



# DL4J：人工智能基础（Java 版）

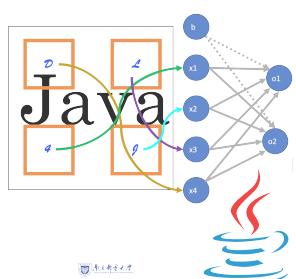
jdeopro 经典之作

作者：@YangJ9966, @imbbty, @simbaba

组织：jdeopro

时间：August 11, 2019

版本：0.05



*Victory won't come to us unless we go to it. — M. Moore*

# 目 录

---

<b>第一部分 预备知识</b>	<b>1</b>
<b>1 开发环境</b>	<b>4</b>
1.1 安装 JDK . . . . .	4
1.2 集成开发环境 . . . . .	5
1.3 配置 DL4J 环境 . . . . .	5
<b>2 编程基础</b>	<b>7</b>
2.1 计算机基础 . . . . .	7
2.2 进制和编码 . . . . .	8
2.3 基本类型 . . . . .	9
2.4 类型转换 . . . . .	11
2.5 数组对象 . . . . .	12
2.6 编程语句 . . . . .	14
2.7 定义函数 . . . . .	20
2.8 流程图 . . . . .	21
<b>3 面向对象</b>	<b>22</b>
3.1 对象封装 . . . . .	23
3.2 创建对象 . . . . .	25
3.3 可见性和权限 . . . . .	26
3.4 如何编译 . . . . .	27
3.5 使用多态 . . . . .	30
3.6 设计模式 . . . . .	32
3.7 枚举类 . . . . .	35
3.8 不可忽视的注解 . . . . .	37
3.9 使用反射 . . . . .	39
3.10 本章总结 . . . . .	40
<b>4 异常处理</b>	<b>41</b>
4.1 异常语句 . . . . .	43
4.2 自定义异常 . . . . .	44
4.3 本章总结 . . . . .	45

<b>5 多线程</b>	<b>46</b>
5.1 创建线程 . . . . .	46
5.2 线程协作 . . . . .	47
5.3 线程池 . . . . .	51
5.4 总结 . . . . .	51
<b>6 输入/输出</b>	<b>52</b>
6.1 输入/输出流 . . . . .	52
6.2 BIO (Blocking IO) . . . . .	52
6.3 Net 编程和 AIO (Asynchronous IO) . . . . .	57
6.4 NIO (New/Non-Blocking IO) . . . . .	59
6.5 异步 Channel . . . . .	60
<b>7 Lambda 表达式</b>	<b>61</b>
7.1 函数分类 . . . . .	61
7.2 应用示例 . . . . .	64
7.3 数学推理 . . . . .	65
7.4 更加强大的集合 . . . . .	66
7.5 本章小结 . . . . .	68
<b>第二部分 数学基础与算法</b>	<b>69</b>
<b>8 微积分</b>	<b>71</b>
8.1 偏导数与梯度 . . . . .	72
8.2 线性拟合 . . . . .	73
8.3 优化问题 . . . . .	74
8.4 损失函数 . . . . .	77
<b>9 概率与分布</b>	<b>80</b>
9.1 全概率公式 . . . . .	81
9.2 最大似然 . . . . .	81
9.3 期望与方差 . . . . .	81
9.4 常见概率分布 . . . . .	82
9.5 矩估计 . . . . .	82
<b>10 线性代数</b>	<b>84</b>
10.1 矩阵和向量 . . . . .	84
10.2 雅克比矩阵 . . . . .	88
10.3 Hessian 矩阵 . . . . .	88
10.4 泰勒展开和牛顿法 . . . . .	89



10.5 向量与矩阵求导 . . . . .	89
<b>11 数学基础-DEMO</b>	<b>90</b>
11.1 向量运算 . . . . .	90
11.2 矩阵运算 . . . . .	93
11.3 梯度下降 . . . . .	95
11.4 概率分布 . . . . .	98
11.5 损失函数 . . . . .	98
11.6 拟合算法 . . . . .	98
<b>12 机器学习</b>	<b>100</b>
12.1 分类问题 . . . . .	100
12.2 线性回归 . . . . .	101
12.3 逻辑回归 . . . . .	104
12.4 正向传播 . . . . .	105
12.5 反向传播 . . . . .	105
12.6 全连接网络 . . . . .	107
<b>13 有监督学习</b>	<b>108</b>
13.1 训练过程 . . . . .	108
13.2 回归问题 . . . . .	108
13.3 SVM . . . . .	110
13.4 卷积神经网络 . . . . .	112
13.5 手写识别示例 . . . . .	112
<b>14 无监督学习</b>	<b>113</b>
14.1 聚类算法 . . . . .	113
14.2 DL4J 示例 . . . . .	115
<b>15 神经网络</b>	<b>116</b>
15.1 评价标准 . . . . .	116
<b>16 深度学习</b>	<b>117</b>
16.1 多层感知机 . . . . .	117
16.2 PlayGround . . . . .	118
<b>第三部分 人工智能应用</b>	<b>120</b>
<b>17 ND4J</b>	<b>122</b>
17.1 ND4J 介绍 . . . . .	122
17.2 使用 ND4J . . . . .	122



17.3 线性回归示例 . . . . .	125
<b>18 DL4J</b>	<b>126</b>
18.1 DL4J 介绍 . . . . .	126
18.2 构造网络 . . . . .	126
18.3 准备数据 . . . . .	128
18.4 配置 DL4J . . . . .	131
18.5 线性回归 . . . . .	131
18.6 逻辑回归应用 . . . . .	133
18.7 多分类问题 . . . . .	133
18.8 聚类问题 . . . . .	133
18.9 卷积神经网络 . . . . .	133
18.10 循环卷积神经网络 . . . . .	133
<b>第四部分 综合运用</b>	<b>134</b>
<b>19 大数据和分布式</b>	<b>136</b>
19.1 云计算 . . . . .	136
19.2 边缘计算 . . . . .	136
19.3 开源架构 . . . . .	136
19.4 示例：人脸识别 . . . . .	138
<b>20 车牌识别</b>	<b>139</b>
20.1 图像切割 . . . . .	139
20.2 算式识别 . . . . .	141
20.3 结果评估 . . . . .	141
<b>21 写诗机器人</b>	<b>142</b>
21.1 word2vec 介绍 . . . . .	142
21.2 如何写诗 . . . . .	142
<b>22 常见问题集</b>	<b>143</b>
<b>A 基本数学工具</b>	<b>145</b>
A.1 求和算子与描述统计量 . . . . .	145
<b>B 最小示例</b>	<b>146</b>



第一部分

预备知识

本书有四部分组成，重点学习 Java 的编程基础。机器学习是本书的一大特色，帮助同学们在学习 Java 的同时也能掌握一些人工智能的知识。现在很多大学甚至中学都开设了机器学习的课程，可见是非常流行，也是大众都能接受的一门知识。虽然涉及到很多数学推理，但对于应用型的人群来说，了解其原理学会使用并不复杂。目前，已有很多成熟的算法模型可以使用。下图 1 引自 scikit-learn。

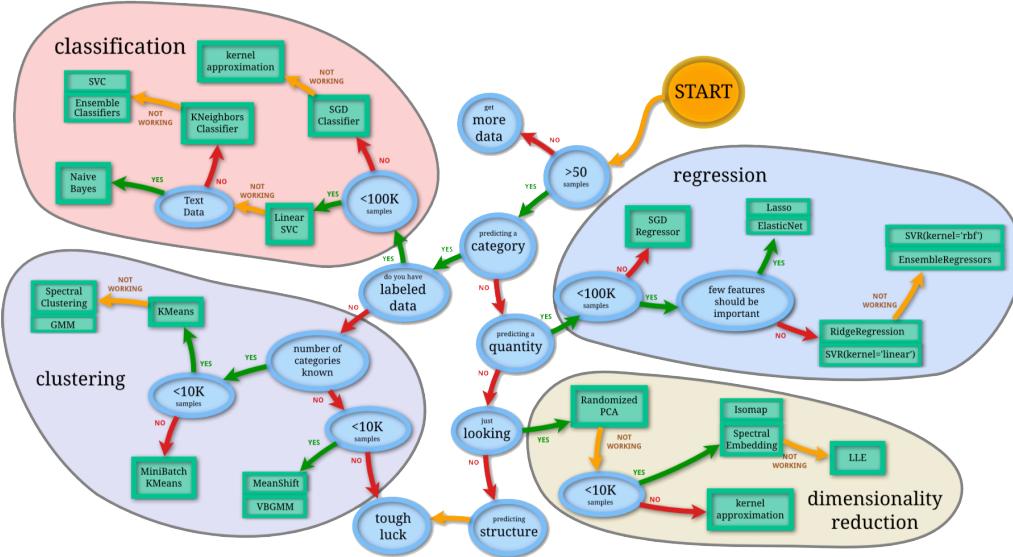


图 1：问题分类

机器学习 (Machine Learning, ML) 是一门多领域交叉学科，涉及概率论、统计学、逼近论、凸分析、算法复杂度理论等多门学科。专门研究计算机怎样模拟或实现人类的学习行为，以获取新的知识或技能重新组织已有的知识结构使之不断改善自身的性能。人类对于大脑的研究，始于计算机诞生之前。而在计算机问世以后，对人工智能的研究掀起了一次次浪潮。直到 21 世纪，这股浪潮席卷了各个领域，人工智能理论也愈加成熟。借助计算机能解决的问题越来越多，变得越来越智能化。计算机从诞生，就被赋予了人工智能的希望。

上个世纪，图灵提出了一个判断计算机是否具备智能的测试标准。测试者与被测试者（一个人和一台机器）隔开的情况下，通过一些装置（如键盘）向被测试者随意提问。进行多次测试后，如果有超过 30% 的测试者不能确定出被测试者是人还是机器，那么这台机器就通过了测试，并被认为具有人类智能。图 2 展示了这样一个场景。

本章作为 Java 学习的入门基础篇，将为大家逐步揭开 Programming 的奥秘。也许，你有很多关于计算机的疑问，请仔细阅读本章相关的内容，相信你必有所获！如若你已具备 Java 编程知识，你可以大胆地跳过本章。但我建议看看也无妨，毕竟 Java 语言出现了很多变化。

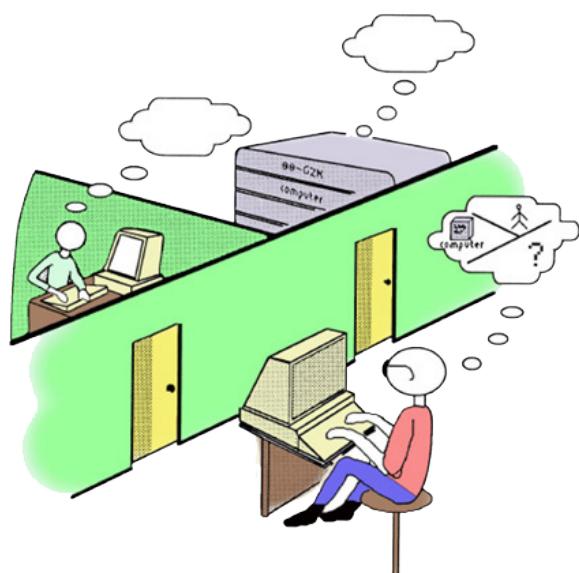


图 2: 图灵测试

# 第 1 章 开发环境

---

本书主要介绍 AI 在 Java 语言上的应用，第一部分内容专为没有编程基础的同学准备，如果你已经具备一定的编程能力可从第二部分开始阅读。下面先介绍本书所需要的开发环境，如果你已配置，也可跳过本章内容。

自从 1996 年诞生第一个 Java 版本以来，已经发布过很多版本，每个版本都会多少引入一些新的语法和功能。本书所提供的示例，都可以在 JDK7+ 上顺利运行，用到开发工具有表 1.1：

开发环境	
1.JDK8+	最新版本是 JDK10，可以尝试
2.IntelliJ IDEA	友好的界面，强大的编程环境
3.Eclipse	不建议使用
4.jupyter	适合验证代码，打开网页就可编码
5.ND4J	强大的运算 Java 库
6.DL4J	深度学习 Java 库

表 1.1：开发环境

## 1.1 安装 JDK

安装 JDK (Java Development Kit) 的时候，我们有 2 个选择：Oracle JDK 和 OpenJDK。经过很多用户验证和测试，你基本可以无视两者之间的差异。使用 Linux 系统的同学，建议保持默认的 OpenJDK。

Sun 在 1998 成立 JCP (Java Community Process) 控制 Java 的开发节奏，任何想要加入的 Java 功能和特性，都必须以 JSR (Java Specification Request) 正式文件的方式提交。然后 JSR 经过 JCP 执行委员会投票通过之后，才可成为最终的标准，有能力的厂商或组织可以根据 JSR 实现 JDK。从 2006 年 Sun 开源 JDK 之后，很顺利地出现了 OpenJDK，它是一个对商业友好的版本。Oracle 在收购 Sun 之后，重申对 JDK 的服务费用。实际上，Sun 在被收购之前就开始索要授权费，但始终没有强制收取。而 Oracle 与 Google Android 之间的 JDK 纠纷加重了产商对 JDK 的忧虑。Oracle Java 并不是完全免费的，要注意其中的收费条款。

1. <http://openjdk.java.net/install/>
2. <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

注意 JDK 和 JRE 的区别，JRE (Java Runtime Environment) 是 Java 程序的运行环境，不要下载错了。安装 JDK 的过程中，全部默认点选“下一步”即可。若在安装过程中，没有添加到环境变量的选项，就需要我们手动配置以下 2 个环境变量。

1. JAVA\_HOME
2. CLASSPATH

环境变量实际上是一种系统变量，方便计算机的软件之间共享配置的方法。有些计算机上安装的 Java 可能不止一个，配置不同的 JAVA\_HOME 就能切换 JDK 版本。如图 ??。

```
drwxr-xr-x 3 root wheel 96 6 9 23:12 jdk-10.0.1.jdk  
drwxr-xr-x 3 root wheel 96 3 5 2016 jdk1.7.0_80.jdk  
drwxr-xr-x 3 root wheel 96 7 7 21:48 jdk1.8.0_171.jdk
```

图 1.1: Java 路径

配置 JAVA\_HOME 非常简单，把它指向对应 JDK 的安装路径就可，不同操作系统有些差异，在此不再赘述。

而 CLASSPATH 这个环境变量，如果使用 IDE 开发环境也可以不配置。如果你的 Java 程序依赖第三方 Jar，不配置 CLASSPATH 就有可能找不到对应的 class。关于 *class not found* 这种问题，第2章会对此进行说明。这里先告诉大家有这个事情，暂时先不配置。

## 1.2 集成开发环境

现代化软件开发，离不开 IDE（集成开发环境）的支持。好用的 IDE 不仅有助于提高编码效率，还可以辅助检查错误，集成项目管理等功能。主流的 Java 开发环境有 Eclipse、IDEA（社区版本免费）和 VS Code，根据个人喜好选择一个即可。本书例子全部在 IDEA 上验证通过。



使用 IDEA，在官网 (<https://www.jetbrains.com/idea/>) 下载社区版本安装即可。软件开发过程，也离不开版本控制 (Revision control) 工具，用于记录一个或多个文件的变化，方便以后查阅特定版本修订情况。最重要的一点，可以使多人协作完成软件开发，确保不同人所编辑的同一档案都是最新的。Git 是目前最流行的版本控制工具，可以在 <https://git-scm.com/downloads> 下载安装。然后，打开 IDEA 配置界面检查 Git 的路径是否正确，修改之后就可以使用了。（Settings->Version Control->Git）

## 1.3 配置 DL4J 环境

DeepLearning4J (DL4J) 是一套基于 Java 语言的神经网络工具包，可以构建、定型和部署神经网络。DL4J 与 Hadoop 和 Spark 集成，支持分布式 CPU 和 GPU，为商业环境（而非研究工具目的）所设计。Skymind 是 DL4J 的商业支持机构。DL4J 包括了分布式、多线程的深度学习框架，以及普通的单线程深度学习框架。定型过程以集群进行，也就是说，Deeplearning4j 可以快速处理大量数据。

本书从理论出发，结合 DL4J 的示例由浅入深地介绍机器学习知识。示例项目由 Maven 进行构建和依赖管理，基于项目 POM（对象模型）定义项目的基本信息，用于描述项目如何构建，声明项目依赖等。<sup>1</sup> 具体的 IDEA 项目配置方法，可参考 DL4J 官网给的建议：

1. Java (SDK) 1.7 之后的版本 (限 64 位)
2. Apache Maven (自动编译和依赖管理)
3. IntelliJ IDEA 或 Eclipse
4. Git

```
$ git clone https://github.com/deeplearning4j/dl4j-examples.git  
$ cd dl4j-examples/  
$ mvn clean install
```

---

<sup>1</sup>Maven 曾是 Jakarta 项目的子项目，现为由 Apache 软件基金会主持的独立 Apache 项目。

## 第 2 章 编程基础

计算机基础薄弱的同学，会感觉编程遥不可及，甚至还没开始就打算放弃。这完全是错误的！随着编程语言的发展，这种顾虑无疑是多余的了。不管你从事什么行业，掌握一门编程语言，都是非常有益的。

首先，你现在面对的是一门语言，就像一门外语一样。其次，它的语法比学了很多年的 English 要简单。从本章开始，我们一起步入 IT 的大门。在开始之前，掌握一些计算机基础非常有必要，可加深对软件运作原理的认知。

### 2.1 计算机基础

当前计算机基本都是冯诺依曼结构，它以 *CPU* 为核心，包含输入/输出以及内存和磁盘设备。程序的指令和数据存在相同的存储设备上，只是物理位置不同。

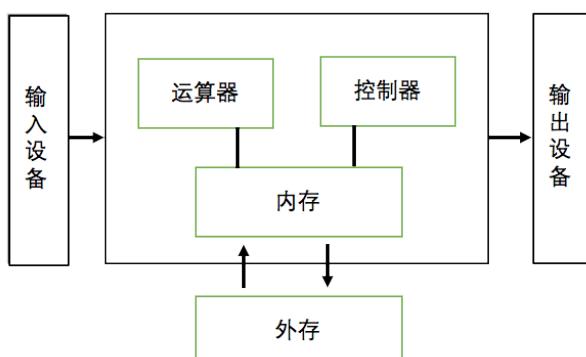


图 2.1：冯诺依曼结构

如图 ??，展示了计算机的架构，它包括很多输入设备：键盘、摄像头、传感器等；而输出设备有显示器和打印机等。<sup>1</sup> 安装在磁盘上的软件，并不能直接运行，等加载到内存才能运转。内存的读写速度要比磁盘快好几个数量级，且磁盘读写以块（block）为主，不利于程序寻址。

第一台计算机虽然每秒只能运算几千次，在当时已经非常快了，成功用于破译电报。计算机刚诞生的那几年，编程是只有科学家才能完成的工作。要编制很多十六进制代码控制计算机，普通人很能掌握。而到 70 年代后期，计算机在商业上的大规模应用，也带动了编程语言的发展。著名的有 C 语言、Objective-C 语言、Pascal 和 Basic 等编程语言。

到了 90 年代，计算机要处理的业务越来越多，需求越来越复杂，相应地程序开发越来越不好控制。传统的软件开发方法已经不能控制局面，迫切需要一种“工程”管理方法。而面向对象设计方法（OOD）和面向对象编程（OOP）技术应运而生。

面向过程到面向对象，是一个必然的过程。早期计算机只用于科研，运算能力和业

<sup>1</sup>STDIN 代表键盘；STDOUT 代表显示器。

面向过程 VS 面向对象	
1. 面向求解问题的过程	1. 数据抽象的过程
2. 函数是编程的核心	2. 函数也是数据的一部分
3. 各种数据	3. 各种对象
4. 函数和数据是分离的	4. 函数离不开对象
5. 代码复用率低	5. 支持继承和组合重用代码

表 2.1: 面向过程与面向对象的差别

务非常有限，代码虽然复杂但业务简单。在商业上大规模应用之后，虽然出现了高级开发语言，但业务代码剧增。面对的不再仅仅是一两个算法问题，而是工程管理和需求设计问题。软件开发参考其它领域，引入软件工程和面向对象开发理论。

## 2.2 进制和编码

学习编程语言，必须先掌握基本的数据类型，了解计算机对二进制数据的使用方法。二进制是只有 0 和 1 的数据类型，好比如十进制包括 0~9。对二进制的运算，也要满位进 1。

表 2.2 展示了它们之间的关系。

二进制 - 十进制	
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10

表 2.2: 进制转换

同学们可思考一下八进制和十进制如何对应，请使用纸和笔验算一下。如果取数字 10，并且告诉你它是一个 8 进制数，应该是多少呢？答案请看脚注。<sup>2</sup>

现在 10 不仅是 10 还是 8 了，为了区分它们要在八进制数前面加个 0 代表这是八进制，注意不是字母 o。在我们生活中，还经常使用 12 进制、60 进制等。而计算机程序常用的进制：二进制、八进制和十六进制等。进制并不是什么高深的数学理论，是常见的数学常识。比如经常说来 2 箱啤酒，这就包含进制在里面，如果一箱啤酒 6 瓶，那就是满 6 进 1。

我们再来说一说进制转换，不论何种进制通过  $\sum_{i=0}^n a_n * H^n$ (H 为进制) 都会换算成十进制。譬如  $18_{16} = 1 * 16^1 + 8 * 16^0 = 24_{10}$ 。通常使用除法取余做进制转换，图 ?? 展示了把 18 转换成二进制的过程。如果要转换为八进制，把 2 换成 8 重复这一个过程即可。

<sup>2</sup>0->1,1->1,2->2,3->3,4->4,5->5,6->6,7->7,10->8，所以答案是 8。

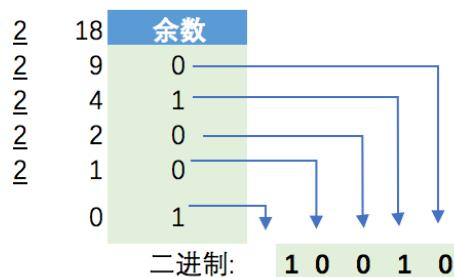


图 2.2: 辗转相除法

接下来，我们再看 16 进制的表示方法。等过很长红绿灯经验的同学，会发现 2 个 LED 也能表示大于 99 的倒计时。譬如用 A0 表示 100，使用 A-F 字母接续，解决了十个制数字不够用的问题。

十六进制 - 十进制	
A	10
B	11
C	12
D	13
E	14
F	15

表 2.3: 十六进制

如果把 10 作为十六进制，就变成十进制的 16 了，不过你得加个前缀 `0x`，不然只能被计算机理解为 10。关于进制，本节就说这么多。值得注意的是，计算机内部只使用二进制，这和咱们人类平时只使用十进制一样。

## 2.3 基本类型

在学完进制之后，咱们再看怎么对他们分类。首先数字可分为：整数、小数。不管是小数还是整数，都有精度的问题。小数的尾数越多，精度也就越高；整数部分越长可表示的数字范围也就越大。对于无理数，在计算机上只能尽可能地保留精度。

由于计算机只能处理二进制，还要把数字转换才能给它处理。对于一个很大的数字，譬如 2147483647 需要很多的二进制位才能表示出来。计算机可以表示的最大的数字，取决于内存和处理器能力。对于 64 位计算机，它的指令和数据加载都是 32 处理器能力的 2 倍。目前 32 位的处理器即将被淘汰，不管是 PC 还是 Arm 都已经普遍采用 64 位处理器。

下表，是 java 支持的所有基本数据类型，也统称为值类型。

编写程序的时候，最好关注一下数据类型的选择。虽然大多数情况下，使用 int 都没

基本类型 - 范围		
byte	8 位 (一个字节)	-128 ~ 127
short	16 位 (两个字节)	-32768 ~ 32767
char	16 位 (两个字节)	-32768 ~ 32767
int	32 位 (四个字节)	-2147483648 ~ 2147483647
long	64 位 (八个字节)	-9223372036854774808 ~ 9223372036854774807
float	32 位 (四个字节)	$3.402823e + 38 \sim 1.401298e - 45$
double	64 位 (八个字节)	$1.797693e + 308 \sim 4.9000000e - 324$

表 2.4: 基本数据类型

有错误。使用 int 类型表示年龄，是很多人的一种奢侈的习惯。实际上，如果使用 byte 就足以 cover 住人类的年龄了。

在 Java 语言中，定义如下：

```
byte age; // 这是注释：还可以有个默认值 byte age=0;
short pages=1234; // 一本书有1234页
int words; // 用来表示字数，虽然不知道但感觉很多的样子
```

我们不直接在代码中使用二进制数字，不然也太长了。但也不只使用十进制，有时候为了便于观察每个 bit 改用十六进制是更好的选择。最大的 int 用十六进制表示  $0x7FFFFFFF$ ，可以清晰地了解很多位都是 1 只有最高位是 0。到这里，我们顺便再说一下负数如何在计算机中表示的。

最大的 int 是  $0x7FFFFFFF$  为什么不是  $0xFFFFFFFF$ ? 就是为了表示负数，把整数对应的二进制最高位用作符号位：1-> 负数、0-> 正数。但还不止于此，实际上-1 并不是 10000001(二进制,byte)

```
byte b = -1;
System.out.printf("0x%X", b); // 输出0xFF
```

尽管计算机没有减法指令，但使用加法指令可以魔术般地实现减法运算。譬如数字 -1 增加 1 之后，是等于 0x100 还是等于 0x00? 对于一个 byte 而言，0xFF 加 1 之后，就超出了它可以表示的范围，进位 1 丢掉之后就剩 0 了。对减法运算中的减号先理解成负号，而负数使用补码表示，就变成只有加法的运算了。

```
3-4*(-5+6)-8 -> 3+(-4)*((-5)+6)+(-8)
-> 00000011 + 11111100 * (11111011 + 00000110) + 11111000
-> 00000011 + 11111100 * 00000001 + 11111000
-> 00000011 + 11111100 + 11111000
-> 11111111 + 11111000
-> 11110111
-> -9
```

所以，负数采用补码表示并参与运算，可消除数据的符号差异，并减少指令的数量。

1. 正数的补码是它自身。
2. 对一个整数的补码再求补码，等于该整数自身。

### 3. 补码的正零与负零表示方法相同。

负数的符号位具有传染性，如果把`byte b = -1`赋值给`int`类型的变量`int n = b`;之后，`n`的值不是`0xFF`而是`0xFFFFFFFF`，它仍旧是-1。如果你错误地理解成`n == 0x000000FF`那就大错特错了！也许你就是期望`n`等于`0x000000FF`，那怎么办呢？

**Listing 2.1:** byte->int

```
byte b = -1;
int n1 = b;
int n2 = b & 0xFF;
System.out.println("byte2Int:\n n1="+n1); // 输出-1
System.out.println("byte2Int:\n n2="+n2); // 输出255
```

上述代码 2.1 清晰地展示了这个区别。

## 2.4 类型转换

类型转换的前提是：不能丢失数据。很明显，`byte`类型的数据，可以无损地赋值给`int`类型；`int`类型的数据也可以赋值给`long`类型。而`float`类型，就不能直接赋值给`int`类型，赋值给`long`也不行的。但是，编程语言一般不阻止这种行为，你可以强制进行转换但存在一定的风险。

1. 隐式转换，安全的转换
2. 强制转换，不安全的转换
3. 数组类型不允许转换，譬如`byte[]`赋值给`int[]`是禁止的。

```
byte b = 127; // 小于127的时候可看作byte类型
int n1 = b; // 隐式转换
long l1 = n1; // 隐式转换

long l2 = 127l; // 显式为long

float f1 = 12;

// 小数默认为double类型，需要强制转换
float f2 = (float) 12.3;
float f3 = 12.3f; // 显式为float

double d1 = 12d; // 显式为double
double d2 = 12.3;

long l3 = (long) d2; // 强制转换

System.out.println("cast:\n d2="+d2); // 输出12.3
System.out.println("cast:\n l3="+l3); // 输出12
```

## 2.5 数组对象

数组，是最常见的数据组织方式，用于存放结构相同的一系列数据，譬如班级所有学生的信息。在 Java 语言中，数组也是一种类型。

**Listing 2.2:** 数组定义和初始化

```
// 定义的时候初始化
int[] numbers1 = {1,4,5,7,2};
int[] numbers2 = numbers1; // 与 number1 共享数据

numbers2[0] = 11;

int[] number3 = new int[]{3,5,6,1}; // 显式创建数组并初始化

int[] number4; // 定以后初始化
number4 = new int[]{4,7,1,6};

int number5[] = number4; // 中括号写后面不建议!

// length 是数组长度
System.out.println("numbers[0] = " + numbers1[0]); // 输出 11
System.out.println(numbers1.length); // 长度
```

数组，它是一个引用类型，它的数据并不存放在栈空间。因此，在函数中返回数组就是返回它的引用。在 Java 中，基本类型（如 int、float 等）的变量都是值类型，函数退出之后局部变量就都被清除了。数组作为引用类型和 C/C++ 是不同的，所以 number2 修改数据之后，number1 的数据也就改变了。

关于数组，知识点总结如下：

1. 数组第一个元素下标是 0，如 number1[0]
2. 使用.length 获取数组长度，没有 sizeof 运算符
3. 最后一个元素的下标是 length-1
4. 数组是一个数据对象，存放在堆内存空间

使用数组，最常见的错误是下标越界，这很有可能是一个空数组导致的。值得关注的是空数组和 null 的区别，空数组是一个正常的数据对象，它的 length 等于 0。另外，byte[] 类型不能强制转换为 int[] 类型。

向量（Vector）通常使用一维数组来表示，甚至矩阵（Matrix）都是用数组来表示的。向量  $\vec{a} = (3, 4)$  和  $\vec{b} = (4, 7)$  的和，就是数组对应元素的加法运算。在 ND4J 库中，矩阵的存储方法有 2 种方式：C 和 F，区别是横向还是纵向划分一维数组。



### 2.5.1 多维数组

对于矩阵这种数据结构，你肯定也希望数组的方式存放，但一维数组类表示矩阵实在是有些复杂，如下代码 2.3。

**Listing 2.3:** 矩阵的一维数组表示

```
/* 3X3的矩阵
 | 1 4 5 |
 | 2 6 8 |
 | 3 1 4 |
 */

int[] matrix = {
    1,4,5,
    2,6,8,
    3,1,4
};

// matrix[2][1]，注意下标0是第一个
int n = matrix[2*3+1];
System.out.println(n);
```

可见使用一维数组的方式有些不便利，Java 支持任意维度的数组，并且不要求每一维度的长度相同！定义多维数组的方式和一维类似。

**Listing 2.4:** 矩阵的二维数组表示

```
// 定义的时候初始化
int[][] matrix1 = {
    {1,4,5},
    {2,6,8},
    {3,1,4}
};

// matrix[2][1]，注意下标0是第一个
System.out.println(matrix1[2][1]);

int[][] matrix2 = new int[3][]; // 最高维度必须明确
matrix2[0] = new int[]{1,4,5};
matrix2[1] = new int[]{2,6,8,9}; // 允许维度不一样
matrix2[2] = new int[]{3,1,4};

System.out.println(matrix2[2][1]);
```

多维数组实际上由一维数组对象组成的，可以单独取出一行赋值给一维数组使用，并



且与多维数组共享数据。

```
int[] row = matrix1[2];
row[1] = 72;

System.out.println(matrix1[2][1]); // 输出72
```

## 2.5.2 数组排序

处理数组最常见的任务就是查找和排序，分为：升序、降序。排序相关的算法确实很多，实现起来难度各不相同。使用 Java 你不用担心如何实现，你提供规则就可以完成排序。Arrays 工具类提供了很多现成的实现：

```
int[] numbers1 = {12, 4, 6, 1, 5, 8};

// 排序，默认是升序
Arrays.sort(numbers1);

// 二分查找，排序之后6在numbers[3]
int index = Arrays.binarySearch(numbers1, 6);
System.out.println("index=" + index);

// 自定义排序，需要int的封装类型Integer
Integer[] numbers2 = {12, 4, 6, 1, 5, 8};

// 定义降序 (a,b)->b-a升序是(a,b)->a-b
Arrays.sort(numbers2, (a,b)->b-a);
```

## 2.6 编程语句

掌握数据类型之后，可以开始语句（statement）的学习了。词汇和句法是所有语言的核心，并且你会发现 Java 语言的语法比 English 少多了。[表 2.5](#)展示了 Java 的关键字。<sup>3</sup>

词汇表				
abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum****	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp**	volatile
const*	float	native	super	while

表 2.5: Java 关键字

使用任何一门语言书写，你都要遵守它的词法和句法，尤其是编程语言。关键字都是敏感信息，不允许用作其它用途。很显然语句（statement）是由一些表达式（expression）组成的。常见的语句有赋值语句、判断语句和循环语句等。

### 2.6.1 作用域

代码和数据有各自的有效范围，譬如处理身高的代码自然不能用于处理年龄。还有一些数据是临时性的，使用完就不再需要了，如果一直在内存里面自然是一种浪费。另外，函数也具有时效性，多次执行可能产生不同的结果，这是因为数据可能已经更新了。

1. 基本的作用域是代码块：{}

在其中定义的数据和代码，超出这个范围就不再有意义。

2. 对象作用域，依存于数据对象：对象. 数据

数据被封装在一个对象里面，在面向对象章节会具体介绍。

3. 模块划分，依存于包命名：包名. 类名

模块化开发经常需要划分为多个 package，避免相同的类名冲突。

针对这些情况，程序使用内存的方式也有 2 种情况：堆内存、栈内存。局部变量只存放在栈内存中，在退出其作用域的时候就会被全部弹出（pop）。而创建的对象因为存放在堆内存中，在程序范围内都可以访问到。

**Listing 2.5:** 变量有效范围

```
if/for/while block {
    int age = 10;
}

if/for/while block {
    print(age); // 不可以使用
}
```

Java 支持一种大括号代码块语法，实际上这个 block 在类（class）加载或 new 对象的时候执行一次，代码 2.6 展示了这个用法。

**Listing 2.6:** 静态代码块

```
static {
    System.out.println("****Welcome****");
    System.out.println("Ver1.0");
}
```

关于作用域的用法，后续章节还会详细说明，涉及 package、class 以及访问权限等内容。

<sup>3</sup> \* not used \*\* added in 1.2 \*\*\* added in 1.4 \*\*\*\* added in 5.0

## 2.6.2 声明语句

声明就是告诉编译器，名称和类型之间的绑定关系。并且在 Java 语言中，定义和声明是相同的概念，变量和函数在声明的时候进行定义。

1. 变量在使用之前必须要定义。
2. 变量定义必须要指明具体的类型。<sup>4</sup>

在 `if/for/while/函数 {}` 中定义的变量，称为局部变量，在使用之前务必要初始化。在一个 block 中优先使用距离自己最近的同名变量。

**Listing 2.7:** 同名变量规则

```
if/for/while block {
    int age = 10;

    if/for/while block {
        int age = 20; // 可以定义同名变量
        print(age); // 输出20
        age = 30;
    }
    print(age); // 仍旧是10
}
```

## 2.6.3 条件语句

判断给定的条件是否满足 (`true`、`false`)，并根据判断的结果决定执行的语句。流程控制就是用条件语句来实现的。如果无条件的话也就无从选择，从本节开始就和同学们好好讲讲条件。判断语句说的就是条件成立可以做啥、条件为假可以做啥。使用条件语句需要 `if` 和 `else` 关键字。

由表 2.6 可知 `else` 语句是可有可无的，不知道做啥写个空的 `else` 也很常见。动手能力强的同学，早就想编个程序试试了。请大家打开上机环境 `jupyter`，或者安装 `jshell` 进行练习。尝试运行一下代码 2.8：

**Listing 2.8:** if-else 练习

```
int n = 25;

if (n % 3 == 0) {
    System.out.printf("%d可以被3整除", n);
} else if (n % 3 == 1) {
    System.out.printf("%d不可以被3整除，余数是1", n);
} else {
    System.out.printf("%d不可以被3整除，余数是2", n);
}
```

