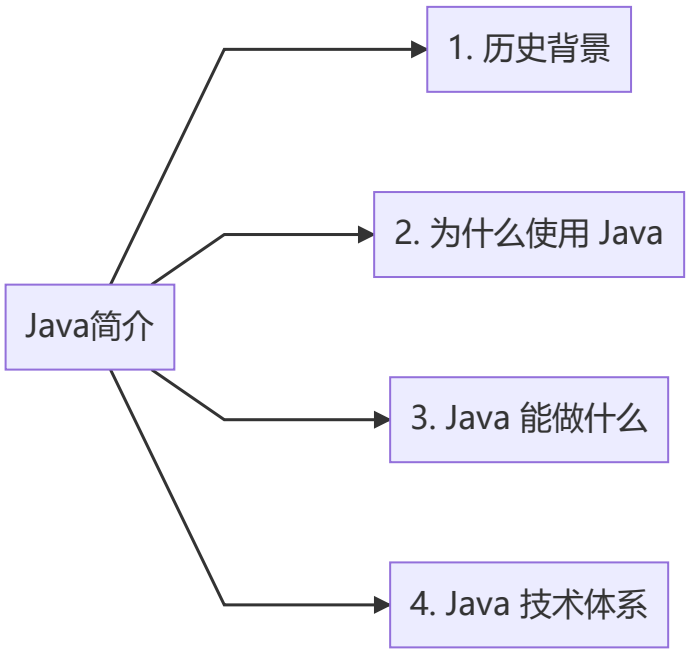
JavaSE

第一章 Java概述

1. Java 入门简介

1. 1 前言

既然要学习一门技术，那么就先来了解下它的历史，我们为什么要使用它，以及我们能用它来干啥。



1.2 历史背景

1990 年代初，`Sun` 公司的詹姆斯·高斯林等人为了实现电视机、电话、闹钟等家用电器的控制和通信，开发了一套用于设置在家用电器等小型系统中的编程语言。在当时，这门语言被命名为 `Oka`。但由于市场需求不高，所以该计划被逐渐放弃。

随着 1990 年代互联网的发展，`Sun` 公司发现 `Oka` 语言在互联网中的应用前景广泛。于是决定对 `Oka` 进行改造，并在 1995 年 5 月以 `Java` 的名义正式发布。

随着互联网的迅猛发展，Java 也得以逐渐成为重要的网络编程语言，詹姆斯·高斯林也被大家公认为 `Java` 之父。

到了 2009 年，`Sun` 公司被 `Oracle` 公司所收购，自此 `Java` 成为 `Oracle` 公司的一大产品直至今日。

以下是截止到本文写作时间（2022 年 7 月 23 日）`Java` 的历史年表，累计经过了 18 次主要版本更新，目前来到了 `Java SE` 18。而 `Java SE` 19 早期预览版也已经流出，根据官方给出的发布时间表，大概会在 9 月份发布 `Java SE` 19 的 `GA` 版本。不过虽然 `Java` 已经经历了这么多版本的更新，但国内目前用的最多的应该还是 Java 1.8 版本。而出于对稳定性的考虑，企业开发更喜欢 `LTS` 版本，这也是为什么至今 `Java` 1.8 还占有如此高的市场比例。

版本	发布日期	主要事件
JDK Beta	1995	`Java` 语言发布，用 `Java` 实现的浏览器和 `Java Applet` 被大量应用
JDK 1.0	1996 年 1 月	奠定了 `JDK`、`JRE`、`JVM` 的体系结构
JDK 1.1	1997 年 2 月	加入 `JIT`，提升 `JDK` 效率
J2SE 1.2	1998 年 12 月	确立 `J2SE`、`J2EE`、`J2ME` 产品结构
J2SE 1.3	2000 年 5 月	内置 HotSpot JVM
J2SE 1.4	2002 年 2 月	XML 处理、断言、支持正则表达式
J2SE 5.0	2004 年 9 月	静态导入、泛型、for-each 循环、自动拆，装箱、枚举、可变参数
Java SE 6	2006 年 12 月	提供动态语言支持、同步垃圾回收
Java SE 7	2011 年 7 月	字符串的 switch 语句、多异常捕抓
Java SE 8 (LTS)	2014 年 3 月	Lambda 表达式
Java SE 9	2017 年 9 月	轻量级 json API、垃圾收集机制更新
Java SE 10	2018 年 3 月	局部变量类型推断，Java 后续版本快速迭代更新
Java SE 11 (LTS)	2018 年 9 月	常用类增强，Java 11 是继 Java 8 之后的首个长期支持本版
Java SE 12	2019 年 3 月	switch 表达式增强
Java SE 13	2019 年 9 月	文本块支持
Java SE 14	2020 年 3 月	`instanceof` 支持模式匹配
Java SE 15	2020 年 9 月	`EdDSA` 数字签名算法、 密封类、隐藏类
Java SE 16	2021 年 3 月	启用 C++ 14 语言特性、Vector API
Java SE 17 (LTS)	2021 年 9 月	恢复总是严格的浮点语义、增强型伪随机数生成器

版本	发布日期	主要事件
Java SE 18	2022 年 3 月	默认 UTF-8 编码、代码片段、简单的网络服务器

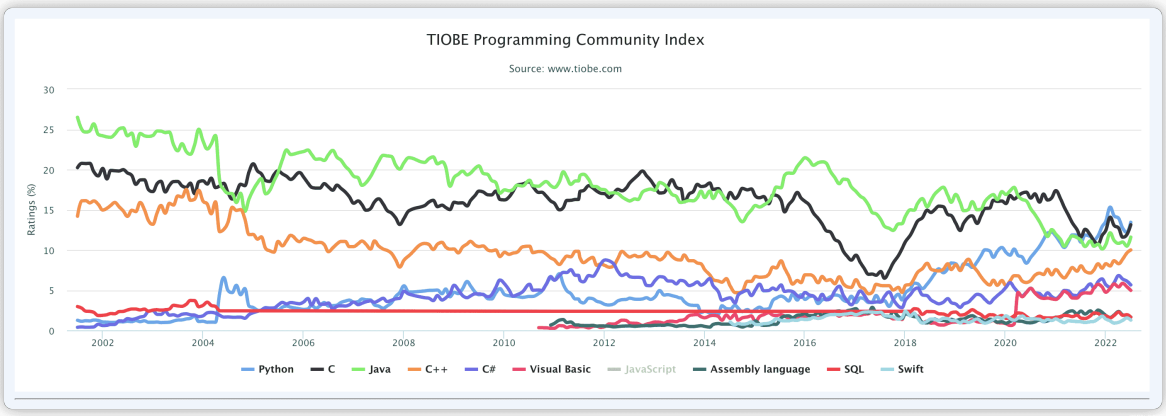
1.3 为什么使用 Java

既然编程语言有那么多，那我们为什么要使用 Java 语言呢？以下就来看看 Java 被广泛使用的几个原因。

1. **世界范围内流行，国内使用最为广泛的编程语言之一。**

以下是截止 2022 年 7 月 TIOBE 统计的编程语言排行榜，可以看到 Java 处于前三的位置。而从历年编程语言所占市场份额趋势图也可以看出，Java 虽然没有一直独占鳌头，但绝大数时间都是占据领先地位。

Jul 2022	Jul 2021	Change	Programming Language	Ratings	Change
1	3	↑	Python	13.44%	+2.48%
2	1	↓	C	13.13%	+1.50%
3	2	↓	Java	11.59%	+0.40%
4	4		C++	10.00%	+1.98%
5	**5**		**C#**	**5.65%**	**+0.82%**
6	6		Visual Basic	4.97%	+0.47%
7	7		JavaScript	1.78%	-0.93%
8	9	↑	Assembly language	1.65%	-0.76%
9	10	↑	SQL	1.64%	+0.11%
10	16	⇈	Swift	1.27%	+0.20%
11	8	↓	PHP	1.20%	-1.38%
12	13	↑	Go	1.14%	-0.03%
13	11	↓	Classic Visual Basic	1.07%	-0.32%
14	20	⇈	Delphi/Object Pascal	1.06%	+0.21%
15	17	↑	Ruby	0.99%	+0.04%



2. **移植性高**

由于 Java 中 JVM 的特性，所以能够做到一次编译，随处运行，其移植性也就更高。

3. **开发社区完善**

社区完善也是 Java 为什么使用多的原因，各种封装 API，比起 C、C++ 语言使用起来更加方便。虽然 `Python` 的封装程度更高，但是其效率比起 `Java` 却打了折扣。毕竟封装程度和效率是成反比的，`Java` 兼具了效率高和使用方便的特点，所以更受人们青睐。

1.4 Java 能做什么

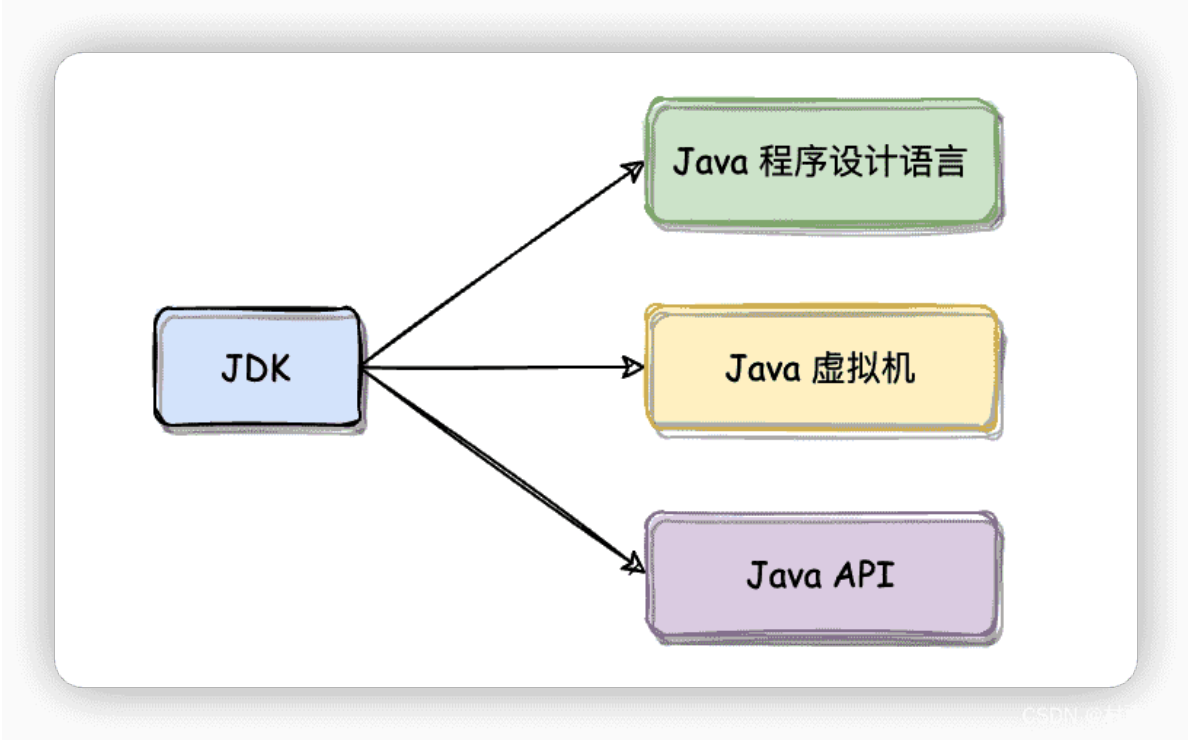
Java 应用广泛，在我们日常中就能经常见到由 Java 所开发的程序。常见的的应用应用领域如下：

1. **桌面 GUI 应用开发**
2. **服务器系统**
3. **企业级应用开发**
4. **大数据开发**
5. **移动应用开发**
6. **游戏开发**
7. ……

1.5 Java 技术体系

根据传统意义上 Sun 官方定义， Java 技术体系的组成主要就是我们常说的 JDK，即以下三部分：

1. **Java 程序设计语言**
2. **Java 虚拟机**
3. **Java API 类库**



同时，Java 技术体系又可以主要分为以下 3 个平台。

1. **Java SE**

Java Standard Edition，即 Java 标准版。主要是 Java 技术的核心和基础，要想学好 Java，那么首先你得掌握好这部分。主要运行在桌面级应用，如 Windows 应用程序。

2. **Java EE**

Java Enterprise Editon，即 Java 企业版。从它的名字就不难看出，这是 Java 针对企业级应用开发所提供的一套解决方案，主要用于架设高性能企业网站。

3. **Java ME**

Java Micro Edition，即 Java 小型版。主要是 Java 针对移动设备应用所提供的一套解决方案，主要运行在手机、pad 等移动端。

第二章 变量与数据类型

1. 变量

1.1 什么是变量?

所谓变量，就是用来命名一个数据的标识符，其定义格式如下：

```
数据类型 变量名称 = 初始值;
```

其中数据类型是用于限制存储数据的形式，后面会讲到 Java 中的常见数据类型；变量名称是用于代表变量的一个符号，就好比我们每个人的名字；初始值则代表该变量存储时的初始数据。

在 Java 中，变量主要分为两种：

- **基本类型的变量**
- **引用类型的变量**

```
// 基本类型的变量
int id = 1;
// 引用类型的变量
String name = "村雨遥";
```

其中 `int` 是基本数据类型，表示这是一个整型数；而 `String` 则是引用类型，表示这是一个引用类型；

`id` 和 `name` 则是标识符，也就是我们所说的 **变量**；

`=` 则是赋值操作符，而 `1` 则是基本类型的值，`村雨遥` 则是引用类型的值；

1.2 变量的特点

在使用变量时，需要注意以下的几个问题。

1. 变量一定要先声明然后再使用。
2. 声明一个变量的类型后，不能用它来存储其类型的数据。
3. 变量定义时可以不赋初始值，但是在使用时必须赋值。
4. 变量是有使用范围的，在同一使用范围内，不能重复定义同一个变量。
5. 变量最重要的一个特点就是可以重新赋值。

```
public class Main {
      public static void main(String[] args) {
        // 定义int类型变量id，并赋予初始值1
            int id = 1;
        // 打印该变量的值，观察是否为1
            System.out.println(id);
        // 重新赋值为2
            id = 2;
            // 打印该变量的值，观察是否为2
            System.out.println(id);
      }
}
```

1.3 变量命名规则

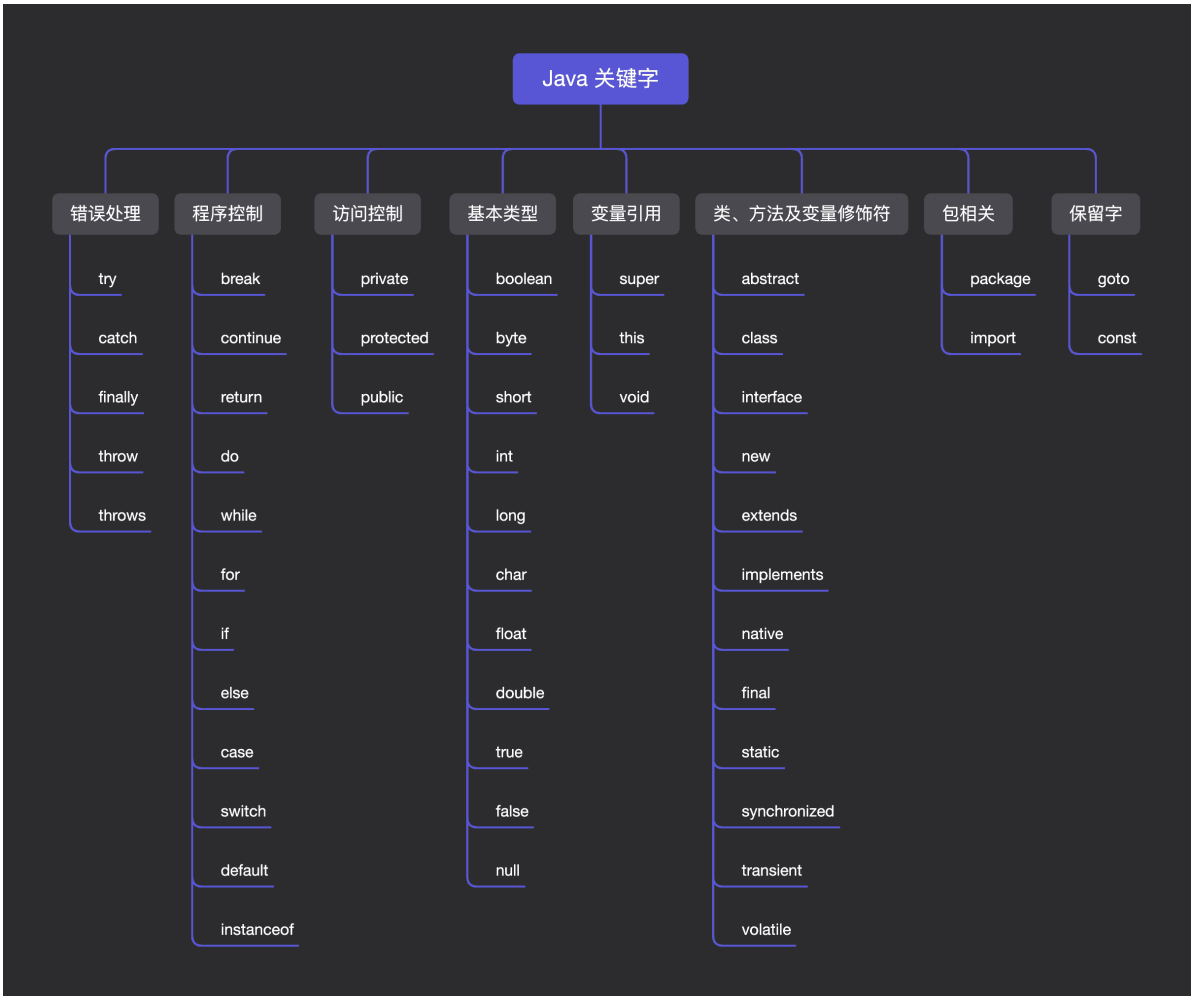
变量命名也是一门学问，并不是我们想怎么命名就怎么命名，日常开发中最常见的变量命名规则主要有如下几条：

1. **强制**：变量命名只能使用 **字母（大小写均可）、数字、$、_**；
2. **强制**：变量名不能使用关键字（就是 Java 中内置的一些关键字，如 `int、for、long…`）；
3. **强制**：变量第一个字符不能使用数字，只能用字母、`$`、`_`；
4. 更多命名规则推荐参考阿里巴巴推出的《Java 开发手册》，下载链接：https://github.com/cunyu1943/ebooks

1.4 常见关键字

这是一种事先定义好的，有特定意义的标识符，也叫做保留字。对于 Java 编译器有着特殊意义，用来表示一种数据类型，或者表示程序的结构等。此外，关键字不能用作变量名、方法名、类名、包名和参数名。常见的关键字可以分为如下几类，具体的关键字如下图所示：

- **访问控制类**
- **类、方法及变量修饰符类**
- **程序控制类**
- **错误处理**
- **包相关**
- **基本类型**
- **变量引用**
- **保留字**



2. 数据类型

2.1 基本数据类型

Java 中，共有 8 种基本数据类型，由 Java 语言预定好的，每个数据类型都属于关键字，而且每种基本变量都有其对应的封装类，这 8 种基本数据类型分别是：

- **整型（4 种）**
- **浮点型（2 种）**
- **字符型（1 种）**
- **布尔型（1 种）**

下面的表就是 Java 中 8 大数据类型所占的内存空间，对应封装类，数据表示范围以及默认值的一下相关情况。

数据类型	bit	字节	封装类	数据范围	默认值
`byte`	8	1	`Byte`	-2^7 ~ 2^7-1	`0`
`short`	16	2	`Short`	-2^{15} ~ $2^{15}-1$	`0`
`char`	16	2	`Character`	`\u0000` ~ `\uffff` (0 ~ 65535)	`u0000`
`int`	32	4	`Integer`	-2^{31} ~ $2^{31}-1$	`0`
`long`	64	8	`Long`	-2^{63} ~ $2^{63}-1$	`0L`
`float`	32	4	`Float`	$1.4e^{-45}$ ~ $3.4e^{38}$	`0.0f`
`double`	64	8	`Double`	$4.9e^{-324}$ ~ $1.8e^{308}$	`0.0D`
`boolean`	1	不确定	`Boolean`	`true` 或 `false`	`false`

注意：

1. `boolean` 一般用 1 `bit` 来存储，但是具体大小并未规定，JVM 在编译期将 `boolean` 类型转换为 `int`，此时 1 代表 `true`，`0` 代表 `false`。此外，JVM 还指出 `boolean` 数组，但底层是通过 `byte` 数组来实现。
2. 使用 `long` 类型时，需要在后边加上 `L/l`，否则将其作为整型解析，可能会导致越界。
3. 浮点数如果没有明确指定 `float` 还是 `double`，统一按 `double` 处理。
4. `char` 是用 **单引号** `''` 将内容括起来，**只能存放一个字符**，相当于一个整型值（ASCII 值），能够参加表达式运算；而 `String` 是用 **双引号** `""` 将内容括起来，代表的是一个地址值。
5. `Java` 在内存中是采用 `Unicode` 表示，所以无论是一个中文字符还是英文字符，都能用 `char` 来表示。

那么如何个一个基本类型变量赋值呢？

在 `Java` 中，基本数据类型属于 `Java` 的一种内置的特殊数据类型，不属于任何类，所以可以直接对其进行赋值；给基本类型的变量赋值的方式就叫做 **字面值**；

```
1    float score = 89.0f;
2    int age = 26;
```

2.2 引用数据类型

2.2.1 常见引用数据类型

数据类型	默认值
数组	`null`
类	`null`
接口	`null`

而对于引用数据类型，我们经常是需要 `new` 关键字来进行赋值，但是引用类型中的 **接口是不能被实例化的，我们需要对其进行实现**；

```
// 初始化一个对象
Pet dog = new Pet();
// 初始化一个数组
int[] arr = new int[10];
```

2.2.2 String

对于引用数据类型中的 `String`，我们需要特别关注。

`String` 不同于 `char`，它属于引用类型，而 `char` 属于基本数据类型。用双引号 `""` 括起来表示字符串，一个字符串能够保存 0 个到任意个字符，它一旦创建就不能被改变。

而针对字符串，如果我们要打印一些特殊的字符，比如字符串本来就包含 `"`，那么这个时候就需要借助于转义字符 `\`，最常见的转义字符主要有：

转义字符	含义
`\"`	字符 `"`
`\'`	字符 `'`
`\\`	字符 `\`
`\n`	换行符
`\t`	制表符 `Tab`
`\r`	回车符

> 那多个字符串之间或者字符串和其他类型数据之间，该如何进行连接呢？

Java 编译器中，对于字符串和其他数据类型之间，可以使用 `+` 进行连接，编译器会自动将其他数据类型自动转换为字符串，然后再进行连接；

> `String` 既然是不可变，那有什么优点呢？

1. **用于缓存 `hash` 值**

由于 `String` 的 `hash` 值被频繁使用，它的不可变性使得 `hash` 值也不可变，此时只需要进行一次计算；

2. **字符串常量池（String Pool）的需要**

如果一个 `String` 对象已经被创建过，那么就会优先从字符串常量池中获取其引用，其不可变性确保了不同引用指向同一 `String` 对象；

3. **安全性**

我们经常用 `String` 作为我们方法的参数，其不变性能够保证参数不可变；

4. **线程安全**

`String` 的不可变性让它天生 **具备线程安全**，能够在多个线程中方便使用而不用考虑线程安全问题。

> `String、StringBuilder、StringBuffer` 对比，该如何选择?

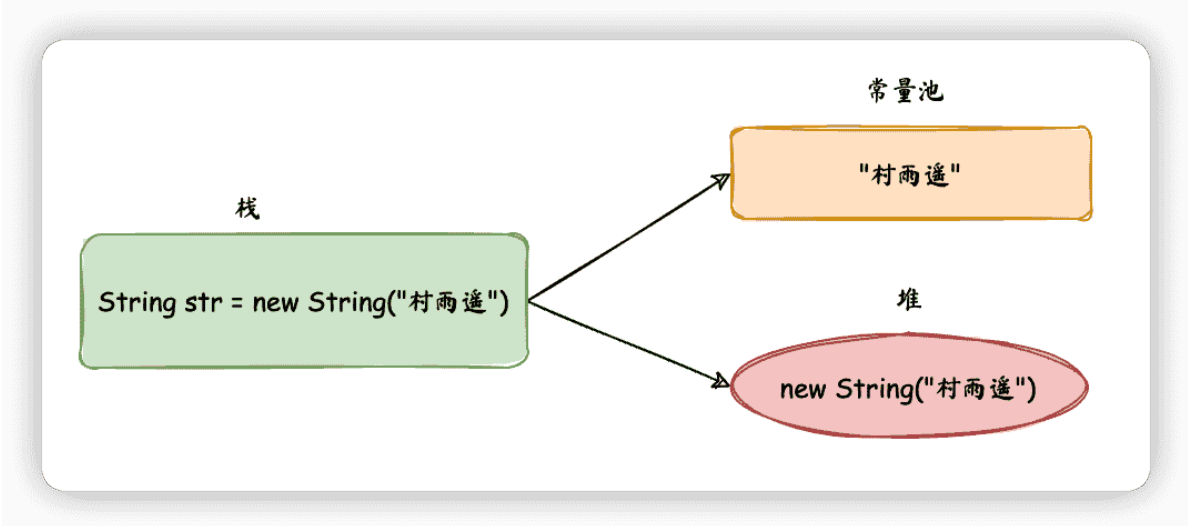
	可变性	线程安全	适用场景
`String`	不可变	安全	操作少量的数据
`StringBuffer`	可变	安全，内部使用 `synchronized` 进行同步	多线程操作字符串缓冲区下操作大量数据
`StringBuilder`	可变	不安全	单线程操作字符串缓冲区下操作大量数据，性能高于 `StringBuffer`

> 通过 `new String("xxx")` 创建字符串的两种情况?

使用 `new` 的方式创建字符串对象，会有两种不同的情况：

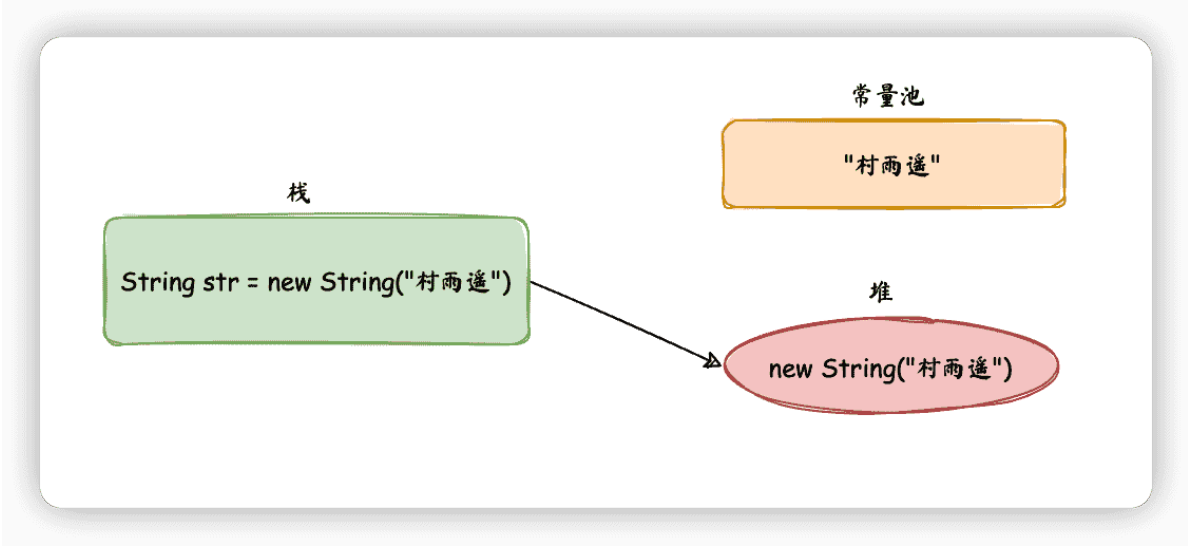
1. **String Pool 中不存在 "xxx"**

此时会创建两个字符串对象，"xxx" 属于字符串字面量，因此在编译期会在 String Pool 中创建一个字符串对象，用于指向该字符串的字面量 "xxx"；然后 `new` 会在堆中创建一个字符串对象。



2. **String Pool 中存在 "xxx"**

此时只需要创建一个字符串对象，由于 String Pool 中已经存在指向 "xxx" 的对象，所以直接在堆中创建一个字符串对象。



2.3 数据类型转换

对于基本数据类型，不同类型之间是可以相互转换的，但是需要满足一定的条件；

> **从小到大自动转，从大到小强制转。**

即就是，对于低精度的数据类型，如果要转换为高精度的数据类型，直接将低精度的值赋给高精度的值即可；

但对于高精度的数据类型，如果想要转换为低精度的数据类型，则需要采用 **强制转换** 的手段，但此时需要承担精度丢失的风险，就像从一个大杯子往一个小杯子里倒水，你要做好小杯子可能装不下溢出的情况；

```
int a = 110;
long b = 113;
// 低精度转高精度，由于 long 的范围比 int 大，所以可以自动转
b = a;
// 高精度住低精度，由于 long 的范围比 int 大，所以需要强制转
a = (int)b;
```

2.3.1 隐式转换（自动类型转换）

当满足如下条件时，如果将一种类型的数据赋值给另一种数据类型变量时，将执行自动类型转换：

> 1. 两种数据类型彼此兼容；
> 2. 目标数据类型的取值范围大于源数据类型；

一般而言，隐式转换的规则是从低级类型数据转换为高级类型数据，对应规则如下：

> - **数值类型**： `byte -> short -> int -> long -> float -> double`
> - **字符类型转整型**： `char -> int`

2.3.2 显式转换（强制类型转换）

那既然满足上述两个条件时会发生隐式转换，那不满足我们又想进行数据类型转换时，我们该怎么办呢？

这个时候就需要我们的 **显式转换** 登场了，其语法格式如下：

```
(type) variableName;
```

我们举个例子来说下：

```
int num = 3;
double ans = 5.0;
// 要将 double 类型的值赋值给 int，则需要强制转换
num = (int)ans;
```

注意：强制转换可能会导致精度丢失，所以一般情况下尽量能不用就不用。

2.3.3 常见数据类型转换方法

1. **字符串与其他类型之间的转换**

- **其他类型 -> 字符串**

> 1. 调用类的串转换方法：`X.toString()`；
> 2. 自动转换：`"" + X`；
> 3. 利用 `String` 的方法：`String.valueOf(X)`；

```
// 方法 1
String str1 = Integer.toString(int num);
String str2 = Long.toString(long num);
String str3 = Float.toString(flaot num);
String str4 = Double.toString(double num);

// 方法 2
String str = "" + num ; // num 是 int、long、float、double 类型

// 方法 3
String str1 = String.valueOf(int num);
String str2 = String.valueOf(long num);
String str3 = String.valueOf(float num);
String str4 = String.valueOf(double num);
```

- **字符串 - > 其他类型**

1. 调用 `parseXXX` 方法，比如 `parseLong、parseFloat、parseDouble...`；
2. 先调用 `valueOf()`，方法，然后再调用 `xxxValue()` 方法；

```
// 方法 1
int num1 = Integer.parseInt(String str);
Long num2 = Long.parseLong(String str);
Float num3 = Float.parseFloat(String str);
Double num4 = Double.parseDouble(String str);

// 方法 2
int num1 = Integer.valueOf(String str).intValue();
Long num2 = Long.valueOf(String str).longValue();
Float num1 = Float.valueOf(String str).floatValue();
Double num1 = Double.valueOf(String str).doubleValue();
```

2. **int、float、double 之间的转换**

- `float -> double`

```
float num = 1.0f;
Float num1 = new Float(num);
double num2 = num1.doubleValue();
```

- `double -> float`

```
double num = 100.0;
float num1 = (float)num;
```

- `double -> int`

```
double num = 100.0;
Double num1 = new Double(num);
int num2 = num1.intValue();
```

- `int -> double`

```
int num = 200;
double num1 = num;
```

3. 常量

3.1 简介

既然有变量，那就有与之相对的常量（也就是值是固定的，不能再变）。

常量又叫做字面常量，是通过数据直接来表示的，在程序运行过程中不能发生改变。通常我们把 Java 中用 `final` 关键字所修饰的成员变量叫做常量，它的值一旦给定就无法再进行改变！

3.2 分类

Java 中使用 `final` 关键字来声明常量，其语法格式如下：

```
final 数据类型 常量名 = 常量初始值;
```

```
public class Main{
      public static void main(String[] args){
            // 声明一个常量并赋值
            final int num = 1024;

            // 再次赋值，将导致编译错误
            num = 1943;

            // 声明一个常量但不赋值
            final int id;
            // 因为声明时未赋值，所以可以进程初次赋值
            id = 1;
            // 常量已经赋值过了，再次赋值将导致编译错误
            id = 2;
      }
}
```

常量可以分为如下 3 种类型：

- **静态常量**：`final` 之前用 `public staic` 修饰，表示该常量的作用域是全局的，我们不用创建对象就能够访问它。
- **成员常量**：类似于成员变量，但是最大的不同在于它不能被修改。
- **局部常量**：作用类似于局部变量，不同之处也在于不能修改。

```java
public class Main{
      // 静态变量
      public static final dobule PI = 3.14;

      // 成员常量
      final int num = 1024;

      public static void main(String[] args){
            // 局部变量
            final long count = 1000;
      }
}
```

PS：`final` 修饰变量后，该变量则变为常量。而 `final` 也还可以用来修饰类和方法，修饰方法时，表示这个方法不能被重写（但可以重载）；修饰类时，则表明该类无法被继承。这些东西这时候你可能会觉得很陌生，不过等我们后续学习了面向对象之后，你就会发现其实很简单。

第三章 运算符和进制转换

1. 运算符

1.1 算术运算符

常见的算术运算符主要有下表中的几种，主要是针对整型和浮点类型的运算。

操作符	描述
`+`	加法 - 相加运算符两侧的值
`-`	减法 - 左操作数减去右操作数
`*`	乘法 - 相乘操作符两侧的值
`/`	除法 - 左操作数除以右操作数
`%`	取余 - 左操作数除以右操作数的余数
`++`	自增: 操作数的值增加1
`--`	自减: 操作数的值减少1

注意：`++` 和 `--` 可以放在操作数之前，也可以放在操作数之后；**位于操作数之前时，先自增/减，再赋值；位于操作数之后，先赋值，再自增/减**；总结起来就是**符号在前就先加/减，符号在后就后加/减。**

以下是部分算术运算符的示例以及运行结果：

```java
/**
 * @author : cunyu1943
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/15 9:43
 * @description : 算术运算符演示
 */

```

```java
public class Main {
    public static void main(String[] args) {
        int num1 = 10;
        int num2 = 20;
        int num3 = 30;
        int num4 = 40;

        System.out.println("num1 + num2 = " + (num1 + num2));
        System.out.println("num1 - num2 = " + (num1 - num2));
        System.out.println("num1 * num2 = " + (num1 * num2));
        System.out.println("num2 / num1 = " + (num2 / num1));
        System.out.println("num2 % num1 = " + (num2 % num1));
        System.out.println("num3 % num1 = " + (num3 % num1));
        System.out.println("num1++    = " + (num1++));
        System.out.println("num1--    = " + (num1--));
        // 查看 ++ 在操作数前后位置时结果的不同
        System.out.println("num4++    = " + (num4++));
        System.out.println("++num4    = " + (++num4));
    }
}
```

```
num1 + num2 = 30
num1 - num2 = -10
num1 * num2 = 200
num2 / num1 = 2
num2 % num1 = 0
num3 % num1 = 0
num1++    = 10
num1--    = 11
num4++    = 40
++num4    = 42

Process finished with exit code 0
```

这里不难看出，无论是 `++` 还是 `--`，当它们单独写一行时，不管是放在变量前边还是后边，其最终结果都是一样的。但如果将它们参与运算，此时的效果就不一样了，这里需要注意。

```java
int a = 10;
int b = a++;
```

以上代码中，先进行了 `b = a` 的赋值操作，所以此时 `b` 的值是 `10`。

```java
int a = 10;
int b = ++a;
```

而此时，先要对 `a` 进行加一的操作之后，再将 `a` 的值赋予 `b`，所以此时 `b` 的值为 `11`。

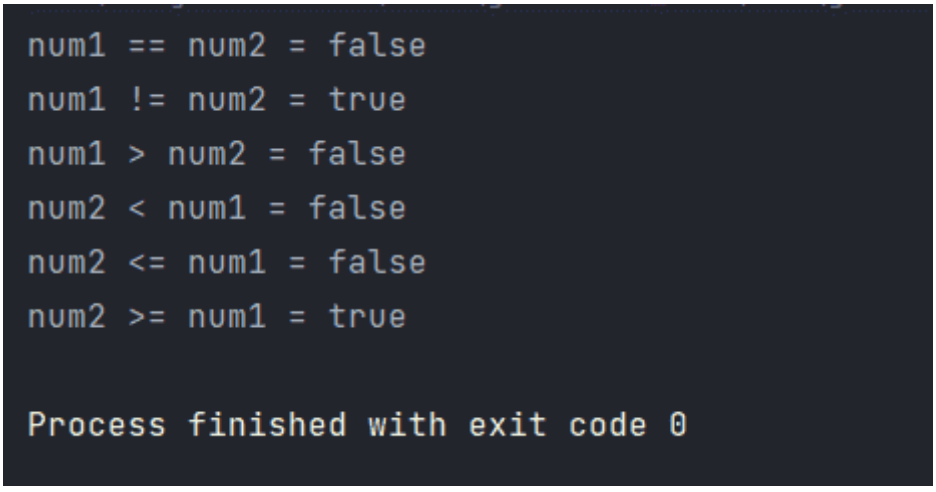
1.2 关系运算符

关系运算符主要是指两个数据间的关系，两者之间的比较结果用逻辑值来表示，常用来比较判断两个变量或常量的大小。常见的关系运算符及含义如下表：

运算符	描述
==	检查如果两个操作数的值是否相等，如果相等则条件为真
!=	检查如果两个操作数的值是否相等，如果值不相等则条件为真
>	检查左操作数的值是否大于右操作数的值，如果是那么条件为真
<	检查左操作数的值是否小于右操作数的值，如果是那么条件为真
>=	检查左操作数的值是否大于或等于右操作数的值，如果是那么条件为真
<=	检查左操作数的值是否小于或等于右操作数的值，如果是那么条件为真

以下是部分关系运算符的示例以及运行结果：

```
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/15 9:45
 * @description : 关系运算符
 */

public class Main {
      public static void main(String[] args) {
            int num1 = 100;
            int num2 = 220;


            System.out.println("num1 == num2 = " + (num1 == num2));
            System.out.println("num1 != num2 = " + (num1 != num2));
            System.out.println("num1 > num2 = " + (num1 > num2));
            System.out.println("num2 < num1 = " + (num2 < num1));
            System.out.println("num2 <= num1 = " + (num2 <= num1));
            System.out.println("num2 >= num1 = " + (num2 >= num1));

      }
}
```



1.3 位运算符

位运算符主要用来对操作数二进制的位进行运算，其运算结果是整型的。常见的位运算符及功能描述如下表所示：

操作符	描述
`&`	如果相对应位都是 1，则结果为 1，否则为 0
`|`	如果相对应位都是 0，则结果为 0，否则为 1
`^`	如果相对应位值相同，则结果为 0，否则为 1
`~`	按位取反运算符翻转操作数的每一位，即 0 变成 1，1 变成 0
`<<`	按位左移运算符。左操作数按位左移右操作数指定的位数
`>>`	按位右移运算符。左操作数按位右移右操作数指定的位数
`>>>`	按位右移补零操作符。左操作数的值按右操作数指定的位数右移，移动得到的空位以零填充

以下是部分位运算符的示例以及运行结果：

```
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/15 10:02
 * @description : 位运算符
 */

public class Main {
      public static void main(String[] args) {
            int num1 = 10;
            int num2 = 20;

            System.out.println("num1 & num2 = " + (num1 & num2));
            System.out.println("num1 | num2 = " + (num1 | num2));
            System.out.println("num1 ^ num2 = " + (num1 ^ num2));
            System.out.println("~ num2 = " + (~num2));
            System.out.println("num1 << 2 = " + (num1 << 2));
            System.out.println("num1 >> 2 = " + (num1 >> 2));
            System.out.println("num1 >>> 2 = " + (num1 >>> 2));
      }
}
```

```
num1 & num2 = 0
num1 | num2 = 30
num1 ^ num2 = 30
~ num2 = -21
num1 << 2 = 40
num1 >> 2 = 2
num1 >>> 2 = 2

Process finished with exit code 0
```

逻辑运算符

逻辑运算符通过将关系表达式连接起来，从而组成一个复杂的逻辑表达式，从而判断程序中的表达式是否成立，其结果返回 `true` 或 `false` 。

操作符	描述
`&&`	称为逻辑与运算符。当且仅当两个操作数都为真，条件才为真
`||`	称为逻辑或操作符。如果任何两个操作数任何一个为真，条件为真
`!`	称为逻辑非运算符。用来反转操作数的逻辑状态。如果条件为 `true` ，则逻辑非运算符将得到 `false`

以下是逻辑运算符的示例以及运行结果：

```java
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/15 10:07
 * @description : 逻辑运算符
 */

public class Main {
      public static void main(String[] args) {
            boolean positive = true;
            boolean negative = false;

            System.out.println("positive && negative = " + (positive && negative));
            System.out.println("positive || negative = " + (positive || negative));
            System.out.println("!(positive || negative) = " + !(positive || negative));

      }
}

```

```
positive && negative = false
positive || negative = true
!(positive || negative) = false

Process finished with exit code 0
```

1.4 赋值运算符

赋值运算符表示一个动作，将其右侧的值送到左侧的变量中。常见的赋值运算符及含义如下表所示：

操作符	描述
=	简单的赋值运算符，将右操作数的值赋给左侧操作数
+=	加和赋值操作符，它把左操作数和右操作数相加赋值给左操作数
-=	减和赋值操作符，它把左操作数和右操作数相减赋值给左操作数
*=	乘和赋值操作符，它把左操作数和右操作数相乘赋值给左操作数
/=	除和赋值操作符，它把左操作数和右操作数相除赋值给左操作数
%=	取模和赋值操作符，它把左操作数和右操作数取模后赋值给左操作数
<< =	左移位赋值运算符
>>=	右移位赋值运算符
&=	按位与赋值运算符
^=	按位异或赋值操作符
|=	按位或赋值操作符

以下是部分赋值运算符的示例以及运行结果：

```java
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/15 10:13
 * @description : 赋值运算符
 */

public class Main {
      public static void main(String[] args) {
            int num1 = 100;
            int num2 = 1000;

            System.out.println("num1 + num2 = " + (num1 + num2));
            System.out.println("num1 += num2 = " + (num1 += num2));
            System.out.println("num1 - num2 = " + (num1 - num2));
            System.out.println("num1 -= num2 = " + (num1 -= num2));
            System.out.println("num1 * num2 = " + (num1 * num2));
            System.out.println("num1 *= num2 = " + (num1 *= num2));
            System.out.println("num1 & num2 = " + (num1 & num2));
```

```
21              System.out.println("num1 &= num2 = " + (num1 &= num2));
22          }
23      }
```

```
num1 + num2 = 1100
num1 += num2 = 1100
num1 - num2 = 100
num1 -= num2 = 100
num1 * num2 = 100000
num1 *= num2 = 100000
num1 & num2 = 672
num1 &= num2 = 672

Process finished with exit code 0
```

1.5 条件运算符（?:）

也叫作三元运算符，共有 3 个操作数，且需要判断布尔表达式的值，常用来取代某个 `if-else` 语句。其语法结构如下所示：

```
关系表达式?表达式 1:表达式 2;
```

```
variable x = (expression) ? value if true : value if false
```

以下是条件运算符的示例以及运行结果：

```
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/15 10:18
 * @description : 条件运算符
 */

public class Main {
      public static void main(String[] args) {
            int num1 = 30;
            int num2 = 300;

            int result = num1 > num2 ? num1 : num2;
            System.out.println("The max between num1 and num2 is " + result);
      }
}
```

```
The max between num1 and num2 is 300

Process finished with exit code 0
```

1.6 instanceof

用于操作对象实例，检查该对象是否是一个特定类型（类类型或接口类型），其语法结构如下。该知识点涉及到类与对象，此处就不做展开，等到后边学习类与对象之后，就能理解了。

```
( Object reference variable ) instanceof  (class/interface type)
```

1.7 运算符优先级

运算符有很多，如果我们将多个运算符组合在一起，那它们又该先计算哪一个，然后再计算哪一个呢？其实就像我们日常排队一样，运算符也是有优先级的。
常用运算符的优先级如下表所示，在我们使用时，如果涉及到一个表达式中含有多个运算符，一定要注意运算符的优先级。当然了，让我们背下来这些也是挺难的。不过不要紧，我们只需要学会灵活运用 `()` 就好了。我们可以利用 `()` 将需要先计算的表达式括起来，然后再去进行下一步的运算。

优先级	运算符
1	`.` 、 `()` 、 `{}`
2	`!` 、 `~` 、 `++` 、 `--`
3	`*` 、 `/` 、 `%`
4	`+` 、 `-`
5	`<<` 、 `>>` 、 `>>>`
6	`<` 、 `<=` 、 `>` 、 `>=` 、 `instanceof`
7	`==` 、 `!=`
8	`&`
9	`^`
10	`|`
11	`&&`
12	`||`
13	`?:`
14	`=` 、 `+=` 、 `-=` 、 `*=` 、 `/=` 、 `%=` 、 `&=`

1.8 equals() 和 ==

- `==`

基本数据类型用 `==` 比较的是值，而 用于引用数据类型时判断两个对象的内存地址是否相等，即两对象是否是同一个对象；

> 本质来讲，由于 Java 中只有值传递，所以不管是基本数据类型还是引用数据类型，比较的其实都是值，只不过引用类型变量存的值是对象的地址；

- `equals()`

作用也是判断两个对象是否相等，但是 **不能用于基本数据类型变量的比较。**存在于 `Object()` 类中，所以所有类都具有 `equals()` 方法存在两种使用情况：

1. **类未覆盖 `equals()` 方法**：此时通过 `equals()` 比较该类的两个对象时，等价于 `==` 比较这两个对象，默认使用 `Object` 类中的 `equals()` 方法；
2. **类覆盖了 `equals()` 方法**：一旦覆盖了该方法，则用来比较两个对象的内容是否相等，如我们常用的 `String、BitSet、Data、File` 就覆盖了 `equals()` 方法；

```
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/15 13:26
 * @description : == 和 equals 用法
 */

public class Main {
      public static void main(String[] args) {
            int num1 = 10;
            int num2 = 10;
            int num3 = 20;
            String str1 = "村雨遥";
            String str2 = new String("村雨遥");

            // == 用于基本数据类型，用于比较两者是否相等
            System.out.println("num1 == num2: " + (num1 == num2)); // true
            System.out.println("num3 == num3: " + (num2 == num3)); // false
            // 而用于引用类型则表示两者是否指向同一对象
            System.out.println("str1 == str2: " + (str1 == str2)); // false

            // equals 用于比较引用类型是否内容是否相同
            System.out.println("str1 equals str2: " + str1.equals(str2)); // true

            // equals 不能用于基本数据类型比较
            //          System.out.println(num1.equals(num2));
      }
}
```

2. 进制转换

2.1 二进制

二进制是计算机中采用最广泛的一种数制，用 0 和 1 两个数码来表示，其进位规则是逢二进一，而借位规则则是借一当二。

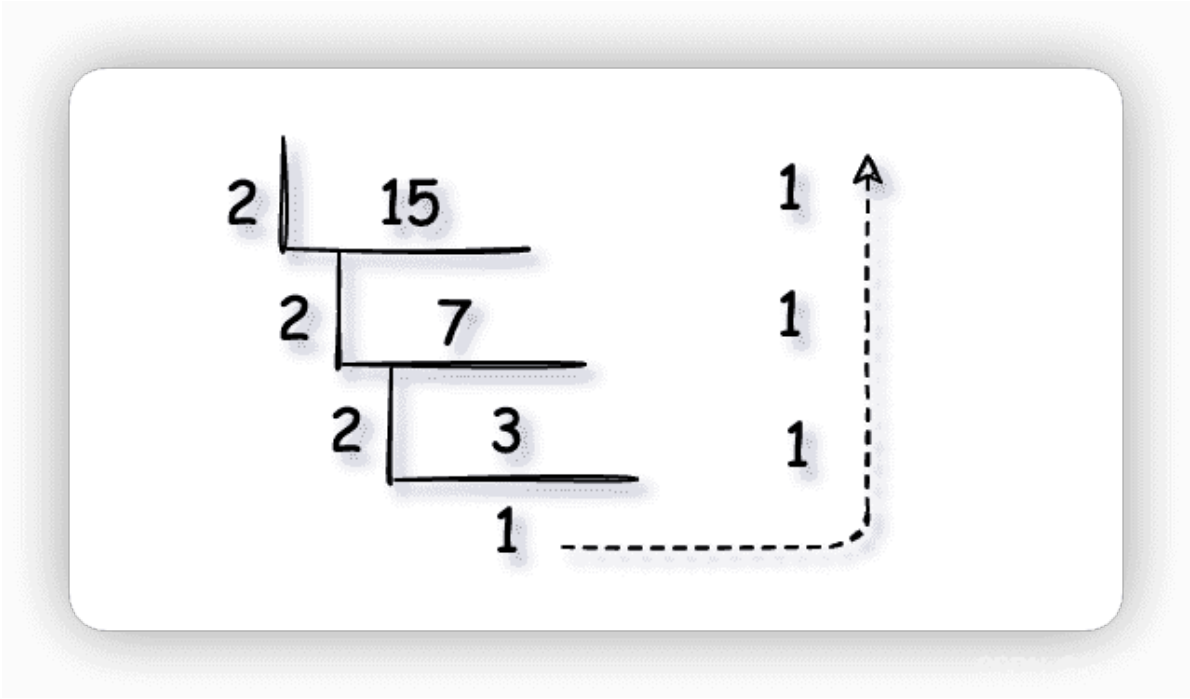
如果要将二进制转换为十进制，则采用 **按权展开求和** 方法，其步骤是先将二进制的数写成加权系数展开式，然后再更具十进制的加法规则进行求和。

$$(1011)_2=1*2^3+0*2^2+1*2^1+1*2^0=(11)_{10}$$

2.2 十进制

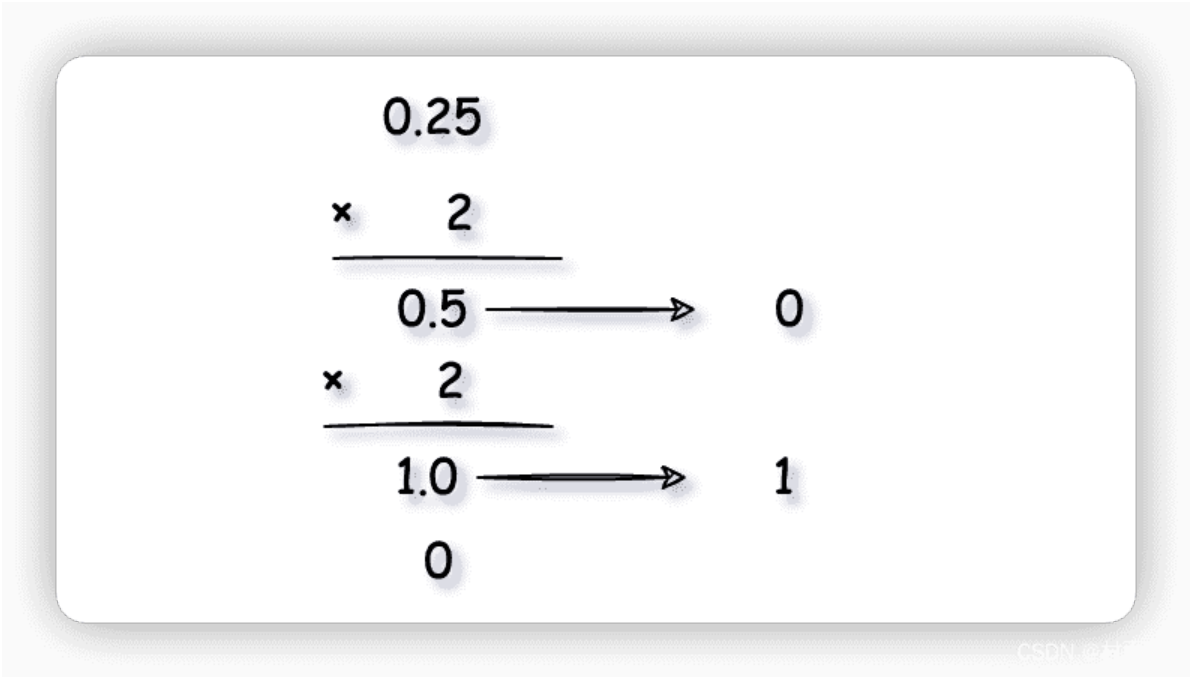
而一个十进制数要转换为二进制数，则需要将整数和小数部分分别转换，最后再进行组合。其中，整数部分采用**除二取余，逆序排序**的方法。具体方法是用 2 来整除一个十进制数，从而得到一个商和余数；然后再用 2 去除以商，从而又得到一个商和余数，重复这个步骤，直到最后得到的商小于 1 时为止。最后把按照得到余数的先后顺序，逆序依次排列，得到的数即为这个十进制数的二进制表示。

$$(15)_{10}=(1111)_2$$



小数部分则不同于整数部分，小数部分要使用**乘 2 取整法**，即用十进制的小数部分乘以 2，然后取结果的整数部分，然后再用剩下的小数重复刚才的步骤，直到最后剩余的小数为 0 时停止。最后将每次得到的整数部分按照先后顺序正序排列，从而得到对应的二进制表示。

$$(0.25)_{10}=(0.01)_2$$

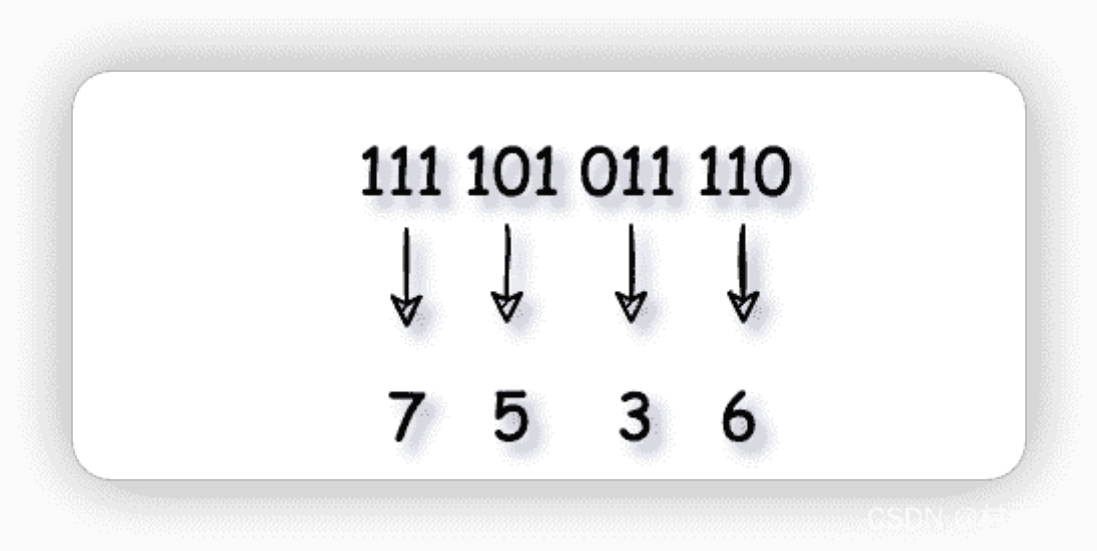


2.3 八进制

所谓八进制，就是每 3 位二进制作为一个单元，其中最小的数是 0，最大的数是 7，一共 8 个数字。

要将二进制的数转换为八进制，需要将 3 个连续的数拼成一组，然后再独立转成八进制中的数字。

例如，二进制的 111101011110 可以转换为八进制中的 7536。

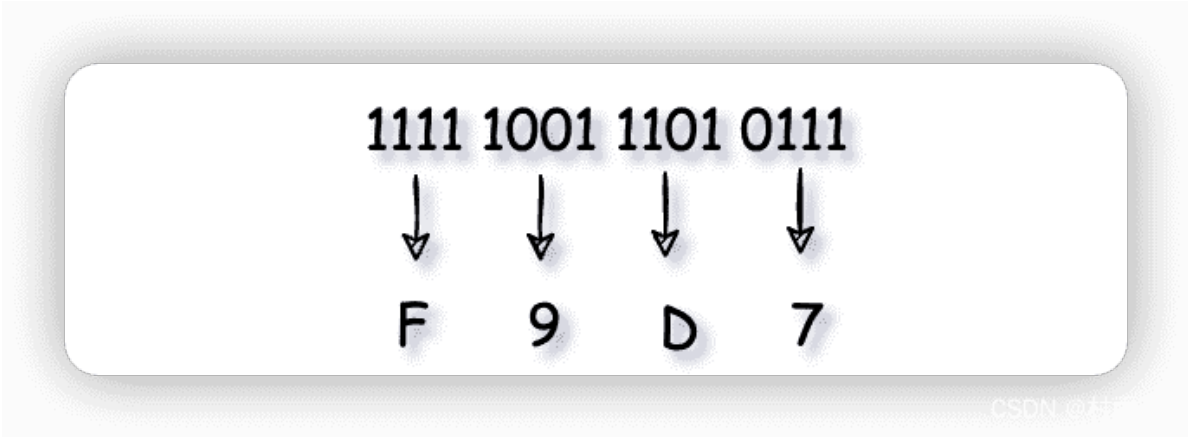


2.4 十六进制

所谓十六进制，就是每 4 位二进制作为一个单元，其中最小数是 0，最大数是 15，一共 16 个数字，分别用 0 ~ 9、A、B、C、D、E、F 表示。

要将二进制转换为十六进制，需要将 4 个连续的数拼成一组，然后再独立转换为十六进制中对应的数字。

例如，二进制的 1111100111010111 可以转换为十六进制中的 $F9D7$。



2.5 常用进制转换

当然，Java 中也已经将常用的进制转换方法封装好了，我们只需要调用对应方法即可。

转换	方法	返回
十进制 -> 二进制	`Integer.toBinary(int num)`	二进制字符串
十进制 -> 八进制	`Integer.toOctalString(int num)`	八进制字符串
十进制 -> 十六进制	`Integer.toHexString(int num)`	十六进制字符串
十进制 -> N 进制	`Integer.toString(int num, int N)`	N 进制字符串

```java
public class Main {
    public static void main(String[] args) {
        int num = 200;
            System.out.println(num + " 的二进制是:" + Integer.toBinaryString(num));
            System.out.println(num + " 的八进制是:" + Integer.toOctalString(num));
            System.out.println(num + " 的十六进制是:" + Integer.toHexString(num));
            System.out.println(num + " 的三进制是:" + Integer.toString(num, 3));
    }
}
```

```
200 的二进制是:11001000
200 的八进制是:310
200 的十六进制是:c8
200 的三进制是:21102
```

相反的，如果我们要将一个 `N` 进制的字符串 `str` 转换为十进制数，那么可以使用以下方法。

转换	方法	返回
N 进制 -> 十进制	`Integer.parseInt(String str, int N)`	十进制数

```java
public class Main {
    public static void main(String[] args) {
        String str = "21104";
            int N = 5;
            System.out.println(str + " 的十进制是:" + Integer.parseInt(str, N));
    }
}
```

第四章 流程控制

1. 输入输出

之前的学习中，我们会发现都是通过定义变量并赋初值的方式来得到一个实现固定好值得变量。加入我们现在不想再以这种方式获取变量值，而想要直接控制变量值，又该怎么做呢？这就涉及到 Java 中的输入输出相关知识了，以下就先来看看，如何实现从控制台输入，并从控制台输出吧。

1.1 输入

先来看一个实例：

```java
import java.util.Scanner;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
```

```java
 * @date : 2021/4/15 13:53
 * @description : 输入
 */

public class Main {
      public static void main(String[] args) {
            Scanner scanner = new Scanner(System.in);
            System.out.println("输入整型");
            int num = scanner.nextInt();
            System.out.println("输入的整型：" + num);

            System.out.println("输入字符型");
            String name = scanner.next();
            System.out.println("输入的字符型：" + name);

            System.out.println("输入浮点型");
            float floatNum = scanner.nextFloat();
            System.out.println("输入的字符型：" + floatNum);
            double doubleNum = scanner.nextDouble();
            System.out.println("输入的字符型：" + doubleNum);


      }
}
```

```
输入整型
1943
输入的整型：1943
输入字符型
村雨遥
输入的字符型：村雨遥
输入浮点型
943.2
输入的字符型：943.2
9042.34
输入的字符型：9042.34

Process finished with exit code 0
```

要实现从控制台输入并读取到我们的程序中时，需要借助 `Scanner` 类，它属于标准输入流，其步骤总结如下：

1. 首先，需要导入 `Scanner` 类。即 `import java.util.Scanner`，其中 `import` 表示导入某个类，并且只能放在程序的开头。
2. 然后创建 `Scanner` 对象。这里需要注意，创建时需要传入 `System.in`，表示标准输入流，与之对应的 `System.out` 则代表标准输出流。
3. 最后就是读取用户输入即可。这里读取时，调用不同的方法 `Scanner` 会自动转换数据类型，不用我们去进行手动转换。

从控制台获取不同类型的输入，其常用方法如下：

返回值	方法名	描述
`boolean`	`hasNext()`	如果还有输入，则返回 `true`，否则返回 `false`
`String`	`next()`	返回输入的字符串，以空格为分隔符
`String`	`nextLine()`	返回输入的字符串，以换行为分隔符
`int`	`nextInt()`	输入整型数
`long`	`nextLong()`	输入长整型数
`float`	`nextFloat()`	输入单精度数
`double`	`nextDouble`	输入双精度数

这里值得注意的是 `next()` 和 `nextLine` 两个方法，虽然他们的作用都是用于获取输入的 `String` 类型的内容，但是它们具体的处理机制又会有所区别。
针对 `next()` 而言，它会自动消除有效字符前的空格，从而只返回输入的字符，得到的字符串都不会带有空格。也就是说，当使用 `next()` 时，如果遇到空格，此时就会停止录入，只录入空格前的内容，而空格后的内容则会保留到缓冲区。除了空格之外，`next()` 也会对制表符和换行符采用同样的处理方式。
而对 `nextLine()` 来说，它会返回换行符之前的所有内容，甚至是带空格的字符串。

因此，在使用时一定要注意它们之间的区别，合理搭配使用，从而得到自己想要的结果。

1.2 输出

其实从一开始的 `hello world` 到目前的代码中，我们已经接触过输出了，也就是我们的 `System.out.println()`。

其中 `println` 表示输出并换行，如果我们不想换行，则使用 `print` 就可以了。

通过上述方式所输出的内容都是挤在一起的，十分不方便我们阅读。为了更加清晰的打印出我们所需要的结果，可以使用格式化输出。

要使用格式化输出，需要使用 `System.out.printf()` 或者 `System.out.format()` 搭配占位符，然后在后面的参数格式化成指定格式即可，两者达成的效果是等价的。常见的占位符如下：

占位符	描述
`%d`	格式化输出整数
`%f`	格式化输出浮点数
`%s`	格式化输出字符串
`%x`	格式化输出十六进制整数
`%e`	格式化输出科学计数法表示的浮点数

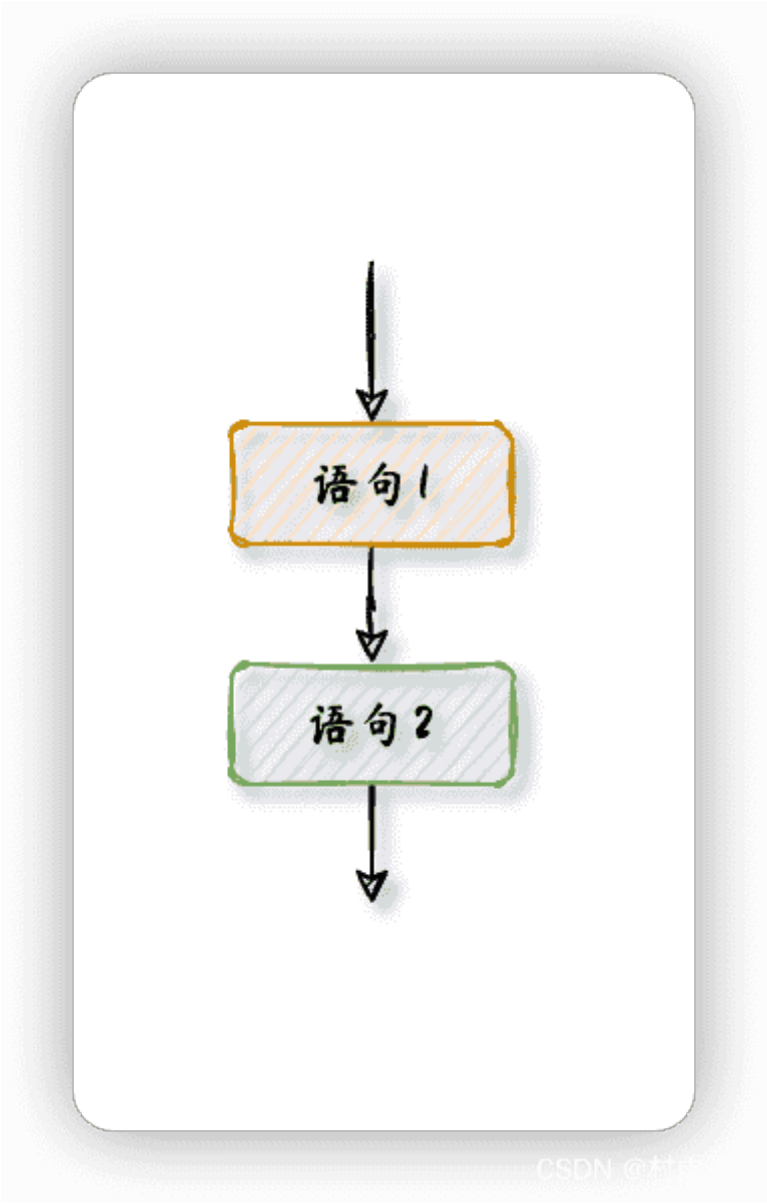
此外，我们还可以使用各种转义字符来使得我们的输出更为简洁，常见的转义字符及意义如下表所示。

