果进行分片，以便将同一分组的数据交给同一 Reducer 处理。

MapReduce 作业的生命周期

MapReduce 的生命周期，及作业从提交到运行结束经历的整个过程，可大致分为 5 个步骤：

1) 作业提交(JobSubmmiter)
   ① client 请求一个新的 Job ID
   ② 检查作业的输出说明
   ③ 计算作业的输入分片
   ④ 将作业所需的资源（jar 包、作业配置文件、分片元信息文件等）上传到分布式文件系统（一般是 HDFS）
   ⑤ 告知 JobTracker 作业准备执行（即准备调度 JobTracker.submitJob()方法）
2) 作业初始化
   JobTracker 接收到新的作业提交请求后，由作业调度模块（Schedule）对作业进行初始化。
   ① 创建一个表示正在运行作业的对象 JobInProgress，用于跟踪作业的运行情况
   ② 根据分片、配置等信息创建任务运行列表，如 Map 任务、Reduce 任务
   ③ JobInProgress 为每一个 Task 创建一个 TaskInProgress 对象，用于跟踪每个任务的运行状态
3) 任务分配
   ① TaskTracker 周期性地通过 Heartbeat 向 JobTracker 汇报节点的资源使用情况
   ② JobTracker 根据目前资源信息通过一定的策略为任务选择合适的 TaskTracker 节点
4) 任务执行
   ① 作业所需的资源（jar 包、作业配置文件、分片元信息文件等）本地化
   ② 为新任务创建本地工作目录
   ③ 创建 TaskRunner 实例执行任务（不同任务使用独立 JVM 和资源隔离）
5) 进度和状态监控
   ① TaskTracker 周期性地通过 Heartbeat 将所有任务状态发送给 JobTracker
   ② JobTracker 将任务状态更新合并，并尝试为失败的 task 重新启动
   ③ Clinet 每秒查询 JobTracker 或 getStatus()来接收作业或任务状态
6) 作业完成
   ① JobTracker 待所有 task 运行完后，标识作业为“完成”状态。
   ② 清空作业或任务的工作状态

2016 年 10 月 29 日 于上海杰普

1.什么 Scala

Scala 编程语言

多范式编程语言

面向对象

函数式

可扩展性：

Tuple 元组

Trait 特质

2.为什么学习(Scala 的优点)Scala Spark

1.可扩展性

2.复用 Java

直接构建 Java 对象

Spring

互相转化

scala 集合

IO Java

3.语法简洁

String str="hsjhrHfiuhg";

4.静态类型

5.支持并发控制

Thread 资源竞争 死锁 Actor 模型

判断(if 表达式)

```
def min(x:Int,y:Int):Int={
    var a=x
    if(x>y) a=y
    return a
}
```

输出

```
printf("hello,%s! You are %d years old.\n","Fred",42)
```

块表达式和赋值

```
val distance ={
    val dx=x-x0;
    val dy=y-y0;
    sqrt(dx*dx+dy*dy)
}
```

2016 年 11 月 1 日 于上海杰普

函数

def abs(x:Double):Double = if (x >=0) x else –x

异常处理(throw/try)

Scala 处理异常一般通过 throw 抛出一个异常并中止

要么捕获并处理该异常，要么简单中止之

异常抛出：throw new IllegalArgumentException

异常捕获与处理：try{函数体} catch{case…；case…}

异常捕获并中止：try{函数体} finally{A.close()}

try{函数体} catch{case…；case…} finally{A.close()}

throw、try-catch-finally 表达式都能生成值

如：throw new IllegalArgumentException(“Error!”)

def f():Int=try{1} finally{2}

数组

val A=new Array[T](N)

val A=new Array[Int](10)

数组声明时若不给出值，会被初始化，初始化后同样能对数组成员进行赋值

数组成员初始化，如 Int 类型的会都被初始化为 0，String 类型的会被初始化为 Null

val B=new Array(N1,N2)

数组声明时若给出值，Scala 可以进行类型推断，可以不用声明数组类型，长度

A(n)=Nn n(0,N-1) B(0)=N1

数组初始化后能赋值，或是对指定数组成员赋值

val G=B+Array(1,2)

val G=B-Array(1,2)