<sup>4</sup>JDK10 已引入 var 推断类型

条件语句	
// 只说明成立的时候做啥 <b>if</b> (条件) { //TODO: 成立做啥 }	// 成立和不成立 <b>if</b> (条件) { //TODO: 条件成立 } <b>else</b> { //TODO: 条件不成立 }
// 多个条件，都不成立啥也不做 <b>if</b> (条件1) { //TODO: 条件1成立 } <b>else if</b> (条件2) { //TODO: 条件2成立 } <b>else if</b> (条件3) { //TODO: 条件3成立 }	// 多个条件 <b>if</b> (条件1) { //TODO: 条件1成立 } <b>else if</b> (条件2) { //TODO: 条件2成立 } <b>else</b> { //TODO: 否则只能 }

表 2.6: 判断语句 if-else

控制语句用于控制代码的执行流程，很显然 if 条件语句就是控制语句。常见的控制语句，有以下几种形式：

1. if...else if...else...
2. switch...case...case...default
3. for... 或 for each...
4. while... 或 do...while

而 break 用于从 switch 和循环语句中跳出，continue 则继续执行循环。

#### 2.6.4 循环语句

与条件语句不同的是，循环语句可以一直执行直到条件不成立。很多计算机软件启动之后，就是一个循环等待用户输入的过程。只要用户点击关闭或退出的时候，软件才从消息循环中退出。从这个角度看，操作系统本身就是一个循环，只有在用户关机的时候，才彻底退出。代码 2.9 展示的就是 win32 消息循环。

Listing 2.9: 消息循环

---

```
while(GetMessage(&msg, NULL, NULL, NULL)) {  
    TranslateMessage(&msg);  
    DispatchMessage(&msg);  
}
```

---

想要循环执行一段代码，有很多种实现形式。本节会在这里做个简要对比，你可以根据需要选择最合适的一种。

■ for 语句形式 1：

**Listing 2.10:** for 循环：步进形式

```
for (初始值; 条件; 步进) {  
    代码;  
}
```

该形式的 for 条件列表，每一部分都可以省略，甚至全部省略为 for(;;) 这样它就变成了一种无限循环的代码（死循环）。在循环过程中，使用 continue 可以中止本次往后执行，而 break 用于跳出循环语句。

```
for (初始值; 条件; 步进) {  
    代码1;  
    continue; // 转到步进，然后检查条件  
    代码2;  
}  
  
for (初始值; 条件; 步进) {  
    代码1;  
    break; // 跳出循环  
    代码2;  
}
```

步进形式的 for 语句特别适合遍历一个范围（Range），譬如下标的范围 [0, N)，所以数组通常采用这种形式。

**■ for 语句形式 2:**

```
for (条目:列表) {  
    //TODO: 处理当前条目  
}
```

使用这种 for 循环，特别时候遍历集合，尤其是不能通过下标访问的数据集合。尽管这种方式已经非常简单，在 JDK8 又引入 *foreach* 方法，使用  $\lambda$  表达式遍历数据。

```
/** 定义打印函数 */  
static void print(Integer item) {  
    System.out.println(item);  
}  
  
/** 主函数 */  
public static void main(String[] args) {  
    List<Integer> list = new ArrayList();  
    list.add(1);  
    list.add(2);
```

```
list.add(3);

list.forEach(Vector::print);
}
```

## 2.6.5 switch 语句

在可选条件很多的时候，使用 if 语句可能不是最好的，建议使用 switch 语句简化这种情况。并且很多人认为 switch 语句的效率要比 if 要高很多，除非编译器也能把 if 语句优化的和 switch 一样。

**Listing 2.11:** switch 语句

```
int choice = 2;

switch (choice) {
    case 0: print(choice);break; // if (choice == 0)
    case 1: print(choice);break; // else if (choice == 1)
    case 2:           // else if (choice == 2 || choice == 3)
    case 3: print(choice);break; // else if (choice == 3)
    case 4: print(choice);break; // else if (choice == 4)
    default:print("不支持!");break; // 类似于else
}
```

使用 switch 结构比 if 语句清晰很多，代码 2.11 列出了 choice 可选择的所有情况。为什么 switch 要比 if 效率高呢？实际上，switch 语句会被编译成查表跳转的方式，不需要逐个比较。

## 2.6.6 异常语句

异常语句（Exception），也是一种流程控制语句。在程序执行过程中，不可避免的会遇到一些意外情况。如果当前函数无法处理这种异常，就抛出给上一个函数，直到线程的入口函数，导致程序崩溃为止。

**Listing 2.12:** 异常语句

```
int 分子 = 0;
int 分母 = 2;

try {
    int n = 分母 / 分子;
    System.out.println("商=" + n);
} catch (ArithmetricException e) {
    分子 = 2;
```

```

} finally {
    int n = 分母 / 分子;
    System.out.println("修正：商=" + n);
}

```

上述代码 2.12 展示了除 0 错误，常见的错误还有数组越界、文件不存在等。实际上，有些人还使用 `throw` 语句处理正常的代码逻辑，但不推荐这样做。代码 2.13 展示了这样一个实例：

**Listing 2.13:** `throw` 语句

```

/** 示例：如果不是偶数，就把它改成偶数 */
int n = 3;

try {
    if (n%2 == 1) {
        throw new Exception("不是偶数！");
    }
    System.out.println("偶数：" + n);
} catch (Exception e) {
    n = n + 1;
    System.out.println("修改：" + n);
}

System.out.println("现在：" + n);

```

在下一节，我们将学习如何在函数抛出异常。

## 2.7 定义函数

由于代码块没有名字，无法在别处引用，也就不能复用代码。这种代码搬运工式的编程，没有什么技术含量，不仅会导致代码过大冗余，而且维护成本高。Java 语言中，函数和方法没有区别，依附于对象而存在<sup>5</sup>。函数划定一段代码，提供某个特定功能，也可在别处重复调用。譬如，编写一个 `power` 函数求乘幂，就可以在多处引用它。

```

// static 函数可直接在 main 函数调用
static int power(int n, int exp) {
    int b = n;
    while (0 < --exp) {
        b *= n;
    }
    return b;
}

```

<sup>5</sup>函数是一段代码，通过名字来调用；方法也是一段代码，通过对对象和名字调用



```

public static void main(String[] args) {
    int ret = power(2, 4);
    System.out.println(ret);
}

```

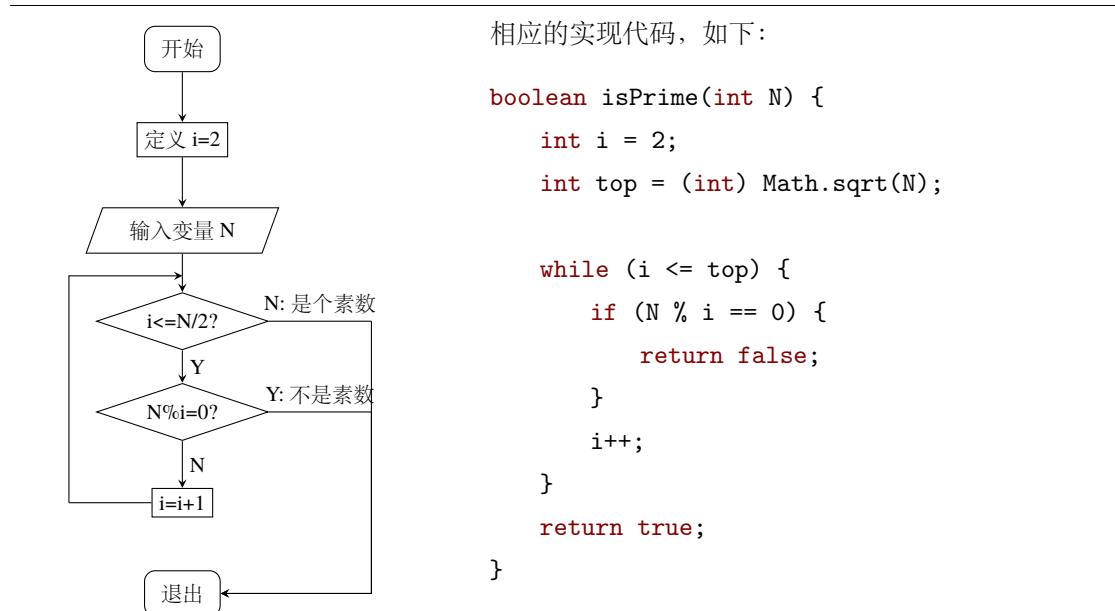
除了函数可以复用代码，后面还会有 $\lambda$ 表达式和对象继承的方式。

## 2.8 流程图

流程图（Flow Chart），它用统一标准的符号描述处理过程。以图形化的方式展示执行步骤，不仅容易理解也方便交流，俗话说“千言万语不如一张图”。对于程序设计，流程图也被称为程序框图，包括：程序框、流程线及文字说明。对于复杂的算法，使用流程图可逐步推演实现代码，作为一种有效的思考方式。

圆角矩形		表示“开始”与“结束”
矩形		表示处理方法
菱形		表示条件判断
平行四边形		表示输入输出
连接线		代表流程的方向

现在，我们要设计一个“判断 $N$ 是否为素数”的程序。素数特指大于1的，除了1和自身外，不能被其他自然数整除的数。设 $m$ 和 $n$ 是 $N$ 的约数 $(N = m \times n)$ ，根据对称性最大的约数不会超过 $\sqrt{N}$ 。使用流程图表示如下：



## 第3章 面向对象

---

从控制流程的角度来看，if、switch 和 while 语句在自己的粒度上控制代码的执行过程。广义上，面向对象（Object Oriented，简称 OO）也是一种流程控制。字面意思上的区别：面向过程和面向对象。但想要解释面向对象是什么，很多资料也都是非常抽象。

使用 *OOP* 的目的当然不是流程控制，而是从数据对象的角度建立整个软件系统。使用面向对象当然会用到流程控制，也离不开函数调用。但 *OO* 的提出，为软件工程提供了理论支持，为纷繁复杂的项目提供了设计工具和开发语言。可以说 *OO* 是为设计师创造的语言，是为具备由上而下掌控全局的程序员开发的语言。所以，*OO* 是一种顶层设计，与面向过程不是一个层面上的开发语言。尽管 *OO* 分解到底层实现，还是要编写函数和控制语句。

使用面向对象编写的代码，具备以下优点：

1. 复用代码，相同的处理逻辑，往往放在基类实现。
2. 容易扩展，通过继承轻松加入新类型。
3. 模块化设计，代码与需求的等价性高。
4. 经验多，UML、设计模式等。

以下是多边形（Polygon）的经典例子：

```
/**  
 * 多边形：三角形、矩形、圆形  
 * 作为参数的polygon是哪个呢?  
 */  
void print(Polygon polygon) {  
    String name = polygon.name;  
    double area = polygon.area();  
    System.out.println("面积：" + area);  
}
```

该代码体现了面向对象的 2 个特点：面向抽象、复用代码。面向抽象（基类）是复用和多态的前提，作为参数的抽象对象 polygon 可能是什么（长方形、圆形等）不重要，重要的是传入的实际对象是否实现了 Polygon 的接口，是否加入了 Polygon 的 Family Tree。只要三角形、长方形继承了 Polygon 就能被看作 Polygon 对象。

如何继承，如何加入这样 Polygon 的家族？在本章都会详细说明，我们先看看 Java 语言用到的一些关键字。下表列出的关键字，我们都会介绍到。

关键词				
class	interface	new	this	super
public	protected	private	package	import
instanceof	extends	implements	abstract	final
default	static	enum	-	-

## 3.1 对象封装

面向对象是一种顶层设计，我们要像设计师那样思考，才能写出正统的面向对象代码。在 coding 之前，要先充分理解业务、创建需求、分解用例和识别交互。现在，我们以多边形计算工具为假设项目，开始本章的内容。

需求分析		
支持的多边形	正方形、长方形、圆形、椭圆、三角形	可能还会要求支持其它多边形
支持的功能	面积、周长	-
提供的数据	坐标点	矩形支持长、宽

表 3.1: 多边形计算工具

根据表 3.1 的内容，是要做一个计算多边形面积和周长的工具。现在不先急于考虑三角形的面积怎么计算、椭圆的周长公式是什么。先从更高的角度看待这些问题，不考虑形状、不考虑实现。能稍微具体一点的是，客户输入坐标之后，我们就能提供面积和周长。这种抽象非常有利，使得软件架构极具扩展性，实际上我们还可能要支持六边形的面积和周长。

定义抽象的多边形类型，请参考左边部分代码，使用到了关键字 `class`。定义好了祖先 `Polygon`，就等其它对象加入这个 Family 了。三角形 `Triangle` 首当其冲，通过关键字 `extends` 表明自己也是一个 `Polygon`。`Polygon` 作为家族树的根，起到了聚合相同接口、划分对象类型的作用。

```
class Polygon {  
    String name = "NA";  
    double[] points;  
  
    /** getter 面积 */  
    double area(){  
        return 0;  
    }  
  
    /** getter 名称 */  
    String name(){  
        return name;  
    }  
}  
  
/** 三角形和圆都继承自 Polygon */  
class Triangle extends Polygon {  
    @Override  
    double area(){  
        return 1.0d; // 假设是1.0  
    }  
}  
  
class Circle extends Polygon {  
    @Override  
    double area(){  
        return 2.0d; // 假设是2.0  
    }  
}
```

上述代码定义了三个类型，其中 `Triangle` 和 `Circle` 都继承自 `Polygon`。使用 `extends` 表示继承关系，也就是加入了 `Polygon` 类别。Java 不支持继承多个类型，好比如你只能同时持有一种国籍，可以减少很多麻烦。还有一个和继承有关的关键字是 `interface`，它可看做 `extends` 的特例，只允许父类有常量和抽象接口。这种父类使用 `interface` 声明，而不是 `class`。从 JDK8 开始，也允许接口类提供 `default` 实现。`interface` 允许你有多本护照，虽然不是本国公民，但也享有很多权利。使用 Java 即使你没有显式地 `extends` 任何类型，也会被继承于 `Object`。也就是说，所有的 Java 类型，都直接或间接地继承于 `Object`。

定义 `class` 就是封装对象的过程，以数据作为属性、以函数作为接口。使用面向对象编写代码，所有的函数都依附于某个类型。所以使用 Java 的时候，你需要先定义一个 `class`，然后才能在其内部定义 `main` 函数。

面向对象还包括设计工具和设计模式。类的这种基础关系，就可以使用 UML 表示，以下是本节例子的类图：

每个对象类型分为三部分：名称、属性、函数接口。其中，属性就是对象包含的数据，函数用于处理数据并向外提供数据。而在属性/函数之前的 `+`、`-`、`#` 表示函数的可访问性。“`+`”代表是 `public` 对我公开的；“`-`”代表 `private` 是私有的，只能在内部使用；“`#`”表示 `protected` 是对外保护但对后代公开。如果不加符号修饰，访问限制也还是存在，它代表只能同由相同 `package` 下的代码才能访问。

什么是 `package`? 英文的意思就是打包，这里代表的就是把一些 Java 类放在一个文件下。至少起到 2 个作用：

1. 不同文件夹下的 `class` 可以重名了。

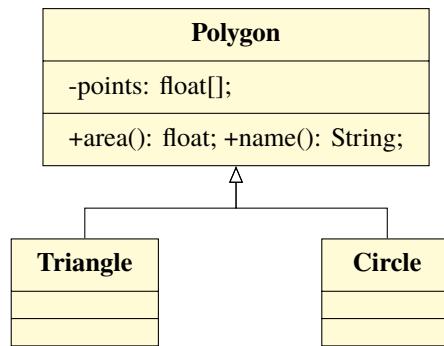


图 3.1: UML: 继承关系/泛化

2. 作为命名空间 (namespace) 使用。

package 可以没有，但建议添加 url 倒序的方式作为 Java 文件的第一行代码，也就是以“.” 分割的字符串作为包名。如以下代码：

---

```

package com.simbabba;

public class Triangle extends Polygon{
    .....
}
  
```

---

声明 package 之后，Java 类就会出现在 com 文件夹下的 simbabba 文件夹里面。相同功能的 Java 类建议使用同一个 package。

## 3.2 创建对象

使用class 定义类型之后，就能通过 new 创建它的实例，也称为对象。继承 (extends) 是一种泛化关系，所有的子类都可以称为是父类的类型。所以三角形对象首先是一个三角形对象，但它也是一个多边形类型的对象。如下代码：

---

```

Triangle t1 = new Triangle();
Polygon t2 = new Triangle(); // 可以使用基类，直观上三角形也是一个多边形

Circle c1 = new Circle(); // 创建Circle对象

Triangle t3 = c1; // 编译失败，圆形肯定不是三角形
  
```

---

使用new关键字，最主要目的是用于区别你调用是当前类的静态方法，还是调用某个类的构造函数。如下代码，定义了 Produce 这样一个静态函数。

---

```

class Polygon {
    Polygon() {
        System.out.println("new_Polygon");
    }
}
  
```

```
}

class Main{
    static Main Polygon(){
        return new Main();
    }

    public static void main(String[] args){
        Main m = Polygon();
        Polygon p = new Polygon();
    }
}
```

这里引入的static关键字，可用它修饰函数和变量，作为该类型对象共享的数据。构造函数用于对象的初始化，并且可定义多个不同参数列表的构造函数。

```
class Polygon {
    Polygon() {}
    Polygon(int n) {}
    Polygon(String m) {}
    // 静态函数可以与类型同名
    static Polygon Polygon(){}
}
```

值得注意的是，静态函数允许和构造函数一样，使用类型的名称。区别是构造函数没有返回值，而函数需要显式地返回对象类型。实际上，类型内部还可以定义其它类型，称为内部类型（inner），有静态内部类型和非静态之分。对于非静态内部类型，它的存储空间是依赖于外部类型的，不能直接定义内部类型的对象。

```
class Main{
    class Test{};

    public static void main(String[] args){
        Test t = new Main().new Test();
    }
}
```

使用new创建对象，就是调用它的构造函数，但如果沒有定义也没问题，因为在你沒定义任何构造函数的时候，Java 编译器会自动生成一个无参构造函数。

### 3.3 可见性和权限

class的确用于定义类型，但它也同时声明了一个 namespace。其中的函数和变量，都是在它的范围内，必须通过它才能访问。命名空间（namespace）是最原始的一种访问限制。访问自有的static函数，不需要引用命名空间，但外部的静态对象或方法，就必

须加类型作为前缀使用。

```
class Polygon {
    static void printOfPolygon() {
    }
}

class Main{
    static void printOfMain() {
    }

    public static void main(String[] args){
        Polygon.printOfPolygon();
        printOfMain();
    }
}
```

使用 `namespace` 只是限制数据的有效范围，算不上真正的访问限制。Java 提供了以下关键字限制属性、函数的可见性。

Java 访问权限				
	同一个类	同一个包	不同包子类	不同包非子类
private	✓			
public	✓	✓		
protected	✓	✓	✓	
缺省 (package)	✓	✓	✓	✓

访问限制可加在类、属性、函数（包括构造函数）的前面，用于限制外部的可见性。使用面向对象减少对外的可见性，灵活性就会越高。因为耦合度低，不会牵一发而动全身。注意，同一个包不要理解为同一个 jar、同一个文件夹。实际上，可以分布在多个 jar 或多个文件夹下面。<sup>1</sup>

## 3.4 如何编译

使用 IDE（集成开发环境）的同学，编写 Java 非常简单，不作更多介绍。本节带大家用记事本，编写和编译 Java。记事本建议使用 notepad++ 或 sublime。



图 3.2: 创建三个 java 文件

刚创建的三个 Java 文件都是空的，并且也都是放在 src 目录下。

<sup>1</sup>jar 文件是 class 的 zip 包，当然 Java 也是可以直接使用 zip 文件的

在 `Polygon.java` 添加以下代码:

```
package com.simbaba;

public class Polygon {
    float[] points; // 坐标点序列 x1,y1,x2,y2...

    float area() {
        log("Polygon", "由我的子孙实现吧! ");
        return 0;
    }

    void log(String tag, String s) {
        System.out.printf("%s:%s", tag, s);
    }
}
```

在 `Triangle` 和 `Circle` 类中加入以下代码:

```
package com.simbaba;

public class Triangle extends Polygon {

    float area() {
        log("Triangle", "我还没来得及写好");
        return 0;
    }
}
```

文件 `Circle.java`:

```
package com.simbaba;

public class Circle extends Polygon {

    float area() {
        log("Circle", "我还没来得及写好");
        return 0;
    }
}
```

启动 CMD 或 shell，在命令行编译这几个 Java 文件 `javac Polygon.java Triangle.java Circle.java`。编译成功的话，就会在当前文件夹下生成 3 个 class 文件。

执行 `javac` 出错的同学，请参考本书 Java 安装和配置部分，检查 JDK 环境变量是否配置



图 3.3: 生成三个 class 文件

正确。单独编译各个 java 文件会怎么样呢？只有 *Polygon* 编译通过了，另外 2 个都没有成功。

```
Triangle.java:3: 错误：找不到符号
public class Triangle extends Polygon {
    ^
    符号：类 Polygon
Triangle.java:6: 错误：找不到符号
    log("Triangle", "我还没来得及写好");
    ^
    符号： 方法 log(String,String)
位置：类 Triangle
2 个错误
```

尽管 *Triangle* 和 *Polygon* 都在 *src* 目录下，但依旧找不到符号。仔细参考 `package` 作用部分内容，创建 *com/simbaba* 目录，并 *cd* 到 *simbaba* 再编译还是失败了。尝试把包名从 3 个 java 文件中都删掉，再编译又能通过了。

有 2 种解决方案：1. 设置环境变量；2. 让 CMD/shell 在 *src* 目录编译，而不是在 *java* 文件所在目录编译。

#### 1. 设置环境变量，可使用 *cp* 传递

```
javac Triangle.java -cp /Users/simbaba/Desktop/gitee/book/part1/src
```

#### 2. 在 *src* 目录下

```
javac com/simbaba/Triangle.java
javac com/simbaba/Circle.java
```

设置环境变量的方式，可以使 *Triangle.java* 在 *simbaba* 目录下编译通过，但在当前目录运行该 class 依旧还会失败。请记住，包名对应的文件夹也是 class 文件的一部分，你应当在 *src* 目录下编译和执行。否则，你可能会遇错误：找不到或无法加载主类 *Triangle*。`main` 是 java 的默认入口函数，并且每个 java 类都可以有一个 `main` 函数。如下，分别在 3 个 java 文件添加 `main` 函数：

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

```
Polygon p = new Polygon();
p.log("Polygon", "main~");
}

public static void main(String[] args) {
    Polygon p = new Circle();
    p.log("Circle", "main~");
}

public static void main(String[] args) {
    Polygon p = new Triangle();
    p.log("Triangle", "main~");
}
```

怎么执行 class 文件? 如果 CLASSPATH 设置正确, 你可在任何地方都可以执行, 但必须把包名带着! 也请注意, 设置 CLASSPATH 之后, 如果是 linux/unix 请 export。不想修改电脑的 CLASSPATH 也可以, 可以使用: “-cp 包所在文件夹, 注意不是类。”

```
java com.simbaba.Triangle -cp /Users/simbaba/Desktop/gitee/book/part1/src
```

## 3.5 使用多态

多态的核心是复用代码, 同一个祖先就可以 (is a 关系, 也称泛化关系)。从创建对象开始, 我们就已经接触到了多态的本质特征。

```
Polygon p = new Triangle();
```

所以, 只要可以使用 *Polygon* 的地方, 都可以替换成它的子类 (Triangle、Circle)。这个替换原则, 是面向对象开发的灵魂, 也是很多人倡导的面向抽象编程。

什么是抽象? 并不是说只有抽象类才叫抽象, 而是更加概括的方式处理问题。实际上, 把 *Polygon* 定义抽象类也才符合现实情况。单纯的多边形 *Polygon* 没有什么意义, 不允许这种对象的存在。所以定义为 **abstract** 类型, 就不再允许被创建出来。

```
public abstract class Polygon {}
```

```
javac com/simbaba/Circle.java
./com/simbaba/Polygon.java:17: 错误: Polygon是抽象的; 无法实例化
    Polygon p = new Polygon();
               ^
```

1 个错误

`abstract`限制 class 的用途还不明显，但放在函数前之后功效就大了！首先，`class` 类型也必须是`abstract`抽象的，抽象的函数只能保留声明不能有函数内容。现在，我们的 `Polygon.java` 如下：

```
package com.simbaba;

public abstract class Polygon {
    String name = "我是谁？";
    float[] points; // 坐标点序列 x1,y1,x2,y2..

    abstract float area();

    void log(String tag, String s) {
        System.out.printf("%s:%s,%s\n", tag, name, s);
    }

    public static void main(String[] args) {
        //Polygon p = new Polygon();
        //p.log("Polygon", "main~");
    }
}
```

把 `area` 声明为`abstract`就会强制要求 `Triangle`、`Circle` 都必须实现 `area` 函数。可见，抽象类与普通类是一样的，但它允许只声明函数而不用实现，且不允许创建它的实例（对象）。但如果该抽象类全部都是抽象函数或常量，建议使用`interface`！

接口主要用于声明函数，也称为“协议接口”。它要求支持它的对象，都必须实现协议中规定的接口。譬如，所有实现 `IArea` 接口的对象，都是可以计算面积的。演示代码如下：

`IArea.java`

```
package com.simbaba;

public interface IArea {
    float area();
}
```

`Polygon.java`

```
public abstract class Polygon implements IArea {
    String name = "我是谁？";
    float[] points; // 坐标点序列 x1,y1,x2,y2..

    public void log(String tag, String s) {
        System.out.printf("%s:%s,%s\n", tag, name, s);
    }
}
```



```
    }  
}
```

---

### Triangle.java

---

```
public class Triangle extends Polygon {  
  
    public float area() {  
        log("Triangle", "我还没来得及写好");  
        return 0;  
    }  
}
```

---

使用interface定义 *IArea* 之后，函数 area 强制为public函数，所有的实现它的类也不能降低访问权限，实际上继承也不可以。根据替换原则很好理解，所有使用父类的地方，都可以换成子类。如果父类一个public供别人使用，你换成子类之后，代码就编译失败了！

使用interface有一个特别的使用方式，建议多使用组合而不是继承。当把 *IArea* 定义为对象的成员变量，可以灵活替换 *IArea* 的实现方式，而不用修改现有代码，譬如增加硬件加速。

## 3.6 设计模式

面向对象编程使我们站在前人的肩膀上设计软件，这种说法一点都不为过。在面向过程的年代，很多 C 语言的初学者很难做到用struct维护数据和函数关系的技巧。想要灵活使用面向对象，有以下几个要点需要掌握：

1. 面向抽象、面向基类、面向接口编程。
2. 接口隔离原则、单一职责原则。
3. 开闭原则。
4. 最少知识原则。
5. 增加一层比少一层有更多的灵活性，但不要跨层交互！

以上五条，说的就是代码要简单、减少互相依赖。要做到简单，你需要经常重构你的代码：分解类、分解函数、合并冗余。

### 3.6.1 工厂模式

使用工厂模式，就是用来创建对象。使用工厂方法比使用new多了一层，实际上就是多了一层灵活。既然使用工厂生产对象，该对象的构造函数一般都是private私有的。以下是 URL 的例子，请仔细斟酌。

#### 1. 打开 FTP 地址

---

```
URL url = new URL("ftp://simbaba.com:1234@192.168.1.47:21/test.txt");
```



```
URLConnection connection = url.openConnection();
System.out.println(connection);
//sun.net.www.protocol.ftp.FtpURLConnection:
//      ftp://simbaba.com:1234@192.168.1.47:21/test.txt
```

## 2. 打开 HTTP 地址

```
URL url = new URL("http://simbaba.com/test.txt");
URLConnection connection = url.openConnection();
System.out.println(connection);
//sun.net.www.protocol.http.HttpURLConnection:
//      http://simbaba.com/test.txt
```

这里使用了工厂方法 `openConnection`, 根据不同的参数创建不同的对象。其中, 也用到了面向抽象编程, `FtpURLConnection` 和 `HttpURLConnection` 都是 `URLConnection` 的具体实现。由于工厂方法生产的是抽象类型, 很容易扩展现有工厂的生产能力。在不影响现有客户端的情况下, 增加对“`ssh://simbaba.com:21`”的支持。

### 3.6.2 装饰模式

装饰模式可用于动态地为对象增加一系列的功能, 当然继承就可以复用父类代码并增加功能, 为何不使用继承呢? 首先, 继承讲究顺序 A->B->C->D, 这会导致逐级增加对象的复杂度, 不符合接口隔离原则, 导致非常臃肿。然而, 使用装饰模式可以灵活的进行各种附加功能组合。装饰模式凸显了面向抽象编程和接口隔离的原则的作用。

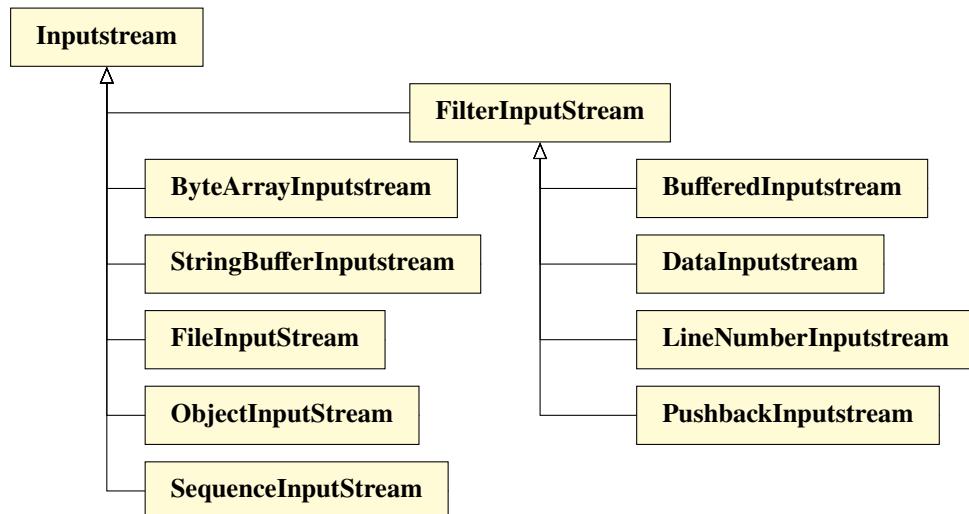


图 3.4: UML: 装饰模式之 IO

使用装饰模式的好处是, 你可以像拼积木一样自由的添加。需要缓冲能力, 你就来一个 `BufferedInputStream`; 要读写各种类型的数据 (`int`、`char`、`double` 等), 那就来一个 `DataInputStream`。你可以不关心它们之间的顺序, 结果总能是正确的, 不过合理的使用才能提高 IO 的读写效率。一般来说, `BufferedInputStream` 总是放在入口处, 这个字节流

的蓄水池有 8M 打下，可有效减少磁盘读写。

以下代码，转换文件输入流中的单引号为双引号：

```
// try...resorce语法
try (FileInputStream fis = new FileInputStream("test.dat"))
{
    // 提供缓冲区
    BufferedInputStream bis = new BufferedInputStream(fis);
    // 提供读取撤回功能
    PushbackInputStream pis = new PushbackInputStream(bis);

    String dst = "";
    int dat = pis.read();

    while (dat != -1) {
        if ((char) dat == '\'') {
            pis.unread('\'');
        } else {
            dst += (char) dat;
        }
        dat = pis.read();
    }
    System.out.println(dst);
} catch (Exception e){
    e.printStackTrace();
}
```

### 3.6.3 建造者模式

函数的参数最多能有多少个？没必要知道答案，但函数有太多参数的话就非常难使用了。Java 也不支持给参数提供默认值，假如你的构造函数有十几个参数的话，创建个对象也变得非常困难了。工厂模式可以解决这个问题，使用建造模式可以更灵活的解决。

这种情况，即使是工厂模式也要借助建造者模式构建对象。它们的区别是：

1. 工厂模式解决如何获取对象，而建造者模式解决如何创建对象。
2. 建造者模式可逐步丰满这个对象，可化解对象参数多的问题。

以下是 Android 创建对话框的示例代码：

```
new AlertDialog.Builder(this)
    .setTitle("请输入")
    .setIcon(android.R.drawable.ic_dialog_info)
    .setView(new EditText(this))
```

```
.setPositiveButton("确定", null)
.setNegativeButton("取消", null)
.show();
```

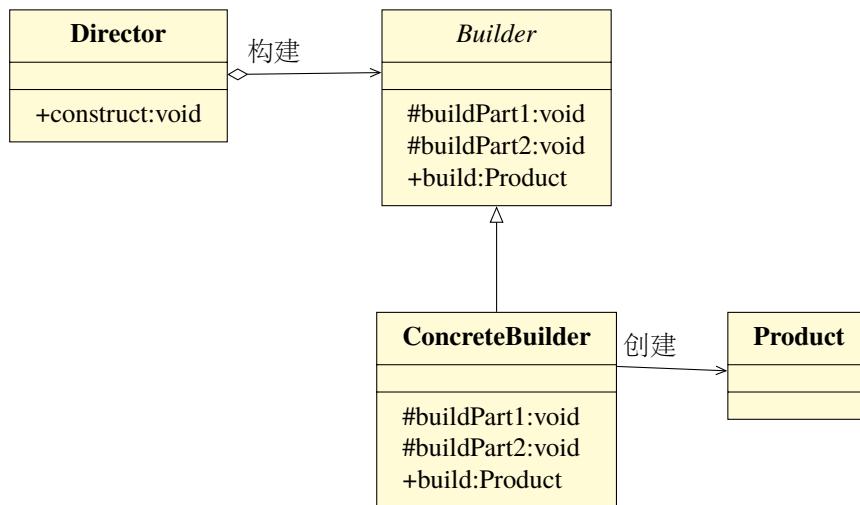


图 3.5: UML: 建造者模式

图 3.5 中，抽象的 **Builder** 提供了丰富的扩展性，在不调整现有代码的情况下，支持更多的 **Builder**。更重要的是 **Builder** 限制了对象不可或缺的部分，甚至也限制它们的调用顺序。

## 3.7 枚举类

使用 Java 定义常量，最常见的手段是 `public static final int MSG_XXX=0;` 这样定义的多数情况下都是好的，但隐藏着很大的隐患。常量的值现在定义为 0，改天可能会被改成 1。常量的名字虽然没改变，但值被人改了。其它同学看了常量定义之后，也直接与 0 比较的！

1. 之前已编译的 class 虽然引用的是 `MSG_XXX` 实际上固定为 0 了。
2. `MSG_XXX` 的值可以被随意修改。
3. 这种定义，误导其它代码直接使用常量的值，丢失常量的全局性。

做一个实际的例子，加深对此问题的体会。首先，在 `IArea.java` 定义 `PI=3.15`，然后在 `Circle.java` 中添加 `PI` 的打印。编译 `IArea.java` 和 `Circle.java`。执行 `java com.simbaba.Circle` 输出 3.15。后来，更正 `PI` 为 3.14 之后，没有重新编译 `Circle.java`，重新执行还是 3.14。这个问题，主要出现在常量和引用它的类不在一个 jar 的情况。

`IArea.java`

```

package com.simbaba;

public interface IArea {
    float PI = 3.15f; // 接口中变量，默认都是public static final
}

```

```
    float area();  
}
```

Circle.java

```
package com.simbaba;  
  
public class Circle extends Polygon {  
    ...省略部分代码  
    public static void main(String[] args) {  
        Polygon p = new Circle();  
        p.log("Circle", "PI=" + PI);  
    }  
}
```

枚举（enum）的字面意思，就是一个可枚举的列表，并且值不可更改，一般不用于定义单个常量。在 Java 中一切皆对象，定义常量使用的枚举类型也不例外，它靠什么保证的呢。下面，我们以颜色 Color 举例说明，它包括：RED、YELLOW、GREEN、BLUE、GRAY。

```
// 定义Color枚举类型  
enum Color {  
    RED, YELLOW, GREEN, BLUE, GRAY  
}  
  
// 在main函数打印  
Color red = Color.RED;  
System.out.println(red); // 输出RED
```