转义字符	描述
\n	换行
\t	水平制表符
\\	表示一个反斜杠
\’	表示一个单引号
\”	表示一个双引号

```java
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/15 14:48
 * @description : 输出
 */

public class Main {
      public static void main(String[] args) {
            int num1 = 10;
            double num2 = 34.9;
            float num3 = 3.34f;
            String name = "村雨遥";

            //          换行及不换行输出
            System.out.println("公众号：" + name);
            System.out.print("公众号：" + name);
            System.out.println(num1);

            //          格式化输出
            System.out.println("格式化输出：");
            System.out.printf("num1 = %d\n", num1);
            System.out.format("num2 = %f\t num3 = %f\n", num2, num3);
            System.out.printf("name = %s\n", name);
            System.out.format("name = %s\n", name);
      }
}
```

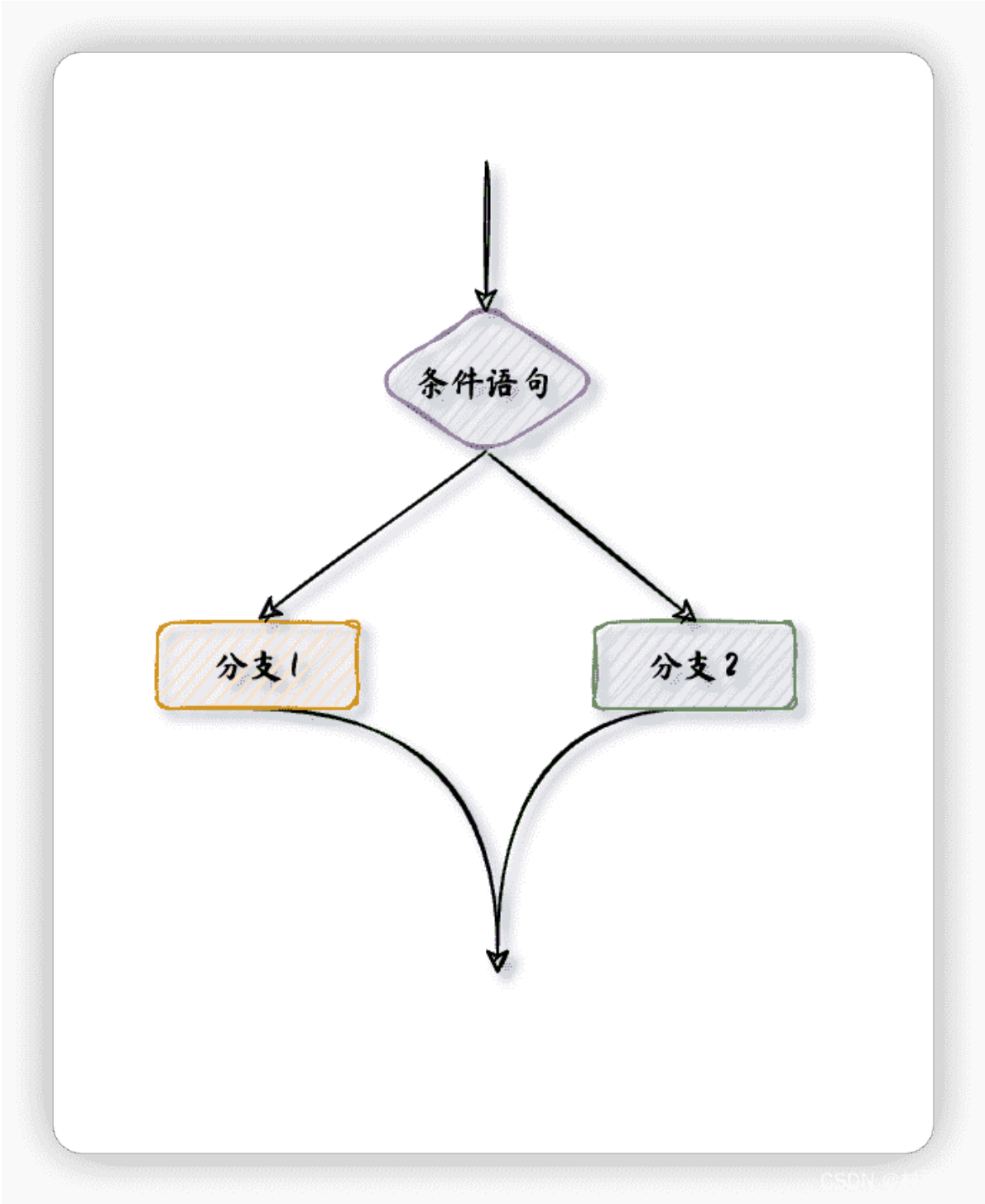
```
公众号：村雨遥
公众号：村雨遥10
格式化输出：
num1 = 10
num2 = 34.900000      num3 = 3.340000
name = 村雨遥
name = 村雨遥

Process finished with exit code 0
```

2. 顺序结构



顺序结构如上图所示，它可以说是最简单的，只需要按照解决问题的顺序写出对应的语句即可，其执行顺序是自上而下，依次执行的。就类似于我们求解一道数学题，你得根据题意一步一步来，直至解出最后的答案。

3. 分支结构



上图是分支结构，顺序结构虽然能够处理计算、输出等问题，当遇到需要判断选择时，顺序结构已经不能很好的解决了，此时就需要使用分支结构。
Java 中，分支结构相关的语句主要涉及到 `if` 和 `switch` 相关，下面就分别来看一下。

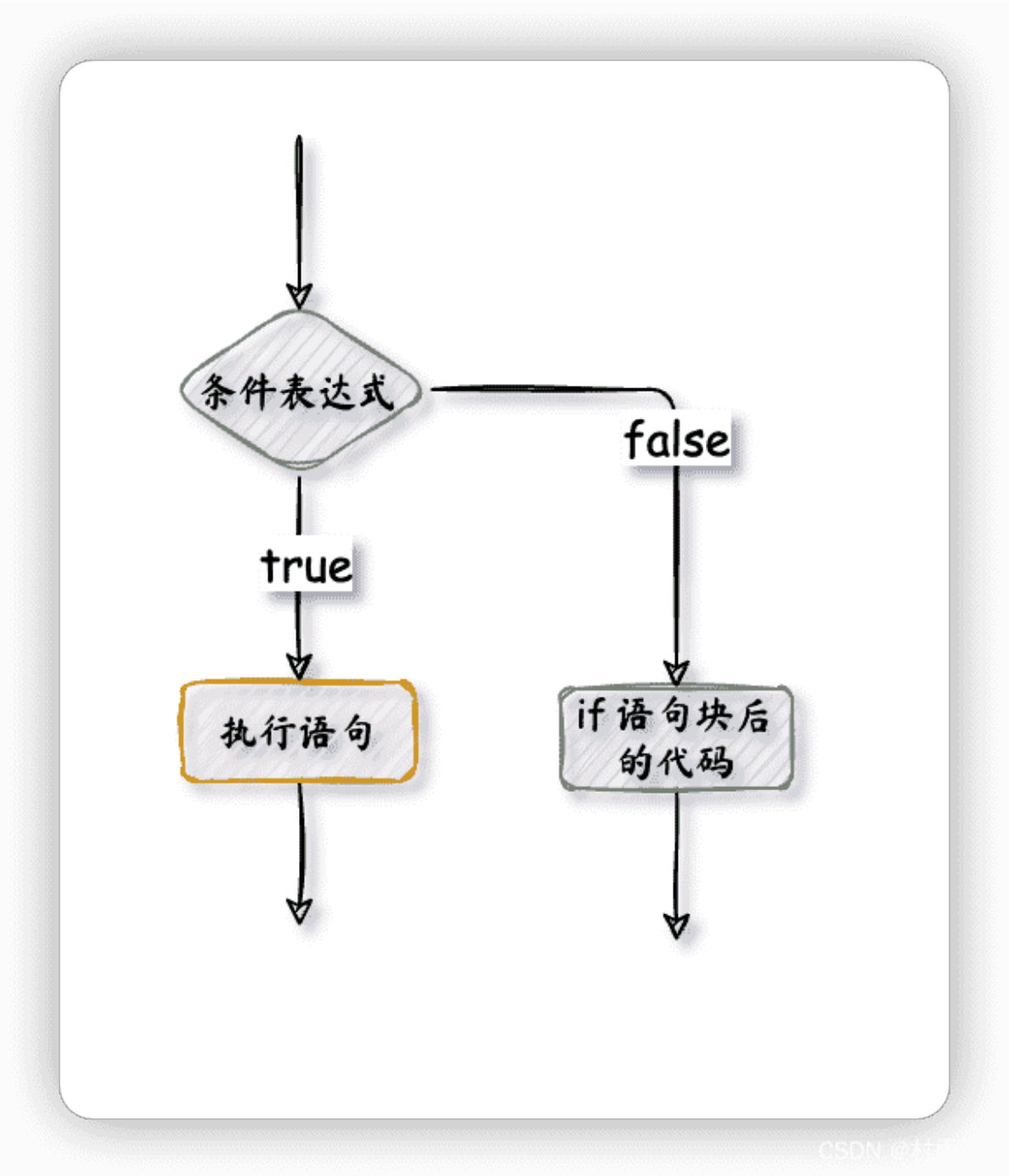
if

1. **单次判断**

当我们只进行一次判断时，可以使用一个 `if` 语句包含一个条件表达式，其语法格式如下；

```
if(条件表达式){
        执行语句;
}
```

其执行逻辑如下图所示，如果条件表达式的值为 `true`，则执行 `if` 语句块中的执行语句，否则就执行 `if` 语句块后边的代码；

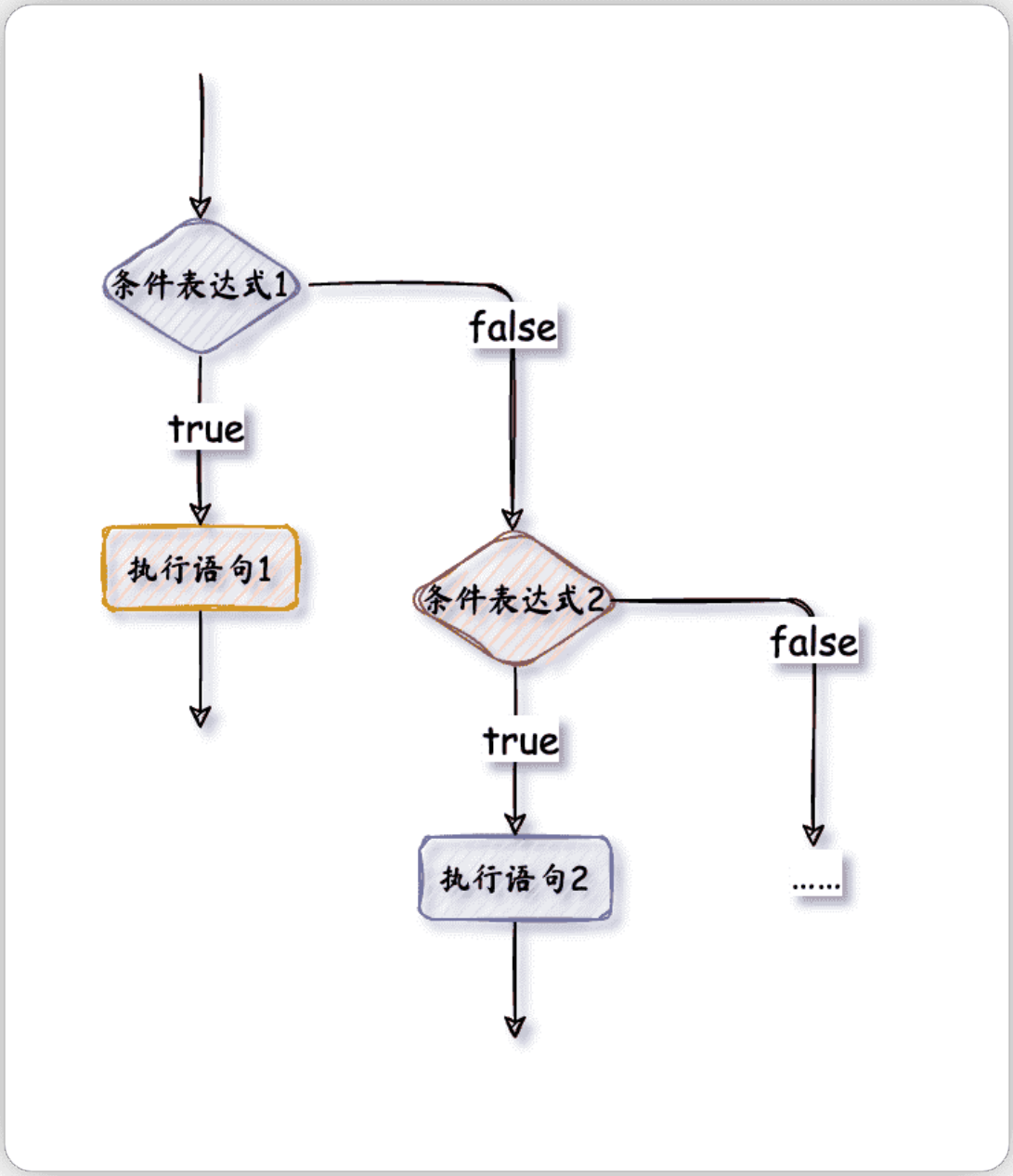


2. **多次判断**

要进行多次判断时，可以使用 `if…else` 的形式，其语法格式如下；

```
if(条件表达式 1){
        执行语句 1;
} else if(条件表达式 2){
     执行语句 2;
} else if(…){
        …
}…
```

其执行逻辑如下图所示，如果条件表达式 1 为 `true`，则执行执行语句 1，否则接着判断条件表达式 2，若为 `true`，则执行执行语句 2，以此类推，直到完成最后一个条件表达式的判断。




```java
import java.util.Scanner;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/15 15:45
 * @description : 条件判断
 */

public class Main {
      public static void main(String[] args) {
            Scanner scanner = new Scanner(System.in);
            System.out.println("请输入分数：");
            double score = scanner.nextDouble();

            if (score < 0 || score > 100) {
                  System.out.println("输入的分数不在0-100之间，不符合要求");
            } else if (score >= 90) {
```

```
                System.out.println("A");

            } else if (score >= 80) {
                System.out.println("B");

            } else if (score >= 60) {
                System.out.println("C");
            } else {
                System.out.println("D");

            }
        }
    }
```

```
请输入分数:
43
D

Process finished with exit code 0
```

switch

当我们需要使用 `if` 进行判断时，难免显得有些繁琐。此时，我们就可以使用 `switch` 来进行替代，通过判断一个变量所属范围，从而划分出不同的分支。

`switch` 分支的语法格式如下：

```
switch(表达式){
    case value1:
        执行语句1;
        break;
    case value2:
        执行语句2;
        break;
    ......
    default:
        执行语句;
        break;
}
```

通过判断表达式的值，然后执行对应值下的执行语句，而 `default` 下的执行语句表示如果 `switch` 表达式未匹配到对应的值时所执行的语句；

一个 `switch` 的实例如下：

```
import java.util.Scanner;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/15 15:49
 * @description : switch
 */

public class Main {
```

```
        public static void main(String[] args) {
            Scanner input = new Scanner(System.in);
            System.out.print("请输入该学生成绩：");
            int grade = input.nextInt();//定义grade且获取键盘输入的整数
            if (grade < 0 || grade > 100) {//输入的整数范围应为0-100
                System.out.println("输入的成绩有误");
                return;
            }
            switch (grade / 10) {
                case 10:
                case 9:
                    System.out.println("该学生成绩优秀");
                    break;
                case 8:
                    System.out.println("该学生成绩良好");
                    break;
                case 7:
                    System.out.println("该学生成绩中等");
                    break;
                case 6:
                    System.out.println("该学生成绩基本合格");
                    break;
                default:
                    System.out.println("该学生成绩不合格");
                    break;
            }
        }
    }
```

除了上面的形式之外，也可以使用以下形式：

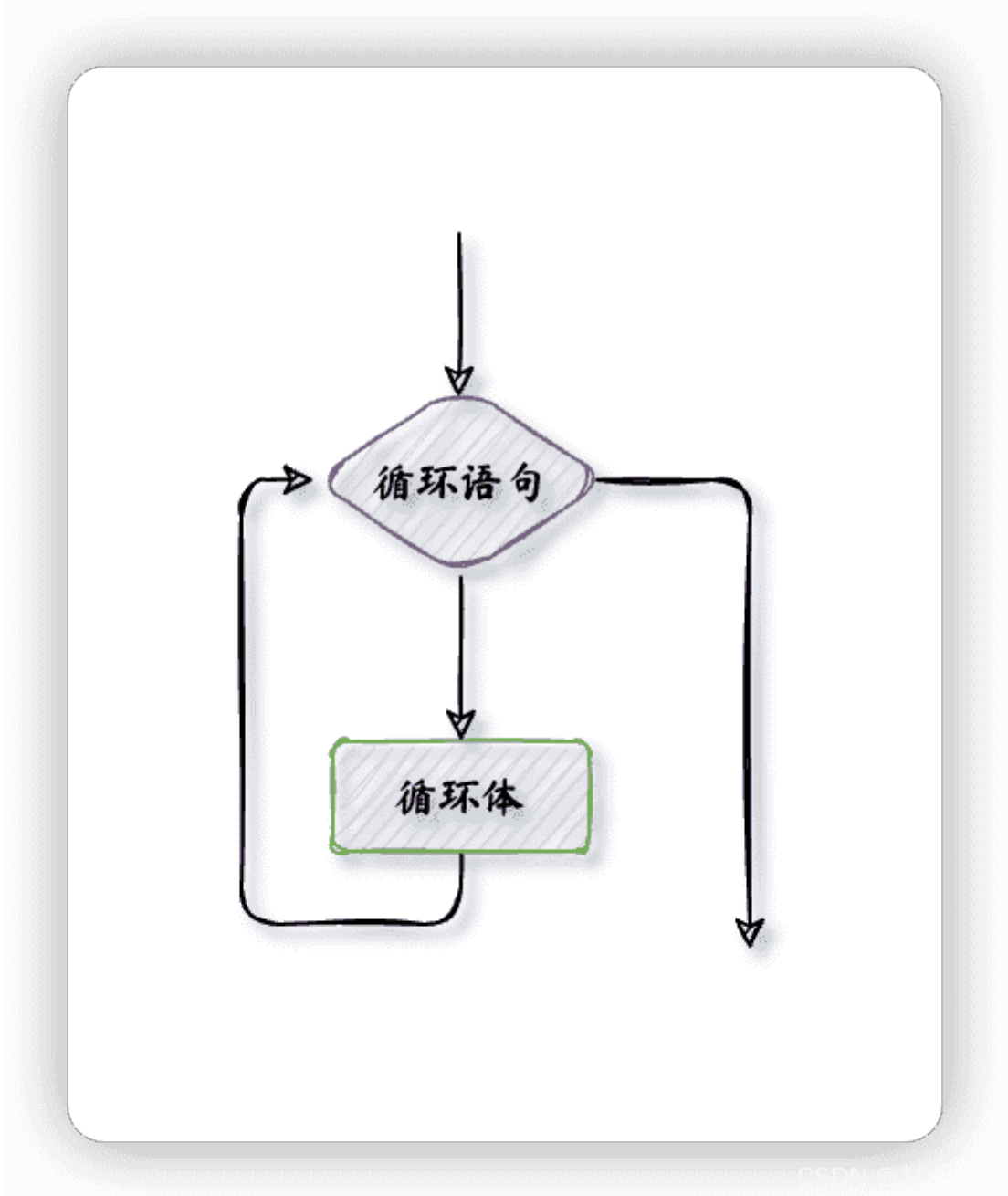
```
import java.util.Scanner;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/15 15:49
 * @description : switch
 */

public class Main {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("请输入该学生成绩：");
        int grade = input.nextInt();//定义grade且获取键盘输入的整数
        if (grade < 0 || grade > 100) {//输入的整数范围应为0-100
            System.out.println("输入的成绩有误");
            return;
        }
        switch (grade / 10) {
            case 10,9 -> System.out.println("该学生成绩优秀");
            case 8 -> System.out.println("该学生成绩良好");
            case 7 -> System.out.println("该学生成绩中等");
            case 6 -> System.out.println("该学生成绩基本合格");
            default -> System.out.println("该学生成绩不合格");
        }
    }
```

```
28    }
```

而在使用 `switch` 分支语法时，需要遵循一定的规则：

1. `switch` 中的变量类型可以是： `byte` 、 `short` 、 `int` 、 `char` 、 `String` （自 JDK 1.7 开始）；
2. `switch` 语句根据表达式的结果跳转到对应的 `case` 结果，然后执行其后跟着的语句，直到遇到 `break` 才结束执行；
3. 默认情况下，一般都会跟着一个 `default` 的分支，用于未匹配到对应情况时的执行情况；

4. 循环结构



上图为循环结构示意，让程序中遇到需要反复执行某一个功能时，我们发现顺序结构以及分支结构实现起来都太过于繁琐，于是又提出了循环结构的相关概念。
通过循环结构，我们就可以通过判断循环语句，然后判断是否进入循环体。Java 中，循环结构主要涉及的语句有 `while` 、 `for` 、 `continue` 、 `break` 等。

while

假设我们现在有一个题目，需要你计算 `1 + 2 + 3 + …… + 50` 的结果，你会怎么办呢？

这么写么：

```
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/16 9:35
 * @description : while
 */

public class Main {
      public static void main(String[] args) {
            int sum = 1 + 2;
            sum += 3;
            sum += 4;
            ……
            sum += 50;
            System.out.println("1 + 2 + 3 + …… + 50 = " + sum);
      }
}
```

这么写就太麻烦了，计算到 50 的值就已经很多了，假如有 1000,10000 甚至更大，那我们岂不是写个好久才能写完。这个时候我们就得找找有没有简单的方法，能够只写几句就实现相同的效果呢？答案是：Yes，这就是我们这一小节将要讲到的循环。

```
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/16 9:35
 * @description : while
 */

public class Main {
      public static void main(String[] args) {
            int sum = 0;
            int num = 1;
            while (num <= 50) {
                  sum += num;
                  num++;
            }
            System.out.println("1 + 2 + 3 + …… + 50 = " + sum);
      }
}
```

```
1 + 2 + 3 + …… + 50 = 1275

Process finished with exit code 0
```

从上面的实例，利用 `while` 循环，我们就能轻易达成循环的效果。其语法格式如下：

```
while(表达式){
        执行语句;
}
```

只要表达式为 `true`，就会不断循环执行其中的执行语句，直到表达式为 `false`，此时便跳出循环，不再执行其中的执行语句。

除开上面的形式之外，还有另一种 `while` 形式：

```
do{
        执行语句;
}while(表达式);
```

两者的最大区别在于：`do……while` 无论 `表达式` 是否为 `true`，都至少会执行一次。

```
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/16 9:35
 * @description : while
 */

public class Main {
      public static void main(String[] args) {
            int sum = 0;
            int num = 10;
            while (num <= 9) {
                  sum += num;
                  num++;
            }
            System.out.println("sum = " + sum);
      }
}
```

```
sum = 0

Process finished with exit code 0
```

```
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/16 9:35
 * @description : while
 */

public class Main {
      public static void main(String[] args) {
            int sum = 0;
            int num = 10;
            do {
                  sum += num;
                  num++;
            } while (num < 10);
            System.out.println("sum = " + sum);
```

```
        }
    }
```

```
sum = 10

Process finished with exit code 0
```

观察以上两个实例，在 `while` 程序中，我们定义 `num = 10`，然后假设 `num <= 9` 时就进入循环体，而 $10 > 9$，所以不进入循环体，直接打印 `sum` 的值为 `0`。而在 `do…while` 程序中，我们同样定义 `num = 10`，然后假设 `num < 10` 时继续循环，很明显不满足该情况，理应跳出循环，打印出 `sum` 的值为 `10`，说明此时还是进行了一次循环。

因此，当我们需要在 `while` 和 `do……while` 之间做出选择时，如果我们最少需要进行一次循环，则选择 `do……while`，其他情况下选用两者都可以。

for

- **普通 for 循环**

除开 `while` 和 `do……while` 之外，我们还有 `for` 循环来达成同样的结果，只是表达方法不一样。同样以上面计算 `1 + 2 + …… + 50` 为例，可以写成如下的形式：

```
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/16 10:20
 * @description : for 循环
 */

public class Main {
      public static void main(String[] args) {
            int sum = 0;
            for (int num = 1; num <= 50; num++) {
                  sum += num;
            }

            System.out.println("1 + 2 + …… + 50 = " + sum);
      }
}
```

```
1 + 2 + …… + 50 = 1275

Process finished with exit code 0
```

`for` 循环的语法形式如下：

```
for(初始条件;终止条件;更新语句){
      循环语句;
}
```

`for` 循环的执行步骤如下：

1. 首先执行初始条件，可以声明一种类型，但可以初始化一个或多个循环控制变量，甚至可以放空。

2. 接着判断终止条件，如果为 `true`，则进入循环体执行循环语句；如果为 `false`，则终止循环，执行循环体后面的语句。
3. 一次循环完成后，执行更新语句来更新循环控制变量。
4. 最后再次判断终止条件，循环以上三个步骤。

`for` 和 `while` 最大的区别就在于 `for` 循环一般都是事先知道需要循环的次数的，而 `while` 循环则不需要。

- **增强 for 循环**

自 Java 5 后，引入了一种增强型 `for` 循环，主要用于数字遍历，其语法格式如下：

```
for(声明语句:表达式){
    // 循环语句
}
```

```
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/16 10:39
 * @description : 增强 for 循环
 */

public class Main {
      public static void main(String[] args) {
            int[] numbers = {1, 4, 5, 6, 9, 10};
            for (int number : numbers) {
                  System.out.print(number + "\t");
            }
      }
}
```

```
1    4    5    6    9    10
Process finished with exit code 0
```

其中，声明语句一般是声明一个同数组数据类型相同的局部变量，而表达式则是要访问的数组名或者返回值是数组的方法。

for 和 while 的区别

经过上面的学习，我们可以发现，基本能用 `for` 循环的，都能将其改写成 `while` 循环。而使用 `while` 循环的，也可以在一定程度上改写成 `for` 循环。两者的运行规则都是一样的，那针对什么场景该使用 `for`，什么场景又该使用 `while` 呢？

通常，如果我们知道循环的次数或者循环的范围，那么我们优先使用 `for` 循环。如果不知道循环的次数和范围，而只知道循环的结束条件，那么此时优先使用 `while` 循环。

continue & break

break

主要用在循环语句或者 `switch` 语句中，表示跳出最里层的循环，然后继续执行该循环下的语句。

`break` 在 `switch` 语句中的用法已经见识过了，我们就来看看它在循环中的应用。

```
/**
 * @author : cunyu
 * @version : 1.0
```

```
 * @className : Main
 * @date : 2021/4/16 10:51
 * @description : break & continue
 */

public class Main {
      public static void main(String[] args) {
            for (int i = 1; i < 10; i++) {
                  System.out.println("i = " + i);
                  if (i == 5) {
                        break;
                  }
            }
      }
}

```

```
i = 1
i = 2
i = 3
i = 4
i = 5

Process finished with exit code 0
```

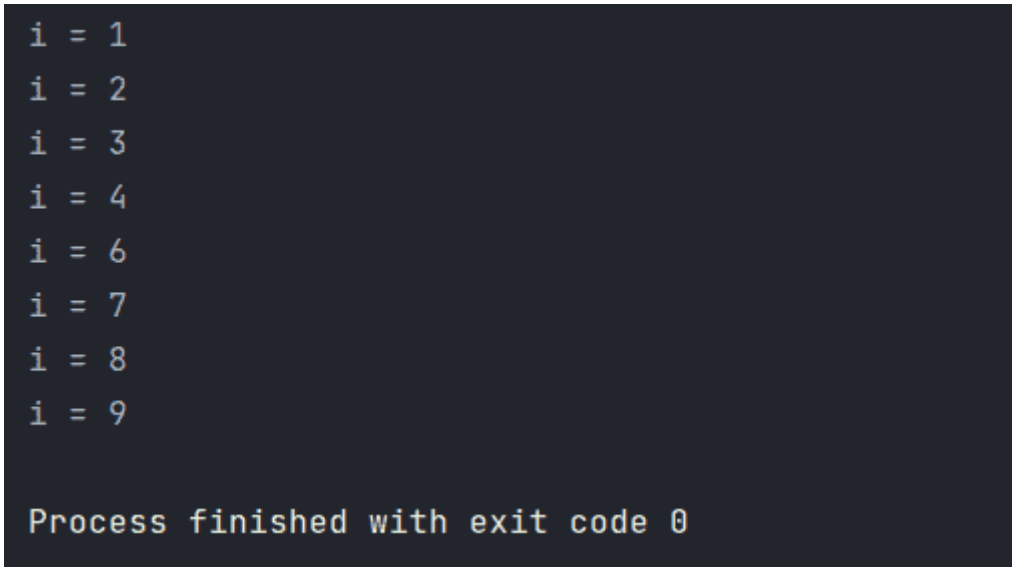
观察结果可知，当 `i == 5` 时，我们执行了 `break` 语句，此时就直接跳出了 `for` 循环，而不再进行下一次的循环。

continue

`continue` 也同样是应用在循环控制结构中，主要是让程序跳出当次循环，进而进入下一次循环的迭代。

在 `for` 循环中，执行 `continue` 语句后，直接跳转到更新语句，而不再执行 `continue` 后的语句。而在 `while` 或 `do……while` 循环中，执行 `continue` 语句后，直接跳转到表达式的判断。

```
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Main
 * @date : 2021/4/16 10:51
 * @description : break & continue
 */

public class Main {
      public static void main(String[] args) {
            for (int i = 1; i < 10; i++) {
                  if (i == 5) {
                        continue;
                  }
                  System.out.println("i = " + i);
            }
      }
}
```



观察上述结果可知，当 `i == 5` 时，我们执行了 `continue` 语句，此时便跳出了当次循环，不再进行后边的打印语句，然后继续下一次的循环，所以最终打印的结果没有 5.

第五章 数组

1. 前言

前边已经讲过了 Java 中的 8 大基本数据类型，这篇文章主要就来讲讲引用类型中的数组。主要内容安排如下：

- **数组简介**
- **遍历**
- **排序**
- **常用方法**

2. 数组简介

所谓数组，其实就是多个相同数据类型的元素按一定顺序排列而成的集合。即将有限个类型相同的变量放到一个容器中，然后按照编号去访问各个元素，而这个容器的名字就叫做数组名，各个元素的编号就叫做索引位置。而其中索引位置是从 `0` 开始计数，而不是我们日常所习惯的 `1`。

要定义一个数组类型的遍历，其语法为 `数据类型 []`，比如 `int[]`，而且在初始化时必须指定数组大小，如 `int[] arr = new int[10]`，表示 `arr` 数组可以容纳 `10` 个 `int` 类型的元素。

数组的特点：

- 数组中的元素初始化时都是有默认值的，整型对应 `0`，浮点型对应 `0.0`，而布尔型对应 `false`。
- 数组一经创建，其大小（长度）就不可再变。
- 要访问数组中的某一元素，需要用到索引，索引从 `0` 开始。
- 如果要修改数组中的某一元素，直接对对应索引位置的元素使用赋值语句即可。

和变量一样，既然定义了，那么就要对数组进行初始化。所谓数组的初始化，指的是在内存中，为数组容器开辟空间，然后将数据存入容器中的过程。而常用的初始化方式主要有两种：

- **静态初始化**
- **动态初始化**

2.1 一维数组

2.1.1 初始化

1. **静态初始化**

所谓静态初始化，就是在定义数组的同时将其初始化。

静态初始化的完整格式如下：

```
数据类型[] 数组名 = new 数据类型[]{元素 1，元素 2，元素 3…};
```

但在开发时，为了方便，也可以对上面的代码进行简写：

```
数据类型[] 数组名 = {元素 1，元素 2，元素 3…};
```

两种方式最终得到的结果是一样的，但第二种更加简洁方便。

```
int[] arr1 = {1, 3, 5, 8, 10};
int[] arr2 = new int[] {1, 3, 5, 8, 10};
```

2. **动态初始化**

动态初始化，就是先声明数组长度定义之后，再由系统对其分配初始值。

```
数据类型[] 数组名 = new 数据类型[数组长度];
```

```
int[] arr = new int[5];
arr[0] = 1;
arr[1] = 3;
arr[2] = 5;
arr[3] = 8;
arr[4] = 10;
```

PS：注意数组的索引位置不能超过数组的长度，如上面例子中的数组长度为 `5`，所以我们数组的索引位置最大只能为 `4`，否则就会报数组越界错误。

2.1.2 求数组长度

求数组的长度，利用数组的 `length` 属性即可；

```
int[] arr = new int[10];
int size = arr.length; // 10
System.out.println("size = " + size);
```

2.2 二维数组

2.2.1 初始化

1. **静态初始化**

```
int[][] arr1 = {{1, 2, 4}, {5, 7, 9}, {19, 12, 18}};
```

2. **动态初始化**

二维数组动态声明时，一种是把行和列的长度都指定，而另一种是只需要指定行的长度，不用指定列的长度，列的长度会在初始化时自动确认。

```
int[][] arr1= new int[3][3];
arr1 = new int[][]{{1, 2, 4}, {5, 7, 9}, {19, 12, 18}};

int[][] arr2= new int[3][];
arr2 = new int[][]{{1, 2, 4}, {5, 7, 9}, {19, 12, 18}};
```

2.2.2 求数组长度

不同于一维数组，因为二维数组有行和列，所需求长度时需要分别求。

```
int[][] arr = new int[10][20];

// 求行的长度
int row = arr.lenght;
// 求列的长度，此时求任意一行所在的列的长度即可
int col = arr[0].length;
```

2.3 静态初始化和动态初始化的区别

上面分别对一维数组和二维数组进行了静态初始化和动态初始化，通过比较总结出两者的区别如下：

动态初始化：手动指定数组长度，然后由系统给出默认初始化值。

静态初始化：手动指定数组元素，然后系统会根据元素个数计算出数组长度。

3. 数组遍历

既然我们已经学会了数组的声明及初始化，接下来就是对数组进行操作，而最常见的则是遍历数组。所谓遍历，就是将数组中的所有元素取出来，然后操作这些取出来的元素。

假设我们有一个数组如下：

```
String[] arr = new String[5];
arr = new String[]{"村雨遥", "海贼王", "进击的巨人", "鬼灭之刃", "斗罗大陆"};
```

3.1 标准 for 循环

首先，我们来使用标准的 `for` 循环来遍历该数组，只需要通过数组的索引位置来访问即可。

```
/**
 * @author : cunyu1943
 * @version : 1.0
 * @className : TraverseTest
 * @date : 2021/4/25 11:52
 * @description : 遍历
 */

public class TraverseTest {
      public static void main(String[] args) {
            String[] arr = new String[5];
            arr = new String[]{"村雨遥", "海贼王", "进击的巨人", "鬼灭之刃", "斗罗大陆"};

//           数组长度
            int size = arr.length;
            for (int i = 0; i < size; i++) {
                  System.out.println("第 " + (i + 1) + " 个元素：" + arr[i]);
            }
      }
}
```

```
第 1 个元素：村雨遥
第 2 个元素：海贼王
第 3 个元素：进击的巨人
第 4 个元素：鬼灭之刃
第 5 个元素：斗罗大陆

Process finished with exit code 0
```


3.2 增强 for 循环

```java
/**
 * @author : cunyu
 * @version : 1.0
 * @className : EnforceTraverseTest
 * @date : 2021/4/26 9:14
 * @description : 增强 for 循环
 */

public class EnforceTraverseTest {
      public static void main(String[] args) {
            String[] arr = new String[5];
            arr = new String[]{"村雨遥", "海贼王", "进击的巨人", "鬼灭之刃", "斗罗大陆"};
            int index = 0;
            for (String name : arr) {
                  System.out.println("第 " + (index + 1) + " 个元素：" + name);
                  index++;
            }
      }
}
```

```
第 1 个元素：村雨遥
第 2 个元素：海贼王
第 3 个元素：进击的巨人
第 4 个元素：鬼灭之刃
第 5 个元素：斗罗大陆

Process finished with exit code 0
```

两者的区别：标准 `for` 循环是通过计数器来进行遍历，我们能够很清晰的得知每个元素所对应的索引位置，而增强 `for each` 循环则是直接访问数组中的元素值，而不关心每个元素对应的索引位置。

3.3 标准库遍历

通过调用标准库 `Arrays` 中的 `toString()` 方法，我们可以将任意类型的数组转换为一个字符串表示形式，从而打印出来。

```java
import java.util.Arrays;

/**
```

```
 * @author : cunyu
 * @version : 1.0
 * @className : StandardLibraryTest
 * @date : 2021/4/26 9:22
 * @description : 标准库遍历
 */

public class StandardLibraryTest {
      public static void main(String[] args) {
            String[] arr = new String[5];
            arr = new String[]{"村雨遥", "海贼王", "进击的巨人", "鬼灭之刃", "斗罗大陆"};
            System.out.println(Arrays.toString(arr));
      }
}
```

```
[村雨遥, 海贼王, 进击的巨人, 鬼灭之刃, 斗罗大陆]

Process finished with exit code 0
```

以上的遍历主要是针对一维数组，而针对二维数组，当我们要访问其中的一个元素时，可以使用 `array[row][col]` 来进行访问，而对于二维及更多维的数组，可以使用 `Arrays.deepToString()` 。

4. 数组排序

4.1 使用排序算法

排序算法多样，最常用的则有 **冒泡排序、插入排序、快速排序等**，进行排序时会修改数组本身；

冒泡排序：经过一轮循环，将最大的数置换到末尾，然后进入下一轮循环，每轮均比上一轮的结束位置靠前一位；

```
import java.util.Arrays;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : BubbleSort
 * @date : 2021/4/26 10:16
 * @description : 冒泡排序
 */

public class BubbleSort {
      public static void main(String[] args) {
            char[] chArray = {'c', 'u', 'n', 'y', 'u'};
            // 排序前
            System.out.println(Arrays.toString(chArray));

            for (int i = 0; i < chArray.length - 1; i++) {
                  for (int j = 0; j < chArray.length - 1 - i; j++) {
                        // 从大到小
                        if (chArray[j] < chArray[j + 1]) {
                              // 交换
                              char temp = chArray[j];
                              chArray[j] = chArray[j + 1];
                              chArray[j + 1] = temp;
                        }
```

```
                }
            }
            // 冒泡排序后
            System.out.println(Arrays.toString(chArray));
        }
    }
```

```
[c, u, n, y, u]
[y, u, u, n, c]

Process finished with exit code 0
```

4.2 标准库排序

```
import java.util.Arrays;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : StandardLibrarySortTest
 * @date : 2021/4/26 10:22
 * @description : 标准库排序
 */

public class StandardLibrarySortTest {
      public static void main(String[] args) {
            String[] arr = new String[]{"村雨遥", "海贼王", "进击的巨人", "鬼灭之刃", "斗罗大陆"};

//           排序前
            System.out.println(Arrays.toString(arr));
//           标准库排序
            Arrays.sort(arr);
//           排序后
            System.out.println(Arrays.toString(arr));
      }
}
```

```
[村雨遥, 海贼王, 进击的巨人, 鬼灭之刃, 斗罗大陆]
[斗罗大陆, 村雨遥, 海贼王, 进击的巨人, 鬼灭之刃]

Process finished with exit code 0
```

5. 常用方法

对于数组而言，Java 标准库中已经内置了许多方法，常用的有如下一些方法：

返回值	方法	描述
`static String`	`toString(Object[] a)`	输出数组的字符串形式
`static <T> List<T>`	`asList(T…… a)`	数组转 `List`

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : CommonMethodsTest
 * @date : 2021/4/26 10:37
 * @description : 常用方法
 */

public class CommonMethodsTest {
      public static void main(String[] args) {
            String[] arr = new String[]{"村雨遥", "海贼王", "进击的巨人", "鬼灭之刃", "斗罗大陆"};
//           输出数组的字符串形式
            System.out.println("打印数组");
            System.out.println(Arrays.toString(arr));
//           数组转列表
            System.out.println("数组转列表");
            List<String> list = new ArrayList<String>(Arrays.asList(arr));
            System.out.println(list);

            System.out.println("列表转数组");
            list.add("镇魂街");
            String[] newArr = new String[list.size()];
            list.toArray(newArr);
            System.out.println(Arrays.toString(newArr));

            System.out.println("数组是否包含某一元素");
            System.out.println(Arrays.asList(newArr).contains("进击的巨人"));
            System.out.println(Arrays.asList(newArr).contains("网球王子"));
      }
}
```

第六章 面向对象(上)

1. 前言

前面已经讲了 Java 的一些环境搭建、IDE 使用、变量及数据类型、操作符、控制流程以及数组操作，今天就来看看 Java 里边的一个核心思想 - **面向对象编程。**

2. 什么是面向对象?

所谓面向对象，是一种通过对象的方式，将现实中的事物映射到计算机模型的一种编程方法。

对象的含义指的是具体的某一个事物，即我们在现实生活中能够看得见摸得着的。在面向对象程序设计中，对象指的是计算机系统中的某一个成分，主要有两层含义。一个是指 **数据**，另一个则是 **动作**。即对象是两者的结合体，通过对象不仅能够进行操作，还能对操作的结果进行记录。

在这之前，另一种编程方式是 **面向过程**，用一个具体的例子来讲，可以描述成下面这样：

> 假如有一天你很想吃酸菜鱼，那么你该怎么办呢？下面就以面向对象和面向过程给你两个选择，让你决定来选哪一个！
>
> 1. **面向对象**：打开手机，打开外卖软件，搜索酸菜鱼，然后下单，等着外卖送到家就行！
> 2. **面向过程**：先去买菜，鱼、酸菜、调料……，然后回家杀鱼、切酸菜、切调料……，再接着开始炒，最后做好盛到盘子里！

对比可以发现两者的优缺点：

- **面向过程**
  - **优点**：性能好；以例子来说自己做比起点外卖，经济又实惠，还吃得放心。
  - **缺点**：不易维护、不易复用、不易扩展；以例子来讲，要是我们自己做，临时又想吃其他的菜，又得跑去买材料啥的，麻烦！但外卖就不一样了，直接打开手机再点就是。
- **面向对象**
  - **优点**：易维护、易复用、易扩展，也就是面向过程的缺点。
  - **缺点**：性能较差；比起自己做，点外卖成本啥的可能就比较高了。

3. 面向对象的5大原则

1. **单一职责原则 SRP**

又称为单一功能原则，它规定了一个类应该只有一个发生变化的原因。也就是说类的功能要单一，不能太复杂。

举个例子来说，学校里边有学生、老师、管理员，如果将这些人统一封装在一个类中，那么到时候难以对他们的身份作区分，那么此时按照 SRP 原则，我们就可以将他们各自分为一个类，从而方便管理。

2. **开放封闭原则 OCP**

指一个模块对于扩展是开放的，但对于修改则是封闭的。也就是说可以增加功能，但是不能修改功能。

也就是说，一个类可以进行扩展（添加属性或者方法），但是对于类中已有的属性和方法，不要修改它们。

3. **里氏替换原则 LSP**

指子类能够替换父类出现在父类能够出现的任何地方。

假设有两个类 `Father` 和 `Child`，其中 `Father` 是 `Child` 的父类，那么在进行调用时，`Father` 类可以引用 `Child` 类，反之却不行。

4. **依赖倒置原则 DIP**

高层次的模块不应该依赖于低层次的模块，而应该都依赖于抽象。抽象不应该依赖于具体实现，但具体实现应该依赖于抽象。

也就是说，我们可以将同类事物的共性抽取出来，将其作为这一类事物的“高层次模块”，然后由“低层次模块”来继承或者实现“高层次模块”。

5. **接口分离原则 ISP**

指设计时可以采用多个与特定客户类相关的接口，而不是采用一个通用的接口。

4. 面向对象的 3 大特性

4.1 封装

利用抽象数据类型把数据和方法封装在一起，然后共同构成一个相互关联的对象。从而隐藏对象的属性和实现细节，只对外提供访问的接口，提高代码的复用性和安全性。

```
public class Hero{
      private String name;
      private String skill;

      public String getName(){
            return name;
      }

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

            public String getSkill(){
            return skill;
      }

      public void setSkill(String skill){
            this.skill = skill;
      }
}
```

对于以上的 `Hero` 类，它封装了 `name` 、 `skill` 等属性，如果我们想要获取 `Hero` 的 `name` 和 `skill` 属性值，那么就只能通过 `get()` 方法来获取，而如果我们想要改变这两个属性值，也只能通过 `set()` 方法来进行设置。

封装时，需要注意其原则，对象代码表什么，就封装对应的数据，并提供数据所对应的行为。

4.2 继承

定义父类之后，子类可以从基础类进行继承，这样一来，子类就可以获得父类中非 `private` 的属性和方法，从而提高了代码的复用性。

继承实现了 `IS-A` 关系，假设我们现在有一个类 `Shooter` 继承自 `Hero` ，那么此时我们就可以定义一个父类引用，然后将该引用指向它的子类对象。

`Java` 中提供了一个关键字 `extends` ，从而让一个类和另一个类建立起继承关系，其格式如下：

```
public class 子类 extends 父类{}
```

```
public Shooter extends Hero{
      ……
}
```

其中，被继承的类叫做父类（也叫超类或基类），如上述代码中的 `Hero` ，另外一个类则叫做子类（也叫派生类），比如上面的 `Shooter` 。

```
Hero shooter = new Shooter();
```

但是要注意一点：**在 Java 中，类只能单继承。**

4.3 多态

所谓多态，就是同类型的对象，表现出的不同形态，表现形式为：

```
父类类型 对象名 = 子类对象;
```

指的是父类或者接口定义的引用变量可以指向子类或具体实现类的实例对象，提高程序的扩展性。

多态又可以分为编译时多态和运行时多态，其中，编译时多态是指方法的重载，而运行时多态则指的是程序中定义的对象引用所指向的具体类型在运行期间才能确定下来。

要确定一个多态是编译时还是运行时多态，可以通过以下三个条件来区分：

- **继承**
- **覆盖（重写）**
- **向上转型**

如果同时满足以上三个条件，那么此时多态是运行时多态。

多态中，调用成员变量和成员方法时，遵循以下原则。

1. **调用成员变量：编译看左边，运行也看左边**：指 `javac` 编译时，会看左边的父类中是否有该变量，如果有则编译成功，如果没有就会编译失败。而用 `java` 运行代码时，实际获取的成员变量是父类中的值。
2. **调用成员方法：编译看左边，运行看右边**：指 `javac` 编译时，会看左边的父类中是否有该方法，如果有则编译成功，如果没有就会编译失败。而用 `java` 运行代码时，实际调用的是子类中的方法。

第七章 面向对象(中)

1. 前言

学习了面向对象编程的思想，今天就来看看面向对象编程思想在 Java 中的体现 - 类。以及有关类的相关知识，比如属性、方法、引用等。

2. 类与对象

以我们日常生活为例，我们现在很多人都养宠物，而宠物 **都有一些共同状态**，比如名字、毛色、年龄…… 这样一来我们就可以设计一个叫做**类**的东西，用来 **代表宠物** 这一类事物。

```
public class Pet{
      // 名字
      public String name;

      // 毛色
      public String furColor;

      // 年龄
      public int age;
}
```

有了这个类之后，它就相当于我们的一个模板，根据这个模板我们就能够创建一个个具体的宠物，而这些宠物，就叫做 **对象**。

```
public class Pet{
      // 名字
      public String name;

      // 毛色
      public String furColor;

      // 年龄
      public int age;

      public static void main(String[] args){
            // 创建一个对象
            Pet dog = new Pet();
            dog.name = "博美";
            dog.furColor = "white";
            dog.age = 1;

            Pet cat = new Pet();
            cat.name = "英短";
            cat.furColor = "orange";
            cat.age = 2;
      }
}
```

总结起来，类就是对对象所共有特征的描述，而对象则是真实存在的具体实例。在 Java 中，必须先设计类，然后才能创建并使用对象。

3. 属性

每个宠物都有自己的名字、毛色和年龄等一系列状态，而这些状态就叫做一个类的**属性**。而属性的类型既可以是基本类型（比如上述例子中的 `int`），也可以是引用类型（上述例子中的 `String`）。而在 Java 语言中，这些属性就叫做成员变量。成员变量的命名虽然没有强制规定，但是一般都是有一套大家通用的命名方法，即：

> 若成员变量是一个单词组成，那么一般都是小写。
>
> 若成员变量是多个单词组成，那么则采用驼峰法。
>
> 关于更多的命名规定，推荐参考阿里巴巴出品的 《Java 开发手册》，下载地址：https://github.com/cunyu1943/ebooks

成员变量的完整定义格式语如下，一般来说无需指定初始化值，它是存在默认值的。

```
修饰符 数据类型 变量名 = 初始化值;
```

数据类型	明细	默认值
基本类型	`byte`、`short`、`char`、`int`、`long`	0
基本类型	`float`、`double`	0.0
基本类型	`boolean`	`false`
引用类型	类、接口、数组、`String`	`null`

要访问属性，通常需要先创建一个对象，然后通过**对象名.成员变量**的方式来进行访问。

4. 方法

4.1 方法的定义

而除开属性之后，每个对象还能够有许多其他的功能，就像宠物都能吃东西、会叫……，那么这些他们能够做的事情，在类里边就可以用 **方法** 来进行表示。所谓方法就是程序中最小的执行单元，一般用于封装重复且具有独立功能的代码，从而提高代码的复用性和可维护性。

```
public class Pet{
      // 名字
      public String name;

      // 毛色
      public String furColor;

      // 年龄
      public int age;

      // 吃东西对应的方法
      public void eat(){
            System.out.println("吃东西！");
      }

      // 吠叫对应的方法
      public void bark(){
            System.out.println("吠叫！");
      }
}
```

方法的定义格式如下：

```
修饰符 返回值类型 方法名(形参列表){
      方法体代码(需要执行的功能代码);
      return 返回值;
}
```

其中，修饰符主要有以下 4 种不同的访问权限：

1. `default`：默认什么都不写的情况，表示在同一个包内可见，主要用于修饰类、接口、变量、方法。
2. `private`：表示在同一类中可见，常用于修饰变量和方法，但要注意，它不能用来修饰类（外部类）。
3. `protected`：表示对同一个包内的类和所有子类可见，常用于修饰变量、方法，同样的，它也不能修饰类（外部类）。
4. `public`：表示对所有类可见，常用于修饰类、接口、变量、方法。

修饰符	当前类	同一包内	子类（同一包）	子类（不同包）	其他包
`private`	☑	☒	☒	☒	☒
`default`	☑	☑	☑	☒	☒
`protected`	☑	☑	☑	☑	☒
`public`	☑	☑	☑	☑	☑

同样的，和属性一样，如果要调用一个方法，那么也需要先创建一个 Java 对象，然后通过**对象名.方法名(……)**的形式调用。

4.2 方法的分类

而对于方法，也有需要注意的几点：

1. 方法是可以有返回值的，如果要返回对应值，则其返回值的类型要与返回值相对于，对于不需要返回值的方法，则将其返回类型设置为 `void`；
2. 方法是可以有参数的，我们上述例子的方法中都是不带参数的，但如果我们有需要，就可以加上自己需要的参数，但此时注意要带上参数的类型；

总结起来，可以分为如下四种方法：

1. **无参无返回值**

```
public void methodName(){
      ……
}
```

2. **无参有返回值**

```
public boolean methodName(){
      ……
      return false;
}
```

3. **有参无返回值**

```
public void methodName(String name){
      ……
}
```

4. **有参有返回值**

```
public boolean methodName(String name){
      ……
      return false;
}
```

```
 public class Pet{
       // 名字
       public String name;

       // 毛色
       public String furColor;

       // 年龄
       public int age;

       // 具有返回值的方法
       int getAge(){
             return age;
       }

       // 带有参数的方法
       void setAge(int age){
             this.age = age;
       }
```

```
20
21          // 吃东西对应的方法
22          void eat(){
23                System.out.println("吃东西！");
24          }
25
26          // 叫唤对应的方法
27          void bark(){
28                System.out.println("叫唤！");
29          }
30    }
```

而对于方法命名的方式，也是有一定讲究的。因为一般而言方法都是一个类的动作行为，所以 **一般都是以动词开头，而如果有多个单词组合，则除开第一个单词全部小写之外，后面每个单词的第一个字母都要使用大写。**

注意到这里有个关键字 `return`，它主要用于设置方法的返回。

如果一个方法没有返回值，那么此时可以省略不写，一旦书写，那么就代表着该方法结束。比如，以下这个方法的两种书写方式最终效果都是一样的。

- 带 `return`

```
1    public void fire(){
2          System.out.println("开火……");
3          return;
4    }
```

- 不带 `return`

```
1    public void fire(){
2          System.out.println("开火……");
3    }
```

而如果一个方法有返回值，那么此时 `return` 就必须写，它表示结束方法并返回结果。

```
1    public int sum(int num1, int num2){
2          return num1 + num2;
3    }
```

4.3 重写与重载

此外，既然提到了方法，那肯定少不了重写和重载了，下面就来看看重写和重载之间的区别。

1. **重写**

所谓重写，其实就是子类对父类中允许访问的方法的实现过程进行加工重新编写，是面向对象编程中多态性的体现，通常发生在父类和子类之间。

重写时，方法的参数和返回值都不能改变。通过重写，子类可以根据自己的需要，从而去重新定义区别于父类的行为。

```
1    public class Hero{
2          public void fight(){
3                System.out.println("战斗");
4          }
5    }
```

```
pubilc class Shooter extends Hero{
      @Override
      public void fight(){
            System.out.println("远程战斗");
      }
}
```

2. **重载**

而重载则是让类以统一的方式来处理不同类型数据的一种方式。一个类中存在多个同名方法，但他们具有不同的参数个数或者类型。

简单来说，就是在 Java 的一个类中，我们可以创建多个相同名字的方法，但是这些方法之间的参数和返回值有所不同。

```
public class Calc{
      public int sum(int num1, int num2){
            return num1 + num2;
      }

      public int sum(int num1, int num2, int num3){
            return num1 + num2 + num3;
      }

      public double sum(double num1, double num2){
            return num1 + num2;
      }
}
```

4.4 参数传递机制

了解参数传递前，先来了解下形参和实参的相关概念。

实参，指在调用时所传递给方法的参数，可以是常量、变量、表达式、函数等。无论实参是何种类型的值，在进行方法调用时，都必须有确定的值，从而将这些值传递给形参。

形参，在方法定义时，`()` 中所声明的参数，目的是用来接收调用方法时传入的参数。不是实际存在的变量，所以又叫做虚拟变量。

Java 中，方法的参数传递都是通过**值传递**的机制来实现的。所谓值传递，就是在传输实参给方法的形参时，并非传输的是实参变量本身，而是通过传输实参变量中所存储的值。

但是要注意区分的是，Java 中的**基本数据类型的参数传输的是存储的数据值**，而**引用类型的参数传输的是存储的地址值**。

5. 构造方法

上面我们说了实例（也就是对象）和属性，那么当我们创建一个实例的时候，通常我们想要把它的属性也给它设置好。为了实现这一功能，这时候我们可以添加方法，从而达到这一目的，以上述设置宠物的年龄为例。

```
// 首先创建一个实例
Pet pet = new Pet();
// 接着调用方法设置年龄
pet.setAge(3);
// 查看设置年龄是否成功
System.out.println(pet.getAge());
```

可以发现通过上述调用方法的方式是可以完成这一目的的，但假设我们需要设置的属性很多，此时要全部设置属性值时就需要调用许多次 `setter` 方法，一旦遗漏一个，则实例内部状态就紊乱了。那我们就想了，有没有一种简单点的方法，能够让我们在创建实例对象的同时就把内部属性初始化了呢？

答案是：Yes！ 🐉🐉🐉

这时候我们就可以用到一类特殊的方法 - **构造方法**，所谓构造方法，就是定义在类中可以用来初始化一个类的对象，并返回对象的地址，以下就来看看这个构造方法的特殊之处。构造方法的格式如下：

```
修饰符 类名(形参列表){
      ......
}
```

其实在上面我们创建实例的时候就已经调用了构造方法了，只不过它是没有带任何参数的构造方法。以上述动物类 `Pet` 为实例，我们来看看如何编写它的构造方法。

```
public class Pet{
            // 名字
      public String name;

      // 毛色
      public String furColor;

      // 年龄
      public int age;

      // 无参构造方法
      public Pet(){}

      // 带参构造方法
      public Pet(String name, String furColor, int age){
            this.name = name;
            this.furColor = furColor;
            this.age = age;
      }
}
```

以上我们只是给出了无参的构造方法和带了所有属性的构造方法，除了上面的两个构造方法之外，我们还可以根据需要创建带有部分属性的构造方法。

其中，无参构造方法（默认存在）在初始化对象时，成员变量的数据均采用的默认值，而有参构造方法在初始化对象时，同时可以接收参数来给对象赋值。

经过对比可以发现，相比于普通的方法，构造方法有着明显的特点：

1. **没有返回值**：是的，无论是带参还是不带参的构造函数，它们都是没有返回值的，而它也是 **每个类默认的构造方法。**
2. **方法名同类名一样**：必须确保构造方法的名字和类名一致，否则它就不是构造方法了。

有了构造方法之后，我们创建实例时就可以直接给它初始化了，而不用再去麻烦地调用各个 `setter` 方法来初始化实例。

```
// 调用无参构造方法
Pet pet1 = new Pet();
// 调用有参构造方法
Pet pet2 = new Pet("柯基", "黄色", 1);
```

Tips：对于实例的属性值，在未经构造方法初始化时，各数据类型都有默认值，整型默认值为 `0`，浮点型默认值为 `0.0`，布尔类型默认值为 `false`，引用类型默认值为 `null`。

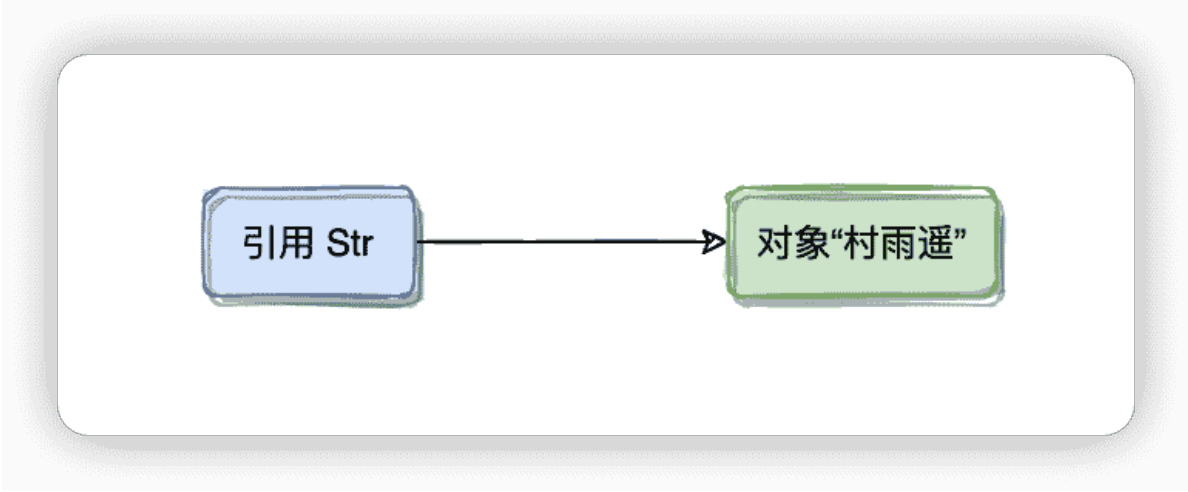
6. 引用

既然知道了什么是面向对象以及面向对象中的一些关键知识点如对象、属性、方法的概念，那我们就趁热来看看啥是引用。

所谓引用，其实在之前学习的时候就已经涉及到了。你是否还记得 `String` 这个特殊的数据类型，其实在我们创建一个 `String` 对象时，也就创建了一个引用。

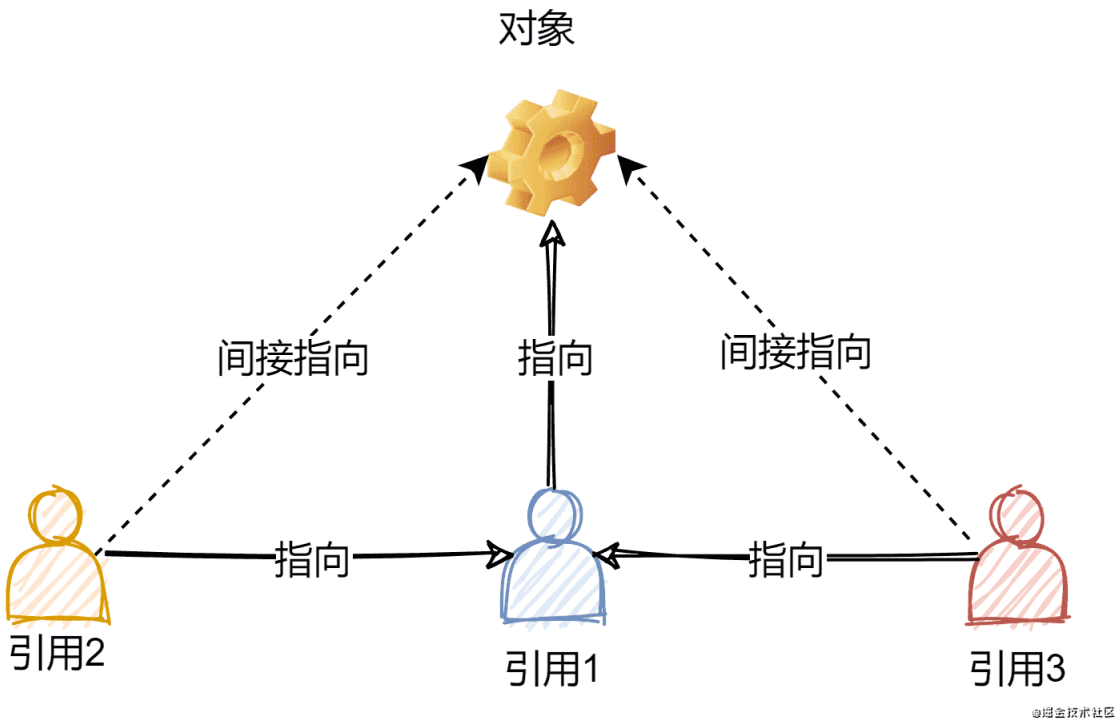
```
String str = new String("村雨遥");
```

其中 `str` 既是一个变量，也是一个引用，指向一个值为 `"村雨遥"` 的 `String` 对象，后续如果我们要访问这个 `String` 对象，就需要使用 `str` 这个引用来代表它。



以上我们说的是一个引用指向一个对象，但是我们也可以用多个引用指向同一个对象。就好比你家买了一辆车，不仅你可以开，你老婆也可以开，你爸妈也可以开。而这时候的车就好比一个“对象”，而使用它的人就是多个“引用”。

```
// 对象 1
String str1 = new String("村雨遥");
// 对象 2
String str2 = str1;
// 对象 3
String str3 = str1;
```



7. JavaBean

`JavaBean` 也可以称为实体类，其对象可以用在程序中封装数据。对于一个标准的 `JavaBean`，定义时需要满足以下要求：

- 成员变量均使用 `private` 修饰。
- 提供成员变量所对应的 `getXxx()/setXxx()` 方法。
- 必须提供一个标准的无参构造器，而有参构造器则是可写可不写。

第八章 面向对象(下)

1. 前言

在之前的文章中，讲到了面向的 3 大特性（**封装、继承、多态**）和面向对象设计的 5 大原则（SRP、OCP、LSP、DIP、ISP）。此外，我们还讲了如何创建一个类，并且在创建类后如何构造一个对象。然后还介绍了类中的属性和方法，并对构造方法和引用也做了简单的讲解。

有了上面的基础之后，今天我们来继续学习面向对象的相关知识，主要内容预告如下：

- **包**
- **注释**
- **jar 文件的创建**

2. 包

假设现在有这么一种情况，诸葛亮、周瑜、曹操共同开发一款程序。其中，周瑜和曹操均在自己代码模块中写了一个 `PublicUtil` 类，现在诸葛亮要调用周瑜和曹操模块中代码，需要同时用到他们中的 `PublicUtil` 类，这时候就犯难了，诸葛亮在他的代码中使用 `PublicUtil` 类时，该怎么区分是调用周瑜的，还是调用的曹操的呢？

针对这个问题，开发 Java 的前辈们当然也想到了。于是，他们在 Java 中定义了一种名字空间，也就是我们今天要讲的包：`package`。通过使用包机制，就十分容易区别类名的命名空间了。

一般包名的规则为：公司域名反写 + 包的作用，而且全部都要用英文小写。

假设曹操的 `PublicUtil` 类代码如下：

```
// 申明包名
package caocao;
public class PublicUtil{
      ……
}
```

周瑜的 `PublicUtil` 类代码如下：

```
// 申明包名
package zhouyu;
public class PublicUtil{
      ……
}
```

此时，如果诸葛亮要同时使用他们俩代码中的 `PublicUtil` 类，此时就可以通过引入他们俩的包，然后通过使用 `包名.类名` 的引用方式来进行区分即可。

```
 package zhugeliang;
 import caocao;
 import zhouyu;
 public class Util{
       // 使用周瑜代码
       zhouyu.PublicUtil.xxx();
       ……
       // 使用曹操代码
       caocao.PublicUtil.xxx();
       ……
 }
```

以上代码中的 `import` 你可能也在其他代码中见到过，但你不知道啥作用。其实它就是为了包的使用而生，如果我要使用另一个人的包，那该怎么做呢？其实很简单，只需要在程序中使用关键字 `import` 即可完成包的导入。

通过使用包，可以达到以下的作用：

1. 将功能类似或或相关的类以及接口组织放在同一个包中，方便类的查找与使用。
2. 包也像文件夹一样，采用了树形目录的存储方式。同一个包中的类名不同，不同包中的类名可以相同。当同时调用两个不同包中的同一类名的类时，通过加上完整的包名就可以加以区分，从而避免类名冲突。
3. 同时包也限定了访问权限，只有拥有包访问权限的类才能间接去访问包中的类。

3. 注释

所谓注释，就是写在程序里边对代码进行结束说明的文字，既方便自己也方便他人查看，更快理解程序含义。而且注释是不影响程序的执行的，在我们对 Java 源代码进行编译后，字节码文件中不含源代码中的注释内容。

在 Java 中，通常支持三种注释方式，它们分别是：

- `//`：单行注释
- `/* */`：多行注释

- `/** */`：文档注释

3.1 单行注释

单行注释是以双斜杠 `//` 来标识，表示只注释当前行内容，一般用在需要注释的内容较少的地方，以下就是一个单行注释的实例。