C += e1

在数组尾部增加一个类型为 T 的元素 e1

C += (e2,e3)

在数组尾部增加类型为 T 的元素 e2,e3

C ++= Array(e2,e3)

在数组尾部增加数组 Array(e2,e3)

C.trimEnd(1)

移除最后一个元素

映射和元组

val Z=Map(a1->b1,a2->b2,a3->b3);

val Z=Map((a1,b1),(a2,b2),(a3,b3));

集合

val digits = Set(1,7,2,9)

2016 年 11 月 4 日 于上海杰普

类是：对象实例)实例的模板，通过构造类，能够使用 new 关键字声明一系列同结构的对象
对象(object：)除了使用类构造对象模板，可以使用 object 构造单例对象
继承：继承是类的拓展
特质：一个类只能继承自一个父类，但可以由多个特质拓展而成用 Trait 来实现混入(mix-in)式的多重继承。
Scala 里相当于 Java 接口的是 Trait(特征)。Trait 的英文意思是特质和性状，实际上他比接口还功能强大。与接口不同的是，它还可以定义属性和方法的实现。Scala 中特征被用于服务于单一目的功能模块的模块化中。通过混合这种特征（模块）群来实现各种应用程序的功能要求，Scala 也是按照这个构想来设计的。
一般情况下 Scala 的类只能够继承单一父类，但是如果是特征的话就可以继承多个，从结果来看就是实现了多重继承。
类定义

```
class HELLOWORLD{
    private val    value1 = "HELLO"
    var value2 = "WORLD"
    def add() {
        println(value1+value2)
    }
    def plus(m:Char)=value2+m
}
```

单例对象
scala 没有静态方法或静态字段，要想实现类似于这种功能，可以借助于单例对象。
object 语法定义了某个类的单个实例
对象的构造器在该对象第一次被使用时调用
object 语法结构与 class 大致相同，除了 object 不能提供构造器参数
通常使用单例对象的环境
作为存放工具函数或常量的地方
共享单个不可变实例
利用单个实例协调某个服务
伴生对象
当一个单例对象存在同名类的时候，称为伴生对象

```
class HELLOWORLD{...}
object HELLOWORLD{...}
```

类和其伴生对象可以互相访问私有属性，但必须存在同一个源文件中
类的伴生对象可以被访问，但并不在作用域中，如;

```
class HELLOWORLD{...}
object HELLOWORLD{ def NOW{...} }
```

HELLOWORLD 类必须通过 HELLOWORLD.NOW 调用伴生对象中的 NOW 方法，而不能直接用 NOW 来调用。

2016 年 11 月 8 日 于上海杰普

拓展

extends 是 Scala 中实现继承的保留字

class week extends month{...}

week 类继承了 month 类所有非私有成员

week 类是 month 类的子类，month 类是 week 类的超类

子类能重写超类的成员（具有相同名称和参数

class week(val num:Int) extends month(var no.：Int){...}

object day extends week{...}

重写

在 Scala 中重写一个非抽象方法必须使用 override 修饰符。

class week extends month{ override def firstday = {...} }

override 保留字实际使用类似与 private，声明这个保留字后的定义、声明是对超类的重写，因此，其也可以写在类定义的参数中

class week(override val lastday：String) extends month{...}

子类的重写或修改 Scala 会检查其超类，但是，超类的修改并不会检查其子类

重写包括字段和方法，但参数不同的方法可以不重写

class month{ def secondday（m:String）={...}}

class week extends month{ def secondday ={...}}

抽象类

抽象类的某个或某几个成员没有被完整定义，这些没有被完整定义的成员称为抽象方法或抽象字段

用 abstract 保留字标记抽象类

abstract class year{

val name：Array[String] //抽象的 val，带有一个抽象的 getter 方法

var num：Int //抽象的 var，带有抽象的 getter/setter 方法

def sign //没有方法体/函数体，是一个抽象方法

}

只要类中有任意一个抽象成员，必须使用 abstract 标记

重写抽象方法、抽象字段不需要使用 override 保留字

构造

```
class month{
    val num = 31
    val days = new Array[Int](num)
}
class week extends month{
    override val num = 7
}
val a=new week
```

2016 年 11 月 11 日 于上海杰普