实际上，Java 还是依靠static final 实现的常量，但它是对对象引用类型，不存在之前提到的问题。在 Java 中，一切皆对象可通过 javap 反编译 Color.class 文件验证。

```
Classfile classes/com/simbaba/Color.class  
Last modified 2018-12-19; size 1055 bytes  
Compiled from "Triangle.java"  
final class com.simbaba.Color extends java.lang.Enum<com.simbaba.Color>  
    flags: ACC_FINAL, ACC_SUPER, ACC_ENUM  
Constant pool:  
    . . .  
    public static final com.simbaba.Color RED;  
        descriptor: Lcom/simbaba/Color;  
        flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL, ACC_ENUM  
  
    public static final com.simbaba.Color YELLOW;  
        descriptor: Lcom/simbaba/Color;
```



```
flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL, ACC_ENUM
. . .
private com.simbaba.Color();
descriptor: (Ljava/lang/String;I)V
flags: ACC_PRIVATE
```

细心的同学也会发现，`enum`被编译成 `class`，并且构造函数是私有（`private`）的。之前定义的 `RED`、`YELLOW` 等全都是 `static final` 类型的常量了。所以，你无法创建更多的 `Color` 对象，只有列表中的那几个。

`RED` 在 `Color` 的构造函数中被初始化 `RED=new Color("RED", 0)`。并且 `Color` 继承于 `Enum` 类型，它有 2 个重要的属性：`name`、`ordinal`。想要获得 `RED` 整数值，就使用 `ordinal`，而它的 `name` 就是 `RED`。默认情况下，`RED` 的值只能是 0，如何修改为 `0xFF0000` 呢？你要自己显示定义 `private` 的构造函数了。

```
enum Color {
    RED(0xFF0000),
    YELLOW(0xFFFF00),
    GREEN(0x00FF00),
    BLUE(0x0000FF),
    GRAY(0x888888);

    // 不允许定义为public，否则无法编译通过
    Color(int ordinal) {
        this.ordinal = ordinal;
    }

    public int value() {
        return ordinal;
    }
    private int ordinal;
}

// 在main函数打印
Color red = Color.RED;
System.out.println(Integer.toHexString(red.value())); //输出ff0000
System.out.println(Integer.toHexString(red.ordinal())); //输出0
```

关于枚举类型的知识基本就是这些，只要你抓住 `enum` 类型的本质，可以自由使用。

## 3.8 不可忽视的注解

注解（annotation）是很多初学者都会跳过的内容，包括笔者初学 Java 的时候也是如此。自从一些依赖注解的编程框架出现之后，它变得越来越重要，已经成为代码不可或缺的一部分。

缺的一部分。比较典型的应用：JavaDoc、JUnit4+、Spring、Struts2。

使用注解可以在权限修饰符之上，为函数/类/属性提供更细粒度地控制。一个很典型的例子，早期的 JUnit 版本要检查函数的名字是否以 *test* 开头，才被识别为一个测试用例函数。

```
public void setup() {  
    //初始化函数，名字必须是setup  
}  
  
public void test_circle_area() {  
    //测试函数必须以test开头  
}
```

如上代码，想要忽略一个测试用例，只有把函数的名字去掉 *test* 才能避免。而在 JUnit3 之后，使用注解之后，控制就非常灵活了。

```
@Before  
public void init() {  
    //初始化函数，名字必须是setup  
}  
  
@Test  
public void circle_area() {  
    //测试函数必须以test开头  
}
```

使用面向对象编程，特别要求属性必须是 **private**，任何读写都通过 Setter/Getter 完成。这可保护对象的健壮性，避免对象处于一种紊乱的状态。譬如，*Triangle* 对象 *t1* 的长、宽、名称创建出来之后，还可以被外部直接修改，甚至修改为 *t1.name=“Circle”*。对外部暴露的隐式越多，越容易出现耦合问题，导致代码粘连在一起。

正确的做法是，定义 *name* 为 **private** 的，并且只提供 **Getter** 函数。

```
private String name;  
  
public String getName() {  
    return name;  
}
```

这种只有一个字段（属性）的情况，添加 **Getter** 比较简单。但如果有十几个或者更多的时候，就满屏是 **Getter**、**Setter** 了。借助 *lombok* 提供的 **@Getter** 和 **@Setter** 注解，可以使情况变得简单。

```
@Getter  
private String name;
```

使用 **@Getter** 注解，可在编译阶段为对象生成 **get** 函数。从事 Web/数据库开发的同学，经

常要编写有十来个属性的 Java Bean 对象。在该注解出现之前，经常要借助反射 (Reflect) 才能减少这种重复性工作。<sup>2</sup> 在此总结一下，注解可以提供的功能：

1. 修饰函数，以便在执行、编译阶段检出需要的函数，譬如 JUnit。
2. 插入代码，添加特殊功能的注解，可在编译阶段插入代码，譬如 Lombok。
3. JavaDoc 等
4. 全局注解，自动应用于所有类型，譬如 IDE 插件、特殊增强（运算符重载）。

实际上，我们很少编写注解，但工作中经常使用。关于注解的知识，在这有限的篇幅只能介绍这么多，大家了解会使用即可。

## 3.9 使用反射

面向对象提供的权限修饰主要在编译阶段进行检查，这种限制在运行阶段就很容易被打破。即使是 C++ 对象，通过对对象地址偏移量也可访问其私有成员。Java 干脆提供了相关的 API，帮助你在运行期动态访问对象的任意成员，对于私有成员，临时调整 accessible 就可正常访问。

```
Circle = Class.forName("com.simbaba.Circle");
Object circle = Circle.newInstance();
Method area = Circle.getMethod("area");

Field circle_name = Circle.getSuperclass().getDeclaredField("name");
circle_name.setAccessible(true);
circle_name.set(circle, "Circle");
printf("型号=%s, 面积=%f", circle_name.get(circle), area.invoke(circle));
```

JDK 提供的反射 API 有很多，以下仅列举 Field 部分 API。

JDK 反射 Field 的 API	
getFields()	获取所有 public 字段，包括父类字段
getDeclaredFields()	获取所有字段，public 和 protected 和 private，但是不包括父类字段

近年来，AOP 编程已经非常流行。在不改变既有类库的情况下，运行期间动态地插入代理，包装监视调用过程，譬如 API 鉴权。

```
before("调用之前");
invoke(object, method, args); // 调用对象的函数
after("调用之后")
```

常见的动态代理有 2 中实现方式：JDK、CGLib。但 JDK 仅支持接口的动态代理，使用非常受限。所以主流的框架，使用 CGLib 更多一些。在这里仅演示 JDK 提供的方式。

```
Polygon polygon = new Triangle();
```

<sup>2</sup> 反射可以读写私有成员，并在运行期临时修改可访问权限。

```
InvocationHandler invoker = (proxy, method, arg) -> {
    System.out.println("调用之前");
    Object obj = method.invoke(polygon, arg);
    System.out.println("调用之后");
    return obj;
};

IArea iArea = (IArea) Proxy.newProxyInstance(
    AreaProxy.class.getClassLoader(),
    Polygon.class.getInterfaces(),
    invoker);

iArea.area();
```

需要注意的是 `getInterfaces` 要用它的直接继承/实现类获取。动态代理技术结合注解(Annotation)改变了 java 改变了工作方式。相关内容超出本书篇幅, 请感兴趣的读者自行查阅资料学习。对于本书, 我们比较感兴趣的是, 如何使 2 个向量类型直接相加。

## 3.10 本章总结

本章对 java 基础和常见开发方法作了较为详细的介绍, 限于篇幅不作更多展开。面向对象编程不是一门语言, 它是一种发展理念。通过本章的学习, 要掌握 Java 创建对象的方法, 并学会运用面向对象设计软件。Java 不支持运算符重载, 但有 GitHub 开源项目<https://github.com/amelentev/java-oo>, 能在编译阶段把对象的“+\*/”替换为 `add`、`subtract`、`multiply`、`divide`。实际上, 它和本章介绍的注解(Annotation)有一些渊源, 有兴趣的读者还是自己阅读开源代码。

```
//不使用运算符重载
c1 = a.negate().add(b.multiply(b)).add(b.divide(a));
//使用开源的运算符重载
c2 = -a + b*b + b/a;
```

## 第4章 异常处理

---

程序在运行过程中，出现异常了怎么办？是崩溃退出还是给用户一个友善的提示。通过本章的学习，你将了解什么是异常、如何处理异常，又如何自定义异常。异常可分为以下三类：系统错误、异常和运行时异常。

### 1. 系统错误 (Error):

是由 Java 虚拟机抛出的，代表的是一些比较严重的错误，这样的错误很少发生，一旦发生无法挽回。

### 2. 异常 (Exception):

是由程序或外部环境引起的，如打开不存在的文件等。

### 3. 运行时异常 (RuntimeException):

主要指的程序中出现的错误，如数组下标越界、类型转换错误等。

其中，系统错误 (Error) 会导致程序的退出或者崩溃，通常无法挽回；而运行时异常 (RuntimeException) 是程序实现过程中引入的错误，可通过修改程序解决，所以这两类异常（包括其子类）称为免检异常 (Unchecked Exception)，不要求在代码中进行额外处理。而其他异常称为需检异常 (Checked Exception)，必须编写处理代码。如何进行检查并进行处理呢？

文件不存在、网络已断开等异常，都是常见的可预见的异常，对于这类异常应加以保护。传统的编程语言，通常先检测文件是否存在，再去执行读写操作。但这种异常检测行为，可能会被很多人忽略。而 Java 语言中，常见异常 (Exception) 基本都是 *checked* 异常，不允许无视。

---

```
File file = new File("test.txt");
if (file.exists()) {
    readFile(file);
} else {
    System.out.println("文件不存在！");
}
```

---

使用异常可强制所有人，遵守相同的规则而提高代码质量。它也是一种 *WARN* 声明，使用该 API 存在哪些风险，你要做好保护。

---

```
File file = new File("test.txt");
try {
    readFile(file);
} catch (IOException e) {
    e.printStackTrace();
}
```

---

以上代码中，`readFile` 显式抛出（throws）读写异常，必须使用`try...catch...`的形式。在 `try` 块中，可以忽视任何异常，不需要考虑意外情况；而在 `catch` 中捕获对应的异常，并加以处理。组成异常语句的关键字有：`try`、`catch`、`finally`。其中，`try` 可以带有参数，而 `catch` 必须明确异常类型的参数，`finally` 用于最终收尾只是一个语句块。JDK 提供了一些基本的异常类型，如图 4.1 所示。

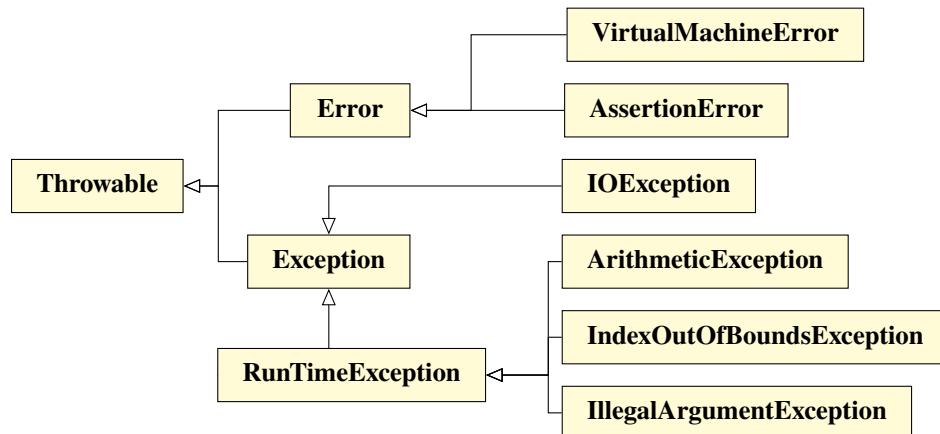


图 4.1: UML: 异常类结构图

异常类的实现代码都非常简单，主要是起到分类和提示的作用，想自定义一个异常也非常简单。先看一个 `IOException` 的代码。

```
public class IOException extends Exception {  
    static final long serialVersionUID = 7818375828146090155L;  
  
    public IOException() {  
        super();  
    }  
  
    public IOException(String message) {  
        super(message);  
    }  
  
    public IOException(String message, Throwable cause) {  
        super(message, cause);  
    }  
  
    public IOException(Throwable cause) {  
        super(cause);  
    }  
}
```

## 4.1 异常语句

鼓励使用异常，但不建议用异常做业务逻辑，有异常处理的代码无论是生成的 class 文件还是运行时都比较占用资源。上一节说过，有 3 个关键字和异常语句有关。其中，try 和 finally 只能有一个，而 catch 语句优先匹配第一个合适的异常类型。再强调一次面向对象的概念：继承关系是一种 is a 关系。

```
try(FileInputStream fis = new FileInputStream(file)) {
    fis.read();
} catch (FileNotFoundException e) {
    System.out.println("文件不存在");
} catch (IOException e) {
    System.out.println("读文件时遇到错误");
} finally {
    System.out.println("无论如何，都到此善尾");
}
```

以上代码，*FileNotFoundException* 是一个 *IOException*。如果交换它们之间的顺序，*FileNotFoundException* 就是一条无效语句，编译也无法通过。try 从 JDK7 之后才可以包含参数，准确的说是一段代码。其中定义的变量必须是实现了 *AutoCloseable* 的对象，因此可省掉关闭资源的步骤。

```
class MyAutoCb implements AutoCloseable {
    public void print() {
        System.out.println("我要输出我自己");
    }
    @Override
    public void close() throws Exception {
        System.out.println("我被关了....");
    }
}

try(MyAutoCb cb = new MyAutoCb();FileInputStream fis = new FileInputStream(file))
{
    fis.read();
    cb.print();
}省略...
```

以上代码中，*MyAutoCb* 对象的 close 函数会自动被调用。接着说，异常语句的组合方式。catch 和 finally 是不能单独存在的，而 try 虽然可以但没有什么意义，其中的异常又被再次抛出。常见的书写方式：

```
try {
```

```
    } catch(XxxException e) { // 捕获到的异常，还可以重写抛出
        throw e; // 再次抛出
    }

    try {
    } catch(XxxException e) {
    } finally { // 先try语句，再catch语句，最后finally语句，无异常也会执行finally
    }

    try {
    } finally { // 有没有异常不关心，但出现异常之后我有机会先执行
    }

    try {
    } catch(XxxException e) { // 多个异常类型，可以合并为一个catch语句
    } catch(XxxException e) { // 存在继承关系的情况，子类放在前面，并且不能使用|合并
    }
```

应避免在 finally 语句中再次抛出异常，它会覆盖原有异常，也会覆盖 try 语句中的 return 语句。

## 4.2 自定义异常

异常都是被throw语句抛出来的，且只有 `Throwable` 类型的对象才可以被抛出。因此，继承某个已知异常或者 `Throwable` 就可以实现自定义。再重申一点：运行时异常还是 `Checked` 异常？

```
void throwRuntime() {
    throw new MyRunException("运行时遇到异常");
}

// 没有正确抛出异常，提示错误
void throwChecked() {
    throw new MyCheckedException("我要在编译时检查");
}

void throwError() {
    throw new MyError("运行时遇到错误");
}
```

上述代码，`throwChecked` 没显式声明要抛出的异常编译失败，正确的做法如下：

```
void throwChecked() throws MyCheckedException {
```

```
    throw new MyCheckedException("我会在编译时检查");
}
```

使用它的代码，必须使用 try...catch... 语句或者重新 throw 才能编译通过，也就是以下 2 种方式：

```
// 再次抛出异常
void testException1() throws MyCheckedException {
    throwChecked();
}

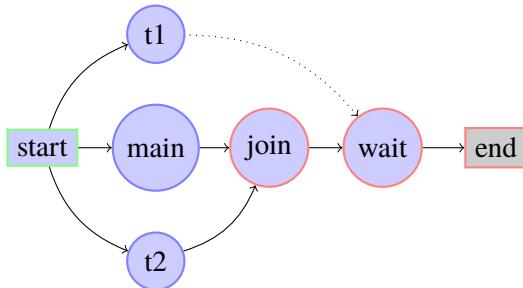
// 捕获异常并处理掉，或者再次throw e
void testException2() {
    try {
        throwChecked();
    } catch (MyCheckedException e) {
        e.printStackTrace();
    }
}
```

## 4.3 本章总结

通过本章的学习，可知异常分为 *Checked* 和 *UnChecked* 2 种。其中，*Checked* 异常需要显式处理，而 *RunTimeException* 和 *Error* 都不要求捕获。无论何种类型的异常，都是 *Throwable* 的子类，可以被抛出 *throw* 和捕获 *catch*。通常不建议捕获 *Error*，像内存不足、虚拟机错误之类的错误也许根本捕获不到。即使捕获到 *Error* 也不能阻止程序（线程）的崩溃，若只是多线程的一个线程崩溃了并不是大问题。你还可以做拯救你的软件，即使线程栈溢出了，也还可以重新再启动一个。

# 第5章 多线程

上一章，不建议捕获 Error 类型的异常，对于单线程应用也没有实际意义。但多线程的情况下，你不仅可以捕获，而且还能挽救业务。这是为什么呢？现在，我们就带着这个问题，开始学习多线程的知识。



上图，是一个多线程示意，包含 3 个线程。其中，中间那条执行线，称之为为主线程。上下执行线是 2 个单独的线程。多个线程一起工作，就需要一种互相协助的方法。使用 join 可以使主线程等待 t2 执行完毕，再继续往后执行；使用 wait 可以主线程等待 t1 执行完毕的通知，再继续往后执行。线程在执行过程中，还可以 sleep 一段时间。<sup>1</sup>

线程可分为：主线程、工作线程。其中，主线程负责 UI 的交互和刷新。程序长时间等待不响应，就是你主线程卡住了。所以，要避免在主线程消耗 CPU，应该创建一个工作线程处理下载任务、读写大文件等。再创建一个线程很简单，掌握同步和异步的用法才是本章的重点。

## 5.1 创建线程

通常，执行一个命令行程序，输出结果就退出了。譬如，`dir` 打印当前文件列表就结束了，而启动 `python` 就会进入交互模式，直到你显式请求退出。

主线程启动业务线程之后，就开始了各自的生命周期。即使主线程先退出 `main` 函数，也还会等子线程结束才算程序退出。这是线程的第一种用法：创建并启动线程，相互独立运行。

```
Section_5_1: main exit!
Section_5_1: end of main!
Section_5_1: thread exit!
```

<sup>1</sup>睡眠时间长度不保证精确。

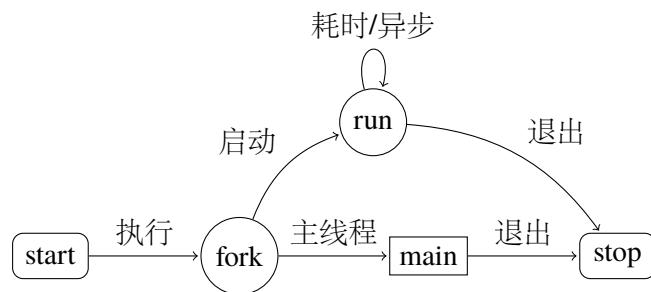


图 5.1: 主线程和工作线程

在主线程退出之后，就不能再响应用户的输入和点击。英文单词 *Thread* 是线程的类名，创建对象是 `t=new Thread()`，启动它用 `t.start()`。很显然这样一个线程什么都做不了。由面向对象知识，可通过 `@Override run` 方法重写线程代码。

---

```

// 1. 匿名方式继承Thread重写run方法
Thread t1 = new Thread() {
    @Override
    public void run() {
        // do something
    }
}

// 2. 匿名方式实现Runnable
Thread t2 = new Thread(new Runnable() {
    @Override
    public void run() {
        // do something
    }
})

// 3. lambda方式实现Runnable
Thread t3 = new Thread(() -> {
    // do something
})

```

---

上述三种方式，`t1` 和 `t2` 很相似但不同，而 `t3` 最简单。在 JDK8 之前，最常使用第 2 种方式创建线程。

## 5.2 线程协作

在本节内容开始之前，有必要先认识一下 Java 是如何执行的。首先把 Java 文件编译成 class 或者 jar，再启动 JVM 把它们加载到内存，找到主类从它的 main 函数开始执行的。进程（Process）是操作系统分配资源（内存、CPU）的最小单位，所有的线程都运行在该进程空间中。你创建的对象都被放在堆内存中，所有的线程都可以访问它。

实际上，内存被划分为若干个区域：堆区、栈区和方法区等。其中堆区是所有线程

共享的，也是所有对象的存放地。而栈主要是用来执行代码的，在函数调用过程中参数传入弹出，都是在这个栈里面。很显然，栈空间就是存放临时数据的，用于准备参数给函数调用。使用完之后，就会规划栈区空间。所以栈是一个来回复用的地方，而我们的局部变量和参数一样，也是在栈区，函数返回之后，局部变量也就出栈了。但是那个内存地址的数据，可能没有擦除。这就要求我们，定义的局部变量一定要初始化！

栈的这种特性，决定了它不需要分配太多的内存。但如果递归调用或者函数调用层次太深，必将出现栈溢出！每次函数调用，都要把参数和函数的局部变量入栈，生成一个栈帧。所以，在使用递归的时候，务必要注意不要太深（大约 1W 次）。

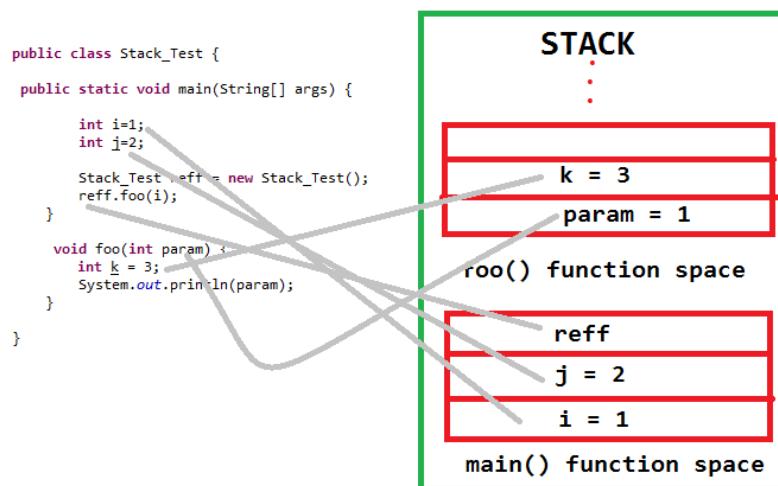


图 5.2: 线程栈帧

每个线程都有自己的栈空间，不是共享的。在 Java 中，创建的对象都在堆空间里，而栈空间基本都是对象的引用。上图中，栈中的 name 指向外部的字符串对象。每个线程都有各自的栈空间，互相独立使用。

### 5.2.1 wait 和 notify

等待（wait）和通知（notify），是常见的多线程同步方法，比起轮询要好。尤其是，有好几个线程的时候，最好使用这种机制（或者同步块）。从字面意思理解，也很容易使用。当缺少继续运行条件的时候，就让当前线程进入等待（wait）状态，而其它线程致使条件成立之后 notify 它。假设，现在有 T1、T2、T3、T4 四个线程，其中 T1、T2、T3 都在等待条件 C1，只有 T4 处理完才会发布 C1 条件。

1. 锁对象，任何 Java 对象或类，都可以作为锁。

---

```

void print(String info) {
    System.out.println(info);
}

private Object C1 = new Object();

```

```

void doLocked(Object lock) {
    try {
        lock.wait();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

---

2. 创建多个线程，等待 C1 的唤醒。

```

Thread T1 = new Thread(() -> {
    print("T1:进入");
    synchronized (C1) {
        print("T1:process");
        doLocked(C1);
    }
    print("T1:完成");
});;

Thread T2 = new Thread(() -> {
    print("T2:进入");
    synchronized (C1) {
        print("T2:process");
        doLocked(C1);
    }
    print("T2:完成");
});;

```

```

Thread T3 = new Thread(() -> {
    print("T3:进入");
    synchronized (C1) {
        print("T3:process");
        doLocked(C1);
    }
    print("T3:完成");
});;

Thread T4 = new Thread(() -> {
    print("T4:进入");
    synchronized (C1) {
        sleep(3000);
        C1.notify();
    }
    print("T4:退出!");
});;

```

---

3. 启动所有线程，T1、T2、T3 虽然会先执行，但很快都会处于 wait 状态。

```

T1.start();
T2.start();
T3.start();
T4.start();

```

---

上述例子，使用 C1 作为锁，也可以使用 C1=类.class，譬如 C1=Object.class。使用 notify 会随机唤醒一个线程，由于选择是任意性的，在设计 JAVA 程序时，不要依赖于这一点。notifyAll 可唤醒所有等待 C1 的线程并允许它们获得对象锁，但并不是同时获得执行权。synchronized 块中的代码，只能有一个线程运行。所以，唤醒一个线程就是允许这个线程获得 C1 对象锁并继续往下运行。notifyAll 对每个 wait 对象顺序调用一次 notify，逐个唤



醒线程。

值得注意的是，执行 `wait` 之前必须先获得对象锁，必须先进入 `synchronized` 代码块。调用 `wait` 之后，就会让出 C1 给其它线程抢占。但是上述例子，T1 即使让出 C1 锁，线程 T2 和 T3 已经提前进入等待队列，不会查询 C1 锁的状态，只有使用 `notify`、`notifyAll` 才能唤醒它们。

使用 `synchronized` 的时候，代码块要尽量小。它不仅可以包含代码块，还可以放在函数名前面保护整个函数，这就是比较大的同步块。

```
synchronized void doLocked(String info) {  
    try {  
        print(info);  
        wait();  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}  
  
Thread T1 = new Thread(() -> {  
    print("T1:└enter");  
    doLocked("T1:└process");  
    print("T1:└done");  
});  
  
Thread T4 = new Thread(() -> {  
    synchronized (this) {  
        sleep(3000);  
        notify(); // this.notify();  
    }  
    print("T4└exit!");  
});
```

从 JDK5 开始，新增的 `java.util.concurrent` 包提供了更多的同步机制，譬如 `ReentrantLock`、`ReadWriteLock`。使用 `Lock` 不需要 `synchronized`，但需要你主动释放锁。

```
private Lock lock = new ReentrantLock(); // 锁对象  
....  
lock.lock(); // 得到锁  
....  
lock.unlock(); // 释放锁
```

限于篇幅不再展开，通常使用 `synchronized` 就可以解决所有同步问题。

## 5.3 线程池

创建线程后，并做一系列的初始化，肯定会消耗不少的时间。尤其是网络读写和数据库初始化可能要十几秒才能完成。这对时间有严苛要求的任务，是无法接受的。所以，需要一种复用线程的方案，随时都能有一个准备的线程。只要把 Task 递给线程池，它就会线程池中，挑选一个空闲的执行任务，除非线程池是空的才会新建工作线程。并且，新建的线程用完之后，也会放到线程池里面。

对于刚接触编程的同学，实现一套线程池代码难度很高。在 `java.util.concurrent` 包中，JDK 提供了 `ThreadPool` 实现，其中， `ThreadPoolExecutor` 是线程池的核心类，

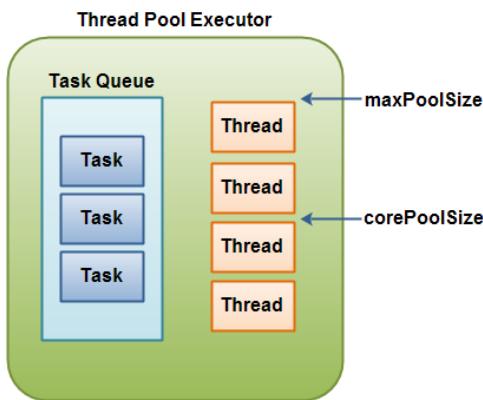


图 5.3: 线程池

如上图 ??，线程池有 3 个核心概念：`corePoolSize`、`maxPoolSize` 和 `TaskQueue`。其中，`corePoolSize` 代表初始准备的线程数，也是最少的线程数；`maxPoolSize` 代表最多可以保留多少个线程；`TaskQueue` 很显然是任务数超出 `maxPoolSize` 的时候，Task 就要入队等待。

```
pool.schedule(() -> print("Task1\u00d7delay\u00d73\u00d7seconds"), 3, TimeUnit.SECONDS);
pool.schedule(() -> print("Task2\u00d7delay\u00d72\u00d7seconds"), 2, TimeUnit.SECONDS);
pool.schedule(() -> print("Task3\u00d7delay\u00d71\u00d7seconds"), 1, TimeUnit.SECONDS);
pool.schedule(() -> print("Task4\u00d7delay\u00d74\u00d7seconds"), 4, TimeUnit.SECONDS);
pool.schedule(() -> print("Task5\u00d7delay\u00d75\u00d7seconds"), 5, TimeUnit.SECONDS);
pool.schedule(() -> print("Task6\u00d7delay\u00d76\u00d7seconds"), 6, TimeUnit.SECONDS);
```

在主线程退出 `main` 函数之后，线程池并不会退出，除非你主动 `pool.shutdown()`；才能看到 `Process finished with exit code 0`。

## 5.4 总结

限于本书篇幅，不能覆盖所有多线程的知识，通过本章的学习至少可以达到管中窥豹的效果。线程池在高性能、数据处理和 Web 开发用的比较多，初学 Java 多线程，掌握 `Thread` 和 `Synchronized` 就可以了。

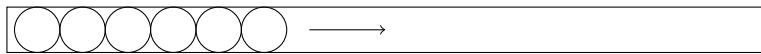
# 第6章 输入/输出

---

所有程序都有输入和输出过程，它可以是显示器打印，也可以是键盘输入，还包括网络的数据传输等。单存谈数据只有 byte 或二进制的形式，所谓的字符编码都是建立在它之上的。

## 6.1 输入/输出流

数据以二进制的形式，在网络上传入或在磁盘上保存。读写文件就像使用一根水管一位一位的传输数据，通常形象化为“流”(Stream)。



使用记事本打开文件，首先从磁盘读取文件数据流，它是无编码的二进制数据（最小单元为 byte）。记事本程序根据操作系统的默认文字编码，或者智能判断出文件编码，就可以显示文字了，

## 6.2 BIO (Blocking IO)

读取文件数据之后，才能处理数据。如果是单线程的代码，肯定是要等待完成数据读取才能往下执行，不然后面的代码没有数据可以处理。所以这个过程是阻塞的，称之为 BIO (Block IO)。

JDK 最早提供的文件读写方式，是基于操作系统的文件 IO 接口，没有提供异步的方式。需要开发者创建一个 IO 线程读写和处理数据。文件类 File 提供了最基本的封装，使用 File 可以删除、创建和遍历文件，也可以检查文件的可读写权限。

---

```
File file = new File("./");
String[] list = file.list(); // 当前目录下的文件
```

---

文件输入流 (FileInputStream) 就是从文件读取数据，而输出流 FileOutputStream 就是写文件。

### 6.2.1 字符编码

打开记事本，写入“012345”保存。注意一下文件的编码，本书使用的是 UTF-8，然后用 Hex 方式查看文件内容，显示如下：

---

```
00000000: 3031 3233 3435
```

---

创建并写入文件内容，可以使用 FileOutputStream 类提供的接口。大家要学会查看 API 列表，选择一个合适的写文件函数，传入正确的参数。

```

File file = new File("/Users/simbaba/Desktop/012345.txt");

try (FileOutputStream fos = new FileOutputStream(file)) {
    fos.write(new byte[]{0, 1, 2, 3, 4, 5});
} catch (IOException e) {
    e.printStackTrace();
}

```

FileOutputStream	
public void write(byte b[])	写字节数组
public void write(int b)	注意：还是写 1 个字节
public void close()	关闭文件释放文件资源
public FileChannel getChannel()	获取文件通道

表 6.1: FileOutputStream 写文件接口

执行以上代码，再次打开文件，使用 Hex 的方式查看，内容如下：

00000000: 0001 0203 0405

由以上差异，可见所谓的“012345”和实际的数据“012345”是由差别的。UTF-8 编码的“012345”，实际上对应 ASCII 码表中的值：

十进制	十六进制	字符
48	30	0
49	31	1
50	32	2
51	33	3

所以，读写文件的方式分为：文本、二进制方式。采用文本方式读写文件的时候，如果遇到无法解码的数据，就会显示为乱码。通常你的笔记，都是以文本的方式读写的，而图片和视频都是二进制的方式存储。

同样内容不同字符编码，存到文件中后的二进制数据也有可能是不一样的。对于英文字符，使用一个字节的 ASCII 码就足够用了，然而其它国家或地区的文字符号，通常超过一个字节所能表示的范围。Java 的默认字符类型 char 占用 2 个字节，使用 UNICODE 编码。但实际上，UNICODE 仅是文字集的编码，并不用来存储和传输用。对于英文字符，也使用 2 字节（16bit）表示，将会有许多的 0，譬如数字 0 可表示为：0x0030。

Unicode 编码：0x00000030

UTF8 编码：0x30

UTF16 编码：0xFEFF0030

UTF32 编码：0x0000FEFF00000030

所以，编码和字符集的对应关系如果搞错，就会出现乱码的情况。不同的字符编码有一定的规律，譬如 FEFF 标志、最高位为 1 等。记住这些特点，可用来自动检测数据流/文

件流的编码，才能正确显示文本内容。

```
try (FileInputStream fis = new FileInputStream(file)) {
    int len = fis.available();
    byte[] bytes = new byte[len];
    fis.read(bytes);

    System.out.println(Hex.encodeHexString(bytes));
    String text = new String(bytes, Charset.forName("UTF-8"));
    System.out.println(text);
} catch (IOException e) {
    e.printStackTrace();
}
```

以上代码输出：e88194e9809a (UTF-8 字符编码)、联通 (实际文字内容)。

## 6.2.2 IO 架构

在面向对象章节，曾经介绍过 IO 的架构，涉及到的对象类型和继承关系。图 6.1 只含有字节输入的相关类，输出和此类似。

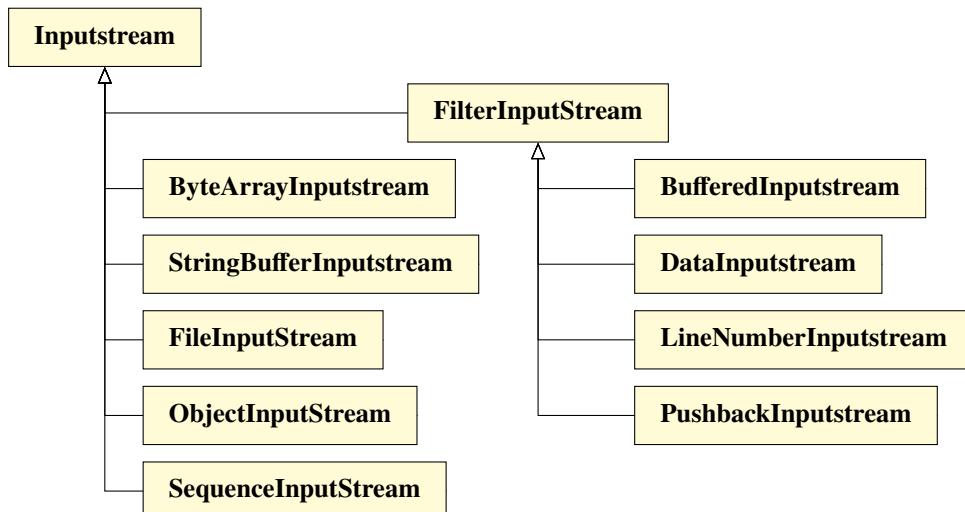


图 6.1: IO 继承图

最基本最常用的是 `FileInputStream`，其它根据需要合理选择。输入流（InputStream）不仅限于文件，也可以是从数组、字符串甚至是对象。作为（as a）一个输入流，就是要实现 `InputStream` 的接口。我们可以实现一个随机数输入流，每次 `read` 都返回一个随机整数，如下：