```
// 第一个 Java 程序
public class HelloWorld{
    public static void main(String[] args){
        System.out.println("Hello World!");
    }
}
```

3.2 多行注释

通常我们把要注释的内容放在 `/*` 和 `*/` 之间，表示在两者之间的内容都是我们的注释内容，以下是一个多行注释的实例。

```
/*
* 第一个 Java 程序
* 这是许多初学者都会写的一个程序
*/
public class HelloWorld{
    public static void main(String[] args){
        System.out.println("Hello World!");
    }
}
```

3.3 文档注释

文档注释和多行注释很像，它是将我们所需要注释的内容包含在 `/**` 和 `*/` 之间。而文档注释和其他两种注释最大的区别就在于：我们可以利用 `javadoc` 工具来提取文档注释，然后生成一个 HTML 文档，类似于 Java 官网所提供的 API 文档，以下是一个文档注释的实例。

```
/**
* 第一个 Java 程序
* 这是许多初学者都会写的一个程序
*/
public class HelloWorld{
    /**
    * 主函数
    * @param args 主函数参数列表
    */
    public static void main(String[] args){
        System.out.println("Hello World!");
    }
}
```

然后我们通过终端，使用 `javadoc` 命令就可以为上述文件生成一个 HTML 文档。

```
javadoc HelloWorld.java
```

而文档注释相比于其他两种注释，也有更多值得注意的地方，下面就分别来看看需要留意的地方。

1. **常用文档注释分类**

- **类注释**

顾名思义，所谓类注释，就是针对整个类的说明，它必须放在 `import` 之后，但又必须放在类定义之前。以下是一个类注释的实例：

```
/**
* Animal，动物类
*/
public class Animal{
    ...
}
```

这里需要注意的是，在 `/**` 和 `*/` 之间的其他行注释，`*` 是可有可无的，之所以加上，更大情况出于美观的考虑，上面的实例写成如下样式也是合法的。

```
/**
   Animal，动物类
*/
public class Animal{
    ...
}
```

- **方法注释**

同样的，方法注释也就是针对类中方法的注释，它必须放在所描述的方法之前。而一般情况下，除开说明该方法的功能之外，我们经常使用如下标记来对方法进行注释。

标记	说明
`@param variable description`	用于介绍当前方法的参数，可以占据多行
`@return description`	用于描述当前方法的返回值，可以跨多行
`@throws class description`	用于表示该方法有可能抛出的异常

以下就是一个方法注释的实例：

```
/**
* 求两数之h
* @param num1 加数1
* @param num2 加数2
@ return 两数之和
*/
public int add(int num1, int num2){
    return num1 + num2;
}
```

- **字段注释**

字段注释顾名思义，也就是对于类中字段的说明，用于描述字段的含义，以下是一个字段注释的例子。

```
public class Cunyu{
    /**
    * 公众号
    */
      public String wePublic;
}
```

当然，如果你不喜欢把一个字段的注释分成多行的话，也可以写成以下格式。

```
public class Cunyu{
    /**公众号*/
      public String wePublic;
}
```

两种方式都是可以的，也没有优劣之分，可以根据自己的风格来选择。但是在 IntelliJ IDEA 等 IDE 中，如果对代码进行格式化，IDEA 会将第二种方式格式化成第一种方式，这一点需要注意。

2. **如何提取文档注释**

假设有以下一段代码，我们需要生成关于代码的文档说明。那么就可以使用 JDK 中所提供的 `javadoc` 命令来提取代码的文档注释。

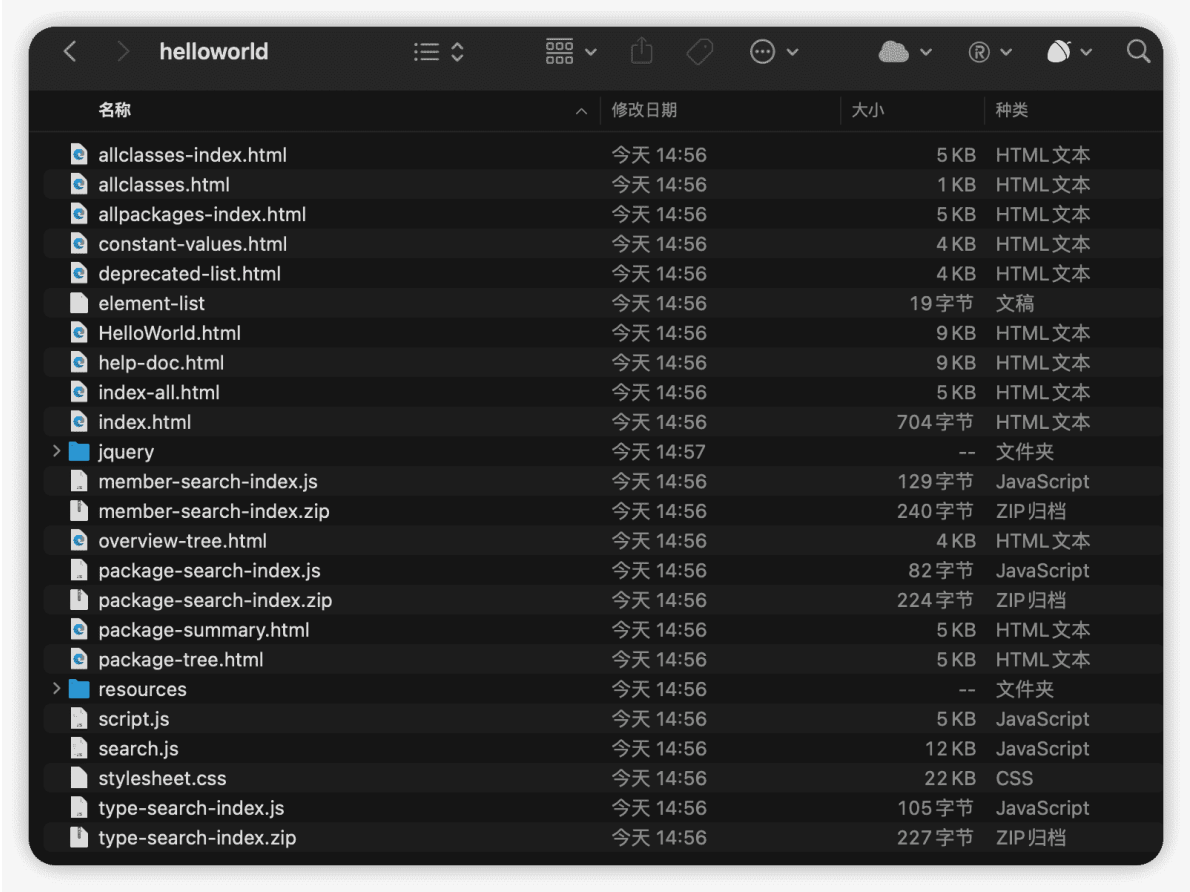
```
/**
* 第一个 Java 程序
* 这是初学者基本都会写的一个程序
* @author 村雨遥
* @version 1.0
*/
public class HelloWorld {
      /**
      * 主函数：程序入口
      * @param args 主函数参数列表
      */
     public static void main(String[] args){
         System.out.println("Hello World!");
      }
}
```

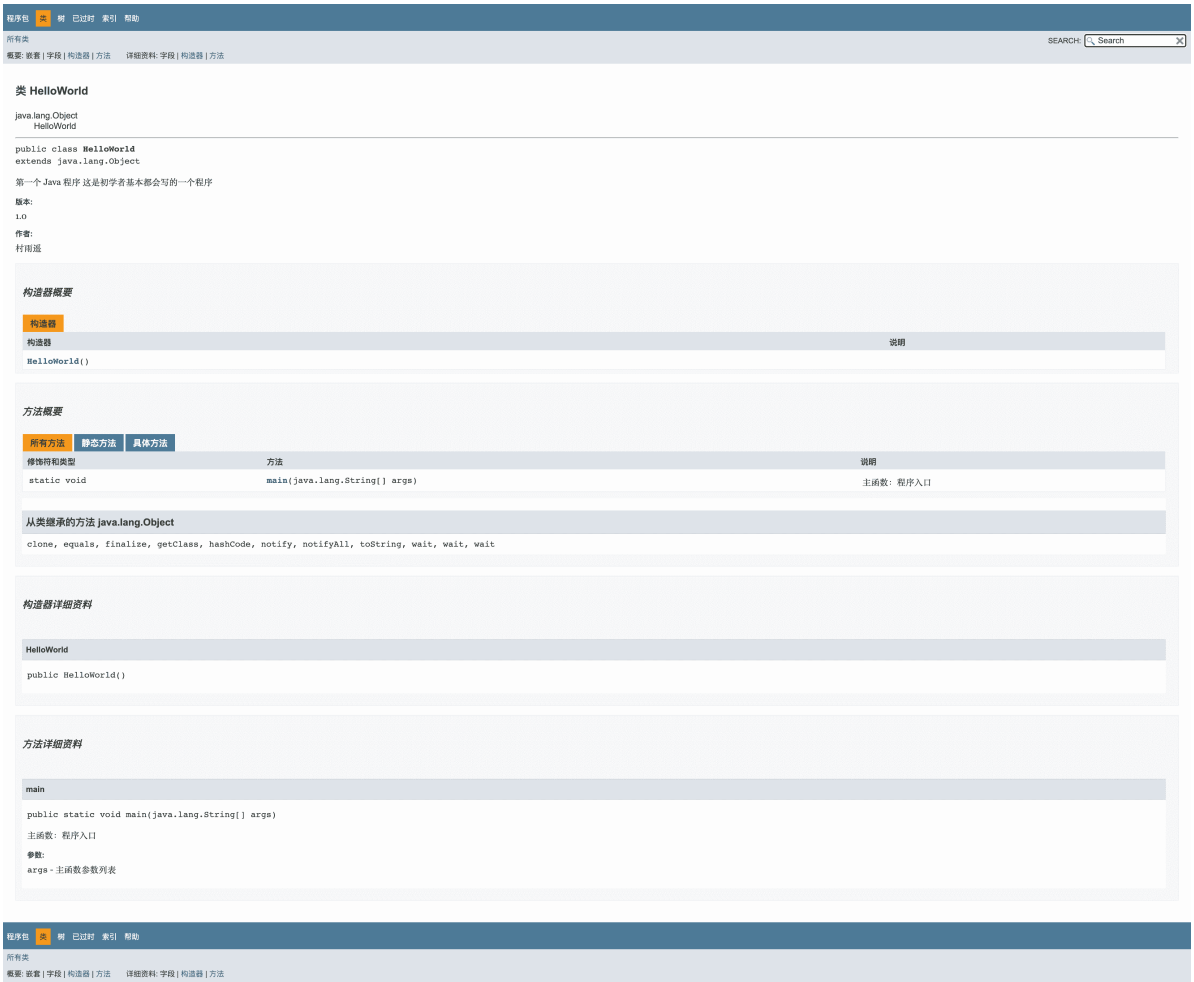
然后利用以下命令就可以生成我们的文档注释。

```
javadoc -d helloworld -author -version -encoding UTF-8 HelloWorld.java
```

以上命令的意思就是，对名为 `HelloWorld.java` 的提取其中的文档注释，并将输出的文件放在 `helloworld` 文件夹下，并且在文档中包含程序作者和版本，编码方式为 `UTF-8`。

生成的文件列表详情见下图，打开其中的 `index.html` 就可以查看提取的文档注释。





4. jar 文件的创建

其实关于这个，我在之前的文章也写过。不过我是利用 IntelliJ IDEA 来对进行代码的打包，如果感兴趣，可以点击下方传送门去看看。

> **如何利用 IntelliJ IDEA 创建 Java 入门应用**

不过那是借助工具来生成的，今天我们来看看如何利用 JDK 所提供的命令行工具，来创建一个能打印出 `Hello World!` 的 `jar` 包。

同样的，我们仍然是需要先准备一个能输出 `Hello World!` 的 Java 源代码，命名为 `HelloWorld.java`。

```
1   public class HelloWorld {
2       public static void main(String[] args){
3           System.out.println("Hello World!");
4       }
5   }
```

接着，利用 `javac` 命令对该文件进行编译，然后会生成 `HelloWorld.class` 字节码文件。

```
1   javac HelloWorld.java
```

然后，利用 `jar` 命令来对生成的字节码文件进行打包。

```
1   jar -cvf hello.jar HelloWorld.class
```

其中 `c` 表示创建一个新 `jar` 包，`v` 表示创建过程中打印创建过程中的信息，`f` 则表示对新生成的 `jar` 命名。

最后，利用以下命令来运行 `jar` 包。

```
1   java -jar hello.jar
```

不过并不会顺利出现我们想要的结果，此时会报错 `hello.jar` 中没有主清单属性。这是因为我们还没有在 `MENIFEST.MF` 文件中添加 `Main-Class` 属性。

用压缩软件打开刚创建的 `hello.jar`，里边除了 `HelloWorld.class` 文件之外，还会多一个 `META-INF` 文件夹，里边还有一个 `MENIFEST.MF` 文件，此时我们只需要用编辑器打开该文件，然后在文件中加入以下代码。（**记得添加之后要保证整个文件最后保留一行空行**）

```
1   Main-Class: HelloWorld
```

添加完成之后，再次运行 `java -jar hello.jar` ，就可以成功在控制台打印 `Hello World!` 了。

第九章 面向对象(其他)

1. 各变量

1.1 实例

```
1   package com.cunyu.demo
2
3   public class Demo {
4
5         private String name;   //成员变量、实例变量
6         private int age;   //成员变量、实例变量
7         private int ID;   //成员变量、实例变量
8
9         public static final String school = "卡塞尔学院";   //成员变量、静态变量(类变量)
10        public static String level = "SSS";      //成员变量、静态变量(类变量)
11
12        public int getAge() {
13              return age;
14        }
15
16        public int getId() {
17              return ID;
18        }
19
20        public String getName() {
21              return name;
22        }
23
24        public void setAge(int age) {
25              this.age = age;
26        }
27
28        public void setId(int ID) {
29              this.ID = ID;
30        }
31
32        public void setName(String name) {
33              this.name = name;
34        }
35
36        public void study(){
37                  String subject1 = "屠龙";   //局部变量
```

```
38                  String subject2 = "炼金术"; //局部变量
39                  System.out.println("学习科目： " + subject1 + "、" + subject2);
40              }
41
42          public static void main(String[] args) {
43              Demo demo = new Demo();
44              demo.setAge(23);
45              demo.setId(14000001);
46              demo.setName("楚子航");
47              System.out.println("ID: " + demo.getId() + "Age: " + demo.getAge() + "Name: " +
    demo.getName());
48              System.out.print("主修科目： ");
49              demo.study();
50              System.out.println("学院：" + Demo.school);
51              System.out.println("等级：" + Demo.level);
52          }
53      }
```

1.2 各变量联系与区别

1.2.1 成员变量

作用范围是整个类，相当于 C 语言中的全局变量，定义在方法体和语句块之外，一般定义在类的声明之下；成员变量包括实例变量和静态变量(类变量)。

1.2.2 实例变量

独立于与方法之外的变量，无 `static` 修饰，声明在一个类中，但在方法、构造方法和语句块之外，数值型变量默认值为 0，布尔型默认值为 `false`，引用类型默认值为 `null`。

1.2.3 静态变量(类变量)

独立于方法之外的变量，用 `static` 修饰，默认值与实例变量相似，一个类中只有一份，属于对象共有，存储在静态存储区，经常被声明为常量，调用一般是类名.静态变量名，也可以用对象名.静态变量名调用。

1.2.4 局部变量

类的方法中的变量，访问修饰符不能用于局部变量，声明在方法、构造方法或语句块中，在栈上分配，无默认值，必须经初始化。

1.2.5 成员变量 VS 局部变量

区别	成员变量	局部变量
类中位置不同	类中、方法外	方法中
初始化值不同	有初始值，无需初始化	无默认值，使用前需完成赋值
内存位置不同	堆内存	栈内存
生命周期不同	随对象创建而存在，随对象的消失而消失	随方法的调用而存在，随方法的运行结束而消失
作用域		所属大括号

2.抽象、接口、内部类

2.1 抽象方法

2.1.1 定义

所谓抽象方法，就是将公共的行为（方法）抽取到一个父类后，由于每个子类中执行的内容是不一致的，所以父类中无法确认具体的方法体，此时就可以将该方法定义为抽象方法。

2.1.2 格式

要定义一个抽象方法，那么需要遵循以下的格式，要注意，抽象方法一般都是没有定义

```
1    public abstract 返回值类型 方法名(参数列表);
```

以下是一个抽象方法定义的具体实例。

```
1    public abstract void welcome(String name);
```

2.2 抽象类

2.2.1 定义

知道了什么是抽象方法，那么抽象类就很简单了，如果我们发现一个类中存在着抽象方法，那么这个类一定要声明为抽象类。

2.2.2 格式

```
1    public abstract class 类名{
2
3    }
```

```
1    public abstract class Person{
2
3    }
```

2.2.3 注意

1. 抽象类不能实例化。
2. 抽象类中不一定有抽象方法，但有抽象方法的类一定是抽象类。
3. 抽象类中可以有构造方法。
4. 如果一个类的父类是抽象类，那么这个类要么是抽象类，要么重写父类中所有的抽象方法。

2.2.4 作用

在抽取共性时，如果无法确定方法体，此时就可以将该方法定义为抽象方法。然后强制让子类按照某种格式重写。

2.3 接口

2.3.1 定义

接口是一种规则，是对行为的抽象。

2.3.2 格式

```
public interface 接口名{

}
```

2.3.3 注意

1. 接口和抽象类一样，不能被实例化。
2. 接口和类之间是实现关系，通过关键字 `implements` 表示，实现格式为:

```
public class 类名 implements 接口名{

}
```

3. 如果一个类实现了某个接口，那么这个类要么是抽象类，要么就得重写接口中所有的抽象方法。
4. 不同于类单继承的特点，一个类是可以实现多个接口的。

2.3.4 成员的特点

1. **成员变量**

接口中，成员变量只能是常量，默认修饰符为 `public static final`。

2. **构造方法**

接口中不存在构造方法。

3. **成员方法**

`JDK 7` 之前，只能是抽象方法，其默认修饰符为 `public abstract`。`JDK 8` 时，能够定义有方法体的方法，`JDK 9` 中，有带来了可以定义私有方法的特性。

2.3.5 默认方法

自 `JDK 8` 之后，就允许在接口中定义默认方法，但是需要使用关键字 `default` 来修饰，从而解决接口升级的问题。

接口中的**默认方法**的定义格式为:

```
public default 返回值类型 方法名(参数列表){}
```

使用默认方法时，需要注意:

- 接口中的默认方法不是抽象方法，所以没有要求强制被重写。但如果要重写默认方法，重写时就需要去掉默认方法的关键字 `default`。
- 默认方法中的权限空置房 `public` 可以省略不写，但关键字 `default` 不能省略。
- 如果一个类实现了多个接口，而且着多个接口中还存在着重名的默认方法，那么该类就必须对重名的默认方法进行重写。

以下是一个存在重名默认方法必须重写的实例。

```
public interface InterOne{
      public abstract void method1();

      public default void method2(){
            System.out.println("默认方法 method2");
      }
}
```

```
public interface InterTwo{

      public default void method2(){
            System.out.println("默认方法 method2");
      }

      public abstract void method3();
}
```

```
 public class Main implements InterOne, InterTwo{
       @Override
       public void method1(){
             System.out.println("抽象方法 1");
       }

       @Override
       public void method3(){
             System.out.println("抽象方法 3");
       }

       @Override
       // 注意重写需要去掉关键字 default
       public void method2(){
             System.out.println("默认方法 2");
       }
 }
```

2.3.6 静态方法

除了默认方法，`JDK 8` 以后还可以在接口中定义静态方法，此时需要用关键字 `static` 修饰。

接口中 **静态方法** 的定义格式为：

```
public static 返回值类型 方法名(参数列表){}
```

在接口中使用静态方法时，需要注意：

- 静态方法只能通过接口名调用，不能通过实现接口的类名或者对象名来调用。
- 权限控制符 `public` 可以省略，但关键字 `static` 不能省略。

以下是一个在接口中定义静态方法和调用接口中静态方法的实例。

```
public interface Inter{

      public static void method(){
            System.out.println("静态方法 method");
      }
}
```

```
public class Main implements Inter{
      public static void main(String[] args){
            // 调用接口中的静态方法
            Inter.method();
      }
}
```

2.3.7 私有方法

除了上述的静态方法和默认方法之外，`JDK 9` 中又引入接口中定义私有方法的特性。

其中，接口中定义私有方法的格式如下：

```
private 返回值类型 方法名(参数列表){}
```

```
private static 返回值类型 方法名(参数列表){}
```

两者的区别在于：静态的私有方法（带关键字 `static`）是为接口中的静态方法服务，而非静态的私有方法（不带关键字 `static`） 则是为接口中的默认方法服务。

以下是在接口中定义私有方法的实例：

1. **私有方法**

```
public interface Inter{

    private void method(){
            System.out.println("私有方法 method");
     }

      public default void method1(){
            method();
            System.out.println("默认方法 method1");
     }
}
```

2. **静态私有方法**

```
public interface Inter{

    private static void method(){
            System.out.println("私有方法 method");
     }

      public static void method1(){
            method();
            System.out.println("静态方法 method1");
     }
}
```

2.3.8 接口和类之间的关系

1. **类和类的关系**

类与类之间只能存在继承关系，且只限于单继承，不能多继承，但是可以多层继承。

2. **类和接口的关系**

类和接口之间是实现关系，既可以单实现，也可以多实现，还能在继承一个类的同时实现多个接口。

3. **接口和接口的关系**

接口之间是继承关系，但不同于类和类之间的关系，接口之间既可以单继承，也可以多继承。

2.4 内部类

2.4.1 定义

顾名思义，所谓内部类就是定义在类中的类。比如说在 `A` 类的内部定义了一个 `B` 类，那么我们就说 `B` 是内部类。其中 `B` 类表示的是 `A` 类的一部分，而且 `B` 类单独存在时没有任何意义。

```
public class Outer{
      public class Inner{

      }
}
```

2.4.2 特点

如果一个类中定义了一个内部类，那么这个内部类就能够直接访问外部类的所有成员，包括私有成员。

但是，如果一个外部类要访问内部类的成员，那么此时就必须创建对象。

```
 public class Outer{
       private String name;
       private String address;

       public class Inner{
             String id;
             String age;

             public void show(){
                   // 内部类可以直接访问外部类的所有成员
                   System.out.println(id + " " + age + " " + name + " " + address)
             }
       }
 }
```

2.4.3 分类

`Java` 中，内部类主要分为以下四种：

1. **成员内部类**
2. **静态内部类**
3. **局部内部类**
4. **匿名内部类**

成员内部类

写在成员位置，属于外部类的成员，其中成员内部类也能够被常用的修饰符 `public、private、protected、static` 等所修饰。

```
public class Outer{
      private String name;
      private String address;

      public class Inner{
            String id;
            String age;
      }
}
```

要想获取成员内部类对象，可以通过以下两种方式：

1. 在外部类中编写方法，然后对外提供内部类的对象。

```
public class Outer{
      private String name;
      private String address;

      public class Inner{
            String id;
            String age;
      }

      public Inner getInnerInstance(){
            return new Inner();
      }
}
```

2. 直接创建，创建格式为：`外部类名.内部类名 对象名 = 外部类对象.内部类对象`。

```
public class Main{
     public static void main(String[] args){
          Outer.Inner inner = new Outer().new Inner();
     }
}
```

静态内部类1

静态内部类就是用关键字 `static` 修饰的内部类。静态内只能访问外部类中的静态变量和静态方法，如果要访问非静态的变量和方法就需要创建对象。

```
public class Outer{
      private String name;
      private String address;
      static String birth = "20220222";

      static class Inner{
            String id;
            String age;

            public static void show(){
                  System.out.println("静态方法" + birth);
            }

            public void display(){
                  System.out.println("非静态方法");
            }

      }
}
```

创建静态内部类对象的格式为：`外部类名.内部类对象名 = new 外部类名.内部类名();`

调用非静态方法的格式：先创建对象，然后用对象调用。

调用静态方法的格式：`外部类名.内部类名.方法名();`

```
public class Main{
      public static void main(String[] args){
            Outer.Inner inner = new Outer().new Inner();
            // 调用非静态方法
            inner.display();
            // 调用静态方法
            Outer.Inner.show();
      }
}
```

局部内部类

如果一个类被定义在方法中，那么这个类就叫做局部内部类。外界是无法直接使用局部内部类，需要在方法内部创建对象并使用。同时，这个内部类既可以访问外部类的成员，也可以访问方法中的局部变量。

```
 public class Outer{
       public String id;
       public void show(){
             class Inner{
                   String name;

                   public void show(){
                         System.out.println("局部类内部方法");
                         System.out.println(id);
                         System.out.println(name);
                   }
             }
             Inner inner = new Inner();
             System.out.println(inner.name);
             inner.show();
       }
 }
```

匿名内部类

匿名内部类就是没有名字的内部类，也正因为名字，所以匿名内部类只能使用一次。而且使用匿名内部类还有个前提条件：必须继承一个父类或者实现一个接口。

```
new 类名/接口名(){
      重写方法;
}
```

```
public interface Inner{
      public abstract void method();
}
```

```
 public class Main{
       public static void main(String[] args){
             new Inner(){
                   @Override
                   public void method(){
                         System.out.println("方法重写");
                   }
             }
       }
 }
```

3.static和final

3.1 前言

学习了面向对象的一些知识点后，在看别人的代码中经常会看到 `static` 和 `final` 两个关键字，那你知道它俩都是做什么用的么，使用的时候需要注意啥呢?

今天的内容就来一起了解下， `static` 和 `final` 关键字的使用。

3.2 static

静态的意思，是 `Java` 中常用的修饰符，既可以用它来修饰成员变量，也可以用它来修饰成员方法。

3.2.1 静态变量

静态变量就是用 `static` 来修饰的成员变量，最重要的特点就是一个类中的静态变量被类中的所有对象所共享。它不属于任何一个对象，而是属于类。其生命周期同类一样，共生共存，是优先于对象的。

和成员变量不同的是，成员变量只能通过新建对象，然后用对象名来进行调用。而静态变量不仅仅可以用这种方式，还可以直接通过类名来进行调用，而这也是更为推荐的方式。

```
public class Hero{
      /** 成员变量 */
      private String name;

      /** 静态变量 */
      pubic static String game;

      // 各种 setter 和 getter
}
```

1. **成员变量调用**

```
public class Main{
      public static void main(String[] args){
            Hero hero = new Hero();
            hero.setName("赵怀真");
      }
}
```

2. **静态变量调用**

```
 public class Main{
       public static void main(String[] args){
            // 第一种方式
             Hero hero = new Hero();
             hero.setGame("王者荣耀");

             // 第二种方式，更为推荐
             Hero.setGame("王者荣耀");
       }
 }
```

3.2.2 静态方法

静态方法就是类中用 `static` 来修饰的成员方法，一般用来修饰公共的工具类或者测试类。

同样的，静态方法既可以用通过新建对象，然后用对象名来调用的方式，也可以直接通过类名来调用的方式，这也是更为推荐的方式。

```
public class Util{
      private Util(){};

      pubilc static void attack(){
            System.out.println("攻击");
      }
}
```

```
public class Main{
      public static void main(String[] args){
            Util.attack();
      }
}
```

3.2.3 注意

使用 `static` 修饰方法或变量后，需要注意以下的小细节。

- 一个静态方法中**只能**访问静态变量和其他的静态方法。而不能访问非静态的变量和方法。
- 但一个非静态方法一方面既可以访问静态变量，也可以访问非静态变量；另一方面，也既可以访问静态方法，也可以访问非静态方法。
- 不同于成员方法，静态方式中是不存在 `this` 关键字的。

3.2.4 静态代码块

在代码中用 `static{}` 包裹起来的代码叫做静态代码块，它会随着类的加载而加载，而且会自动触发，只执行一次，一般用来对一些数据初始化。

```
public class Main{
      static String password;

      static{
            password = "123456";
      }
}
```

3.3 final

3.3.1 修饰变量

用 `final` 修饰的变量叫做常量，说明它只能被赋值一次。

实际开发中，一般用常量来作为系统的配置信息，一方面既方便维护，另一方面又可以提高代码可读性。

对常量进行命名时，一般遵循以下的规范：

- 如果是单个单词，那么将它全部大写即可。
- 如果是多个单词，那么将每个单词都大写，并且单词之间用下划线 `_` 隔开。

此外，对于修饰的变量的类型不同，含义也是不一样的。

如果修饰的变量是一个基本类型，那么表示的是该变量存储的**数据值**不可改变。而如果修饰的变量是一个引用类型，则表示该变量存储的**地址值**不能改变，但是对象内部是可以发生变化的。

```
public class Comic{
      private String name;
      private String type;
      // setter、getter、Constructor 省略
}
```

```
 public class Main{
       public static void main(String[] args){
             final int SIZE = 5;

             // 此时会报错
             // SIZE = 10;

             final Comic comic = new Comic("灌篮高手", "运动");
             // 不会报错，因为地址值未变，变的是对象内部
             comic.setName("海贼王");
             comic.setType("冒险");
       }
 }
```

3.3.2 修饰方法

说明该方法是最终方法，不能再被重写。

```
 public class Person{
       public final void walk(){
             System.out.println("行走")
       }
 }

 public class Student extends Person{
       // 会报错，因为 final 修饰的方法不能再被重写
       @Override
       public void walk(){
             System.out.println("行走")
       }
 }
```

3.3.3 修饰类

说明这个类时最终类，不能够再被继承。也就是说，如果一个类被 `final` 所修饰，那么这个类不能作为其他任意类的福来。如果试图对一个用 `final` 修饰的类进行继承，则在编译期间可能会发生错误。

```
public final class Person{
      private String name;
}

// 继承用 final 修饰的类，此时会报错
class Student extends Person{
      ......
}
```

第十章 枚举类

Java 枚举是一个特殊的类，一般表示一组常量，比如一年的 4 个季节，一年的 12 个月份，一个星期的 7 天，方向有东南西北等。

Java 枚举类使用 enum 关键字来定义，各个常量使用逗号 , 来分割。

例如定义一个颜色的枚举类。

```
enum Color
{
      RED, GREEN, BLUE;
}
```

以上枚举类 Color 颜色常量有 RED, GREEN, BLUE，分别表示红色，绿色，蓝色。

```
enum Color{
   RED, GREEN, BLUE;
}

public class Test{
   public** static void main(String[] args){
      Color c1 = Color.RED;
      System.out.println(c1);
   }
}
```

1. 内部类中使用枚举

枚举类也可以声明在内部类中：

```
public class Test{
   enum Color{
      RED, GREEN, BLUE;
   }

   public static void main(String[] args){
      Color c1 = Color.RED;
      System.out.println(c1);
   }
}
```

每个枚举都是通过 Class 在内部实现的，且所有的枚举值都是 public static final 的。

以上的枚举类 Color 转化在内部类实现：

```
class Color
{
      public static final Color RED = new Color();
      public static final Color BLUE = new Color();
      public static final Color GREEN = new Color();
}
```

2. 迭代枚举元素

可以使用 for 语句来迭代枚举元素：

```
enum Color{
      RED, GREEN, BLUE;
}
public class MyClass {
      public static void main(String[] args) {
            for(Color myVar : Color.values()) {
                  System.out.println(myVar);
            }
      }
}
```

3. 在 switch 中使用枚举类

枚举类常应用于 switch 语句中：

```
enum Color {
   RED, GREEN, BLUE;
}
public class MyClass {
      public static void main(String[] args) {
            Color myVar = Color.BLUE;
            switch (myVar) {
                  case RED:
                        System.out.println("红色");
                        break;
                  case GREEN:
                        System.out.println("绿色");
                        break;
                  case BLUE:
                        System.out.println("蓝色");
                        break;
            }
      }
}
```

4. values(), ordinal() 和 valueOf() 方法

enum 定义的枚举类默认继承了 java.lang.Enum 类，并实现了 java.lang.Serializable 和 java.lang.Comparable 两个接口。

values(), ordinal() 和 valueOf() 方法位于 java.lang.Enum 类中：

- values() 返回枚举类中所有的值。
- ordinal()方法可以找到每个枚举常量的索引，就像数组索引一样。
- valueOf()方法返回指定字符串值的枚举常量。

```
enum Color {
   RED, GREEN, BLUE;
}
public class Test{
      public static void main(String[] args){
            // 调用 values()
            Color[] arr = Color.values();
```

```
8
9               // 迭代枚举
10              for (Color col : arr){
11                    // 查看索引
12                    System.out.println(col + " at index " + col.ordinal());
13              }
14
15              // 使用 valueOf() 返回枚举常量，不存在的会报错 IllegalArgumentException
16              System.out.println(Color.valueOf("RED"));
17              // System.out.println(Color.valueOf("WHITE"));
18        }
19  }
```

5. 枚举类成员

枚举跟普通类一样可以用自己的变量、方法和构造函数，构造函数只能使用 private 访问修饰符，所以外部无法调用。

枚举既可以包含具体方法，也可以包含抽象方法。 如果枚举类具有抽象方法，则枚举类的每个实例都必须实现它。

```
1   enum Color{
2         RED, GREEN, BLUE;
3
4         // 构造函数
5         private Color(){
6               System.out.println("Constructor called for : " + this.toString());
7         }
8
9         public void colorInfo(){
10              System.out.println("Universal Color");
11        }
12  }
13
14  public class Test{
15        // 输出
16        public static void main(String[] args){
17              Color c1 = Color.RED;
18              System.out.println(c1);
19              c1.colorInfo();
20        }
21  }
```

执行以上代码输出结果为：

```
1   Constructor called for : RED
2   Constructor called for : GREEN
3   Constructor called for : BLUE
4   RED
5   Universal Color
```

第十一章 注解

1. 注解简介

所谓注解，其实就像一种拥有特定作用的注释，自 JDK1.5 及之后版本所引入的特性，它是放在 Java 源码的类、方法、字段、参数前的一种用作标注的“元数据”，与类、接口、枚举处于同一个层次中。

通过其作用的不同，我们常常将注解分为如下 3 类：

1. **编写文档**：通过代码中标识的注解生成对应文档（即类似于 Java doc 的文档）；
2. **代码分析**：通过代码中标识的注解对代码进行分析（使用反射）；
3. **编译检查**：通过代码中标识的注解让编译器能实现基本的编译检查（`@Override`）；

2. 常用的预定义注解

`@Override`

一般是用在方法上，表示重写该父类的方法，比如我们使用最多的 `toString()` 方法，它是 `Object` 类的一个方法，而我们的写的类都是继承自 `Object` 类，所以我们自定义的所有类都是有 `toString()` 方法的。但是如果我们自定义类中的方法在父类中没有，则不能使用该注解，否则会导致无法编译通过。

```
1   package com.cunyu;
2
3   /**
4    * Created with IntelliJ IDEA.
5    *
6    * @author : cunyu
7    * @version : 1.0
8    * @email : 747731461@qq.com
9    * @website : https://cunyu1943.github.io
10   * @date : 2021/6/20 10:04
11   * @project : JavaWeb
12   * @package : com.cunyu
13   * @className : OverrideTest
14   * @description :
15   */
16
17  public class OverrideTest {
18        private Integer id;
19        private String name;
20
21        public OverrideTest(Integer id, String name) {
22              this.id = id;
23              this.name = name;
24        }
25
26        @Override
27        public String toString() {
28              final StringBuffer sb = new StringBuffer("OverrideTest{");
29              sb.append("id=").append(id);
30              sb.append(", name='").append(name).append('\'');
31              sb.append('}');
32              return sb.toString();
33        }
34
35        public static void main(String[] args) {
36              Integer id = 101;
37              String name = "村雨遥";
38              OverrideTest overrideTest = new OverrideTest(id, name);
```

```
39
40              System.out.println(overrideTest);
41          }
42      }
```

@Deprecated

一般用在方法之前，表示该方法已经过期，不建议再继续使用（但是仍然有效，只不过可能有更新的版本，推荐使用更新的版本）。

```
1   package com.cunyu;
2
3   /**
4    * Created with IntelliJ IDEA.
5    *
6    * @author : cunyu
7    * @version : 1.0
8    * @email : 747731461@qq.com
9    * @website : https://cunyu1943.github.io
10   * @公众号 : 村雨遥
11   * @date : 2021/6/20 10:07
12   * @project : JavaWeb
13   * @package : com.cunyu
14   * @className : DeprecateTest
15   * @description :
16   */
17
18  public class DeprecateTest {
19        @Deprecated
20        public static void sayHello() {
21              System.out.println("Hello World!");
22        }
23
24        public static void newSayHello() {
25              System.out.println("Hello, Welcome to Java !");
26        }
27
28        public static void main(String[] args) {
29              sayHello();
30              newSayHello();
31        }
32  }
```

@SuppressWarnings

表示忽略警告信息，常用的值以及含义如下表：

值	描述
deprecation	使用了不赞成使用的类或方法时的警告
unchecked	使用了未经检查的转换时的警告
fallthrough	当 switch 程序块直接通往下一种情况而没有 break 时的警告
path	在类路径、源文件路径等中有不存在的路径时的警告
serial	当在可序列化的类上缺少 serialVersionUID 定义时的警告
finally	任何 finally 子句不能正常完成时的警告
rawtypes	泛型类型未指明
unused	引用定义了，但是没有被使用
all	关闭以上所有情况的警告

```java
package com.cunyu;

import java.util.ArrayList;
import java.util.List;

/**
 * Created with IntelliJ IDEA.
 *
 * @author : cunyu
 * @version : 1.0
 * @email : 747731461@qq.com
 * @website : https://cunyu1943.github.io
 * @公众号 : 村雨遥
 * @date : 2021/6/20 10:07
 * @project : JavaWeb
 * @package : com.cunyu
 * @className : SuppressWarningsTest
 * @description :
 */

public class SuppressWarningsTest {
      @SuppressWarnings("unchecked")
      public static void main(String[] args) {
            String item = "村雨遥";
            @SuppressWarnings("rawtypes")
            List items = new ArrayList();
            items.add(item);

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

3. 自定义注解

3.1 格式

我们可以使用 `@interface` 来自定义注解，其格式如下：

```
public @interface AnnotationName{
      // 属性列表
      ……
}
```

一个简单的示例如下，其中 `AnnoDemo` 代表着我们自定义注解的名称，而 `name()`、`age()`、`score()` 则分别表示自定义注解的三个属性，而且我们利用关键字 `default` 对每个属性都赋予了默认值。

```
public @interface AnnoDemo{
      String name() default "村雨遥";
      int age() default 20;
      float score() default 60.0f;
}
```

3.2 原理

注解本质上相当于一个接口，它默认继承自 `java.lang.annotation.Annotation`。

```
public interface AnnotationName extends java.lang.annotation.Annotation{}
```

3.3 参数

注解的参数类似于无参的方法，通常我们推荐用 `default` 来设定一个默认值，对于方法的基本要求通常有如下几点：

1. 方法的返回值类型不可以是 `void`；
2. 如果定义了方法，那么在使用时需要给方法进行赋值，赋值的规则如下：
    1. 若定义方法时，使用了关键字 `default` 对方法赋予了默认初始值，那么在使用注解时，可以不用对方法进行再次赋值；
    2. 若只有一个方法需要赋值，且方法名为 `value`，那么此时 `value` 可以省略，直接定义值即可；
    3. 数组赋值时，值需要用大括号 `{}` 包裹，若数组中只有一个值，那么此时 `{}` 可以省略；

```
public @interface AnnoDemo{
      String name() default "村雨遥";
      int age() default 20;
      float score() default 60.0f;
}
```

如上述例子中，`name()`、`age()`、`score()` 就是我们自定义注解的参数。而当我们要是用该注解时，则通过如下方式来对参数进行赋值。

```
@AnnoDemo(name = "村雨遥", age = 26, score = 95.0f)
public class Demo{
      ……
}
```

4. 元注解

4.1 定义

所谓元注解(`meta annotation`)，就是可以用来修饰其他注解的注解。

4.2 常用的元注解

1. `@Target`

描述注解所修饰的对象范围，其取值主要有如下几种：

值	说明
`ElementType.TYPE`	表示可以作用于类或接口
`ElementType.FIELD`	表示可以作用于成员变量
`ElementType.METHOD`	表示可以作用于方法
`ElementType.CONSTRUCTOR`	表示可以作用于构造方法
`ElementType.PARAMETER`	表示可以作用于方法的参数

```
@Target(ElementType.TYPE)
public @interface AnnoDemo{
      String name() default "村雨遥";
      int age() default 20;
      float score() default 60.0f;
}
```

2. `@Retention`

用于约束注解的生命周期，其取值如下：

值	说明
`RetentionPolicy.SOURCE`	表示在源代码文件中有效，注解将被编译器丢弃（注解信息仅保留在源码中，源码经编译后注解信息丢失，不再保留到字节码文件中）
`RetentionPolicy.CLASS`	表示在字节码文件中有效，注解在字节码文件中可用，但会被 JVM 丢弃
`RetentionPolicy.RUNTIME`	表示在运行时有效，此时可以通过反射机制来读取注解的信息

```
@Target(ElementType.TYPE)
@Retention(RetentionPoicy.RUNTIME)
public @interface AnnoDemo{
      String name() default "村雨遥";
      int age() default 20;
      float score() default 60.0f;
}
```

3. `@Documented`

描述其他类型的注解是否被抽取到 API 文档中。

```
@Target(ElementType.TYPE)
@Retention(RetentionPoicy.RUNTIME)
@Documented
public @interface AnnoDemo{
      String name() default "村雨遥";
      int age() default 20;
      float score() default 60.0f;
}
```

4. `@Inherited`

这是一个标记注解，描述某个注解能够被子类继承，但是该元注解只适合已经配置了 `@Target(ElementType.TYPE)` 类型的自定义注解，而且仅针对于类的继承，而对于接口的继承则无效。

```
@Inherited
public @interface AnnoDemo{
      String name() default "村雨遥";
      int age() default 20;
      float score() default 60.0f;
}
```

5. `@Repeatable`

该注解是从 JDK1.8 新引入的元注解，表示在同一位置能够重复相同的注解。在没有该注解之前，我们一般是无法在同一类型上使用相同注解的，但引入该注解后，我们就可以在同一类型上使用相同注解。

```
 @Target(ElementType.TYPE)
 @Repeatable(AnnoDemos.class)
 public @interface AnnoDemo{
       String name() default "村雨遥";
       int age() default 20;
       float score() default 60.0f;
 }

 public @interface AnnoDemos{
     AnnoDemo[] value();
 }
```

利用 `@Repeatable` 配置自定义注解之后，我们就可以在某个类型声明处添加多个我们自定义的注解了。

```
@AnnoDemo(name = "村雨遥", age = 26, score = 88.0f)
@AnnoDemo(name = "晓瑜", age = 27, score = 90.0f)
public class Student{
      ……
}
```

5. 总结

总结上述的知识点，我们将自定义注解的过程归纳为如下 3 步。

1. **定义一个注解**

```
public @interface AnnoDemo{
}
```

2. **添加参数并设置默认值**

```
public @interface AnnoDemo{
      String name() default "村雨遥";
      int age() default 20;
      float score() default 60.0f;
}
```

3. **利用元注解来配置我们的自定义注解**

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface AnnoDemo{
      String name() default "村雨遥";
      int age() default 20;
      float score() default 60.0f;
}
```

在实际应用过程中，利用元注解配置自定义注解时，必须设置 `@Target` 和 `@Retention` 两个元注解，而且 `@Retention` 的值通常是设置为 `RetentionPolicy.RUNTIME`。

第十二章 异常

异常是程序中的一些错误，但并不是所有的错误都是异常，并且错误有时候是可以避免的。

比如说，你的代码少了一个分号，那么运行出来结果是提示是错误 java.lang.Error；如果你用System.out.println(11/0)，那么你是因为你用0做了除数，会抛出 java.lang.ArithmeticException 的异常。

异常发生的原因有很多，通常包含以下几大类：

- 用户输入了非法数据。
- 要打开的文件不存在。
- 网络通信时连接中断，或者JVM内存溢出。

这些异常有的是因为用户错误引起，有的是程序错误引起的，还有其它一些是因为物理错误引起的。-

要理解Java异常处理是如何工作的，你需要掌握以下三种类型的异常：

- **检查性异常：**最具代表的检查性异常是用户错误或问题引起的异常，这是程序员无法预见的。例如要打开一个不存在文件时，一个异常就发生了，这些异常在编译时不能被简单地忽略。
- **运行时异常：** 运行时异常是可能被程序员避免的异常。与检查性异常相反，运行时异常可以在编译时被忽略。
- **错误：** 错误不是异常，而是脱离程序员控制的问题。错误在代码中通常被忽略。例如，当栈溢出时，一个错误就发生了，它们在编译也检查不到的。

1. Exception 类的层次

所有的异常类是从 java.lang.Exception 类继承的子类。

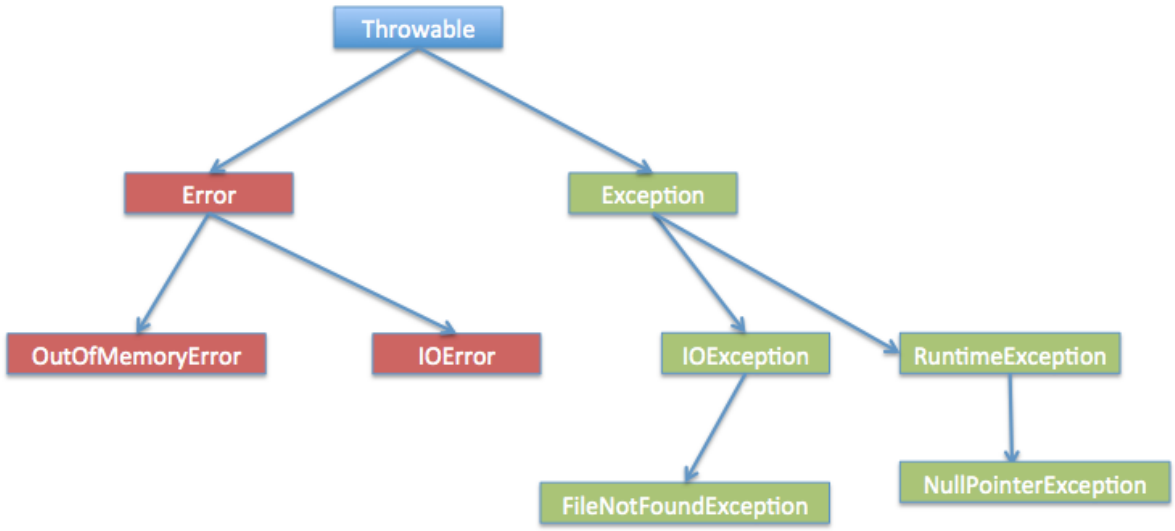
Exception 类是 Throwable 类的子类。除了Exception类外，Throwable还有一个子类Error 。

Java 程序通常不捕获错误。错误一般发生在严重故障时，它们在Java程序处理的范畴之外。

Error 用来指示运行时环境发生的错误。

例如，JVM 内存溢出。一般地，程序不会从错误中恢复。

异常类有两个主要的子类：IOException 类和 RuntimeException 类。



在 Java 内置类中(接下来会说明)，有大部分常用检查性和非检查性异常。

2. Java 内置异常类

Java 语言定义了一些异常类在 java.lang 标准包中。

标准运行时异常类的子类是最常见的异常类。由于 java.lang 包是默认加载到所有的 Java 程序的，所以大部分从运行时异常类继承而来的异常都可以直接使用。

Java 根据各个类库也定义了一些其他的异常，下面的表中列出了 Java 的非检查性异常。

异常	描述
ArithmeticException	当出现异常的运算条件时，抛出此异常。例如，一个整数"除以零"时，抛出此类的一个实例。
ArrayIndexOutOfBoundsException	用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引。
ArrayStoreException	试图将错误类型的对象存储到一个对象数组时抛出的异常。
ClassCastException	当试图将对象强制转换为不是实例的子类时，抛出该异常。
IllegalArgumentException	抛出的异常表明向方法传递了一个不合法或不正确的参数。
IllegalMonitorStateException	抛出的异常表明某一线程已经试图等待对象的监视器，或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。
IllegalStateException	在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环境或 Java 应用程序没有处于请求操作所要求的适当状态下。
IllegalThreadStateException	线程没有处于请求操作所要求的适当状态时抛出的异常。
IndexOutOfBoundsException	指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
NegativeArraySizeException	如果应用程序试图创建大小为负的数组，则抛出该异常。
NullPointerException	当应用程序试图在需要对象的地方使用 `null` 时，抛出该异常
NumberFormatException	当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。
SecurityException	由安全管理器抛出的异常，指示存在安全侵犯。
StringIndexOutOfBoundsException	此异常由 `String` 方法抛出，指示索引或者为负，或者超出字符串的大小。
UnsupportedOperationException	当不支持请求的操作时，抛出该异常。

下面的表中列出了 Java 定义在 java.lang 包中的检查性异常类。

异常	描述
ClassNotFoundException	应用程序试图加载类时，找不到相应的类，抛出该异常。
CloneNotSupportedException	当调用 `Object` 类中的 `clone` 方法克隆对象，但该对象的类无法实现 `Cloneable` 接口时，抛出该异常。
IllegalAccessException	拒绝访问一个类的时候，抛出该异常。
InstantiationException	当试图使用 `Class` 类中的 `newInstance` 方法创建一个类的实例，而指定的类对象因为是一个接口或是一个抽象类而无法实例化时，抛出该异常。
InterruptedException	一个线程被另一个线程中断，抛出该异常。
NoSuchFieldException	请求的变量不存在
NoSuchMethodException	请求的方法不存在

3. 异常方法

下面的列表是 Throwable 类的主要方法:

序号	方法及说明
1	**public String getMessage()** 返回关于发生的异常的详细信息。这个消息在Throwable 类的构造函数中初始化了。
2	**public Throwable getCause()** 返回一个 Throwable 对象代表异常原因。
3	**public String toString()** 返回此 Throwable 的简短描述。
4	**public void printStackTrace()** 将此 Throwable 及其回溯打印到标准错误流。。
5	**public StackTraceElement [] getStackTrace()** 返回一个包含堆栈层次的数组。下标为0的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底。
6	**public Throwable fillInStackTrace()** 用当前的调用栈层次填充Throwable 对象栈层次，添加到栈层次任何先前信息中。

4. 捕获异常

使用 try 和 catch 关键字可以捕获异常。try/catch 代码块放在异常可能发生的地方。

try/catch代码块中的代码称为保护代码，使用 try/catch 的语法如下：

```
try
{
    // 程序代码
}catch(ExceptionName e)
{
    //Catch 块
}
```

Catch 语句包含要捕获异常类型的声明。当保护代码块中发生一个异常时，try 后面的 catch 块就会被检查。

如果发生的异常包含在 catch 块中，异常会被传递到该 catch 块，这和传递一个参数到方法是一样。

5. 多重捕获块

一个 try 代码块后面跟随多个 catch 代码块的情况就叫多重捕获。

```
try{
      // 程序代码
}catch(异常类型1 异常的变量名1){
      // 程序代码
}catch(异常类型2 异常的变量名2){
      // 程序代码
}catch(异常类型3 异常的变量名3){
      // 程序代码
}
```

上面的代码段包含了 3 个 catch块。

可以在 try 语句后面添加任意数量的 catch 块。

如果保护代码中发生异常，异常被抛给第一个 catch 块。

如果抛出异常的数据类型与 ExceptionType1 匹配，它在这里就会被捕获。

如果不匹配，它会被传递给第二个 catch 块。

如此，直到异常被捕获或者通过所有的 catch 块。

6. throws/throw 关键字

在Java中， **throw** 和 **throws** 关键字是用于处理异常的。

throw 关键字用于在代码中抛出异常，而 **throws** 关键字用于在方法声明中指定可能会抛出的异常类型。

6.1 throw 关键字

throw 关键字用于在当前方法中抛出一个异常。

通常情况下，当代码执行到某个条件下无法继续正常执行时，可以使用 **throw** 关键字抛出异常，以告知调用者当前代码的执行状态。

例如，下面的代码中，在方法中判断 num 是否小于 0，如果是，则抛出一个 IllegalArgumentException 异常。

```
public void checkNumber(int num) {
      if (num < 0) {
            throw new IllegalArgumentException("Number must be positive");
      }
}
```

6.2 throws 关键字

throws 关键字用于在方法声明中指定该方法可能抛出的异常。当方法内部抛出指定类型的异常时，该异常会被传递给调用该方法的代码，并在该代码中处理异常。

例如，下面的代码中，当 readFile 方法内部发生 IOException 异常时，会将该异常传递给调用该方法的代码。在调用该方法的代码中，必须捕获或声明处理 IOException 异常。

```
public void readFile(String filePath) throws IOException {
      BufferedReader reader = new BufferedReader(new FileReader(filePath));
      String line = reader.readLine();
      while (line != null) {
            System.out.println(line);
            line = reader.readLine();
      }
      reader.close();
}
```

一个方法可以声明抛出多个异常，多个异常之间用逗号隔开。

例如，下面的方法声明抛出 RemoteException 和 InsufficientFundsException：

```
import java.io.*;
public class className {
      public void withdraw(double amount) throws RemoteException,
InsufficientFundsException {
            // Method implementation
      }
      //Remainder of class definition
}
```

7. finally关键字

finally 关键字用来创建在 try 代码块后面执行的代码块。

无论是否发生异常，finally 代码块中的代码总会被执行。

在 finally 代码块中，可以运行清理类型等收尾善后性质的语句。

finally 代码块出现在 catch 代码块最后，语法如下：

```
try{
        // 程序代码
}catch(异常类型1 异常的变量名1){
        // 程序代码
}catch(异常类型2 异常的变量名2){
        // 程序代码
}finally{
        // 程序代码
}
```

注意下面事项：

- catch 不能独立于 try 存在。
- 在 try/catch 后面添加 finally 块并非强制性要求的。
- try 代码后不能既没 catch 块也没 finally 块。
- try, catch, finally 块之间不能添加任何代码。

8. try-with-resources

JDK7 之后，Java 新增的 **try-with-resource** 语法糖来打开资源，并且可以在语句执行完毕后确保每个资源都被自动关闭 。

try-with-resources 是一种异常处理机制，它可以简化资源管理代码的编写。

JDK7 之前所有被打开的系统资源，比如流、文件或者 Socket 连接等，都需要被开发者手动关闭，否则将会造成资源泄露。

```
try (resource declaration) {
  // 使用的资源
} catch (ExceptionType e1) {
  // 异常块
}
```

以上的语法中 try 用于声明和实例化资源，catch 用于处理关闭资源时可能引发的所有异常。

注意： try-with-resources 语句关闭所有实现 AutoCloseable 接口的资源。

```
import java.io.*;

public class RunoobTest {
      public static void main(String[] args) {
            String line;
            try(BufferedReader br = new BufferedReader(**new** FileReader("test.txt"))) {
                  while((line = br.readLine()) != null) {
                        System.out.println("Line =>"+line);
                  }
            } catch (IOException e) {
                  System.out.println("IOException in try block =>" + e.getMessage());
            }
      }
}
```

以上实例输出结果为：

```
IOException in try block =>test.txt (No such file or directory)
```

以上实例中，我们实例一个 BufferedReader 对象从 test.txt 文件中读取数据。

在 try-with-resources 语句中声明和实例化 BufferedReader 对象，执行完毕后实例资源，不需要考虑 try 语句是正常执行还是抛出异常。

如果发生异常，可以使用 catch 来处理异常。

再看下不使用 **try-with-resources** 而改成 **finally** 来关闭资源，整体代码量多了很多，而且更复杂繁琐了：

```
import java.io.*;

class RunoobTest {
      public static void main(String[] args) {
            BufferedReader br = null;
            String line;
            try {
                  System.out.println("Entering try block");
                  br = new BufferedReader(new FileReader("test.txt"));
                  while ((line = br.readLine()) != null) {
                        System.out.println("Line =>"+line);
                  }
            } catch (IOException e) {
                  System.out.println("IOException in try block =>" + e.getMessage());
            } finally {
                  System.out.println("Entering finally block");
                  try {
                        if (br != null) {
                              br.close();
                        }
                  } catch (IOException e) {
                        System.out.println("IOException in finally block =>"+e.getMessage());
                  }
            }
      }
}
```

以上实例输出结果为：

```
Entering try block
IOException in try block =>test.txt (No such file or directory)
Entering finally block
```

9. try-with-resources 处理多个资源

try-with-resources 语句中可以声明多个资源，方法是使用分号；分隔各个资源：

```
import java.io.*;
import java.util.*;
class RunoobTest {
      public static void main(String[] args) throws IOException{
            try (Scanner scanner = new Scanner(new File("testRead.txt"));
                  PrintWriter writer = new PrintWriter(new File("testWrite.txt"))) {
                while (scanner.hasNext()) {
                      writer.print(scanner.nextLine());
                }
            }
      }
}
```

以上实例使用 Scanner 对象从 testRead.txt 文件中读取一行并将其写入新的 testWrite.txt 文件中。

多个声明资源时， **try-with-resources** 语句以相反的顺序关闭这些资源。 在本例中，PrintWriter 对象先关闭，然后 Scanner 对象关闭。

10. 声明自定义异常

在 Java 中你可以自定义异常。编写自己的异常类时需要记住下面的几点。

- 所有异常都必须是 Throwable 的子类。
- 如果希望写一个检查性异常类，则需要继承 Exception 类。
- 如果你想写一个运行时异常类，那么需要继承 RuntimeException 类。

可以像下面这样定义自己的异常类：

class MyException extends Exception{ }

只继承Exception 类来创建的异常类是检查性异常类。

下面的 InsufficientFundsException 类是用户定义的异常类，它继承自 Exception。

一个异常类和其它任何类一样，包含有变量和方法。

第十三章 Java标准库

1. Object类

1.1 Object 通用方法

概览

`java.lang.Object` 类是Java语言中的根类，即所有类的父类。它中描述的所有方法子类都可以使用。在对象实例化的时候，最终找的父类就是Object。

如果一个类没有特别指定父类， 那么默认则继承自Object类。

```
public native int hashCode()
public boolean equals(Object obj)
protected native Object clone() throws CloneNotSupportedException
public String toString()
public final native Class<?> getClass()
protected void finalize() throws Throwable {}
public final native void notify()
public final native void notifyAll()
public final native void wait(long timeout) throws InterruptedException
public final void wait(long timeout, int nanos) throws InterruptedException
public final void wait() throws InterruptedException
```

equals()

1. 等价关系

两个对象具有等价关系，需要满足以下五个条件：

Ⅰ 自反性

```
x.equals(x); // true
```

Ⅱ 对称性

```
x.equals(y) == y.equals(x); // true
```

Ⅲ 传递性

```
if (x.equals(y) && y.equals(z))
      x.equals(z); // true;
```

Ⅳ 一致性

多次调用 equals() 方法结果不变

```
x.equals(y) == x.equals(y); // true
```

Ⅴ 与 null 的比较

对任何不是 null 的对象 x 调用 x.equals(null) 结果都为 false

```
x.equals(null); // false;
```

2. 等价与相等

- 对于基本类型，== 判断两个值是否相等，基本类型没有 equals() 方法。
- 对于引用类型，== 判断两个变量是否引用同一个对象，而 equals() 判断引用的对象是否等价。

```
Integer x = new Integer(1);
Integer y = new Integer(1);
System.out.println(x.equals(y)); // true
System.out.println(x == y);      // false
```

3. 实现

- 检查是否为同一个对象的引用，如果是直接返回 true；
- 检查是否是同一个类型，如果不是，直接返回 false；
- 将 Object 对象进行转型；
- 判断每个关键域是否相等。

```
public class EqualExample {

        private int x;
        private int y;
        private int z;

        public EqualExample(int x, int y, int z) {
              this.x = x;
              this.y = y;
              this.z = z;
        }

        @Override
        public boolean equals(Object o) {
              if (this == o) return true;
              if (o == null || getClass() != o.getClass()) return false;

              EqualExample that = (EqualExample) o;

              if (x != that.x) return false;
              if (y != that.y) return false;
              return z == that.z;
        }
}
```

hashCode()

hashCode() 返回哈希值，而 equals() 是用来判断两个对象是否等价。等价的两个对象散列值一定相同，但是散列值相同的两个对象不一定等价，这是因为计算哈希值具有随机性，两个值不同的对象可能计算出相同的哈希值。

在覆盖 equals() 方法时应当总是覆盖 hashCode() 方法，保证等价的两个对象哈希值也相等。

HashSet 和 HashMap 等集合类使用了 hashCode() 方法来计算对象应该存储的位置，因此要将对象添加到这些集合类中，需要让对应的类实现 hashCode() 方法。

下面的代码中，新建了两个等价的对象，并将它们添加到 HashSet 中。我们希望将这两个对象当成一样的，只在集合中添加一个对象。但是 EqualExample 没有实现 hashCode() 方法，因此这两个对象的哈希值是不同的，最终导致集合添加了两个等价的对象。

```
EqualExample e1 = new EqualExample(1, 1, 1);
EqualExample e2 = new EqualExample(1, 1, 1);
System.out.println(e1.equals(e2)); // true
HashSet<EqualExample> set = new HashSet<>();
set.add(e1);
set.add(e2);
System.out.println(set.size());    // 2
```

理想的哈希函数应当具有均匀性，即不相等的对象应当均匀分布到所有可能的哈希值上。这就要求了哈希函数要把所有域的值都考虑进来。可以将每个域都当成 R 进制的某一位，然后组成一个 R 进制的整数。

R 一般取 31，因为它是一个奇素数，如果是偶数的话，当出现乘法溢出，信息就会丢失，因为与 2 相乘相当于向左移一位，最左边的位丢失。并且一个数与 31 相乘可以转换成移位和减法：`31*x == (x<<5)-x`，编译器会自动进行这个优化。

```
@Override
public int hashCode() {
      int result = 17;
      result = 31 * result + x;
      result = 31 * result + y;
      result = 31 * result + z;
      return result;
}
```

toString()

默认返回 ToStringExample@4554617c 这种形式，其中 @ 后面的数值为散列码的无符号十六进制表示。

```
public class ToStringExample {

      private int number;

      public ToStringExample(int number) {
            this.number = number;
      }
}
```

```
ToStringExample example = new ToStringExample(123);
System.out.println(example.toString());
```

```
ToStringExample@4554617c
```

clone()

1. cloneable

clone() 是 Object 的 protected 方法，它不是 public，一个类不显式去重写 clone()，其它类就不能直接去调用该类实例的 clone() 方法。

```
public class CloneExample {
      private int a;
      private int b;
}
```

```
CloneExample e1 = new CloneExample();
// CloneExample e2 = e1.clone(); // 'clone()' has protected access in 'java.lang.Object'
```

重写 clone() 得到以下实现：

```
public class CloneExample {
      private int a;
      private int b;

      @Override
      public CloneExample clone() throws CloneNotSupportedException {
            return (CloneExample)super.clone();
      }
}
```

```
CloneExample e1 = new CloneExample();
try {
      CloneExample e2 = e1.clone();
} catch (CloneNotSupportedException e) {
      e.printStackTrace();
}
```

```
java.lang.CloneNotSupportedException: CloneExample
```

以上抛出了 CloneNotSupportedException，这是因为 CloneExample 没有实现 Cloneable 接口。

应该注意的是，clone() 方法并不是 Cloneable 接口的方法，而是 Object 的一个 protected 方法。Cloneable 接口只是规定，如果一个类没有实现 Cloneable 接口又调用了 clone() 方法，就会抛出 CloneNotSupportedException。

```
public class CloneExample implements Cloneable {
      private int a;
      private int b;

      @Override
      public Object clone() throws CloneNotSupportedException {
            return super.clone();
      }
}
```

2. 浅拷贝

拷贝对象和原始对象的引用类型引用同一个对象。

```
 public class ShallowCloneExample implements Cloneable {

       private int[] arr;

       public ShallowCloneExample() {
             arr = new int[10];
             for (int i = 0; i < arr.length; i++) {
                   arr[i] = i;
             }
       }

       public void set(int index, int value) {
             arr[index] = value;
       }

       public int get(int index) {
             return arr[index];
       }

       @Override
       protected ShallowCloneExample clone() throws CloneNotSupportedException {
             return (ShallowCloneExample) super.clone();
       }
 }
```

```
ShallowCloneExample e1 = new ShallowCloneExample();
ShallowCloneExample e2 = null;
try {
      e2 = e1.clone();
} catch (CloneNotSupportedException e) {
      e.printStackTrace();
}
e1.set(2, 222);
System.out.println(e2.get(2)); // 222
```

3. 深拷贝

拷贝对象和原始对象的引用类型引用不同对象。

```
 public class DeepCloneExample implements Cloneable {

       private int[] arr;

       public DeepCloneExample() {
             arr = new int[10];
             for (int i = 0; i < arr.length; i++) {
                   arr[i] = i;
             }
       }

       public void set(int index, int value) {
             arr[index] = value;
       }

       public int get(int index) {
             return arr[index];
       }

       @Override
       protected DeepCloneExample clone() throws CloneNotSupportedException {
             DeepCloneExample result = (DeepCloneExample) super.clone();
             result.arr = new int[arr.length];
             for (int i = 0; i < arr.length; i++) {
                   result.arr[i] = arr[i];
             }
             return result;
       }
 }
```

```
DeepCloneExample e1 = new DeepCloneExample();
DeepCloneExample e2 = null;
try {
      e2 = e1.clone();
} catch (CloneNotSupportedException e) {
      e.printStackTrace();
}
e1.set(2, 222);
System.out.println(e2.get(2)); // 2
```

4. clone() 的替代方案

使用 clone() 方法来拷贝一个对象即复杂又有风险，它会抛出异常，并且还需要类型转换。Effective Java 书上讲到，最好不要去使用 clone()，可以使用拷贝构造函数或者拷贝工厂来拷贝一个对象。