```
//仅仅是示例代码
RandomInputStream ris = new RandomInputStream();
ris.setRange(1, 10); // 设置范围
```

---

```
int n = ris.read(); // 返回一个随机数
```

---

如上代码，RandomInputStream 是一个无限流，没有文件结尾。图 6.1 中，每个输入流的具体用法，可参考下表：

IO 类	作用	示例
ByteArrayInputStream	包装一个 byte[] 作为输入	b = new byte[]{100,2,3,4}; bInput = new ByteArrayInputStream(b); int n1 = bInput.read(); //100 int n2 = bInput.read(); //2
StringBufferInputStream	把字符串作为输入	bInput = new StringBufferInputStream("123456"); int n1 = bInput.read() - '0'; //1 char n2 = (char) bInput.read(); //2
FileInputStream	读文件	省略。
ObjectInputStream	从流中获取一个对象	参考代码见后
SequenceInputStream	合并流	参考代码见后

ByteArrayOutputStream 的参考代码：

---

```
Student student = new Student(10, "jim");

ByteArrayOutputStream bai = new ByteArrayOutputStream(100);
ObjectOutputStream oos = new ObjectOutputStream(bai);
oos.writeObject(student);
oos.close();
byte[] dst = bai.toByteArray();

ByteArrayInputStream bais = new ByteArrayInputStream(dst);
ObjectInputStream bInput = new ObjectInputStream(bais);
Student object = (Student) bInput.readObject();
System.out.printf("name=%s,age=%d", object.name, object.age);

// SequenceInputStream 示例
byte[] b1 = new byte[]{1, 2, 3, 4};
ByteArrayInputStream bis1 = new ByteArrayInputStream(b1);
byte[] b2 = new byte[]{10, 20, 30, 40};
ByteArrayInputStream bis2 = new ByteArrayInputStream(b2);

SequenceInputStream sis = new SequenceInputStream(bis1, bis2);
System.out.println("a=" + sis.available()); // 4
```

```
int b;
while ((b=sis.read()) != -1) {
    System.out.println(b); // 1 2 3 4 10 20 30 40
}
```

关于 Object 对象串行化，需要注意serialVersionUID 和transient 的作用。不同 VersionUID 的数据文件，是不能再读取成对象实例的。在 Student 中添加一个 sex 字段，修饰为transient 就不会保存到文件中了。如果保存的时候 serialVersionUID=1，后来修改为 2L 就会导致加载对象失败。

```
class Student implements Serializable{
    static final long serialVersionUID = 2L;
    int age;
    String name;
    transient boolean sex;
    Student(int age, String name){
        this.age = age;
        this.name = name;
        this.sex = true;
    }
    FileInputStream fis = new FileInputStream("student.txt");
    ObjectInputStream bInput = new ObjectInputStream(fis);
    Student o = (Student) bInput.readObject();
    // 打印o.sex=false而不是true
    System.out.printf("name=%s,age=%d,sex=%b", o.name, o.age, o.sex);
```

对传统的读写方式介绍就介绍这么多，只要能掌握 FileInputStream 的用法，其它都与此相似，因为都是复用的 InputStream 的接口。最简单的解决办法，就是单独创建一个 IO 线程，在这个线程里完成数据的读写和处理。

对于字符流有 Reader 和 Writer 两个基类，提供了最基本的读写接口。谈到字符就离不开编码，不是所有的文件都能显示为可读内容。譬如，你打开一张图片，尽管也能看到一些汉字，但都是乱七八糟的还夹杂着很多未知符号。如下图 6.2 和字节流有很相似的类继承结构，主要的差别是 Reader/Writer 依赖字节流完成读写，所以最底层还是先读入字节然后根据字符编码解析文字。

```
FileWriter fw = new FileWriter("student.txt");
fw.write("张三, 10岁");
fw.close();

FileReader fr = new FileReader("student.txt");
BufferedReader br = new BufferedReader(fr);
```

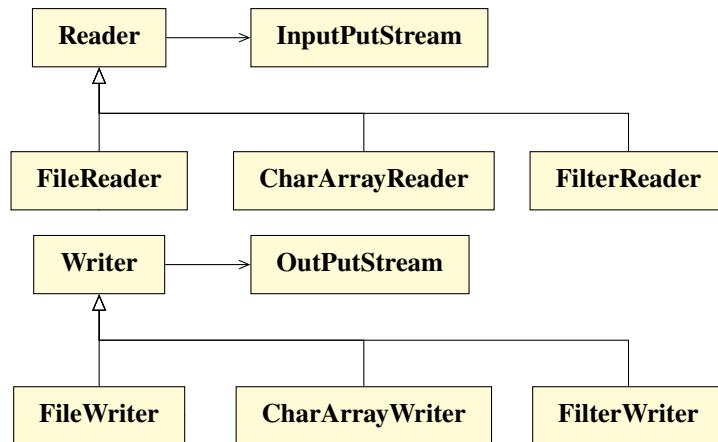


图 6.2: 字符流架构

---

```

String text = br.readLine();
System.out.println(text);
fr.close();

```

---

以上这段代码，使用的系统默认的编码。如果把上述文件另外为 GBK 编码，上述代码就会显示乱码甚至异常。

---

```

String str = "张三, 10岁";
byte[] bytes = str.getBytes("GBK");

System.out.println(Hex.encodeHexString(bytes));
String text = new String(bytes, Charset.forName("UTF-8"));
System.out.println(text);
// 输出:
// d5c5c8fda3ac3130cbea
// ??????10??

```

---

## 6.3 Net 编程和 AIO (Asynchronous IO)

IO 的阻塞读写方式，在很多场合都不适用，它会让当前线程甚至程序卡住。举一个简单的例子：从大于 1G 的文件里面查找数据，或者从网络下载一个页面。网络程序也提供了 InputStream、OutputStream 对象，有了它们就能像文件 IO 一样上传和下载数据。

---

```

Socket socket=new Socket(InetAddress.getByName("www.baidu.com"),80);
OutputStream os=socket.getOutputStream();
OutputStreamWriter osw=new OutputStreamWriter(os);
BufferedWriter bw=new BufferedWriter(osw);

//最基本的请求头
bw.write("GET\u002fHTTP/1.1\r\n");

```



```
bw.write("Host:www.baidu.com\r\n\r\n");
bw.flush();
socket.shutdownOutput();

//读取response
InputStream is=socket.getInputStream();
BufferedReader br=new BufferedReader(new InputStreamReader(is));
String str=null;
while((str=br.readLine())!=null){
    System.out.println(str);
}
socket.close();
```

上述下载 Web 页面的代码，用到了 HTTP GET 指令。实际下载代码，可以直接使用 HttpClient 或者 URL 类型提供的接口。

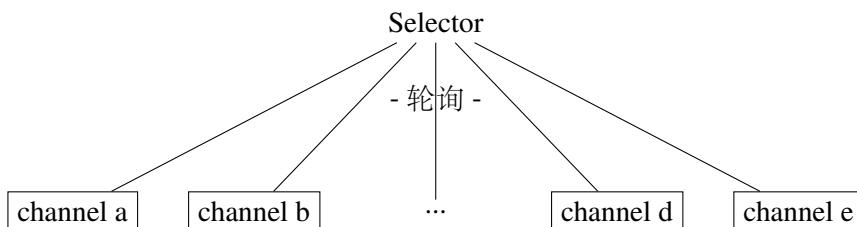
```
// 注意带上http以便URL识别协议
URL url=new URL("http://www.baidu.com");
InputStream is= url.openStream();;

//读取response
BufferedReader br=new BufferedReader(new InputStreamReader(is));
String str=null;
while((str=br.readLine())!=null){
    System.out.println(str);
}
is.close();
//-----
HttpGet http=new HttpGet("https://www.baidu.com");
try {
    CloseableHttpClient httpClient = HttpClients.createDefault();
    CloseableHttpResponse response = httpClient.execute(http);
} {
    HttpEntity entity = response.getEntity();
    String page = EntityUtils.toString(entity, "UTF-8");
    System.out.println(page);
} catch (IOException e) {
    e.printStackTrace();
}
```

网络不畅会导致上传、下载阻塞，所以不适合在主线程做。对于 UI 交互类应用，主线程要能响应用户操作，即使在等待 Web 网页下载也不能卡住用户的各种点击，尤其是“取消”。

## 6.4 NIO (New/Non-Blocking IO)

NIO(New Input/ Output) 是在 JDK1.4 引入的基于通道和缓冲区的 I/O 方式，它是一种非阻塞的 IO 模型，使用 Selector 轮询 IO 事件，但不阻塞当前线程往下执行。Selector 作为选择器，也称为多路复用器，是 NIO 的核心组件，用于检查一个或多个 NIO Channel(通道) 是否处于可读、可写，由此实现了单个线程管理多个 Channels，同时管理多个网络链接。另外，NIO 可使用 Native 函数直接分配堆外内存，避免在 Java 堆和 Native 堆中来回复制数据。



如上图，Selector 往往是一个无限循环，检查每个 channel 的状态。这些状态包括：

状态	说明
OP_READ	通道中已经有了可读的数据
OP_WRITE	已经可以向通道写数据了
OP_ACCEPT	监听到了连接，可以接收这个连接了
OP_CONNECT	连接已经建立成功
注意：注册了写事件，不在合适的时间去除掉，会一直触发写事件	

### 6.4.1 FileChannel

FileChannel 是面向文件的通道，但无法设置为非阻塞模式，只能是阻塞模式。值得注意的是 FileChannel 只是一个抽象类，提供了读、写、映射和锁住一个文件的抽象方法。不过 FileChannel 是线程安全的，可以被多个并发线程使用。

FileChannel 对象可通过 open() 方法被创建，也可通过 FileInputStream、 FileOutputStream、RandomAccessFile 对象的 getChannel() 得到。

```

Path path = Paths.get("/Users/simbaba",
                      "Desktop",
                      "test.txt");
//Files.deleteIfExists(path);
//Files.createFile(path);
FileChannel fc = FileChannel.open(path, StandardOpenOption.CREATE,
                                   StandardOpenOption.WRITE);

ByteBuffer bb = ByteBuffer.allocate(10);
bb.put(new byte[]{'1','2','3',4,5,6,7,8,9,10});
bb.flip(); // 非常关键，不然什么都没写！！！
fc.write(bb);
  
```

```
fc.close();
```

## 6.5 异步 Channel

NIO 提供的 Selector 作为“多路复用器”，用来检查一个或多个 NIO Channel（通道）的状态是否处于可读、可写。以此实现单线程管理多个 Channels，譬如可以管理多个网络链接。通过调用 Selector.open() 方法创建一个 Selector 对象，其它非阻塞<sup>1</sup> 的 Channel 主动注册到该 Selector 对象上。通过 Channel 的 configureBlocking () 方法，使通道处于阻塞模式或非阻塞模式。

---

<sup>1</sup>FileChannel 不适用 Selector，因为它没有继承 SelectableChannel。

## 第7章 Lambda 表达式

---

Lambda 表达式是 Java8 引入的一个重要新特性，允许你通过表达式来代替函数。用起来和方法一样，有一个参数列表和使用这些参数的代码。Lambda 对集合做了必要扩展，使用流 (stream) 遍历和操作数据。先看几个例子：

```
// 1. 无参数，返回值为7
IntSupplier fn = () -> 7;

// 2. 接收一个参数(数字类型),返回它的2倍
IntFunction<String> fx = x -> (2 * x) + "$";

// 3. 接受2个参数(数字),并返回他们的乘积
IntBinaryOperator f_xy_mul = (x, y) -> x * y;

// 4. 接收2个int型整数,返回他们的和
IntBinaryOperator f_xy_add = (int x, int y) -> x + y;

// 5. 接受一个 string 对象,并在控制台打印,不返回任何值(看起来像是返回void)
Consumer<String> print = (String s) -> System.out.print(s);

int n = fn.getAsInt();
String d = fx.apply(3);
int mul = f_xy_mul.applyAsInt(2, 3);
int add = f_xy_add.applyAsInt(2, 4);
print.accept(String.format("%d,%s,%d,%d", n, d, mul, add));

\\输出： 7, 6$, 6, 6
```

### 7.1 函数分类

$\lambda$  表达式由来已久，并不是 Java 提出的概念，在计算机诞生之初就出现了。这种  $\lambda$  表达式不仅自然，而且短小精悍。设计之初，原本  $\lambda$  表达式是线程安全的，因它不会产生副作用（影响外部的值，side effect）。

分类	说明
----	----

<b>Function</b> <ul style="list-style-type: none"> <li>• R apply(T t...);</li> <li>• applyAsInt(double);</li> <li>• applyAsLong(double);</li> </ul>	<p>运算, 有入参有返回值</p> <ol style="list-style-type: none"> <li>1. BiFunction&lt;T, U, R&gt; 参数 T 和 U 类型, 返回 R 类型的值</li> <li>2. DoubleFunction&lt;R&gt; 返回 R 类型的值</li> <li>3. DoubleToIntFunction 类似于类型转换</li> <li>4. DoubleToLongFunction 类似于类型转换</li> <li>5. IntFunction&lt;R&gt; 一个整数参数返回 R 类型的值</li> <li>6. IntToDoubleFunction 类似于类型转换</li> <li>7. IntToLongFunction 类似于类型转换</li> <li>8. LongFunction&lt;R&gt; 一个长整数返回 R 类型的值</li> <li>9. LongToDoubleFunction 类似于类型转换</li> <li>10. LongToIntFunction 类似于类型转换</li> <li>11. ToDoubleBiFunction&lt;T, U&gt; 接收 T 和 U 类型参数 返回实数</li> <li>12. ToDoubleFunction&lt;T&gt; 类似于强制转换</li> <li>13.ToIntBiFunction&lt;T, U&gt; 收 T 和 U 类型参数返回整数</li> <li>14.ToIntFunction&lt;T&gt; 类似于强制转换</li> <li>15. ToLongBiFunction&lt;T, U&gt; 收 T 和 U 类型参数返回长整数</li> <li>16. ToLongFunction&lt;T&gt; 类似于强制转换</li> </ol>
<b>Consumer</b> <ul style="list-style-type: none"> <li>• accept(T t, U u);</li> <li>• andThen(after);</li> </ul>	<p>有参无返回值</p> <ol style="list-style-type: none"> <li>1. Consumer 一个参数无返回值</li> <li>2. BiConsumer 两个参数无返回值</li> <li>3. DoubleConsumer 实数的特例</li> <li>4. IntConsumer 整数特例</li> <li>5. LongConsumer 长整数特例</li> <li>6. ObjDoubleConsumer 一个 Obj 和实数</li> <li>7. ObjIntConsumer 一个 Obj 和整数</li> <li>8. ObjLongConsumer 一个 Obj 和长整数</li> </ol>
<b>Operator</b> <ul style="list-style-type: none"> <li>• Function 接口;</li> <li>• minBy(Comparator);</li> <li>• maxBy(Comparator);</li> <li>• compose();</li> <li>• andThen();</li> <li>• identity();</li> </ul>	<p>操作, <i>Function</i> 的特殊类别</p> <ol style="list-style-type: none"> <li>1. BinaryOperator&lt;T&gt; 同 BiFunction&lt;T,T,T&gt;</li> <li>2. DoubleBinaryOperator 同 BiFunction&lt;T,T,T&gt;</li> <li>3. DoubleUnaryOperator 单目运算</li> </ol>

Predicate	谓词，用于真假判断 1. BiPredicate<T, U> 入参类型 T 和 U 条件检查 2. DoublePredicate 入参是实数类型 3. IntPredicate 入参是整数类型 4. LongPredicate 入参是长整数类型 5. Predicate 入参是 T 类型
Supplier	产生器，无入参有返回值 1. Supplier 返回一个值 2. BooleanSupplier 返回一个布尔值 3. DoubleSupplier 返回一个实数值 4. IntSupplier 返回一个整数值

除了以上已定义的  $\lambda$  表达式，你也可以自定义。实际上， $\lambda$  表达式都是用接口（interface）定义的。然而在使用的时候，函数名又被省略了，所以其中存在某种对应关系。那又怎么消除二义性的呢？这就要求接口中只能有一个未知函数。就像这样：

---

```
/// 定义Ixy接口
interface Ixy {
    int add(int x, int y);
    default int sub(int x, int y) {return x-y;}
}

/// 定义xy变量
Ixy xy = (a, b)-> a+b;
int sum = xy.add(1,2);
int sub = xy.sub(1,2);
System.out.printf("sum=%d, sub=%d", sum, sub);
```

---

注意default的特殊用法，代表的是该函数接口有默认实现，可定义多个 default 接口。

你也许发现了， $\lambda$  表达式根本不关心关联的函数叫什么名字，只关注名字以外的参数和类型是否一致。

---

```
// 借用Math类的addExact
Ixy xy = Math::addExact;
int sum = xy.add(1,2);
int sub = xy.sub(1,2);
// 借用System.out::printf
Consumer<String> print = System.out::printf;
print.accept(String.format("sum=%d, sub=%d", sum, sub));
```

---

## 7.2 应用示例

使用  $\lambda$  表达式可以极大地简化代码，也使代码更加容易理解。但  $\lambda$  表达式不要太复杂，建议功能单一。接下来，将针对每一种  $\lambda$  表达式，举例二三。

### 1. 过滤数组的例子

```
int[] ids = {0,1,2,3,4,5,6,7,8,9};

Consumer<String> print = System.out::print;
Consumer<String> println = System.out::println;
IntPredicate isOdd = (n) -> n%2==1;

IntConsumer c1 = (n)-> {
    if (isOdd.test(n))print.accept("单数");
    else print.accept("双数");
};

IntConsumer c2 = (n)-> {
    if (isOdd.test(n))println.accept("->" +n);
    else println.accept("~>" +n);
};

IntStream.of(ids).forEach(c1.andThen(c2));
```

这段过滤数组的例子，输出是应该怎样的呢？使用了 2 个 `Consumer` 连续处理数据，而 `IntStream` 提供了这种流式动力。从 `Stream` 中每取出一个数字，先给 `c1` 处理然后再交给 `c2` 处理。所以，以上代码的输出是：

```
双数=>0
单数->1
双数=>2
单数->3
双数=>4
单数->5
双数=>6
单数->7
双数=>8
单数->9
```

### 2. 读写文件

```
Path path = Paths.get("/Users/simbaba",
                      "Desktop",
                      "test.txt");
```



```
Files.lines(path).forEach(System.out::println);
```

使用 Files 轻松应对大多数的文件操作，借助  $\lambda$  表达式也非常简单。

### 3. 找出最大和最小

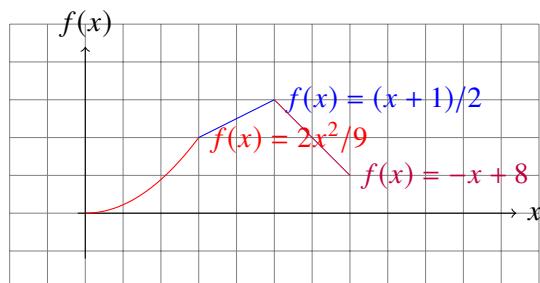
```
Integer[] ids = {11, 2, 6, 4, 7, 10, 5};
OptionalInt min = Stream.of(ids).mapToInt(id->id).min();
OptionalInt max = Stream.of(ids).mapToInt(id->id).max();

min.ifPresent(System.out::println);
max.ifPresent(System.out::println);
```

需要注意 Integer[] 和 int[] 的区别，同学们可以尝试改成 int[] 怎么才能获得结果，又为什么出现这种问题呢？

## 7.3 数学推理

$\lambda$  表达式，对数学是非常友好的，不必再写一堆函数或者分支语句，可以写出更自然的代码。



$$f(x) = \begin{cases} 2x^2/9 & x < 3, \\ (x+1)/2 & [3, 5] \\ -x + 8 & x >= 5 \end{cases}$$

参考代码：

```
DoubleUnaryOperator op1 = (d)->2*d*d/9;
DoubleUnaryOperator op2 = (d)->(d+1)/2;
DoubleUnaryOperator op3 = (d)->-d+8;
// 浮点数不建议直接比较大小，此处忽略
double d = 4.5d;
if (d > 5) {
    d = op3.applyAsDouble(d);
} else if (d > 3) {
    d = op2.applyAsDouble(d);
} else {
    d= op1.applyAsDouble(d);
}
System.out.println("d=" + d);
```

## 7.4 更加强大的集合

支持  $\lambda$  表达式的编程语言，提供的集合操作也都非常强大。限于篇幅，本节不能全部涵盖，只介绍基本的使用方法。

### 7.4.1 获得 stream

借助接口的 default 扩展语法，现在所有的 Java 集合都已经支持数据流 stream，用以遍历和操作数据。这里的 stream 和文件流 (file stream) 稍微有些差别，都是对集合的映射。

我们先看数组是如何转换成 stream 的，分为两种情况：基本类型和对象类型。处理方式上有一些区别，请注意使用上的差异。Java 是通过泛型实现的数据容器，其中的  $T$  类型指的是引用类型。以下是 int[] 和 Integer[] 的样例代码：

---

```
@Test
public void visitIntList() {
    int[] intList = {1,2,3,4,5};
    IntStream stream = IntStream.of(intList);
    int[] array = stream.map(n -> n * 2).toArray();
    assertArrayEquals(array, new int[]{2,4,6,8,10});
}

@Test
public void visitIntegerList() {
    Integer[] intList = {1,2,3,4,5};
    IntStream stream = Arrays.stream(intList).mapToInt(n->n);
    int[] array = stream.map(n -> n * 2).toArray();
    assertArrayEquals(array, new int[]{2,4,6,8,10});
}
```

---

与 List 不一样，Map 相当于有 2 个集合：keySet, values。遍历数据的时候，根据需要把 keySet 或 values 转换为 stream。使用示例如下：

---

```
@Test
public void map2Stream() {
    HashMap<Integer, String> n2ch = new HashMap<>();
    n2ch.put(1, "abcdefg");
    n2ch.put(2, "hijklmn");
    n2ch.put(3, "opqrst");

    Stream<Integer> keyStream = n2ch.keySet().stream();
    Object[] even = keyStream.filter(key -> key % 2 == 0).map(n2ch::get).
        toArray();
```



```
        assertEquals(even, new String[]{"hijklmn"});  
    }
```

上述代码，遍历 keySet 的时候先进行一次过滤，然后再进行一次 map 操作。通过 map 操作又把 key 遍历转换为 values 遍历，也就是经过 map() 操作之后，stream<Integer> 已经转换为了 stream<String>。而 n2ch::get 语法，是一种 lambda 表达式借用代码和省略参数的写法。

#### 7.4.2 统计功能

仅把 stream 当作 for 循环的替代品，只是用于遍历就有点大材小用了。对于函数式编程，比较有特色的一个操作 *reduce* 在此表现地淋漓尽致。

```
int[] intList = {1,2,3,4,5};  
IntStream stream = IntStream.of(intList);  
  
OptionalInt optSum = stream.reduce((a, b)->a+b);  
  
int sum = optSum.getAsInt();  
assertEquals(sum, 1+2+3+4+5);
```

使用 stream 还可以对数据进行分类集约，本书不作详细解释，仅提供演示代码。如下代码，根据性别把学生分为两类。

```
Student jim = new Student("jim", 1);  
Student jane = new Student("jane", 0);  
Student tom = new Student("tom", 1);  
Student sunny = new Student("sunny", 0);  
  
Student[] students = new Student[] {jim, jane, tom, sunny};  
  
Map<Integer, List<Student>> group = Stream.of(students).collect(  
    Collectors.groupingBy(s -> s.sex));  
  
List female = group.get(1);  
Student[] dst = new Student[] {jim, tom};  
assertEquals(female.toArray(), dst);
```

对于数据排序、求最大值/最小值和平均值，使用 stream 处理也都很简单直观。



### 7.4.3 并行处理

也可以把 stream 分成多个子流，同时在多个线程并发处理，充分发挥系统多核硬件性能。将一个顺序执行的流转变成一个并发的流只需要调用 parallel() 方法。先用不同的线程分别处理，最后再合并每个数据流的计算结果。

```
@Test
public void parallelSum() {
    int[] nums = {1,2,3,4,5,6,7,8,9,10};

    OptionalInt sum = IntStream.of(nums)
        .parallel()
        .reduce((a, b)->a+b);
    assertEquals(sum.getAsInt(), 55);
}
```

### 7.4.4 无限流

无论是 List 还是 Map，所能提供的数据都是有限的，并不是取之不竭。使用 stream 可以实现一种随取随提供的数据流。典型的应用场景，如随机数集合，每次都可以取不同的数据。

```
Random random = new Random(System.currentTimeMillis());
IntStream stream = random.ints(1000, 9999);
stream.limit(10).forEach(System.out::println);
```

如上代码，生成 1000 9999 之间的数据，取 10 个数据出来打印。

## 7.5 本章小结

很显然，使用 lambda 表达式和 stream 可以极大地简化程序。不过这不是 JDK8 语法糖，确实通过语法和语义手段实现了函数式编程。值得关注的是，lambda 表达式的上下文的依赖值得你去思考。还有很多的知识不能详尽列举，请自行查阅相关资料。

## 第二部分

# 数学基础与算法

AI 最近变得非常流行了，但它的历史已经有六十多年。在 1956 年的达特茅斯会议，首次提出了 AI 的概念，而在一年后，Rosenblatt 就提出了感知机（perceptron）模型。

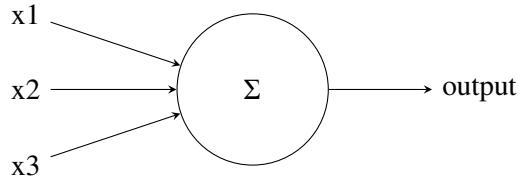


图 1：感知机

如上图 1，感知机就是二分类的线性分类模型，其输入为样本的特征向量，输出为样本的类别，取 +1 和 -1 二值。即通过某样本的特征，就可以准确判断该样本属于哪一类。顾名思义，感知机能够解决的问题首先要求特征空间是线性可分的，再者是二分类，即将样本分为 +1, -1 两类。

使用感知机可轻松解决异、或、非的问题，却存在一个致命缺陷：无法解决异或问题。导致 AI 进入第一个衰退期，直到 19 世纪八十年代，才逐渐复苏。1984 年，Hiton 教授提出 Boltzman 机模型。到 1986 年 Kumelhart 等人提出误差反向传播（Back Propagation）算法，简称 BP 网络。然而 90 年中期支持向量机（Support Vector Machines，SVM）强势登场，全方位碾压 NN。SVM 具有 1) 高效，可以快速训练；2) 无需调参，没有梯度消失问题；3) 高效泛化，全局最优解，不存在过拟合问题。几乎全方位的碾压 NN，NN 也再次被打入冷宫。

目前，BP 网络已成为广泛使用的网络。1987 年至今为发展期，神经网络受到各个国家的重视，纷纷展开研究，形成神经网络发展的另一个高潮。人工智能是一个将数学、算法和工程相结合的学科，包含微积分、概率论、统计学等数学知识。本章将为大家简要介绍人工智相关的数学基础理论。

## 第8章 微积分

微积分 (Calculus) 是数学的一个基础学科，包括极限、微分学、积分学及其应用。与机器学习相关的微积分问题有：极值问题、偏导数和梯度。在机器学习中，定义的所谓模型就是一个包含参数和特征的函数。定义如下：

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$$

其中， $\theta_i$  称为参数，而  $x_i$  称为特征， $h_{\theta}$  是预测模型。训练模型之前，都会采集很多组样本数据  $(x_1, x_2, \dots, x_n)$ ， $x_i$  就是预设的特征。模型的准确性，取决于特征的完备性和数据的充分性。取什么样的特征（采集什么样的数据）有时候也很难确定，过少的或过多的特征，都影响模型的准确性。

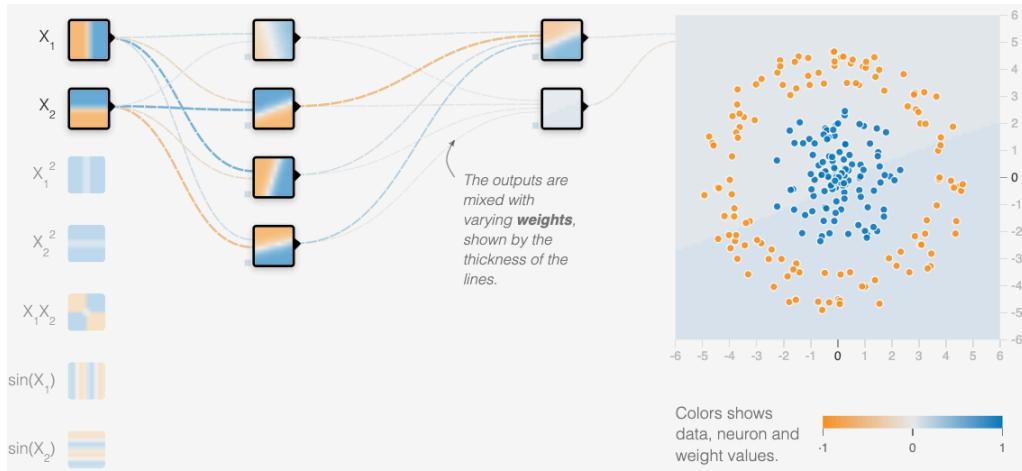


图 8.1: 线性特征分类

极端情况下，已有的样本数据少于特征的维度（相当于未知数多于方程个数），需要引入  $\lambda$  正则项。通过岭回归 (Ridge Regression) 得出最优的  $\lambda$  和  $\theta$ ，下一章线性代数部分会详细说明。

如果特征值  $(x_1, x_2, \dots, x_n)$  向量线性相关，就有必要减少特征的维度。很显然，特征空间的维度降低可以极大地简化模型。采集的样本数据通常都有一些噪点，要引入正则化降低它们对训练结果的影响，防止过拟合的出现。注意，欠拟合是指模型包含的特征维度低于现实情况，也无法训练出有效的结果。

由图 8.1 可知，两个线性特征  $x_1$  和  $x_2$  通过线性组合只能训练出多边形特征的模型。加入特征  $x_3$  并定义为  $x_1^2$ ，特征  $x_4$  定义为  $x_2^2$ ，改进后的模型分类预测更符合预期。实际上，需要哪些特征也并不是那么很明显。首先是我们能获取到哪些数据，主观判断是否相关。譬如，你可以拿到的数据：(年龄，性别，身高，体重，籍贯)，想用于预测某人的籍贯。

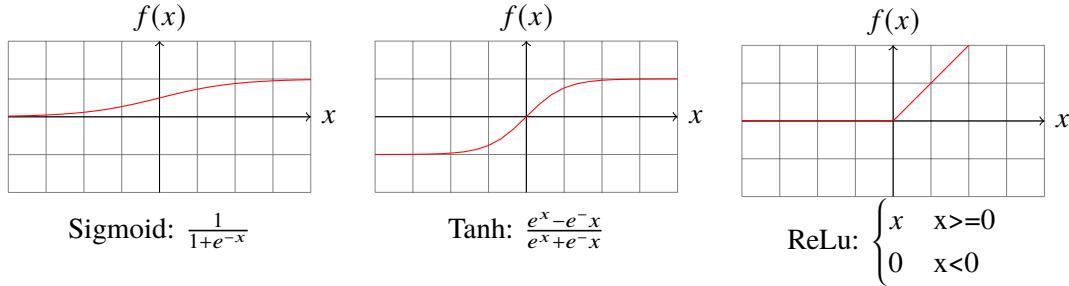
$$f(\text{年龄}, \text{性别}, \text{身高}, \text{体重}) \rightarrow \text{籍贯}$$

在定义模型之后，通常使用最小二乘法定义损失函数（代价函数，Cost Function）。然

后把准备好的大量样本喂给模型，进行迭代训练使其学习出最优的  $\theta$ ，使得代价函数的值尽量小。

$$J(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{2m} \sum_{i=1}^m (y^i - h_\theta(x^i))^2$$

在机器学习中，使用激活函数对结果进行逻辑分类：Sigmoid 函数，Tanh 函数，ReLU 函数。它们的主要区别是什么？请同学们仔细观察函数图形之间的共性和区别。



引入激活函数不仅用于逻辑分类，关键是还引入了非线性因素。Sigmoid 函数的输出在 (0,1) 之间，单调连续，并且容易求导。但它一旦输入落入饱和区，一阶导数就会接近于 0，容易产生梯度消失；Tanh 函数也是传统神经网络中常用的激活函数，与 Sigmoid 一样存在饱和问题。不过它的输出以 0 为中心，Tanh 函数看上去是放大的 Sigmoid 函数。

ReLU 是目前用的最多也是最受欢迎的激活函数，可加速 SGD(梯度下降算法) 的收敛。但 ReLU 随着训练的进行，部分输入也会落到硬饱和区。除了 ReLU，还有许多 ReLU 衍生的激活函数，比如：Leaky ReLU、ReLU6、SReLU、PReLU、RReLU、CReLU。

## 8.1 偏导数与梯度

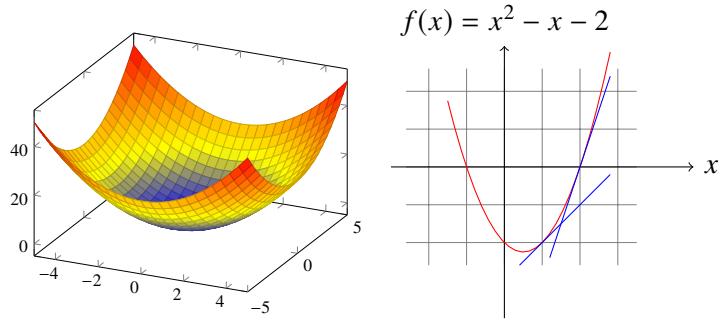
在微分中，导数表示函数的变化率，也是变化率最大的方向。梯度是全部偏导数构成的向量，一般将函数  $f$  的梯度记为  $\Delta f$ 。根据方向导数的知识，可知梯度的方向即为函数值增长最快的方向。

$$\text{grad}f(x_0, x_1, \dots, x_n) = \left( \frac{\delta f}{\delta x_0}, \frac{\delta f}{\delta x_1}, \dots, \frac{\delta f}{\delta x_n} \right)$$

函数在某一点的梯度，是一个向量，与最大方向导数的方向一致，而它的模就是方向导数的最大值。既然，沿梯度方向具有最大的变化率，所以沿着负梯度方向去减小函数值，才能最快达到优化目标。

梯度下降就像是你低头只看脚下的下山过程，若想最快到达山底，最好每一步都找最陡峭的方向走，这就是梯度的负方向。由于视野的局限性，使用梯度下降也有可能落入局部最小值。

例子函数  $f(x) = x^2 - x - 2$ , 梯度:  $f'(x) = 2x - 1$



假设  $\alpha=0.2$  可以这样推算：设起点为  $(4, 0)$

```

1次: 参数x=4, 4-0.2*(2*4-1)=2.6
2次: 参数x=2.6, 2.6-0.2*(2*2.6-1)= 1.76
3次: 参数x=1.76, 1.76-0.2*(2*1.76-1)=1.256
4次: 参数x=1.256, 1.256-0.2*(2*1.256-1)=0.9536
5次: 参数x=0.9536, 0.9536-0.2*(2*0.9536-1)=0.77216
.....
After 13 steps: x = 0.502743, f(x)= -2.249992.
After 14 steps: x = 0.501646, f(x)= -2.249997.
After 15 steps: x = 0.500987, f(x)= -2.249999.
After 16 steps: x = 0.500592, f(x)= -2.250000.
After 17 steps: x = 0.500355, f(x)= -2.250000.
After 18 steps: x = 0.500213, f(x)= -2.250000.

```

## 8.2 线性拟合

样本的散点图有助于发现关键特征，建立正确的模型。如下图，可直观地识别出线性关系，尽管有干扰数据。

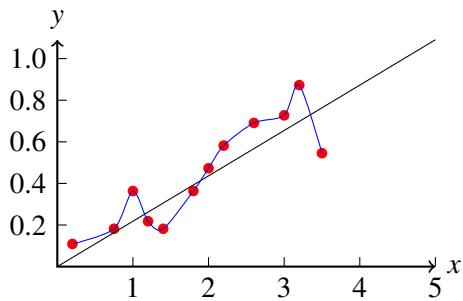


图 8.2: 线性拟合

拟合指的是你逼近目标函数的远近程度。欠拟合，是指模型复杂度过低，不能很好的拟合所有的数据，训练误差大，在训练和预测时表现都不好。欠拟合很容易被发现，在训练的时候就表现很差。过拟合，是指模型复杂度很高，但训练数据很少，导致训练误

差小，测试误差也大。过拟合，过度地学习训练数据中的细节和噪音，以至于模型在新的数据上表现很差，泛化性能变差。<sup>1</sup>

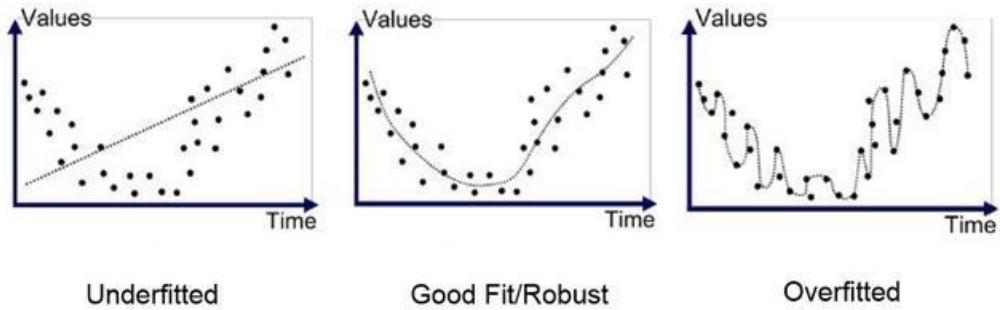


图 8.3: 拟合情况分类

### 8.2.1 最小二乘法

最小二乘法是勒让德 (A. M. Legendre) 于 1805 年在其著作《计算彗星轨道的新方法》中提出的。它的主要思想就是求解未知参数，使得理论值与观测值之差（即误差，或者说残差）的平方和达到最小。

$$E = \sum_{i=1}^n (y_i - \hat{y})^2$$

## 8.3 优化问题

拉格朗日乘子法 (Lagrange Multiplier)，用于求解带等式约束的极值问题，而 KKT (Karush Kuhn Tucker) 条件是拉格朗日乘子法的推广。在有等式约束时使用拉格朗日乘子法，在有不等约束时使用 KKT 条件。

### 8.3.1 拉格朗日乘子

拉格朗日乘子法是一种寻找多元函数在其变量受到一个或多个条件的约束时的极值的方法。设目标函数为  $f(x)$ ，约束条件为  $h_k(x)$ ，其中 s.t. 表示 subject to，“受限于”的意思， $l$  表示有  $l$  个约束条件。定义如下：

$$L(x, \lambda) = f(x) + \lambda h_k(x)$$

$$\min f(x) \quad | \quad \text{s.t. } h_k(x) = 0 \quad k = 1, 2, \dots, l$$

1. 构造拉格朗日函数
2. 解偏导方程

<sup>1</sup>欠拟合 (underfitting)，或者叫作叫做高偏差 (bias)，过拟合 (overfitting)，也叫高方差 (variance)

<sup>2</sup> $y_i$  是样本数据， $\hat{y}$  是理论值

### 3. 代入上述函数

由拉格朗日乘子法的定义可知，它是一种寻找极值的方法，因此该方法并不能保证极值点是最低点或者最高点。

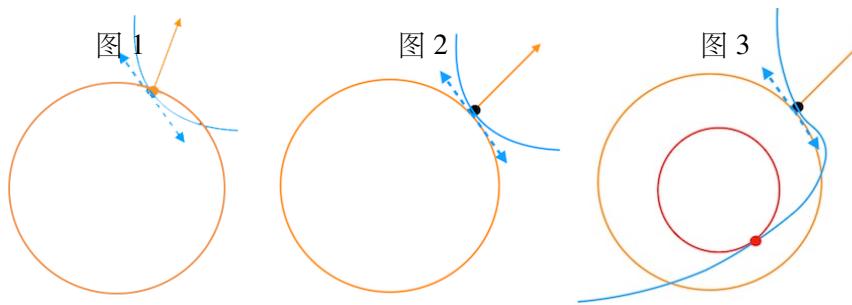


图 8.4: 拉格朗日算子法

这里不加任何推导，很明显相切的时候才能到最低点。橙线的梯度（上图 1 橙色箭头）和蓝线的切线（蓝色虚线）不是垂直关系。蓝线的两个切线方向，分布往函数高处走（与梯度的夹角小于 90 度），和往函数低处走（与梯度的夹角大于 90 度）。因此，两条曲线相交时，肯定不在最低点或最高点的位置。

如果两条曲线相切（上图 2），那么蓝线的切线和橙线的梯度是在切点这个位置垂直。这时，蓝线的切线方向指向橙线的等高线方向。图 3 中相切的点有两个，而红点的明显比黑点小。要判断找到的点是极低点还是极高点，需要将切点代入原函数再进行判断。在实际求解时，令各偏导为 0 就能满足相切的条件。

## 8.3.2 KKT 条件

在优化理论中，KKT 条件是非线性规划（Nonlinear Programming）最优解的必要条件。KKT 条件将拉格朗日算子法中的等式约束优化问题推广至不等式约束。将上一节中的约束等式  $h_k(x) = 0$  推广为  $g_i(x) \leq 0$ ，优化问题如下：

$$\min f(x) \quad | \quad s.t. g(x) \leq 0$$

约束不等式  $g(x) \leq 0$  称为 Primal Feasibility，定义可行域为（Feasible Region） $K = \{x \in \mathbb{R}^n | g(x) \leq 0\}$ 。设  $x^*$  为满足约束条件的最优解，可分为内部解、边界解两种情况。内部解的时候， $g(x)$  不起作用，退化为无约束问题。边界解的时候，取不等式的边界，约束转换为  $g(x) = 0$ ，转化为 Lagrange 算子法。

1.  $g(x^*) \leq 0$ ，最佳解位于  $K$  的内部，此时约束条件是无效的；
2.  $g(x^*) = 0$ ，最佳解落在  $K$  的边界，此时约束条件是有效的。

KKT-图 1 中，把 Lagrange 条件变成一个区域，该图的切点处仍旧是最优解。KKT-图 2 中，在边界处  $g(x) = 0$  等价于 Lagrange 算子法。KKT-图 3 中，最佳解位于  $K$  的内部，约束条件无效。KKT 是 SVM（support vector machine）支持向量机的重要理论基础。

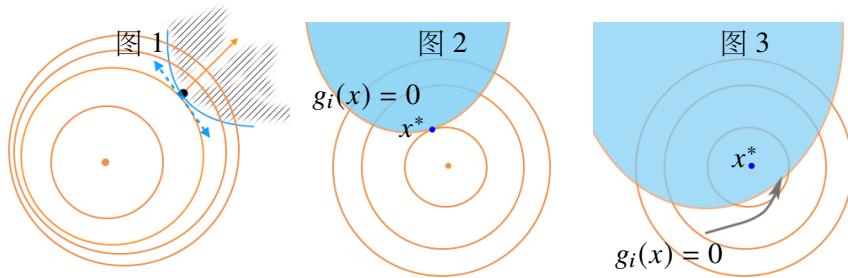


图 8.5: KKT

### 8.3.3 凸优化

凸优化在数学规划领域具有非常重要的地位。若能把一个实际问题表述为凸优化问题，基本上意味着该问题已经得到解决，这是非凸的优化问题所不具有的性质。机器学习中有很多优化问题都要通过凸优化来求解，即便是在非凸优化中，凸优化同样起着重要的作用。实际上，很多非凸优化问题，可以转化为凸优化问题来解决。

$$f_i(\alpha x + \beta y) \leq \alpha f_i(x) + \beta f_i(y), \quad \text{其中 } \alpha + \beta = 1, \alpha \geq 0, \beta \geq 0$$

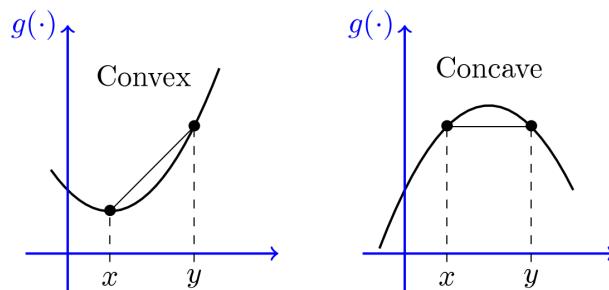


图 8.6: 凸函数和凹函数

如图 8.6 所示，凸函数的几何意义为任意两点连线上的取值大于该点在函数上的取值，而凹函数正好相反。很显然，凸函数总是在其任意一点的切线的上方。通常使用函数的二阶导来判断一个函数是否为一个凸函数。

$$\nabla_x^2 f(x) \geq 0$$

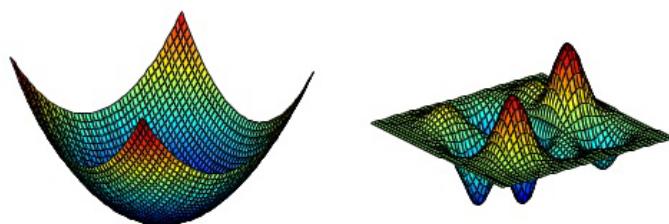


图 8.7: 凹函数转化为多个凸函数优化问题

凸优化相对简单，具有良好的几何性质，比如分离平面和支撑平面。通常凸问题的

局部最优解也是全局的最优解，Lagrange 对偶是凸优化理论中最重要的工具。往往只要把某一问题抽象为凸问题，就可近似认为这个问题已经解决了。

SVM 本身就是把一个分类问题抽象为凸优化问题，利用凸优化的各种工具（如 Lagrange 对偶）求解和解释。深度学习中关键的算法是反向传播（Back Propagation），本质就是凸优化算法中的梯度下降算法。总的来说，凸优化在工程领域的应用中有着无可撼动的地位。

实际上，生活中几乎所有问题的本质都是非凸的。如图 8.7 所示，很多非凸问题可以转化成多个凸优化问题，加速问题的求解。

## 8.4 损失函数

在机器学习中，机器的学习需要某个指标来表示现在的状态，然后，以这个指标为基准，寻找最优权重参数。机器学习利用已知样本，推演隐藏在背后的真实曲线。这里假设该某曲线为  $h_\theta = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$ ，通过样本求解  $\theta_i (i = 0 \dots 4)$ 。不过，这个假设（Hypothesis）很有可能不是最好的。实际上，我们获得的训练数据总是有误差，不能完全拟合这些数据点，否则就会导致过拟合（overfit）的问题。

评价训练过程是否有效，可使用方差衡量源数据和期望值相差的偏离程度。若是逐渐减小，就是一个有效的训练过程。这个方差也是一个函数，称为损失函数（loss function）<sup>3</sup>，用于计算预测值  $f(x)$  与真实值  $y$  的不一致程度。它是一个非负的实值函数，通常使用  $L(Y, f(x))$  来表示。损失函数值越小，拟合效果就越好。当样本个数为  $n$  时，此时的损失函数变为：

$$L(Y, f(x)) = \frac{1}{N} \sum_N (Y - f(x))^2$$

除 2 是为了方便求导

$Y - f(X)$  表示的是残差，整个式子表示的是残差的平方和，而我们的目的就是最小化残差的平方和。在实际应用中，通常使用均方误差（mean squared error）作为衡量指标，如下：

$$E = \frac{1}{2} \sum_k (y_k - t_k)^2$$

在上式中  $k$  表示着数据处于哪一个维度， $t_k$  表示着各维度的监督数据。通常情况下，监督数据仅将正确解标签表示为 1，而其他非正确解则表示为 0。而机器学习输出的结果  $y_k$ ，则是会在各个维度都显示该维度是正确解标签的可能性。均方误差通过计算机器学习的输出和正确解监督数据的各个元素之差的平方，再求总和。从而得到该权重参数的输出结果与正确结果之间的偏差。

我们现在再介绍另一种常用函数，交叉熵误差（cross entropy error）。其公式如下：

$$E = - \sum_k t_k \log y_k$$

---

<sup>3</sup>也称为代价函数 (Cost Function)



在上式中,  $y_k$  代表着机器学习的输出,  $\log$  表示以  $e$  为底数的自然对数 ( $\log_e$ )。 $t_k$  代表着正确解标签, 仅当解标签为正确时,  $t_k$  的索引才为 1。其余情况都为 0。因此,  $E$  所代表的实际为解标签为正确时所输出的自然对数。

线性回归是确定两个或两个以上变量间关系的一种常见统计分析方法, 被广泛用于回归分析。只有一个自变量的情况称为简单回归, 多于一个自变量的情况叫做多元回归。故可分为一元线性回归分析方程和多元线性回归分析方程。给定一个随机样本  $(Y_i, X_{i1}, \dots, X_{ip})$ , 线性回归模型表示为以下的形式:

$$Y_i = \beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2} + \dots + \beta_p X_{ip} + \epsilon_i \\ i = 1, \dots, n.$$

使用最小二乘法 (Least Square Method) 需要做矩阵的逆运算, 下一章我们再介绍。而梯度下降法起点和学习率都非常重要, 顺着梯度  $\Delta$  下降最快的方向迭代调整。若干次迭代之后就会落入局部最小点附近, 有可能来回震荡无法达到极值点。所以, 调整学习率就非常关键, 因此到极值点附近的时候收敛速度也会变慢。

#### 8.4.1 反向传播 (Back Propagation)

前述几节, 我们利用偏导获得梯度, 然后逐步调节参数, 朝着误差越来越小的方向迭代。这还算不上神经元, 实际比这还要复杂一些, 通常还有一个激活函数 (Activation Function), 只有对神经元的刺激足够强才会前向传递。典型的深度神经网络, 至少包含: 输入层、隐藏层、输出层。

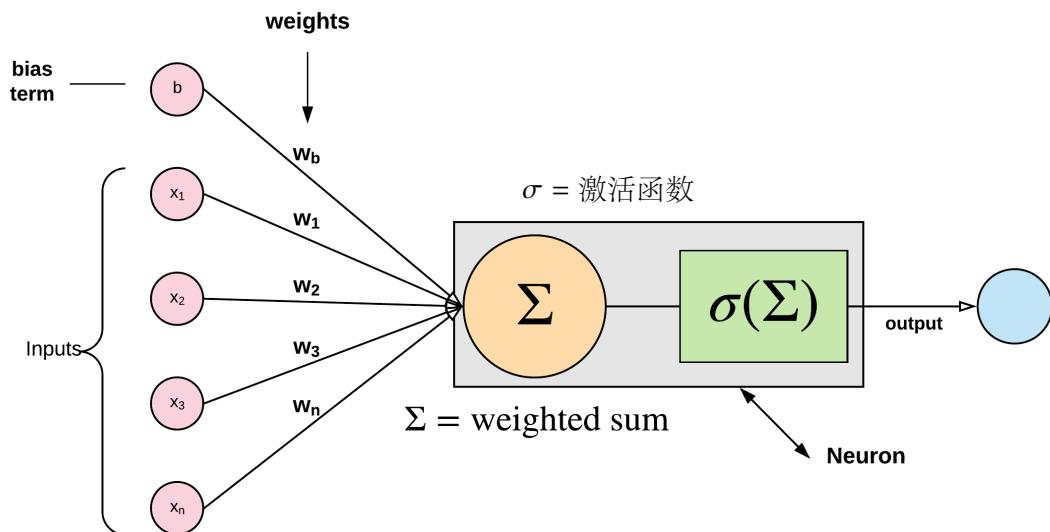


图 8.8: 神经网络

缺少激活函数的线性模型, 甚至都无法解决异或问题。如果选用 Linear 函数  $g(x) = x$  作为激活函数就无法划分下图的区域, 很显然隐藏层的混入也非常关键。通过隐藏层引入了更多线段, 以便合成一个封闭的多边形, 恰好能对数据分类。激活函数可以在隐藏层引入更多特征, 产生非线性结果以解决线性不可分问题。

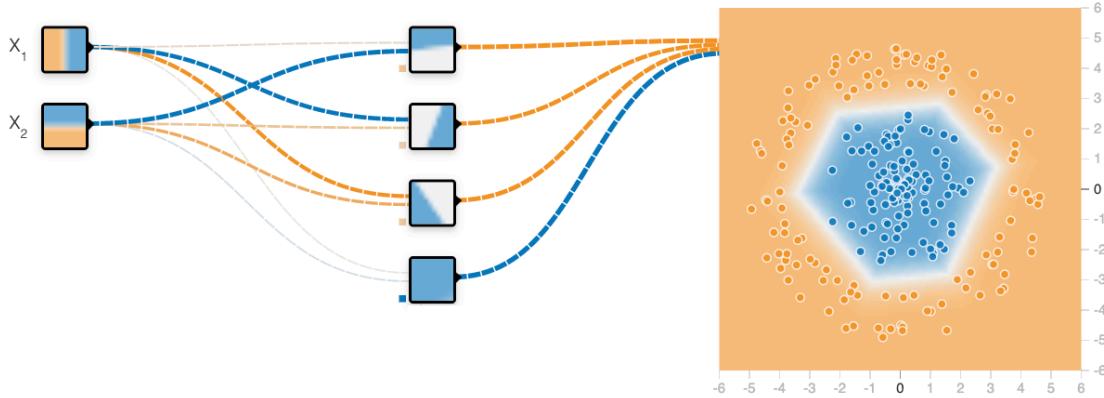
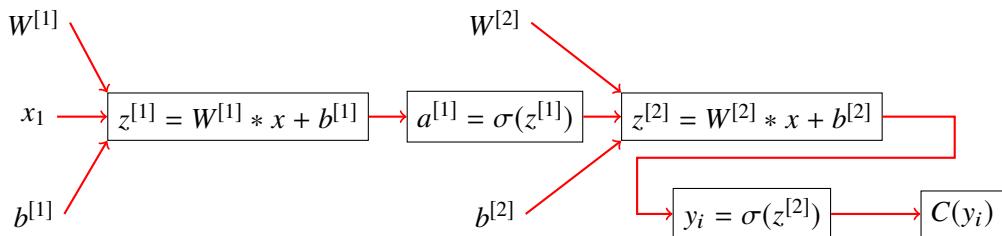


图 8.9: 使用激活函数，合成多边形区域

BP 神经网络是一种多层的前馈神经网络，分为两个阶段：前向传播和反向传播。前向传播从输入层经过隐含层，最后到达输出层。而反向传播从输出层到隐含层，最后到输入层，逐步调节隐藏层（Hidden Layer）到输出层的权重和偏置（bias），输入层到隐含层的权重和偏置。经由反向传播，把误差反馈给神经网络用于调节参数。此处不作严格证明，

假设有一个两层深的网络（1个隐藏层），并且每一层只有一个神经元， $\sigma$  为 sigmoid 函数。相应的数学模型如下：



从输入到输出，函数都是平滑的可求偏导的，由误差估计  $C(\hat{y})$  使用梯度下降法，反向调节参数。其中， $a^{[i]}$  作为下一个神经元的输入，也就是  $x_i$ 。

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial b_1} = \frac{\partial C}{\partial y_i} \cdot \sigma'(z_2) \cdot W_2 \cdot \sigma'(z_1)$$

如图 8.8，神经网络的输入层就是我们要分类的样本特征，而隐藏层和输出层每个节点都代表了一个 sigmoid 单元。计算的时候，通常把  $b$  转化为  $x_0$  和  $w_0$ ，可把 sigmoid 单元简化为  $net = \sum_{i=0}^n w_i x_i$ 。求各输入的和用 sigmoid 函数得到输出  $out = \sigma(net) = \frac{1}{1+e^{-net}}$ ，这便是一个单元的计算过程。

整个网络便是前一层作为输入与后一层的每各单元连接，计算出各单元的输出后，再把这一层做为输入传递到后一层。算法本身并不复杂，隐含在其中的数学推理比较晦涩，包含的过程主要是梯度下降和函数迭代。

## 第9章 概率与分布

---

日常生活中，我们差不多一直在做各种判断，这样或者那样。俗话说，选择大于努力。如果你的认知或者价值观是错误的，将不会得出正确的结果。这种价值观的偏离，需要“误差分析”保证在正确的轨道上。然而，误差很难线性（任意曲线）吻合的，这取决于个人的生活经验，



手写数字：4 还是 9？

如上图，手写数字很不容易判断，只能根据大多数人的情况猜测。如果，你能拿到该人之前的手稿了解一下，就能有 99.999% 概率做出正确判断。

有很多情形，我们也很难正确判断，只需要人工智能输出 TOPN 可能性。譬如，仅根据病人是否发烧、咳嗽难以确诊是否肺炎。进一步，借助沟通和血液化验，才更加确诊病人的实际病症。这也只能说是 99.99% 的准确率，即使是非常有经验的医生也有误诊的概率，只是非常小而已。

人工智能研究的多是不确定性问题，需要你掌握概率和统计学的基础知识。现实生活中，很多不确定事件都服从某种分布，譬如公交地铁站客流量服从泊松分布，而 1 小时内到银行办理业务的客户数都也服从泊松分布（Poisson distribution）。泊松分布具有以下特点：

1. 无限分隔为若干小时间段，在这接近于零的小时间段里，发生 1 次的概率与时间段的长度成正比。
2. 在每一个极小时间段内，该事件发生两次及以上的概率恒等于零。
3. 在不同的小时间段里，发生与否相互独立。

$$P(x) = \frac{e^{-\lambda} \lambda^x}{x!}$$

泊松分布服从上述公式，很显然它所有事件的概率和等于 1。

$$\frac{e^{-\lambda} \lambda^x}{x!} = e^{-\lambda} \left( \frac{\lambda^x}{x!} \right) = e^{-\lambda} * e^{\lambda} = 1$$

在医疗质量上，美国误诊率是 30-40% 之间，因医疗差错而死亡的人，仅次于心血管疾病与肿瘤。

个别疾病的误诊率高得使你不敢想象，达到 70% 以上。

我国的误诊率尚没有公认的数据，但与美国相比只会高不会低。

## 9.1 全概率公式

全概率公式是概率论中重要的公式。他的意义在于直接计算  $P(A)$  比较困难时，可以将事件  $A$  分解为几个小事件  $B_i$ ，通过计算这些小事件的概率和，从而达到简化问题的效果。

全概率公式可以用如下的公式表示：

$$P(A) = \sum_{i=1}^{\infty} P(B_i) P(A|B_i) \quad (9.1)$$

其中，事件  $B_1, B_2, B_3, \dots, B_n$  需要满足完备事件组，即两两之间不能有交集，它们的和为全集，且概率大于零。这样，事件  $A$  就被分解成了小事件  $AB_1, AB_2, AB_3, \dots, AB_n$ ，由概率的加法公式即可得出事件  $A$  的概率。

与全概率公式相反，贝叶斯公式是基于  $P(A)$  已知的前提下，寻找事件发生的原因  $P(B_i|A)$ 。贝叶斯公式可以用以下公式表示：

$$P(B_t|A) = \frac{P(B_t) P(A|B_t)}{\sum_{j=1}^n P(B_j) P(A|B_j)}$$

其中事件  $B_1, B_2, B_3, \dots, B_n$  须满足完备事件组。

## 9.2 最大似然

最大似然估计是一种统计方法，用来求一个样本集的相关概率密度函数的参数。通俗地来讲，就是通过已知的样本结果，反推最有可能出现这样结果的参数值。

例如，假设一个口袋中同时存在黑球和白球，我们从中随机抽取十个球，得到了 8 个黑球和 2 个白球。在求解最有可能的黑白球比例时，我们就会采用最大似然法：假设从中抽到黑球的概率为  $p$ ，那么得到 8 次黑球 2 次白球的概率为：

$$P(A) = p^8 * (1 - p)^2$$

在这个公式中，使  $P(A)$  达到最大的  $p$  值即为我们要求的结果。这就是最大似然问题的基本过程。

## 9.3 期望与方差

期望也称数学期望，在概率论与数理统计中指的是一个离散型随机变量在实验中每次可能的结果乘上各自的概率的总和。在机器学习中，期望值是衡量一组数据离散程度的重要度量。

期望值的计算用以下公式表达  $E[X] = \sum_i p_i x_i$ ，其中  $E[X]$  代表期望值， $x_i$  和  $p_i$  分别代表每次随机变量实验的可能结果与出现的概率。

与期望值类似，方差是概率论中衡量随机变量离散程度的度量。它具体指每个样本值与全体样本值平均数之差的平方值的平均数。一般情况下，方差的公式可以用如下公

式定义： $\sigma^2 = \frac{\sum(X-\mu)^2}{N}$ ；其中， $\sigma^2$  为总体方差， $X$  为单个样本值， $\mu$  为总体样本的平均值， $N$  为总体样本的个数。

同时，方差还可以用期望值来求出  $\text{Var}(X) = E[X^2] - (E[X])^2$ ，由此可见期望值与方差之间紧密的联系。

## 9.4 常见概率分布

本书不打算把所有的概率知识进行讲解，主要为后续章节做铺垫。生活中，常见的概率事件都是离散的，譬如泊松分布。

$$P(X = x_i) = p_i, i = 1, 2, \dots, n \quad (9.2)$$

且概率  $P_i$  满足  $\sum_{i=1}^n P_i = 1$ 。因此，离散型随机变量  $X$  的概率分布函数为，其中  $x_i$  为可能的状态。

$$F(x) = \sum_{x_i < n} P_i. \quad (9.3)$$

这是离散型概率的特征，具备以下特征：

1.  $P(x_i) = 0$  代表不会发生， $P(x_i) = 1$  表示一定会发生。
2. 总概率不会大于 1，也就是  $F(x) \leq 1$

概率分布函数是概率论的重要概念，在实际应用中常用的有正态分布函数、泊松分布函数、二项分布函数等。对于离散型随机变量，分布函数是“0-1 分布”、“二项式分布”、“泊松分布”等；而连续型随机变量有“均匀分布”、“正态分布”、“瑞利分布”等。

### 9.4.1 正态分布

正态分布，也称常态分布，高斯分布，是一种概率分布模型。正态分布在数学，工程与物理领域有着重要的意义。在机器学习中，正态分布的统计模型应用非常广泛。

若一个随机变量  $X$  服从位置参数为  $\mu$ ，尺度参数为  $\sigma$  的概率分布，且公式为

$$X \sim N(\mu, \sigma^2)$$

则称变量  $X$  满足正态分布。

正态分布的概率密度函数则表示为

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

值得注意的是正态分布的数学期望值即为位置参数  $\mu$ ，标准差即为尺度参数  $\sigma$ 。

## 9.5 矩估计

矩估计又称矩法估计，指的是利用样本矩来估计总体所对应的参数。矩估计在概率论中有着广泛的应用，主要的理论基础有大数定理和中心极限定理。



### 9.5.1 大数定理

大数定理是描述实验次数很大时所呈现的概率性质的定律。在随机事件大量重复出现的情况下，事件往往呈现出近乎必然的概率，这就是大数定律最简单的解释。

大数定律在高等数学上有以下两种数学解释：切比雪夫大数定理和伯努利大数定理。切比雪夫大数定理假设有  $x_1, x_2, x_3, \dots, x_n, \dots$  这样一列相互独立的随机变量，那么对于任意小的正数  $\xi$ ，有公式：

$$\lim_{n \rightarrow \infty} P \left\{ \left| \frac{1}{n} \sum_{k=1}^n x_k - \frac{1}{n} \sum_{k=1}^n E x_k \right| < \xi \right\} = 1$$

切比雪夫大数定理可以有以下理解：当样本容量  $n$  不断增大，样本的平均值将不断接近总体的平均值。

伯努利大数定理则对大数定理进行了概率的解释，假设  $\mu_n$  是独立事件中事件  $A$  发生的次数，则对于任意小的正数  $\xi$ ，满足公式：

$$\lim_{n \rightarrow \infty} P \left( \left| \frac{\mu_n}{n} - p \right| < \xi \right) = 1$$

伯努利大数定理可以这样解释：当事件发生次数  $n$  足够大时，事件  $A$  发生频率将接近其发生的频率。

### 9.5.2 中心极限定理

中心极限定理与大数定理一样，都是描述试验次数很大时，所呈现的概率性质定理。但与大数定理不同的是，中心极限定理描述的是随机事件大量重复，结果服从正态分布的情况。

假设有随机变量  $X_1, X_2, \dots, X_n, \dots$  相互独立，并具有方差和期望值，则对任意  $x$  有  $F_n(x)$

$$F_n(x) = P \left\{ \frac{\sum_{i=1}^n X_i - n\mu}{\sigma\sqrt{n}} \leq x \right\}$$

利用中心极限定理和大数定理，我们可以利用少量数据对总体进行精确的预测，从而达到矩估计的效果。

# 第 10 章 线性代数

## 10.1 矩阵和向量

矩阵是从许多实际问题中抽象得来的数学概念，是整个线性代数学科的基础，在自然科学，工程数学和国民经济中有着广泛的应用。在机器学习领域，大部分计算方法都是以矩阵的形式进行，因此学习矩阵的性质和计算显得尤为重要。

矩阵的定义是  $m * n$  个数  $a_{ij}$  ( $i = 1, 2, \dots, m; j = 1, 2, \dots, n$ ) 排成  $m$  行  $n$  列的矩阵数表。这样的一个数表称为  $m * n$  矩阵，简称为矩阵， $a_{ij}$  也被称为矩阵的第  $i$  行第  $j$  列元素。

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad (10.1)$$

特殊的，当  $m=1$  时，矩阵  $A = (a_1, a_2, \dots, a_n)$  被称为行矩阵，也叫  $n$  维行向量；同样的，当  $n=1$  时，矩阵  $A = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$  被称为列矩阵，也称为  $m$  维列向量；当  $m=n$  时，矩阵  $A = (a_{ij})_{n*n}$  称为  $n$  阶矩阵或  $n$  阶方阵，也可记为  $A_{n*n}$ 。

向量指的是既包含大小，又包含方向的一类量，例如位移，速度，加速度，力等，又称为矢量。在线性代数中，向量可以用矩阵的形式进行表示，例如一个一维的向量可以用向量  $A = (a_1)$  表示，二维向量可以用  $A = (a_1, a_2)$  表示，如图10.1和图10.2所示。同理，我们可以推出，包含  $n$  个分量的向量可以称为  $n$  维向量，用  $A = (a_1, a_2, a_3, \dots, a_n)$  表示。

在 DL4J 中，普通矩阵的创建用 `create` 方法实现，源码如图10.3所示。首先我们需要传入两个数组 `data` 和 `shape`，作为源数据组并决定矩阵的形状与维度。然后，系统会调用

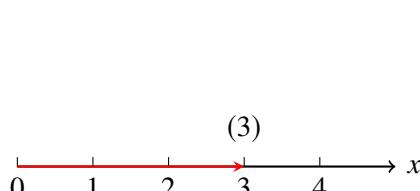


图 10.1: 一维向量示意图

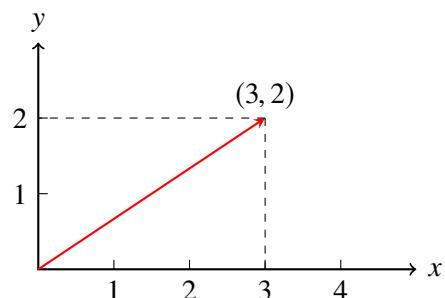


图 10.2: 二维向量示意图

checkShapeValues 方法来确保输入 data 数组的长度与 shape 数组相符。最后，利用这些数据创建一个 INDArray 并返回，这就完成了矩阵的创建。

特殊的，我们可以用简便的方法来创建部分简单的矩阵，例如 zeros() 方法可以创建一个全零矩阵，仅需要传入行数和列数即可。同理还有 ones(), valueArrayof() 等方法。

### 10.1.1 基本运算

矩阵有四种运算：加减、数乘、乘法和转置。下面我们将用例子展示矩阵的运算，假设我们有两个矩阵  $A$  和  $B$ ，如图10.4与图10.5所示。

#### 10.1.1.1 矩阵的加减

$A$  和  $B$  是两个  $m * n$  型矩阵，那么存在矩阵  $C$ ，记  $C = A + B$ ，称  $C$  为矩阵  $A$  和  $B$  的加法运算。需要注意的是，进行加法运算的两个矩阵行数和列数必须相等，否则无法进行运算。

矩阵的加法在 DL4J 中由 add 方法和 addi 方法实现。代码会先检测输入的数组是否为数字矩阵，以此判断是否可进行加法操作，最后将两个矩阵进行加法操作。

#### 10.1.1.2 矩阵的数乘

矩阵  $A$  是一个  $m * n$  型矩阵， $k$  是一个数，则称矩阵  $D$  为矩阵  $A$  与数  $k$  的乘积，称为数乘，记为  $kA$ 。

矩阵的数乘与加法基本一致，都是先检验是否为数字矩阵的操作，随后对矩阵中的每个元素进行乘法操作。

矩阵的加法和数乘运算统称为矩阵的线性运算。矩阵的线性运算满足交换律，分配律和结合律。

#### 10.1.1.3 矩阵的乘法

设  $A = (a_{ik})_{m*s}$ ,  $B = (b_{kj})_{s*n}$ , 定义  $A$  与  $B$  的乘积  $AB$  是一个  $m * n$  矩阵  $C = (C_{ij})_{m*n}$ ,  $C$  的第  $i$  行第  $j$  列元素等于  $A$  的第  $i$  行元素与  $B$  的第  $j$  列对应元素乘积的代数和，如图10.8:

```
public static INDArray create(double[] data, long[] shape) {
    checkShapeValues(data.length, shape);

    INDArray ret = INSTANCE.create(data, shape, Nd4j.getStrides(shape, Nd4j.order()), DataType.DOUBLE, Nd4j.getMemoryManager().getCurrentWorkspace());
    return ret;
}
```

图 10.3: NDArray 的创建

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

图 10.4: 矩阵 A

$$\begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix}$$

图 10.5: 矩阵 B

DL4J 中，矩阵乘法的源码先对进行乘法操作的两个矩阵判断是否为同一数据类型，之后再得出乘法计算后矩阵的维度和形状，最后计算后将结果写入结果矩阵，返回结果，如图 10.9。

#### 10.1.1.4 矩阵的转置

设存在  $m * n$  矩阵  $A$ ，则称  $n * m$  矩阵  $B$  为  $A$  的转置矩阵，记为  $A^T$ 。

在 DL4J 中，矩阵的转置调用了 `permute` 方法，先将原矩阵的形状，源数据，排列规则等属性提出并进行转置，再利用新属性生成一个新的 `INDArray` 并返回，在实际使用中我们需要调用 `transpose()` 方法即可完成矩阵的转置。

#### 10.1.2 多维空间

多维空间一般指多维向量空间。一般来说，我们将实数域上的全体  $n$  维向量构成的集合称为  $n$  维向量空间  $R^n$ 。

在此基础上，我们设  $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_r$  是向量空间中  $V$  中的向量，若它们满足：(1)  $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_r$  线性无关；(2)  $V$  中的任何一个向量都可以用  $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_r$  线性表示，则称  $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_r$  为向量空间  $V$  的一个基， $r$  称为向量空间  $V$  的维数，并称  $V$  为  $r$  维向量空间。

显然，向量空间的基不是唯一的。同时，我们应该注意到向量的维数和向量空间的维数并不是同一个概念。

$$\begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots & a_{2n} + b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \cdots & a_{mn} + b_{mn} \end{pmatrix}$$

图 10.6: 矩阵 C

---