```
public class CloneConstructorExample {

      private int[] arr;

      public CloneConstructorExample() {
            arr = new int[10];
            for (int i = 0; i < arr.length; i++) {
                  arr[i] = i;
            }
      }

      public CloneConstructorExample(CloneConstructorExample original) {
            arr = new int[original.arr.length];
            for (int i = 0; i < original.arr.length; i++) {
                  arr[i] = original.arr[i];
            }
      }

      public void set(int index, int value) {
            arr[index] = value;
      }

      public int get(int index) {
            return arr[index];
      }
}
```

```
CloneConstructorExample e1 = new CloneConstructorExample();
CloneConstructorExample e2 = new CloneConstructorExample(e1);
e1.set(2, 222);
System.out.println(e2.get(2)); // 2
```

1.2 Objects工具类

Objects类是对象工具类，它里面的的方法都是用来操作对象的。

equals方法

在**JDK7**添加了一个Objects工具类，它提供了一些方法来操作对象，它由一些静态的实用方法组成，这些方法是null-save（空指针安全的）或null-tolerant（容忍空指针的），用于计算对象的hashcode、返回对象的字符串表示形式、比较两个对象。

在比较两个对象的时候，Object的equals方法容易抛出空指针异常，而Objects类中的equals方法就优化了这个问题。方法如下：

- `public static boolean equals(Object a, Object b)`:判断两个对象是否相等。

我们可以查看一下源码，学习一下：

```
public static boolean equals(Object a, Object b) {
      return (a == b) || (a != null && a.equals(b));
}
```

isNull

`static boolean isNull(Object obj)` 判断对象是否为null，如果为null返回true。

```
Student s1 = null;
Student s2 = new Student("蔡徐坤", 22);

// static boolean isNull(Object obj) 判断对象是否为null,如果为null返回true
System.out.println(Objects.isNull(s1)); // true
System.out.println(Objects.isNull(s2)); // false
```

2. 包装器类

2.1 概述

Number类

包装类（Integer、Long、Byte、Double、Float、Short）都是抽象类 Number 的子类

包装器类

Java为了能将8种基本类型当对象来处理，能够连接相关的方法，设置了包装器类。

- byte—Byte
- short—Short
- int — Integer
- long—Long
- char—Character
- float—Float
- double—Double
- boolean—Boolean

包装器类创建

由字面值或基本类型的变量创建包装器类对象的方法。

- 构造方法 new

```
Integer i = new Interger(1);
```

- 调用包装器类型的valueOf方法

```
Double d = Double.valueOf(3.14);
```

装箱拆箱

- 装箱Boxing: 将基本类型转化为包装器类型 包装器类.valueOf(基本数据类型变量或常量)。装箱共享内存。
- 拆箱unBoxing：将包装器类型转化为基本数据类型XX.XXXvalue();拆箱也共享内存

装箱操作

```
Integer i = Integer.valueOf(10);//10是基本数据类型，i是包装器类型
int n = i.intValue();//i是包装器类型，n是包装器类型
```

拆箱操作

```
Boolean.booleanValue()
Character.charValue()
Byte.byteValue()
Short.shortValue()
Integer.intValue()
Long.longValue()
Float.floatValue()
Double.doubleValue()
```

可以自动进行拆箱装箱

```
Integer i = 4;//自动装箱。相当于Integer i = Integer.valueOf(4);
i = i + 5;//等号右边：将i对象转成基本数值(自动拆箱) i.intValue() + 5;
//加法运算完成后，再次装箱，把基本数值转成对象。

```

> 注意事项

- 对象一旦赋值，其值不能在改变。
- ++/--自增自减运算符只能对基本数据类型操作
- 集合中只能存放包装器类型的对象

数据缓存池

new Integer(123) 与 Integer.valueOf(123) 的区别在于：

- new Integer(123) 每次都会新建一个对象；
- Integer.valueOf(123) 会使用缓存池中的对象，多次调用会取得同一个对象的引用。

```
Integer x = new Integer(123);
Integer y = new Integer(123);
System.out.println(x == y);      // false
Integer z = Integer.valueOf(123);
Integer k = Integer.valueOf(123);
System.out.println(z == k);    // true
```

valueOf() 方法的实现比较简单，就是先判断值是否在缓存池中，如果在的话就直接返回缓存池的内容。

```
public static Integer valueOf(int i) {
      if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
      return new Integer(i);
}
```

在 Java 8 中，Integer 缓存池的大小默认为 -128~127。

```
 static final int low = -128;
 static final int high;
 static final Integer cache[];

 static {
       // high value may be configured by property
       int h = 127;
       String integerCacheHighPropValue =
```

```
            sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
        if (integerCacheHighPropValue != null) {
            try {
                int i = parseInt(integerCacheHighPropValue);
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
            } catch( NumberFormatException nfe) {
                // If the property cannot be parsed into an int, ignore it.
            }
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);

        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }
```

编译器会在自动装箱过程调用 valueOf() 方法，因此多个值相同且值在缓存池范围内的 Integer 实例使用自动装箱来创建，那么就会引用相同的对象。

```
Integer m = 123;
Integer n = 123;
System.out.println(m == n); // true
```

基本类型对应的缓冲池如下：

- boolean values true and false
- all byte values
- short values between -128 and 127
- int values between -128 and 127
- char in the range \u0000 to \u007F

在使用这些基本类型对应的包装类型时，如果该数值范围在缓冲池范围内，就可以直接使用缓冲池中的对象。

在 jdk 1.8 所有的数值类缓冲池中，Integer 的缓冲池 IntegerCache 很特殊，这个缓冲池的下界是 - 128，上界默认是 127，但是这个上界是可调的，在启动 jvm 的时候，通过 -XX:AutoBoxCacheMax=<size> 来指定这个缓冲池的大小，该选项在 JVM 初始化的时候会设定一个名为 java.lang.IntegerCache.high 系统属性，然后 IntegerCache 初始化的时候就会读取该系统属性来决定上界。

2.2 Character包装器

转义字符

- `\t` 在文中该处插入一个tab键
- `\b` 在文中该处插入一个后退键
- `\n` 在文中该处换行
- `\r` 在文中该处插入回车
- `\f` 在文中该处插入换页符
- `\'` 在文中该处插入单引号
- `\"` 在文中该处插入双引号

- `\\` 在文中该处插入反斜杠

主要方法

1. isLetter()
   是否是一个字母
2. isDigit()
   是否是一个数字字符
3. isWhitespace()
   是否是一个空白字符
4. isUpperCase()
   是否是大写字母
5. isLowerCase()
   是否是小写字母
6. toUpperCase()
   指定字母的大写形式
7. toLowerCase()
   指定字母的小写形式
8. toString()
   返回字符的字符串形式，字符串的长度仅为1

2.3 Integer类

- Integer类概述

  包装一个对象中的原始类型 int 的值

- Integer类构造方法及静态方法

方法名	说明
public Integer(int value)	根据 int 值创建 Integer 对象(过时)
public Integer(String s)	根据 String 值创建 Integer 对象(过时)
public static Integer valueOf(int i)	返回表示指定的 int 值的 Integer 实例
public static Integer valueOf(String s)	返回保存指定String值的 Integer 对象

- 示例代码

```
public class IntegerDemo {
	public static void main(String[] args) {
		//public Integer(int value)：根据 int 值创建 Integer 对象(过时)
		Integer i1 = new Integer(100);
		System.out.println(i1);

		//public Integer(String s)：根据 String 值创建 Integer 对象(过时)
		Integer i2 = new Integer("100");
	  //Integer i2 = new Integer("abc"); //NumberFormatException
		System.out.println(i2);
		System.out.println("---------");

		//public static Integer valueOf(int i)：返回表示指定的 int 值的 Integer 实例
		Integer i3 = Integer.valueOf(100);
		System.out.println(i3);

		//public static Integer valueOf(String s)：返回保存指定String值的Integer对象
```

```
18              Integer i4 = Integer.valueOf("100");
19              System.out.println(i4);
20          }
21      }
```

3. String类

3.1 String原理

概览

String 被声明为 final，因此它不可被继承。(Integer 等包装类也不能被继承)

在 Java 8 中，String 内部使用 char 数组存储数据。

```
1   public final class String
2         implements java.io.Serializable, Comparable<String>, CharSequence {
3         /** The value is used for character storage. */
4         private final char value[];
5   }
```

在 Java 9 之后，String 类的实现改用 byte 数组存储字符串，同时使用 `coder` 来标识使用了哪种编码。

```
1   public final class String
2         implements java.io.Serializable, Comparable<String>, CharSequence {
3         /** The value is used for character storage. */
4         private final byte[] value;
5
6         /** The identifier of the encoding used to encode the bytes in {@code value}. */
7         private final byte coder;
8   }
```

value 数组被声明为 final，这意味着 value 数组初始化之后就不能再引用其它数组。并且 String 内部没有改变 value 数组的方法，因此可以保证 String 不可变。

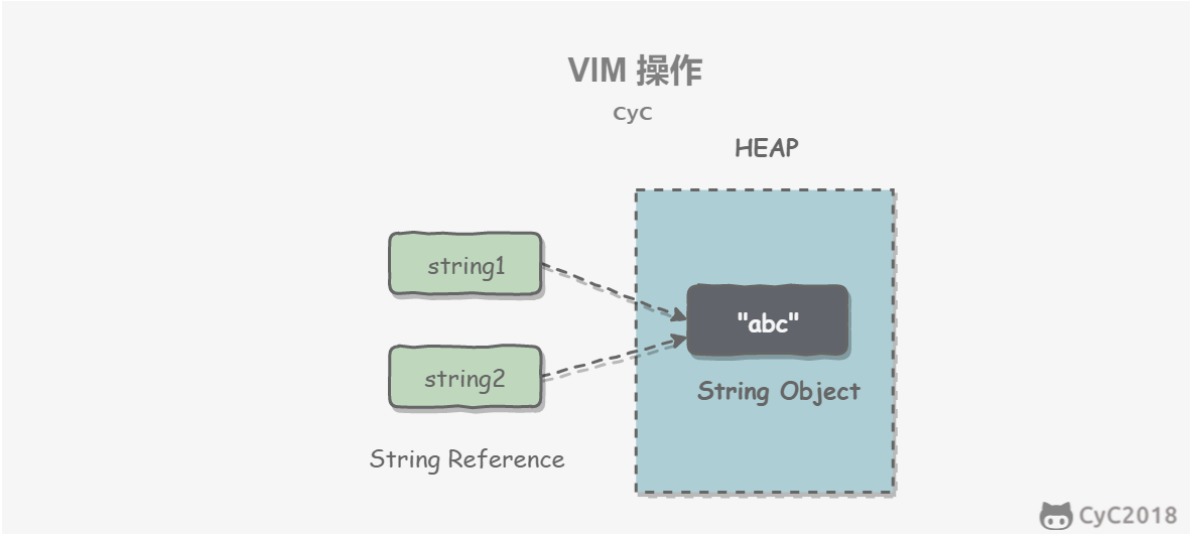
不可变的好处

1. 可以缓存 hash 值

因为 String 的 hash 值经常被使用，例如 String 用做 HashMap 的 key。不可变的特性可以使得 hash 值也不可变，因此只需要进行一次计算。

2. String Pool 的需要

如果一个 String 对象已经被创建过了，那么就会从 String Pool 中取得引用。只有 String 是不可变的，才可能使用 String Pool。



3. 安全性

String 经常作为参数，String 不可变性可以保证参数不可变。例如在作为网络连接参数的情况下如果 String 是可变的，那么在网络连接过程中，String 被改变，改变 String 的那一方以为现在连接的是其它主机，而实际情况却不一定是。

4. 线程安全

String 不可变性天生具备线程安全，可以在多个线程中安全地使用。

Program Creek : Why String is immutable in Java?

String, StringBuffer and StringBuilder

1. 可变性

- String 不可变
- StringBuffer 和 StringBuilder 可变

2. 线程安全

- String 不可变，因此是线程安全的
- StringBuilder 不是线程安全的
- StringBuffer 是线程安全的，内部使用 synchronized 进行同步

String Pool

字符串常量池（String Pool）保存着所有字符串字面量（literal strings），这些字面量在编译时期就确定。不仅如此，还可以使用 String 的 intern() 方法在运行过程将字符串添加到 String Pool 中。

当一个字符串调用 intern() 方法时，如果 String Pool 中已经存在一个字符串和该字符串值相等（使用 equals() 方法进行确定），那么就会返回 String Pool 中字符串的引用；否则，就会在 String Pool 中添加一个新的字符串，并返回这个新字符串的引用。

下面示例中，s1 和 s2 采用 new String() 的方式新建了两个不同字符串，而 s3 和 s4 是通过 s1.intern() 和 s2.intern() 方法取得同一个字符串引用。intern() 首先把 "aaa" 放到 String Pool 中，然后返回这个字符串引用，因此 s3 和 s4 引用的是同一个字符串。

```
String s1 = new String("aaa");
String s2 = new String("aaa");
System.out.println(s1 == s2);           // false
String s3 = s1.intern();
String s4 = s2.intern();
System.out.println(s3 == s4);           // true
```

如果是采用 "bbb" 这种字面量的形式创建字符串，会自动地将字符串放入 String Pool 中。

```
String s5 = "bbb";
String s6 = "bbb";
System.out.println(s5 == s6);  // true
```

在 Java 7 之前，String Pool 被放在运行时常量池中，它属于永久代。而在 Java 7，String Pool 被移到堆中。这是因为永久代的空间有限，在大量使用字符串的场景下会导致 OutOfMemoryError 错误。

- StackOverflow : What is String interning?
- 深入解析 String#intern

new String("abc")

使用这种方式一共会创建两个字符串对象（前提是 String Pool 中还没有 "abc" 字符串对象）。

- "abc" 属于字符串字面量，因此编译时期会在 String Pool 中创建一个字符串对象，指向这个 "abc" 字符串字面量；
- 而使用 new 的方式会在堆中创建一个字符串对象。

创建一个测试类，其 main 方法中使用这种方式来创建字符串对象。

```
public class NewStringTest {
      public static void main(String[] args) {
            String s = new String("abc");
      }
}
```

使用 javap -verbose 进行反编译，得到以下内容：

```
// ...
Constant pool:
// ...
    #2 = Class                    #18                 // java/lang/String
    #3 = String                   #19                 // abc
// ...
   #18 = Utf8                     java/lang/String
   #19 = Utf8                     abc
// ...

   public static void main(java.lang.String[]);
      descriptor: ([Ljava/lang/String;)V
      flags: ACC_PUBLIC, ACC_STATIC
      Code:
         stack=3, locals=2, args_size=1
            0: new           #2                  // class java/lang/String
            3: dup
            4: ldc           #3                  // String abc
            6: invokespecial #4                  // Method java/lang/String."<init>":
(Ljava/lang/String;)V
            9: astore_1
// ...
```

在 Constant Pool 中，#19 存储这字符串字面量 "abc"，#3 是 String Pool 的字符串对象，它指向 #19 这个字符串字面量。在 main 方法中，0: 行使用 new #2 在堆中创建一个字符串对象，并且使用 ldc #3 将 String Pool 中的字符串对象作为 String 构造函数的参数。

以下是 String 构造函数的源码，可以看到，在将一个字符串对象作为另一个字符串对象的构造函数参数时，并不会完全复制 value 数组内容，而是都会指向同一个 value 数组。

```
public String(String original) {
      this.value = original.value;
      this.hash = original.hash;
}
```

```
String sa = new String("hello");
String sb = new String("hello");
System.out.println(sa==sb);
//False

String sc = "hello";
String sd = "hello";
System.out.println(sc==sd);
//True
```

3.2 字符串使用

创建方法

- 直接等于字符串返回的是字面常量的引用。intern创建的字面常量的引用。所以s1==s2
- 使用new创建字符串相当于在堆上创建了字符串。其引用不相等。所以s4!=s5

```
 String str = "Runoob";
 String str2=new String("Runoob");
 String s1 = "Runoob";                   // String 直接创建
 String s2 = "Runoob";                   // String 直
 // s1==s2 true
 String s3 = s1;                            // 相同引用

 String s4 = new String("Runoob");    // String 对象创建
 String s5 = new String("Runoob");    // String 对象创建
 //s4==s5 false
```

常见的构造方法

- String s = “xxx” 最常用
- String(String original) String(“xxx”)
- String(char数组)
- String(char数组,起始下标,长度)
- String(byte数组)
- String(byte数组,起始下标,长度)
- String(StringBuffer buffer)
- String(StringBuilder builder)

字符串拼接

- 字符串链接concat和+号的功能类似。

```
"我的名字是 ".concat("Runoob");
"Hello," + " runoob" + "!"
```

格式化方法

- 创建格式化字符串printf和format两个方法

```
System.out.printf("浮点型变量的值为 " +
                  "%f, 整型变量的值为 " +
                  " %d, 字符串变量的值为 " +
                  "is %s", floatVar, intVar, stringVar);
```

带正则表达式的方法

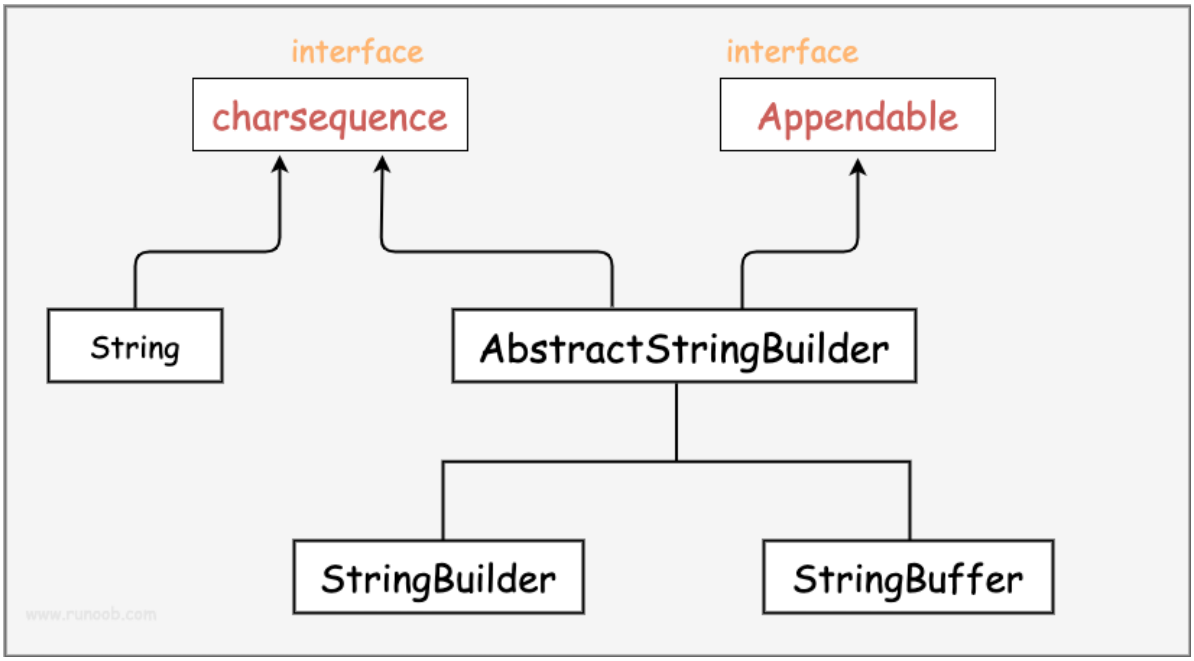
- boolean matches(String regex)告知此字符串是否匹配给定的正则表达式。
- String replaceAll(String regex, String replacement)使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。
- String replaceFirst(String regex, String replacement) 使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串。
- String[] split(String regex)根据给定正则表达式的匹配拆分此字符串。
- String[] split(String regex, int limit)根据匹配给定的正则表达式来拆分此字符串。

其他方法

- char charAt(int index)返回指定索引处的 char 值。
- int compareTo(Object o)把这个字符串和另一个对象比较。
- int compareTo(String anotherString)按字典顺序比较两个字符串。
- int compareToIgnoreCase(String str)按字典顺序比较两个字符串，不考虑大小写。
- String concat(String str)将指定字符串连接到此字符串的结尾。
- boolean contentEquals(StringBuffer sb)当且仅当字符串与指定的StringBuffer有相同顺序的字符时候返回真。
- static String copyValueOf(char[] data)返回指定数组中表示该字符序列的 String。
- static String copyValueOf(char[] data, int offset, int count)返回指定数组中表示该字符序列的 String。
- boolean endsWith(String suffix)测试此字符串是否以指定的后缀结束。
- boolean equals(Object anObject)将此字符串与指定的对象比较。
- boolean equalsIgnoreCase(String anotherString)将此 String 与另一个 String 比较，不考虑大小写。
- byte[] getBytes() 使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。
- byte[] getBytes(String charsetName)使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。
- void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)将字符从此字符串复制到目标字符数组。
- int hashCode()返回此字符串的哈希码。
- int indexOf(int ch)返回指定字符在此字符串中第一次出现处的索引。
- int indexOf(int ch, int fromIndex)返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索。
- int indexOf(String str) 返回指定子字符串在此字符串中第一次出现处的索引。
- int indexOf(String str, int fromIndex)返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始。
- String intern() 返回字符串对象的规范化表示形式。
- int lastIndexOf(int ch) 返回指定字符在此字符串中最后一次出现处的索引。
- int lastIndexOf(int ch, int fromIndex)返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行反向搜索。
- int lastIndexOf(String str)返回指定子字符串在此字符串中最右边出现处的索引。
- int lastIndexOf(String str, int fromIndex) 返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索。
- int length()返回此字符串的长度。

- boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)测试两个字符串区域是否相等。
- boolean regionMatches(int toffset, String other, int ooffset, int len)测试两个字符串区域是否相等。
- String replace(char oldChar, char newChar)返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。
- boolean startsWith(String prefix)测试此字符串是否以指定的前缀开始。
- boolean startsWith(String prefix, int toffset)测试此字符串从指定索引开始的子字符串是否以指定前缀开始。
- CharSequence subSequence(int beginIndex, int endIndex) 返回一个新的字符序列，它是此序列的一个子序列。
- String substring(int beginIndex)返回一个新的字符串，它是此字符串的一个子字符串。
- String substring(int beginIndex, int endIndex)返回一个新字符串，它是此字符串的一个子字符串。
- char[] toCharArray()将此字符串转换为一个新的字符数组。
- String toLowerCase()使用默认语言环境的规则将此 String 中的所有字符都转换为小写。
- String toLowerCase(Locale locale) 使用给定 Locale 的规则将此 String 中的所有字符都转换为小写。
- String toString() 返回此对象本身（它已经是一个字符串！）。
- String toUpperCase()使用默认语言环境的规则将此 String 中的所有字符都转换为大写。
- String toUpperCase(Locale locale)使用给定 Locale 的规则将此 String 中的所有字符都转换为大写。
- String trim()返回字符串的副本，忽略前导空白和尾部空白。
- static String valueOf(primitive data type x)返回给定data type类型x参数的字符串表示形式。
- contains(CharSequence chars)判断是否包含指定的字符系列。
- isEmpty()判断字符串是否为空。

3.3 StringBuffer&StringBuilder



简介

StringBuffer 和 StringBuilder 类的对象能够被多次的修改，并且不产生新的未使用对象。

- 使用 StringBuffer 类时，每次都会对 StringBuffer 对象本身进行操作，而不是生成新的对象，所以如果需要对字符串进行修改推荐使用 StringBuffer。
- StringBuilder 的方法不是线程安全的（不能同步访问）。StringBuffer是线程安全的。
- 由于 StringBuilder 相较于 StringBuffer 有速度优势，所以多数情况下建议使用 StringBuilder 类。

StringBuffer的内部实现方式和String不同.

- StringBuffer在进行字符串处理时，不生成新的对象，在内存使用上要优于String类。所以在实际使用时，如果经常需要对一个字符串进行修改，例如插入、删除等操作，使用StringBuffer要更加适合一些。
- String:在String类中没有用来改变已有字符串中的某个字符的方法，由于不能改变一个java字符串中的某个单独字符，所以在JDK文档中称String类的对象是不可改变的。然而，不可改变的字符串具有一个很大的优点:编译器可以把字符串设为共享的。

使用

```
public class RunoobTest{
      public static void main(String args[]){
            StringBuilder sb = new StringBuilder(10);
            sb.append("Runoob..");
            System.out.println(sb);
            sb.append("!");
            System.out.println(sb);
            sb.insert(8, "Java");
            System.out.println(sb);
            sb.delete(5,8);
            System.out.println(sb);
      }
}
```

StringBuffer常用方法

常用方法

- public StringBuffer append(String s)
  将指定的字符串追加到此字符序列。
- public StringBuffer reverse()
  将此字符序列用其反转形式取代。
- public delete(int start, int end)
  移除此序列的子字符串中的字符。
- public insert(int offset, int i)
  将 int 参数的字符串表示形式插入此序列中。
- insert(int offset, String str)
  将 str 参数的字符串插入此序列中。
- replace(int start, int end, String str)
  使用给定 String 中的字符替换此序列的子字符串中的字符。

其他方法

- char charAt(int index)　　返回此序列中指定索引处的 char 值。
- void ensureCapacity(int minimumCapacity)　　确保容量至少等于指定的最小值。
- void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)　　将字符从此序列复制到目标字符数组 dst。
- int indexOf(String str)　　返回第一次出现的指定子字符串在该字符串中的索引。
- int indexOf(String str, int fromIndex)　　从指定的索引处开始，返回第一次出现的指定子字符串在该字符串中的索引。
- int lastIndexOf(String str)　　返回最右边出现的指定子字符串在此字符串中的索引。
- int lastIndexOf(String str, int fromIndex)返回 String 对象中子字符串最后出现的位置。
- int length()　　返回长度（字符数）。
- void setCharAt(int index, char ch)　　将给定索引处的字符设置为 ch。
- void setLength(int newLength)　　设置字符序列的长度。
- CharSequence subSequence(int start, int end)　　返回一个新的字符序列，该字符序列是此序列的子序列。

- String substring(int start)　　返回一个新的 String，它包含此字符序列当前所包含的字符子序列。
- String substring(int start, int end)　　返回一个新的 String，它包含此序列当前所包含的字符子序列。
- String toString()　　返回此序列中数据的字符串表示形式。

3.4 字符串格式化

利用了Formatter类。printf、printfln等方法原理一直。

字符串的格式化相当于将字符串按照指定的格式进行toString(),一般有两种形式：

```
//使用指定的格式字符串和参数返回一个格式化字符串。
 public static String format(String format, Object... args) {
            return new Formatter().format(format, args).toString();
        }

//使用指定的语言环境、格式字符串和参数返回一个格式化字符串。
public static String format(Locale l, String format, Object... args) {
            return new Formatter(l).format(format, args).toString();
        }
```

- 实例

```
String fs;
fs = String.format("浮点型变量的值为 " +
                   "%f, 整型变量的值为 " +
                   " %d, 字符串变量的值为 " +
                   " %s", floatVar, intVar, stringVar);
```

数据转化符

数据类型	说明	转化形式
%s	字符串类型	“string”
%c	字符类型	‘A’
%b	布尔类型	true/false
%o	整数类型（八进制）	111
%d	整数类型（十进制）	17
%x	整数类型（十六进制）	11
%f	浮点类型（基本）	66.66
%e	指数类型	1.11e+5
%a	浮点类型（十六进制）	FF.22
%h	散列码	11
%%	百分比类型	17%
%n	换行符	
%tx	日期与时间类型	

```
1   public class Format01 {
2       public static void main(String[] args) {
3           System.out.println(String.format("字符串：%s", "String"));
4           System.out.println(String.format("字符：%c", 'M'));
5           System.out.println(String.format("布尔类型：%b", 'M'>'A'));
6           System.out.println(String.format("八进制整数类型：%o", 17));
7           System.out.println(String.format("十进制整数类型：%d", 17));
8           System.out.println(String.format("十六进制整数类型：%x", 17));
9           System.out.println(String.format("基本浮点类型：%f", 99.1));
10          System.out.println(String.format("指数类型：%e", 100.111111111));
11          System.out.println(String.format("十六进制浮点类型：%a", 17.111111));
12          System.out.println(String.format("散列码：%h", 17));
13          System.out.println(String.format("百分比类型：17%%"));
14          System.out.print(String.format("换行符：%n", 17));
15      }
16
17  }
```

格式化控制符

标志	说明	示例	输出
+	为正数添加符号	(“正数：%+f”,11.11))	正数：+11.110000
-	左对齐	(“左对齐：%-5d”,11)	左对齐：11
0	整数前面补0	(“数字前面补0：%04d”,11)	数字前面补0：0011
,	对数字分组	(“按,对数字分组：%,d”,111111111)	按,对数字分组：111,111,111
空格	数字前面补空格	(“空格：% 4d”,11)	空格： 11
(	包含负数	(“使用括号包含负数：%(f”,-11.11)	使用括号包含负数：(11.110000)
#	浮点数包含小数，八进制包含0，十六进制包含0x		
<	格式化前一个转换符描述的参数	(“格式化前描述的参数：%f转化后%<.3f”,111.1111111)	格式化前描述的参数：111.111111转化后111.111
$	被格式化的参数索引	(“被格式化的参数索引：%1	

```
public class formatString {
    public static void main(String[] args) {
        System.out.println(String.format("正数：%+f",11.11));
        System.out.println(String.format("右对齐：%+10d",11));
        System.out.println(String.format("左对齐：%-5d",11));
        System.out.println(String.format("数字前面补0：%044d",11));
        System.out.println(String.format("空格：% 4d",11));
        System.out.println(String.format("按,对数字分组：%,d",111111111));
        System.out.println(String.format("使用括号包含负数：%(f",-11.11));
        System.out.println(String.format("浮点数包含小数点：%#f",11.1));
        System.out.println(String.format("八进制包含0：%#o",11));
        System.out.println(String.format("十六进制包含0x：%#x",11));
        System.out.println(String.format("格式化前描述的参数：%f转化后%<3.3f",111.1111111));
        System.out.println(String.format("被格式化的参数索引：%1$d,%2$s",11,"111.1111111"));

    }

}
```

日期格式化符

转换符	说明	示例
c	全部时间日期	星期四 十二月 17 13:11:35 CST 2020
F	年-月-日格式	2020-12-17
D	月/日/年格式	12/17/20
r	HH:MM:SS PM格式（12时制）	01:11:35 下午
T	HH:MM:SS格式（24时制）	13:11:35
R	HH:MM格式（24时制）	13:11

```
public class formatDate {
    public static void main(String[] args) {
        Date date = new Date();
        System.out.println(String.format("全部时间日期：%tc",date));
        System.out.println(String.format("年-月-日格式：%tF",date));
        System.out.println(String.format("月/日/年格式：%tD",date));
        System.out.println(String.format("HH:MM:SS PM格式（12时制）：%tr",date));
        System.out.println(String.format("HH:MM:SS格式（24时制）：%tT",date));
        System.out.println(String.format("HH:MM格式（24时制）：%tR",date));
    }
}
```

时间格式化符

转换符	说明	示例
H	2位数字24时制的小时（不足2位前面补0）	13
l	2位数字12时制的小时	1
k	2位数字24时制的小时	13
M	2位数字的分钟	33
L	3位数字的毫秒	745
S	2位数字的秒	33
N	9位数字的毫秒数	745000000
p	Locale.US,"小写字母的上午或下午标记(英)	下午
Z	时区缩写字符串	CST
z	相对于GMT的RFC822时区的偏移量	+0800
s	1970-1-1 00:00:00 到现在所经过的秒数	1608183213
Q	1970-1-1 00:00:00 到现在所经过的毫秒数	1608183213745

```
public class formatTime {
    public static void main(String[] args) {
        Date date = new Date();
        System.out.println(String.format("2位数字24时制的小时（不足2位前面补0）：%tH", date));
        System.out.println(String.format("2位数字12时制的小时：%tl", date));
```

```
            System.out.println(String.format("2位数字24时制的小时：%tk", date));
            System.out.println(String.format("2位数字的分钟：%tM", date));
            System.out.println(String.format("3位数字的毫秒：%tL", date));
            System.out.println(String.format("2位数字的秒：%tS", date));
            System.out.println(String.format("9位数字的毫秒数：%tN", date));
            System.out.println(String.format("时区缩写字符串：%tZ", date));
            System.out.println(String.format("相对于GMT的RFC822时区的偏移量：%tz", date));
            System.out.println(String.format("Locale.US,\"小写字母的上午或下午标记(英)：%tp", date));
            System.out.println(String.format("1970-1-1 00:00:00 到现在所经过的秒数：%ts", date));
            System.out.println(String.format("1970-1-1 00:00:00 到现在所经过的毫秒数：%tQ", date));

        }

    }
```

类型转换

- 其他类型转字符串

```
1.String s=""+i;
2.String s=Integer.toString(i);
3.String s=String.valueOf(i);
```

- 字符串转其他类型

```
1.int i=Integer.parsenInt(s);
2.int i=Integer.valueOf(s).intValue();
```

4. 数学计算

4.1 Math

Math类

Java 的 Math 包含了用于执行基本数学运算的属性和方法，如初等指数、对数、平方根和三角函数。

Math中的常量

- Math.PI 记录的圆周率
- Math.E 记录e的常量

Math中的函数

三角函数

- Math.sin 正弦函数 Math.asin 反正弦函数
- Math.cos 余弦函数 Math.acos 反余弦函数
- Math.tan 正切函数 Math.atan 反正切函数 Math.atan2 商的反正切函数
- Math.toDegrees 弧度转化为角度 Math.toRadians 角度转化为弧度

舍入函数

- Math.abs 求绝对值
- Math.ceil 得到不小于某数的最大整数
- Math.floor 得到不大于某数的最大整数
- Math.IEEEremainder 求余
- Math.max 求两数中最大

- Math.min 求两数中最小
- Math.round 同上，返回int型或者long型（上一个函数返回double型）

指数幂计算

- Math.sqrt 求开方
- Math.pow 求某数的任意次方, 抛出ArithmeticException处理溢出异常
- Math.exp 求e的任意次方
- Math.log10 以10为底的对数
- Math.log 自然对数
- Math.rint 求距离某数最近的整数（可能比某数大，也可能比它小）

随机数

- Math.random 返回0，1之间的一个随机数

4.2 BigDecimal类

概述

相关内容	具体描述
包	java.math 使用时需要导包
类声明	public class BigDecimal extends Number implements Comparable
描述	BigDecimal类提供了算术，缩放操作，舍入，比较，散列和格式转换的操作。提供了更加精准的数据计算方式

构造方法

构造方法名	描述
BigDecimal(double val)	将double类型的数据封装为BigDecimal对象
BigDecimal(String val)	将 BigDecimal 的字符串表示形式转换为 BigDecimal

注意：推荐使用第二种方式，第一种存在精度问题；

常用方法

BigDecimal类中使用最多的还是提供的进行四则运算的方法，如下：

方法声明	描述
public BigDecimal add(BigDecimal value)	加法运算
public BigDecimal subtract(BigDecimal value)	减法运算
public BigDecimal multiply(BigDecimal value)	乘法运算
public BigDecimal divide(BigDecimal value)	触发运算

注意：对于divide方法来说，如果除不尽的话，就会出现java.lang.ArithmeticException异常。此时可以使用divide方法的另一个重载方法；

> BigDecimal divide(BigDecimal divisor, int scale, int roundingMode): divisor：除数对应的BigDecimal对象；scale:精确的位数；roundingMode取舍模式

> 小结：Java中小数运算有可能会有精度问题，如果要解决这种精度问题，可以使用BigDecimal

5. 日期时间

5.1 概述

创建日期

java.util 包提供了 Date 类来封装当前的日期和时间。 Date 类提供两个构造函数来实例化 Date 对象。

第一个构造函数使用当前日期和时间来初始化对象。

```
Date( )
```

第二个构造函数接收一个参数，该参数是从 1970 年 1 月 1 日起的毫秒数。

```
Date(long millisec)
```

日期操作

- boolean after(Date date)
  若当调用此方法的Date对象在指定日期之后返回true,否则返回false。
- boolean before(Date date)
  若当调用此方法的Date对象在指定日期之前返回true,否则返回false。
- object clone( )
  返回此对象的副本。
- int compareTo(Date date)
  比较当调用此方法的Date对象和指定日期。两者相等时候返回0。调用对象在指定日期之前则返回负数。调用对象在指定日期之后则返回正数。
- int compareTo(Object obj)
  若obj是Date类型则操作等同于compareTo(Date) 。否则它抛出ClassCastException。
- boolean equals(Object date)
  当调用此方法的Date对象和指定日期相等时候返回true,否则返回false。
- long getTime( )
  返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。
- int hashCode( )
  返回此对象的哈希码值。
- void setTime(long time)用自1970年1月1日00:00:00 GMT以后time毫秒数设置时间和日期。
- String toString( )
  把此 Date 对象转换为以下形式的 String： dow mon dd hh:mm:ss zzz yyyy 其中： dow 是一周中的某一天 (Sun, Mon, Tue, Wed, Thu, Fri, Sat)。

日期比较

Java使用以下三种方法来比较两个日期：

- 使用 getTime() 方法获取两个日期（自1970年1月1日经历的毫秒数值），然后比较这两个值。
- 使用方法 before()，after() 和 equals()。例如，一个月的12号比18号早，则 new Date(99, 2, 12).before(new Date (99, 2, 18)) 返回true。
- 使用 compareTo() 方法，它是由 Comparable 接口定义的，Date 类实现了这个接口。

5.2 日期格式化SimpleDateFormat

格式化的方法format

SimpleDateFormat 是一个以语言环境敏感的方式来格式化和分析日期的类。SimpleDateFormat 允许你选择任何用户自定义日期时间格式来运行。

```
import   java.util.*;
import java.text.*;

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

        Date dNow = new Date( );
        SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd hh:mm:ss");

        System.out.println("当前时间为: " + ft.format(dNow));
    }
}
```

反格式化方法parse

```
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
/**
 * 把String转换成Date对象
*/
public class Demo04DateFormatMethod {
      public static void main(String[] args) throws ParseException {
            DateFormat df = new SimpleDateFormat("yyyy年MM月dd日");
            String str = "2020年6月11日";
            Date date = df.parse(str);
            System.out.println(date);
      }
}
```

SimpleDateFormat格式化编码

字母	描述	示例
G	纪元标记	AD
y	四位年份	2001
M	月份	July or 07
d	一个月的日期	10
h	A.M./P.M. (1~12)格式小时	12
H	一天中的小时 (0~23)	22
m	分钟数	30
s	秒数	55
S	毫秒数	234
E	星期几	Tuesday
D	一年中的日子	360
F	一个月中第几周的周几	2 (second Wed. in July)
w	一年中第几周	40
W	一个月中第几周	1
a	A.M./P.M. 标记	PM
k	一天中的小时(1~24)	24
K	A.M./P.M. (0~11)格式小时	10
z	时区	Eastern Standard Time
'	文字定界符	Delimiter
"	单引号	`

Printf&String.format格式化日期

printf 方法可以很轻松地格式化时间和日期。使用两个字母格式，它以 %t 开头并且以下面表格中的一个字母结尾。

转 换 符	说　明	示　例
c	包括全部日期和时间信息	星期六 十月 27 14:21:20 CST 2007
F	"年-月-日"格式	2007-10-27
D	"月/日/年"格式	10/27/07
r	"HH:MM:SS PM"格式（12时制）	02:25:51 下午
T	"HH:MM:SS"格式（24时制）	14:28:16
R	"HH:MM"格式（24时制）	14:28

```java
import java.util.Date;

public class DateDemo {

   public static void main(String[] args) {
       // 初始化 Date 对象
       Date date = new Date();

       //c的使用
       System.out.printf("全部日期和时间信息：%tc%n",date);
       //f的使用
       System.out.printf("年-月-日格式：%tF%n",date);
       //d的使用
       System.out.printf("月/日/年格式：%tD%n",date);
       //r的使用
       System.out.printf("HH:MM:SS PM格式（12时制）：%tr%n",date);
       //t的使用
       System.out.printf("HH:MM:SS格式（24时制）：%tT%n",date);
       //R的使用
       System.out.printf("HH:MM格式（24时制）：%tR",date);
    }
}
```

5.3 系统运行时间

sleep

sleep()使当前线程进入停滞状态（阻塞当前线程），让出CPU的使用、目的是不让当前线程独自霸占该进程所获的CPU资源，以留一定时间给其他线程执行的机会。

```java
import java.util.*;

public class SleepDemo {
    public static void main(String[] args) {
         try {
             System.out.println(new Date( ) + "\n");
             Thread.sleep(1000*3);    // 休眠3秒
             System.out.println(new Date( ) + "\n");
         } catch (Exception e) {
               System.out.println("Got an exception!");
         }
    }
}
```

System.currentTimeMillis( )

```java
import java.util.*;

public class DiffDemo {

    public static void main(String[] args) {
         try {
             long start = System.currentTimeMillis( );
             System.out.println(new Date( ) + "\n");
             Thread.sleep(5*60*10);
             System.out.println(new Date( ) + "\n");
             long end = System.currentTimeMillis( );
             long diff = end - start;
```

```
                System.out.println("Difference is : " + diff);
            } catch (Exception e) {
                System.out.println("Got an exception!");
            }
        }
    }
```

5.4 Calendar类

概念

`java.util.Calendar`是日历类，在Date后出现，替换掉了许多Date的方法。该类将所有可能用到的时间信息封装为静态成员变量，方便获取。日历类就是方便获取各个时间属性的。

获取方式

Calendar为抽象类，由于语言敏感性，Calendar类在创建对象时并非直接创建，而是通过静态方法创建，返回子类对象，如下：

Calendar静态方法

- `public static Calendar getInstance()`：使用默认时区和语言环境获得一个日历

例如：

```
import java.util.Calendar;

public class Demo06CalendarInit {
      public static void main(String[] args) {
            Calendar cal = Calendar.getInstance();
      }
}
```

常用方法

根据Calendar类的API文档，常用方法有：

- `public int get(int field)`：返回给定日历字段的值。
- `public void set(int field, int value)`：将给定的日历字段设置为给定值。
- `public abstract void add(int field, int amount)`：根据日历的规则，为给定的日历字段添加或减去指定的时间量。
- `public Date getTime()`：返回一个表示此Calendar时间值（从历元到现在的毫秒偏移量）的Date对象。

Calendar类中提供很多成员常量，代表给定的日历字段：

字段值	含义
YEAR	年
MONTH	月（从0开始，可以+1使用）
DAY_OF_MONTH	月中的天（几号）
HOUR	时（12小时制）
HOUR_OF_DAY	时（24小时制）
MINUTE	分
SECOND	秒
DAY_OF_WEEK	周中的天（周几，周日为1，可以-1使用）

get/set方法

get方法用来获取指定字段的值，set方法用来设置指定字段的值，代码使用演示：

```
import java.util.Calendar;

public class CalendarUtil {
	public static void main(String[] args) {
		// 创建Calendar对象
		Calendar cal = Calendar.getInstance();
		// 获取年
		int year = cal.get(Calendar.YEAR);
		// 获取月
		int month = cal.get(Calendar.MONTH) + 1;
		// 获取日
		int dayOfMonth = cal.get(Calendar.DAY_OF_MONTH);
		System.out.print(year + "年" + month + "月" + dayOfMonth + "日");
	}
}
```

```
import java.util.Calendar;

public class Demo07CalendarMethod {
	public static void main(String[] args) {
		Calendar cal = Calendar.getInstance();
		// 设置年
		cal.set(Calendar.YEAR, 2020);
		System.out.print(year + "年" + month + "月" + dayOfMonth + "日"); // 2020年6月11日
	}
}
```

add方法

add方法可以对指定日历字段的值进行加减操作，如果第二个参数为正数则加上偏移量，如果为负数则减去偏移量。代码如：

```
import java.util.Calendar;

public class CalendarMethod {
	public static void main(String[] args) {
		Calendar cal = Calendar.getInstance();
```

```
            // 获取年
            int year = cal.get(Calendar.YEAR);
            // 获取月
            int month = cal.get(Calendar.MONTH) + 1;
            // 获取日
            int dayOfMonth = cal.get(Calendar.DAY_OF_MONTH);
            // 2020年6月11日
            System.out.println(year + "年" + month + "月" + dayOfMonth + "日");
            // 使用add方法
            // 加2天
            cal.add(Calendar.DAY_OF_MONTH, 2);
            // 减3年
            cal.add(Calendar.YEAR, -3);
            // 获取年
            int year1 = cal.get(Calendar.YEAR);
            // 获取月
            int month1 = cal.get(Calendar.MONTH) + 1;
            // 获取日
            int dayOfMonth1 = cal.get(Calendar.DAY_OF_MONTH);
            // 2017年6月13日;
            System.out.println(year1 + "年" + month1 + "月" + dayOfMonth1 + "日");
        }
    }
```

getTime方法

Calendar中的getTime方法并不是获取毫秒时刻，而是拿到对应的Date对象。

```
import java.util.Calendar;
import java.util.Date;

public class Demo09CalendarMethod {
      public static void main(String[] args) {
            Calendar cal = Calendar.getInstance();
            Date date = cal.getTime();
            System.out.println(date); // Thu Jun 11 09:37:57 CST 2020
      }
}
```

> 小贴士：
>
> 西方星期的开始为周日，中国为周一。
>
> 在Calendar类中，月份的表示是以0-11代表1-12月。
>
> 日期是有大小关系的，时间靠后，时间越大。

6. Scanner

基本使用方法

```java
Scanner s = new Scanner(System.in);

s.next();
s.nextLine();

s.hasNext();
s.hasNextLine();

//尽量关闭掉
s.close();
```

读取不同类型的数据

- 读取会阻塞，但是读取错误的数据会返回false
  - 没有数据——阻塞
  - 有数据，类型正确——true
  - 有数据，类型错误——false

```java
package com.ykl;

import java.util.Scanner;

public class ScannerTest {
	public static void main(String[] args) {
		Scanner s = new Scanner(System.in);

		System.out.println("使用不同的方式读取数据");

		// hasNext会一直阻塞，直到由新的内容。
		while(s.hasNext()){
			System.out.println("读取前的环节");
			String str = s.next();
			System.out.println("读取到的内容位："+str);
		}


		// hasNextLine会一直阻塞，直到由新的内容。测试一下next()是否会阻塞
		while(s.hasNextLine()){
			System.out.println("读取前的环节");
			String str = s.nextLine();
			System.out.println("读取到的内容位："+str);
		}


		//hasNextInt方法会阻塞，如果不是整数会返回False
		if(s.hasNextInt()){
			System.out.println("读取前的环节");
			int str = s.nextInt();
			System.out.println("读取到的内容位："+str);
		}
		else{
			System.out.println("你输入的不是整数");
		}

		s.close();
	}

```

```
40    }
```

7. 正则表达式

7.1 概述

简介

正则表达式（regex）是一个字符串，由字面值字符和特殊符号组成，是用来描述匹配一个字符串集合的模式，可以用来匹配、查找字符串。

正则表达式的两个主要作用：

- 查找：在字符串中查找符合固定模式的子串
- 匹配：整个字符串是否符合某个格式

在匹配和查找的基础上，实现替换、分割等操作。

基本实例

```
1   import java.util.regex.Matcher;
2   import java.util.regex.Pattern;
3
4   public class RegexMatches
5   {
6         public static void main( String[] args ){
7
8            // 按指定模式在字符串查找
9            String line = "This order was placed for QT3000! OK?";
10           String pattern = "(\\D*)(\\d+)(.*)";
11
12           // 创建 Pattern 对象
13           Pattern r = Pattern.compile(pattern);
14
15           // 现在创建 matcher 对象
16           Matcher m = r.matcher(line);
17           if (m.find( )) {
18               System.out.println("Found value: " + m.group(0) );
19               System.out.println("Found value: " + m.group(1) );
20               System.out.println("Found value: " + m.group(2) );
21               System.out.println("Found value: " + m.group(3) );
22           } else {
23               System.out.println("NO MATCH");
24           }
25       }
26  }
```

7.2 正则表达式语法

- 在其他语言中，`\\`表示：我想要在正则表达式中插入一个普通的（字面上的）反斜杠，请不要给它任何特殊的意义。
- 在 Java 中，`\\` 表示：我要插入一个正则表达式的反斜线，所以其后的字符具有特殊的意义。
- 不要在重复词符中使用空白。如B{3,6}，不能写成 B{3, 6}。空格也是有含义的。
- 可以使用括号来将模式分组。(ab){3}匹配ababab，而ab{3} 匹配 abbb。

字符	匹配	示例
.	任意单个字符，除换行符外	jav.匹配java
[ ]	[ ] 中的任意一个字符	java匹配j[abc]va
-	[ ] 内表示字符范围	java匹配[a-z]av[a-g]
^	在[ ]内的开头，匹配除[ ]内的字符之外的任意一个字符	java匹配j[^b-f]va
|		或
\	将下一字符标记为特殊字符、文本、反向引用或八进制转义符	(匹配(
$	匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性，$ 还会与"\n"或"\r"之前的位置匹配。	;$匹配位于一行及外围的;号
*	零次或多次匹配前面的字符	zo*匹配zoo或z
+	一次或多次匹配前面的字符	zo+匹配zo或zoo
?	零次或一次匹配前面的字符	zo?匹配z或zo
p{n}	n 是非负整数。正好匹配 n 次	o{2}匹配food中的两个o
p{n,}	n 是非负整数。至少匹配 n 次	o{2}匹配foood中的所有o
p{n,m}	M 和 n 是非负整数，其中 n <= m。匹配至少 n 次，至多 m 次	o{1,3}匹配fooood中的三个o
\p{P}	一个标点字符 !"#$%&'()*+,-./:;<=>?@[]^_'{	}~
\b	匹配一个字边界	va\b匹配java中的va，但不匹配javar中的va
\B	非字边界匹配	va\B匹配javar中的va，但不匹配java中的va
\d	数字字符匹配	1[\d]匹配13
\D	非数字字符匹配	[\D]java匹配Jjava
\w	单词字符	java匹配[\w]ava
\W	非单词字符	$java匹配[\W]java
\s	空白字符	Java 2匹配Java\s2
\S	非空白字符	java匹配 j[\S]va
\f	匹配换页符	等效于\x0c和\cL
\n	匹配换行符	等效于\x0a和\cJ

分组说明

```
正则表达式-字符类

- 语法示例：

1. \[abc\]：代表a或者b，或者c字符中的一个。
2. \[^abc\]：代表除a,b,c以外的任何字符。
3. [a-z]：代表a-z的所有小写字符中的一个。
4. [A-Z]：代表A-Z的所有大写字符中的一个。
5. [0-9]：代表0-9之间的某一个数字字符。
6. [a-zA-Z0-9]：代表a-z或者A-Z或者0-9之间的任意一个字符。
7. [a-dm-p]：a 到 d 或 m 到 p之间的任意一个字符。

正则表达式-逻辑运算符

- 语法示例：
  1. &&：并且
  2. |     ：或者

正则表达式-预定义字符

- 语法示例：
  1. "." ：  匹配任何字符。
  2. "\d"：任何数字[0-9]的简写；
  3. "\D"：任何非数字\[^0-9\]的简写；
  4. "\s"：  空白字符：[ \t\n\xOB\f\r] 的简写
  5. "\S"：  非空白字符：\[^\s\] 的简写
  6. "\w"：单词字符：[a-zA-Z_0-9]的简写
  7. "\W"：非单词字符：\[^\w\]

正则表达式-数量词

- 语法示例：
  1. X? ：0次或1次
  2. X* ：0次到多次
  3. X+ ：1次或多次
  4. X{n} ：恰好n次
  5. X{n,} ：至少n次
  6. X{n,m}：n到m次(n和m都是包含的)


正则表达式-分组括号()
```

7.3 基本概念

Patter类和Matcher类

Pattern 类：
pattern 对象是一个正则表达式的编译表示。Pattern 类没有公共构造方法。要创建一个 Pattern 对象，你必须首先调用其公共静态编译方法，它返回一个 Pattern 对象。该方法接受一个正则表达式作为它的第一个参数。

Matcher 类：
Matcher 对象是对输入字符串进行解释和匹配操作的引擎。与Pattern 类一样，Matcher 也没有公共构造方法。你需要调用 Pattern 对象的 matcher 方法来获得一个 Matcher 对象。

捕获组

1. 捕获组是把多个字符当成一个单独单元进行处理的方法，它通过对括号内的字符分组来创建。

```
捕获组通过从左到右计算其括号来编号。

例如：在表达式((A)(B(C)))中，存在四个这样的组：

((A)(B(C)))
(A)
(B(C))
(C)
```

2. 捕获组可以通过调用matcher对象的groupCount方法来查看表达式有多少个分组。（groupCount方法返回一个int值，来表示matcher对象当前有多少个捕获组）
3. 还有一个特殊的组零（group(0)），它代表整个表达式。（该组不包括在groupCount的返回值中）
4. 以 (?) 开头的组是纯的非捕获 组，它不捕获文本，也不针对组合计进行计数。

7.4 Matcher用法

索引方法

1. public int start()
   返回以前匹配的初始索引。
2. public int start(int group)
   返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引
3. public int end()
   返回最后匹配字符之后的偏移量。
4. public int end(int group)
   返回在以前的匹配操作期间，由给定组所捕获子序列的最后字符之后的偏移量。

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
      private static final String REGEX = "\\bcat\\b";
      private static final String INPUT =
                                                "cat cat cat cattie cat";

      public static void main( String[] args ){
          Pattern p = Pattern.compile(REGEX);
          Matcher m = p.matcher(INPUT); // 获取 matcher 对象
          int count = 0;

          while(m.find()) {
             count++;
             System.out.println("Match number "+count);
             System.out.println("start(): "+m.start());
             System.out.println("end(): "+m.end());
         }
     }
}
```

匹配和查找方法

1. public boolean lookingAt()
   尝试将从区域开头开始的输入序列与该模式匹配。开头匹配。
2. public boolean find()
   尝试查找与该模式匹配的输入序列的下一个子序列。
3. public boolean find(int start)
   重置此匹配器，然后尝试查找匹配该模式、从指定索引开始的输入序列的下一个子序列。
4. public boolean matches()
   尝试将整个区域与模式匹配。全局匹配。

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
      private static final String REGEX = "foo";
      private static final String INPUT = "fooooooooooooooooo";
      private static final String INPUT2 = "ooooofooooooooooo";
      private static Pattern pattern;
      private static Matcher matcher;
      private static Matcher matcher2;

      public static void main( String[] args ){
          pattern = Pattern.compile(REGEX);
          matcher = pattern.matcher(INPUT);
          matcher2 = pattern.matcher(INPUT2);

          System.out.println("Current REGEX is: "+REGEX);
          System.out.println("Current INPUT is: "+INPUT);
          System.out.println("Current INPUT2 is: "+INPUT2);


          System.out.println("lookingAt(): "+matcher.lookingAt());
          System.out.println("matches(): "+matcher.matches());
          System.out.println("lookingAt(): "+matcher2.lookingAt());
    }
}
```

替换方法

1. public Matcher appendReplacement(StringBuffer sb, String replacement)
   实现非终端添加和替换步骤。
2. public StringBuffer appendTail(StringBuffer sb)
   实现终端添加和替换步骤。
3. public String replaceAll(String replacement)
   替换模式与给定替换字符串相匹配的输入序列的每个子序列。
4. public String replaceFirst(String replacement)
   替换模式与给定替换字符串匹配的输入序列的第一个子序列。
5. public static String quoteReplacement(String s)
   返回指定字符串的字面替换字符串。这个方法返回一个字符串，就像传递给Matcher类的appendReplacement 方法一个字面字符串一样工作。

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

```

```
public class RegexMatches
{
      private static String REGEX = "dog";
      private static String INPUT = "The dog says meow. " +
                                                    "All dogs say meow.";
      private static String REPLACE = "cat";

      public static void main(String[] args) {
          Pattern p = Pattern.compile(REGEX);
          // get a matcher object
          Matcher m = p.matcher(INPUT);
          INPUT = m.replaceAll(REPLACE);
          System.out.println(INPUT);
    }
}
```

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
    private static String REGEX = "a*b";
    private static String INPUT = "aabfooaabfooabfoobkkk";
    private static String REPLACE = "-";
    public static void main(String[] args) {
         Pattern p = Pattern.compile(REGEX);
         // 获取 matcher 对象
         Matcher m = p.matcher(INPUT);
         StringBuffer sb = new StringBuffer();
         while(m.find()){
             m.appendReplacement(sb,REPLACE);
        }
         m.appendTail(sb);
         System.out.println(sb.toString());
    }
}
```

8. Random

简介

在 Java中要生成一个指定范围之内的随机数字有两种方法：一种是调用 Math 类的 random() 方法，一种是使用 Random 类。

Random()：该构造方法使用一个和当前系统时间对应的数字作为种子数，然后使用这个种子数构造 Random 对象。
Random(long seed)：使用单个 long 类型的参数创建一个新的随机数生成器。

Random 类提供的所有方法生成的随机数字都是均匀分布的，也就是说区间内部的数字生成的概率是均等的

```
package cn.itcast.demo1;

import java.util.Random;//使用时需要先导包
import java.util.Scanner;

```

```
6
7   public class RAndom {
8         public static void main(String[] args) {
9               Random r = new Random();//以系统自身时间为种子数
10              int i = r.nextInt();
11              System.out.println("i"+i);
12              Scanner sc   =new Scanner(System.in);
13              int j = sc.nextInt();
14              Random r2 = new Random(j);//自定义种子数
15              Random r3 = new Random(j);//这里是为了验证上方的注意事项：Random类是伪随机，相同种子数
    相同次数产生的随机数相同
16              int num   = r2.nextInt();
17              int num2 = r3.nextInt();
18              System.out.println("num"+num);
19              System.out.println("num2"+num2);
20        }
21  }
```

常用方法

random.nextInt()	**返回值为整数,范围是int类型范围**
random.nextLong()	返回值为长整型，范围是long类型的范围
random.nextFloat()	返回值为小数，范围是[0,0.1]
random.nextDouble()	返回值为小数，范围是[0,0.1]
random.nextBoolean（）	返回值为boolean值，true和false概率相同
radom.nextGaussian()	返回值为呈高斯（“正态”）分布的 double 值，其平均值是 0.0，标准差是 1.0

9. System

比较有用的方法

```
1   static void setIn(InputStream in)       // 标准输入的重定向
2   static void setOut(PrintStream out)    // 标准输出的重定向
3   static void setErr(PrintStream err)    // 标准错误的重定向
4   /*****************************/
5   static Map<String,String> getenv()      // 返回所有的环境变量的键值对
6   static String getenv(String name)       // 返回特定环境变量的值
7   /*****************************/
8   static Properties getProperties()       // 返回所有的系统属性
9   static String getProperty(String key) // 返回特定的系统属性的值
10  /*****************************/
11  static void setProperties(Properties props)             // 设置所有的系统属性
12  static String setProperty(String key, String value) // 设置特定的系统属性的值
13  /*****************************/
14  static long currentTimeMillis()      // 获取当前系统时间，用毫秒表示，从 1970 年开始。HotSpot VM
    中可用
```

arraycopy(...)方法

arraycopy(...)方法将指定原数组中的数据从指定位置复制到目标数组的指定位置。

```
static void arraycopy(Object src,  int srcPos, Object dest, int destPos, int length)
```

```
package com.ibelifly.commonclass.system;

public class Test1 {
	public static void main(String[] args) {
		int[] arr={23,45,20,67,57,34,98,95};
		int[] dest=new int[8];
		System.arraycopy(arr,4,dest,4,4);
		for (int x:dest) {
			System.out.print(x+" ");
		}
	}
}
```

exit(int status)方法

exit(int status)方法用于终止当前运行的Java虚拟机。如果参数是0，表示正常退出JVM；如果参数非0，表示异常退出JVM。

```
package com.ibelifly.commonclass.system;

public class Test4 {
	public static void main(String[] args) {
		System.out.println("程序开始了");
		System.exit(0); //因为此处已经终止当前运行的Java虚拟机，故不会执行之后的代码
		System.out.println("程序结束了");
	}
}
```

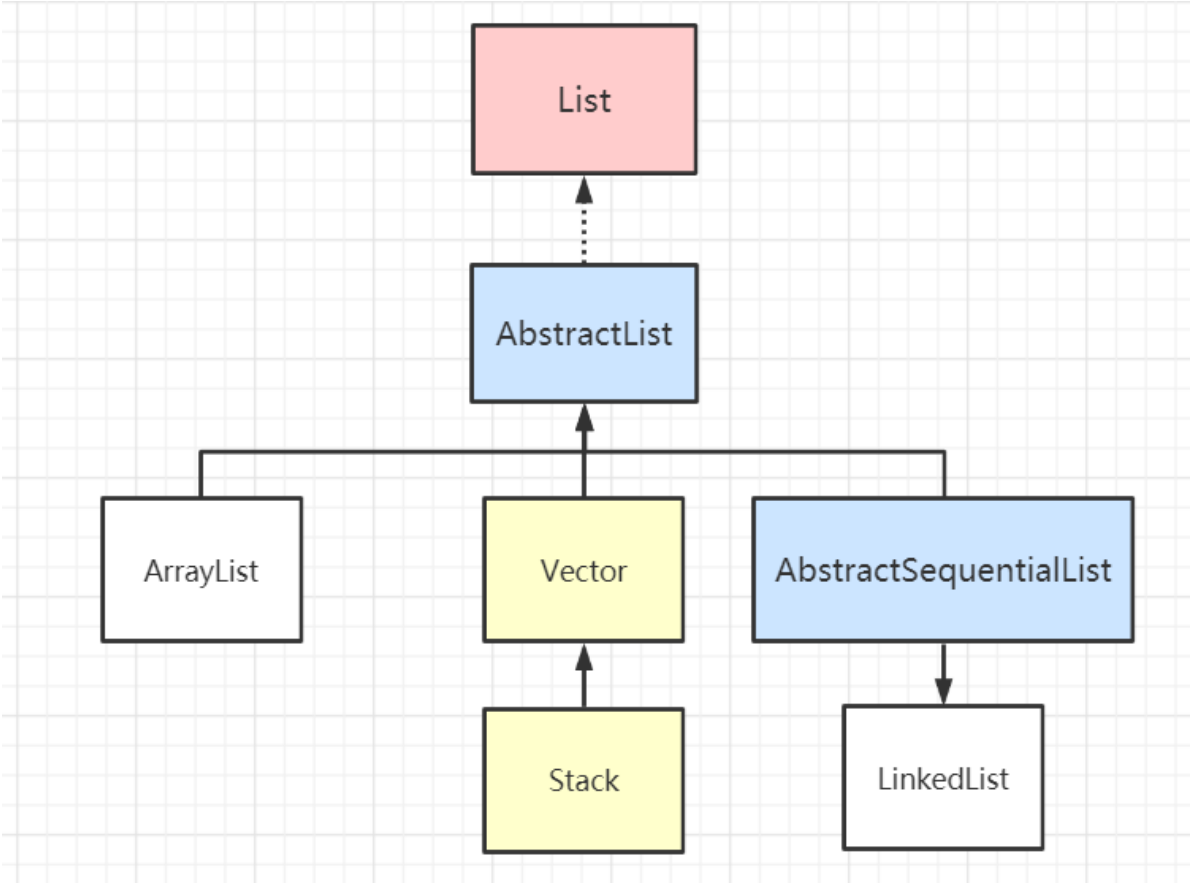
第十四章 集合

1. List 接口

概述

简介

List 接口和 Set 接口齐头并进，是我们日常开发中接触的很多的一种集合类型了。整个 List 集合的组成部分如下图



`List` 接口直接继承 Collection 接口，它定义为可以存储**重复**元素的集合，并且元素按照插入顺序**有序排列**，且可以通过**索引**访问指定位置的元素。常见的实现有：ArrayList、LinkedList、Vector和Stack

AbstractList 和 AbstractSequentialList

AbstractList 抽象类实现了 List 接口，其内部实现了所有的 List 都需具备的功能，子类可以专注于实现自己具体的操作逻辑。

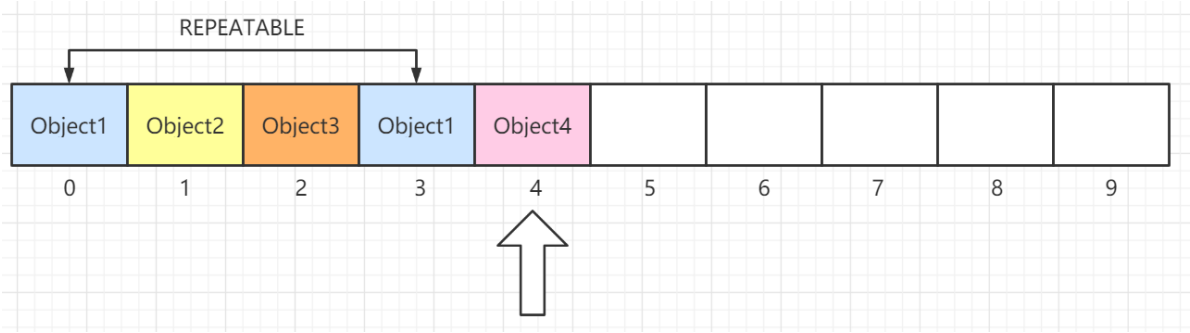
```
// 查找元素 o 第一次出现的索引位置
public int indexOf(Object o)
// 查找元素 o 最后一次出现的索引位置
public int lastIndexOf(Object o)
//...
```

AbstractSequentialList 抽象类继承了 AbstractList，在原基础上限制了访问元素的顺序**只能够按照顺序访问**，而**不支持随机访问**，如果需要满足随机访问的特性，则继承 AbstractList。子类 LinkedList 使用链表实现，所以仅能支持**顺序访问**，顾继承了 `AbstractSequentialList` 而不是 AbstractList。

ArrayList

底层原理

ArrayList 以**数组**作为存储结构，它是**线程不安全**的集合；具有**查询快、在数组中间或头部增删慢**的特点，所以它除了线程不安全这一点，其余可以替代 `Vector`，而且线程安全的 ArrayList 可以使用 `CopyOnWriteArrayList` 代替 Vector。



关于 ArrayList 有几个重要的点需要注意的：

- 具备**随机访问**特点，**访问元素的效率**较高，ArrayList 在**频繁插入、删除**集合元素的场景下效率较低。
- 底层数据结构：ArrayList 底层使用数组作为存储结构，具备**查找快、增删慢**的特点
- 线程安全性：ArrayList 是**线程不安全**的集合
- ArrayList **首次扩容**后的长度为 10，调用 add() 时需要计算容器的最小容量。可以看到如果数组 elementData 为空数组，会将最小容量设置为 10，之后会将数组长度完成首次扩容到 10。

另外一篇文章

- Ordered – arraylist中的元素保留其排序，默认情况下是其添加到列表的顺序。
- Index based –可以使用索引位置随机访问元素。 索引以'0'开头。
- Dynamic resizing –当需要添加的元素数量超过当前大小时，ArrayList会动态增长。
- Non synchronized –默认情况下，ArrayList不同步。 程序员需要适当地使用synchronized关键字，或者仅使用Vector类。
- Duplicates allowed -我们可以在arraylist中添加重复元素。 不能成组放置。