```
public INDArray add(INDArray other) {
    validateNumericalArray("add", false);
    return addi(other, Nd4j.createUninitialized(Shape.pickPairwiseDataType
        (this.dataType(), other.dataType()), this.shape(), this.ordering()));}
```

---

$$(ka_{ij})_{m \times n} = \begin{pmatrix} ka_{11} & ka_{12} & \cdots & ka_{1n} \\ ka_{21} & ka_{22} & \cdots & ka_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ ka_{m1} & ka_{m2} & \cdots & ka_{mn} \end{pmatrix} \quad (10.2)$$

图 10.7: 矩阵  $D$ 


---

```
public INDArray mul(Number n) {
    validateNumericalArray("mul", false);
    return muli(n, Nd4j.createUninitialized(Shape.pickPairwiseDataType
(this.dataType(), n), this.shape(), this.ordering()));}
```

---

$$AB = \begin{pmatrix} a_1 b_1 & a_1 b_2 & \cdots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \cdots & a_2 b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_n b_1 & a_n b_2 & \cdots & a_n b_n \end{pmatrix}$$

图 10.8: 矩阵  $E$ 


---

```
public INDArray mmul(INDArray other) {
    Preconditions.checkState(this.dataType() == other.dataType(), "Matrix multiplication:
arrays must have same dtype: %s vs. %s", this.dataType(), other.
dataType());
    long[] shape = {rows(), other.rank() == 1 ? 1 : other.columns()};
    INDArray result = createUninitialized(this.dataType(), shape, 'f');
    if (result.isScalar())
        return Nd4j.scalar(Nd4j.getBlasWrapper().dot(this, other)).reshape(1, 1)
    ;
    return mmuli(other, result);
}
```

---

图 10.9: 矩阵乘法源码

### 10.1.3 特征值与特征向量

设  $A$  是一个  $n$  阶矩阵，如果存在数  $\lambda$  和非零列向量  $\alpha$ ，使得  $A\alpha = \lambda\alpha$ ，则称  $\lambda$  为矩阵  $A$  的一个特征值，称  $\alpha$  为  $A$  属于特征值  $\lambda$  的一个特征向量。

代码块展示了简单的二维矩阵计算特征值与特征向量的方法。在此过程中，我们首先求出矩阵  $A$  的特征多项式，并求出  $A$  的所有特征值；然后再根据特征值求出齐次线性方程组  $(\lambda * I - A)x = 0$  的基础解系，进而得出特征向量。

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad (10.3)$$

图 10.10: 矩阵  $A$ 

$$\begin{pmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{pmatrix} \quad (10.4)$$

图 10.11: 矩阵  $B$

```

public INDArray permute(int... rearrange) {
    checkArrangeArray(rearrange);
    int[] newShape = doPermuteSwap(shapeOf(), rearrange);
    int[] newStride = doPermuteSwap(strideOf(), rearrange);

    char newOrder = Shape.getOrder(newShape, newStride, 1);

    INDArray value = create(data(), newShape, newStride, offset(), newOrder
        );
    return value;
}

private static void getEigenValue(int[] A){
    int a = 1;
    int b = -(A[0] + A[3]);
    int c = A[0]* A[3] - A[1] * A[2];
    double t = b*b - 4 * (a * c);
    if (t >= 0) {
        double lambda1 = (Math.sqrt(t) - b) * 1.0 / (2 * a);
        double lambda2 = (-(Math.sqrt(t))-b) * 1.0 / (2 * a);
        int[] resultArray1 = {1,(int)((A[2])/(lambda1 - A[3]))};
        System.out.print("矩阵的第一个特征值为" + lambda1 + "，特征向量为" + "[" +
            resultArray1[0]
            + "," + resultArray1[1] + "]");
    }
}

```

## 10.2 雅克比矩阵

雅克比矩阵是一阶偏导数以一定方式排列组成的矩阵。在线性代数中，雅克比矩阵体现了一个可微方程与给定点的最优线性逼近，有着重要的意义。

雅克比矩阵的定义为假设存在  $n$  个  $y_n = f(x_1, x_2, \dots, x_n)$  型函数。如果这些函数的偏导数存在，则可组成一个  $m$  行  $n$  列的矩阵，这个矩阵就是雅克比矩阵，用符号表示为  $J_F(x_1, \dots, x_n)$ ，如图10.12所示。

$$J_F(x_1, \dots, x_n) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \quad (10.5)$$

图 10.12: 雅克比矩阵

## 10.3 Hessian 矩阵

Hessian 矩阵又称黑塞矩阵，海瑟矩阵。与雅克比矩阵相对应，是多元函数的二阶偏导数构成的方阵，描述了函数的局部曲率情况。

Hessian 矩阵的定义为设有一个  $n$  元实函数  $f(x_1, x_2, \dots, x_n)$  在定义域内有二阶连续偏导，这些二阶偏导组成的矩阵被称为 Hessian 矩阵，如下图所示。

$$\begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix} \quad (10.6)$$

图 10.13: Hessian 矩阵

## 10.4 泰勒展开和牛顿法

在数学中，泰勒展开式主要指利用函数在一点的各阶导数值，来构建一个多项式近似函数值，同时还能给出近似值与实际值的误差。如果一个函数在点  $x_0$  处可以计算  $n$  阶导数，则我们有 Taylor 展开：

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + R_n(x)$$

其中  $R_n(x)$  称为拉格朗日余项，值为  $R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1}$ ，当  $x_0 = 0$  时，我们能得到 Taylor 的麦克劳林公式：

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + o(x^n)$$

## 10.5 向量与矩阵求导

利用向量和矩阵求导可以让原本有许多变量的单个函数的偏导值收集到可以视为单个个体的矩阵和向量当中，这大大简化了我们寻找多元函数的最大值或最小值，微分方程组操作等问题。

在上面的两节里，我们介绍了雅克比矩阵和 Hessian 矩阵，它们的本质都是特殊的矩阵求导。在这里我们将给出一般的形式。对于一个向量  $y = (y_1, y_2, \dots, y_m)$  对  $x$  求导，我们可以写成  $\frac{\partial y}{\partial x} = \left( \frac{\partial y_1}{\partial x}, \frac{\partial y_2}{\partial x}, \dots, \frac{\partial y_m}{\partial x} \right)$ 。在求解梯度下降的加速度问题时，向量求导能比较好地解决问题。矩阵的求导与雅克比矩阵的形式十分类似，都是对矩阵中的元素逐个求导。

同理，矩阵的求导也与此类似，设有函数矩阵  $Y$ ，其对于  $x$  的求导结果可以表示为：

$$J_F(x_1, \dots, x_n) = \begin{pmatrix} \frac{\partial y_{11}}{\partial x} & \cdots & \frac{\partial y_{1n}}{\partial x} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_{m1}}{\partial x} & \cdots & \frac{\partial y_{mn}}{\partial x} \end{pmatrix} \quad (10.7)$$

图 10.14: 矩阵的求导

# 第 11 章 数学基础-DEMO

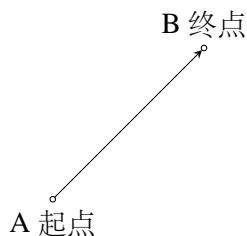
AI 实在是一个多学科综合性应用，涉及到的理论不仅多而且深。限于笔者知识水平以及篇幅要求，不会对此进行长篇巨幅地解释。

## 11.1 向量运算

单独的一个数字，称为标量（scalar），而向量通常用数组的形式表示。一个向量就是一列（行）数字：

$$x = [x_1 x_2 \dots x_n], x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad (11.1)$$

向量是一种带方向指示性的量，代表空间中的一个点。一维向量  $\vec{a} = [4]$  代表 a 点在原点右侧距离为 4 的位置。而二维向量  $\vec{b} = [3 4]$  就代表了在第一象限坐标 (3,4) 有一个点 b。所以，向量是一个方向性的偏移量。向量可以在坐标系中自由平移，选定一个起点就确定了它的终点。



向量支持的运算有内积（点乘）和外积（叉乘）运算，以及加减运算。向量 A、B 的运算过程，设  $\vec{A} = (a_1, a_2, a_3), \vec{B} = (b_1, b_2, b_3)$

1. 点乘，结果是一个标量： $A \cdot B = a_1 * b_1 + a_2 * b_2 + a_3 * b_3$
2. 叉乘，结果还是个向量： $A \times B = (a_2 * b_3 - a_3 * b_2, a_3 * b_1 - a_1 * b_3, a_1 * b_2 - a_2 * b_1)$
3. 标量，用于每一个元素： $A + 2 = (a_1 + 2, a_2 + 2, a_3 + 2)$

通常使用数组来表示向量，但这样扩展性不够好。咱们使用 Java 面向对象的方法，把数组和函数封装进一个向量类型里面去。定义一个 Vector 类，如图 ??：

```
public IntVector(int... array) {
    this.array = array;
}

public IntVector add(IntVector vector) {
    foreach(this.array, vector.array, (a,b)->a+b);
```

InVector
array : int[]
add(v : InVector) : InVector
sub(v : InVector) : InVector
dot(v : InVector) : int
cross(v : InVector) : IntVector

图 11.1: 类图: IntVector

```

    return this;
}

public int dot(IntVector vector) {
    return foreach(this.array, vector.array, (a,b)->a*b);
}

```

如上代码，构造函数用到了可变参数，你可以理解成一个int[]。在编译的时候，Java会自动地把函数声明和调用代码都转换成数组。*add* 运算返回this，用来支持链式运算：v1.add(v2).add(v3)...。容易看出，*add* 和 *dot* 运算，是对应元素相加、相乘的过程。使用*foreach* 从相加的 2 个向量中，逐个地取出数据并执行  $\lambda$  操作。

Listing 11.1: 叉乘运算

```

public IntVector cross(IntVector vector) {
    int[] v1 = this.array;
    int[] v2 = vector.array;

    int length = vector.array.length;
    if (length == 2) {
        return new IntVector(v1[0]*v2[1]-v1[1]*v2[0]);
    }

    int[] result = IntStream.range(0, length).map(
        i->v1[(i+1)%length]*v2[(i+2)%length] -v1[(i+2)%length]*v2[(i+1)%length]
    ).toArray();

    return new IntVector(result);
}

// sum用于dot运算
private static int foreach(int[] src, int[] dst, IntBinaryOperator op) {
    int sum = 0;
    for (int i = 0; i < src.length; i++) {

```



```
    src[i] = op.applyAsInt(src[i], dst[i]);
    sum += src[i];
}
return sum;
}
```

以上，仅仅只能处理 int 类型的向量。结合 IntVector 的实现代码，相信大家也能理解实数类型的向量。也许你也意识到，使用 *add* 太麻烦了，但 Java 目前还不支持运算符重载。如果确实想使用，可以借助 *Java-OO* 开源插件<sup>1</sup>

#### Listing 11.2: 运算符重载

```
@Test
public void add_operator_override() {
    IntVector a = new IntVector(1,2,3);
    IntVector b = new IntVector(10,20,30);

    IntVector c = a + b; // 看上去运算符重载了
    int[] expected = new int[]{11, 22, 33};

    assertEquals(expected, c.array());
    assertEquals(new IntVector(expected), c);
}
```

实际上，很少有自己实现的，并且也较难保障准确性和稳定性。开源的 ND4J 作为 DL4J 的张量计算库，提供了丰富的运算接口，支持几乎所有的数学操作，相当于 Python 界的 numpy。以下代码 11.3

#### Listing 11.3: dl4j 例子

```
INDArray vec1 = Nd4j.create(new float[]{1,2,3});

// vector add
INDArray vec2 = Nd4j.create(new float[]{10,20,30});

// [[11.0000, 22.0000, 33.0000]]
System.out.println(vec1.add(vec2));

// scalar add
INDArray vec3 = vec1.add(10);

// [[11.0000, 12.0000, 13.0000]]
System.out.println(vec3);
```

<sup>1</sup>一种使用 APT 在编译过程中替换运算符为相应函数的方法。

ND4J 在开源、分布式、支持 GPU 的库内，为 Java 带来了符合直觉的，类似 Python 编程人员所用的科学计算工具。它具有：

1. 多用途多维数组对象
2. 多平台功能，包括 GPU
3. 线性代数和信号处理功能

本节之后都将使用 ND4J 来演示算法，有兴趣的同学可以研究它的实现代码。

## 11.2 矩阵运算

矩阵是一个  $m \times n$  个数组成的  $m$  行  $n$  列的矩形表格。特别地，向量可以看作矩阵的特殊形式。对于  $1 \times n$  或  $n \times 1$  的矩阵，就是一个行向量或列向量。实际上，矩阵的加法/减法运算，也可适用于向量。尽管矩阵是多维度数据，却很少采用多维数组表示。如果向量和矩阵都使用一维数组表示的话，*add* 运算当然可以用与矩阵。

对于这样一个数组：int[] **numbers**={1,2,3,4,5,6}。

1. 可能是一个  $1 \times 6$  的矩阵，或是一个向量。
2. 可能是一个  $2 \times 3$  的矩阵
3. 可能是一个  $3 \times 2$  的矩阵

IntNDArray
array : int[] shape : int[]
add(v : IntNDArray) : IntNDArray mul(v : IntNDArray) : IntNDArray rank(): int isVector(): boolean

图 11.2: 类图：IntNDArray

为了让 IntVector 升级为 IntNDArray，势必要加入维度信息才行，记为 *shape*。而矩阵只有 2 个维度，如果不考虑更多维度的情况下，使用 int[2] 就可以。但你立即就会认识到，改成 int[] 有更好的可扩展性。使用 IntNDArray 创建向量就是创建  $1 \times n$  或  $n \times 1$  的矩阵。

Listing 11.4: 创建 NDArry

```
int[] array;  
int[] shape;  
  
// 默认: 1 x n  
public IntNDArray(int... array) {  
    this.array = array;  
    this.shape = new int[]{1, array.length};  
}
```

```
// 可以把1x6转换成2x3或者3x2的矩阵
public IntNDArray reshape(int[] shape) {
    this.shape = shape;
    return this;
}
```

根据数学定义，只有当矩阵 A 的列数等于矩阵 B 的行数时，A 与 B 才可以相乘。乘积 C 的第 m 行第 n 列的元素等于矩阵 A 的第 m 行的元素与矩阵 B 的第 n 列对应元素乘积之和。

$$c_{ij} = \sum_{k=1}^p a_{ik} * b_{kj} \quad (11.2)$$

其中，A 为  $m \times p$  的矩阵，B 为  $p \times n$ ，结果为  $mxn$  的矩阵 C。相应地，实现算法如下：

**Listing 11.5:** 矩阵乘法

```
public IntNDArray mul(IntNDArray ndArray) {
    final int ROWS = this.shape[0];
    final int COLS = ndArray.shape[1];

    int[] shape = new int[]{ROWS, COLS};
    int[] data = new int[ROWS*COLS];

    for(int i=0; i<ROWS; i++) {
        for (int j=0; j<COLS; j++) {
            for (int k=0; k<ndArray.shape[0]; k++) {
                int a_i_k = this.array[i*this.shape[1]+k];
                int b_k_j = ndArray.array[k*ndArray.shape[1]+j];
                data[i*COLS+j] += a_i_k * b_k_j;
            }
        }
    }
    return new IntNDArray(data).reshape(shape);
}
```

现在，我们就可以用一维数组记录矩阵了，不管维度如何变化 `reshape` 之后不需要变更 `array` 的内容。对于向量的点乘和叉乘，也可以借助矩阵处理。点乘比较容易解决，但叉乘需要反对称矩阵（anti-symmetric）辅助才能计算出结果。不过叉乘主要应用于几何和动量计算，在此就不再详细解释。

以下代码是 ND4J 的矩阵示例：

```
INDArray v1 = Nd4j.create(new float[]{1,2,3}).reshape(1,3);
```



```
INDArray v2 = Nd4j.create(new float[]{10,20,30}).reshape(3,1);
System.out.printf("v1v2=[%s]\n", v1.mmul(v2));
```

## 11.3 梯度下降

在机器学习里，梯度下降虽然不是什么高深的算法，但它却是机器学习的关键。经常会用到梯度下降法来进行训练，常见的有：批量梯度下降法 BGD，随机梯度下降法 SGD，小批量梯度下降法 MBGD。

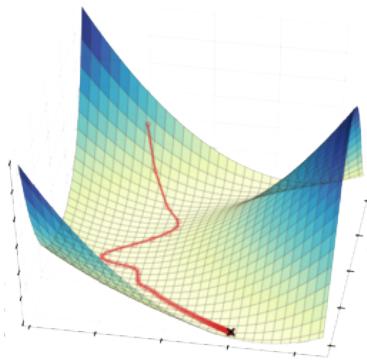


图 11.3: 梯度下降

梯度是方向上升/下降最快的方向，它的幅值代表陡峭的程度。所以，在最小值的地方，曲面轮廓几乎是平坦的，理论上最小值是梯度为 0。但事实上，我们又可能无法达到最小值，只能在最小值附近的平坦区域来回震荡。当在这个区域震荡时，损失（Loss）值几乎是我们能达到的最小值了，并且不会有很大的变化，因此我们是在真实的最小值附近跳动。通常，当损失值在预定的数字内没有改善的时候就会停止迭代，例如 10 次或者 20 次迭代。当这种情况发生时，就说训练已经收敛了，或者说收敛已经完成了。

### 11.3.1 BGD

使用 BGD(Batch Gradient Descent，批量梯度下降)，在目标函数为凸函数时，虽然可以找到全局最优解，但收敛速度慢，需要用到全部数据，因此内存消耗也大。因此，BGD 不适用于大数据集。其公式如下：

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}}$$

如公式所示，BGD 的策略就是朝着当前所在位置的坡度最大的方向前进。但它也有缺点，它在面对峡谷型函数时，效率会变得很低，呈现出震荡的姿态。

### 11.3.2 SGD

而 SGD 相当于 BGD 的升级版。SGD 被称为随机梯度下降 (Stochastic Gradient Descent)。正如它的名字所说，它首先通过 mini-batch 学习，意思就是从训练数据中随机选

择一部分数据(称为 mini-batch), 将这些 mini-batch 作为对象, 使用梯度法进行学习。其代码如下所示:

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .weightInit(WeightInit.XAVIER)
    .activation("relu")
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .updater(new Sgd(0.05))
    // ... other hyperparameters
    .list()
    .backprop(true)
    .build();
```

### 11.3.3 Momentum

Momentum 也是一种常见的优化算法, 也被称为 SGD with Momentum。恰如其名, 它为了抑制 SGD 的震荡, 在梯度下降过程中加入惯性。简单来说, 它就是在梯度越陡时, 下降越快。较平缓时, 下降较慢。其公式如下:

$$\nu \leftarrow \alpha \nu - \eta \frac{\partial L}{\partial W}$$

$$W \leftarrow W + \nu$$

如上式所示,  $W$  表示要更新的权重参数,  $\frac{\partial L}{\partial W}$  表示损失函数关于  $W$  的梯度,  $\eta$  表示学习率。而  $\nu$  这一变量便是用来模拟惯性的。其通过在 SGD 的基础上引入一阶动量, 这代表着现在下降方向不仅由当前点的梯度方向决定, 而且由此前累积的下降方向决定。

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .weightInit(WeightInit.XAVIER)
    .activation("relu")
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .updater(new Nesterovs(0.05))
    // ... other hyperparameters
    .list()
    .backprop(true)
    .build();
```

Momentum 的更新路径就如同小球在碗中滚动一样。与 SGD 相比, 其大大缓解了震荡问题, 原因就是它加入的一阶动量。即使在某一方向“受力”很小, 但因为其一直在同一方向受力, 所以它会朝着同一方向有一定的加速。通俗地讲, 就是它所加入的惯性, 抵消了试图改变它的力。

### 11.3.4 AdaGrad

在梯度下降中，学习率的值很重要（记为  $\eta$ ），而在学习率的相关研究中，有一种被称为 **学习率衰减** (learning rate decay) 的方法，随着机器学习的过程，使学习率逐渐减小。它具体表现为，在一开始与其他方法类似，进行参数学习，但在学习的过程中，当准确率越来越高时，便减少学习率。其公式如下：

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial \mathcal{L}}{\partial \mathbf{W}} \odot \frac{\partial \mathcal{L}}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial \mathcal{L}}{\partial \mathbf{W}}$$

如公式所示，AdaGrad 会为参数的每个元素适当地调整学习率，与此同时进行参数的学习。AdaGrad 的公式比起之前的，新出现了一个变量  $\mathbf{h}$ ，它代表着之前所有梯度值的平方和，在更新参数时，通过乘以  $\frac{1}{\sqrt{\mathbf{h}}}$ ，AdaGrad 便可以为每个元素调整适宜它的学习率。其代码如下所示：

---

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .weightInit(WeightInit.XAVIER)
    .activation("relu")
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .updater(new AdaGrad(0.05))
    // ... other hyperparameters
    .list()
    .backprop(true)
    .build();
```

---

AdaGrad 比起 Momentum 更好地抑制了 SGD 的震荡，函数的取值高效地向着最小值移动。刚开始时，也许还会有些震荡，但越接近最小值，几乎是呈直线般向着目标前进。

### 11.3.5 Adam

---

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .weightInit(WeightInit.XAVIER)
    .activation("relu")
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .updater(new Adam(0.05))
    // ... other hyperparameters
    .list()
    .backprop(true)
    .build();
```

---



## 11.4 概率分布

## 11.5 损失函数

在机器学习中，机器的学习需要某个指标来表示现在的状态，然后，以这个指标为基准，寻找最优权重参数。而在机器学习中所用的指标便被称为损失函数（loss function）。对于损失函数，我们主要使用均方误差和交叉熵误差。

首先，我们介绍一种常用的函数，交叉熵误差（cross entropy error）。

$$E = - \sum_k t_k \log y_k \quad (11.3)$$

在上式中， $y_k$  代表着机器学习的输出， $\log$  表示以 e 为底数的自然对数 ( $\log_e$ )。 $t_k$  代表着正确解标签，仅当解标签为正确时， $t_k$  的索引才为 1。其余情况都为 0。因此，E 所代表的实际为解标签为正确时所输出的自然对数。

如上图所示，当输出结果  $y_k$  越发趋向于 1.0 时，得到的 E 越发增大而趋向于 0。交叉熵误差通过 E 所表示的负值，来表明该权重参数与正确结果之间的偏差程度。

下面，我们用代码来实现交叉熵误差：

---

**Listing 11.6:** 交叉熵误差

---

下面我们介绍另一种，在损失函数中最有名的均方误差（mean squared error）。它的公式如下所示：

$$E = \frac{1}{2} \sum_k (y_k - t_k)^2 \quad (11.4)$$

在上式中 k 表示着数据处于哪一个维度， $t_k$  表示着各维度的监督数据。通常情况下，监督数据仅将正确解标签表示为 1，而其他非正确解则表示为 0。而机器学习输出的结果  $y_k$ ，则是会在各个维度都显示该维度是正确解标签的可能性。均方误差通过计算机器学习的输出和正确解监督数据的各个元素之差的平方，再求总和。从而得到该权重参数的输出结果与正确结果之间的偏差。

---

**Listing 11.7:** 均方误差

---

## 11.6 拟合算法

很多时候，我们希望通过一些样本来反应总体的特征，因此我们需要拟合曲线来判断总体的情况。假设有如下这些个样本，看起来各点分布趋于一条直线，因此我们希望通过一条直线来描述该样本所在总体的一些特征，对总体进行预测。一般的方法就是先假设一条直线，如  $L=ax+b$ ，之后再根据前面这些样本，确定最优的 a,b，所谓最优就是通



过这些点计算出合适的  $a, b$ , 使各个点对直线上垂直距离的平方和最小 (最小二乘法)。具体的方法是通过迭代来计算的。



# 第 12 章 机器学习

## 12.1 分类问题

分类问题就是将数据以一定的分类标准分为几簇，在本节中，我们将介绍几种分类方法。第一种分类方法是 SVM (Support Vector Machine, 支持向量机)，是机器学习中经典的算法。我们以简单的逻辑分类为例，如图12.1所示，我们需要找到一条直线，这样就能将所有的数据分为两类。

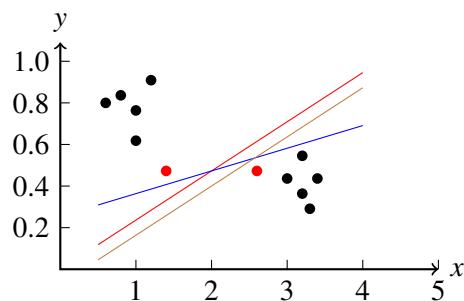


图 12.1：支持向量机示意图

如果不加以限制，这样的分类直线将有无数条。为了在众多的直线中找到最优的分类直线，我们要遵循间隔最大化原则，即分类直线离数据越远就是最优的，因为如果不是这样，分类直线会紧贴边界的数据；这时如果出现一个与边界数据略有不同的新数据，就很容易被分到错误的一边。同时，过于接近分类直线的数据点不具有太大的实际意义。想要达到分类直线离数据最远的效果，我们需要找到离分类直线最近的数据点，用它们与直线的距离来训练神经网络。这些数据点被称为支持向量，这也是支持向量机名字的由来。

还是以图12.1为例，图中两个红色的点为支持向量，也就是离分类直线最近的两个点。蓝线因为离两个支持向量都不是最远，所以不是最佳分类直线；棕线离上方的支持向量足够远，然而最佳分类直线应与所有支持向量保持最远距离，即所有支持向量离直线等距且最远，所以棕线也不符合要求。红线符合上述所有要求，时这个数据集的最佳分类直线。

分类问题的第二种分类方法是 RBM (Restricted Boltzmann Machine, 受限波兹曼机)。RBM 是一种无向图模型，具有两层结构：可见层 (V 层)，隐藏层 (H 层)，这两层的节点相互全部连接，但是每一层各自的神经元之间没有连接，因此被称为受限波兹曼机。波兹曼机是允许同一层神经元连接的。

另外，受限波兹曼机是二值化的，也就是说，其神经元的输出只有两种状态：激活和未激活，一般情况下我们分别用 1 和 0 表示。在图12.2所示的例子中， $n_v$  和  $n_h$  分别表示可见层和隐藏层的神经元数目；前面我们提到，受限波兹曼机的神经元只有激活和

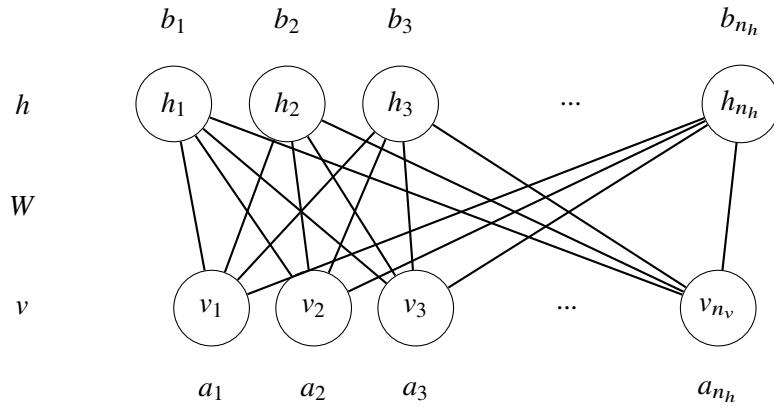


图 12.2: 受限波兹曼机模型结构

未激活两种状态，因此，我们可以用向量  $\mathbf{v}$  来表示可见层的状态向量，具体表达式为  $\mathbf{v} = (v_1, v_2, \dots, v_{n_v})^T$ ，其中  $v_i$  表示可见层中第  $i$  个神经元的状态。同理，隐藏层的状态向量也可以表示为  $\mathbf{h} = (h_1, h_2, \dots, h_{n_h})^T$ ，其中  $h_i$  表示可见层中第  $i$  个神经元的状态。可见层的偏置向量用  $\mathbf{a}$  来表示，具体表达式为  $\mathbf{a} = (a_1, a_2, \dots, a_{n_v})^T$ ，同样的，隐藏层的偏置向量为  $\mathbf{b}$ ，表达式为  $\mathbf{b} = (b_1, b_2, \dots, b_{n_h})^T$ 。

最后，连接隐藏层和可见层的权值矩阵用  $W$  来表示，表达式为  $W = (w_{i,j})$ ， $w_{i,j}$  代表隐藏层中第  $i$  个神经元与可见层中第  $j$  个神经元之间的连接权重。

有了这些前置参数，我们便可以探索受限波兹曼机的更多特性和用途。受限波兹曼机是一个能量模型 (Energy Based Model, EBM)，是由物理学能量模型演变而来；能量模型需要先定义一个合适的能量函数，然后基于这个能量函数得到变量的概率分布，最后基于概率分布去求解一个目标函数。受限波兹曼机的能量函数定义为：

$$E_\theta(\mathbf{v}, \mathbf{h}) = -\sum_{i=1}^{n_v} a_i v_i - \sum_{j=1}^{n_h} b_j h_j - \sum_{i=1}^{n_v} \sum_{j=1}^{n_h} h_j w_{j,i} v_i \quad (12.1)$$

如果写成矩阵的形式，则可改写为

$$E_\theta(\mathbf{v}, \mathbf{h}) = -\mathbf{a}^T \mathbf{v} - \mathbf{b}^T \mathbf{h} - \mathbf{h}^T W \mathbf{v} \quad (12.2)$$

相比于波兹曼机 (BM)，受限波兹曼机 (RBM) 因其具有快速学习的特点而被广泛地使用。与此同时，RBM 在分类，回归和图像特征提取上也得到了广泛的应用。

## 12.2 线性回归

线性回归是机器学习解决预测问题的重要途径之一。在进行线性回归算法时，我们通常会指定一个数据集，其中包括了正确答案。线性回归算法会根据给出的数据集去拟合一个函数，使之尽量符合所给出的数据集，同时达到预测更多正确答案的效果。（如图12.3所示）这种模式类似于老师给定多个参数和结果，让学生自己寻找函数并且不断地自我修正；同时老师也会监督并量化期望与实际输出间的误差。因此，线性回归算法

也被称为监督学习。

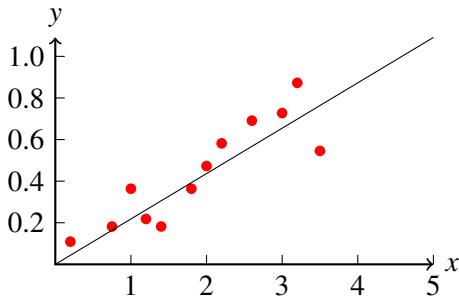


图 12.3：根据所给的数据集拟合函数，并根据此函数预测将来的数值

如图12.3所示，自变量  $x$  会与要预测的目标变量  $y$  建立一个函数关系。为了使结果更精确，我们可以指定特定的模型，决定是使用一次函数还是二次函数来更好地贴合数据集，从而达到更好的预测效果。

在 DL4J 中，线性回归算法主要有三个部分组成，模型的建立，过程的监听和数据集的构造。建立模型需要一些超参数。在下面的代码块中指定了一些超参数：第一是随机种子，由于每次进行神经网络训练时，函数的偏置和权重都是随机的，我们需要用种子来确保初始情况大致一致，使结果具有可复现性。第二是 OptimizationAlgorithm 和 updater，代表了模型的优化算法，分别决定了学习的方向和学习率的大小。第三是对模型权重的初始化。第四是对神经元的设置，第六是设置是否进行反向传播。

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_
    GRADIENT_DESCENT)
    .weightInit(WeightInit.XAVIER)
    .updater(new Sgd(learningRate))
    .layer(0, new OutputLayer.Builder(LossFunctions.LossFunction.MSE)
        .backprop(true)
        .build
```

在进行训练的同时我们也要对模型进行监听，主要用于对函数的损失函数进行监听，从而更好地对算法进行改进。

最后是数据集的构造。我们说过线性回归的数据集包括数据和结果两个部分。在这里，我们把自变量  $x$  称为特征，因变量  $y$  称为标签，首先先创建两个长度为样本数量的数据组，分别代表输入和输出数据的数据组；接着根据特征的值随机生成输入和输出的数，最后将其包装成一个类返回给模型，代码如图12.4所示。

当然，这里的线性回归只含有一个自变量和一个因变量，因此被称为一次线性回归或单变量线性回归。这种模型比较简单，用二维的  $xy$  坐标轴即可表示出来。在实际的操作中还会遇到还有多变量线性回归，在面对更多维度的参数能更好地进行处理。

```

private static DataSetIterator getTrainingData(int batchSize, Random rand) {
    double [] output = new double[SampleNum];
    double [] input = new double[SampleNum];
    for (int i= 0; i< SampleNum; i++) {
        input[i] = MIN_RANGE + (MAX_RANGE - MIN_RANGE) * rand.nextDouble();
        output[i] = 0.5 * input[i] + 0.1;
    }
    INDArray inputNDArray = Nd4j.create(input, new int[]{nSamples,1});

    INDArray outPut = Nd4j.create(output, new int[]{nSamples, 1});
    DataSet dataSet = new DataSet(inputNDArray, outPut);
    List<DataSet> listDs = dataSet.asList();
    return new ListDataSetIterator(listDs,batchSize);
}

```

图 12.4: 数据集的构造: 包括特征和标签

### 12.2.1 过拟合

对于机器学习来说，过拟合是一个十分常见的问题。过拟合具体是指经过机器学习所获得的权重参数，可以拟合训练数据，却在面对不包含在训练数据中的实际数据时，不能很好的拟合他们。机器学习的目的就是获得一个更加抽象，以及泛化的模型。即使是对未包含在训练数据中的数据，我们也希望能获得较好的拟合结果。所以我们需要通过一定方法来抑制过拟合。

过拟合产生的原因，主要有两点：1. 模型具有大量的参数，表现力强 2. 训练数据的数量很少

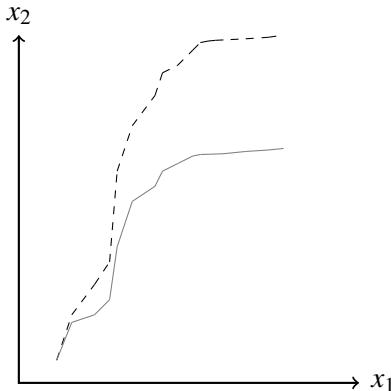


图 12.5: 过拟合情况

如图12.5所示，对于训练用数据来说，大约 100 个 epoch 之后，精度便可以达到接近 100%。但是，对于测试用数据来说，识别精度确只能达到 70% 左右，这并非我们所想要的结果。从上面的分析中可知，模型对训练时没有使用的数据拟合得并不好。

### 12.2.2 正则化

过拟合作为机器学习中一种常见的问题，自然也有解决的方法。基于本书主要介绍关于入门方面的知识，在此便不深入介绍。仅简单介绍一下权值衰减这一方法。

权值衰减是一种常见的用来抑制过拟合的方法。其通过在学习过程中对大的权重进行一定比例的惩罚，从而达到过拟合的目的。过拟合往往就是因为权重参数取值过大才发生的。因此权值衰减不失为一种简单而有效的办法。

实现权值衰减具体方法是为损失函数加上权重的平方范数（L2 范数）。机器学习的目的是减少损失函数的值，使用权值衰减后，便可以对变大的权重进行惩罚，从而达到抑制的作用。具体来说，如果将权重记为  $\mathbf{W}$ ，L2 范数的值便是  $\frac{1}{2}\lambda\mathbf{W}^2$ 。权值衰减便是将 L2 范数加到损失函数中。在 L2 范数中， $\lambda$  是控制正则化程度的参数。 $\lambda$  越大，对数值较大的权重施加的惩罚也就越重。下面我们用图片来具体解释：