```
1   // new ArrayList 时的默认空数组
2   private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
3   // 默认容量
4   private static final int DEFAULT_CAPACITY = 10;
5   // 计算该容器应该满足的最小容量
6   private static int calculateCapacity(Object[] elementData, int minCapacity) {
7         if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
8               return Math.max(DEFAULT_CAPACITY, minCapacity);
9         }
10        return minCapacity;
11  }
```

- 集合从**第二次扩容**开始，数组长度将扩容为原来的 1.5 倍，即：newLength = oldLength * 1.5

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

构造函数

- 没有逐个元素初始化的方法，使用Arrays.asList能够添加对象初始化。

```
//Empty arraylist
List<String> names = new ArrayList<>();

//Arraylist initialized with another collection
List<Integer> numbers = new ArrayList<>(Arrays.asList(1,2,3,4,5));
```

常用方法

- add()添加单个元素

```
public boolean add(E e) {
      ensureCapacityInternal(size + 1);   // Increments modCount!!
      elementData[size++] = e;
      return true;
}
```

- addAll()方法将给定集合的所有元素添加到arraylist中。 始终使用泛型来确保您仅在给定列表中添加某种类型的元素。

```
 import java.util.ArrayList;

 public class ArrayListExample
 {
       public static void main(String[] args)
       {
             ArrayList<String> list1 = new ArrayList<>();      //list 1

             list1.add("A");
             list1.add("B");
             list1.add("C");
             list1.add("D");

             ArrayList<String> list2 = new ArrayList<>();      //list 2

             list2.add("E");

             list1.addAll(list2);

             System.out.println(list1);         //combined list
       }
 }
```

- clear()方法将arraylist clear

```
 import java.util.ArrayList;

 public class ArrayListExample
 {
       public static void main(String[] args)
       {
             ArrayList<String> arrayList = new ArrayList<>();

             arrayList.add("A");
             arrayList.add("B");
             arrayList.add("C");
```

```
            arrayList.add("D");

            System.out.println(arrayList);

            arrayList.clear();

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

- clone()方法创建arraylist的浅表副本 。

```
import java.util.ArrayList;

public class ArrayListExample
{
      @SuppressWarnings("unchecked")
      public static void main(String[] args)
      {
            ArrayList<String> arrayListObject = new ArrayList<>();

            arrayListObject.add("A");
            arrayListObject.add("B");
            arrayListObject.add("C");
            arrayListObject.add("D");

            System.out.println(arrayListObject);

            ArrayList<String> arrayListClone =  (ArrayList<String>) arrayListObject.clone();

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

- clone深拷贝。创建集合的深层副本非常容易。 我们需要创建一个新的collection实例，并将给定collection中的所有元素一一复制到克隆的collection中。 请注意，我们将在克隆集合中复制元素的克隆。

```
ArrayList<Employee> employeeList = new ArrayList<>();
ArrayList<Employee> employeeListClone = new ArrayList<>();

Iterator<Employee> iterator = employeeList.iterator();

while(iterator.hasNext())
{
      //Add the object clones
      employeeListClone.add((Employee) iterator.next().clone());
}
```

- cantains()数组列表中存储了几个字母。 我们将尝试找出列表中是否包含字母“ A”和“ Z”。

```
public class ArrayListExample
{
      public static void main(String[] args)
      {
            ArrayList<String> list = new ArrayList<>(2);

            list.add("A");
```

```
            list.add("B");
            list.add("C");
            list.add("D");

            System.out.println( list.contains("A") );          //true

            System.out.println( list.contains("Z") );          //false
      }
}
```

- indexOf()判断是否存在.返回此列表中指定元素的首次出现的索引。 如果列表不包含元素，它将返回'-1'。

```
public class ArrayListExample
{
      public static void main(String[] args)
      {
            ArrayList<String> list = new ArrayList<>(2);

            list.add("A");
            list.add("B");
            list.add("C");
            list.add("D");

            System.out.println( list.indexOf("A") > 0 );          //true

            System.out.println( list.indexOf("Z") > 0);          //false
      }
}
```

- lastIndexOf()返回此列表中最后一次出现的指定元素的索引。 如果列表不包含元素，它将返回'-1'。

```
public int lastIndexOf(Object object) {
      if (o == null) {
            for (int i = size-1; i >= 0; i--)
                  if (elementData[i]==null)
                        return i;
      } else {
            for (int i = size-1; i >= 0; i--)
                  if (o.equals(elementData[i]))
                        return i;
      }
      return -1;
}

```

- get(int index)方法返回列表中指定位置'index'处的元素。

```
import java.util.ArrayList;
import java.util.Arrays;

public class ArrayListExample
{
      public static void main(String[] args)
      {
            ArrayList<String> list = new ArrayList<>(Arrays.asList("alex", "brian", "charles",
"dough"));

```

```
            String firstName = list.get(0);             //alex
            String secondName = list.get(1);            //brian

            System.out.println(firstName);
            System.out.println(secondName);
        }
    }
```

- boolean remove(Object o) –从列表中删除第一次出现的指定元素。 true从列表中删除了任何元素，则返回true ，否则返回false 。对象remove （int index）引发IndexOutOfBoundsException-移除此列表中指定位置的元素。 将所有后续元素向左移动。 返回从列表中移除的元素。 如果参数索引无效，则引发异常。

```
import java.util.ArrayList;
import java.util.Arrays;

public class ArrayListExample
{
      public static void main(String[] args) throws CloneNotSupportedException
      {
            ArrayList<String> alphabets = new ArrayList<>(Arrays.asList("A", "B", "C", "D"));

            System.out.println(alphabets);

            alphabets.remove("C");              //Element is present

            System.out.println(alphabets);

            alphabets.remove("Z");              //Element is NOT present

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

- removeAll()方法遍历arraylist的所有元素。 对于每个元素，它将元素传递给参数集合的contains()方法。如果在参数集合中找到element，它将重新排列索引。 如果未找到element，则将其保留在后备数组中。

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;

public class ArrayListExample
{
      public static void main(String[] args) throws CloneNotSupportedException
      {
            ArrayList<String> alphabets = new ArrayList<>(Arrays.asList("A", "B", "A", "D", "A"));

            System.out.println(alphabets);

            alphabets.removeAll(Collections.singleton("A"));

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

- removeIf()方法采用Predicate类型的单个参数。 谓词接口是一种功能接口，表示一个参数的条件（布尔值函数）。 它检查给定参数是否满足条件。

```
import java.util.ArrayList;
import java.util.Arrays;

public class ArrayListExample
{
      public static void main(String[] args) throws CloneNotSupportedException
      {
            ArrayList<Integer> numbers = new ArrayList<>(Arrays.asList(1,2,3,4,5,6,7,8,9,10));

            numbers.removeIf( number -> number%2 == 0 );

            System.out.println(numbers);
      }
}

```

- retainAll()方法来保留列表中存在于指定参数集合中的所有元素。

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;

public class ArrayListExample
{
      public static void main(String[] args) throws CloneNotSupportedException
      {
            ArrayList<String> alphabets = new ArrayList<>(Arrays.asList("A", "B", "A", "D", "A"));

            System.out.println(alphabets);

            alphabets.retainAll(Collections.singleton("A"));

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

- sort()方法使用Arrays.sort()方法对列表中的元素进行比较和排序。sort()方法接受Comparator实现类的实例，该实例必须能够比较arraylist中包含的元素

```
import java.util.Comparator;

public class NameSorter implements Comparator<Employee>
{
      @Override
      public int compare(Employee o1, Employee o2) {
            return o1.getName().compareToIgnoreCase(o1.getName());
      }
}



import java.time.LocalDate;
import java.time.Month;
import java.util.ArrayList;

public class ArrayListExample
```

```
{
      public static void main(String[] args) throws CloneNotSupportedException
      {
            ArrayList<Employee> employees = new ArrayList<>();

            employees.add(new Employee(11, "Alex", LocalDate.of(2018, Month.APRIL, 21)));
            employees.add(new Employee(41, "Brian", LocalDate.of(2018, Month.APRIL, 22)));
            employees.add(new Employee(31, "David", LocalDate.of(2018, Month.APRIL, 25)));
            employees.add(new Employee(51, "Charles", LocalDate.of(2018, Month.APRIL, 23)));
            employees.add(new Employee(21, "Edwin", LocalDate.of(2018, Month.APRIL, 24)));

            employees.sort(new NameSorter());
            System.out.println(employees);
      }
}

```

- toArray()将arraylist转换为对象数组并遍历数组内容

```
import java.util.ArrayList;
import java.util.Arrays;

public class ArrayListExample
{
      public static void main(String[] args)
      {
            ArrayList<String> list = new ArrayList<>(2);

            list.add("A");
            list.add("B");
            list.add("C");
            list.add("D");

            //Convert to object array
            Object[] array = list.toArray();

            System.out.println( Arrays.toString(array) );

            //Iterate and convert to desired type
            for(Object o : array) {
                  String s = (String) o;

                  System.out.println(s);
            }
      }
}
// 可以控制转换完成的结果。
import java.util.ArrayList;
import java.util.Arrays;

public class ArrayListExample
{
      public static void main(String[] args)
      {
            ArrayList<String> list = new ArrayList<>(2);

            list.add("A");
            list.add("B");
```

```
            list.add("C");
            list.add("D");

            //Convert to string array
            String[] array = list.toArray(new String[list.size()]);

            System.out.println(Arrays.toString(array));
        }
    }
```

- sublist获取子列表

```
import java.util.ArrayList;
import java.util.Arrays;

public class ArrayListExample
{
      public static void main(String[] args)
      {
            ArrayList<Integer> list = new ArrayList<>(Arrays.asList(0,1,2,3,4,5,6,7,8,9));

            ArrayList<Integer> sublist = new ArrayList<Integer>( list.subList(2, 6) );

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

遍历方法

五种loop ArrayList

- Java程序使用standard for loop遍历对象的数组列表。(能够控制遍历计数，方便进行排序等算法操作)

```
ArrayList<String> namesList = new ArrayList<String>(Arrays.asList( "alex", "brian", "charles") );

for(int i = 0; i < namesList.size(); i++)
{
      System.out.println(namesList.get(i));
}
```

- 使用foreach loop遍历对象的数组列表。（最方便的遍历。）

```
ArrayList<String> namesList = new ArrayList<String>(Arrays.asList( "alex", "brian", "charles") );

for(String name : namesList)
{
      System.out.println(name);
}
```

- 使用列表迭代器对象迭代对象的数组列表。(能够在遍历的时候删除。)

```
ArrayList<String> namesList = new ArrayList<String>(Arrays.asList( "alex", "brian", "charles") );

ListIterator<String> listItr = namesList.listIterator();

while(listItr.hasNext())
{
      System.out.println(listItr.next());
}
```

- while循环。 （可能在某些算法中比较方便，除非你想让循环体控制遍历计数）

```
ArrayList<String> namesList = new ArrayList<String>(Arrays.asList( "alex", "brian", "charles") );

int index = 0;
while (namesList.size() > index)
{
    System.out.println(namesList.get(index++));
}
```

- 使用Java 8流API遍历对象的数组列表。 使用stream.foreach()方法从arraylist对象创建元素流，并stream.foreach()获取元素。

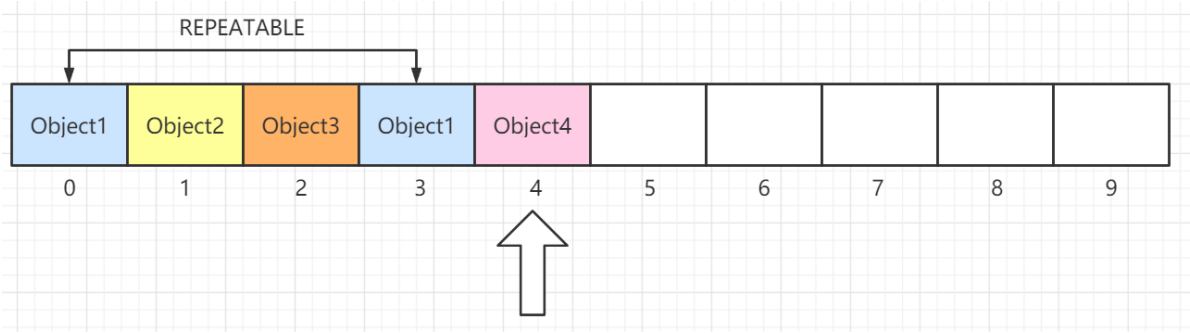
```
Iterate arraylist with stream api
ArrayList<String> namesList = new ArrayList<String>(Arrays.asList( "alex", "brian", "charles") );
namesList.forEach(name -> System.out.println(name));
```

- listIterator()方法获得的列表迭代器来迭代arraylist。 ListIterator支持在迭代列表时添加和删除列表中的元素。
  - listIterator.add(Element e) -将该元素立即插入将由next()返回的元素之前或将要返回的previous()方法的元素之后。
  - listIterator.remove() -从列表中删除next()或previous()方法返回的最后一个元素。

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.ListIterator;

public class ArrayListExample
{
      public static void main(String[] args) throws CloneNotSupportedException
      {
            ArrayList<String> alphabets = new ArrayList<>(Arrays.asList("A", "B", "C", "D"));

            ListIterator<String> listItr = alphabets.listIterator();

            System.out.println("==========Forward=========");

            while(listItr.hasNext()) {
                  System.out.println(listItr.next());
            }

            System.out.println("==========Backward=========");

            while(listItr.hasPrevious()) {
                  System.out.println(listItr.previous());
            }
      }
}
```

Vector

底层原理



`Vector` 在现在已经是一种过时的集合了，包括继承它的 `Stack` 集合也如此，它们被淘汰的原因都是因为**性能**低下。

> JDK 1.0 时代，ArrayList 还没诞生，大家都是使用 Vector 集合，但由于 Vector 的**每个操作**都被 **synchronized** 关键字修饰，即使在线程安全的情况下，仍然**进行无意义的加锁与释放锁**，造成额外的性能开销，做了无用功。

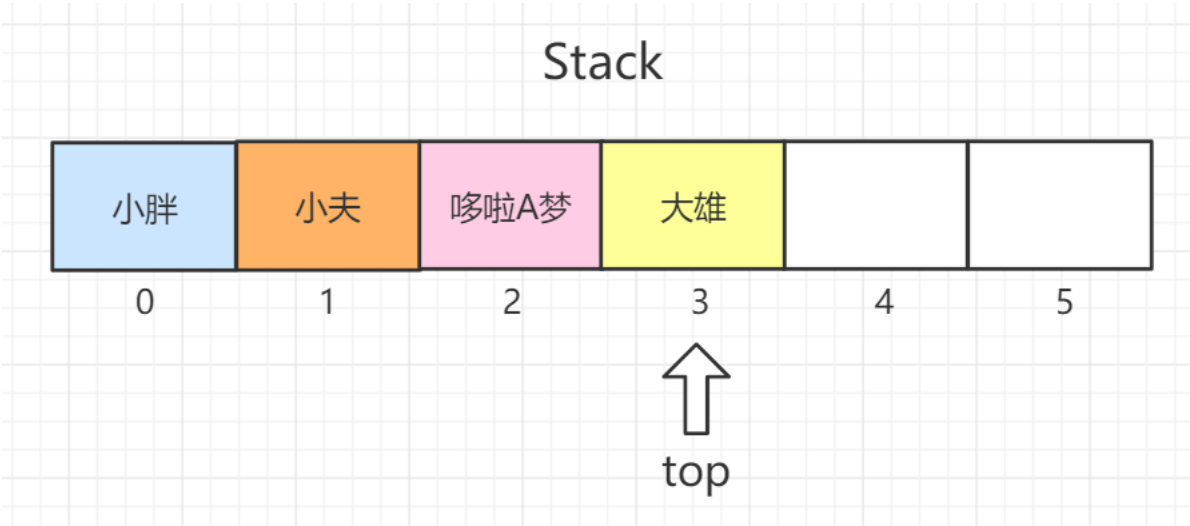
```
1    public synchronized boolean add(E e);
2    public synchronized E get(int index);
```

在 JDK 1.2 时，Collection 家族出现了，它提供了大量**高性能、适用於不同场合**的集合，而 Vector 也是其中一员，但由于 Vector 在每个方法上都加了锁，由于需要兼容许多老的项目，很难在此基础上优化 `Vector` 了，所以渐渐地也就被历史淘汰了。

现在，在**线程安全**的情况下，不需要选用 Vector 集合，取而代之的是 **ArrayList** 集合；在并发环境下，出现了 `CopyOnWriteArrayList`，Vector 完全被弃用了。

Stack

底层原理



`Stack` 是一种 `后入先出（LIFO）` 型的集合容器，如图中所示，`大雄` 是最后一个进入容器的，top指针指向大雄，那么弹出元素时，大雄也是第一个被弹出去的。

Stack 继承了 Vector 类，提供了栈顶的压入元素操作（push）和弹出元素操作（pop），以及查看栈顶元素的方法（peek）等等，但由于继承了 Vector，正所谓跟错老大没福报，Stack 也渐渐被淘汰了。

取而代之的是后起之秀 `Deque` 接口，其实现有 `ArrayDeque`，该数据结构更加完善、可靠性更好，依靠队列也可以实现 `LIFO` 的栈操作，所以优先选择 ArrayDeque 实现栈。

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

ArrayDeque 的数据结构是：`数组`，并提供**头尾指针下标**对数组元素进行操作。本文也会讲到哦，客官请继续往下看，莫着急！😄

CopyOnWriteArrayList

> 用来替代vector，提供现成安全的list

底层原理

Java CopyOnWriteArrayList是ArrayList的thread-safe变体，其中所有可变操作（添加，设置等）都通过对基础array进行全新复制来实现。

- CopyOnWriteArrayList类实现List和RandomAccess接口，因此提供ArrayList类中可用的所有功能。
- 使用CopyOnWriteArrayList进行更新操作的成本很高，因为每个突变都会创建基础数组的克隆副本，并为其添加/更新元素。
- 它是ArrayList的线程安全版本。 每个访问列表的线程在初始化此列表的迭代器时都会看到自己创建的后备阵列快照版本。
- 因为它在创建迭代器时获取基础数组的快照，所以它不会抛出ConcurrentModificationException 。
- 不支持对迭代器的删除操作（删除，设置和添加）。 这些方法抛出UnsupportedOperationException 。
- CopyOnWriteArrayList是synchronized List的并发替代，当迭代的次数超过突变次数时，CopyOnWriteArrayList可以提供更好的并发性。
- 它允许重复的元素和异构对象（使用泛型来获取编译时错误）。因为它每次创建迭代器时都会创建一个新的数组副本，所以performance is slower比ArrayList performance is slower 。

实例

```
CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<>(new Integer[] {1,2,3});

System.out.println(list);    //[1, 2, 3]

//Get iterator 1
Iterator<Integer> itr1 = list.iterator();

//Add one element and verify list is updated
list.add(4);

System.out.println(list);    //[1, 2, 3, 4]

//Get iterator 2
Iterator<Integer> itr2 = list.iterator();

System.out.println("====Verify Iterator 1 content====");

itr1.forEachRemaining(System.out :: println);    //1,2,3

System.out.println("====Verify Iterator 2 content====");

itr2.forEachRemaining(System.out :: println);    //1,2,3,4
```

主要方法

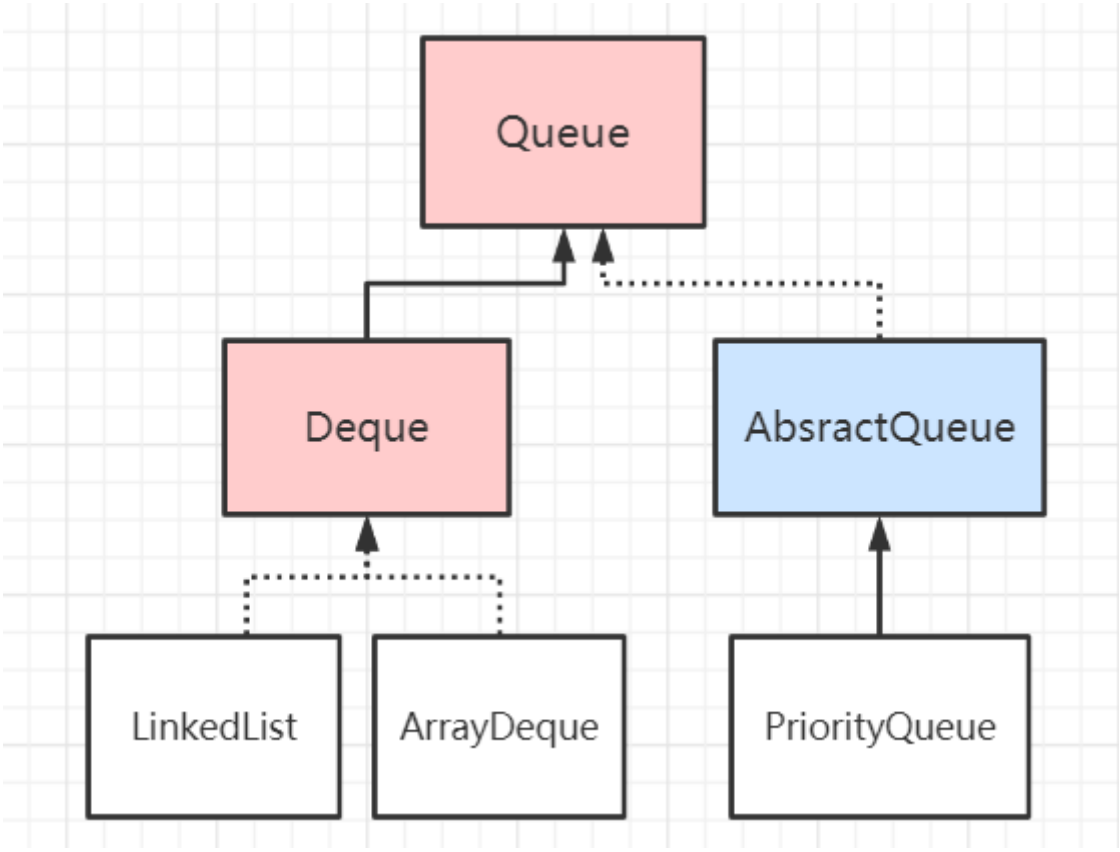
```
CopyOnWriteArrayList() ：创建一个空列表。
CopyOnWriteArrayList(Collection c) ：创建一个列表，该列表包含指定集合的••元素，并按集合的迭代器返回它们的顺序。
CopyOnWriteArrayList(object[] array) ：创建一个保存给定数组副本的列表。
boolean addIfAbsent(object o) ：如果不存在则追加元素。
int addAllAbsent(Collection c) ：以指定集合的••迭代器返回的顺序，将指定集合中尚未包含在此列表中的所有元素追加到此列表的末尾。
```

2. Queue

Queue介绍

主要方法

`Queue` 队列，在 JDK 中有两种不同类型的集合实现：**单向队列**（AbstractQueue）和 **双端队列**（Deque）



Queue 中提供了两套增加、删除元素的 API，当插入或删除元素失败时，会有**两种不同的失败处理策略。**

方法及失败策略	插入方法	删除方法	查找方法
抛出异常	add()	remove()	get()
返回失败默认值	offer()	poll()	peek()

选取哪种方法的决定因素：插入和删除元素失败时，希望 `抛出异常` 还是返回 `布尔值`

`add()` 和 `offer()` 对比：

在队列长度大小确定的场景下，队列放满元素后，添加下一个元素时，add() 会抛出 `IllegalStateException` 异常，而 `offer()` 会返回 `false` 。

但是它们两个方法在插入**某些不合法的元素**时都会抛出三个相同的异常。

```
* @throws ClassCastException if the class of the specified element
*         prevents it from being added to this queue
* @throws NullPointerException if the specified element is null and
*         this queue does not permit null elements
* @throws IllegalArgumentException if some property of this element
*         prevents it from being added to this queue
```

`remove()` 和 `poll()` 对比：

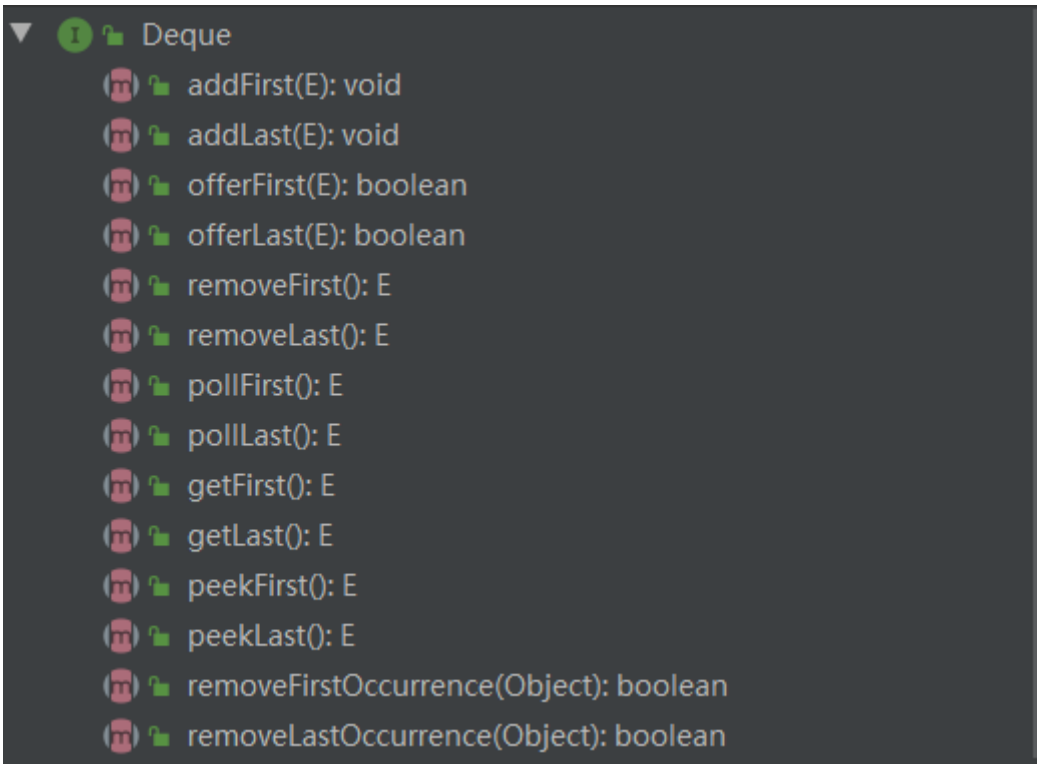
在**队列为空**的场景下， `remove()` 会抛出 `NoSuchElementException` 异常，而 `poll()` 则返回 `null` 。

`get()` 和 `peek()` 对比：

在队列为空的情况下， `get()` 会抛出 `NoSuchElementException` 异常，而 `peek()` 则返回 `null` 。

Deque 接口

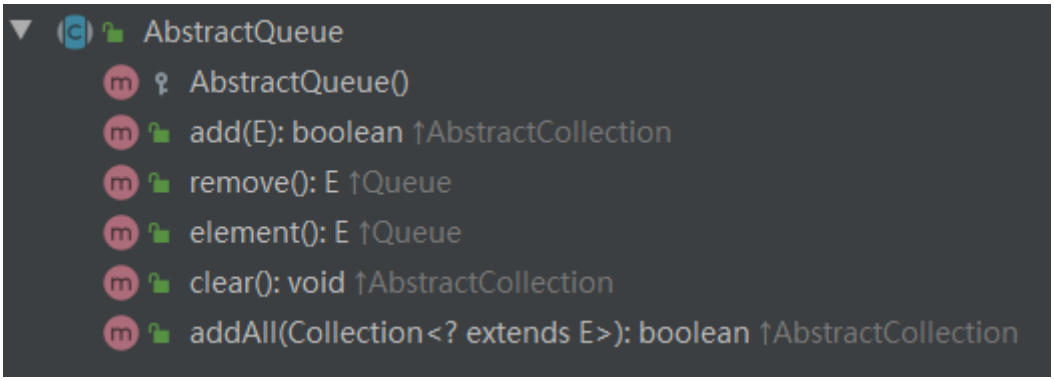
`Deque` 接口的实现非常好理解：从**单向**队列演变为**双向**队列，内部额外提供**双向队列的操作方法**即可：



Deque 接口额外提供了**针对队列的头结点和尾结点**操作的方法，而**插入、删除方法同样也提供了两套不同的失败策略**。除了 `add()` 和 `offer()` ， `remove()` 和 `poll()` 以外，还有 `get()` 和 `peek()` 出现了不同的策略

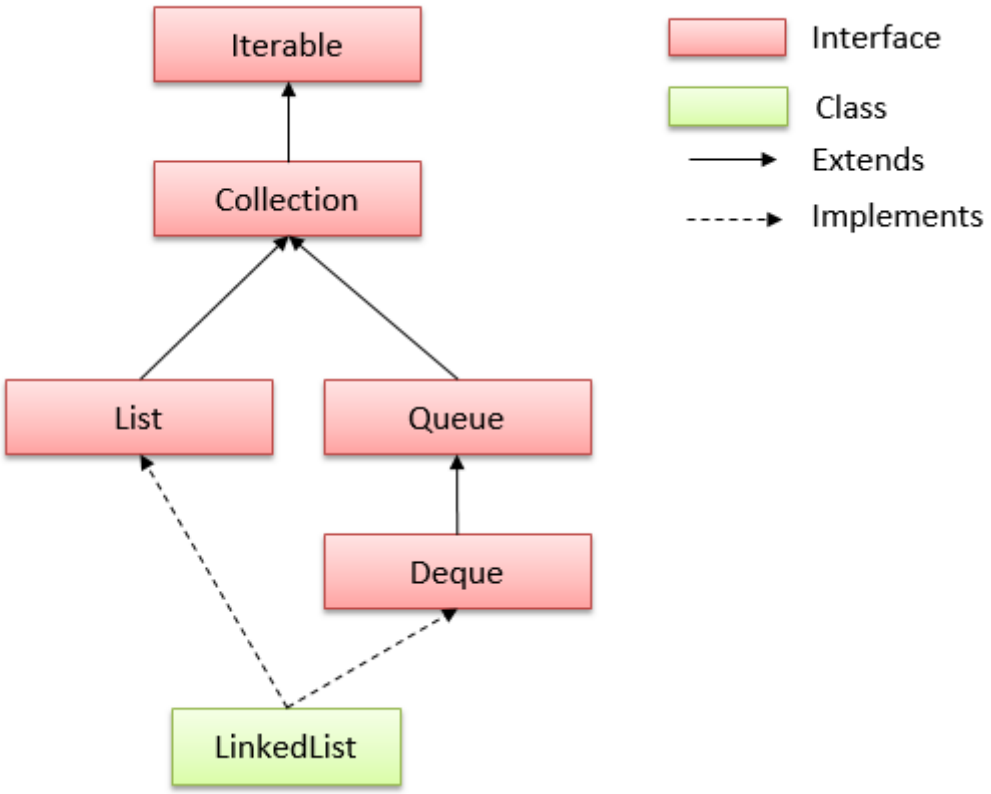
AbstractQueue 抽象类

AbstractQueue 类中提供了各个 API 的基本实现，主要针对各个不同的处理策略给出基本的方法实现，定义在这里的作用是让 `子类` 根据其 `方法规范` （操作失败时抛出异常还是返回默认值）实现具体的业务逻辑。



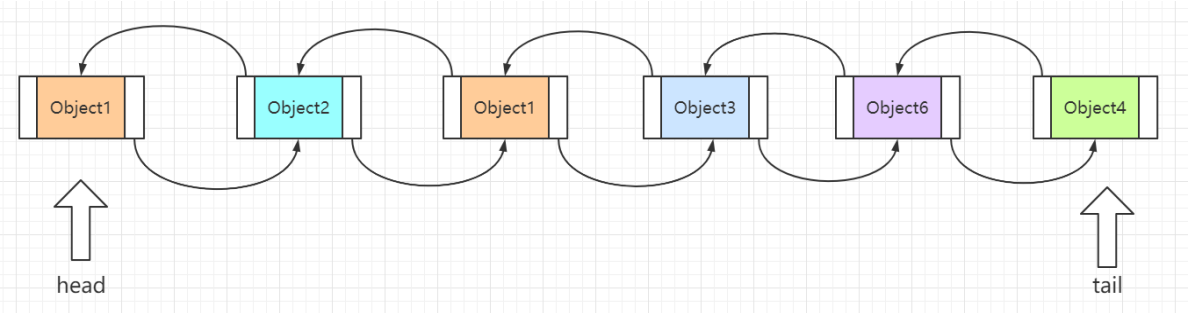
LinkedList

继承关系

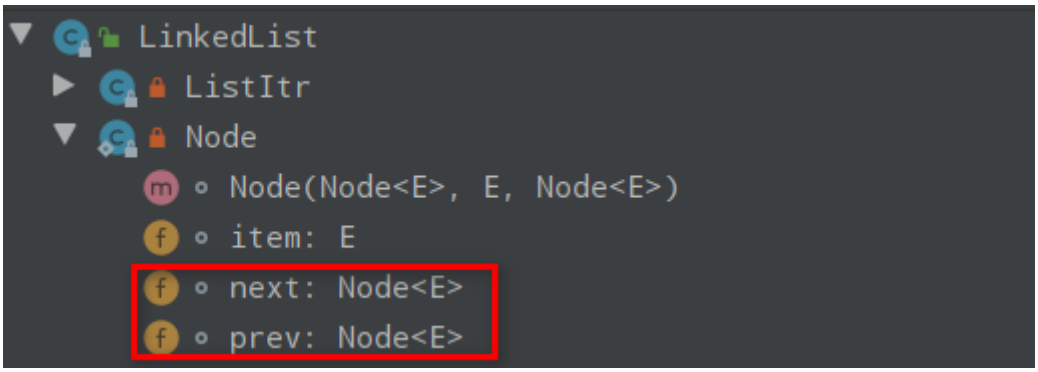


底层实现

LinkedList 底层采用 双向链表 数据结构存储元素，由于链表的内存地址 非连续 ，所以它不具备随机访问的特点，但由于它利用指针连接各个元素，所以插入、删除元素只需要 操作指针 ，不需要 移动元素 ，故具有**增删快、查询慢**的特点。它也是一个非线程安全的集合。



由于以双向链表作为数据结构，它是**线程不安全**的集合；存储的每个节点称为一个 `Node`，下图可以看到 Node 中保存了 `next` 和 `prev` 指针，`item` 是该节点的值。在插入和删除时，时间复杂度都保持为 `O(1)`

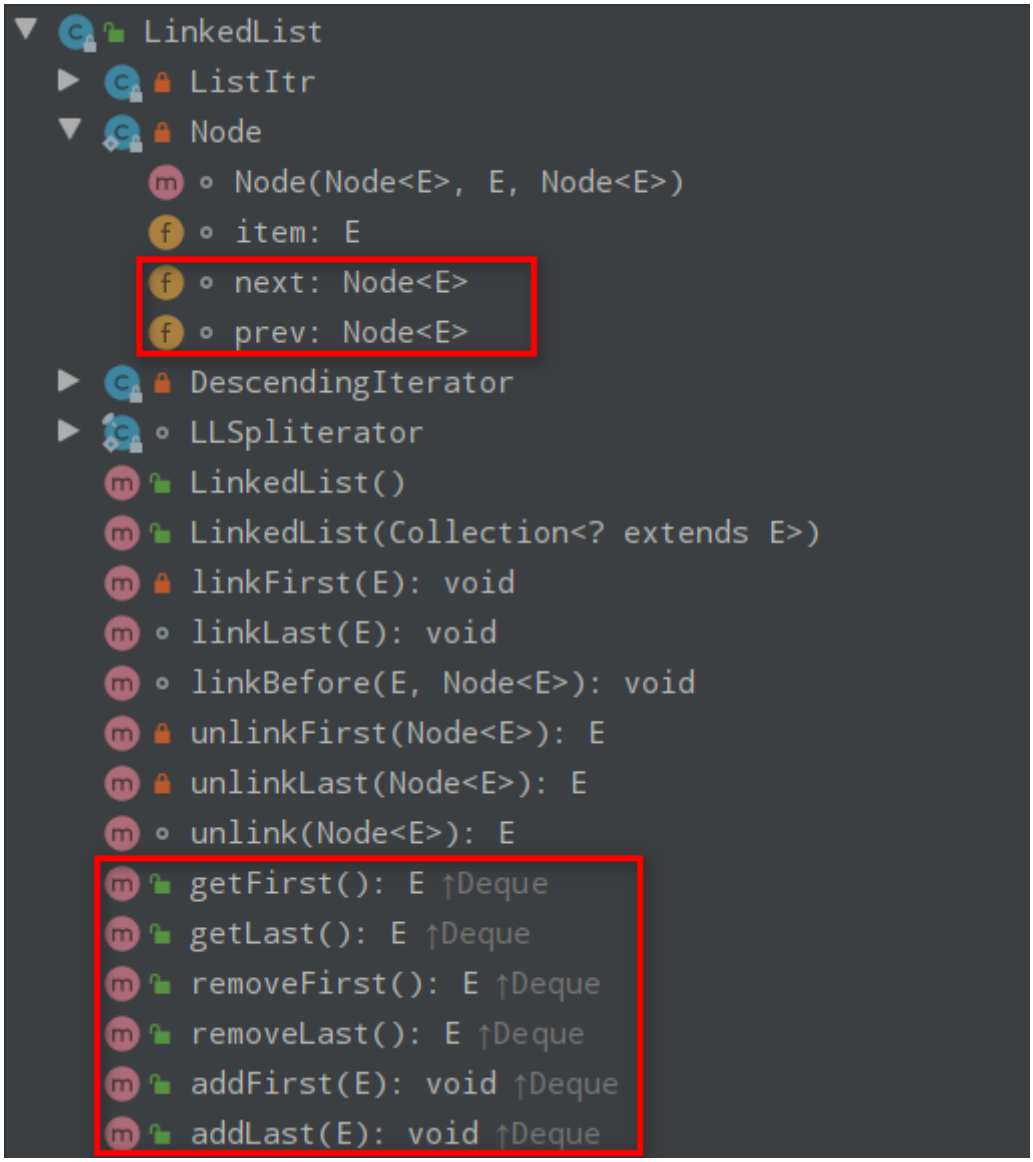


关于 LinkedList，除了它是以链表实现的集合外，还有一些特殊的特性需要注意的。

- 优势：LinkedList 底层没有 `扩容机制`，使用 `双向链表` 存储元素，所以插入和删除元素效率较高，适用于频繁操作元素的场景
- 劣势：LinkedList 不具备 `随机访问` 的特点，查找某个元素只能从 `head` 或 `tail` 指针一个一个比较，所以**查找中间的元素时效率很低**
- 查找优化：LinkedList 查找某个下标 `index` 的元素时**做了优化**，若 `index > (size / 2)`，则从 `head` 往后查找，否则从 `tail` 开始往前查找，代码如下所示：

```
1   LinkedList.Node<E> node(int index) {
2         LinkedList.Node x;
3         int i;
4         if (index < this.size >> 1) { // 查找的下标处于链表前半部分则从头找
5               x = this.first;
6               for(i = 0; i < index; ++i) { x = x.next; }
7               return x;
8         } else { // 查找的下标处于数组的后半部分则从尾开始找
9               x = this.last;
10              for(i = this.size - 1; i > index; --i) { x = x.prev; }
11              return x;
12        }
13  }
```

- 双端队列：使用双端链表实现，并且实现了 `Deque` 接口，使得 LinkedList 可以用作**双端队列**。下图可以看到 Node 是集合中的元素，提供了前驱指针和后继指针，还提供了一系列操作 `头结点` 和 `尾结点` 的方法，具有双端队列的特性。



LinkedList 集合最让人树枝的是它的链表结构，但是我们同时也要注意它是一个双端队列型的集合。

```
Deque<Object> deque = new LinkedList<>();
```

常用方法

```
LinkedList() ：初始化一个空的LinkedList实现。
LinkedListExample(Collection c) ：初始化一个LinkedList，该LinkedList包含指定集合的••元素，并按集合的迭代器返回它们的顺序。
boolean add(Object o) ：将指定的元素追加到列表的末尾。
void add（int index，Object element） ：将指定的元素插入列表中指定位置的索引处。
void addFirst(Object o) ：将给定元素插入列表的开头。
void addLast(Object o) ：将给定元素附加到列表的末尾。
int size() ：返回列表中的元素数
boolean contains(Object o) ：如果列表包含指定的元素，则返回true ，否则返回false 。
boolean remove(Object o) ：删除列表中指定元素的第一次出现。
Object getFirst() ：返回列表中的第一个元素。
Object getLast() ：返回列表中的最后一个元素。
int indexOf(Object o) ：返回指定元素首次出现的列表中的索引；如果列表不包含指定元素，则返回-1。
lastIndexOf(Object o) ：返回指定元素最后一次出现的列表中的索引；如果列表不包含指定元素，则返回-1。
Iterator iterator() ：以适当的顺序返回对该列表中的元素进行迭代的迭代器。
Object[] toArray() ：以正确的顺序返回包含此列表中所有元素的数组。
List subList（int fromIndex，int toIndex） ：返回此列表中指定的fromIndex（包括）和toIndex（不包括）之间的视图。
```

LinkedList与ArrayList

- ArrayList是使用动态可调整大小的数组的概念实现的。 而LinkedList是双向链表实现。
- ArrayList允许随机访问其元素，而LinkedList则不允许。
- LinkedList还实现了Queue接口，该接口添加了比ArrayList更多的方法，例如offer（），peek（），poll（）等。
- 与LinkedList相比， ArrayList添加和删除速度较慢，但在获取时却较快，因为如果LinkedList中的array已满，则无需调整数组大小并将内容复制到新数组。
- LinkedList比ArrayList具有更多的内存开销，因为在ArrayList中，每个索引仅保存实际对象，但是在LinkedList的情况下，每个节点都保存下一个和上一个节点的数据和地址。

ArrayDeque

使用**数组**实现的双端队列，它是**无界**的双端队列，最小的容量是 `8`（JDK 1.8）。在 JDK 11 看到它默认容量已经是 `16` 了。

```
/**
 * The minimum capacity that we'll use for a newly created deque.
 * Must be a power of 2.
 */
private static final int MIN_INITIAL_CAPACITY = 8;
```

`ArrayDeque` 在日常使用得不多，值得注意的是它与 `LinkedList` 的对比：`LinkedList` 采用**链表**实现双端队列，而 `ArrayDeque` 使用**数组**实现双端队列。

> 在文档中作者写到：**ArrayDeque 作为栈时比 Stack 性能好，作为队列时比 LinkedList 性能好**

由于双端队列**只能在头部和尾部**操作元素，所以删除元素和插入元素的时间复杂度大部分都稳定在 `O(1)`，除非在扩容时会涉及到元素的批量复制操作。但是在大多数情况下，使用它时应该指定一个大概的数组长度，避免频繁的扩容。

PriorityQueue

底层原理

PriorityQueue 基于**优先级堆实现**的优先级队列，而堆是采用**数组**实现：

```
Priority queue represented as a balanced binary heap: the two children of
queue[n] are queue[2*n+1] and queue[2*(n+1)]. The priority queue is ordered
by comparator, or by the elements' natural ordering, if comparator is null:
For each node n in the heap and each descendant d of n, n <= d. The element
with the lowest value is in queue[0], assuming the queue is nonempty.
transient Object[] queue; // non-private to simplify nested class access
```

文档中的描述告诉我们：该数组中的元素通过传入 `Comparator` 进行定制排序，如果不传入`Comparator` 时，则按照元素本身 `自然排序`，但要求元素实现了 `Comparable` 接口，所以 PriorityQueue **不允许存储 NULL 元素。**

PriorityQueue 应用场景：元素本身具有优先级，需要按照**优先级处理元素**

- 例如游戏中的VIP玩家与普通玩家，VIP 等级越高的玩家越先安排进入服务器玩耍，减少玩家流失。

```
1    public static void main(String[] args) {
2          Student vip1 = new Student("张三", 1);
3          Student vip3 = new Student("洪七", 2);
4          Student vip4 = new Student("老八", 4);
5          Student vip2 = new Student("李四", 1);
```

```
        Student normal1 = new Student("王五", 0);
        Student normal2 = new Student("赵六", 0);
        // 根据玩家的 VIP 等级进行降序排序
        PriorityQueue<Student> queue = new PriorityQueue<>((o1, o2) ->
    o2.getScore().compareTo(o1.getScore()));
        queue.add(vip1);queue.add(vip4);queue.add(vip3);
        queue.add(normal1);queue.add(normal2);queue.add(vip2);
        while (!queue.isEmpty()) {
            Student s1 = queue.poll();
            System.out.println(s1.getName() + "进入游戏; " + "VIP等级: " + s1.getScore());
        }
    }
    public static class Student implements Comparable<Student> {
        private String name;
        private Integer score;
        public Student(String name, Integer score) {
            this.name = name;
            this.score = score;
        }
        @Override
        public int compareTo(Student o) {
            return this.score.compareTo(o.getScore());
        }
    }
```

执行上面的代码可以得到下面这种有趣的结果，可以看到 氪金 使人带来快乐。

VIP 等级越高（优先级越高）就越优先安排进入游戏（优先处理），类似这种有优先级的场景还有非常多，各位可以发挥自己的想象力。

PriorityQueue 总结：

- PriorityQueue 是基于**优先级堆**实现的优先级队列，而堆是用**数组**维护的
- PriorityQueue 适用于**元素按优先级处理**的业务场景，例如用户在请求人工客服需要排队时，根据用户的**VIP等级**进行 插队 处理，等级越高，越先安排客服。

主要方法

```
boolean add(object) ：将指定的元素插入此优先级队列。
boolean offer(object) ：将指定的元素插入此优先级队列。
boolean remove(object) ：从此队列中移除指定元素的单个实例（如果存在）。
Object poll() ：检索并删除此队列的头部；如果此队列为空，则返回null。
Object element() ：获取但不删除此队列的头部，如果此队列为空，则抛出NoSuchElementException 。
Object peek() ：检索但不删除此队列的头部；如果此队列为空，则返回null。
void clear() ：从此优先级队列中删除所有元素。
Comparator comparator() ：返回用于对此队列中的元素进行排序的Comparator comparator()如果此队列是根据其元素的自然顺序排序的，则返回null。
boolean contains(Object o) ：如果此队列包含指定的元素，则返回true。
Iterator iterator() ：返回对该队列中的元素进行迭代的迭代器。
int size() ：返回此队列中的元素数。
Object[] toArray() ：返回一个包含此队列中所有元素的数组。
```

PriorityBlockingQueue

底层原理

Java PriorityBlockingQueue类是concurrent阻塞队列数据结构的实现，其中根据对象的priority对其进行处理。 名称的“阻塞”部分已添加，表示线程将阻塞等待，直到队列上有可用的项目为止 。

在priority blocking queue ，添加的对象根据其优先级进行排序。 默认情况下，优先级由对象的自然顺序决定。 队列构建时提供的Comparator器可以覆盖默认优先级。

- PriorityBlockingQueue是一个无界队列，并且会动态增长。 默认初始容量为'11'，可以在适当的构造函数中使用initialCapacity参数覆盖此初始容量。
- 它**提供了阻塞检索操作。**
- 它不允许使用NULL对象。
- 添加到PriorityBlockingQueue的对象必须具有可比性，否则它将引发ClassCastException 。
- 默认情况下，优先级队列的对象以自然顺序排序 。
- 比较器可用于队列中对象的自定义排序。
- 优先级队列的head是基于自然排序或基于比较器排序的least元素。 当我们轮询队列时，它从队列中返回头对象。
- 如果存在多个具有相同优先级的对象，则它可以随机轮询其中的任何一个。
- PriorityBlockingQueue是thread safe 。

主要方法

```
boolean add(object) ：将指定的元素插入此优先级队列。
boolean offer(object) ：将指定的元素插入此优先级队列。
boolean remove(object) ：从此队列中移除指定元素的单个实例（如果存在）。
Object poll() ：检索并删除此队列的头部，并在必要时等待指定的等待时间，以使元素可用。
Object poll(timeout, timeUnit) ：检索并删除此队列的头部，如果有必要，直到指定的等待时间，元素才可用。
Object take() ：检索并删除此队列的头部，如有必要，请等待直到元素可用。
void put(Object o) ：将指定的元素插入此优先级队列。
void clear() ：从此优先级队列中删除所有元素。
Comparator comparator() ：返回用于对此队列中的元素进行排序的Comparator comparator()如果此队列是根据其元素的自然顺序排序的，则返回null。
boolean contains(Object o) ：如果此队列包含指定的元素，则返回true。
Iterator iterator() ：返回对该队列中的元素进行迭代的迭代器。
int size() ：返回此队列中的元素数。
int drainTo(Collection c) ：从此队列中删除所有可用元素，并将它们添加到给定的collection中。
intrainToTo（Collection c，int maxElements） ：从此队列中最多移除给定数量的可用元素，并将它们添加到给定的collection中。
int remainingCapacity() Integer.MAX_VALUE int remainingCapacity() ：总是返回Integer.MAX_VALUE因为PriorityBlockingQueue不受容量限制。
```

```
Object[] toArray() ：返回一个包含此队列中所有元素的数组。
```

实例

```
import java.util.concurrent.PriorityBlockingQueue;
import java.util.concurrent.TimeUnit;

public class PriorityQueueExample
{
      public static void main(String[] args) throws InterruptedException
      {
            PriorityBlockingQueue<Integer> priorityBlockingQueue = new PriorityBlockingQueue<>();

            new Thread(() ->
            {
                System.out.println("Waiting to poll ...");

                try
                {
                      while(true)
                      {
                            Integer poll = priorityBlockingQueue.take();
                            System.out.println("Polled : " + poll);

                            Thread.sleep(TimeUnit.SECONDS.toMillis(1));
                      }

                } catch (InterruptedException e) {
                      e.printStackTrace();
                }

            }).start();

            Thread.sleep(TimeUnit.SECONDS.toMillis(2));
            priorityBlockingQueue.add(1);

            Thread.sleep(TimeUnit.SECONDS.toMillis(2));
            priorityBlockingQueue.add(2);

            Thread.sleep(TimeUnit.SECONDS.toMillis(2));
            priorityBlockingQueue.add(3);
      }
}
```

ArrayBlockingQueue

底层原理

ArrayBlockingQueue类是由数组支持的Java concurrent和bounded阻塞队列实现。 它对元素FIFO（先进先出）进行排序。

ArrayBlockingQueue的head是一直在队列中最长时间的那个元素。 ArrayBlockingQueue的tail是最短时间进入队列的元素。 新元素插入到队列的尾部， 并且队列检索操作在队列的开头获取元素 。

- ArrayBlockingQueue是由数组支持的固定大小的有界队列。
- 它对元素FIFO（先进先出）进行排序。
- 元素插入到尾部，并从队列的开头检索。
- 创建后，队列的容量无法更改。

- 它提供阻塞的插入和检索操作。
- 它不允许使用NULL对象。
- ArrayBlockingQueue是thread safe。
- 方法iterator()提供的Iterator按从第一个（头）到最后一个（尾部）的顺序遍历元素。
- ArrayBlockingQueue支持可选的fairness policy用于订购等待的生产者线程和使用者线程。 将fairness设置为true， 队列按FIFO顺序授予线程访问权限。

生产消费者实例

使用阻塞插入和检索从ArrayBlockingQueue中放入和取出元素的Java示例。

- 当队列已满时，生产者线程将等待。 一旦从队列中取出一个元素，它就会将该元素添加到队列中。
- 如果队列为空，使用者线程将等待。 队列中只有一个元素时，它将取出该元素。

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.TimeUnit;

public class ArrayBlockingQueueExample
{
      public static void main(String[] args) throws InterruptedException
     {
            ArrayBlockingQueue<Integer> priorityBlockingQueue = new ArrayBlockingQueue<>(5);

            //Producer thread
            new Thread(() ->
            {
                  int i = 0;
                  try
                  {
                        while (true)
                        {
                              priorityBlockingQueue.put(++i);
                              System.out.println("Added : " + i);

                              Thread.sleep(TimeUnit.SECONDS.toMillis(1));
                        }

                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }

            }).start();

            //Consumer thread
            new Thread(() ->
            {
                  try
                  {
                        while (true)
                        {
                              Integer poll = priorityBlockingQueue.take();
                              System.out.println("Polled : " + poll);

                              Thread.sleep(TimeUnit.SECONDS.toMillis(2));
                        }

```

```
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

            }).start();
        }
    }
```

主要方法

```
ArrayBlockingQueue(int capacity) ：构造具有给定（固定）容量和默认访问策略的空队列。
ArrayBlockingQueue（int capacity，boolean fair） ：构造具有给定（固定）容量和指定访问策略的空队列。 如果公允值为true ，则按FIFO顺序处理在插入或移除时阻塞的线程的队列访问； 如果为false，则未指定访问顺序。
ArrayBlockingQueue（int capacity，boolean fair，Collection c） ：构造一个队列，该队列具有给定（固定）的容量，指定的访问策略，并最初包含给定集合的元素，并以集合迭代器的遍历顺序添加。
void put(Object o) ：将指定的元素插入此队列的尾部，如果队列已满，则等待空间变为可用。
boolean add(object) : Inserts the specified element at the tail of this queue if it is possible to do so immediately without exceeding the queue’s capacity, returning true upon success and throwing an IllegalStateException if this queue is full.
boolean offer(object) ：如果可以在不超出队列容量的情况下立即执行此操作，则在此队列的尾部插入指定的元素，如果成功，则返回true，如果此队列已满，则抛出IllegalStateException。
boolean remove(object) ：从此队列中移除指定元素的单个实例（如果存在）。
Object peek() ：检索但不删除此队列的头部；如果此队列为空，则返回null。
Object poll() ：检索并删除此队列的头部；如果此队列为空，则返回null。
Object poll(timeout, timeUnit) ：检索并删除此队列的头部，如果有必要，直到指定的等待时间，元素才可用。
Object take() ：检索并删除此队列的头部，如有必要，请等待直到元素可用。
void clear() ：从队列中删除所有元素。
boolean contains(Object o) ：如果此队列包含指定的元素，则返回true。
Iterator iterator() ：以适当的顺序返回对该队列中的元素进行迭代的迭代器。
int size() ：返回此队列中的元素数。
int drainTo(Collection c) ：从此队列中删除所有可用元素，并将它们添加到给定的collection中。
intrainToTo（Collection c，int maxElements） ：从此队列中最多移除给定数量的可用元素，并将它们添加到给定的collection中。
int remainingCapacity() ：返回该队列理想情况下（在没有内存或资源限制的情况下）可以接受而不阻塞的其他元素的数量。
Object[] toArray() ：以适当的顺序返回一个包含此队列中所有元素的数组。
```

LinkedTransferQueue

底层原理

直接消息队列。也就是说，生产者生产后，必须等待消费者来消费才能继续执行。

Java TransferQueue是并发阻塞队列的实现，生产者可以在其中等待使用者使用消息。
LinkedTransferQueue类是Java中TransferQueue的实现。

- LinkedTransferQueue是链接节点上的unbounded队列。
- 此队列针对任何给定的生产者对元素FIFO（先进先出）进行排序。
- 元素插入到尾部，并从队列的开头检索。
- 它提供阻塞的插入和检索操作。
- 它不允许使用NULL对象。
- LinkedTransferQueue是thread safe。
- 由于异步性质，size（）方法不是固定时间操作，因此，如果在遍历期间修改此集合，则可能会报告不正确的结果。

- 不保证批量操作addAll，removeAll，retainAll，containsAll，equals和toArray是原子执行的。 例如，与addAll操作并发操作的迭代器可能仅查看某些添加的元素。

实例

非阻塞实例

```
LinkedTransferQueue<Integer> linkedTransferQueue = new LinkedTransferQueue<>();

linkedTransferQueue.put(1);

System.out.println("Added Message = 1");

Integer message = linkedTransferQueue.poll();

System.out.println("Recieved Message = " + message);
```

阻塞插入实例，用于现成状态同步通信
使用阻塞插入和检索从LinkedTransferQueue放入和取出元素的Java示例。

- 生产者线程将等待，直到有消费者准备从队列中取出项目为止。
- 如果队列为空，使用者线程将等待。 队列中只有一个元素时，它将取出该元素。 只有在消费者接受了消息之后，生产者才可以再发送一条消息。

```
import java.util.Random;
import java.util.concurrent.LinkedTransferQueue;
import java.util.concurrent.TimeUnit;

public class LinkedTransferQueueExample
{
      public static void main(String[] args) throws InterruptedException
     {
            LinkedTransferQueue<Integer> linkedTransferQueue = new LinkedTransferQueue<>();

            new Thread(() ->
           {
                 Random random = new Random(1);
                 try
                {
                       while (true)
                      {
                            System.out.println("Producer is waiting to transfer message...");

                            Integer message = random.nextInt();
                            boolean added = linkedTransferQueue.tryTransfer(message);
                            if(added) {
                                  System.out.println("Producer added the message - " + message);
                           }
                            Thread.sleep(TimeUnit.SECONDS.toMillis(3));
                      }

                } catch (InterruptedException e) {
                       e.printStackTrace();
                }

```

```
            }).start();

             new Thread(() ->
            {
                  try
                 {
                        while (true)
                       {
                              System.out.println("Consumer is waiting to take message...");

                              Integer message = linkedTransferQueue.take();

                              System.out.println("Consumer recieved the message - " + message);

                              Thread.sleep(TimeUnit.SECONDS.toMillis(3));
                       }

                 } catch (InterruptedException e) {
                        e.printStackTrace();
                 }

            }).start();
      }
}
```

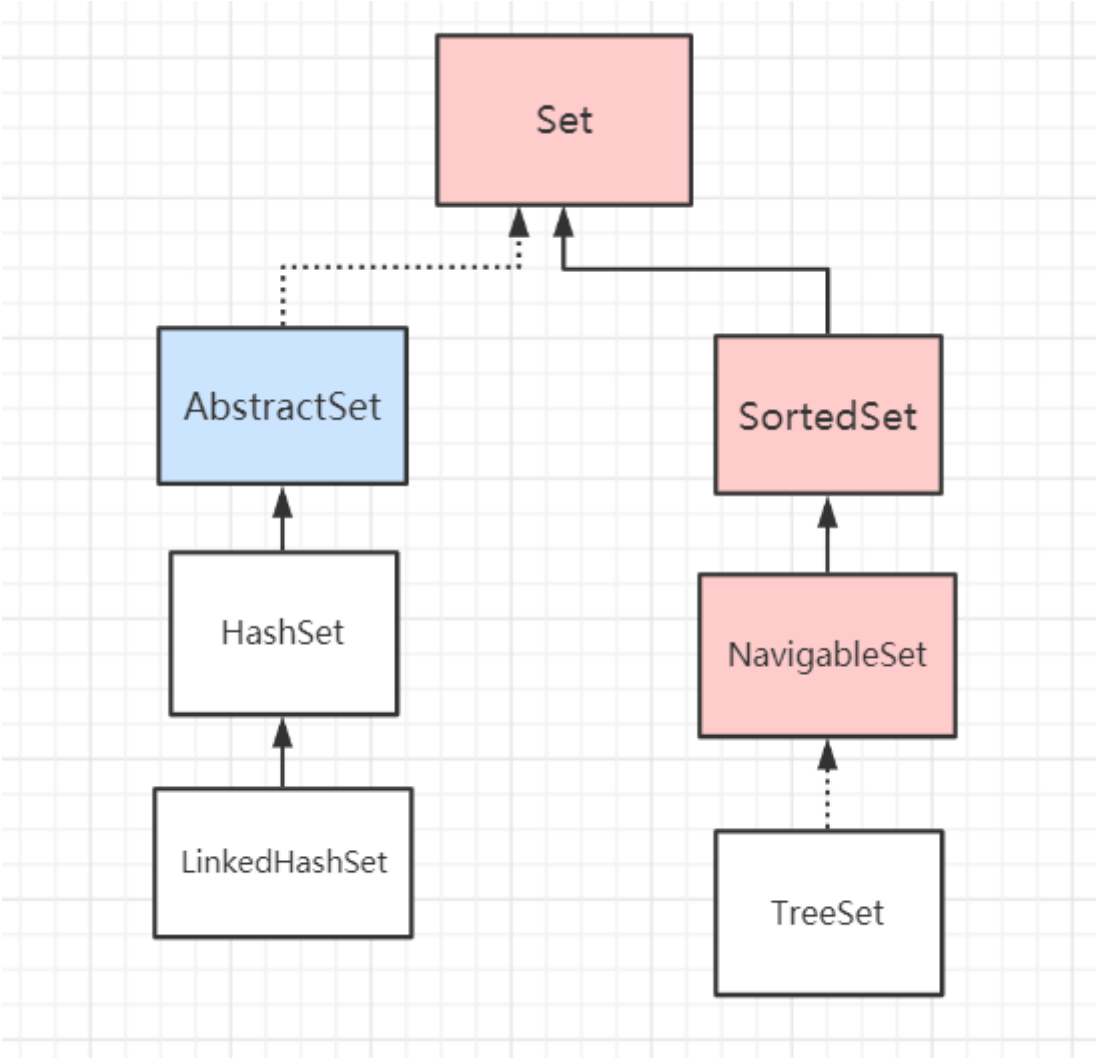
主要方法

```
LinkedTransferQueue() : 构造一个初始为空的LinkedTransferQueue。
LinkedTransferQueue(Collection c) : 构造一个LinkedTransferQueue，最初包含给定集合的元素，并以该集合的迭代器的遍历顺序添加。
Object take() : 检索并删除此队列的头部，如有必要，请等待直到元素可用。
void transfer(Object o) : 将元素传输给使用者，如有必要，请等待。
boolean tryTransfer(Object o) : 如果可能，立即将元素传输到等待的使用者。
boolean tryTransfer（Object o，long timeout，TimeUnit unit） : 如果有可能，则在超时之前将元素传输给使用者。
int getWaitingConsumerCount() : 返回等待通过BlockingQueue.take（）或定时轮询接收元素的使用者数量的估计值。
boolean hasWaitingConsumer() : 如果至少有一个使用者正在等待通过BlockingQueue.take（）或定时轮询接收元素，则返回true。
void put(Object o) : 将指定的元素插入此队列的尾部。
boolean add(object) : Inserts the specified element at the tail of this queue.
boolean offer(object) : 将指定的元素插入此队列的尾部。
boolean remove(object) : 从此队列中移除指定元素的单个实例（如果存在）。
Object peek() : 检索但不删除此队列的头部；如果此队列为空，则返回null。
Object poll() : 检索并删除此队列的头部；如果此队列为空，则返回null。
Object poll(timeout, timeUnit) : 检索并删除此队列的头部，如果有必要，直到指定的等待时间，元素才可用。
void clear() : 从队列中删除所有元素。
boolean contains(Object o) : 如果此队列包含指定的元素，则返回true。
Iterator iterator() : 以适当的顺序返回对该队列中的元素进行迭代的迭代器。
int size() : 返回此队列中的元素数。
int drainTo(Collection c) : 从此队列中删除所有可用元素，并将它们添加到给定的collection中。
intrainToTo（Collection c，int maxElements） : 从此队列中最多移除给定数量的可用元素，并将它们添加到给定的collection中。
int remainingCapacity() : 返回该队列理想情况下（在没有内存或资源限制的情况下）可以接受而不阻塞的其他元素的数量。
Object[] toArray() : 以适当的顺序返回一个包含此队列中所有元素的数组。
```

3. Set 接口

Set接口

`Set` 接口继承了 `Collection` 接口，是一个不包括重复元素的集合，更确切地说，Set 中任意两个元素不会出现 `o1.equals(o2)`，而且 Set **至多**只能存储一个 `NULL` 值元素，Set 集合的组成部分可以用下面这张图概括：



在 Set 集合体系中，我们需要着重关注两点：

- 存入**可变元素**时，必须非常小心，因为任意时候元素状态的改变都有可能使得 Set 内部出现两个**相等**的元素，即 `o1.equals(o2) = true`，所以一般不要更改存入 Set 中的元素，否则将会破坏了 `equals()` 的作用！
- Set 的最大作用就是判重，在项目中最大的作用也是**判重**！

接下来我们去看它的实现类和子类： `AbstractSet` 和 `SortedSet`

AbstractSet 抽象类

`AbstractSet` 是一个实现 Set 的一个抽象类，定义在这里可以将所有具体 Set 集合的**相同行为**在这里实现，**避免子类包含大量的重复代码**

所有的 Set 也应该要有相同的 `hashCode()` 和 `equals()` 方法，所以使用抽象类把该方法重写后，子类无需关心这两个方法。

```
1    public abstract class AbstractSet<E> implements Set<E> {
```

```
        // 判断两个 set 是否相等
        public boolean equals(Object o) {
            if (o == this) { // 集合本身
                return true;
            } else if (!(o instanceof Set)) { // 集合不是 set
                return false;
            } else {
                // 比较两个集合的元素是否全部相同
            }
        }
        // 计算所有元素的 hashcode 总和
        public int hashCode() {
            int h = 0;
            Iterator i = this.iterator();
            while(i.hasNext()) {
                E obj = i.next();
                if (obj != null) {
                    h += obj.hashCode();
                }
            }
            return h;
        }
    }
```

SortedSet 接口

`SortedSet` 是一个接口，它在 Set 的基础上扩展了**排序**的行为，所以所有实现它的子类都会拥有排序功能。

```
public interface SortedSet<E> extends Set<E> {
    // 元素的比较器,决定元素的排列顺序
    Comparator<? super E> comparator();
    // 获取 [var1, var2] 之间的 set
    SortedSet<E> subSet(E var1, E var2);
    // 获取以 var1 开头的 Set
    SortedSet<E> headSet(E var1);
    // 获取以 var1 结尾的 Set
    SortedSet<E> tailSet(E var1);
    // 获取首个元素
    E first();
    // 获取最后一个元素
    E last();
}
```

HashSet

底层实现

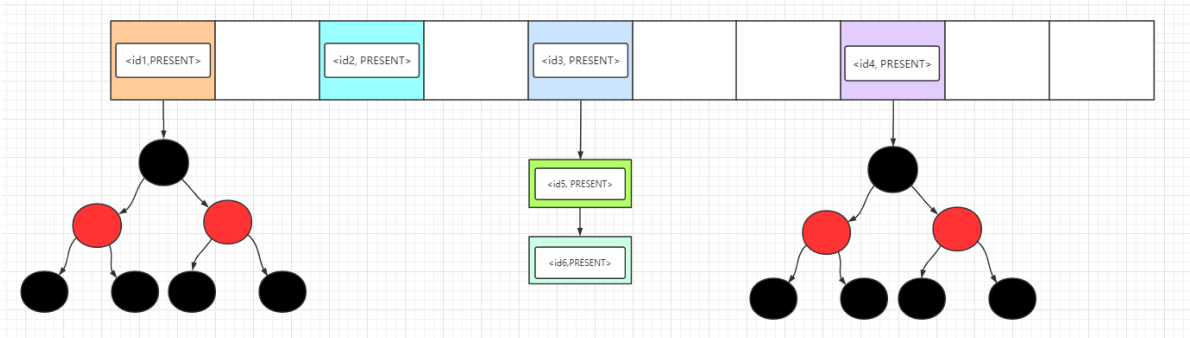
HashSet 底层借助 `HashMap` 实现，我们可以观察它的多个构造方法，本质上都是 new 一个 HashMap

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable {
      public HashSet() {
            this.map = new HashMap();
      }
      public HashSet(int initialCapacity, float loadFactor) {
            this.map = new HashMap(initialCapacity, loadFactor);
      }
      public HashSet(int initialCapacity) {
            this.map = new HashMap(initialCapacity);
      }
}
```

我们可以观察 `add()` 方法和 `remove()` 方法是如何将 HashSet 的操作嫁接到 HashMap 的。

```
private static final Object PRESENT = new Object();

public boolean add(E e) {
      return this.map.put(e, PRESENT) == null;
}
public boolean remove(Object o) {
            return this.map.remove(o) == PRESENT;
}
```

我们看到 `PRESENT` 就是一个**静态常量**：使用 PRESENT 作为 HashMap 的 value 值，使用HashSet的开发者只需**关注**于需要插入的 `key`，**屏蔽**了 HashMap 的 `value`



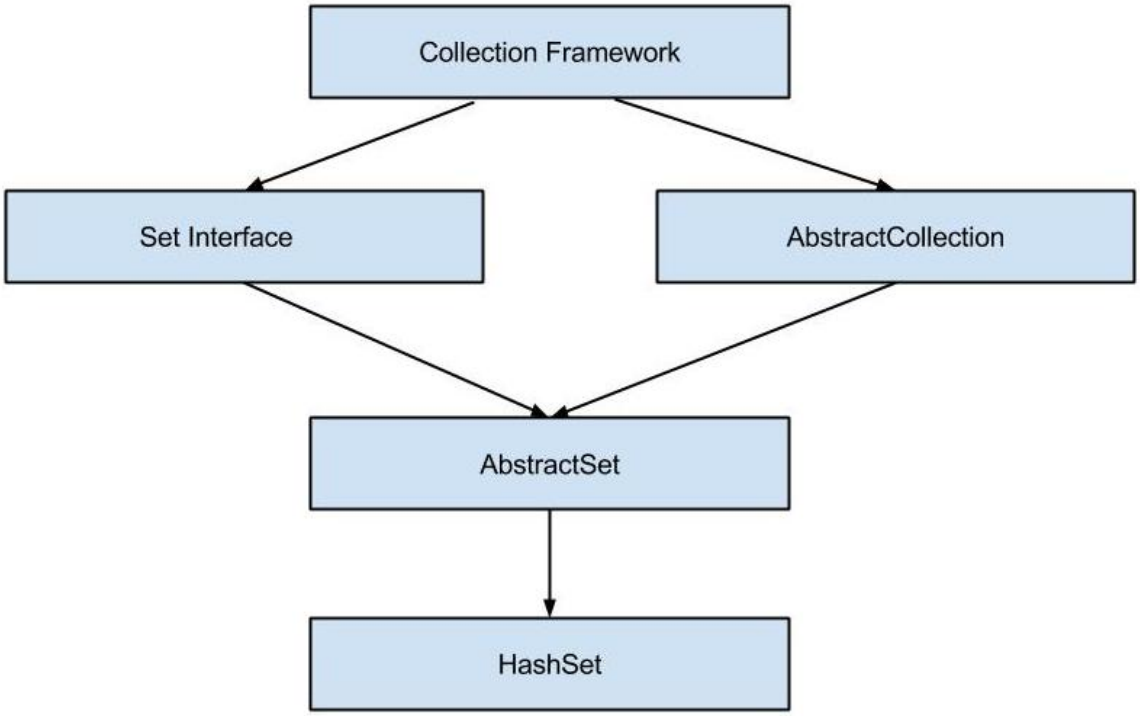
上图可以观察到每个 `Entry` 的 `value` 都是 PRESENT 空对象，我们就不用再理会它了。

HashSet 在 HashMap 基础上实现，所以很多地方可以联系到 HashMap：

- 底层数据结构：HashSet 也是采用 `数组 + 链表 + 红黑树` 实现
- 线程安全性：由于采用 HashMap 实现，而 HashMap 本身线程不安全，在HashSet 中没有添加额外的同步策略，所以 HashSet 也**线程不安全**
- 存入 HashSet 的对象的状态**最好不要发生变化**，因为有可能改变状态后，在集合内部出现两个元素 `o1.equals(o2)`，破坏了 `equals()` 的语义。

继承结构

- 实现了Set接口。
- HashSet中不允许重复的值。
- HashSet中允许一个NULL元素。
- 无序集合，并且不保证集合的迭代顺序。
- 为基本操作（添加，删除，包含和调整大小）提供恒定的时间性能。
- HashSet不同步。 如果多个线程同时访问哈希集，并且至少有一个线程修改了哈希集，则必须在外部对其进行同步。使用Collections.synchronizedSet(new HashSet())方法获取同步的哈希集。



基本使用

- 默认初始容量为16。 我们可以通过在构造函数HashSet(int initialCapacity)传递默认容量来覆盖此默认容量。

```
public boolean add（E e） ：如果指定的元素不存在，则将其添加到Set中。 此方法在内部使用equals()方法检查重复项。 如果元素重复，则元素被拒绝，并且不替换值。
public void clear() ：从哈希集中删除所有元素。
public boolean contains（Object o） ：如果哈希集包含指定的元素othrweise false ，则返回false 。
public boolean isEmpty() ：如果哈希集不包含任何元素，则返回true ，否则返回false 。
public int size() ：返回哈希集中的元素数量。
public Iterator<E> iterator() ：返回对此哈希集中的元素的迭代器。 从迭代器返回的元素没有特定的顺序。
public boolean remove（Object o） ：从哈希集中删除指定的元素（如果存在）并返回true ，否则返回false 。
public boolean removeAll（Collection <? > c） ：删除哈希集中属于指定集合的••所有元素。
public Object clone() ：返回哈希集的浅表副本。
public Spliterator<E> spliterator() ：在此哈希集中的元素上创建后绑定和故障快速的Spliterator。
```

LinkedHashSet

底层原理

LinkedHashSet 的代码少的可怜，不信我给你我粘出来

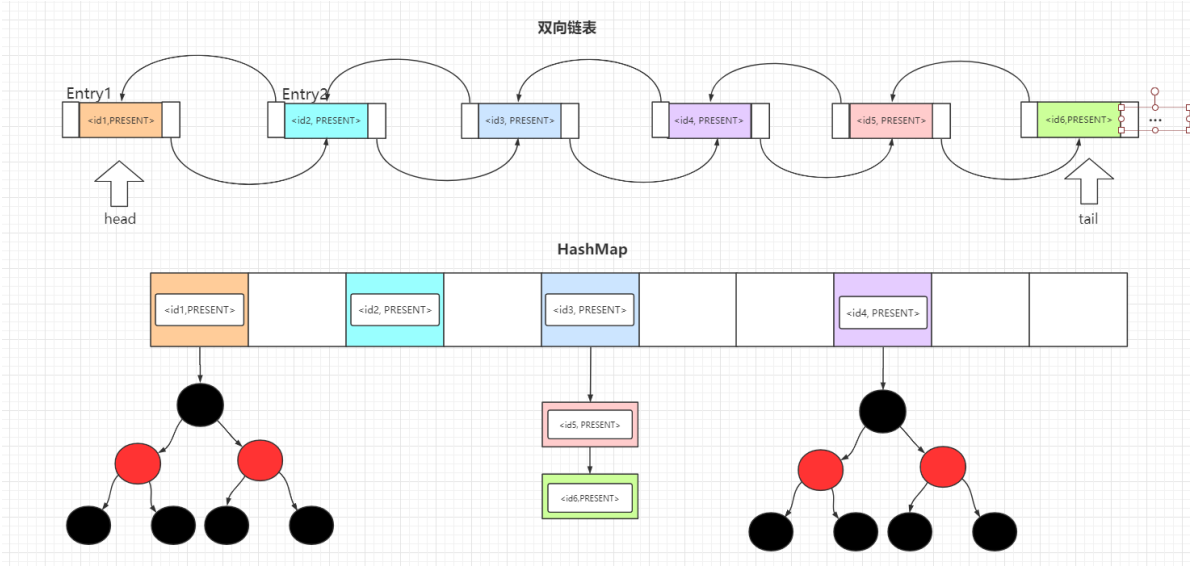
```java
package java.util;

import java.io.Serializable;

public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, Serializable {
    private static final long serialVersionUID = -2851667679971038690L;

    public LinkedHashSet(int initialCapacity, float loadFactor) { super(initialCapacity, loadFactor,  dummy: true); }

    public LinkedHashSet(int initialCapacity) { super(initialCapacity,  loadFactor: 0.75F,  dummy: true); }

    public LinkedHashSet() { super( initialCapacity: 16,  loadFactor: 0.75F,  dummy: true); }

    public LinkedHashSet( @NotNull()  @Flow(sourceIsContainer = true, targetIsContainer = true) Collection<? extends E> c) {
        super(Math.max(2 * c.size(), 11),  loadFactor: 0.75F,  dummy: true);
        this.addAll(c);
    }

    public Spliterator<E> spliterator() { return Spliterators.spliterator( c: this,  characteristics: 17); }
}
```

少归少，还是不能闹，`LinkedHashSet` 继承了 `HashSet`，我们跟随到父类 HashSet 的构造方法看看

```java
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
      this.map = new LinkedHashMap(initialCapacity, loadFactor);
}
```

发现父类中 map 的实现采用 `LinkedHashMap`，这里注意不是 `HashMap`，而 LinkedHashMap 底层又采用 HashMap + 双向链表 实现的，所以本质上 LinkedHashSet 还是使用 HashMap 实现的。

> LinkedHashSet -> LinkedHashMap -> HashMap + 双向链表

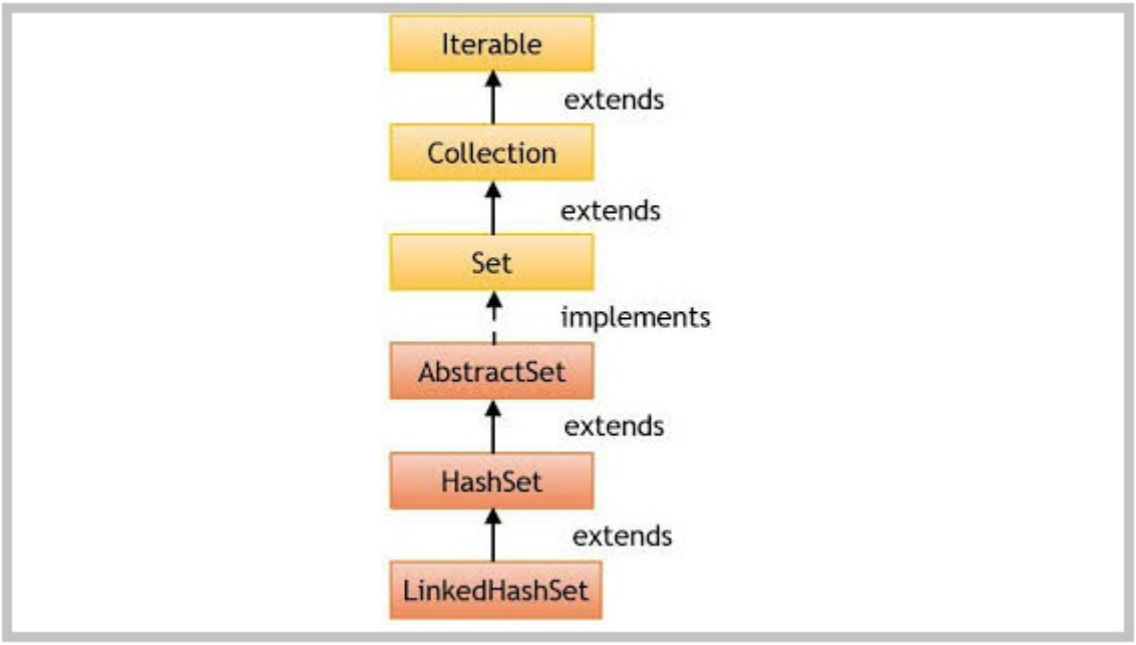


而 LinkedHashMap 是采用 `HashMap` 和 `双向链表` 实现的，这条双向链表中保存了元素的插入顺序。所以 LinkedHashSet 可以按照元素的插入顺序遍历元素，如果你熟悉 `LinkedHashMap`，那 LinkedHashSet 也就更不在话下了。

关于 LinkedHashSet 需要注意几个地方：

- 它继承了 `HashSet`，而 HashSet 默认是采用 HashMap 存储数据的，但是 LinkedHashSet 调用父类构造方法初始化 map 时是 LinkedHashMap 而不是 HashMap，这个要额外注意一下
- 由于 LinkedHashMap 不是线程安全的，且在 LinkedHashSet 中没有添加额外的同步策略，所以 LinkedHashSet 集合**也不是线程安全**的

继承关系



- 它扩展了HashSet类，后者扩展了AbstractSet类。
- 它实现Set接口。
- LinkedHashSet中不允许重复的值。
- LinkedHashSet中允许一个NULL元素。
- 它是一个ordered collection，它是元素插入到集合中insertion-order （insertion-order）。
- 像HashSet一样，此类为基本操作（添加，删除，包含和调整大小）提供constant time performance。
- LinkedHashSet not synchronized。 如果多个线程同时访问哈希集，并且至少有一个线程修改了哈希集，则必须在外部对其进行同步。
- 使用Collections.synchronizedSet(new LinkedHashSet())方法来获取同步的LinkedHashSet。

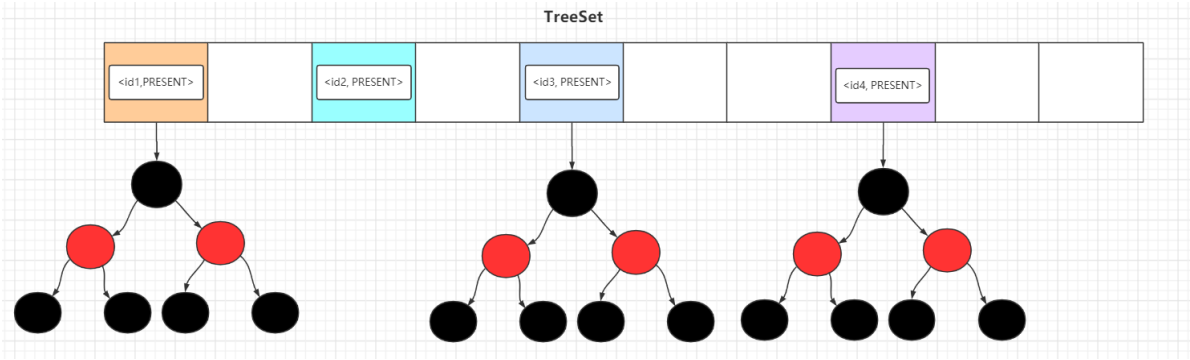
使用方法

```
public boolean add（E e） ：如果指定的元素不存在，则将其添加到Set中。 此方法在内部使用equals()方法检查重复项。 如果元素重复，则元素被拒绝，并且不替换值。
public void clear() ：从LinkedHashSet中删除所有元素。
public boolean contains（Object o） ：如果LinkedHashSet包含指定的元素othrweise false ，则返回false 。
public boolean isEmpty() ：如果LinkedHashSet不包含任何元素，则返回true ，否则返回false 。
public int size() ：返回LinkedHashSet中的元素数。
public Iterator<E> iterator() ：返回对此LinkedHashSet中的元素的迭代器。 从迭代器返回的元素没有特定的顺序。
public boolean remove（Object o） ：从LinkedHashSet中移除指定的元素（如果存在）并返回true ，否则返回false 。
public boolean removeAll（Collection <? > c） ：删除LinkedHashSet中属于指定集合的••所有元素。
public Object clone() ：返回LinkedHashSet的浅表副本。
public Spliterator<E> spliterator() ：在此LinkedHashSet中的元素上创建后绑定和故障快速的Spliterator。 它具有以下初始化属性Spliterator.DISTINCT ， Spliterator.ORDERED 。
```

TreeSet

底层原理

TreeSet 是基于 TreeMap 的实现，所以存储的元素是**有序**的，底层的数据结构是 `数组 + 红黑树`。



而元素的排列顺序有 2 种，和 TreeMap 相同：自然排序和定制排序，常用的构造方法已经在下面展示出来了，TreeSet 默认按照自然排序，如果需要定制排序，需要传入 `Comparator`。

```
public TreeSet() {
      this(new TreeMap<E,Object>());
}
public TreeSet(Comparator<? super E> comparator) {
      this(new TreeMap<>(comparator));
}
```

TreeSet 应用场景有很多，像在游戏里的玩家战斗力排行榜

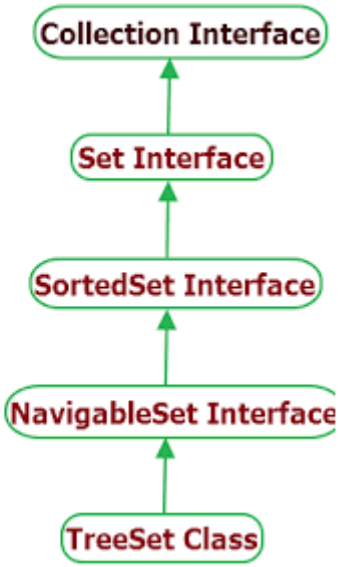
```
 public class Player implements Comparable<Integer> {
       public String name;
       public int score;
       @Override
       public int compareTo(Student o) {
             return Integer.compareTo(this.score, o.score);
       }
}
public static void main(String[] args) {
       Player s1 = new Player("张三", 100);
       Player s2 = new Player("李四", 90);
       Player s3 = new Player("王五", 80);
       TreeSet<Player> set = new TreeSet();
       set.add(s2); set.add(s1); set.add(s3);
       System.out.println(set);
}
// [Student{name='王五', score=80}, Student{name='李四', score=90}, Student{name='张三', score=100}]
```

对 TreeSet 介绍了它的主要实现方式和应用场景，有几个值得注意的点。

- TreeSet 的所有操作都会转换为对 TreeMap 的操作，TreeMap 采用**红黑树**实现，任意操作的平均时间复杂度为 `O(logN)`
- TreeSet 是一个**线程不安全**的集合
- TreeSet 常应用于对**不重复**的元素**定制排序**，例如玩家战力排行榜

> 注意:TreeSet判断元素是否重复的方法是判断compareTo()方法是否返回0，而不是调用 hashcode() 和 equals() 方法，如果返回 0 则认为集合内已经存在相同的元素，不会再加入到集合当中。

继承关系



- 它扩展了AbstractSet类，该类扩展了AbstractCollection类。
- 它实现了NavigableSet接口，该接口扩展了SortedSet接口。
- TreeSet中不允许重复的值。
- 在TreeSet中不允许NULL。
- 它是一个ordered collection，按排序顺序存储元素。
- 与HashSet一样，此类为基本操作（添加，删除，包含和调整大小）提供恒定的时间性能。
- TreeSet不允许插入异构对象，因为它必须比较对象以确定排序顺序。
- TreeSet不synchronized。 如果多个线程同时访问哈希集，并且至少有一个线程修改了哈希集，则必须在外部对其进行同步。
- 使用Collections.synchronizedSortedSet(new TreeSet())方法获取同步的TreeSet。

使用方法

```
boolean add(E e) ：将指定的元素添加到Set中（如果尚不存在）。
Comparator comparator() ：返回用于对该集合中的元素进行排序的Comparator comparator()如果此集合使用其元素的自然排序，则返回null。
Object first() ：返回当前在此集合中的第一个（最低）元素。
Object last() ：返回当前在此集合中的最后一个（最大）元素。
void clear() ：从TreeSet中删除所有元素。
boolean contains(Object o) ：如果TreeSet包含指定的元素，则返回true否则返回false 。
boolean isEmpty() ：如果TreeSet不包含任何元素，则返回true ，否则返回false 。
int size() ：返回TreeSet中的元素数。
Iterator<E> iterator() ：以ascending order返回此集合中元素的迭代器。
Iterator<E> descendingIterator() ：以Iterator<E> descendingIterator()返回此集合中元素的迭代器。
NavigableSet<E> descendingSet() ：返回此集合中包含的元素的逆序视图。
boolean remove(Object o) ：从TreeSet中移除指定的元素（如果存在）并返回true ，否则返回false 。
Object clone() ：返回TreeSet的浅表副本。
Spliterator<E> spliterator() ：在此TreeSet中的元素上创建后绑定和故障快速的Spliterator。 它与树集提供的顺序相同。
```

CopyOnWriteArraySet

底层原理

HashSet的thread-safe变体，它对所有操作都使用基础CopyOnWriteArrayList

与CopyOnWriteArrayList相似，它的immutable snapshot样式iterator方法在创建iterator使用对数组状态（在后备列表内）的引用。 这在遍历操作远远超过集合更新操作且我们不想同步遍历并且在更新集合时仍希望线程安全的用例中很有用。

- 作为正常设置的数据结构，它不允许重复。
- CopyOnWriteArraySet类实现Serializable接口并扩展AbstractSet类。
- 使用CopyOnWriteArraySet进行更新操作成本很高，因为每个突变都会创建基础数组的克隆副本并向其添加/更新元素。
- 它是HashSet的线程安全版本。 每个访问该集合的线程在初始化此集合的迭代器时都会看到自己创建的后备阵列快照版本。
- 因为它在创建迭代器时获取基础数组的快照，所以它不会抛出ConcurrentModificationException 。不支持迭代器上的变异操作。 这些方法抛出UnsupportedOperationException 。
- CopyOnWriteArraySet是synchronized Set的并发替代，当迭代的次数超过突变次数时，CopyOnWriteArraySet提供更好的并发性。
- 它允许重复的元素和异构对象（使用泛型来获取编译时错误）。
- 由于每次创建迭代器时都会创建基础数组的新副本，因此performance is slower HashSet

主要方法

```
CopyOnWriteArraySet() ：创建一个空集。
CopyOnWriteArraySet(Collection c) ：创建一个包含指定集合元素的集合，其顺序由集合的迭代器返回。
boolean add(object o) ：将指定的元素添加到此集合（如果尚不存在）。
boolean addAll(collection c) ：将指定集合中的所有元素（如果尚不存在boolean addAll(collection c)添加到此集合中。
void clear() ：从此集合中删除所有元素。
boolean contains(Object o) ：如果此集合包含指定的元素，则返回true。
boolean isEmpty() ：如果此集合不包含任何元素，则返回true。
Iterator iterator() ：以添加这些元素的顺序在此集合中包含的元素上返回一个迭代器。
boolean remove(Object o) ：从指定的集合中删除指定的元素（如果存在）。
int size() ：返回此集合中的元素数
```

实例

```
CopyOnWriteArraySet<Integer> set = new CopyOnWriteArraySet<>(Arrays.asList(1,2,3));

System.out.println(set);      //[1, 2, 3]

//Get iterator 1
Iterator<Integer> itr1 = set.iterator();

//Add one element and verify set is updated
set.add(4);
System.out.println(set);      //[1, 2, 3, 4]

//Get iterator 2
Iterator<Integer> itr2 = set.iterator();

System.out.println("====Verify Iterator 1 content====");

itr1.forEachRemaining(System.out :: println);    //1,2,3

```

```
19    System.out.println("====Verify Iterator 2 content====");
20
21    itr2.forEachRemaining(System.out :: println);     //1,2,3,4
```

4. Map

Map 集合体系详解

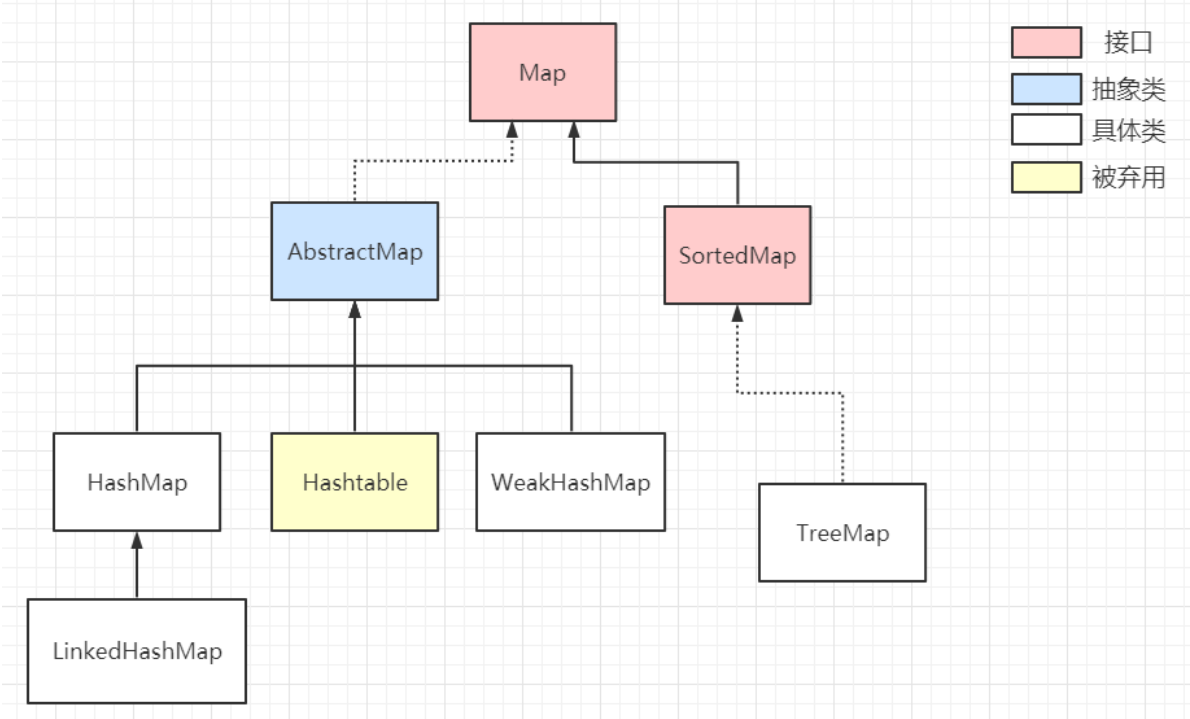
`Map` 接口是由 `<key, value>` 组成的集合，由 `key` 映射到**唯一**的 `value`，所以 `Map` 不能包含重复的 `key`，每个键**至多**映射一个值。下图是整个 Map 集合体系的主要组成部分，我将会按照日常使用频率从高到低一一讲解。

不得不提的是 Map 的设计理念：**定位元素**的时间复杂度优化到 `O(1)`

Map 体系下主要分为 AbstractMap 和 SortedMap两类集合

`AbstractMap` 是对 Map 接口的扩展，它定义了普通的 Map 集合具有的**通用行为**，可以避免子类重复编写大量相同的代码，子类继承 AbstractMap 后可以重写它的方法，**实现额外的逻辑**，对外提供更多的功能。

`SortedMap` 定义了该类 Map 具有 `排序` 行为，同时它在内部定义好有关排序的抽象方法，当子类实现它时，必须重写所有方法，对外提供排序功能。

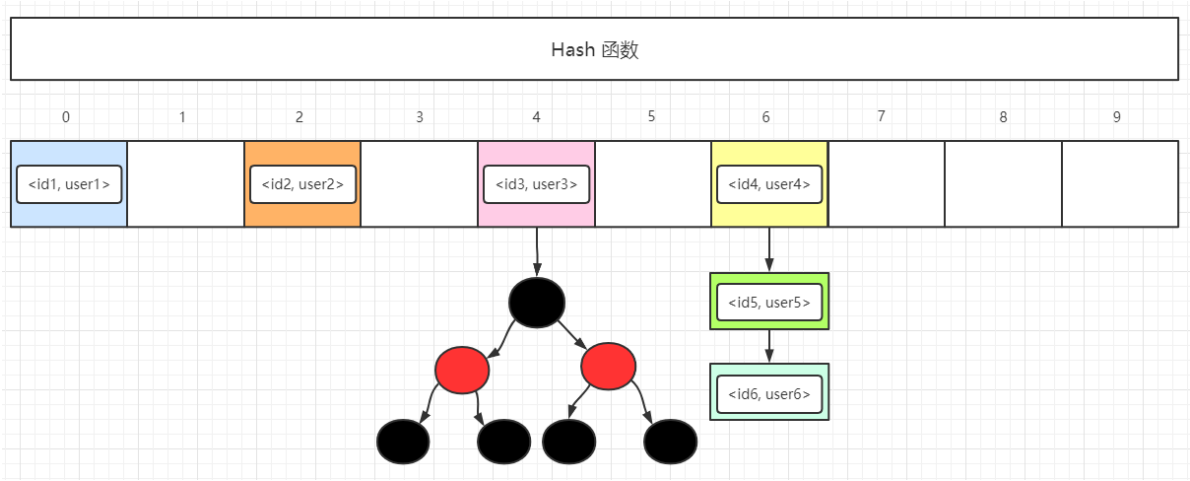


HashMap

底层原理

HashMap 是一个**最通用的**利用哈希表存储元素的集合，将元素放入 HashMap 时，将 `key` 的哈希值转换为数组的 `索引` 下标**确定存放位置**，查找时，根据 `key` 的哈希地址转换成数组的 `索引` 下标**确定查找位置**。

HashMap 底层是用数组 + 链表 + 红黑树这三种数据结构实现，它是**非线程安全**的集合。

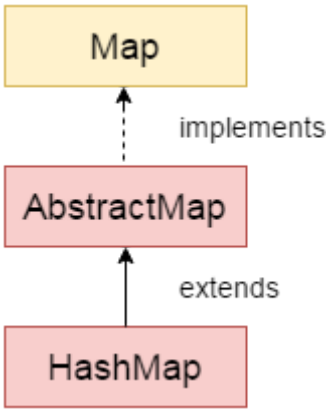


发送哈希冲突时，HashMap 的解决方法是将相同映射地址的元素连成一条 `链表`，如果链表的长度大于 `8` 时，且数组的长度大于 `64` 则会转换成 `红黑树` 数据结构。

关于 HashMap 的简要总结：

1. 它是集合中最常用的 `Map` 集合类型，底层由 `数组 + 链表 + 红黑树` 组成
2. HashMap不是线程安全的
3. 插入元素时，通过计算元素的 `哈希值`，通过**哈希映射函数**转换为 `数组下标`；查找元素时，同样通过哈希映射函数得到数组下标 `定位元素的位置`

继承关系



- HashMap不能包含重复的键。
- HashMap允许多个null值，但只允许一个null键。
- HashMap是一个unordered collection 。 它不保证元素的任何特定顺序。
- HashMap not thread-safe 。 您必须显式同步对HashMap的并发修改。 或者，您可以使用Collections.synchronizedMap(hashMap)来获取HashMap的同步版本。
- 只能使用关联的键来检索值。
- HashMap仅存储对象引用。 因此，必须将原语与其对应的包装器类一起使用。 如int将存储为Integer 。

主要方法

```
void clear() ：从HashMap中删除所有键-值对。
Object clone() ：返回指定HashMap的浅表副本。
boolean containsKey(Object key) ：根据是否在地图中找到指定的键，返回true或false 。
boolean containsValue(Object Value) ：类似于containsKey（）方法，它查找指定的值而不是键。
Object get(Object key) ：返回HashMap中指定键的值。
boolean isEmpty() ：检查地图是否为空。
Set keySet() ：返回存储在HashMap中的所有密钥的Set 。
Object put（Key k，Value v） ：将键值对插入HashMap中。
int size() ：返回地图的大小，该大小等于存储在HashMap中的键值对的数量。
Collection values() ：返回地图中所有值的集合。
Value remove(Object key) ：删除指定键的键值对。
void putAll(Map m) ：将地图的所有元素复制到另一个指定的地图。
```

合并两个hashmap

- 使用HashMap.putAll(HashMap)方法，即可将所有映射从第二张地图复制到第一张地图。hashmap不允许重复的键。 因此，当我们以这种方式合并map时，对于map1的重复键，其值会被map2相同键的值覆盖。

```
//map 1
HashMap<Integer, String> map1 = new HashMap<>();

map1.put(1, "A");
map1.put(2, "B");
map1.put(3, "C");
map1.put(5, "E");

//map 2
HashMap<Integer, String> map2 = new HashMap<>();

map2.put(1, "G");    //It will replace the value 'A'
map2.put(2, "B");
map2.put(3, "C");
map2.put(4, "D");    //A new pair to be added

//Merge maps
map1.putAll(map2);

System.out.println(map1);
```

- merge()函数如果我们要处理在地图中存在重复键的情况，并且我们不想丢失任何地图和任何键的数据。HashMap.merge()函数3个参数。 键，值，并使用用户提供的BiFunction合并重复键的值。跟put一样，实现了重复key的处理。

```
Merge HashMaps Example
//map 1
HashMap<Integer, String> map1 = new HashMap<>();

map1.put(1, "A");
map1.put(2, "B");
map1.put(3, "C");
map1.put(5, "E");

//map 2
HashMap<Integer, String> map2 = new HashMap<>();

```

```
map2.put(1, "G");    //It will replace the value 'A'
map2.put(2, "B");
map2.put(3, "C");
map2.put(4, "D");    //A new pair to be added

//Merge maps
map2.forEach(
     (key, value) -> map1.merge( key, value, (v1, v2) -> v1.equalsIgnoreCase(v2) ? v1 : v1 + "," + v2)
);

System.out.println(map1);
```

遍历方法

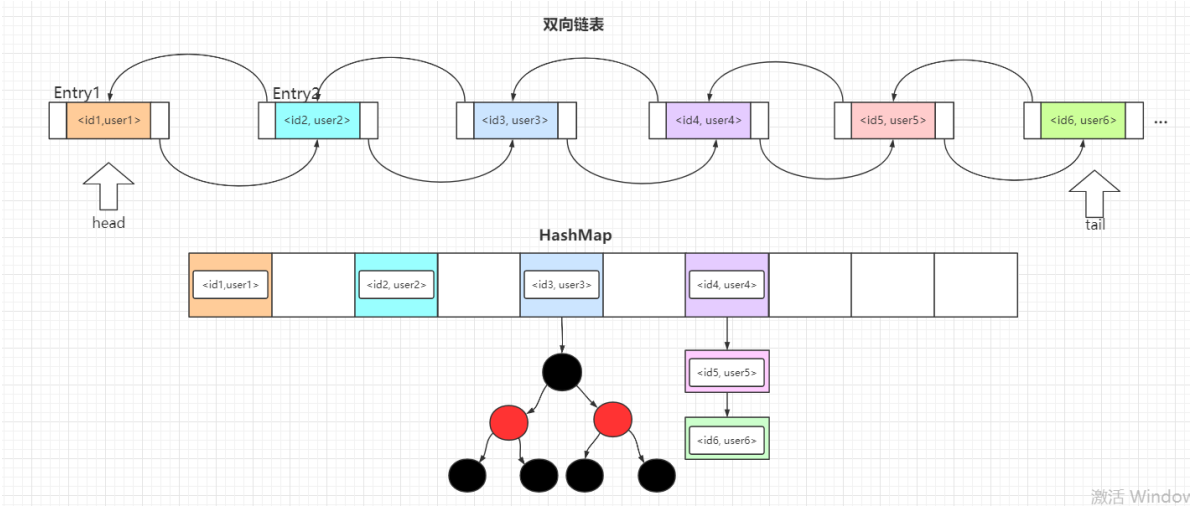
通过不同的set遍历呗。包括EntrySet遍历、keyset遍历

```
1)在每个循环中使用enrtySet（）
for (Map.Entry<String,Integer> entry : testMap.entrySet()) {
      entry.getKey();
      entry.getValue();
}
2）在每个循环中使用keySet（）
for (String key : testMap.keySet()) {
      testMap.get(key);
}
3）使用enrtySet（）和迭代器
Iterator<Map.Entry<String,Integer>> itr1 = testMap.entrySet().iterator();
while(itr1.hasNext())
{
      Map.Entry<String,Integer> entry = itr1.next();
      entry.getKey();
      entry.getValue();
}
4）使用keySet（）和迭代器
Iterator itr2 = testMap.keySet().iterator();
while(itr2.hasNext())
{
      String key = itr2.next();
      testMap.get(key);
}
```

LinkedHashMap

底层原理

LinkedHashMap 可以看作是 `HashMap` 和 `LinkedList` 的结合：它在 HashMap 的基础上添加了一条双向链表，`默认`存储各个元素的插入顺序，但由于这条双向链表，使得 LinkedHashMap 可以实现 `LRU` 缓存淘汰策略，因为我们可以设置这条双向链表按照`元素的访问次序`进行排序



LinkedHashMap 是 HashMap 的子类，所以它具备 HashMap 的所有特点，其次，它在 HashMap 的基础上维护了一条 双向链表，该链表存储了**所有元素**， 默认 元素的顺序与插入顺序**一致**。若 accessOrder 属性为 true，则遍历顺序按元素的访问次序进行排序。

```
// 头节点
transient LinkedHashMap.Entry<K, V> head;
// 尾结点
transient LinkedHashMap.Entry<K, V> tail;
```

利用 LinkedHashMap 可以实现 LRU 缓存淘汰策略，因为它提供了一个方法：

```
protected boolean removeEldestEntry(java.util.Map.Entry<K, V> eldest) {
      return false;
}
```

该方法可以移除 最靠近链表头部 的一个节点，而在 get() 方法中可以看到下面这段代码，其作用是挪动结点的位置：

```
if (this.accessOrder) {
      this.afterNodeAccess(e);
}
```

只要调用了 get() 且 accessOrder = true，则会将该节点更新到链表 尾部，具体的逻辑在 afterNodeAccess() 中，感兴趣的可翻看源码，篇幅原因这里不再展开。

现在如果要实现一个 LRU 缓存策略，则需要做两件事情：

- 指定 accessOrder = true 可以设定链表按照访问顺序排列，通过提供的构造器可以设定 accessOrder

```
public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder) {
      super(initialCapacity, loadFactor);
      this.accessOrder = accessOrder;
}
```

- 重写 removeEldestEntry() 方法，内部定义逻辑，通常是判断 容量 是否达到上限，若是则执行淘汰。

这里就要贴出一道大厂面试必考题目： **146. LRU缓存机制**，只要跟着我的步骤，就能顺利完成这道大厂题了。

关于 LinkedHashMap 主要介绍两点：

1. 它底层维护了一条 双向链表，因为继承了 HashMap，所以它也不是线程安全的

2. LinkedHashMap 可实现 `LRU` 缓存淘汰策略，其原理是通过设置 `accessOrder` 为 `true` 并重写 `removeEldestEntry` 方法定义淘汰元素时需满足的条件

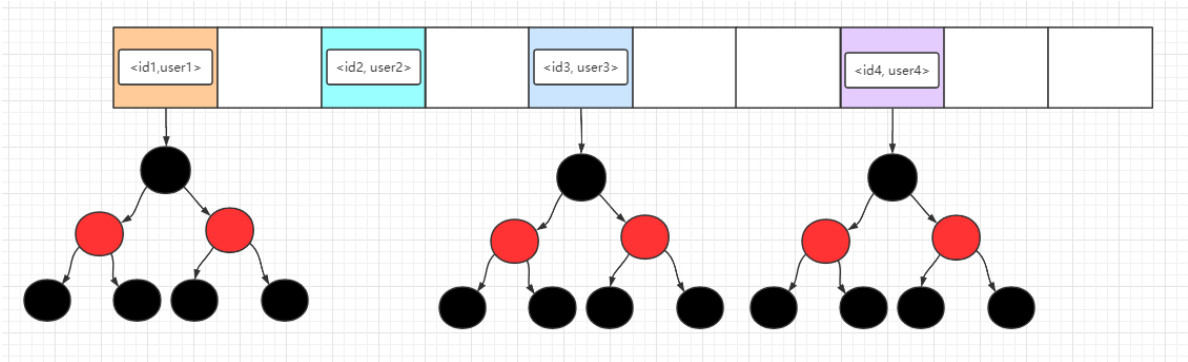
TreeMap

底层原理

TreeMap 是 `SortedMap` 的子类，所以它具有**排序**功能。它是基于 `红黑树` 数据结构实现的，每一个键值对 `<key, value>` 都是一个结点，默认情况下按照 `key` 自然排序，另一种是可以通过传入定制的 `Comparator` 进行自定义规则排序。

```
// 按照 key 自然排序，Integer 的自然排序是升序
TreeMap<Integer, Object> naturalSort = new TreeMap<>();
// 定制排序，按照 key 降序排序
TreeMap<Integer, Object> customSort = new TreeMap<>((o1, o2) -> Integer.compare(o2, o1));
```

TreeMap 底层使用了数组+红黑树实现，所以里面的存储结构可以理解成下面这幅图哦。



图中红黑树的每一个节点都是一个 `Entry`，在这里为了图片的简洁性，就不标明 key 和 value 了，注意这些元素都是已经按照 `key` 排好序了，整个数据结构都是保持着 `有序` 的状态！

关于 `自然` 排序与 `定制` 排序：

- 自然排序：要求 `key` 必须实现 `Comparable` 接口。

由于 `Integer` 类实现了 Comparable 接口，按照自然排序规则是按照 `key` 从小到大排序。

```
TreeMap<Integer, String> treeMap = new TreeMap<>();
treeMap.put(2, "TWO");
treeMap.put(1, "ONE");
System.out.print(treeMap);
// {1=ONE, 2=TWO}
```

- 定制排序：在初始化 TreeMap 时传入新的 `Comparator`，**不**要求 `key` 实现 Comparable 接口

```
TreeMap<Integer, String> treeMap = new TreeMap<>((o1, o2) -> Integer.compare(o2, o1));
treeMap.put(1, "ONE");
treeMap.put(2, "TWO");
treeMap.put(4, "FOUR");
treeMap.put(3, "THREE");
System.out.println(treeMap);
// {4=FOUR, 3=THREE, 2=TWO, 1=ONE}
```

通过传入新的 `Comparator` 比较器，可以覆盖默认的排序规则，上面的代码按照 `key` 降序排序，在实际应用中还可以按照其它规则自定义排序。

`compare()` 方法的返回值有三种，分别是：`0`，`-1`，`+1`

（1）如果返回 `0`，代表两个元素相等，不需要调换顺序

（2）如果返回 `+1`，代表前面的元素需要与后面的元素调换位置

（3）如果返回 `-1`，代表前面的元素不需要与后面的元素调换位置

而何时返回 `+1` 和 `-1`，则由我们自己去定义，JDK默认是按照**自然排序**，而我们可以根据 `key` 的不同去定义降序还是升序排序。

关于 TreeMap 主要介绍了两点：

1. 它底层是由 `红黑树` 这种数据结构实现的，所以操作的时间复杂度恒为 `O(logN)`
2. TreeMap 可以对 `key` 进行自然排序或者自定义排序，自定义排序时需要传入 `Comparator`，而自然排序要求 `key` 实现了 `Comparable` 接口
3. TreeMap 不是线程安全的。它不synchronized 。 使用Collections.synchronizedSortedMap(new TreeMap())在并发环境中工作。
4. 它不能具有null键，但可以具有多个null值。
5. 它以排序顺序（自然顺序）或地图创建时提供的Comparator来存储键。
6. 它为containsKey， get， put和remove操作提供了保证的log(n)时间成本。

主要方法

```
void clear():从地图中删除所有键/值对。
void size():返回此映射中存在的键值对的数量。
void isEmpty():如果此映射不包含键值映射，则返回true。
boolean containsKey(Object key):如果地图中存在指定的键，则返回'true' 。
boolean containsValue(Object key):如果将指定值映射到映射中的至少一个键，则返回'true' 。
Object get(Object key):检索由指定key映射的值；如果此映射不包含key映射关系，则返回null。
Object remove(Object key):如果存在，则从映射中删除指定键的键值对。
Comparator comparator():它返回用于对该映射中的键进行排序的比较器；如果此映射使用其键的自然排序，则返回null。
Object firstKey():返回树图中当前的第一个（最小）键。
Object lastKey():返回树图中当前的最后一个（最大）键。
Object ceilingKey(Object key):返回大于或等于给定键的最小键；如果没有这样的键，则返回null。
Object higherKey(Object key):返回严格大于指定键的最小键。
NavigableMap descendingMap():它返回此地图中包含的映射的reverse order view 。
```

WeakHashMap

WeakHashMap 日常开发中比较少见，它是基于普通的 `Map` 实现的，而里面 `Entry` 中的键在每一次的 `垃圾回收` 都会被清除掉，所以非常适合用于**短暂访问、仅访问一次**的元素，缓存在 `WeakHashMap` 中，并尽早地把它回收掉。

当 `Entry` 被 `GC` 时，WeakHashMap 是如何感知到某个元素被回收的呢?

在 WeakHashMap 内部维护了一个引用队列 `queue`

```
private final ReferenceQueue<Object> queue = new ReferenceQueue<>();
```

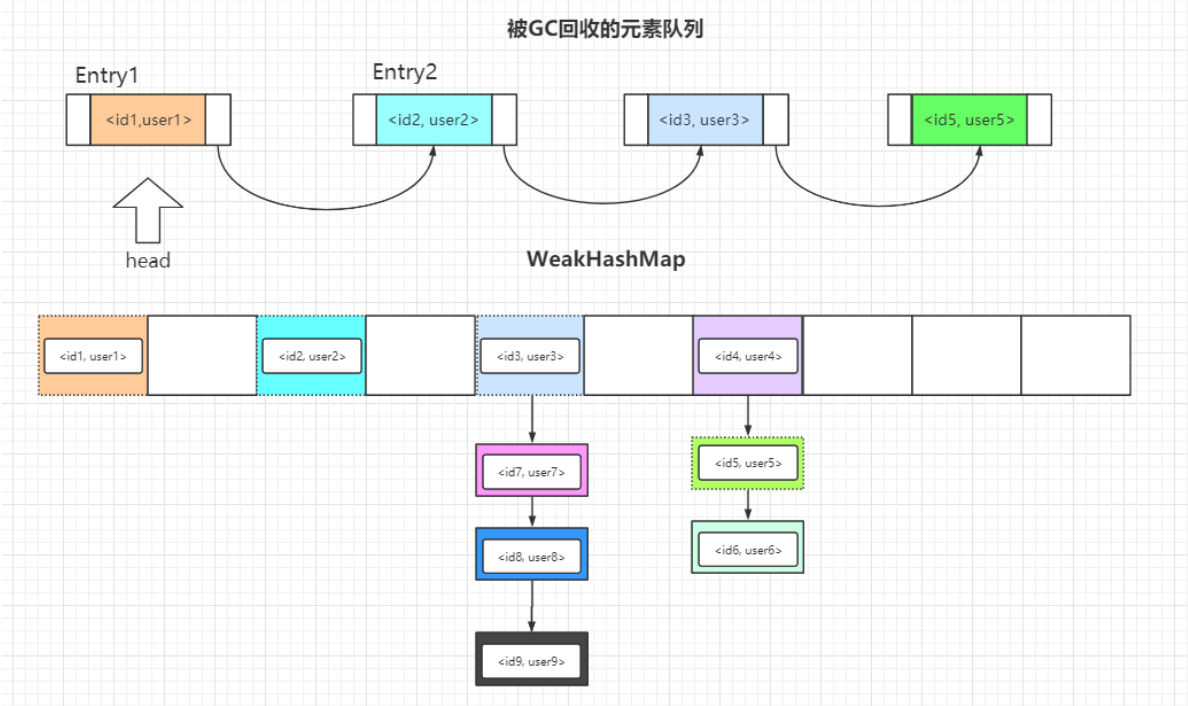
这个 queue 里包含了所有被 `GC` 掉的键，当JVM开启 `GC` 后，如果回收掉 WeakHashMap 中的 key，会将 key 放入queue 中，在 `expungeStaleEntries()` 中遍历 queue，把 queue 中的所有 `key` 拿出来，并在 WeakHashMap 中删除掉，以达到**同步**。

```
private void expungeStaleEntries() {
      for (Object x; (x = queue.poll()) != null; ) {
            synchronized (queue) {
                  // 去 WeakHashMap 中删除该键值对
            }
      }
}
```

再者，需要注意 WeakHashMap 底层存储的元素的数据结构是 `数组 + 链表`，**没有红黑树**哦，可以换一个角度想，如果还有红黑树，那干脆直接继承 HashMap ，然后再扩展就完事了嘛，然而它并没有这样做：

```
public class WeakHashMap<K, V> extends AbstractMap<K, V> implements Map<K, V> {

}
```

所以，WeakHashMap 的数据结构图我也为你准备好啦。



图中被虚线标识的元素将会在下一次访问 WeakHashMap 时被删除掉，WeakHashMap 内部会做好一系列的调整工作，所以记住队列的作用就是标志那些已经被 `GC` 回收掉的元素。

关于 WeakHashMap 需要注意两点：

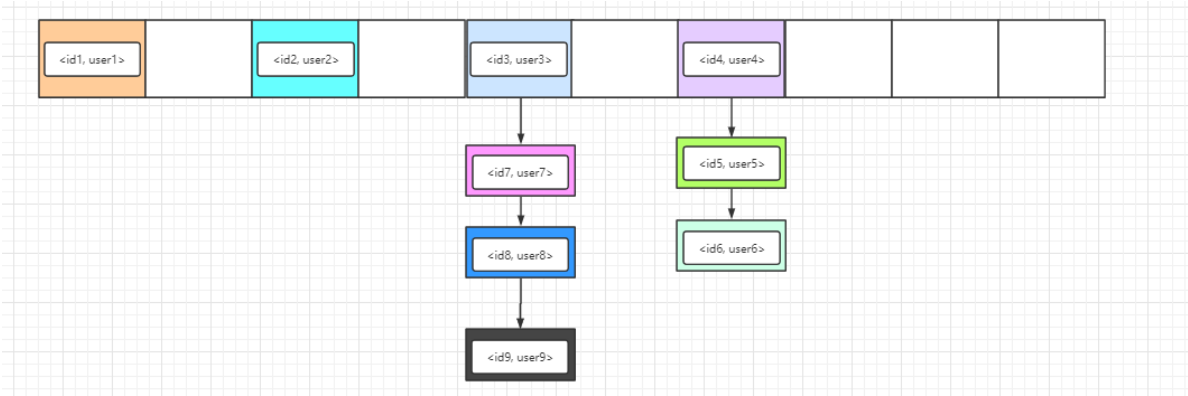
1. 它的键是一种**弱键**，放入 WeakHashMap 时，随时会被回收掉，所以不能确保某次访问元素一定存在
2. 它依赖普通的 `Map` 进行实现，是一个非线程安全的集合
3. WeakHashMap 通常作为**缓存**使用，适合存储那些**只需访问一次**、或**只需保存短暂时间**的键值对

Hashtable

底层原理

Hashtable 底层的存储结构是 `数组 + 链表`，而它是一个**线程安全**的集合，但是因为这个线程安全，它就被淘汰掉了。

下面是Hashtable存储元素时的数据结构图，它只会存在数组+链表，当链表过长时，查询的效率过低，而且会长时间**锁住** Hashtable。



> 本质上就是 WeakHashMap 的底层存储结构了。你千万别问为什么 WeakHashMap 不继承 Hashtable 哦，Hashtable 的 `性能` 在并发环境下非常差，在非并发环境下可以用 `HashMap` 更优。

HashTable 本质上是 HashMap 的前辈，它被淘汰的原因也主要因为两个字：**性能**

HashTable 是一个 **线程安全** 的 Map，它所有的方法都被加上了 **synchronized** 关键字，也是因为这个关键字，它注定成为了时代的弃儿。

HashTable 底层采用 **数组+链表** 存储键值对，由于被弃用，后人也没有对它进行任何改进

HashTable 默认长度为 `11`，负载因子为 `0.75F`，即元素个数达到数组长度的 75% 时，会进行一次扩容，每次扩容为原来数组长度的 `2` 倍

HashTable 所有的操作都是线程安全的。

方法

跟hashtable一样

ConcurrentHashMap

底层原理

oncurrentHashMap通过设计支持并发访问其键值对。

使用方法

创建和读写

```
import java.util.Iterator;
import java.util.concurrent.ConcurrentHashMap;

public class HashMapExample
{
      public static void main(String[] args) throws CloneNotSupportedException
      {
            ConcurrentHashMap<Integer, String> concurrHashMap = new ConcurrentHashMap<>();

            //Put require no synchronization
            concurrHashMap.put(1, "A");
            concurrHashMap.put(2, "B");

            //Get require no synchronization
            concurrHashMap.get(1);

            Iterator<Integer> itr = concurrHashMap.keySet().iterator();
```

```
18
19              //Using synchronized block is advisable
20              synchronized (concurrHashMap)
21             {
22                    while(itr.hasNext()) {
23                          System.out.println(concurrHashMap.get(itr.next()));
24                    }
25             }
26       }
27  }
```

使用Collection.synchronizedMap也有同样的方法

```
1   import java.util.Collections;
2   import java.util.HashMap;
3   import java.util.Iterator;
4   import java.util.Map;
5
6   public class HashMapExample
7   {
8         public static void main(String[] args) throws CloneNotSupportedException
9        {
10             Map<Integer, String> syncHashMap = Collections.synchronizedMap(new HashMap<>());
11
12             //Put require no synchronization
13             syncHashMap.put(1, "A");
14             syncHashMap.put(2, "B");
15
16             //Get require no synchronization
17             syncHashMap.get(1);
18
19             Iterator<Integer> itr = syncHashMap.keySet().iterator();
20
21             //Using synchronized block is advisable
22             synchronized (syncHashMap)
23            {
24                   while(itr.hasNext()) {
25                         System.out.println(syncHashMap.get(itr.next()));
26                   }
27            }
28       }
29  }
```

5. Iterator Iterable ListIterator

Iterator

所有Java集合类都提供iterator()方法，该方法返回Iterator的实例以遍历该集合中的元素。

```
1   public interface Iterator<E> {
2        boolean hasNext();
3        E next();
4        void remove();
5   }
```

提供的API接口含义如下：

- `hasNext()`：判断集合中是否存在下一个对象
- `next()`：返回集合中的下一个对象，并将访问指针移动一位
- `remove()`：删除集合中调用`next()`方法返回的对象.每次调用next（）只能调用一次此方法。

在早期，遍历集合的方式只有一种，通过`Iterator`迭代器操作

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);
Iterator iter = list.iterator();
while (iter.hasNext()) {
      Integer next = iter.next();
      System.out.println(next);
      if (next == 2) { iter.remove(); }
}
```

Iterable

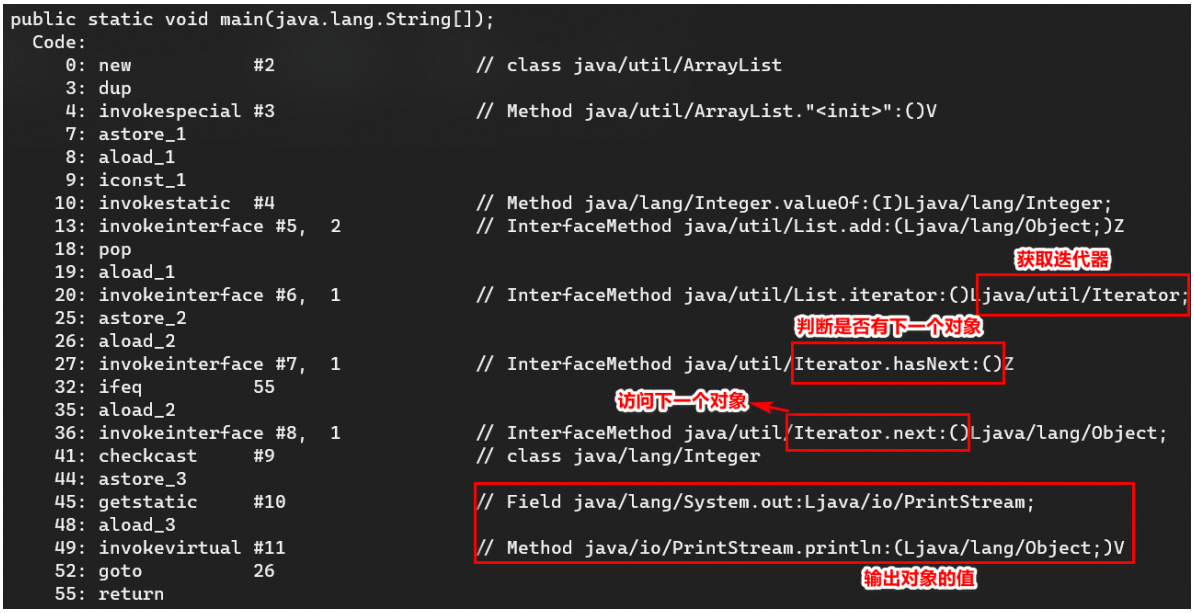
```
public interface Iterable<T> {
      Iterator<T> iterator();
      // JDK 1.8
      default void forEach(Consumer<? super T> action) {
            Objects.requireNonNull(action);
            for (T t : this) {
                  action.accept(t);
            }
      }
}
```

可以看到`Iterable`接口里面提供了`Iterator`接口，所以实现了`Iterable`接口的集合依旧可以使用`迭代器`遍历和操作集合中的对象；

而在`JDK 1.8`中，`Iterable`提供了一个新的方法`forEach()`，它允许使用增强 for 循环遍历对象。

```
List<Integer> list = new ArrayList<>();
for (Integer num : list) {
      System.out.println(num);
}
```

我们通过命令：`javap -c`反编译上面的这段代码后，发现它只是 Java 中的一个`语法糖`，本质上还是调用`Iterator`去遍历。



翻译成代码，就和一开始的 `Iterator` 迭代器遍历方式基本相同了。

```
Iterator iter = list.iterator();
while (iter.hasNext()) {
      Integer num = iter.next();
      System.out.println(num);
}
```

> 还有更深层次的探讨：为什么要设计两个接口 `Iterable` 和 `Iterator`，而不是保留其中一个就可以了。
>
> 简单讲解：`Iterator` 的保留可以让子类去**实现自己的迭代器**，而 `Iterable` 接口更加关注于 `for-each` 的增强语法。具体可参考：**Java中的Iterable与Iterator详解**

Iterator 和Iterable

关于 `Iterator` 和 `Iterable` 的讲解告一段落，下面来总结一下它们的重点：

1. `Iterator` 是提供集合操作内部对象的一个迭代器，它可以**遍历、移除**对象，且只能够**单向移动**
2. `Iterable` 是对 `Iterator` 的封装，在 `JDK 1.8` 时，实现了 `Iterable` 接口的集合可以使用**增强 for 循环**遍历集合对象，我们通过**反编译**后发现底层还是使用 `Iterator` 迭代器进行遍历

等等，这一章还没完，还有一个 `ListIterator`。它继承 Iterator 接口，在遍历 `List` 集合时可以从**任意索引下标**开始遍历，而且支持**双向遍历**。

ListIterator 存在于 List 集合之中，通过调用方法可以返回**起始下标**为 `index` 的迭代器

```
List<Integer> list = new ArrayList<>();
// 返回下标为0的迭代器
ListIterator<Integer> listIter1 = list.listIterator();
// 返回下标为5的迭代器
ListIterator<Integer> listIter2 = list.listIterator(5);
```

ListIterator 中有几个重要方法，大多数方法与 Iterator 中定义的含义相同，但是比 Iterator 强大的地方是可以在**任意一个下标位置**返回该迭代器，且可以实现**双向遍历**。

```java
public interface ListIterator<E> extends Iterator<E> {
	boolean hasNext();
	E next();
	boolean hasPrevious();
	E previous();
	int nextIndex();
	int previousIndex();
	void remove();
	// 替换当前下标的元素,即访问过的最后一个元素
	void set(E e);
	void add(E e);
}
```

ListIterator

简介

- ListIterator支持在元素列表上的所有CRUD操作（CREATE，READ，UPDATE和DELETE）。
- 与Iterator不同，ListIterator是bi-directional。 它支持正向和反向迭代。
- 它没有当前元素； 它的光标位置始终位于通过调用previous（）返回的元素和通过调用next（）返回的元素之间。

实例

```java
ArrayList<String> list = new ArrayList<>();

list.add("A");
list.add("B");
list.add("C");
list.add("D");
list.add("E");
list.add("F");

ListIterator<String> listIterator = list.listIterator();

System.out.println("Forward iteration");

//Forward iterator
while(listIterator.hasNext()) {
	System.out.print(listIterator.next() + ",");
}

System.out.println("Backward iteration");

//Backward iterator
while(listIterator.hasPrevious()) {
	System.out.print(listIterator.previous() + ",");
}

System.out.println("Iteration from specified position");

//Start iterating from index 2
listIterator = list.listIterator(2);

while(listIterator.hasNext()) {
	System.out.print(listIterator.next() + ",");
```

主要方法

```
void add(Object o) ：将指定的元素插入列表（可选操作）。
boolean hasNext() ：如果在向前遍历列表时此列表迭代器包含更多元素，则返回true 。
boolean hasPrevious() ：如果在反向遍历列表时此列表迭代器包含更多元素，则返回true 。
Object next() ：返回列表中的下一个元素并前进光标位置。
int nextIndex() ：返回元素的索引，该元素的索引将由对next（）的后续调用返回。
Object previous() ：返回列表中的上一个元素，并将光标位置向后移动。
int previousIndex() ：返回元素的索引，该元素的索引将由对next（）的后续调用返回。
void remove() ：从列表中移除next（）或previous（）返回的最后一个元素（可选操作）。
void set(Object o) ：将next（）或previous（）返回的最后一个元素替换为指定的元素（可选操作）。
```

6. Comparable

概述

Java Comparable接口，用于根据对象的natural order对array或对象list进行natural order 。 元素的自然排序是通过在对象中实现其compareTo()方法来实现的。

```
public interface Comparable<T>
{
      public int compareTo(T o);
}
```

使用

```
import java.time.LocalDate;

public class Employee implements Comparable<Employee> {

       private Long id;
       private String name;
       private LocalDate dob;

       @Override
       public int compareTo(Employee o)
       {
             return this.getId().compareTo( o.getId() );
       }
}

```

- 使用Collections.sort（）方法对对象list进行排序。
- 使用Arrays.sort（）方法对对象array进行排序。
- Collections.reverseOrder()

Comparator比较器

我们还是先研究这个方法

`public static <T> void sort(List<T> list)` :将集合中元素按照默认规则排序。

不过这次存储的是字符串类型。

```
public class CollectionsDemo2 {
	public static void main(String[] args) {
		ArrayList<String>   list = new ArrayList<String>();
		list.add("cba");
		list.add("aba");
		list.add("sba");
		list.add("nba");
		//排序方法
		Collections.sort(list);
		System.out.println(list);
	}
}
```

结果：

```
[aba, cba, nba, sba]
```

- 使用Collections.sort(list, Comparator)方法按提供的比较器实例施加的顺序对对象list进行排序。
- 使用Arrays.sort(array, Comparator)方法按提供的比较器实例施加的顺序对对象array进行排序。

Comparator.compare()

- `public int compare(String o1, String o2)`：比较其两个参数的顺序。

> 两个对象比较的结果有三种：大于，等于，小于。
>
> 如果要按照升序排序，
> 则o1 小于o2，返回（负数），相等返回0，01大于02返回（正数）
> 如果要按照降序排序
> 则o1 小于o2，返回（正数），相等返回0，01大于02返回（负数）

操作如下：

```
public class CollectionsDemo3 {
	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<String>();
		list.add("cba");
		list.add("aba");
		list.add("sba");
		list.add("nba");
		//排序方法  按照第一个单词的降序
		Collections.sort(list, new Comparator<String>() {
			@Override
			public int compare(String o1, String o2) {
				return o2.charAt(0) - o1.charAt(0);
			}
		});
		System.out.println(list);
	}
}
```

结果如下：

```
[sba, nba, cba, aba]
```

Collections.comparing()

该实用程序方法接受一个为类提取排序键的函数。 本质上，这是一个将对类对象进行排序的字段。

```
//Order by name
Comparator.comparing(Employee::getName);

//Order by name in reverse order
Comparator.comparing(Employee::getName).reversed();

//Order by id field
Comparator.comparing(Employee::getId);

//Order by employee age
Comparator.comparing(Employee::getDate);

ArrayList<Employee> list = new ArrayList<>();

list.add(new Employee(221, "Lokesh", LocalDate.now()));
list.add(new Employee(301, "Bob", LocalDate.now()));
list.add(new Employee(181, "Alex", LocalDate.now()));
list.add(new Employee(51, "David", LocalDate.now()));
list.add(new Employee(6001, "Charles", LocalDate.now()));

Collections.sort(list, Comparator.comparing( Employee::getName ).reversed());

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

Collections.thenComparing()

此实用程序方法用于group by sort 。

```
//Order by name and then by age
Comparator.comparing(Employee::getName)
                  .thenComparing(Employee::getDob);

//Order by name -> date of birth -> id
Comparator.comparing(Employee::getName)
                  .thenComparing(Employee::getDob)
                  .thenComparing(Employee::getId);

ArrayList<Employee> list = new ArrayList<>();

list.add(new Employee(221, "Lokesh", LocalDate.now()));
list.add(new Employee(301, "Lokesh", LocalDate.now()));
list.add(new Employee(181, "Alex", LocalDate.now()));
list.add(new Employee(51, "Lokesh", LocalDate.now()));
list.add(new Employee(6001, "Charles", LocalDate.now()));

Comparator<Employee> groupByComparator = Comparator.comparing(Employee::getName)
                                                   .thenComparing(Employee::getDob)
                                                   .thenComparing(Employee::getId);

Collections.sort(list, groupByComparator);

System.out.println(list);

```

Collections.reverseOrder()

此实用程序方法返回一个比较器，该比较器对实现Comparable接口的对象集合强加natural ordering或total ordering的逆序。

```
//Reverse of natural order as specified in
//Comparable interface's compareTo() method

Comparator.reversed();

//Reverse of order by name

Comparator.comparing(Employee::getName).reversed();

```

简述Comparable和Comparator两个接口的区别

Comparable：强行对实现它的每个类的对象进行整体排序。这种排序被称为类的自然排序，类的compareTo方法被称为它的自然比较方法。只能在类中实现compareTo()一次，不能经常修改类的代码实现自己想要的排序。实现此接口的对象列表（和数组）可以通过Collections.sort（和Arrays.sort）进行自动排序，对象可以用作有序映射中的键或有序集合中的元素，无需指定比较器。