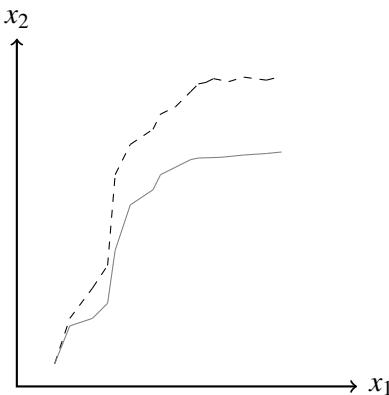


图 12.6：权值衰减方法

如上图所示，测试数据的识别精度没有太大变化。但训练数据和测试数据的识别精度之间的差距明显缩小了。这证明，过拟合收到了抑制，而训练数据和测试数据的整体识别精度的增加需要依靠其他的方法，并非权值衰减的作用范围。

## 12.3 逻辑回归

逻辑回归与线性回归相似，同样用于处理数据的分类问题。他们的不同之处在于逻辑回归的数据集中只包含数据，神经网络会自我学习，自行生成一套数据结构模式。在这种模式中，我们只需要提供数据，不需要提供数据集地的正确答案，其余部分均由神经网络完成对数据的分类。因此，逻辑回归也被称为无监督学习。

在下图的例子中，我们给出一个数据集（如图12.7）。模型并不知道这些数据的意义和用途，只会尝试着找出其中潜藏的数据结构。在这个例子中，逻辑回归算法会判定这个数据集从属于两个数据簇  $a$ ,  $b$ , 从而完成数据分类的任务。

在 DL4J 中，逻辑回归与线性回归配置类似，主要区别在于神经网络的最后一层输出层：在线性回归中，最后一个隐藏层是 Activation.IDENTITY，即不对数据作处理。而逻辑回归的最后一层为 Activation.SOFTMAX，指定了对当前数据进行分类任务。代码如下：

## 12.4 正向传播

正向传播也叫前向传播，是指数据从输入层到隐藏层，再到输出层的单向过程。数据会从输入层输入，经过隐藏层的一层层地变换，最终得出结果数据。如图12.8是一个典型的三层神经网络，我们便以此为例解释正向传播。

隐藏层  $H_1$  的值由输入层的值和权重所决定， $W_{(1,1)}$ ,  $W_{(2,1)}$ ,  $W_{(3,1)}$  分别代表输入数据  $I_1$ ,  $I_2$ ,  $I_3$  对于隐藏层  $H_1$  的权重。公式如下：

$$H_1 = I_1 W_{(1,1)} + I_2 W_{(2,1)} + I_3 W_{(3,1)}$$

同理， $H_2$  也可表达为相似的公式。经由隐藏层激活函数计算后得到该层的激活值。同时，输出层的值由隐藏层的值和权值决定， $W_1$ ,  $W_2$  分别代表各个隐藏层数据对于输出层的权重。 $b_1$  表示偏置，公式如下：

$$O_1 = H_1 W_1 + H_2 W_2 + b_1$$

这样一个由输入层至隐藏层，再由隐藏层至输出层的过程就被称为正向传播。如图12.8所示，只含有正向传播的神经网络可以用一个有向无环图来表示。只含有正向传播的神经网络被称为前馈神经网络，是现今发展较为成熟的一种神经网络。

## 12.5 反向传播

在正向传播中，神经网络中的数据单向向前传播；因此，我们无法准确知道每一步造成的误差大小，也无法根据误差对神经网络进行调整更新；反向传播就提供了另一种

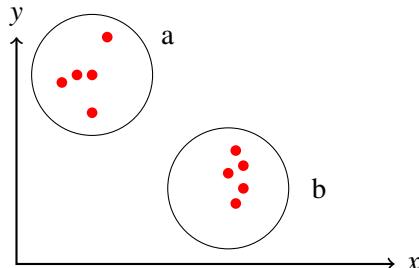


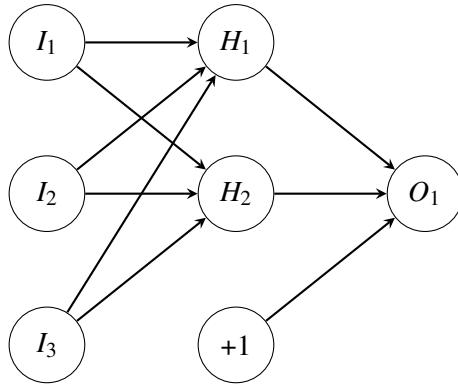
图 12.7：逻辑回归注重数据的分类

---

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .updater(new Nesterovs(learningRate, 0.9))
    .layer(0, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.SOFTMAX)
        .build())
    .build();

```

---



输入层      隐藏层      输出层

图 12.8: 经典的三层神经网络

途径来量化误差并更新权值。反向传播用于将代价函数最小化，将每一层的误差反向传播给上一层，由此将误差最小化，如图12.9。

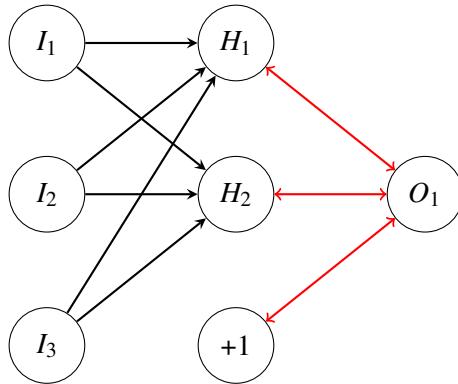


图 12.9: 具有反向传播的神经网络

为了计算每一层神经元所产生的误差，我们引入  $\delta_j^{(l)}$ ，代表第  $l$  层的第  $j$  个节点所产生的误差。由此可得最后一层输出层的误差公式为  $\delta_j^{(3)} = a_j^{(3)} - y_j$ ，其中  $a_j^{(3)}$  代表神经网络计算出的值，而  $y_j$  代表数据对应的真实值。

误差向前一层传播时，公式需更改为  $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} * g'(z^{(2)})$ ， $(\Theta^{(n)})^T$  代表第  $n$  层代价函数的转置， $z^{(n)}$  等于  $\Theta^{(n-1)} a^{(n-1)}$ ，最后  $g(z)$  代表该层的激活函数。依据这个公式，我们就而可以将误差逐层上传，并实现对神经网络参数的修改。

反向传播实际上与正向传播很相似。他们的计算方法类似，区别在于正向传播是从前向后计算，而反向传播是从后向前计算。其实， $\delta_j^{(l)}$  即是该层代价函数关于该层对应的所有输入单元加权和的偏导。以图12.9为例， $\delta_1^{(3)}$  可表达为  $\delta_1^{(3)} = \frac{\partial}{\partial z_1^{(3)}} \text{cost}(i)$ ，其中  $\text{cost}(i)$  代表该层的代价函数， $z_1^{(3)}$  等于  $a_1^{(2)} * W_{(1,2)} + a_2^{(2)} * W_{(2,2)}$ 。 $\delta_j^{(l)}$  衡量了神经网络计算中的中间值，代表了神经网络权重改变的程度大小，使我们对整个网络有更深入的了解。

反向传播只是正向传播的逆向过程。我们还是以图12.9为例，由上可知  $\delta_1^{(3)} = a_1^{(3)} - y_1$ ，而逆向到  $\delta_2^{(2)}$  的算式可表达为  $\delta_2^{(2)} = (W_{(2,2)})^{-1} * \delta_1^{(3)}$ 。这样的逆过程与正向传播的计算过

程区别不大；所以说，反向传播与正向传播在形式和计算方法上都很相似。

反向传播广泛地应用于循环神经网络，例如机器翻译、语言理解等前沿应用。这种结构的网络往往存在记忆单元和注意力机制，要求复杂的求导计算，例如 Google 语音助手的上下文理解功能。反向传播促进了记忆神经网络的发展，目前正在探索和发展中。

## 12.6 全连接网络

全连接层网络（MLP, Multilayer Perceptron）又叫多层感知机，特点是上一层的所有神经元都与下一层的所有神经元相连接，因此又叫全连接层。如图12.8就是一个简单的全连接层，最底层是输入层，中间是隐藏层，最后是输出层。在计算第  $n$  层的某个节点的时候，输入的参数是第  $n-1$  层的所有节点的加权。

从图12.10可以看出，全连接网络的所有参数就是各个层的权重和偏置。确定并最优化这些参数就是一个最优化的问题，最简单的就是利用梯度下降（SGD），首先随机初始化所有参数，然后进行迭代训练，不断地更新梯度和参数，直到满足某个条件为止。全连接网络应用广泛，例如 MNIST 手写数字识别等。

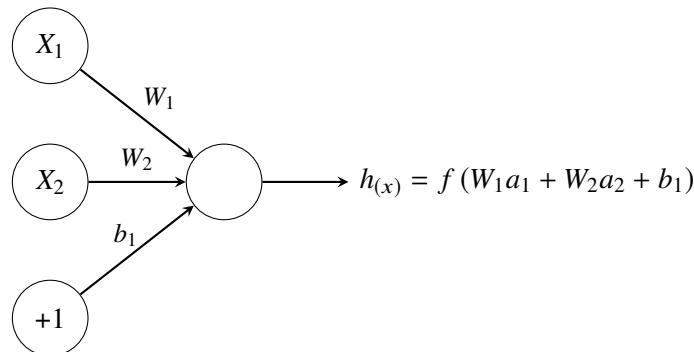


图 12.10：全连接层中的一个节点

# 第 13 章 有监督学习

---

机器学习分为：监督学习，无监督学习。本章先介绍监督学习（supervised learning），它从给定的训练数据集中学习出一个函数，用于对新数据的预测。监督学习的训练集包括输入和输出，也即是特征和目标。训练集中的目标是由人标注的，利用这些已知数据和其输出训练得到一个最优模型（目标函数）。监督学习是训练神经网络和决策树的常见方法。最典型的算法是 KNN 和 SVM，用于回归分析和统计分类。

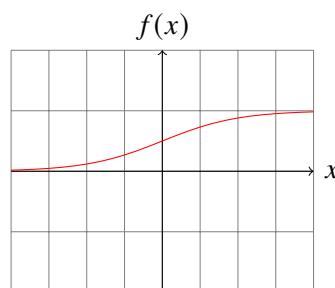
## 13.1 训练过程

有监督学习最常见的就是：回归（Regression）和分类（Classification）。一般来说，定量输出称为回归，或是对连续变量预测；定性输出称为分类，就是对离散变量进行预测。K-近邻（KNN）是一种分类算法，而之前提到的线性回归，就是利用已知数据集求线性函数，使其尽可能地拟合数据，也就是让损失函数最小。

## 13.2 回归问题

回归分析分为：线性回归、逻辑回归（Logistic Regression）。实际上，逻辑回归就是对线性回归的输出又进行了一次特殊函数的处理，使其输出一个分类可能性的概率，这个特殊的函数通常是 sigmoid 函数：

$$y = \frac{1}{1 + e^{-x}}$$



Sigmoid 函数是机器学习非常重要的一个函数，当  $x > 0$  时，Sigmoid 函数大于 0.5；当  $x < 0$  时，Sigmoid 函数小于 0.5。因此，把拟合曲线的函数值带入 Sigmoid 函数，比较  $f(x)$  与 0.5 的大小就可判断其分类。总的来说，Sigmoid 函数具有以下几个良好性质：

1. 单调可微，具有对称性。
2. 便于求导。
3. 定义域为  $[-\infty, +\infty]$ ，值域为  $(0, 1)$ ，可将任意值映射为概率。

### 13.2.1 线性回归

之前章节介绍过线性回归的数学模型，并尝试用梯度下降法，找到线性函数的参数，来拟合数据集。所以实现线性回归的代码，得首先实现梯度的算法函数，分为：SGB、BGD 和 MBGD。

因为批梯度下降是使用所有的数据样本拟合出来的假设函数，所以计算梯度也就会涉及到所有的样本数据，这种情况就是所谓的 BGD (Batch Gradient Descent)。结合最小二乘法的定义，定义损失函数  $J(\theta) = \frac{1}{2} \sum_i^m (y_i - h_\theta(x^i))^2$  现在问题就转化为求解最优的  $\theta$ ，使损失函数  $J(\theta)$  取最小值。

$$\theta_j := \theta_j - \alpha \frac{\delta}{\delta \theta_j} J(\theta)$$

我们先随便给  $\theta$  一个初始化的值(多默认为 1)，然后改变  $\theta$  值让  $J(\theta)$  的取值变小，不断重复改变  $\theta$  使  $J(\theta)$  变小的过程直至  $J(\theta)$  约等于最小值。此算法也称为最小均方算法 (Least mean square, LMS 算法)。

$$\theta_j := \theta_j + \frac{1}{\alpha} \sum_i^m (y_i - h_\theta(x^i)) x_j^i$$

使用伪代码表示如下，：

---

```
repeat{
     $\theta_j := \theta_j + \frac{1}{\alpha} \sum_i^m (y_i - h_\theta(x^i)) x_j^i$ 
    for every j=0, ... , n)
}
```

---

其中  $\alpha$  称为步长 (learning rate)，控制  $\theta$  每次向  $J(\theta)$  变小的方向迭代的幅度。由以上伪代码，也容易发现 BGD(批梯度下降) 每次迭代都得把所有样本计算一次，计算量很大。把整个过程分解一下，包含：初始化、求偏导和梯度下降。

Java 代码，大致如下。

---

```
//  $h(x) = \theta_0 * x_0 + \theta_1 * x_1 + \theta_2 * x_2$ 
public class LinearRegression {
    public LinearRegression(double[][] data, double alpha, int iteration){}

    private void initialize_theta(){}
    public void train(){}
    private double[] partial_derivative(){}
    private double partial_derivative_of_theta(int j){}
    private double h_theta_x_i_minus_y_i_times_x_j_i(int i, int j){}
}
```

---



### 13.2.2 逻辑回归

Sigmoid 的其中一个优点就是容易求解导数，利于编码实现。推导过程略，相应的实现代码如下：

$$y = \frac{1}{1 + e^{-x}}, y' = y * (1 - y)$$

---

```

public static double sigMoid(double value) {
    double e_x = Math.pow(Math.E, -value);
    double result = 1 / (1 + e_x);
    return result;
}

public static double sigMoidDerivative(double value) {
    double A = sigMoid(value);
    double B = 1 - sigMoid(value);
    double result = A * B;
    return result;
}

```

---

## 13.3 SVM

在机器学习章节中，我们提到过利用 **SVM**（支持向量机）作为数据分类的一种方法，并提到了 **SVM** 的实质其实就是正确分类数据的直线，并且分类直线与支持向量的间隔较远。在这样的基础上，我们有必要对坐标系内点与直线的距离进行研究。假设  $\mathbf{w}_i$  为分类直线的函数， $\delta_i$  代表坐标与分类直线的距离， $\|\mathbf{w}\|$  代表向量  $\mathbf{w}$  的范数<sup>1</sup>，由此我们可以得到一般性的点到直线距离公式：

$$\delta_i = \frac{1}{\|\mathbf{w}\|} |g(x_i)| \quad (13.1)$$

在二维坐标中，利用解析几何的知识，我们可以将公式简化为更容易理解的形式： $D = (Ax + By + c)/\sqrt{A^2 + B^2}$ ，其中  $Ax + By + c$  就相当于  $g(x_i)$ ， $\sqrt{A^2 + B^2}$  相当于  $\mathbf{w}$  的范数。这个公式的代码实现如下：

$$ax + by + c = 0$$

---

```

public double getDistance(double x, double y){
    double denominator = a * x + b * y + c;
    double numerator = Math.sqrt(a * a + b * b);
    double result = denominator / numerator;
    return result;
}

```

---

<sup>1</sup>范数是对向量长度的度量，如果一个向量可以表示为  $\mathbf{W} = (w_1, w_2, w_3, \dots, w_n)$ ，那么该向量的范数则为  $\|\mathbf{w}\|_n = \sqrt[n]{m_1^n + m_2^n + \dots + m_n^n}$

}

在界定了分隔直线后，空间就被分成了  $g(x_i) > 0$  和  $g(x_i) < 0$  两个部分，以此便可以判断数据类型的不同。与此同时，根据我们在机器学习章节中提出的要求，最佳直线应该与所有支持向量等距且最远；因此，我们我们需要对最佳直线与支持向量间的距离进行研究。

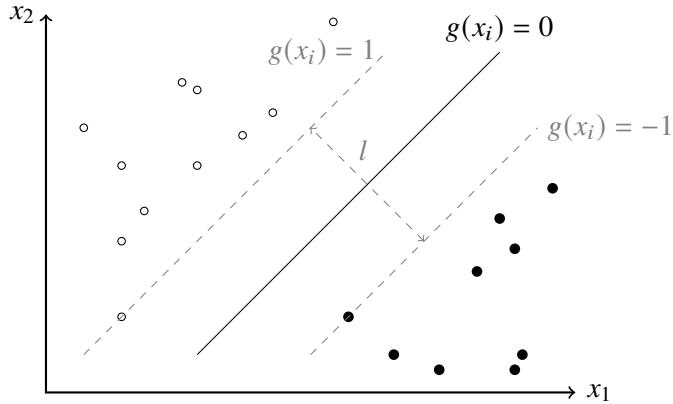


图 13.1: 二维向量的线性分类器

在这里，我们构建了名为“线性分类器”的模型来解决这一问题，如图??。这个问题的关键在于比较支持向量与分类直线的距离，那么我们不妨设当  $g(x_i) > 0$  时，直线上方的支持向量经过直线  $g(x_i) = 1$ ；同理， $g(x_i) < 0$  时，直线下方的支持向量经过直线  $g(x_i) = -1$ ，这样假设方便我们进行进一步的处理。我们可以得出  $g(x_i) = 1$  和  $g(x_i) = -1$  之间的距离，即  $l = 2/\|w\|$ 。同时，SVM 的目标是在对数据进行正确分类的情况下，最大化分类直线和支持向量的距离；由此，我们可以得到以下关系：

$$y_i (wx_i - b) \geq 1$$

$$\max(2/\|w\|)$$

可以看到最大化  $2/\|w\|$ ，也就是最小化  $\|w\|$ ；同时，利用高等数学的知识，我们能对其进行等价转化，这样就得到了 SVM 的基本形式：

$$y_i (wx_i - b) \geq 1$$

$$\min \frac{1}{2} \|w\|^2$$

最后，通过拉格朗日对偶，我们能够将上面的式子进一步转化为拉格朗日对偶性问题。在这个问题中我们通过添加拉格朗日乘子  $\alpha_i \geq 0$ ，写出拉格朗日函数和对应的目标函数：

$$L(w, b, \alpha) = \frac{1}{2} \|w\|^2 - \sum_{i=1}^n \alpha_i (y_i (wx_i - b) - 1)$$



$$\min_{w,b} \max_{\alpha \geq 0} L(w, b, \alpha)$$

在实际的 SVM 应用中，向量的维度往往不止二维，会出现三维甚至更多维度。线性函数在不同维度的空间中有不同的形式，例如在一维空间中是一个点，二维空间中是一条直线，三维空间是一个平面，如果不关注维数，线性函数又统称为超平面；此时我们研究函数的工具可能有变化，但线性分类器的基本思路仍然不变。我们依然可以通过基本形式和拉格朗日函数进行求解，这种方法具有普遍性。

## 13.4 卷积神经网络

卷积神经网络（Convolutional Neural Network，简称 CNN）属于前馈神经网络。与全连接神经网络不同，卷积神经网络只会计算上一层神经网络的部分单元，而全连接网络会计算所有单元。在大型图像处理上，卷积神经网络具有极大的优势；以常见的图像识别为例，神经网络的输入就是图像中的各个像素点，全连接网络的做法是将所有像素点不加以区分地全部输入下一层计算；然而图像识别的关键在于找到图像中的相邻或成群的像素点集，从而找到图像中各部分的“边缘”，全连接网络囫囵吞枣的做法显然效率不高。卷积神经网络则利用滤波器（Fliter）预先将图像中相邻像素之间的“边缘”过滤出来，从而大大提高了效率。

### 13.4.1 卷积层

### 13.4.2 池化层

### 13.4.3 全连接层

### 13.4.4 LeNet

## 13.5 手写识别示例

# 第 14 章 无监督学习

---

机器学习主要分为三类：监督学习（supervised Learning）、增强学习（reinforcement learning）、无监督学习（unsupervised learning）。也存在半监督学习（semi-supervised learning）这种情况，但在此不予讨论。

简单来说，区分监督学习和无监督学习的方法就是输入数据是否有标签（label）。如若没有标签则为无监督学习。我们以机器学习中的分类（classification）算法来举例，对于分类算法来说，输入的训练数据有特征（feature），有标签（label）。所谓的学习，其本质就是找到特征和标签间的关系（mapping）。而去判断一个分类算法的成功与否的方法，便是当我们输入有特征无标签的未知数据时，能否通过已知关系（参数）得到未知数据标签。

举个简单的例子，有监督学习相当于刷算法题的时候你知道答案，而无监督学习相当于你根本就没有答案，只能靠自己摸索。所以，与我们的常识相符，无监督学习的准确度往往比起有监督学习要低得多。那也许你会问了，既然无监督学习准确度那么低，为什么我们还要用它呢？那是因为在实际情况下，标签的获取需要极大的人工量，还不能保证变迁的准确性。所以，无监督学习也是十分重要的。

## 14.1 聚类算法

聚类是数据挖掘中的概念，其定义为按照某个标准把一个数据集分隔成不同的类或簇，使得相同类里的数据相似程度尽可能大。反之，不在同一个类里的数据，其差异也要尽可能大。“物以类聚，人以群分”就能大概指代聚类算法的意思。

讲到这里，我们需要区分聚类（clustering）与分类（classification）之间的区别。对于聚类来说，我们在聚类时，并不关心其中某一类是什么，我们需要实现的目标只是把相似的东西分到一起。因此，对于聚类算法来说，只要知道相似度衡量的标准就可以开始训练了。与之相对的，分类算法需要你告知它，它需要将数据分成哪些具体的类别，所以分类算法需要从训练集中进行学习，从而明白如何对未知数据进行分类。这就是聚类和分类的区别。

聚类的步骤主要如下：1. 数据准备：将数据转换为标准输入形式，使用特征标准化等方法；2. 特征选择：从最初的特征中选择最有效的特征，并将其存储于向量中；3. 特征提取：通过对所选择的特征进行转换，得到新的重点特征；4. 聚类：首先选择合适特征类型的某种距离函数进行接近程度的度量，而后执行聚类或分组；5. 聚类结果评估：对聚类结果进行评估，评估主要分为3种：外部有效性评估、内部有效性评估和相关性测试评估。

聚类主要分为层次化聚类算法，划分式聚类算法，基于密度的聚类算法，基于网格的聚类算法，基于模型的聚类算法等。下面我们就挑出几个重要的算法为大家进行讲解。

### 14.1.1 K-means 算法

K-Means 算法的特点是类别的个数是人为给定的，其假设数据之间的相似度可以使用欧氏距离度量，如果不能使用欧氏距离度量，要先把数据转换到能用欧氏距离度量。

接下来，我们简单介绍一下流程：首先，我们有在  $n$  维向量当中的一堆点，这里以二维空间为例。

接着我们随机生成  $k$  个聚类中心点，相当于将其分为几个类别。然后分别计算每一个数据点到这些中心的距离，把距离最短的那个当成自己的类别。这样就可以发现每个点都已经被分类了（有一个中心点），但是并不准确。

接下来就是无监督学习，使得分类变得更加准确的时候。我们一开始随机确定的分类点，这时候就要变化了。而它变化的标准就是“收复”附近的点，所以它将往所有它这一类别的点的坐标平均值移动，也就是移向中心。而到达中心后，将再一次判断各个点到  $k$  个中心点的距离，选取离每个点最近的中心点作为它的类别，以此类推。

伪代码流程如下所示：

---

```

public static double K-Means(输入数据, 中心点个数K){
    获取输入数据的维度Dim和个数N
    随机生成K个Dim维的点
    while(算法未收敛){
        对N个点: 计算每个点属于哪一类。
        对于K个中心点:
            1, 找出所有属于自己这一类的所有数据点
            2, 把自己的坐标修改为这些数据点的中心点坐标
    }
    return 结果
}

```

---

接下来，我们来说明一下 k-means 算法使用过程中有可能会遇到的问题。1. 测量距离的方法并非一定要使用欧氏空间这个方法，只需满足以下条件都可以用：首先有个分类两个点的方法的算符记作

$$\langle \vec{a}, \vec{b} \rangle$$

， 并且其具有交换性  $\langle \vec{a}, \vec{b} \rangle = \langle \vec{b}, \vec{a} \rangle$ 。其次需要可以求一堆点的平均值的算法（求中心点）： $\vec{\mu} = \text{Mean}(\vec{a}_1, \dots, \vec{a}_n)$  求出后只需满足： $\sum_{i=1}^n (\vec{\mu} - \vec{a}_i)^2$ 。

2. 如何知道是否收敛？使用代价函数： $J = \sum_{i=1}^C \sum_{j=1}^N r_{ij} \times v(x_j, \mu_i)$ 。其中： $v(x_j, \mu_i) = \|x_j - \mu_i\|^2$ 。代价函数的差分值小于一定数值的时候（ $N$  次越不过最小值点）即可认为是收敛了。

3. 代价函数不收敛，怎么办？首先说一下什么时候容易发生震荡：在数据点个数比较少而且比较稀疏的时候容易发生这种事情，发生的原因大约有两种常见的：1、陷入某个环里，然后开始震荡，它将会绕着中心点进行低频震荡。2、两个点互相交换，每次交换不改变  $J$  的值就收敛了，如果交换以后不幸影响了其它的点，就出现了高频震荡。这



个时候给出一种简单的解决方案：阻尼。简而言之，就是更新自己位置的时候考虑一下原来的位置，一般阻尼比（在 0 1 之间取值）决定收敛速度，收敛的慢了也就不容易震荡，也就越容易陷入局部极小值，也就是说，不震荡的情况下我们应该把阻尼比尽可能取小一点  $\vec{C}^{upd} = \vec{C}^{new} \times (1 - \xi) + \vec{C}^{old} \times \xi$   $\vec{C}^{new}$  是最后中心点的取值， $\vec{C}^{new}$  是当前集合的中心点， $\vec{C}^{old}$  是原来的中心点坐标。

### 14.1.2 DBScan 算法

## 14.2 DL4J 示例



## 第 15 章 神经网络

---

### 15.1 评价标准

准确率、召回率、精准率

# 第 16 章 深度学习

由神经元提出的感知机模型，只适用于解决二分类问题，对于多分类却无能为力，而多层次感知机却能解决这个问题。

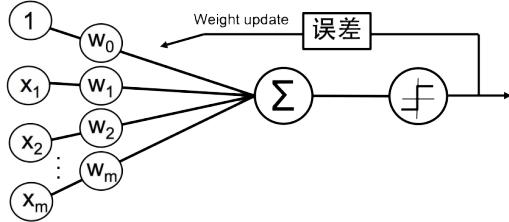


图 16.1：感知机

我们知道感知机可以实现 AND、OR、NAND 逻辑，而 XOR 正好可以由它们推算得出：  
 $x_1 \oplus x_2 = (x_1 | x_2) \& (x_1 \bar{\&} x_2)$

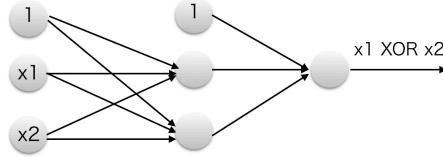


图 16.2：异或

这就是深度学习的魅力，但为什么在感知机提出来之后却停滞发展二十年呢？对于普通的二分类问题，使用的损失函数是基于点线距离的，但毕竟不是一个通用的解决方案。用于解决多分类问题的时候，就会显得非常笨重，越来越不好用。实际上，感知机和 SVM 非常相似，主要差别在于超平面（分割线）是否唯一，是否满足间隔最大化。感知机是不要求间隔最大的，只要能分开数据就算解决了。

## 16.1 多层感知机

本节以 XOR 为例，作为深度学习的入门铺垫。已知，单层感知机是可以实现 AND、OR、NAND 逻辑功能的。如下图：

感知机的参数如上，我们定义了 AND 和 OR 的运算模型，而 NAND 是 AND 相反运算，只要把上图 AND 的参数都取反即可： $(0.5, 0.5, 0.7) \Rightarrow (-0.5, -0.5, 0.7)$ 。示意图，如下。

感知机的最大缺点是误差计算是基于经验的（点到直线距离），而不是基于输出结果。如果误差的调节，能反应到  $w$  和  $b$ ，势必要求整个调整过程是可微的，也就是一个平滑的过程。这个平滑的调节就是基于偏导数实现的，也就是梯度。因此，这也要求从输入到输出，从误差到输入是一个可以微调的平滑过程，这样才有利于机器学习。实际上，这

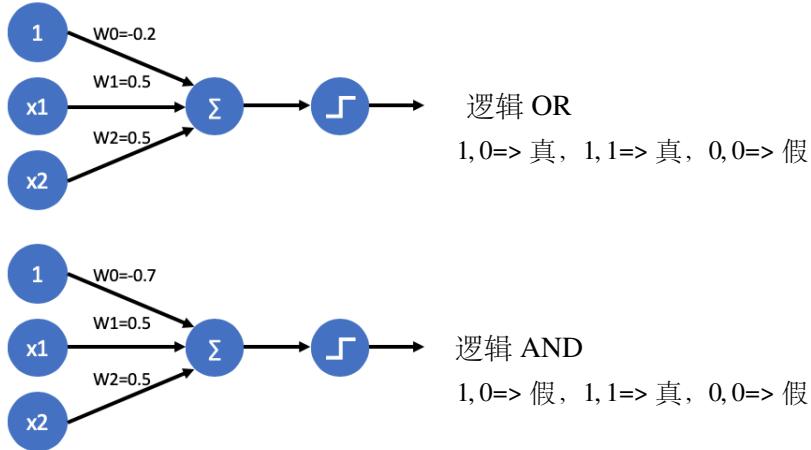
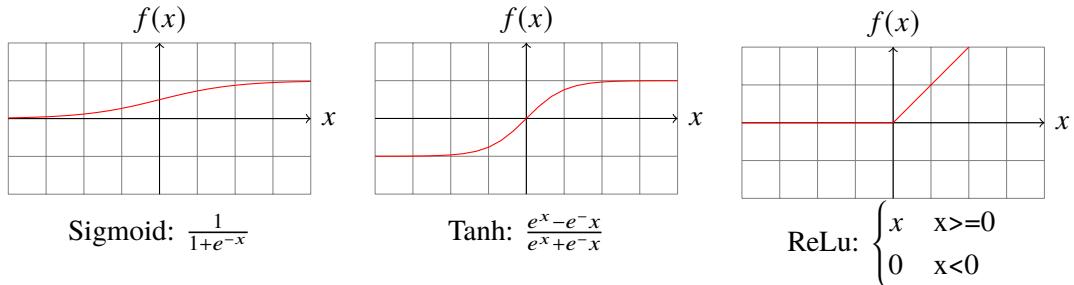


图 16.3: 感知机模型: AND、OR



图 16.4: 多层感知机求解异或

就是一个迭代求极值点的过程，直到误差最小或者迭代次数到达。所谓的前向传播和反向传播，就是这个道理。



感知机的缺点来源于激活函数是阶跃函数 *Sign*，它不是连续可导的函数，无法微调误差。换成 *ReLU*, *Sigmoid*, *TanH* 就可以让感知机重获新生。输出结果不再是 0 和 1 而是 (0, 1) 之间的概率值，代表更应该是哪个结果。

## 16.2 PlayGround

PlayGround 是一个在线演示神经网络的平台，是一个入门神经网络非常直观的网站。它图形化地展示神经网络的训练过程，非常有利于初学者获得感性认识。PlayGround 演示页面由 DATA (数据)、FEATURES (特征)、HIDDEN LAYERS (隐含层)、OUTPUT (输出层)，四个部分组成。

在 DATA 一栏里提供了 4 种不同形态的数据，分别是圆形、异或、高斯和螺旋。还可以对这些数据进行配置：噪声、训测比、批大小 (Batch)。平面内的数据分为蓝色和黄

色两种。



# 第三部分

## 人工智能应用



在人工智能的早期，那些对人类智力来说非常困难、但对计算机来说相对简单的问题得到迅速解决，比如，那些可以通过一系列形式化的数学规则来描述的问题。人工智能的真正挑战在于解决那些对人来说很容易执行、但很难形式化描述的任务，如识别人们所说的话或图像中的脸。对于这些问题，我们人类往往可以凭借直觉轻易地解决。

线性回归是机器学习中最基本的一个算法。回归分析中，又依据描述自变量与因变量之间因果关系的函数表达式是线性的还是非线性的，分为线性回归分析和非线性回归分析。

# 第 17 章 ND4J

---

## 17.1 ND4J 介绍

Nd4j 是一个基于 JVM 的，接口和 numpy 接近的张量运算库。目前流行的数据分析框架，大多是基于 Python 或 MatLab 开发的，很难移植到 JVM 运行环境。虽然有 Colt 这样的 Java 类库，但它对商业不友好。

*ND4J 主要特点：（摘自官网）*

1. 多用途多维数组对象
2. 支持多平台扩展，包括 GPU
3. 提供线性代数和信号处理功能

使用 ND4J 创建多维向量和矩阵非常容易，用起来和 numpy 非常类似，甚至还提供了 ND4S 这种更加接近 numpy 的类库。

创建 2X2 的 NDArray:

---

```
INDArray arr1 = Nd4j.create(new float[] {1,2,3,4}, new int[] {2,2});  
System.out.println(arr1);
```

---

输出： [[1.0 ,3.0] [2.0 ,4.0] ]

## 17.2 使用 ND4J

在学习使用 ND4J 之前，有必要了解数组在内存中的组织方式。数组通常是一段连续的内存，但 Java 并不能保证整个维度是连续的，这也很容易证明：

**Listing 17.1:** Java 二维数组

---

```
int[][] numbers = new int[2][];  
int[] N1 = {1,2,3};  
int[] N2 = {2,3,4};  
  
numbers[0] = N1;  
numbers[1] = N2;
```

---

代码 17.1，展示了 N1 和 N2 是 2 个不同的 int[] 对象，但无法确定数据在内存中的组织是否连续。Java 这种分散的数据组织范式，非常不利于批量运算和 GPU 加速。为了更有效率的使用数据，ND4J 没采用 Java 的多维数组形式来表示张量（标量、向量、矩

阵), 而是在 JVM 之外申请的内存。不仅有更好的性能, 还可以结合更多的 Native 加速线性代数库, 譬如 BLAS。<sup>1</sup>

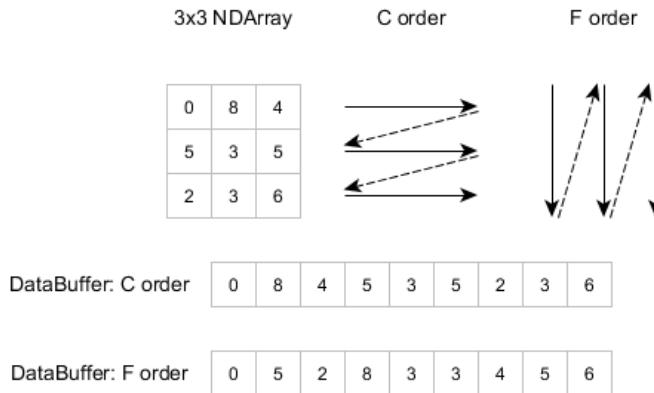


图 17.1: C 序和 F 序

图 ??展示了这种不同。创建 NDArray 的时候, 可以指定使用哪种方式。

### 17.2.1 创建和更新

与前面 IntVector 不同的是, NDArray 的数据只有 double 或 float 类型, 并且所有的 NDArray 共享相同的 dtype。默认情况下, NDArray 的数据都是 float 类型的。创建 NDArray 有很多方式, ND4J 还为全 0 和全 1 数组提供了便利接口:

---

```
Nd4j.zeros(int...)
Nd4j.ones(int...)
```

---

其中, int... 是维度信息。

创建一般的 NDArray 则可以用章节一开始时提到的 create() 方法, 传入两个数组作为数据组和维度即可。

另外, 对于元素服从正态分布的 NDArray, 或是元素属于 (0,1) 区间的 NDArray, ND4J 都提供了相应的接口:

---

```
Nd4j.rand(int, int);
Nd4j.randn(int, int);
```

---

在 DL4J 中, 更新操作也很便利, 利用 putScalar 和 putRow 方法, 传入需要改变元素的位置与值, 即可完成操作, 例如下面的代码, putScalar 方法将矩阵下标为 (1,3) 的元素修改成 100, putRow 方法则将第一行的所有元素重新设置成传入的 INDArray 中对应的元素, 这样就完成了元素的更新。

---

```
arr1.putScalar(1, 3, 100);
arr1.putRow(1, INDArray);
```

---

<sup>1</sup>BLAS: Basic Linear Algebra Subprograms, 目前最快的是 Intel 的 MKL

## 17.2.2 切片操作

对于 `NDArray`，我们往往需要其中的一部分来进行操作，这就需要用切片操作来完成。切片操作的代码如下图所示，我们传入一个或多个行数信息，就能获得对应行数的 `NDArray`。但是需要注意的是这两种方法只适用于二维矩阵。

```
arr1.getRow(int);  
arr1.getRows(int...);
```

但在日常使用中，我们对 `NDArray` 的切片要求往往更加复杂，仅仅使用上述的两个方法会比较困难。例如我们对一个 `NDArray` 进行取前三行，前三列的操作时，仅仅用 `getRow` 和 `getRows` 方法就显得比较麻烦。这时我们就需要用 `get` 方法进行操作：

```
arr1.create(new float[]{1,2,3,4,5,6,7,8,9}, new int[]{3,3});  
INDArray arr2 = arr1.get(NDArrayIndex.interval(0,2), NDArrayIndex.interval(0,2)  
);  
System.out.println(arr2);
```

输出结果为 `[[1.0,2.0] [4.0,5.0]]`，这就完成了对 `NDArray` 的切片操作。

## 17.2.3 基本运算

`NDArray` 的基本运算包括张量操作和元素操作。其中张量操作包括加减乘除四则运算，操作对 `NDArray` 中的所有元素进行，在调用时传入一个数即可，例如：

```
double myDouble = 1.0;  
INDArray arr1 = Nd4j.create(new float[]{1,2}, new int[]{1, 2});  
NDArray arr2 = arr1.add(myDouble);  
arr1.sub(myDouble);  
arr1.mul(myDouble);  
arr1.add(myDouble);  
System.out.println(arr2);
```

输出结果为 `[[2.0,3.0]]`。

元素操作与张量操作类似，都是对 `NDArray` 中的元素进行运算操作，不同的是张量操作传入的是一个数，所有元素执行同一个操作；而元素操作传入一个 `NDArray`，每个元素与另一个 `NDArray` 中对应的元素进行操作。因此，元素操作要求进行操作的两个矩阵形状一致。

```
INDArray arr1 = Nd4j.create(new float[]{1,2}, new int[]{1, 2});  
INDArray arr2 = Nd4j.create(new float[]{2,3}, new int[]{1, 2});  
System.out.println(arr1.add(arr2));
```

输出结果为 `[[3.0,5.0]]`。

## 17.3 线性回归示例



## 第 18 章 DL4J

---

### 18.1 DL4J 介绍

Deeplearning4j(简称 DL4J) 是为 Java 和 Scala 编写的首个商业级开源分布式深度学习库。DL4J 还可与 Hadoop 和 Spark 集成, 为商业化提供了很好的支持, 而不仅仅是学术研究工具。而在使用 Deep Learning 之前, 我们还要先确定问题的类型, 是有监督学习还是无监督问题? 对于有监督学习, 你需要先提供有标签的数据进行训练; 对于无监督学习, 需要能自主检测数据的相似性和异常状况。

还要考虑处理的特征数量有多少? 特征数量越多, 需要的内存也越大。就图像而言, 第一层的特征数量等于图像所包含的像素数。所以 MNIST 数据集中的  $28 \times 28$  像素的图像有 784 个特征。医疗诊断中的图像则可能有 14 兆像素。



主流的深度学习框架有 TensorFlow、Caffe、Keras、Theano 等, 它们大多在单节点服务器通过 GPU 加速完成模型训练。随着大数据时代的来临, 使用分布式计算极大提高了计算性能。因此将分布式计算与深度学习结合成为必然趋势。DL4J 就是为此而生, 利用 Spark 在多服务器多 GPU 上进行分布式的深度学习模型训练, 让模型跑得更快。运用 DL4J 高效的训练一个完整神经网络模型包括:

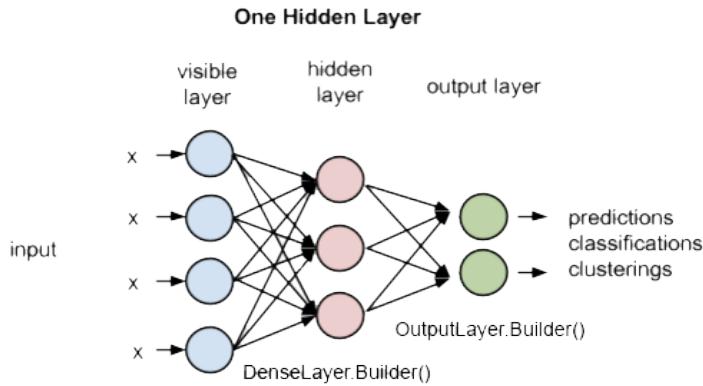
1. 预加载数据, 对数据进行预处理。DL4J 提供了 *DataVec* 简化数据采集过程。
2. 超参数配置, 各种优化算法和学习率、激活函数等。
3. 构建神经网络, 从单层到多层神经网络的构建。
4. 模型训练, 借助 ND4J 和分布式计算, 高效地训练模型。
5. 模型评估和保存。这正是 DL4J 的核心算法部分。

所以, 在开始训练之前, 要先花点时间学习如何使用 DL4J 的构建神经网络以及加载数据。由于加载数据的过程 (*reshape*、正则化、归一化等) 很容易引入意想不到的错误, DL4J 提供了 *DataVec* 类对数据加工预处理, 不仅可以减少错误, 还支持多种数据源, 譬如图片、CSV、ARFF 和纯文本等。

### 18.2 构造网络

DL4J 的神经网络都是用 *NeuralNetConfiguration* 构建的, 用它可以设定网络的层 (layer) 以及超参数。典型的神经网络包含: 输入和输出层, 以及隐藏层。对于全连接

层，在DL4J中也称为 *DenseLayer*。



## 1. 设置超参数

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(rngSeed)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .iterations(1)
    .learningRate(0.006)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .regularization(true).l2(1e-4)
    .list()
```

超参数 (Hyperparameter) 决定了神经网络学习方式的变量，包括模型的权重、激活函数、学习率和优化算法等。

## 2. 构建神经网络层

```
.layer(0, new DenseLayer.Builder()
    .nIn(numRows * numColumns) // Number of input datapoints.
    .nOut(1000) // Number of output datapoints.
    .activation("relu") // Activation function.
    .weightInit(WeightInit.XAVIER) // Weight initialization.
    .build())
.layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
    .nIn(1000)
    .nOut(outputNum)
    .activation("softmax")
    .weightInit(WeightInit.XAVIER)
    .build())
.pretrain(false).backprop(true)
```

```
.build();
```

而超参数(Hyperparameter)决定了神经网络学习方式的变量，包括模型的权重、激活函数、学习率和优化算法等。

## 18.3 准备数据

再好的训练模型，如果没有数据支撑，得出的结果也不是你想要的。本节将详细介绍常规数据使用方法，以及规范化手段。

### 18.3.1 MNIST 数据集

MNIST (Mixed National Institute of Standards and Technology database) 是一个计算机视觉数据集，包含 60000 张手写数字的灰度图片训练集(training set)，以及 10000 图片测试集(test set)。由来自 250 个不同人手写的数字构成，其中 50% 是高中生，50% 来自人口普查局(the Census Bureau)的工作人员。测试集也是同样比例的手写数字数据。其中每一张图片都包含  $28 \times 28$  个像素点，并配有对应的标签，也即是图片对应的数字。

图 ??，展示了测试集的组织形式：

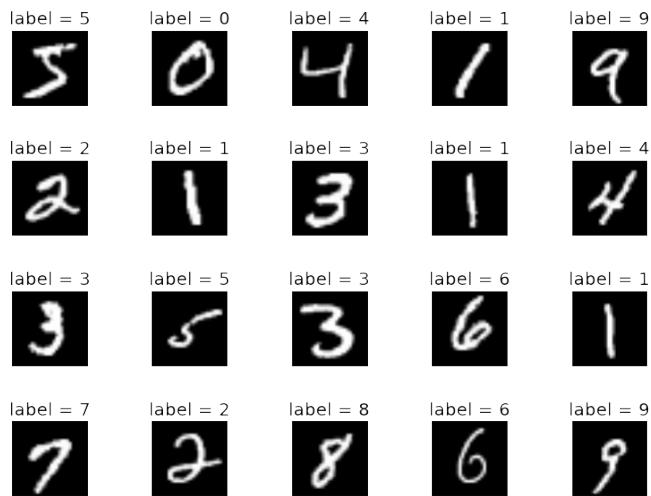


图 18.1: MNIST 数据

MNIST 官网提供了以下 4 个下载链接：

1. train-images-idx3-ubyte.gz: training set images (9912422 bytes)
2. train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
3. t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
4. t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

看到这里，大家最关心的还是下载的 MNIST 数据集，是如何定义的。这可以在官网查到，格式如下：

**训练集标签文件 (train-labels-idx1-ubyte):**

[偏移]	[数据类型]	[值]	[描述]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....			
xxxx	unsigned byte	??	label

标签的值: 0 ~ 9

16进制数据: 训练集-标签集合

```
0000 0801 0000 ea60 0500 0401 0902 0103
0104 0305 0306 0107 0208 0609 0400 0901
0102 0403 0207 0308 0609 0005 0600 0706
....
```

**训练集数据文件 (train-images-idx3-ubyte):**

[偏移]	[数据类型]	[值]	[描述]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	60000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
.....			
xxxx	unsigned byte	??	pixel

16进制数据: 训练集-图片集合

```
0000 0803 0000 ea60 0000 001c 0000 001c
0000 0000 0000 0000 0000 0000 0000 0000
.....
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0312 1212 7e88 af1a
a6ff f77f 0000 0000 0000 0000 0000 0000
1e24 5e9a aaf9 fdf9 fdf9 e1ac fdf2 c340
....
```

DL4J 对 MNIST 数据集支持得很好，提供有 deeplearning4j-datasets-xxxx.jar，其中有很多便利的 API 接口。



### 18.3.2 IRIS 数据集

Iris 也称鸢尾花卉数据集，是常用的分类实验数据集，由 Fisher, 1936 收集整理。包含 150 个数据集，分为 3 类，每类 50 个数据，每个数据包含 4 个属性。可通过花萼长度，花萼宽度，花瓣长度，花瓣宽度 4 个属性预测鸢尾花卉属于 (Setosa, Versicolour, Virginica) 三个种类中的哪一类。



图 18.2: 鸢尾花

该数据集由 3 种不同类型的鸢尾花的 50 个样本数据构成。其中的一个种类与另外两个种类是线性可分离的，后两个种类是非线性可分离的。Iris 以鸢尾花的特征作为数据来源，常用在分类操作中。该数据集包含了 5 个属性：

1. Sepal.Length (花萼长度)，单位是 cm;
2. Sepal.Width (花萼宽度)，单位是 cm;
3. Petal.Length (花瓣长度)，单位是 cm;
4. Petal.Width (花瓣宽度)，单位是 cm;
5. 种类: Iris Setosa (山鸢尾)、Iris Versicolour (杂色鸢尾)，以及 Iris Virginica (维吉尼亚鸢尾)。

### 18.3.3 波斯顿房价数据集

波士顿房价数据集 (Boston House Price Dataset)，该数据集是美国人口普查局收集的有关波士顿马萨诸塞州住房的信息，发表于 1978 年美国某经济学杂志。该数据集包含波士顿房屋的价格及其各项数据，每个数据项有 14 个数据，常用于回归分析。

- CRIM - 城镇人均犯罪率
- ZN - 占地面积超过 25,000 平方英尺的住宅用地比例。
- INDUS - 城镇非零售商用土地的比例。
- CHAS - Charles River 虚拟变量（如果是河道，则为 1；否则为 0）。
- NOX - 一氧化氮浓度（每千万）
- RM - 住宅平均房间数。
- 年龄 - 1940 年之前建成的自用房屋比例。
- DIS - 到波士顿五个中心区域的加权距离。
- RAD - 径向高速公路的可达性指数
- 税 - 每 10,000 美元的全额物业税率
- PTRATIO - 城镇的学生与教师比例

- $B - 1000Bk - 0.63^2$ , 其中  $B_k$  是城镇黑人的比例
- LSTAT - 人口中地位低下者的比例。
- MEDV - 自有住房的中位数价值 1000 美元

波士顿房价数据集，共有 506 个观察，有 13 个输入变量和 1 个输出变量。在回归分析中，如果有两个或两个以上的自变量，就称为多元回归。使用多个自变量的共同来预测，比只用一个自变量进行预测更符合实际，因此多元线性回归比一元线性回归的实用意义更大。实际上，经常抽取波士顿房价数据的 1-2 个特征，做线性回归分析的演示。

## 18.4 配置 DL4J

使用 IntelliJ IDEA 开发环境，建议创建 Maven 工程，根据官网的说明配置 pom.xml 联网下载依赖就可以。如果你比较熟悉 Gradle，官网也有相应的说明。

**Listing 18.1:** pom.xml

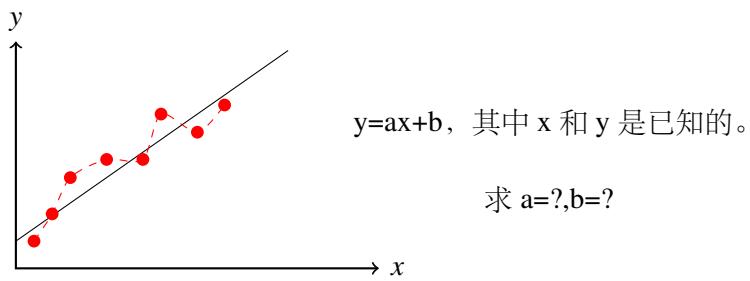
---

```
<dependencies>
  <dependency>
    <groupId>org.deeplearning4j</groupId>
    <artifactId>deeplearning4j-core</artifactId>
    <version>1.0.0-beta2</version>
  </dependency>
  <dependency>
    <groupId>org.nd4j</groupId>
    <artifactId>nd4j-native-platform</artifactId>
    <version>1.0.0-beta2</version>
  </dependency>
</dependencies>
```

---

## 18.5 线性回归

线性回归可分为一元线性回归和多元线性回归。只包括一个自变量和一个因变量的线性回归，结果近似一条直线。



上图，根据已知点如何

预测下一个位置呢？若选用线性模型  $y = a_n * x_n + a_{n-1} * x_{n-1} + \dots + a_0$  可以拟合所有点，

但  $y = a * x + b$  简单的只是一条直线。选用哪个预测模型更好呢？更高维度的线性模型，实际上也就是多元线性回归：

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p + \epsilon \quad (18.1)$$

其中， $\epsilon$  代表一些未知因素引入的随机误差(残差)，譬如丢失一些关键特征。这个误差会导致有些点偏离预测的函数曲线。在机器学习中，线性回归的模型描述通常使用  $\hat{y} = w^T x + b$  的形式。其中， $\hat{y}$  是模型预测  $y$  应该取的值。

已知数据							
1	2	3	4	5	6	7	8
(0.2,0.3)	(0.4,0.6)	(0.6,1.0)	(1.0,1.2)	(1.4,1.2)	(1.6,1.7)	(2.0,1.5)	(2.3,1.8)

对于简单的线性回归问题，使用 *ND4J* 和 *DL4J* 都可以求解。不同的是 *ND4J* 结合最小二乘法直接公式求解，而 *DL4J* 根据已知数据训练的结果。

## 1、创建神经网络

```
OutputLayer outputLayer = new OutputLayer.Builder(LossFunctions.LossFunction.  
MSE)  
    .activation(Activation.IDENTITY)  
    .nIn(1)  
    .nOut(1).build();  
  
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()  
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)  
    .weightInit(WeightInit.ZERO)  
    .updater(new Sgd(0.01))  
    .list()  
    .layer(0, outputLayer)  
    .pretrain(false)  
    .backprop(true).build();  
  
MultiLayerNetwork net = new MultiLayerNetwork(conf);  
net.init();
```

## 2、初始化数据

```
double [] input = {  
    0.2, 0.4, 0.6, 1.0, 1.4, 1.6, 2.0, 2.3  
};  
  
double [] output = {  
    0.3, 0.6, 1.0, 1.2, 1.2, 1.7, 1.5, 1.8  
};
```



```
INDArray inputNDArray = Nd4j.create(input, new int[]{input.length,1});  
INDArray outPut = Nd4j.create(output, new int[]{output.length, 1});
```

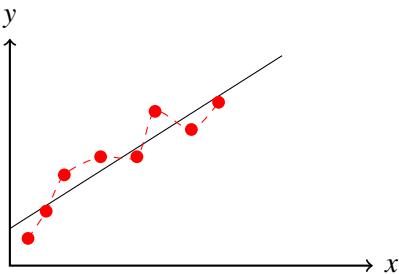
### 3、机器学习过程

```
for( int i=0; i<5000; i++ ) {  
    net.fit(inputNDArray, outPut);  
}
```

```
Map<String, INDArray> params = net.paramTable();  
params.forEach((key, value) -> System.out.println("key:" + key +", value = " +  
    value));
```

```
key:0_W, value = 0.6350  
key:0_b, value = 0.4085
```

因此，拟合出的直线方程： $y = 0.635 * x + 0.4085$



很显然，机器学习预测的图像，比上一页凭我直观的绘制直线更加符合实际。

## 18.6 逻辑回归应用

## 18.7 多分类问题

## 18.8 聚类问题

## 18.9 卷积神经网络

## 18.10 循环卷积神经网络

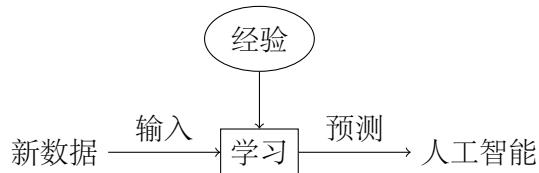
第四部分

综合运用



十年前，手写识别还只是专业技术人员的玩物，而如今手写识别已经变成深度学习的入门教程。而模式识别（Pattern Recognition）意味着需要专业人员提供很多特征。模式识别从 19 世纪 50 年代开始出现，在 20 世纪 80 年代左右曾经风靡一时，也是信息科学和人工智能的重要组成部分。它被应用于图像分析与处理、语音识别、数据挖掘等很多方面。尽管模式识别也很高大上，也有了较长时间的应用，但效果一直差强人意。

这种由人提取特征交给机器，再交给机器去判断的方法识别率一直差强人意。而机器学习可以从海量数据学习特征，使用已知经验数据（样本）自己提炼特征。另外，再强调一下概念，深度学习只是机器学习算法的一种，而人工智能也不限于机器学习一种。



大数据和深度学习，让我们能够摆脱人工特征提取的繁琐和束缚，让更多的人参与人工智能并为社会创造价值。互联网 + 没有过时，再加上分布式和云计算，使得人工智能 + 如鱼得水，再次迎来十年兴盛，相信接下来是数据为王的时代！

# 第 19 章 大数据和分布式

---

## 19.1 云计算

云计算是基于互联网的一种全新的模式。通过“云”我们可以轻松地存储数据，运行应用，甚至进行强度很大的运算。云计算实质上实现了对计算机资源的完全掌控，我们可以很轻易地通过云对资源进行分配，例如分配内存、处理器等计算资源。

机器学习和云计算、大数据的关系十分密切。大数据技术具有大量的数据与资源，能促进机器学习的进一步发展。反过来，机器学习的发展也能促进数据挖掘，分析等方面的进步。

## 19.2 边缘计算

边缘计算的定义是在靠近数据生成的设备端，就近提供服务。相比于云计算需要将所有数据上传至云端的特性，边缘计算具有天然的优势。第一，由于边缘计算强调“边缘”，数据的处理相比于云计算更为实时，高效；第二，边缘计算处理的数据多为“小数据”，在数据存储和计算上成本都比较低；第三，边缘计算的出现降低了对网络带宽的需求。随着物联网的不断发展，越来越多的联网设备将出现在我们的生活中，这大大增加了网络传输的压力。边缘计算则通过数据本地处理解决了这一问题。

边缘计算是对云计算的一种补充与完善。思科预计到 2022 年，移动设备和连接数量将达到 123 亿<sup>1</sup>。如果仅仅用云计算来解决这么多互联设备的数据量往往会造成资源浪费，及时性较差，以及隐私泄露等问题。边缘计算在一方面能减轻云端的负担，另一方面也能保护用户的隐私，具有重要的实际意义。

## 19.3 开源架构

随着云计算和大数据的不断发展，人们对于数据存储和数据处理的效率提出了越来越严格的要求，因此一批优秀的开源架构应运而生。

### 19.3.1 Hadoop

Hadoop 是基于分布式计算的大数据开源架构，它的特点在于能高效，可靠地对大量数据进行处理。Hadoop 由 HDFS 和 MapReduce 组成，分别负责数据的分布式储存和分布式计算。

HDFS 提供了大规模存储的分布式解决方案，可以横跨成千上百台机器，操作却像对本地文件进行操作一样简单，并能存储 TB 甚至 PB 级别的特大文件。这些优势来源于

---

<sup>1</sup> 数据来源思科官网《思科移动网络 VNI 预测（2017-2022 年）》

在 HDFS 中，文件的储存是以较大的抽象块为单位，这样做的好处在于能将文件存储于不同的硬盘中，使得文件的大小不受限于硬盘大小；同时较大的抽象块有利于对大型文件进行寻址操作，最小化寻址开销。

除了抽象块，HDFS 还包含了 NameNode 和 DataNode。用户在用 HDFS 进行存储时，客户端首先会将文件切块，再由 DataNode 储存在多个硬盘中。同时，NameNode 负责记录每一个文件的切块信息，以及每一块具体的存储机器。这样的系统就构成了 NameNode 为主服务器，多个 DataNode 为从服务器的 HDFS 文件系统。

MapReduce 是 Hadoop 的计算框架，能够以容错、可靠的方式并行处理海量的数据。MapReduce 主要包括了两个主要部分：Map 阶段和 Reduce 阶段。

### 19.3.2 Storm

Storm 是类似于 Hadoop 的实时数据处理框架，两者的区别在于 Storm 用于数据的实时计算，Hadoop 用于对离线数据进行处理。同时，Storm 将数据存储在内存中，这也有区别于 Hadoop 的 HDFS 存储方式。Storm 也提供了类似于 MapReduce 的简单编程模型，便于进行开发。

一个完整的 Storm 集群应该包括 Nimbus，Zookeeper 和 Supervisor，工作架构如图19.1所示。其中，Nimbus 负责接受客户端传来的拓扑代码并分拆 task，Zookeeper 负责 Nimbus 和 Supervisor 之间的通信和协调，Supervisor 接收并处理数据。这样高效的工作架构使得 Storm 具有实时处理大批量数据的能力，同时具有高扩展性，可以通过增加节点实现性能的线性提高。

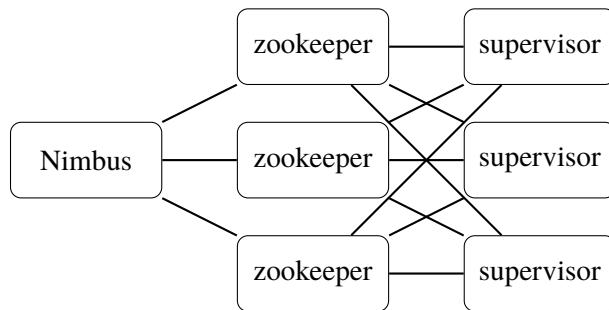


图 19.1: Storm 架构示意图

### 19.3.3 Spark

Spark 是一种通用的大数据计算框架，与 Hadoop 的 MapReduce，Storm 的流式实时计算引擎十分相似。同时 Spark 也提供了大数据和机器学习方面的框架，逐渐形成了大数据处理的一站式解决平台。

Spark 的特点是支持大型，低延迟的数据处理方式。拓展了常用的 MapReduce 计算模型。由于 Spark 支持在内存中进行计算，在必要时依赖硬盘进行复杂运算，因此 Spark 比 MapReduce 更高效。

Spark 是 MapReduce 的一种替代方案，兼容 HDFS，Hive 等存储方案，能够方便地融入 Hadoop 的生态环境，以弥补 MapReduce 的不足。

## 19.4 示例：人脸识别



## 第 20 章 车牌识别

---

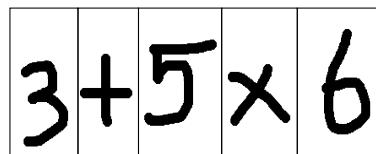
手写识别机器学习领域中的一个经典问题。自从深度学习的兴起，MNIST 已经是一个被“嚼烂”了的数据集。该问题要解决的是把  $28 \times 28$  像素的灰度手写数字图片，识别为 0 到 9 的数字。而手写算式识别在数字识别的基础上，需要识别运算符号，并输出计算结果。

$$3+5 \times 6 \rightarrow \text{识别: } 3 + 5 \times 6$$

图 20.1: 手写算式识别

限于篇幅和笔者能力，本演示代码仅支持四则运算 ( $+ - \times \div$ )，其它运算符号留给同学们研究。利用深度学习算法，为用户提供手写算式识别，可以极大的提高用户体验。目前，有很多成熟的产品可以借鉴，相关的论文也非常多。

手写数字识别属于图像分类，可用的算法有：KNN、SVM、BP 和 CNN。其中 KNN 和 SVM 比较善于数据挖掘，用做手写数字识别的话，比较难达到实用目的；而 BP 使用全连接，难于构建深度网络，并存在梯度消失和过拟合问题。本章例子，选用 CNN 深度学习训练手写算式识别。



现在我们要识别一个算式，其实和 MINIST 数字识别没有太多差别，主要是增加了图像切割和四则运算符的识别任务。为了降低难度，约束字符之间最好有一些间距，不要字符连笔。

### 20.1 图像切割

手写算式的输入的形式，有 2 种：拍照、手绘。照片要经过图像处理之后，才能进行切割；而手绘可直接采集像素轨迹，就省掉了这一步。对于照片的预处理，需要先二值化和降噪把手写数字区域保留下来。如果考虑图片的背景干扰，可先使用 Canny 算子检测边缘，这借助 OpenCV 的 API 很容易实现。

我们这种要求手写数字不黏连的情况，把 Canny 输出的图像往 X 轴投影，就能得出每个数字的横向宽度坐标。虽然要求数字不黏连，但难免有横向重叠的情况，必须用类似范水法进行切割。

这种图像预处理，OpenCV<sup>1</sup>非常擅长，不需要使用机器学习算法。

---

<sup>1</sup>OpenCV 是一个开源的跨平台视觉库，实现了图像处理和计算机视觉方面的很多通用算法，支持 Linux、



图 20.2: 图像处理: 二值化和边缘识别 (OpenCV)

*Java 代码:*

---

```
System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
Mat img = Imgcodecs.imread(src);
Imgproc.cvtColor(img, img, Imgproc.COLOR_BGR2GRAY);
Imgproc.Canny(img, img, threshold, threshold * 3, 3, true);
Imgcodecs.imwrite(dstImg, img);
```

---

*Python 代码:*

---

```
import cv2
from matplotlib import pyplot as plt

img = cv2.imread("handwriting_input.png", 0)
edges = cv2.Canny(img, 30, 60)
plt.subplot(1,2,1), plt.imshow(img, cmap='gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(1,2,2), plt.imshow(edges, cmap='gray')
plt.title('Edge Image'), plt.xticks([]), plt.yticks([])
plt.show()
```

---

实现 Canny 算子不在本书讨论范围, GitHub 上的一份 Java 实现接近 OpenCV 的效果 [Github 上获取](#)。而 CSDN 上的一份实现简化了很多处理细节, 有一定差距。有条件的同学, 建议使用 OpenCV 的接口。遇到的 OpenCV 问题, 也容易在网络上检索。

1. 转换为灰度图像
2. 高斯模糊处理
3. 根据梯度计算图像边缘幅值与角度
4. 非最大信号压制处理 (边缘细化)
5. 双阈值边缘连接处理
6. 二值化图像输出结果



经过 Canny 算子处理之后, 切割图像就非常容易了。把 Canny 输出的二值图像, 在水平方向上投影, 对数字进行竖向分割; 同理, 在纵向再做一次, 就能切割得紧凑一些。以下, 是在原图上切割的效果:

Windows、Android 和 Mac OS 操作系统上。它轻量且高效, 同时提供 C++、Python、MATLAB 等语言的接口。



Minist 数据集是白底黑字的，使用它训练出来的模型，就很难识别我们的手写算式。所以，有必要在二值图像上切割。



但 Canny 生成的是空心字，并且有很多断线，使用 FloodFill 算法是填不满的。笔者采用的是膨胀法，对于每一个白色点向周围扩张 6 个像素，这样的结果还不错，差不多和 Minist 数据集很接近了。



尺寸也要标准化，做成和 Minister 一样的  $28 \times 28$ ，这个对图片缩放就可以实现。

## 20.2 算式识别

## 20.3 结果评估

# 第 21 章 写诗机器人

---

## 21.1 word2vec 介绍

word2vec 是 google 在 2013 年推出的一个 NLP 工具, 它的特点是将所有的词向量化, 这样词与词之间就可以定量的去度量他们之间的关系, 挖掘词之间的联系。

## 21.2 如何写诗

## 第 22 章 常见问题集

---

**问题** 有没有办法章节用“第一章，第一节，(一)”这种？

**解** 你可以修改模板中对于章节的设置，利用 `ctex` 宏集的 `\zhnumber` 命令可以把计数器的数字形式转为中文。

**问题** 3.07 版本的 `cls` 的 `natbib` 加了 `numbers` 编译完了没变化，群主设置了不可更改了？

**解** 3.07 中在 `gbt7714` 宏包使用时，加入了 `authoryear` 选项，这个使得 `natbib` 设置了 `numbers` 也无法生效。3.08 版本中，模板增加了 `numbers` 和 `authoryear` 文献选项，你可以参考前文设置说明。

**问题** 大佬，我想把正文字体改为亮色，背景色改为黑灰色。

**解** 页面颜色可以使用 `\pagecolor` 命令设置，文本命令可以参考[这里](#)进行设置。

**问题** ! LaTeX Error: Unknown option ‘scheme=plain’ for package ‘`ctex`’.

**解** 你用的 CT<sub>E</sub>X 套装吧？这个里面的 `ctex` 宏包已经是 10 年前的了，与本模板使用的 `ctex` 宏集有很大区别。不建议 CT<sub>E</sub>X 套装了，请卸载并安装 Te<sub>X</sub> Live 2019。

**问题** 我该使用什么版本？

**解** 请务必使用[最新正式发行版](#)，发行版间不定期可能会有更新（修复 bug 或者改进之类），如果你在使用过程中没有遇到问题，不需要每次更新[最新版](#)，但是在发行版更新之后，请尽可能使用最新版（发行版）！最新发行版可以在 Github 或者 Te<sub>X</sub> Live 2019 内获取。

**问题** 我该使用什么编辑器？

**解** 你可以使用 Te<sub>X</sub> Live 2019 自带的编辑器 Te<sub>X</sub>works 或者使用 Te<sub>X</sub>studio，Te<sub>X</sub>works 的自动补全，你可以参考我们的总结 [TeXworks 自动补全](#)。推荐使用 Te<sub>X</sub> Live 2019 + Te<sub>X</sub>Studio。我自己用 VS Code 和 Sublime Text，相关的配置说明，请参考 [LaTeX 编译环境配置：Visual Studio Code 配置简介](#) 和 [Sublime Text 搭建 LaTeX 编写环境](#)。

**问题** 您好，我们想用您的 ElegantBook 模板写一本书。关于机器学习的教材，希望获得您的授权，谢谢您的宝贵时间。

**解** 模板的使用修改都是自由的，你们声明模板来源以及模板地址（github 地址）即可，其他未尽事宜按照开源协议 LPPL-1.3c。做好之后，如果方便的话，可以给我们一个链接，我把你们的教材放在 ElegantLaTeX 用户作品集里。

**问题** 我想要原来的封面！

**解** 我们计划在未来版本加入封面选择，让用户可以选择旧版封面。

**问题** 我想修改中文字体！

**解** 首先，我们强烈建议你不要去修改字体！如果你一定坚持修改字体，请在 `newtxttext` 宏包加载前加入中文字体设置（`xeCJK` 宏包）。如果你选择自定义字体，请设置好 `\kaishu`, `\heiti` 等命令，否则会报错。如果你看不懂我现在说的，请停止你的字体自定义行为。

**问题** 请问交叉引用是什么？

**解** 本群和本模板适合有一定 LATEX 基础的用户使用，新手请先学习 LATEX 的基础，理解各种概念，否则你将寸步难行。

**问题** 定义等环境中无法使用加粗命令么？

**解** 是这样的，默认中文并没有加粗命令，如果你想在定义等环境中使用加粗命令，请使用 `\heiti` 等字体命令，而不要使用 `\textbf`。或者，你可以将 `\textbf` 重新定义为 `\heiti`。英文模式不存在这个问题。

**问题** 代码高亮环境能用其他语言吗？

**解** 可以的，ElegantBook 模板用的是 `listings` 宏包，你可以在环境之后加上语言，全局语言修改请使用 `\lstset` 命令，更多信息请参考宏包文档。

**问题** 群主，什么时候出 Beamer 的模板（主题），ElegantSlide 或者 ElegantBeamer？

**解** 这个问题问的人比较多，我这里给个明确的答案。由于 Beamer 中有一个很优秀的主题 `Metropolis`。我觉得在我们找到非常好的创意之前不会发布正式的 Beamer 主题，如果你非常希望得到 ElegantLATEX “官方”的主题，请在用户 QQ 群内下载我们测试主题 PreElegantSlide（未来不一定按照这个制作）。正式版制作计划在 2020 年之后。

**问题** 群主好棒，想嫁！

**解** 我取向正常！



## 附录 基本数学工具

---

本附录包括了计量经济学中用到的一些基本数学，我们扼要论述了求和算子的各种性质，研究了线性和某些非线性方程的性质，并复习了比例和百分数。我们还介绍了一些在应用计量经济学中常见的特殊函数，包括二次函数和自然对数，前 4 节只要求基本的代数技巧，第 5 节则对微分学进行了简要回顾；虽然要理解本书的大部分内容，微积分并非必需，但在一些章末附录和第 3 篇某些高深专题中，我们还是用到了微积分。

### A.1 求和算子与描述统计量

**求和算子**是用以表达多个数求和运算的一个缩略符号，它在统计学和计量经济学分析中扮演着重要作用。如果  $\{x_i : i = 1, 2, \dots, n\}$  表示  $n$  个数的一个序列，那么我们就把这  $n$  个数的和写为：

$$\sum_{i=1}^n x_i \equiv x_1 + x_2 + \dots + x_n \quad (\text{A.1})$$

## 附录 最小示例

---

```
\documentclass[lang=cn,11pt]{elegantbook}
% title info
\title{Title}
\subtitle{Subtitle is here}
% bio info
\author{Your Name}
\institute{XXX University}
\date{\today}
% extra info
\version{1.00}
\extrainfo{Victory won\rq{}t come to us unless we go to it. --- M. Moore}
\logo{logo.png}
\cover{cover.jpg}

\begin{document}

\maketitle
\tableofcontents
\mainmatter
\hypersetup{pageanchor=true}
% add preface chapter here if needed
\chapter{Example Chapter Title}
The content of chapter one.

\bibliography{reference}

\end{document}
```

---