Comparator：强行对某个对象进行整体排序。可以将Comparator 传递给sort方法（如Collections.sort或Arrays.sort），从而允许在排序顺序上实现精确控制。还可以使用Comparator来控制某些数据结构（如有序set或有序映射）的顺序，或者为那些没有自然顺序的对象collection提供排序。

7. Sort

1 数组排序Arrays.sort

Java程序使用Arrays.sort()方法升序排序

```
import java.util.Arrays;

public class JavaSortExample
{
      public static void main(String[] args)
      {
            //Unsorted array
            Integer[] numbers = new Integer[] { 15, 11, 9, 55, 47, 18, 520, 1123, 366, 420 };

            //Sort the array
            Arrays.sort(numbers);

            //Print array to confirm
            System.out.println(Arrays.toString(numbers));
      }
}
```

逆序

```
Integer[] numbers = new Integer[] { 15, 11, 9, 55, 47, 18, 520, 1123, 366, 420 };

//Sort the array in reverse order
Arrays.sort(numbers, Collections.reverseOrder());

//Print array to confirm
System.out.println(Arrays.toString(numbers));
```

部分排序

```
//Unsorted array
Integer[] numbers = new Integer[] { 15, 11, 9, 55, 47, 18, 1123, 520, 366, 420 };

//Sort the array
Arrays.sort(numbers, 2, 6);

//Print array to confirm
System.out.println(Arrays.toString(numbers));
```

并发排序

它将数组分解为不同的子数组，并且每个子数组在different threads使用Arrays.sort()进行排序。 最终，所有排序的子数组将合并为一个数组。

```
Arrays.parallelSort(numbers);

Arrays.parallelSort(numbers, 2, 6);

Arrays.parallelSort(numbers, Collections.reverseOrder());
```

> 不支持集合排序。转换为列表，然后排序，然后转换为集合。
> 不支持map排序。获取keyset，然后排序访问。
> 但是treeSet和TreeMap本身都是排序好的。

2 字符串排序方法

Stream API

使用Stream.sorted() API对字符串的字符进行排序的示例。

```
String randomString = "adcbgekhs";

String sortedChars = Stream.of( randomString.split("") )
                                    .sorted()
                                    .collect(Collectors.joining());

System.out.println(sortedChars);      // abcdeghks
```

Arrays.sort()

使用Arrays.sort()方法对字符串排序的示例。

```
String randomString = "adcbgekhs";

//Convert string to char array
char[] chars = randomString.toCharArray();

//Sort char array
Arrays.sort(chars);

//Convert char array to string
String sortedString = String.valueOf(chars);

System.out.println(sortedChars);      // abcdeghks
```

3 ArraysList排序

自带的sort方法

```
//Unsorted list
List<String> names = Arrays.asList("Alex", "Charles", "Brian", "David");

//1. Natural order
names.sort( Comparator.comparing( String::toString ) );

System.out.println(names);

//2. Reverse order
names.sort( Comparator.comparing( String::toString ).reversed() );

System.out.println(names);
```

Collections.sort

```
//Unsorted list
List<String> names = Arrays.asList("Alex", "Charles", "Brian", "David");

//1. Natural order
Collections.sort(names);

System.out.println(names);

//2. Reverse order
Collections.sort(names, Collections.reverseOrder());

System.out.println(names);
```

Stream

```
//Unsorted list
List<String> names = Arrays.asList("Alex", "Charles", "Brian", "David");

//1. Natural order
List<String> sortedNames = names
.stream()
.sorted(Comparator.comparing(String::toString))
.collect(Collectors.toList());

System.out.println(sortedNames);
```

```
11
12  //2. Reverse order
13  List<String> reverseSortedNames = names
14        .stream()
15        .sorted(Comparator.comparing(String::toString).reversed())
16        .collect(Collectors.toList());
17
18  System.out.println(reverseSortedNames);
```

8. Stream

1 数据流原理

基本原理

> 数据流和输入输出流不同
> 这个示例驱动的教程是Java8**数据流**（Stream）的深入总结。当我第一次看到 `Stream` API时，我非常疑惑，因为它听起来和Java IO的 `InputStream` 和 `OutputStream` 一样。但是Java8的数据流是完全不同的东西。

> 在函数式编程中，单体是一个结构，表示定义为步骤序列的计算。单体结构的类型定义了它对链式操作，或具有相同类型的嵌套函数的含义。

数据流的链式操作

数据流表示元素的序列，并支持不同种类的操作来执行元素上的计算：

```
1   List<String> myList =
2         Arrays.asList("a1", "a2", "b1", "c2", "c1");
3
4   myList
5        .stream()
6        .filter(s -> s.startsWith("c"))
7        .map(String::toUpperCase)
8        .sorted()
9        .forEach(System.out::println);
```

- 这种数据流的链式操作也叫作操作流水线。
- 多数数据流操作都接受一些lambda表达式参数，函数式接口用来指定操作的具体行为。这些操作的大多数必须是无干扰而且是无状态的。
    - 当一个函数不修改数据流的底层数据源，它就是**无干扰的**。例如，在上面的例子中，没有任何lambda表达式通过添加或删除集合元素修改 `myList` 。
    - 当一个函数的操作的执行是确定性的，它就是**无状态的**。例如，在上面的例子中，没有任何lambda表达式依赖于外部作用域中任何在操作过程中可变的变量或状态。

2 数据流的不同类型

数据流可以从多种数据源创建，

从集合创建

`List` 和 `Set` 支持新方法 `stream()` 和 `parallelStream()`，来创建串行流或并行流。并行流能够在多个线程上执行操作。我们现在来看看串行流：

```
Arrays.asList("a1", "a2", "a3")
     .stream()
     .findFirst()
     .ifPresent(System.out::println);   // a1
```

Stream类创建

在对象列表上调用 `stream()` 方法会返回一个通常的对象流。但是我们不需要创建一个集合来创建数据流，就像下面那样：

```
Stream.of("a1", "a2", "a3")
     .findFirst()
     .ifPresent(System.out::println);   // a1
```

只要使用 `Stream.of()`，就可以从一系列对象引用中创建数据流。

基本数据流

Java8还自带了特殊种类的流，用于处理基本数据类型 `int`、`long` 和 `double`。`IntStream`、`LongStream` 和 `DoubleStream`。

`IntStream` 可以使用 `IntStream.range()` 替换通常的 `for` 循环：

```
IntStream.range(1, 4)
     .forEach(System.out::println);
// 1
// 2
// 3
```

所有这些基本数据流都像通常的对象数据流一样，但有一些不同。基本的数据流使用特殊的lambda表达式，例如，`IntFunction` 而不是 `Function`，`IntPredicate` 而不是 `Predicate`。而且基本数据流支持额外的聚合终止操作 `sum()` 和 `average()`：

```
Arrays.stream(new int[] {1, 2, 3})
     .map(n -> 2 * n + 1)
     .average()
     .ifPresent(System.out::println);   // 5.0
```

有时需要将通常的对象数据流转换为基本数据流，或者相反。出于这种目的，对象数据流支持特殊的映射操作 `mapToInt()`、`mapToLong()` 和 `mapToDouble()`：

```
Stream.of("a1", "a2", "a3")
     .map(s -> s.substring(1))
     .mapToInt(Integer::parseInt)
     .max()
     .ifPresent(System.out::println);   // 3
```

基本数据流可以通过 `mapToObj()` 转换为对象数据流：

```
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

下面是组合示例：浮点数据流首先映射为整数数据流，之后映射为字符串的对象数据流：

```
Stream.of(1.0, 2.0, 3.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

3 处理顺序

衔接操作和终止操作

衔接操作的一个重要特性就是延迟性。观察下面没有终止操作的例子：

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    });
```

执行这段代码时，不向控制台打印任何东西。这是因为衔接操作只在终止操作调用时被执行。

垂直执行

让我们通过添加终止操作 `forEach` 来扩展这个例子：

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

执行这段代码会得到如下输出：

```
filter:  d2
forEach: d2
filter:  a2
forEach: a2
filter:  b1
forEach: b1
filter:  b3
forEach: b3
filter:  c
forEach: c
```

结果的顺序可能出人意料。原始的方法会在数据流的所有元素上，一个接一个地水平执行所有操作。但是每个元素在调用链上垂直移动。第一个字符串 `"d2"` 首先经过 `filter` 然后是 `forEach`，执行完后才开始处理第二个字符串 `"a2"`。

这种行为可以减少每个元素上所执行的实际操作数量，就像我们在下个例子中看到的那样：

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
        return s.startsWith("A");
    });

// map:        d2
// anyMatch: D2
// map:        a2
// anyMatch: A2
```

只要提供的数据元素满足了谓词，`anyMatch` 操作就会返回 `true`。对于第二个传递 `"A2"` 的元素，它的结果为真。由于数据流的链式调用是垂直执行的，`map` 这里只需要执行两次。所以 `map` 会执行尽可能少的次数，而不是把所有元素都映射一遍。

先过滤排除

下面的例子由两个衔接操作 `map` 和 `filter`，以及一个终止操作 `forEach` 组成。让我们再来看看这些操作如何执行：

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
    .forEach(s -> System.out.println("forEach: " + s));

// map:       d2
// filter:  D2
// map:       a2
// filter:  A2
// forEach: A2
// map:       b1
// filter:  B1
// map:       b3
// filter:  B3
// map:       c
// filter:  C
```

就像你可能猜到的那样，`map` 和 `filter` 会对底层集合的每个字符串调用五次，而 `forEach` 只会调用一次。

如果我们调整操作顺序，将 `filter` 移动到调用链的顶端，就可以极大减少操作的执行次数：

```
Stream.of("d2", "a2", "b1", "b3", "c")
        .filter(s -> {
                System.out.println("filter: " + s);
                return s.startsWith("a");
        })
        .map(s -> {
                System.out.println("map: " + s);
                return s.toUpperCase();
        })
        .forEach(s -> System.out.println("forEach: " + s));

// filter:  d2
// filter:  a2
// map:       a2
// forEach: A2
// filter:  b1
// filter:  b3
// filter:  c
```

4 复用数据流

Java8的数据流不能被复用。一旦你调用了任何终止操作，数据流就关闭了：

```
Stream<String> stream =
        Stream.of("d2", "a2", "b1", "b3", "c")
             .filter(s -> s.startsWith("a"));

stream.anyMatch(s -> true);      // ok
stream.noneMatch(s -> true);    // exception
```

在相同数据流上，在 `anyMatch` 之后调用 `noneMatch` 会产生下面的异常：

```
java.lang.IllegalStateException: stream has already been operated upon or closed
        at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
        at java.util.stream.ReferencePipeline.noneMatch(ReferencePipeline.java:459)
        at com.winterbe.java8.Streams5.test7(Streams5.java:38)
        at com.winterbe.java8.Streams5.main(Streams5.java:28)
```

要克服这个限制，我们需要为每个我们想要执行的终止操作创建新的数据流调用链。例如，我们创建一个数据流供应器，来构建新的数据流，并且设置好所有衔接操作：

```
Supplier<Stream<String>> streamSupplier =
        () -> Stream.of("d2", "a2", "b1", "b3", "c")
                    .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true);    // ok
streamSupplier.get().noneMatch(s -> true);   // ok
```

每次对 `get()` 的调用都构造了一个新的数据流，我们将其保存来调用终止操作。

5 高级操作

数据流执行大量的不同操作。我们已经了解了一些最重要的操作，例如 `filter` 和 `map` 。我将它们留给你来探索所有其他的可用操作。下面让我们深入了解一些更复杂的操作： `collect` 、 `flatMap` 和 `reduce` 。

这一节的大部分代码示例使用下面的 `Person` 列表来演示：

```
class Person {
        String name;
```

```
        int age;

        Person(String name, int age) {
            this.name = name;
            this.age = age;
        }

        @Override
        public String toString() {
            return name;
        }
    }

    List<Person> persons =
        Arrays.asList(
            new Person("Max", 18),
            new Person("Peter", 23),
            new Person("Pamela", 23),
            new Person("David", 12));
```

collect

collect是非常有用的终止操作，将流中的元素存放在不同类型的结果中，例如List、Set或者Map。collect接受收集器（Collector），它由四个不同的操作组成：供应器（supplier）、累加器（accumulator）、组合器（combiner）和终止器（finisher）。这在开始听起来十分复杂，但是Java8通过内置的Collectors类支持多种内置的收集器。所以对于大部分常见操作，你并不需要自己实现收集器。

让我们以一个非常常见的用例来开始：

```
List<Person> filtered =
    persons
        .stream()
        .filter(p -> p.name.startsWith("P"))
        .collect(Collectors.toList());

System.out.println(filtered);    // [Peter, Pamela]
```

就像你看到的那样，它非常简单，只是从流的元素中构造了一个列表。如果需要以Set来替代List，只需要使用Collectors.toSet()就好了。

下面的例子按照年龄对所有人进行分组：

```
Map<Integer, List<Person>> personsByAge = persons
    .stream()
    .collect(Collectors.groupingBy(p -> p.age));

personsByAge
    .forEach((age, p) -> System.out.format("age %s: %s\n", age, p));

// age 18: [Max]
// age 23: [Peter, Pamela]
// age 12: [David]
```

收集器十分灵活。你也可以在流的元素上执行聚合，例如，计算所有人的平均年龄：

```
Double averageAge = persons
      .stream()
      .collect(Collectors.averagingInt(p -> p.age));

System.out.println(averageAge);        // 19.0
```

如果你对更多统计学方法感兴趣，概要收集器返回一个特殊的内置概要统计对象，所以我们可以简单计算最小年龄、最大年龄、算术平均年龄、总和和数量。

```
IntSummaryStatistics ageSummary =
       persons
            .stream()
            .collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76, min=12, average=19.000000, max=23}
```

下面的例子将所有人连接为一个字符串：

```
String phrase = persons
      .stream()
      .filter(p -> p.age >= 18)
      .map(p -> p.name)
      .collect(Collectors.joining(" and ", "In Germany ", " are of legal age."));

System.out.println(phrase);
// In Germany Max and Peter and Pamela are of legal age.
```

连接收集器接受分隔符，以及可选的前缀和后缀。

既然我们知道了一些最强大的内置收集器，让我们来尝试构建自己的特殊收集器吧。我们希望将流中的所有人转换为一个字符串，包含所有大写的名称，并以 | 分割。为了完成它，我们通过 `Collector.of()` 创建了一个新的收集器。我们需要传递一个收集器的四个组成部分：供应器、累加器、组合器和终止器。

```
 Collector<Person, StringJoiner, String> personNameCollector =
       Collector.of(
            () -> new StringJoiner(" | "),              // supplier
            (j, p) -> j.add(p.name.toUpperCase()),   // accumulator
            (j1, j2) -> j1.merge(j2),                      // combiner
             StringJoiner::toString);                        // finisher

 String names = persons
      .stream()
      .collect(personNameCollector);

 System.out.println(names);   // MAX | PETER | PAMELA | DAVID
```

由于Java中的字符串是不可变的，我们需要一个助手类 `StringJointer` 。让收集器构造我们的字符串。供应器最开始使用相应的分隔符构造了这样一个 `StringJointer` 。累加器用于将每个人的大写名称加到 `StringJointer` 中。组合器知道如何把两个 `StringJointer` 合并为一个。最后一步，终结器从 `StringJointer` 构造出预期的字符串。

`flatMap`

我们已经了解了如何通过使用`map`操作，将流中的对象转换为另一种类型。`map`有时十分受限，因为每个对象只能映射为一个其它对象。但如何我希望将一个对象转换为多个或零个其他对象呢？`flatMap`这时就会派上用场。

`flatMap`将流中的每个元素，转换为其它对象的流。所以每个对象会被转换为零个、一个或多个其它对象，以流的形式返回。这些流的内容之后会放进`flatMap`所返回的流中。

在我们了解`flatMap`如何使用之前，我们需要相应的类型体系：

```
class Foo {
      String name;
      List<Bar> bars = new ArrayList<>();

      Foo(String name) {
            this.name = name;
      }
}

class Bar {
      String name;

      Bar(String name) {
            this.name = name;
      }
}
```

下面，我们使用我们自己的关于流的知识来实例化一些对象：

```
List<Foo> foos = new ArrayList<>();

// create foos
IntStream
     .range(1, 4)
     .forEach(i -> foos.add(new Foo("Foo" + i)));

// create bars
foos.forEach(f ->
      IntStream
           .range(1, 4)
           .forEach(i -> f.bars.add(new Bar("Bar" + i + " <- " + f.name))));
```

现在我们拥有了含有三个`foo`的列表，每个都含有三个`bar`。

`flatMap`接受返回对象流的函数。所以为了处理每个`foo`上的`bar`对象，我们需要传递相应的函数：

```
foos.stream()
      .flatMap(f -> f.bars.stream())
      .forEach(b -> System.out.println(b.name));

// Bar1 <- Foo1
// Bar2 <- Foo1
// Bar3 <- Foo1
// Bar1 <- Foo2
// Bar2 <- Foo2
// Bar3 <- Foo2
// Bar1 <- Foo3
// Bar2 <- Foo3
// Bar3 <- Foo3
```

像你看到的那样，我们成功地将含有三个 `foo` 对象中的流转换为含有九个 `bar` 对象的流。

最后，上面的代码示例可以简化为流式操作的单一流水线：

```
IntStream.range(1, 4)
      .mapToObj(i -> new Foo("Foo" + i))
      .peek(f -> IntStream.range(1, 4)
            .mapToObj(i -> new Bar("Bar" + i + " <- " + f.name))
            .forEach(f.bars::add))
      .flatMap(f -> f.bars.stream())
      .forEach(b -> System.out.println(b.name));
```

`flatMap` 也可用于Java8引入的 `Optional` 类。 `Optional` 的 `flatMap` 操作返回一个 `Optional` 或其他类型的对象。所以它可以用于避免烦人的 `null` 检查。

考虑像这样更复杂的层次结构：

```
class Outer {
      Nested nested;
}

class Nested {
      Inner inner;
}

class Inner {
      String foo;
}
```

为了处理外层示例上的内层字符串 `foo` ，你需要添加多个 `null` 检查来避免潜在的 `NullPointerException` ：

```
Outer outer = new Outer();
if (outer != null && outer.nested != null && outer.nested.inner != null) {
      System.out.println(outer.nested.inner.foo);
}
```

可以使用 `Optional` 的 `flatMap` 操作来完成相同的行为：

```
Optional.of(new Outer())
      .flatMap(o -> Optional.ofNullable(o.nested))
      .flatMap(n -> Optional.ofNullable(n.inner))
      .flatMap(i -> Optional.ofNullable(i.foo))
      .ifPresent(System.out::println);
```

如果存在的话，每个 `flatMap` 的调用都会返回预期对象的 `Optional` 包装，否则为 `null` 的 `Optional` 包装。

`reduce`

归约操作将所有流中的元素组合为单一结果。Java8支持三种不同类型的 `reduce` 方法。第一种将流中的元素归约为流中的一个元素。让我们看看我们如何使用这个方法来计算出最老的人：

```
persons
    .stream()
    .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
    .ifPresent(System.out::println);      // Pamela
```

`reduce` 方法接受 `BinaryOperator` 积累函数。它实际上是两个操作数类型相同的 `BiFunction`。`BiFunction` 就像是 `Function`，但是接受两个参数。示例中的函数比较两个人的年龄，来返回年龄较大的人。

第二个 `reduce` 方法接受一个初始值，和一个 `BinaryOperator` 累加器。这个方法可以用于从流中的其它 `Person` 对象中构造带有聚合后名称和年龄的新 `Person` 对象。

```
Person result =
      persons
           .stream()
           .reduce(new Person("", 0), (p1, p2) -> {
                 p1.age += p2.age;
                 p1.name += p2.name;
                 return p1;
           });

System.out.format("name=%s; age=%s", result.name, result.age);
// name=MaxPeterPamelaDavid; age=76
```

第三个 `reduce` 对象接受三个参数：初始值，`BiFunction` 累加器和 `BinaryOperator` 类型的组合器函数。由于初始值的类型不一定为 `Person`，我们可以使用这个归约函数来计算所有人的年龄总和。：

```
Integer ageSum = persons
    .stream()
    .reduce(0, (sum, p) -> sum += p.age, (sum1, sum2) -> sum1 + sum2);

System.out.println(ageSum);   // 76
```

你可以看到结果是76。但是背后发生了什么？让我们通过添加一些调试输出来扩展上面的代码：

```
Integer ageSum = persons
     .stream()
     .reduce(0,
           (sum, p) -> {
                 System.out.format("accumulator: sum=%s; person=%s\n", sum, p);
                 return sum += p.age;
           },
           (sum1, sum2) -> {
                 System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2);
                 return sum1 + sum2;
           });

// accumulator: sum=0; person=Max
// accumulator: sum=18; person=Peter
// accumulator: sum=41; person=Pamela
```

```
// accumulator: sum=64; person=David
```

你可以看到，累加器函数做了所有工作。它首先使用初始值`0`和第一个人Max来调用累加器。接下来的三步中`sum`会持续增加，直到76。

等一下。好像组合器从来没有调用过？以并行方式执行相同的流会揭开这个秘密：

```
Integer ageSum = persons
      .parallelStream()
      .reduce(0,
            (sum, p) -> {
                  System.out.format("accumulator: sum=%s; person=%s\n", sum, p);
                  return sum += p.age;
            },
            (sum1, sum2) -> {
                  System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2);
                  return sum1 + sum2;
            });

// accumulator: sum=0; person=Pamela
// accumulator: sum=0; person=David
// accumulator: sum=0; person=Max
// accumulator: sum=0; person=Peter
// combiner: sum1=18; sum2=23
// combiner: sum1=23; sum2=12
// combiner: sum1=41; sum2=35
```

这个流的并行执行行为会完全不同。现在实际上调用了组合器。由于累加器被并行调用，组合器需要用于计算部分累加值的总和。

下一节我们会深入了解并行流。

6 并行流

流可以并行执行，在大量输入元素上可以提升运行时的性能。并行流使用公共的`ForkJoinPool`，由`ForkJoinPool.commonPool()`方法提供。底层线程池的大小最大为五个线程 -- 取决于CPU的物理核数。

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
System.out.println(commonPool.getParallelism());      // 3
```

在我的机器上，公共池默认初始化为3。这个值可以通过设置下列JVM参数来增减：

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

集合支持`parallelStream()`方法来创建元素的并行流。或者你可以在已存在的数据流上调用衔接方法`parallel()`，将串行流转换为并行流。

为了描述并行流的执行行为，下面的例子向`sout`打印了当前线程的信息。

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
      .parallelStream()
      .filter(s -> {
            System.out.format("filter: %s [%s]\n",
                  s, Thread.currentThread().getName());
            return true;
      })
      .map(s -> {
            System.out.format("map: %s [%s]\n",
                  s, Thread.currentThread().getName());
```

```
            return s.toUpperCase();
        })
        .forEach(s -> System.out.format("forEach: %s [%s]\n",
                s, Thread.currentThread().getName()));
```

通过分析调试输出，我们可以对哪个线程用于执行流式操作拥有更深入的理解：

```
filter:  b1 [main]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:       a2 [ForkJoinPool.commonPool-worker-1]
filter:  c2 [ForkJoinPool.commonPool-worker-3]
map:       c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:       c1 [ForkJoinPool.commonPool-worker-2]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: A2 [ForkJoinPool.commonPool-worker-1]
map:       b1 [main]
forEach: B1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-3]
map:       a1 [ForkJoinPool.commonPool-worker-3]
forEach: A1 [ForkJoinPool.commonPool-worker-3]
forEach: C1 [ForkJoinPool.commonPool-worker-2]
```

就像你看到的那样，并行流使用了所有公共的 `ForkJoinPool` 中的可用线程来执行流式操作。在连续的运行中输出可能有所不同，因为所使用的特定线程是非特定的。

让我们通过添加额外的流式操作 `sort` 来扩展这个示例：

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
            System.out.format("filter: %s [%s]\n",
                    s, Thread.currentThread().getName());
            return true;
    })
    .map(s -> {
            System.out.format("map: %s [%s]\n",
                    s, Thread.currentThread().getName());
            return s.toUpperCase();
    })
    .sorted((s1, s2) -> {
            System.out.format("sort: %s <> %s [%s]\n",
                    s1, s2, Thread.currentThread().getName());
            return s1.compareTo(s2);
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
            s, Thread.currentThread().getName()));
```

结果起初可能比较奇怪：

```
filter:  c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:       c1 [ForkJoinPool.commonPool-worker-2]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:       a2 [ForkJoinPool.commonPool-worker-1]
filter:  b1 [main]
map:       b1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-2]
```

```
map:       a1 [ForkJoinPool.commonPool-worker-2]
map:       c2 [ForkJoinPool.commonPool-worker-3]
sort:     A2 <> A1 [main]
sort:     B1 <> A2 [main]
sort:     C2 <> B1 [main]
sort:     C1 <> C2 [main]
sort:     C1 <> B1 [main]
sort:     C1 <> C2 [main]
forEach: A1 [ForkJoinPool.commonPool-worker-1]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: B1 [main]
forEach: A2 [ForkJoinPool.commonPool-worker-2]
forEach: C1 [ForkJoinPool.commonPool-worker-1]
```

`sort` 看起来只在主线程上串行执行。实际上，并行流上的 `sort` 在背后使用了Java8中新的方法 `Arrays.parallelSort()` 。如javadoc所说，这个方法会参照数据长度来决定以串行或并行来执行。

> 如果指定数据的长度小于最小粒度，它使用相应的 `Arrays.sort` 方法来排序。

返回上一节中 `reduce` 的例子。我们已经发现了组合器函数只在并行流中调用，而不在串行流中调用。让我们来观察实际上涉及到哪个线程：

```
List<Person> persons = Arrays.asList(
      new Person("Max", 18),
      new Person("Peter", 23),
      new Person("Pamela", 23),
      new Person("David", 12));

persons
     .parallelStream()
     .reduce(0,
           (sum, p) -> {
                 System.out.format("accumulator: sum=%s; person=%s [%s]\n",
                       sum, p, Thread.currentThread().getName());
                 return sum += p.age;
           },
           (sum1, sum2) -> {
                 System.out.format("combiner: sum1=%s; sum2=%s [%s]\n",
                       sum1, sum2, Thread.currentThread().getName());
                 return sum1 + sum2;
           });
```

控制台的输出表明，累加器和组合器都在所有可用的线程上并行执行：

```
accumulator: sum=0; person=Pamela; [main]
accumulator: sum=0; person=Max;    [ForkJoinPool.commonPool-worker-3]
accumulator: sum=0; person=David;  [ForkJoinPool.commonPool-worker-2]
accumulator: sum=0; person=Peter;  [ForkJoinPool.commonPool-worker-1]
combiner:     sum1=18; sum2=23;       [ForkJoinPool.commonPool-worker-1]
combiner:     sum1=23; sum2=12;       [ForkJoinPool.commonPool-worker-2]
combiner:     sum1=41; sum2=35;       [ForkJoinPool.commonPool-worker-2]
```

总之，并行流对拥有大量输入元素的数据流具有极大的性能提升。但是要记住一些并行流的操作，例如 `reduce` 和 `collect` 需要额外的计算（组合操作），这在串行执行时并不需要。

此外我们已经了解，所有并行流操作都共享相同的JVM相关的公共 `ForkJoinPool` 。所以你可能需要避免实现又慢又卡的流式操作，因为它可能会拖慢你应用中严重依赖并行流的其它部分。

7 功能列表

中间管道

API	功能说明
filter()	按照条件过滤符合要求的元素， 返回新的stream流
map()	将已有元素转换为另一个对象类型，一对一逻辑，返回新的stream流
flatMap()	将已有元素转换为另一个对象类型，一对多逻辑，即原来一个元素对象可能会转换为1个或者多个新类型的元素，返回新的stream流
limit()	仅保留集合前面指定个数的元素，返回新的stream流
skip()	跳过集合前面指定个数的元素，返回新的stream流
concat()	将两个流的数据合并起来为1个新的流，返回新的stream流
distinct()	对Stream中所有元素进行去重，返回新的stream流
sorted()	对stream中所有的元素按照指定规则进行排序，返回新的stream流
peek()	对stream流中的每个元素进行逐个遍历处理，返回处理后的stream流

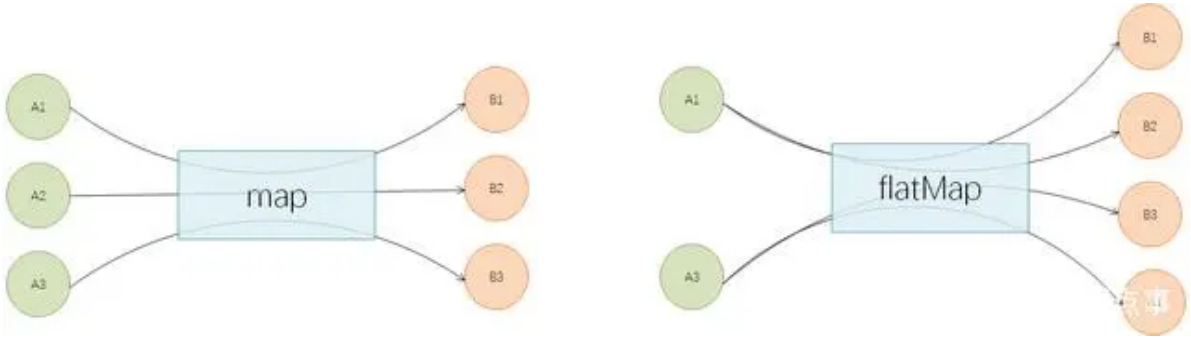
终止管道

API	功能说明
count()	返回stream处理后最终的元素个数
max()	返回stream处理后的元素最大值
min()	返回stream处理后的元素最小值
findFirst()	找到第一个符合条件的元素时则终止流处理
findAny()	找到任何一个符合条件的元素时则退出流处理，这个**对于串行流时与findFirst相同，对于并行流时比较高效**，任何分片中找到都会终止后续计算逻辑
anyMatch()	返回一个boolean值，类似于isContains(),用于判断是否有符合条件的元素
allMatch()	返回一个boolean值，用于判断是否所有元素都符合条件
noneMatch()	返回一个boolean值， 用于判断是否所有元素都不符合条件
collect()	将流转换为指定的类型，通过Collectors进行指定
toArray()	将流转换为数组
iterator()	将流转换为Iterator对象
foreach()	无返回值，对元素进行逐个遍历，然后执行给定的处理逻辑

map&flatmap

map与flatMap都是用于转换已有的元素为其它元素，区别点在于：

- map 必须是一对一的，即每个元素都只能转换为1个新的元素
- flatMap 可以是一对多的，即每个元素都可以转换为1个或者多个新的元素



peek和foreach方法

peek和foreach，都可以用于对元素进行遍历然后逐个的进行处理。

但根据前面的介绍，peek属于中间方法，而foreach属于终止方法。这也就意味着peek只能作为管道中途的一个处理步骤，而没法直接执行得到结果，其后面必须还要有其它终止操作的时候才会被执行；而foreach作为无返回值的终止方法，则可以直接执行相关操作。

第十五章 Java 泛型

Java 泛型（generics）是 JDK 5 中引入的一个新特性, 泛型提供了`编译时类型安全检测机制`，该机制允许程序员在编译时`检测到非法的类型`。

泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

> 假定我们有这样一个需求：写一个排序方法，能够对整型数组、字符串数组甚至其他任何类型的数组进行排序，该如何实现?
>
> 答案是可以使用 **Java 泛型**。
>
> 使用 Java 泛型的概念，我们可以写一个泛型方法来对一个对象数组排序。然后，调用该泛型方法来对整型数组、浮点数数组、字符串数组等进行排序。

泛型方法

你可以写一个泛型方法，该方法在调用时可以接收不同类型的参数。根据传递给泛型方法的参数类型，编译器适当地处理每一个方法调用。

下面是定义泛型方法的规则：

- 所有泛型方法声明都有一个类型参数声明部分（由尖括号分隔），该类型参数声明部分在方法返回类型之前（在下面例子中的 ）。
- 每一个类型参数声明部分包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。
- 类型参数能被用来声明返回值类型，并且能作为泛型方法得到的实际参数类型的占位符。
- 泛型方法体的声明和其他方法一样。注意类型参数只能代表引用型类型，不能是原始类型（像 **int、double、char** 等）。

java 中泛型标记符：

- **E** - Element (在集合中使用，因为集合中存放的是元素)
- **T** - Type（Java 类）
- **K** - Key（键）
- **V** - Value（值）
- **N** - Number（数值类型）
- **?** - 表示不确定的 java 类型

下面的例子演示了如何使用泛型方法打印不同类型的数组元素：

```
1   public class GenericMethodTest
2   {
3       // 泛型方法 printArray
4       public static < E > void printArray( E[] inputArray )
5       {
6           // 输出数组元素
7               for ( E element : inputArray ){
8                   System.out.printf( "%s ", element );
```

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

        public static void main( String args[] )
        {
            // 创建不同类型数组： Integer, Double 和 Character
            Integer[] intArray = { 1, 2, 3, 4, 5 };
            Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
            Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

            System.out.println( "整型数组元素为:" );
            printArray( intArray  ); // 传递一个整型数组

            System.out.println( "\n双精度型数组元素为:" );
            printArray( doubleArray ); // 传递一个双精度型数组

            System.out.println( "\n字符型数组元素为:" );
            printArray( charArray ); // 传递一个字符型数组
        }
    }
```

编译以上代码，运行结果如下所示：

```
整型数组元素为:
1 2 3 4 5

双精度型数组元素为:
1.1 2.2 3.3 4.4

字符型数组元素为:
H E L L O
```

有界的类型参数:

可能有时候，你会想限制那些被允许传递到一个类型参数的类型种类范围。例如，一个操作数字的方法可能只希望接受Number或者Number子类的实例。这就是有界类型参数的目的。

要声明一个有界的类型参数，首先列出类型参数的名称，后跟extends关键字，最后紧跟它的上界。

下面的例子演示了"extends"如何使用在一般意义上的意思"extends"（类）或者"implements"（接口）。该例子中的泛型方法返回三个可比较对象的最大值。

```
public class MaximumTest
{
    // 比较三个值并返回最大值
    public static <T extends Comparable<T>> T maximum(T x, T y, T z)
    {
        T max = x; // 假设x是初始最大值
        if ( y.compareTo( max ) > 0 ){
            max = y; //y 更大
        }
        if ( z.compareTo( max ) > 0 ){
            max = z; // 现在 z 更大
        }
```

```
        return max; // 返回最大对象
    }
    public static void main( String args[] )
    {
        System.out.printf( "%d, %d 和 %d 中最大的数为 %d\n\n",
                           3, 4, 5, maximum( 3, 4, 5 ) );

        System.out.printf( "%.1f, %.1f 和 %.1f 中最大的数为 %.1f\n\n",
                           6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

        System.out.printf( "%s, %s 和 %s 中最大的数为 %s\n","pear",
            "apple", "orange", maximum( "pear", "apple", "orange" ) );
    }
}
```

编译以上代码，运行结果如下所示：

```
3, 4 和 5 中最大的数为 5

6.6, 8.8 和 7.7 中最大的数为 8.8

pear, apple 和 orange 中最大的数为 pear
```

泛型类

泛型类的声明和非泛型类的声明类似，除了在类名后面添加了类型参数声明部分。

和泛型方法一样，泛型类的类型参数声明部分也包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。因为他们接受一个或多个参数，这些类被称为参数化的类或参数化的类型。

如下实例演示了我们如何定义一个泛型类:

```
public class Box<T> {

   private T t;

   public void add(T t) {
      this.t = t;
   }

   public T get() {
      return t;
   }

   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
      Box<String> stringBox = new Box<String>();

      integerBox.add(new Integer(10));
      stringBox.add(new String("菜鸟教程"));

      System.out.printf("整型值为 :%d\n\n", integerBox.get());
      System.out.printf("字符串为 :%s\n", stringBox.get());
   }
```

```
23    }
```

编译以上代码，运行结果如下所示：

```
1    整型值为 :10
2
3    字符串为 :菜鸟教程
```

类型通配符

1、类型通配符一般是使用 **?** 代替具体的类型参数。例如 **List<?>** 在逻辑上是 **List,List** 等所有 **List<具体类型实参>** 的父类。

```
1    import java.util.*;
2
3    public class GenericTest {
4
5          public static void main(String[] args) {
6                List<String> name = new ArrayList<String>();
7                List<Integer> age = new ArrayList<Integer>();
8                List<Number> number = new ArrayList<Number>();
9
10               name.add("icon");
11               age.add(18);
12               number.add(314);
13
14               getData(name);
15               getData(age);
16               getData(number);
17
18         }
19
20         public static void getData(List<?> data) {
21               System.out.println("data :" + data.get(0));
22         }
23   }
```

输出结果为：

```
1    data :icon
2    data :18
3    data :314
```

解析： 因为 **getData()** 方法的参数是 **List<?>** 类型的，所以 **name，age，number** 都可以作为这个方法的实参，这就是通配符的作用。

2、类型通配符上限通过形如List来定义，如此定义就是通配符泛型值接受Number及其下层子类类型。

```
1    import java.util.*;
2
3    public class GenericTest {
4
```

```
5         public static void main(String[] args) {
6               List<String> name = new ArrayList<String>();
7               List<Integer> age = new ArrayList<Integer>();
8               List<Number> number = new ArrayList<Number>();
9
10              name.add("icon");
11              age.add(18);
12              number.add(314);
13
14              //getUperNumber(name);//1
15              getUperNumber(age);//2
16              getUperNumber(number);//3
17
18      }
19
20      public static void getData(List<?> data) {
21           System.out.println("data :" + data.get(0));
22      }
23
24      public static void getUperNumber(List<? extends Number> data) {
25                System.out.println("data :" + data.get(0));
26           }
27  }
```

输出结果：

```
1   data :18
2   data :314
```

解析： 在 **//1** 处会出现错误，因为 **getUperNumber()** 方法中的参数已经限定了参数泛型上限为 **Number**，所以泛型为 **String** 是不在这个范围之内，所以会报错。

3、类型通配符下限通过形如 **List<? super Number>** 来定义，表示类型只能接受 **Number** 及其上层父类类型，如 **Object** 类型的实例。

第十六章 并发编程

1. java并发实现

并发实现

3中基本方法和4中拓展方法。

使用线程的3中基本方法

- 实现 Runnable 接口；
- 实现 Callable 接口；
- 继承 Thread 类。

使用四种拓展的方法

- 匿名内部类
- ThreadPool&Executor

- 定时器timer
- stream

实现 Runnable 和 Callable 接口的类只能当做一个可以在线程中运行的任务，不是真正意义上的线程，因此最后还需要通过 Thread 来调用。

通过线程驱动执行线程任务

实现 Runnable 接口

需要实现接口中的 run() 方法。

```
public class MyRunnable implements Runnable {
      @Override
      public void run() {
            // ...
      }
}
```

使用 Runnable 实例再创建一个 Thread 实例，然后调用 Thread 实例的 start() 方法来启动线程。

```
public static void main(String[] args) {
      MyRunnable instance = new MyRunnable();
      Thread thread = new Thread(instance);
      thread.start();
}
```

实现 Callable 接口

与 Runnable 相比，Callable 可以有返回值，返回值通过 FutureTask 进行封装。

```
public class MyCallable implements Callable<Integer> {
      public Integer call() {
            return 123;
      }
}
```

```
public static void main(String[] args) throws ExecutionException, InterruptedException {
      MyCallable mc = new MyCallable();
      FutureTask<Integer> ft = new FutureTask<>(mc);
      Thread thread = new Thread(ft);
      thread.start();
      System.out.println(ft.get());
}
```

继承 Thread 类

同样也是需要实现 run() 方法，因为 Thread 类也实现了 Runable 接口。

当调用 start() 方法启动一个线程时，虚拟机会将该线程放入就绪队列中等待被调度，当一个线程被调度时会执行该线程的 run() 方法。

```
public class MyThread extends Thread {
      public void run() {
            // ...
      }
}
```

```
public static void main(String[] args) {
      MyThread mt = new MyThread();
      mt.start();
}
```

通过内部类的方式实现多线程

直接可以通过Thread类的start()方法进行实现，因为Thread类实现了Runnable接口，并重写了run方法，在run方法中实现自己的逻辑，例如：

```
//这里通过了CountDownLatch，来进行阻塞，来观察两个线程的启动，这样更加体现的明显一些：
public static CountDownLatch countDownLatch=new CountDownLatch(2);

public static void main(String[] args) {
      new Thread(()->{
            countDownLatch.countDown();
            try {
                  countDownLatch.await();
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
            System.out.println("T1");
      }).start();

      new Thread(()->{
            countDownLatch.countDown();
            try {
                  countDownLatch.await();
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
            System.out.println("T2");
      }).start();
}
```

通过线程池来实现多线程

线程池可以根据不同的场景来选择不同的线程池来进行实现

```
//实现代码如下：
public class Demo5 {

      public static void main(String[] args) {
            ExecutorService executorService = Executors.newFixedThreadPool(5);
            for(int i=0;i<5;i++){
                  int finalI = i;
                  executorService.execute(()-> {
                        System.out.println(finalI);
                  });
            }
      }
}
```

通过Timer定时器来实现多线程

就Timer来讲就是一个调度器,而TimerTask呢只是一个实现了run方法的一个类,而具体的TimerTask需要由你自己来实现,同样根据参数得不同存在多种执行方式，例如其中延迟定时任务这样:

```
//具体代码如下：
public class Demo6 {

	public static void main(String[] args) {
		Timer timer=new Timer();
		timer.schedule(new TimerTask(){
			@Override
			public void run() {
				System.out.println(1);
			}
		},20001,10001);
	}
}
```

通过stream实现多线程

jdk1.8 API添加了一个新的抽象称为流Stream，可以让你以一种声明的方式处理数据。
Stream 使用一种类似用 SQL 语句从数据库查询数据的直观方式来提供一种对 Java 集合运算和表达的高阶抽象。
具体简单代码实现如下：

```
//代码实现：
public class Demo7 {

//为了更形象体现并发，通过countDownLatch进行阻塞
static CountDownLatch countDownLatch=new CountDownLatch(6);
	public static void main(String[] args) {
		List list=new ArrayList<>();
		list.add(1);
		list.add(2);
		list.add(3);
		list.add(4);
		list.add(5);
		list.add(6);

		list.parallelStream().forEach(p->{
			//将所有请求在打印之前进行阻塞，方便观察
			countDownLatch.countDown();
			try {
				System.out.println("线程执行到这里啦");
				Thread.sleep(10000);
				countDownLatch.await();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.println(p);
		});
	}
}
```

线程基础设置

Daemon

守护线程是程序运行时在后台提供服务的线程，不属于程序中不可或缺的部分。

当所有非守护线程结束时，程序也就终止，同时会杀死所有守护线程。

main() 属于非守护线程。

在线程启动之前使用 setDaemon() 方法可以将一个线程设置为守护线程。

```
public static void main(String[] args) {
      Thread thread = new Thread(new MyRunnable());
      thread.setDaemon(true);
}
```

sleep

Thread.sleep(millisec) 方法会休眠当前正在执行的线程，millisec 单位为毫秒。

sleep() 可能会抛出 InterruptedException，因为异常不能跨线程传播回 main() 中，因此必须在本地进行处理。线程中抛出的其它异常也同样需要在本地进行处理。

```
public void run() {
      try {
            Thread.sleep(3000);
      } catch (InterruptedException e) {
            e.printStackTrace();
      }
}
```

yield

对静态方法 Thread.yield() 的调用声明了当前线程已经完成了生命周期中最重要的部分，可以切换给其它线程来执行。该方法只是对线程调度器的一个建议，而且也只是建议具有相同优先级的其它线程可以运行。

```
public void run() {
      Thread.yield();
}
```

中断

一个线程执行完毕之后会自动结束，如果在运行过程中发生异常也会提前结束。

InterruptedException

通过调用一个线程的 interrupt() 来中断该线程，如果该线程处于阻塞、限期等待或者无限期等待状态，那么就会抛出 InterruptedException，从而提前结束该线程。但是不能中断 I/O 阻塞和 synchronized 锁阻塞。

对于以下代码，在 main() 中启动一个线程之后再中断它，由于线程中调用了 Thread.sleep() 方法，因此会抛出一个 InterruptedException，从而提前结束线程，不执行之后的语句。

```
public class InterruptExample {

      private static class MyThread1 extends Thread {
            @Override
            public void run() {
```

```
                try {
                    Thread.sleep(2000);
                    System.out.println("Thread run");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

```
public static void main(String[] args) throws InterruptedException {
        Thread thread1 = new MyThread1();
        thread1.start();
        thread1.interrupt();
        System.out.println("Main run");
}
```

```
Main run
java.lang.InterruptedException: sleep interrupted
     at java.lang.Thread.sleep(Native Method)
     at InterruptExample.lambda$main$0(InterruptExample.java:5)
     at InterruptExample$$Lambda$1/713338599.run(Unknown Source)
     at java.lang.Thread.run(Thread.java:745)
```

interrupted()

如果一个线程的 run() 方法执行一个无限循环，并且没有执行 sleep() 等会抛出 InterruptedException 的操作，那么调用线程的 interrupt() 方法就无法使线程提前结束。

但是调用 interrupt() 方法会设置线程的中断标记，此时调用 interrupted() 方法会返回 true。因此可以在循环体中使用 interrupted() 方法来判断线程是否处于中断状态，从而提前结束线程。

```
 public class InterruptExample {

        private static class MyThread2 extends Thread {
            @Override
            public void run() {
                while (!interrupted()) {
                    // ..
                }
                System.out.println("Thread end");
            }
        }
 }
```

```
public static void main(String[] args) throws InterruptedException {
        Thread thread2 = new MyThread2();
        thread2.start();
        thread2.interrupt();
}
```

```
Thread end
```

Executor 的中断操作

调用 Executor 的 shutdown() 方法会等待线程都执行完毕之后再关闭，但是如果调用的是 shutdownNow() 方法，则相当于调用每个线程的 interrupt() 方法。

以下使用 Lambda 创建线程，相当于创建了一个匿名内部线程。

```
public static void main(String[] args) {
      ExecutorService executorService = Executors.newCachedThreadPool();
      executorService.execute(() -> {
            try {
                  Thread.sleep(2000);
                  System.out.println("Thread run");
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
      });
      executorService.shutdownNow();
      System.out.println("Main run");
}
```

```
Main run
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at ExecutorInterruptExample.lambda$main$0(ExecutorInterruptExample.java:9)
    at ExecutorInterruptExample$$Lambda$1/1160460865.run(Unknown Source)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
    at java.lang.Thread.run(Thread.java:745)
```

如果只想中断 Executor 中的一个线程，可以通过使用 submit() 方法来提交一个线程，它会返回一个 Future<?> 对象，通过调用该对象的 cancel(true) 方法就可以中断线程。

```
Future<?> future = executorService.submit(() -> {
      // ..
});
future.cancel(true);
```

线程状态转换

一个线程只能处于一种状态，并且这里的线程状态特指 Java 虚拟机的线程状态，不能反映线程在特定操作系统下的状态。

新建（NEW）

创建后尚未启动。

可运行（RUNABLE）

正在 Java 虚拟机中运行。但是在操作系统层面，它可能处于运行状态，也可能等待资源调度（例如处理器资源），资源调度完成就进入运行状态。所以该状态的可运行是指可以被运行，具体有没有运行要看底层操作系统的资源调度。

阻塞（BLOCKED）

请求获取 monitor lock 从而进入 synchronized 函数或者代码块，但是其它线程已经占用了该 monitor lock，所以出于阻塞状态。要结束该状态进入从而 RUNABLE 需要其他线程释放 monitor lock。

无限期等待（WAITING）

等待其它线程显式地唤醒。

阻塞和等待的区别在于，阻塞是被动的，它是在等待获取 monitor lock。而等待是主动的，通过调用 Object.wait() 等方法进入。

进入方法	退出方法
没有设置 Timeout 参数的 Object.wait() 方法	Object.notify() / Object.notifyAll()
没有设置 Timeout 参数的 Thread.join() 方法	被调用的线程执行完毕
LockSupport.park() 方法	LockSupport.unpark(Thread)

限期等待（TIMED_WAITING）

无需等待其它线程显式地唤醒，在一定时间之后会被系统自动唤醒。

进入方法	退出方法
Thread.sleep() 方法	时间结束
设置了 Timeout 参数的 Object.wait() 方法	时间结束 / Object.notify() / Object.notifyAll()
设置了 Timeout 参数的 Thread.join() 方法	时间结束 / 被调用的线程执行完毕
LockSupport.parkNanos() 方法	LockSupport.unpark(Thread)
LockSupport.parkUntil() 方法	LockSupport.unpark(Thread)

调用 Thread.sleep() 方法使线程进入限期等待状态时，常常用“使一个线程睡眠”进行描述。调用 Object.wait() 方法使线程进入限期等待或者无限期等待时，常常用“挂起一个线程”进行描述。睡眠和挂起是用来描述行为，而阻塞和等待用来描述状态。

死亡（TERMINATED）

可以是线程结束任务之后自己结束，或者产生了异常而结束

2. 互斥同步

互斥同步

Java 提供了两种锁机制来控制多个线程对共享资源的互斥访问，第一个是 JVM 实现的 synchronized，而另一个是 JDK 实现的 ReentrantLock。

synchronized

1. 同步一个代码块

```
public void func() {
      synchronized (this) {
            // ...
      }
}
```

它只作用于同一个对象，如果调用两个对象上的同步代码块，就不会进行同步。

对于以下代码，使用 ExecutorService 执行了两个线程，由于调用的是同一个对象的同步代码块，因此这两个线程会进行同步，当一个线程进入同步语句块时，另一个线程就必须等待。

```
 public class SynchronizedExample {

       public void func1() {
             synchronized (this) {
                   for (int i = 0; i < 10; i++) {
                         System.out.print(i + " ");
                   }
             }
       }
 }
```

```
public static void main(String[] args) {
      SynchronizedExample e1 = new SynchronizedExample();
      ExecutorService executorService = Executors.newCachedThreadPool();
      executorService.execute(() -> e1.func1());
      executorService.execute(() -> e1.func1());
}
```

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
```

对于以下代码，两个线程调用了不同对象的同步代码块，因此这两个线程就不需要同步。从输出结果可以看出，两个线程交叉执行。

```
public static void main(String[] args) {
      SynchronizedExample e1 = new SynchronizedExample();
      SynchronizedExample e2 = new SynchronizedExample();
      ExecutorService executorService = Executors.newCachedThreadPool();
      executorService.execute(() -> e1.func1());
      executorService.execute(() -> e2.func1());
}
```

```
0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
```

2. 同步一个方法

- 编写同步方法的一般语法如下。 这里的lockObject是对对象的引用，该对象的锁与同步语句表示的监视器相关联。
    - '.class' object -如果方法是静态的。
    - this' object -如果方法不是静态的。 “ this”是指对其中调用同步方法的当前对象的引用。

```
public synchronized void func () {
      // ...
}
```

它和同步代码块一样，作用于同一个对象。

3. 同步一个类

```
public void func() {
      synchronized (SynchronizedExample.class) {
            // ...
      }
}
```

作用于整个类，也就是说两个线程调用同一个类的不同对象上的这种同步语句，也会进行同步。

```
 public class SynchronizedExample {

       public void func2() {
             synchronized (SynchronizedExample.class) {
                   for (int i = 0; i < 10; i++) {
                         System.out.print(i + " ");
                   }
             }
       }
 }
```

```
public static void main(String[] args) {
      SynchronizedExample e1 = new SynchronizedExample();
      SynchronizedExample e2 = new SynchronizedExample();
      ExecutorService executorService = Executors.newCachedThreadPool();
      executorService.execute(() -> e1.func2());
      executorService.execute(() -> e2.func2());
}
```

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
```

4. 同步一个静态方法

```
public synchronized static void fun() {
      // ...
}
```

作用于整个类。

对象级别和类级别的同步

- 当我们要同步non-static method non-static code block时， Object level lock是一种机制，这样，只有一个线程将能够在给定的类实例上执行代码块。 应该始终这样做， 以确保实例级数据线程安全 。

```
 public class DemoClass
 {
       public synchronized void demoMethod(){}
 }

 or

 public class DemoClass
 {
       public void demoMethod(){
             synchronized (this)
             {
                   //other thread safe code
```

```
            }
        }
    }

    or

    public class DemoClass
    {
          private final Object lock = new Object();
          public void demoMethod(){
                synchronized (lock)
                {
                      //other thread safe code
                }
          }
    }
```

- Class level lock可防止多个线程在运行时在该类的所有可用实例中的任何一个中进入synchronized块。 这意味着，如果在运行时有100个demoMethod()实例，则一次只能在一个实例中的任何一个线程上执行demoMethod()，而所有其他实例将被其他线程锁定。

```
    public class DemoClass
    {
          //Method is static
          public synchronized static void demoMethod(){

          }
    }

    or

    public class DemoClass
    {
          public void demoMethod()
          {
                //Acquire lock on .class reference
                synchronized (DemoClass.class)
                {
                      //other thread safe code
                }
          }
    }

    or

    public class DemoClass
    {
          private final static Object lock = new Object();

          public void demoMethod()
          {
                //Lock object is static
                synchronized (lock)
                {
                      //other thread safe code
                }
          }
    }
```

注意事项

- Java中的同步保证了没有两个线程可以同时或并发执行需要相同锁的同步方法。
- synchronized关键字只能与方法和代码块一起使用。 这些方法或块可以是static也可以non-static 。
- 每当线程进入Java synchronized方法或块时，它都会获取一个锁，而当线程离开同步方法或块时，它将释放该锁。 即使线程在完成后或由于任何错误或异常而离开同步方法时，也会释放锁定。
- Java synchronized关键字本质上是可re-entrant ，这意味着如果一个同步方法调用另一个需要相同锁的同步方法，则持有锁的当前线程可以进入该方法而无需获取锁。
- 如果在同步块中使用的对象为null，则Java同步将引发NullPointerException 。 例如，在上面的代码示例中，如果将锁初始化为null，则“ synchronized (lock) ”将抛出NullPointerException 。
- Java中的同步方法使您的应用程序性能降低。 因此，在绝对需要时使用同步。 另外，请考虑使用同步的代码块仅同步代码的关键部分。
- 静态同步方法和非静态同步方法都可能同时或同时运行，因为它们锁定在不同的对象上。
- 根据Java语言规范，您不能在构造函数中使用synchronized关键字。 这是非法的，并导致编译错误。
- 不要在Java中的同步块上的非final字段上进行同步。 因为非最终字段的引用可能随时更改，然后不同的线程可能会在不同的对象上进行同步，即完全没有同步。
- 不要使用String文字，因为它们可能在应用程序中的其他地方被引用，并且可能导致死锁。 使用new关键字创建的字符串对象可以安全使用。 但作为最佳实践，请在我们要保护的共享变量本身上创建一个新的private作用域Object实例OR锁。

ReentrantLock

ReentrantLock 是 java.util.concurrent（J.U.C）包中的锁。

```
1   public class LockExample {
2
3         private Lock lock = new ReentrantLock();
4
5         public void func() {
6               lock.lock();
7               try {
8                     for (int i = 0; i < 10; i++) {
9                           System.out.print(i + " ");
10                    }
11              } finally {
12                    lock.unlock(); // 确保释放锁，从而避免发生死锁。
13              }
14        }
15  }
```

```
1   public static void main(String[] args) {
2         LockExample lockExample = new LockExample();
3         ExecutorService executorService = Executors.newCachedThreadPool();
4         executorService.execute(() -> lockExample.func());
5         executorService.execute(() -> lockExample.func());
6   }
```

```
1   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
```

比较

除非需要使用 ReentrantLock 的高级功能，否则优先使用 synchronized。这是因为 synchronized 是 JVM 实现的一种锁机制，JVM 原生地支持它，而 ReentrantLock 不是所有的 JDK 版本都支持。并且使用 synchronized 不用担心没有释放锁而导致死锁问题，因为 JVM 会确保锁的释放。

1. 锁的实现

synchronized 是 JVM 实现的，而 ReentrantLock 是 JDK 实现的。

2. 性能

新版本 Java 对 synchronized 进行了很多优化，例如自旋锁等，synchronized 与 ReentrantLock 大致相同。

3. 等待可中断

当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情。

ReentrantLock 可中断，而 synchronized 不行。

4. 公平锁

公平锁是指多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁。

synchronized 中的锁是非公平的，ReentrantLock 默认情况下也是非公平的，但是也可以是公平的。

5. 锁绑定多个条件

一个 ReentrantLock 可以同时绑定多个 Condition 对象。

线程之间的协作

当多个线程可以一起工作去解决某个问题时，如果某些部分必须在其它部分之前完成，那么就需要对线程进行协调。

Thread:join

在线程中调用另一个线程的 join() 方法，会将当前线程挂起，而不是忙等待，直到目标线程结束。

对于以下代码，虽然 b 线程先启动，但是因为在 b 线程中调用了 a 线程的 join() 方法，b 线程会等待 a 线程结束才继续执行，因此最后能够保证 a 线程的输出先于 b 线程的输出。

```
public class JoinExample {

    private class A extends Thread {
        @Override
        public void run() {
            System.out.println("A");
        }
    }

    private class B extends Thread {

        private A a;

        B(A a) {
            this.a = a;
        }

```

```
            @Override
            public void run() {
                  try {
                        a.join();
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
                  System.out.println("B");
            }
      }

      public void test() {
            A a = new A();
            B b = new B(a);
            b.start();
            a.start();
      }
}
```

```
public static void main(String[] args) {
      JoinExample example = new JoinExample();
      example.test();
}
```

```
A
B
```

Object:wait、notify、notifyAll

调用 wait() 使得线程等待某个条件满足，线程在等待时会被挂起，当其他线程的运行使得这个条件满足时，其它线程会调用 notify() 或者 notifyAll() 来唤醒挂起的线程。

它们都属于 Object 的一部分，而不属于 Thread。

只能用在同步方法或者同步控制块中使用，否则会在运行时抛出 IllegalMonitorStateException。

使用 wait() 挂起期间，线程会释放锁。这是因为，如果没有释放锁，那么其它线程就无法进入对象的同步方法或者同步控制块中，那么就无法执行 notify() 或者 notifyAll() 来唤醒挂起的线程，造成死锁。

```
public class WaitNotifyExample {

      public synchronized void before() {
            System.out.println("before");
            notifyAll();
      }

      public synchronized void after() {
            try {
                  wait();
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
            System.out.println("after");
      }
}
```

```
public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        WaitNotifyExample example = new WaitNotifyExample();
        executorService.execute(() -> example.after());
        executorService.execute(() -> example.before());
}
```

```
before
after
```

wait() 和 sleep() 的区别

- wait() 是 Object 的方法，而 sleep() 是 Thread 的静态方法；
- wait() 会释放锁，sleep() 不会。

Condition:await、signal、signalAll

java.util.concurrent 类库中提供了 Condition 类来实现线程之间的协调，可以在 Condition 上调用 await() 方法使线程等待，其它线程调用 signal() 或 signalAll() 方法唤醒等待的线程。

相比于 wait() 这种等待方式，await() 可以指定等待的条件，因此更加灵活。

使用 Lock 来获取一个 Condition 对象。

```
 public class AwaitSignalExample {

        private Lock lock = new ReentrantLock();
        private Condition condition = lock.newCondition();

        public void before() {
              lock.lock();
              try {
                    System.out.println("before");
                    condition.signalAll();
              } finally {
                    lock.unlock();
              }
        }

        public void after() {
              lock.lock();
              try {
                    condition.await();
                    System.out.println("after");
              } catch (InterruptedException e) {
                    e.printStackTrace();
              } finally {
                    lock.unlock();
              }
        }
 }
```

```
public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        AwaitSignalExample example = new AwaitSignalExample();
        executorService.execute(() -> example.after());
        executorService.execute(() -> example.before());
}
```

```
before
after
```

3. 进程通信

Pipe

PipedReader是Reader类的扩展，用于读取字符流。 它的read（）方法读取连接的PipedWriter的流。 同样， PipedWriter是Writer类的扩展，它完成Reader类所收缩的所有工作。

- 读线程

```
public class PipeReaderThread implements Runnable
{
      PipedReader pr;
      String name = null;

      public PipeReaderThread(String name, PipedReader pr)
      {
            this.name = name;
            this.pr = pr;
      }

      public void run()
      {
            try {
                  // continuously read data from stream and print it in console
                  while (true) {
                        char c = (char) pr.read(); // read a char
                        if (c != -1) { // check for -1 indicating end of file
                              System.out.print(c);
                        }
                  }
            } catch (Exception e) {
                  System.out.println(" PipeThread Exception: " + e);
            }
      }
}
```

- 写线程

```
public class PipeWriterThread implements Runnable
{
      PipedWriter pw;
      String name = null;

      public PipeWriterThread(String name, PipedWriter pw) {
            this.name = name;
            this.pw = pw;
      }

      public void run() {
            try {
                  while (true) {
                        // Write some data after every two seconds
                        pw.write("Testing data written...n");
```

```
                        pw.flush();
                        Thread.sleep(2000);
                    }
                } catch (Exception e) {
                        System.out.println(" PipeThread Exception: " + e);
                }
        }
}
```

- 测试代码

```
package multiThread;

import java.io.*;

public class PipedCommunicationTest
{
      public static void main(String[] args)
      {
            new PipedCommunicationTest();
      }

      public PipedCommunicationTest()
      {
            try
            {
                  // Create writer and reader instances
                  PipedReader pr = new PipedReader();
                  PipedWriter pw = new PipedWriter();

                  // Connect the writer with reader
                  pw.connect(pr);

                  // Create one writer thread and one reader thread
                  Thread thread1 = new Thread(new PipeReaderThread("ReaderThread", pr));

                  Thread thread2 = new Thread(new PipeWriterThread("WriterThread", pw));

                  // start both threads
                  thread1.start();
                  thread2.start();

            }
            catch (Exception e)
            {
                  System.out.println("PipeThread Exception: " + e);
            }
      }
}
```

BlockQueue（新的最佳实践）

```
package corejava.thread;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.RejectedExecutionHandler;
import java.util.concurrent.ThreadPoolExecutor;
```

```
import java.util.concurrent.TimeUnit;

public class DemoExecutor
{
      public static void main(String[] args)
      {
            Integer threadCounter = 0;
            BlockingQueue<Runnable> blockingQueue = new ArrayBlockingQueue<Runnable>(50);

            CustomThreadPoolExecutor executor = new CustomThreadPoolExecutor(10,
            20, 5000, TimeUnit.MILLISECONDS, blockingQueue);

            executor.setRejectedExecutionHandler(new RejectedExecutionHandler() {
                  @Override
                  public void rejectedExecution(Runnable r,
                              ThreadPoolExecutor executor) {
                        System.out.println("DemoTask Rejected : "
                                    + ((DemoTask) r).getName());
                        System.out.println("Waiting for a second !!");
                        try {
                              Thread.sleep(1000);
                        } catch (InterruptedException e) {
                              e.printStackTrace();
                        }
                        System.out.println("Lets add another time : "
                                    + ((DemoTask) r).getName());
                        executor.execute(r);
                  }
            });
             // Let start all core threads initially
             executor.prestartAllCoreThreads();
             while (true) {
                  threadCounter++;
                  // Adding threads one by one
                  System.out.println("Adding DemoTask : " + threadCounter);
                  executor.execute(new DemoTask(threadCounter.toString()));

                  if (threadCounter == 100)
                        break;
            }
      }

}

```

4. 线程池

Executor概述

概述

Executor framework框架由三个主要接口（以及许多子接口）组成，即Executor， ExecutorService和ThreadPoolExecutor 。

- 该框架主要将任务创建和执行分开。 任务创建主要是样板代码，并且很容易替换。
- 对于执行者，我们必须创建实现Runnable或Callable接口的任务，并将其发送给执行者。

- 执行程序在内部维护一个（可配置的）线程池，以通过避免连续产生线程来提高应用程序性能。
- 执行程序负责执行任务，并使用池中的必要线程运行它们。

分类

Executor 管理多个异步任务的执行，而无需程序员显式地管理线程的生命周期。这里的异步是指多个任务的执行互不干扰，不需要进行同步操作。

- CachedThreadPool：缓存线程池执行程序 -创建一个线程池，该线程池可根据需要创建新线程，但在可用时将重用以前构造的线程。 如果任务长时间运行，请勿使用此线程池。 如果线程数超出系统可以处理的范围，则可能导致系统崩溃。
- FixedThreadPool：固定线程池执行程序 -创建一个线程池，该线程池可重用固定数量的线程来执行任意数量的任务。 如果在所有线程都处于活动状态时提交了其他任务，则它们将在队列中等待，直到某个线程可用为止。 最适合现实生活中的大多数用例。
- SingleThreadExecutor：相当于大小为 1 的 FixedThreadPool，单线程池执行程序 -创建单线程以执行所有任务。 当您只有一个任务要执行时，请使用它。
- ScheduledThreadPool：调度线程池执行程序 -创建一个线程池，该线程池可以调度命令以在给定延迟后运行或定期执行。
- ForkJoinPool：分支合并线程池，适合用于处理复杂任务。初始化线程容量与 CPU 核心数相关。线程池中运行的内容必须是 ForkJoinTask 的子类型（RecursiveTask,RecursiveAction）。
- WorkStealingPool：JDK1.8 新增的线程池。工作窃取线程池。当线程池中有空闲连接时，自动到等待队列中窃取未完成任务，自动执行。初始化线程容量与 CPU 核心数相关。此线程池中维护的是精灵线程。ExecutorService.newWorkStealingPool();

ThreadPoolExecutor类具有四个不同的构造函数，但是由于它们的复杂性，Java并发API提供了Executors类来构造执行程序和其他相关对象。 尽管我们可以使用其构造函数之一直接创建ThreadPoolExecutor，但是建议使用Executors类。

```
public static void main(String[] args) {
      ExecutorService executorService = Executors.newCachedThreadPool();
      for (int i = 0; i < 5; i++) {
            executorService.execute(new MyRunnable());
     }
      executorService.shutdown();
}
```

使用

1. 通过executors工具类创建.Executors是一个实用程序类，它提供用于创建接口实现的工厂方法。

```
//Executes only one thread
ExecutorService es = Executors.newSingleThreadExecutor();

//Internally manages thread pool of 2 threads
ExecutorService es = Executors.newFixedThreadPool(2);

//Internally manages thread pool of 10 threads to run scheduled tasks
ExecutorService es = Executors.newScheduledThreadPool(10);
```

2. 创建ExecutorService。我们可以选择ExecutorService接口的实现类，然后直接创建它的实例。 下面的语句创建一个线程池执行程序，该线程池执行程序的最小线程数为10，最大线程数为100，存活时间为5毫秒，并且有一个阻塞队列来监视将来的任务。

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

ExecutorService executorService = new ThreadPoolExecutor(10, 100, 5L, TimeUnit.MILLISECONDS,
                                                                    new
LinkedBlockingQueue<Runnable>());
```

3. Execute Runnable tasks

    1. void execute(Runnable task) –在将来的某个时间执行给定命令。
    2. Future submit(Runnable task)可运行Future submit(Runnable task) –提交要执行的可运行任务，并返回代表该任务的Future。 Future的get()方法成功完成后将返回null。
    3. 提交可运行任务以执行并返回代表该任务的Future。 Future的get()方法将在成功完成后返回给定的result

```
import java.time.LocalDateTime;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class Main
{
      public static void main(String[] args)
     {
            //Demo task
            Runnable runnableTask = () -> {
                  try {
                        TimeUnit.MILLISECONDS.sleep(1000);
                        System.out.println("Current time :: " + LocalDateTime.now());
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            };

            //Executor service instance
            ExecutorService executor = Executors.newFixedThreadPool(10);

            //1. execute task using execute() method
            executor.execute(runnableTask);

            //2. execute task using submit() method
            Future<String> result = executor.submit(runnableTask, "DONE");

            while(result.isDone() == false)
            {
                  try
                  {
                        System.out.println("The method return value : " + result.get());
                        break;
                  }
                  catch (InterruptedException | ExecutionException e)
                  {
                        e.printStackTrace();
                  }
```

```
42
43                  //Sleep for 1 second
44                  try {
45                        Thread.sleep(1000L);
46                  } catch (InterruptedException e) {
47                        e.printStackTrace();
48                  }
49            }
50
51             //Shut down the executor service
52             executor.shutdownNow();
53       }
54  }
```

4. Execute Callable tasks

    1. Future submit(callableTask) -提交一个执行返回值的任务，并返回代表该任务的未决结果的未来。
    2. List invokeAll(Collection tasks) -执行给定的任务，并when all complete返回保存其状态和结果的Future列表。 注意，仅当所有任务完成时结果才可用。请注意，已完成的任务可能已正常终止，也可能引发异常。
    3. List invokeAll（Collection task，timeOut，timeUnit） -执行给定的任务，并在所有完成或超时到期时返回保存其状态和结果的Future列表。

```
1   import java.time.LocalDateTime;
2   import java.util.Arrays;
3   import java.util.List;
4   import java.util.concurrent.Callable;
5   import java.util.concurrent.ExecutionException;
6   import java.util.concurrent.ExecutorService;
7   import java.util.concurrent.Executors;
8   import java.util.concurrent.Future;
9   import java.util.concurrent.TimeUnit;
10
11  public class Main
12  {
13        public static void main(String[] args) throws ExecutionException
14       {
15              //Demo Callable task
16              Callable<String> callableTask = () -> {
17                    TimeUnit.MILLISECONDS.sleep(1000);
18                    return "Current time :: " + LocalDateTime.now();
19              };
20
21              //Executor service instance
22              ExecutorService executor = Executors.newFixedThreadPool(1);
23
24              List<Callable<String>> tasksList = Arrays.asList(callableTask, callableTask,
    callableTask);
25
26              //1. execute tasks list using invokeAll() method
27              try
28              {
29                    List<Future<String>> results = executor.invokeAll(tasksList);
30
31                    for(Future<String> result : results) {
32                          System.out.println(result.get());
33                    }
```

```
            }
             catch (InterruptedException e1)
            {
                  e1.printStackTrace();
            }

             //2. execute individual tasks using submit() method
             Future<String> result = executor.submit(callableTask);

             while(result.isDone() == false)
            {
                  try
                 {
                        System.out.println("The method return value : " + result.get());
                        break;
                 }
                  catch (InterruptedException | ExecutionException e)
                 {
                        e.printStackTrace();
                 }

                  //Sleep for 1 second
                  try {
                        Thread.sleep(1000L);
                 } catch (InterruptedException e) {
                        e.printStackTrace();
                 }
            }

             //Shut down the executor service
             executor.shutdownNow();
      }
}
```

FixedThreadPool

- 如果有任何任务引发异常，则应用程序将捕获该异常并重新启动该任务。
- 如果有任何任务运行完毕，应用程序将注意到并重新启动任务。

```
package com.howtodoinjava.multiThreading.executors;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class DemoExecutorUsage {

      private static ExecutorService executor = null;
      private static volatile Future taskOneResults = null;
      private static volatile Future taskTwoResults = null;

      public static void main(String[] args) {
            executor = Executors.newFixedThreadPool(2);
            while (true)
            {
                  try
                 {
                        checkTasks();
```

```
                        Thread.sleep(1000);
                } catch (Exception e) {
                        System.err.println("Caught exception: " + e.getMessage());
                }
            }
        }

        private static void checkTasks() throws Exception {
             if (taskOneResults == null
                        || taskOneResults.isDone()
                        || taskOneResults.isCancelled())
            {
                  taskOneResults = executor.submit(new TestOne());
            }

             if (taskTwoResults == null
                        || taskTwoResults.isDone()
                        || taskTwoResults.isCancelled())
            {
                  taskTwoResults = executor.submit(new TestTwo());
            }
        }
    }

    class TestOne implements Runnable {
          public void run() {
                while (true)
                {
                      System.out.println("Executing task one");
                      try
                     {
                            Thread.sleep(1000);
                     } catch (Throwable e) {
                            e.printStackTrace();
                     }
                }

          }
    }

    class TestTwo implements Runnable {
          public void run() {
                while (true)
                {
                      System.out.println("Executing task two");
                      try
                     {
                            Thread.sleep(1000);
                     } catch (Throwable e) {
                            e.printStackTrace();
                     }
                }
          }
    }
```

ForkJoinPool

主要用于并行计算中，和 MapReduce 原理类似，都是把大的计算任务拆分成多个小任务并行计算。

```
public class ForkJoinExample extends RecursiveTask<Integer> {

        private final int threshold = 5;
        private int first;
        private int last;

        public ForkJoinExample(int first, int last) {
              this.first = first;
              this.last = last;
        }

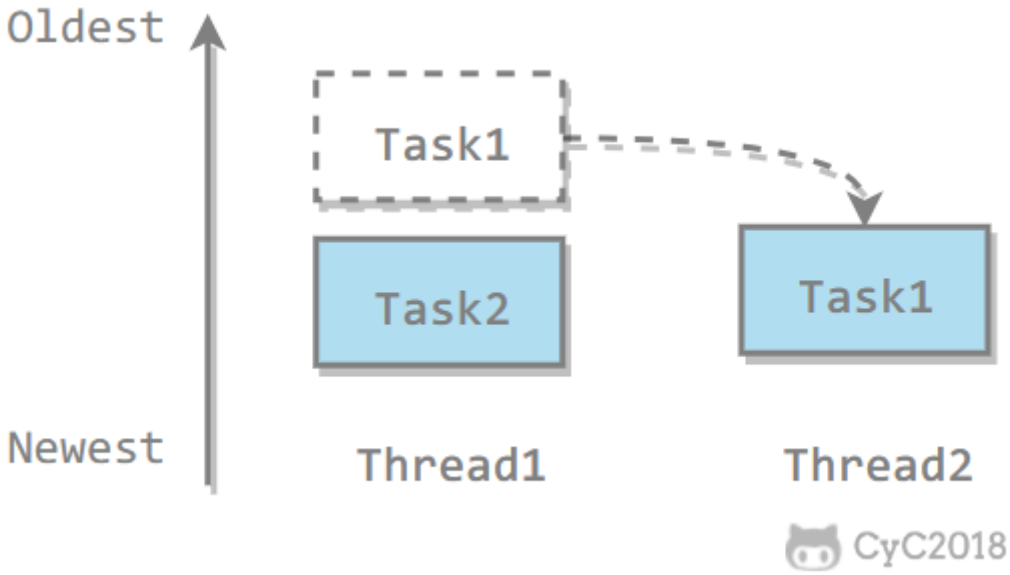
        @Override
        protected Integer compute() {
              int result = 0;
              if (last - first <= threshold) {
                    // 任务足够小则直接计算
                    for (int i = first; i <= last; i++) {
                          result += i;
                    }
              } else {
                    // 拆分成小任务
                    int middle = first + (last - first) / 2;
                    ForkJoinExample leftTask = new ForkJoinExample(first, middle);
                    ForkJoinExample rightTask = new ForkJoinExample(middle + 1, last);
                    leftTask.fork();
                    rightTask.fork();
                    result = leftTask.join() + rightTask.join();
              }
              return result;
        }
}
```

```
public static void main(String[] args) throws ExecutionException, InterruptedException {
        ForkJoinExample example = new ForkJoinExample(1, 10000);
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        Future result = forkJoinPool.submit(example);
        System.out.println(result.get());
}
```

ForkJoin 使用 ForkJoinPool 来启动，它是一个特殊的线程池，线程数量取决于 CPU 核数。

```
public class ForkJoinPool extends AbstractExecutorService
```

ForkJoinPool 实现了工作窃取算法来提高 CPU 的利用率。每个线程都维护了一个双端队列，用来存储需要执行的任务。工作窃取算法允许空闲的线程从其它线程的双端队列中窃取一个任务来执行。窃取的任务必须是最晚的任务，避免和队列所属线程发生竞争。例如下图中，Thread2 从 Thread1 的队列中拿出最晚的 Task1 任务，Thread1 会拿出 Task2 来执行，这样就避免发生竞争。但是如果队列中只有一个任务时还是会发生竞争。



ScheduledThreadPool

而是您可能要在一段时间后执行任务或定期执行任务。 为此，Executor框架提供ScheduledThreadPoolExecutor类。

1. 实现一个任务

```
class Task implements Runnable
{
      private String name;

      public Task(String name) {
            this.name = name;
      }

      public String getName() {
            return name;
      }

      @Override
      public void run()
      {
            try {
                  System.out.println("Doing a task during : " + name + " - Time - " + new Date());
            }
            catch (Exception e) {
                  e.printStackTrace();
            }
      }
}
```

2. Execute a task after a period of time.

```
package com.howtodoinjava.demo.multithreading;

import java.util.Date;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ScheduledThreadPoolExecutorExample
```

```
{
      public static void main(String[] args)
      {
            ScheduledExecutorService executor = Executors.newScheduledThreadPool(2);
            Task task1 = new Task ("Demo Task 1");
            Task task2 = new Task ("Demo Task 2");

            System.out.println("The time is : " + new Date());

            executor.schedule(task1, 5 , TimeUnit.SECONDS);
            executor.schedule(task2, 10 , TimeUnit.SECONDS);

            try {
                  executor.awaitTermination(1, TimeUnit.DAYS);
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }

            executor.shutdown();
      }
}

Output:

The time is : Wed Mar 25 16:14:07 IST 2015
Doing a task during : Demo Task 1 - Time - Wed Mar 25 16:14:12 IST 2015
Doing a task during : Demo Task 2 - Time - Wed Mar 25 16:14:17 IST 2015
```

3. Execute a task periodically

```
public class ScheduledThreadPoolExecutorExample
{
      public static void main(String[] args)
      {
            ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
            Task task1 = new Task ("Demo Task 1");

            System.out.println("The time is : " + new Date());

            ScheduledFuture<?> result = executor.scheduleAtFixedRate(task1, 2, 5,
TimeUnit.SECONDS);

            try {
                  TimeUnit.MILLISECONDS.sleep(20000);
            }
            catch (InterruptedException e) {
                  e.printStackTrace();
            }

            executor.shutdown();
      }
}

Output:

The time is : Wed Mar 25 16:20:12 IST 2015
Doing a task during : Demo Task 1 - Time - Wed Mar 25 16:20:14 IST 2015
Doing a task during : Demo Task 1 - Time - Wed Mar 25 16:20:19 IST 2015
```

```
Doing a task during : Demo Task 1 - Time - Wed Mar 25 16:20:24 IST 2015
Doing a task during : Demo Task 1 - Time - Wed Mar 25 16:20:29 IST 2015
```

5. JUC并发组件

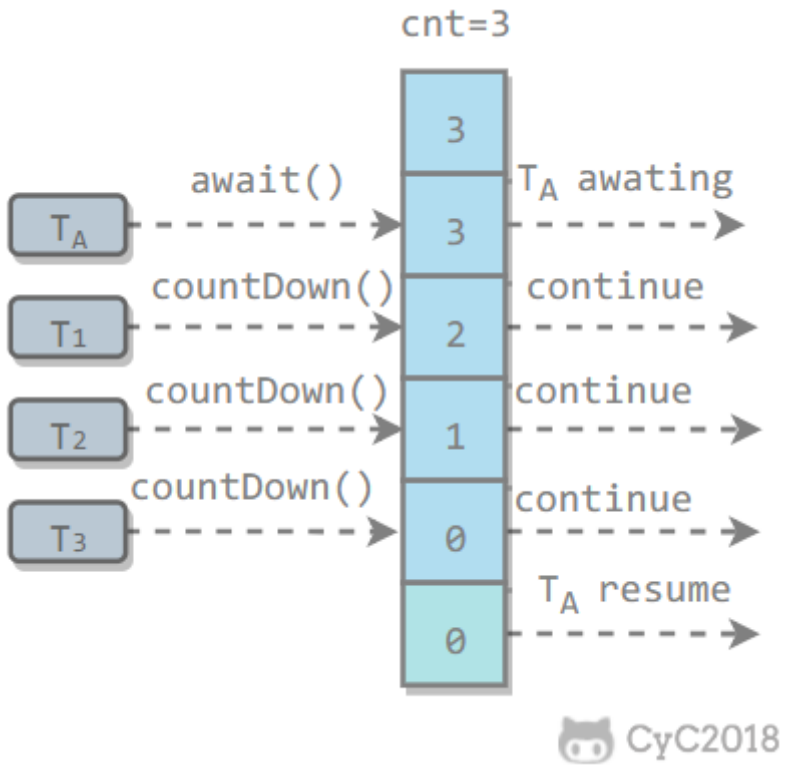
J.U.C - AQS

java.util.concurrent（J.U.C）大大提高了并发性能，AQS 被认为是 J.U.C 的核心。

CountDownLatch

用来控制一个或者多个线程等待多个线程。

维护了一个计数器 cnt，每次调用 countDown() 方法会让计数器的值减 1，减到 0 的时候，那些因为调用 await() 方法而在等待的线程就会被唤醒。




```
public class CountdownLatchExample {

      public static void main(String[] args) throws InterruptedException {
            final int totalThread = 10;
            CountDownLatch countDownLatch = new CountDownLatch(totalThread);
            ExecutorService executorService = Executors.newCachedThreadPool();
            for (int i = 0; i < totalThread; i++) {
                  executorService.execute(() -> {
                        System.out.print("run..");
                        countDownLatch.countDown();
                  });
            }
            countDownLatch.await();
            System.out.println("end");
            executorService.shutdown();
      }
}
```

```
run..run..run..run..run..run..run..run..run..run..end
```

CyclicBarrier

用来控制多个线程互相等待，只有当多个线程都到达时，这些线程才会继续执行。

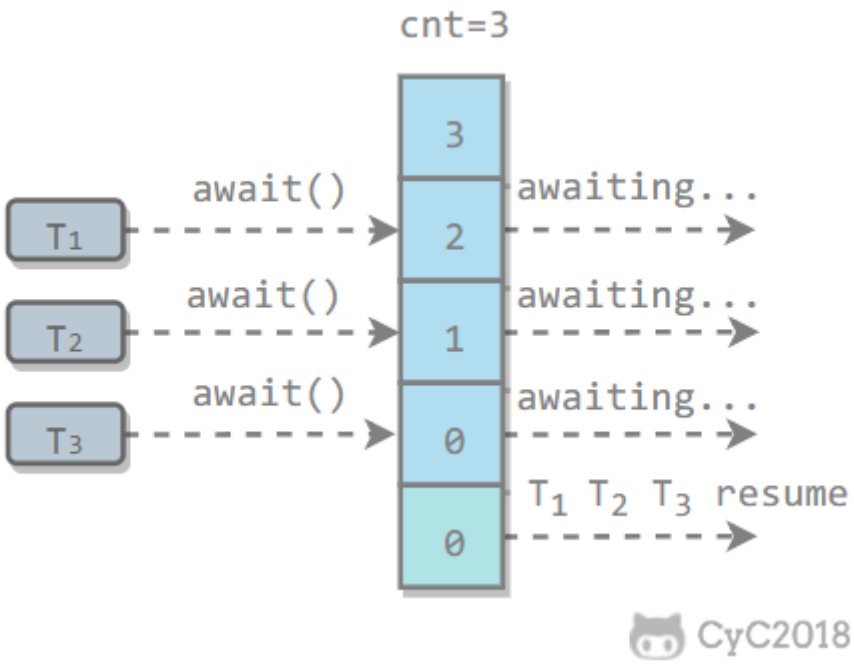
和 CountdownLatch 相似，都是通过维护计数器来实现的。线程执行 await() 方法之后计数器会减 1，并进行等待，直到计数器为 0，所有调用 await() 方法而在等待的线程才能继续执行。

CyclicBarrier 和 CountdownLatch 的一个区别是，CyclicBarrier 的计数器通过调用 reset() 方法可以循环使用，所以它才叫做循环屏障。

CyclicBarrier 有两个构造函数，其中 parties 指示计数器的初始值，barrierAction 在所有线程都到达屏障的时候会执行一次。

```
public CyclicBarrier(int parties, Runnable barrierAction) {
      if (parties <= 0) throw new IllegalArgumentException();
      this.parties = parties;
      this.count = parties;
      this.barrierCommand = barrierAction;
}

public CyclicBarrier(int parties) {
      this(parties, null);
}
```




```
public class CyclicBarrierExample {

      public static void main(String[] args) {
            final int totalThread = 10;
            CyclicBarrier cyclicBarrier = new CyclicBarrier(totalThread);
            ExecutorService executorService = Executors.newCachedThreadPool();
            for (int i = 0; i < totalThread; i++) {
                  executorService.execute(() -> {
                        System.out.print("before..");
                        try {
                              cyclicBarrier.await();
                        } catch (InterruptedException | BrokenBarrierException e) {
                              e.printStackTrace();
```

```
                        }
                        System.out.print("after..");
                    });
            }
            executorService.shutdown();
        }
    }
```

```
before..before..before..before..before..before..before..before..before..before..after..after..after..after..after..after..after..after..after..after..
```

Semaphore

Semaphore 类似于操作系统中的信号量，可以控制对互斥资源的访问线程数。

以下代码模拟了对某个服务的并发请求，每次只能有 3 个客户端同时访问，请求总数为 10。

```
public class SemaphoreExample {

    public static void main(String[] args) {
        final int clientCount = 3;
        final int totalRequestCount = 10;
        Semaphore semaphore = new Semaphore(clientCount);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < totalRequestCount; i++) {
            executorService.execute(()->{
                try {
                    semaphore.acquire();
                    System.out.print(semaphore.availablePermits() + " ");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    semaphore.release();
                }
            });
        }
        executorService.shutdown();
    }
}
```

```
2 1 2 2 2 2 2 1 2 2
```

FutureTask

在介绍 Callable 时我们知道它可以有返回值，返回值通过 Future<V> 进行封装。FutureTask 实现了 RunnableFuture 接口，该接口继承自 Runnable 和 Future<V> 接口，这使得 FutureTask 既可以当做一个任务执行，也可以有返回值。

```
public class FutureTask<V> implements RunnableFuture<V>
```

```
public interface RunnableFuture<V> extends Runnable, Future<V>
```

FutureTask 可用于异步获取执行结果或取消执行任务的场景。当一个计算任务需要执行很长时间，那么就可以用 FutureTask 来封装这个任务，主线程在完成自己的任务之后再去获取结果。

```
public class FutureTaskExample {

    public static void main(String[] args) throws ExecutionException, InterruptedException {
```

```
            FutureTask<Integer> futureTask = new FutureTask<Integer>(new Callable<Integer>() {
                  @Override
                  public Integer call() throws Exception {
                        int result = 0;
                        for (int i = 0; i < 100; i++) {
                              Thread.sleep(10);
                              result += i;
                        }
                        return result;
                  }
            });

            Thread computeThread = new Thread(futureTask);
            computeThread.start();

            Thread otherThread = new Thread(() -> {
                  System.out.println("other task is running...");
                  try {
                        Thread.sleep(1000);
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            });
            otherThread.start();
            System.out.println(futureTask.get());
      }
}
```

```
other task is running...
4950
```

J.U.C -并发容器

并发集合是指使用了最新并发能力的集合，在JUC包下。而同步集合指之前用同步锁实现的集合

CopyOnWrite

CopyOnWriteArrayList在写的时候会复制一个副本，对副本写，写完用副本替换原值，读的时候不需要同步，适用于写少读多的场合。

CopyOnWriteArraySet基于CopyOnWriteArrayList来实现的，只是在不允许存在重复的对象这个特性上遍历处理了一下。

BlockingQueue

在并发队列上JDK提供了两套实现，

- 一个是以ConcurrentLinkedQueue为代表的高性能队列
- 一个是以BlockingQueue接口为代表的阻塞队列。

ConcurrentLinkedQueue适用于高并发场景下的队列，通过无锁的方式实现，通常ConcurrentLinkedQueue的性能要优于BlockingQueue。BlockingQueue的典型应用场景是生产者-消费者模式中，如果生产快于消费，生产队列装满时会阻塞，等待消费。

java.util.concurrent.BlockingQueue 接口有以下阻塞队列的实现：

- **FIFO 队列** ： LinkedBlockingQueue、ArrayBlockingQueue（固定长度）
- **优先级队列** ： PriorityBlockingQueue

提供了阻塞的 take() 和 put() 方法：如果队列为空 take() 将阻塞，直到队列中有内容；如果队列为满 put() 将阻塞，直到队列有空闲位置。

使用 BlockingQueue 实现生产者消费者问题

```
public class ProducerConsumer {

	private static BlockingQueue<String> queue = new ArrayBlockingQueue<>(5);

	private static class Producer extends Thread {
		@Override
		public void run() {
			try {
				queue.put("product");
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.print("produce..");
		}
	}

	private static class Consumer extends Thread {

		@Override
		public void run() {
			try {
				String product = queue.take();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.print("consume..");
		}
	}
}
```

```
public static void main(String[] args) {
	for (int i = 0; i < 2; i++) {
		Producer producer = new Producer();
		producer.start();
	}
	for (int i = 0; i < 5; i++) {
		Consumer consumer = new Consumer();
		consumer.start();
	}
	for (int i = 0; i < 3; i++) {
		Producer producer = new Producer();
		producer.start();
	}
}
```

```
produce..produce..consume..consume..produce..consume..produce..consume..produce..consume..
```

Concurrent

- ConcurrentLinkedQueue
- ConcurrentLinkedDeque
- ConcurrentHashMap
- ConcurrentHashSet

- ConcurrentSkipListMap
- ConcurrentSkipListSet

ConcurrentHashMap是专用于高并发的Map实现，内部实现进行了锁分离，get操作是无锁的。

java api也提供了一个实现ConcurrentSkipListMap接口的类，ConcurrentSkipListMap接口实现了与ConcurrentNavigableMap接口有相同行为的一个非阻塞式列表。从内部实现机制来讲，它使用了一个Skip List来存放数据。Skip List是基于并发列表的数据结构，效率与二叉树相近。
当插入元素到映射中时，ConcurrentSkipListMap接口类使用键值来排序所有元素。除了提供返回一个具体元素的方法外，这个类也提供获取子映射的方法。

ConcurrentSkipListMap类提供的常用方法：

```
1.headMap(K toKey)：K是在ConcurrentSkipListMap对象的 泛型参数里用到的键。这个方法返回映射中所有键值小于参数值toKey的子映射。
2.tailMap(K fromKey)：K是在ConcurrentSkipListMap对象的 泛型参数里用到的键。这个方法返回映射中所有键值大于参数值fromKey的子映射。
3.putIfAbsent(K key,V value)：如果映射中不存在键key，那么就将key和value保存到映射中。
4.pollLastEntry()：返回并移除映射中的最后一个Map.Entry对象。
5.replace(K key,V value)：如果映射中已经存在键key，则用参数中的value替换现有的值。
```

6. 线程安全

线程安全概述

线程不安全示例

如果多个线程对同一个共享数据进行访问而不采取同步操作的话，那么操作的结果是不一致的。

以下代码演示了 1000 个线程同时对 cnt 执行自增操作，操作结束之后它的值有可能小于 1000。

```
public class ThreadUnsafeExample {

	private int cnt = 0;

	public void add() {
		cnt++;
	}

	public int get() {
		return cnt;
	}
}
```

```
public static void main(String[] args) throws InterruptedException {
	final int threadSize = 1000;
	ThreadUnsafeExample example = new ThreadUnsafeExample();
	final CountDownLatch countDownLatch = new CountDownLatch(threadSize);
```

```
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < threadSize; i++) {
            executorService.execute(() -> {
                example.add();
                countDownLatch.countDown();
            });
        }
        countDownLatch.await();
        executorService.shutdown();
        System.out.println(example.get());
    }
```

```
997
```

线程安全

多个线程不管以何种方式访问某个类，并且在主调代码中不需要进行同步，都能表现正确的行为。

线程安全有以下几种实现方式：

- 不可变对象
- 无同步方案
- 互斥同步
- 非阻塞同步

不可变对象

不可变（Immutable）的对象一定是线程安全的，不需要再采取任何的线程安全保障措施。只要一个不可变的对象被正确地构建出来，永远也不会看到它在多个线程之中处于不一致的状态。多线程环境下，应当尽量使对象成为不可变，来满足线程安全。

不可变的类型：

- final 关键字修饰的基本数据类型
- String
- 枚举类型
- Number 部分子类，如 Long 和 Double 等数值包装类型，BigInteger 和 BigDecimal 等大数据类型。但同为 Number 的原子类 AtomicInteger 和 AtomicLong 则是可变的。

对于集合类型，可以使用 Collections.unmodifiableXXX() 方法来获取一个不可变的集合。

```
public class ImmutableExample {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        Map<String, Integer> unmodifiableMap = Collections.unmodifiableMap(map);
        unmodifiableMap.put("a", 1);
    }
}
```

```
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableMap.put(Collections.java:1457)
    at ImmutableExample.main(ImmutableExample.java:9)
```

Collections.unmodifiableXXX() 先对原始的集合进行拷贝，需要对集合进行修改的方法都直接抛出异常。

```
public V put(K key, V value) {
    throw new UnsupportedOperationException();
}
```

无同步方案

要保证线程安全，并不是一定就要进行同步。如果一个方法本来就不涉及共享数据，那它自然就无须任何同步措施去保证正确性。

栈封闭

多个线程访问同一个方法的局部变量时，不会出现线程安全问题，因为局部变量存储在虚拟机栈中，属于线程私有的。

```
public class StackClosedExample {
      public void add100() {
            int cnt = 0;
            for (int i = 0; i < 100; i++) {
                  cnt++;
            }
            System.out.println(cnt);
      }
}
```

```
public static void main(String[] args) {
      StackClosedExample example = new StackClosedExample();
      ExecutorService executorService = Executors.newCachedThreadPool();
      executorService.execute(() -> example.add100());
      executorService.execute(() -> example.add100());
      executorService.shutdown();
}
```

```
100
100
```

线程本地存储（Thread Local Storage）

如果一段代码中所需要的数据必须与其他代码共享，那就看看这些共享数据的代码是否能保证在同一个线程中执行。如果能保证，我们就可以把共享数据的可见范围限制在同一个线程之内，这样，无须同步也能保证线程之间不出现数据争用的问题。

符合这种特点的应用并不少见，大部分使用消费队列的架构模式（如“生产者-消费者”模式）都会将产品的消费过程尽量在一个线程中消费完。其中最重要的一个应用实例就是经典 Web 交互模型中的“一个请求对应一个服务器线程”（Thread-per-Request）的处理方式，这种处理方式的广泛应用使得很多 Web 服务端应用都可以使用线程本地存储来解决线程安全问题。

可以使用 java.lang.ThreadLocal 类来实现线程本地存储功能。

对于以下代码，thread1 中设置 threadLocal 为 1，而 thread2 设置 threadLocal 为 2。过了一段时间之后，thread1 读取 threadLocal 依然是 1，不受 thread2 的影响。

```
public class ThreadLocalExample {
      public static void main(String[] args) {
            ThreadLocal threadLocal = new ThreadLocal();
            Thread thread1 = new Thread(() -> {
                  threadLocal.set(1);
                  try {
                        Thread.sleep(1000);
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
                  System.out.println(threadLocal.get());
```

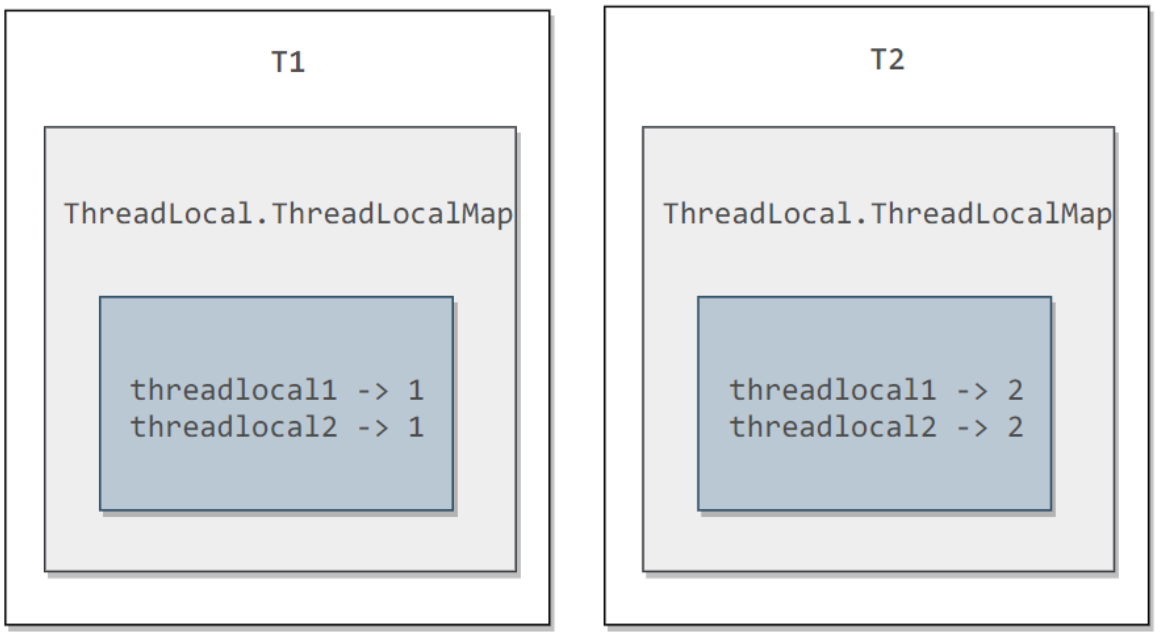
```
                threadLocal.remove();
            });
            Thread thread2 = new Thread(() -> {
                threadLocal.set(2);
                threadLocal.remove();
            });
            thread1.start();
            thread2.start();
        }
    }
```

```
1
```

为了理解 ThreadLocal，先看以下代码：

```
public class ThreadLocalExample1 {
    public static void main(String[] args) {
        ThreadLocal threadLocal1 = new ThreadLocal();
        ThreadLocal threadLocal2 = new ThreadLocal();
        Thread thread1 = new Thread(() -> {
            threadLocal1.set(1);
            threadLocal2.set(1);
        });
        Thread thread2 = new Thread(() -> {
            threadLocal1.set(2);
            threadLocal2.set(2);
        });
        thread1.start();
        thread2.start();
    }
}
```

它所对应的底层结构图为：



每个 Thread 都有一个 ThreadLocal.ThreadLocalMap 对象。

```
/* ThreadLocal values pertaining to this thread. This map is maintained
 * by the ThreadLocal class. */
ThreadLocal.ThreadLocalMap threadLocals = null;
```

当调用一个 ThreadLocal 的 set(T value) 方法时，先得到当前线程的 ThreadLocalMap 对象，然后将 ThreadLocal->value 键值对插入到该 Map 中。

```
public void set(T value) {
      Thread t = Thread.currentThread();
      ThreadLocalMap map = getMap(t);
      if (map != null)
            map.set(this, value);
      else
            createMap(t, value);
}
```

get() 方法类似。

```
 public T get() {
       Thread t = Thread.currentThread();
       ThreadLocalMap map = getMap(t);
       if (map != null) {
             ThreadLocalMap.Entry e = map.getEntry(this);
             if (e != null) {
                   @SuppressWarnings("unchecked")
                   T result = (T)e.value;
                   return result;
             }
       }
       return setInitialValue();
 }
```

ThreadLocal 从理论上讲并不是用来解决多线程并发问题的，因为根本不存在多线程竞争。

在一些场景 (尤其是使用线程池) 下，由于 ThreadLocal.ThreadLocalMap 的底层数据结构导致 ThreadLocal 有内存泄漏的情况，应该尽可能在每次使用 ThreadLocal 后手动调用 remove()，以避免出现 ThreadLocal 经典的内存泄漏甚至是造成自身业务混乱的风险。

可重入代码（Reentrant Code）

这种代码也叫做纯代码（Pure Code），可以在代码执行的任何时刻中断它，转而去执行另外一段代码（包括递归调用它本身），而在控制权返回后，原来的程序不会出现任何错误。

可重入代码有一些共同的特征，例如不依赖存储在堆上的数据和公用的系统资源、用到的状态量都由参数中传入、不调用非可重入的方法等。

A Stateless Servlet

```
public class StatelessFactorizer implements Servlet
{
      public void service(ServletRequest req, ServletResponse resp)
      {
            BigInteger i = extractFromRequest(req);
            BigInteger[] factors = factor(i);
            encodeIntoResponse(resp, factors);
      }
}
```

互斥同步

悲观同步，损失性能

synchronized 和 ReentrantLock。

Object.wait/notify

Condition.await/signal

非阻塞同步

互斥同步最主要的问题就是线程阻塞和唤醒所带来的性能问题，因此这种同步也称为阻塞同步。

互斥同步属于一种悲观的并发策略，总是认为只要不去做正确的同步措施，那就肯定会出现问题。无论共享数据是否真的会出现竞争，它都要进行加锁（这里讨论的是概念模型，实际上虚拟机会优化掉很大一部分不必要的加锁）、用户态核心态转换、维护锁计数器和检查是否有被阻塞的线程需要唤醒等操作。

随着硬件指令集的发展，我们可以使用基于冲突检测的乐观并发策略：先进行操作，如果没有其它线程争用共享数据，那操作就成功了，否则采取补偿措施（不断地重试，直到成功为止）。这种乐观的并发策略的许多实现都不需要将线程阻塞，因此这种同步操作称为非阻塞同步。

CAS(Compare and Swap Algorithm)

乐观锁需要操作和冲突检测这两个步骤具备原子性，这里就不能再使用互斥同步来保证了，只能靠硬件来完成。硬件支持的原子性操作最典型的是：比较并交换（Compare-and-Swap，CAS）。CAS 指令需要有 3 个操作数，分别是

- 内存地址 V、
- 旧的预期值 A
- 新值 B。

当执行操作时，只有当 V 的值等于 A，才将 V 的值更新为 B。

让我们通过一个例子来了解整个过程。 假设V是存储值“ 10”的存储位置。 有多个线程想要递增此值并将递增的值用于其他操作，这是一种非常实际的方案。 让我们分步分解整个CAS操作：

1. 线程1和2想要增加它，它们都读取值并将其增加到11。

```
V = 10, A = 0, B = 0
```

2. 现在线程1首先出现，并将V与它的最后一个读取值进行比较：

```
V = 10, A = 10, B = 11

if       A = V
    V = B
 else
    operation failed
    return V
显然，V的值将被覆盖为11，即操作成功。
```

3. 线程2到来并尝试与线程1相同的操作

```
V = 11, A = 10, B = 11

if      A = V
    V = B
 else
    operation failed
    return V
```

4. 在这种情况下，V不等于A，因此不替换值，并返回V的当前值，即11。 现在，线程2再次使用值重试此操作：

```
V = 11, A = 11, B = 12
```

而这一次，条件得到满足，增量值12返回线程2。

总而言之，当多个线程尝试使用CAS同时更新同一变量时，一个将获胜并更新该变量的值，而其余则将丢失。 但是失败者并不会因为线程中断而受到惩罚。 他们可以自由地重试该操作，或者什么也不做。

AtomicInteger

J.U.C 包里面的整数原子类 AtomicInteger 的方法调用了 Unsafe 类的 CAS 操作。

以下代码使用了 AtomicInteger 执行了自增的操作。

```
private AtomicInteger cnt = new AtomicInteger();

public void add() {
      cnt.incrementAndGet();
}
```

以下代码是 incrementAndGet() 的源码，它调用了 Unsafe 的 getAndAddInt() 。

```
public final int incrementAndGet() {
      return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
}
```

以下代码是 getAndAddInt() 源码，var1 指示对象内存地址，var2 指示该字段相对对象内存地址的偏移，var4 指示操作需要加的数值，这里为 1。通过 getIntVolatile(var1, var2) 得到旧的预期值，通过调用 compareAndSwapInt() 来进行 CAS 比较，如果该字段内存地址中的值等于 var5，那么就更新内存地址为 var1+var2 的变量为 var5+var4。

可以看到 getAndAddInt() 在一个循环中进行，发生冲突的做法是不断的进行重试。

```
public final int getAndAddInt(Object var1, long var2, int var4) {
      int var5;
      do {
            var5 = this.getIntVolatile(var1, var2);
      } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

      return var5;
}
```

ABA

如果一个变量初次读取的时候是 A 值，它的值被改成了 B，后来又被改回为 A，那 CAS 操作就会误认为它从来没有被改变过。

J.U.C 包提供了一个带有标记的原子引用类 AtomicStampedReference 来解决这个问题，它可以通过控制变量值的版本来保证 CAS 的正确性。大部分情况下 ABA 问题不会影响程序并发的正确性，如果需要解决 ABA 问题，改用传统的互斥同步可能会比原子类更高效。

7. 内存模型

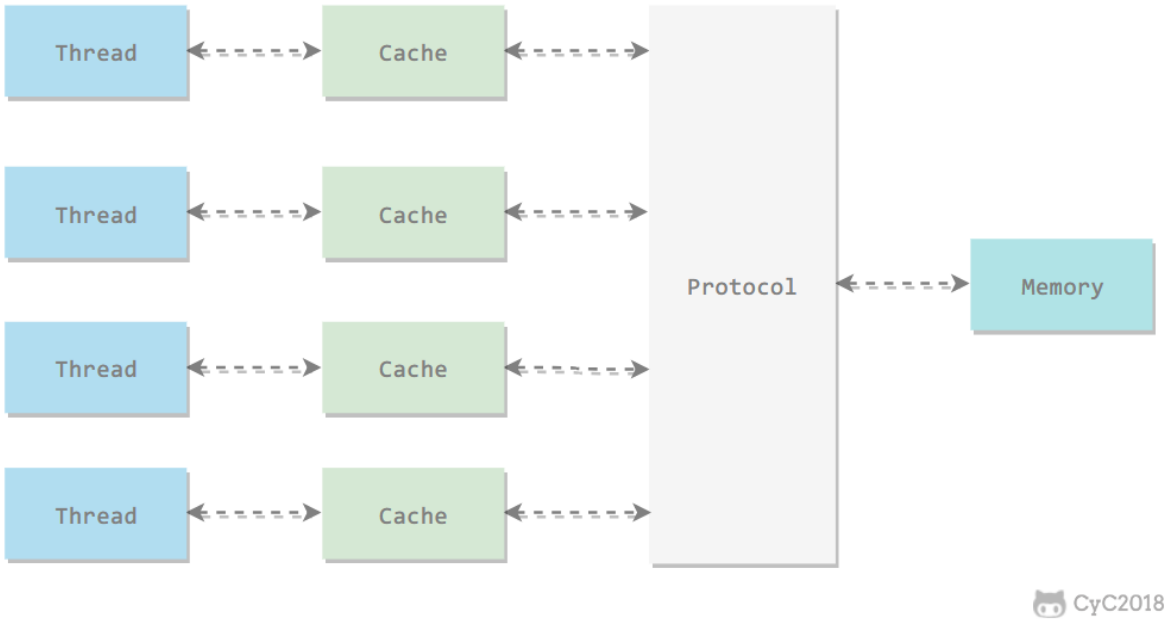
Java 内存模型

Java 内存模型试图屏蔽各种硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致的内存访问效果。

主内存与工作内存

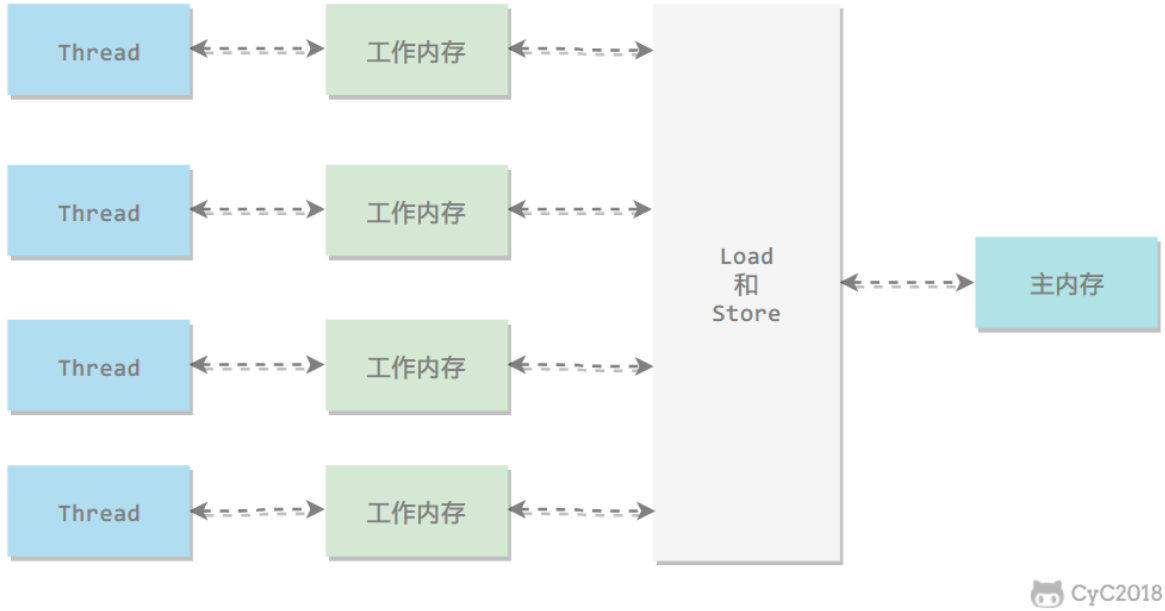
处理器上的寄存器的读写的速度比内存快几个数量级，为了解决这种速度矛盾，在它们之间加入了高速缓存。

加入高速缓存带来了一个新的问题：缓存一致性。如果多个缓存共享同一块主内存区域，那么多个缓存的数据可能会不一致，需要一些协议来解决这个问题。



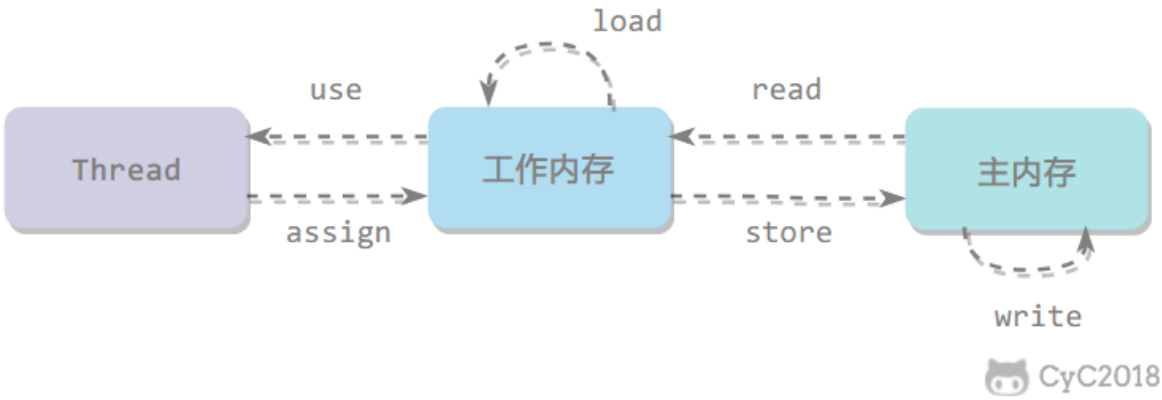
所有的变量都存储在主内存中，每个线程还有自己的工作内存，工作内存存储在高速缓存或者寄存器中，保存了该线程使用的变量的主内存副本拷贝。

线程只能直接操作工作内存中的变量，不同线程之间的变量值传递需要通过主内存来完成。



内存间交互操作

Java 内存模型定义了 8 个操作来完成主内存和工作内存的交互操作。



- read：把一个变量的值从主内存传输到工作内存中
- load：在 read 之后执行，把 read 得到的值放入工作内存的变量副本中
- use：把工作内存中一个变量的值传递给执行引擎
- assign：把一个从执行引擎接收到的值赋给工作内存的变量
- store：把工作内存的一个变量的值传送到主内存中
- write：在 store 之后执行，把 store 得到的值放入主内存的变量中
- lock：作用于主内存的变量
- unlock

内存模型三大特性

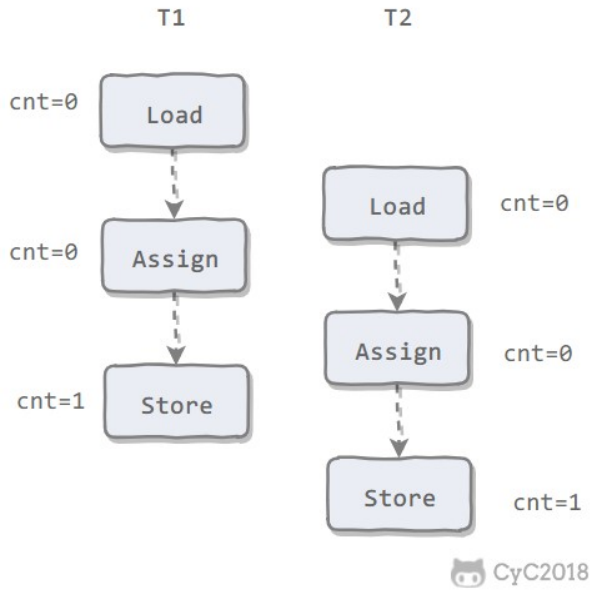
原子性

Java 内存模型保证了 read、load、use、assign、store、write、lock 和 unlock 操作具有原子性，例如对一个 int 类型的变量执行 assign 赋值操作，这个操作就是原子性的。但是 Java 内存模型允许虚拟机将没有被 volatile 修饰的 64 位数据（long，double）的读写操作划分为两次 32 位的操作来进行，即 load、store、read 和 write 操作可以不具备原子性。

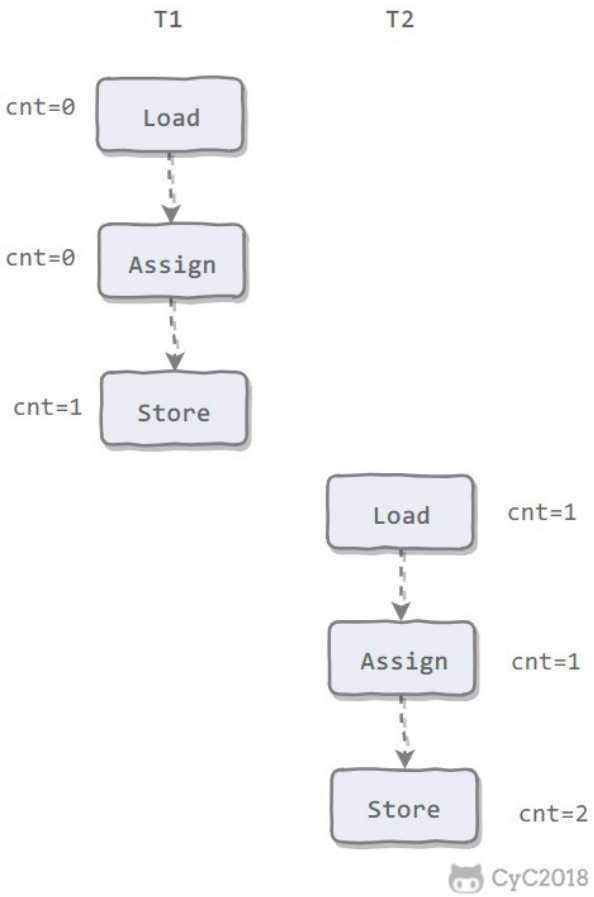
有一个错误认识就是，int 等原子性的类型在多线程环境中不会出现线程安全问题。前面的线程不安全示例代码中，cnt 属于 int 类型变量，1000 个线程对它进行自增操作之后，得到的值为 997 而不是 1000。

为了方便讨论，将内存间的交互操作简化为 3 个：load、assign、store。

下图演示了两个线程同时对 cnt 进行操作，load、assign、store 这一系列操作整体上看不具备原子性，那么在 T1 修改 cnt 并且还没有将修改后的值写入主内存，T2 依然可以读入旧值。可以看出，这两个线程虽然执行了两次自增运算，但是主内存中 cnt 的值最后为 1 而不是 2。因此对 int 类型读写操作满足原子性只是说明 load、assign、store 这些单个操作具备原子性。



AtomicInteger 能保证多个线程修改的原子性。



使用 AtomicInteger 重写之前线程不安全的代码之后得到以下线程安全实现：

```
public class AtomicExample {
	private AtomicInteger cnt = new AtomicInteger();

	public void add() {
		cnt.incrementAndGet();
	}

	public int get() {
		return cnt.get();
	}
}
```

```
public static void main(String[] args) throws InterruptedException {
	final int threadSize = 1000;
	AtomicExample example = new AtomicExample(); // 只修改这条语句
	final CountDownLatch countDownLatch = new CountDownLatch(threadSize);
	ExecutorService executorService = Executors.newCachedThreadPool();
	for (int i = 0; i < threadSize; i++) {
		executorService.execute(() -> {
			example.add();
			countDownLatch.countDown();
		});
	}
	countDownLatch.await();
	executorService.shutdown();
	System.out.println(example.get());
}
```

```
1000
```

除了使用原子类之外，也可以使用 synchronized 互斥锁来保证操作的原子性。它对应的内存间交互操作为：lock 和 unlock，在虚拟机实现上对应的字节码指令为 monitorenter 和 monitorexit。

```
public class AtomicSynchronizedExample {
	private int cnt = 0;

	public synchronized void add() {
		cnt++;
	}

	public synchronized int get() {
		return cnt;
	}
}
```

```
public static void main(String[] args) throws InterruptedException {
	final int threadSize = 1000;
	AtomicSynchronizedExample example = new AtomicSynchronizedExample();
	final CountDownLatch countDownLatch = new CountDownLatch(threadSize);
	ExecutorService executorService = Executors.newCachedThreadPool();
	for (int i = 0; i < threadSize; i++) {
		executorService.execute(() -> {
			example.add();
			countDownLatch.countDown();
		});
	}
	countDownLatch.await();
```

```
        executorService.shutdown();
        System.out.println(example.get());
    }
```

```
1000
```

可见性

可见性指当一个线程修改了共享变量的值，其它线程能够立即得知这个修改。Java 内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值来实现可见性的。

主要有三种实现可见性的方式：

- volatile
- synchronized，对一个变量执行 unlock 操作之前，必须把变量值同步回主内存。
- final，被 final 关键字修饰的字段在构造器中一旦初始化完成，并且没有发生 this 逃逸（其它线程通过 this 引用访问到初始化了一半的对象），那么其它线程就能看见 final 字段的值。

对前面的线程不安全示例中的 cnt 变量使用 volatile 修饰，不能解决线程不安全问题，因为 volatile 并不能保证操作的原子性。

有序性

有序性是指：在本线程内观察，所有操作都是有序的。在一个线程观察另一个线程，所有操作都是无序的，无序是因为发生了指令重排序。在 Java 内存模型中，允许编译器和处理器对指令进行重排序，重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

volatile 关键字通过添加内存屏障的方式来禁止指令重排，即重排序时不能把后面的指令放到内存屏障之前。

也可以通过 synchronized 来保证有序性，它保证每个时刻只有一个线程执行同步代码，相当于是让线程顺序执行同步代码。

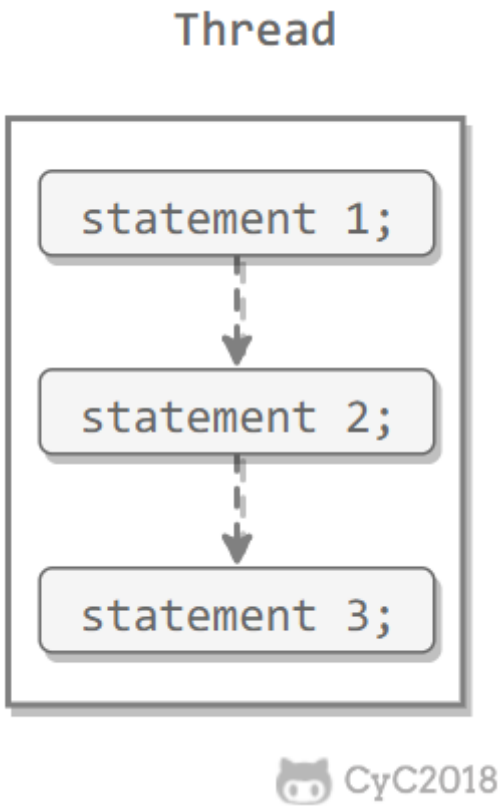
先行发生原则

上面提到了可以用 volatile 和 synchronized 来保证有序性。除此之外，JVM 还规定了先行发生原则，让一个操作无需控制就能先于另一个操作完成。

单一线程原则

> Single Thread rule

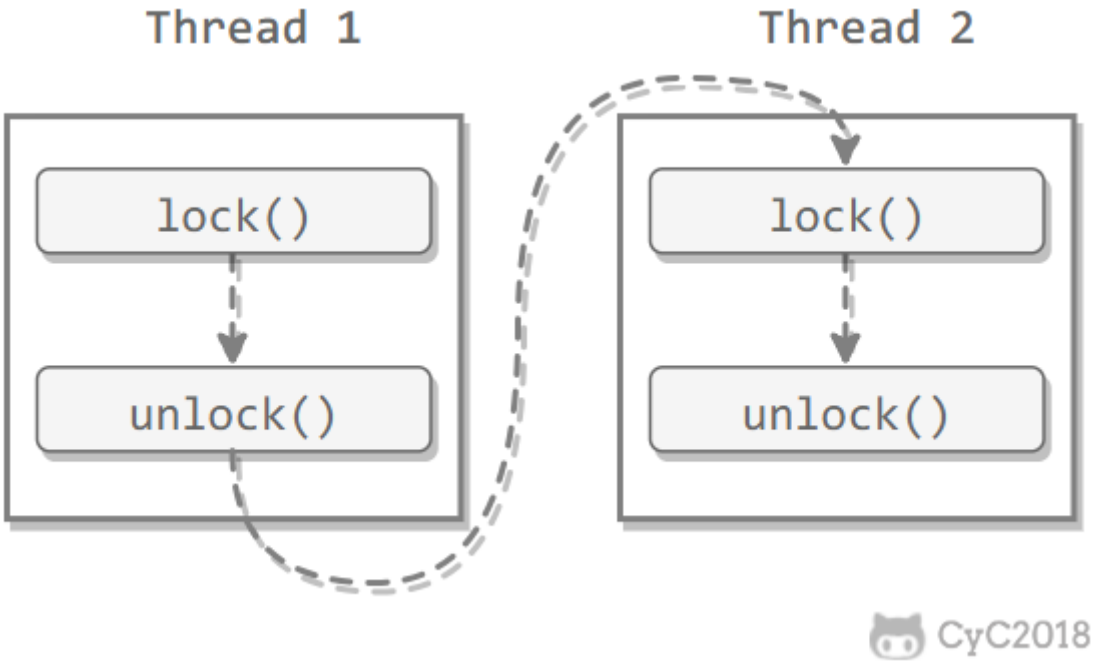
在一个线程内，在程序前面的操作先行发生于后面的操作。



管程锁定规则

> Monitor Lock Rule

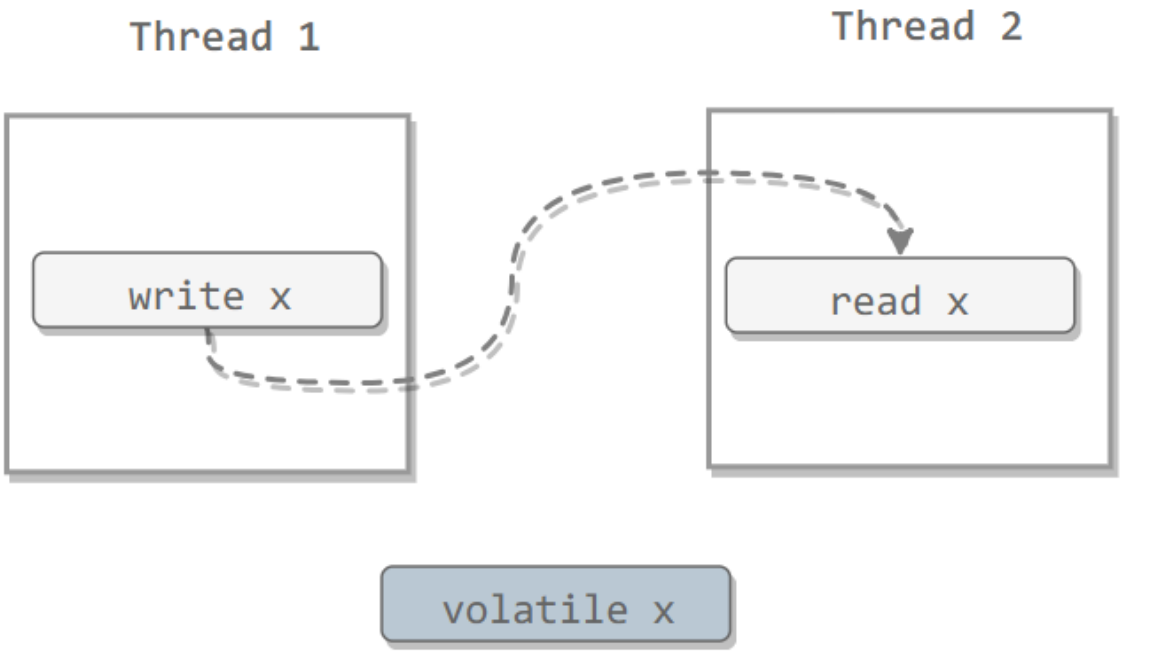
一个 unlock 操作先行发生于后面对同一个锁的 lock 操作。



volatile 变量规则

> Volatile Variable Rule

对一个 volatile 变量的写操作先行发生于后面对这个变量的读操作。

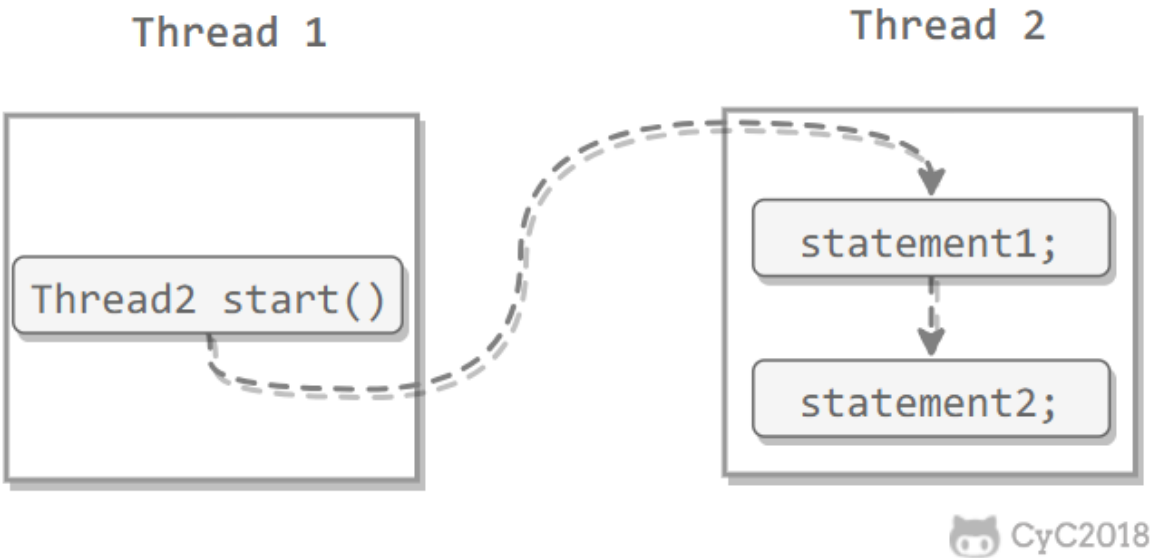


CyC2018

线程启动规则

> Thread Start Rule

Thread 对象的 start() 方法调用先行发生于此线程的每一个动作。

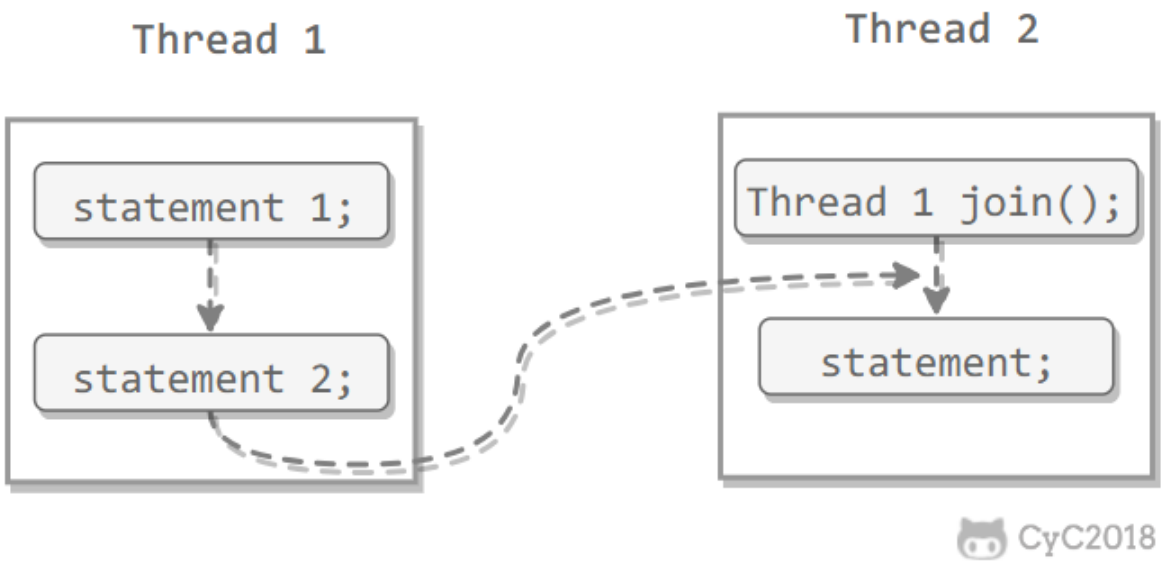


CyC2018

线程加入规则

> Thread Join Rule

Thread 对象的结束先行发生于 join() 方法返回。



线程中断规则

> Thread Interruption Rule

对线程 interrupt() 方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过 interrupted() 方法检测到是否有中断发生。

对象终结规则

> Finalizer Rule

一个对象的初始化完成（构造函数执行结束）先行发生于它的 finalize() 方法的开始。

传递性

> Transitivity

如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那么操作 A 先行发生于操作 C。

第十七章 IO流与网络编程

1. 标准IO

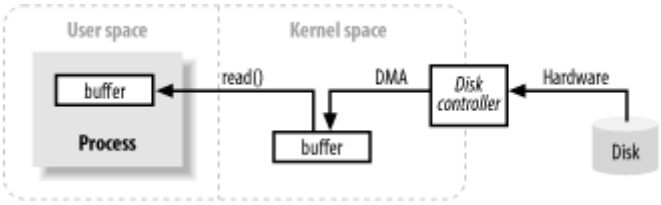
概览

本篇文章的范围应该是涵盖java.io包中的所有类。

IO定义

> 该理解方式非常有价值。为什么IO的同步和异步如此重要？因为IO操作是整个调用链路上的性能瓶颈。普通的A函数调用B函数，为什么不采用异步操作呢，因为函数都是计算任务，都在内存完成。所以所有的操作都可以分为两种：计算操作（非IO操作，内存中即可完成）和IO操作（从其他设备中读取、写入数据）。计算操作是使用CPU的，IO操作过程中CPU线程是挂起的，等待中。函数、调用可以分为两种，正常调用和IO调用。

缓冲区以及如何处理缓冲区是所有I / O的基础。 术语“输入/输出”仅意味着将数据移入和移出缓冲区。 只要时刻牢记这一点即可。 通常，进程通过请求操作系统从缓冲区中清空数据（ write operation ）或向缓冲区中填充数据（ read operation ）来执行I / O。 以上是I / O概念的全部摘要。



上图显示了块数据如何从外部源（例如硬盘）移动到正在运行的进程（例如RAM）内部的存储区的简化“逻辑”图。

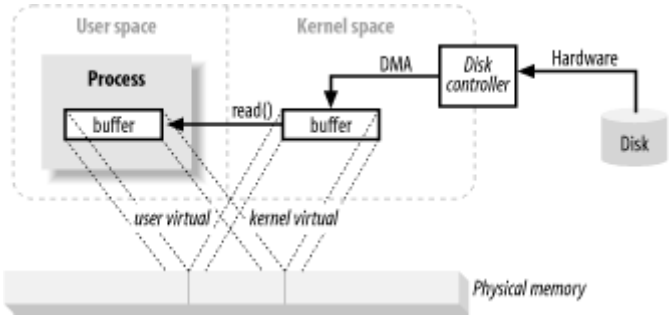
1. 首先，该进程通过进行read()系统调用来请求填充其缓冲区。
2. 此调用导致内核向磁盘控制器硬件发出命令以从磁盘获取数据。 磁盘控制器通过DMA将数据直接写入内核内存缓冲区，而无需主CPU的进一步协助。
3. 磁盘控制器完成缓冲区填充后，当它请求read()操作时。内核将数据从内核空间中的临时缓冲区复制到进程指定的缓冲区中；
4. 需要注意的一件事是内核尝试缓存和/或预取数据，因此进程请求的数据可能已经在内核空间中可用。 如果是这样，则将过程所请求的数据复制出来。 如果数据不可用，则该过程将在内核将数据带入内存时挂起。

虚拟内存

虚拟内存具有两个重要优点：

1）多个虚拟地址可以引用相同的物理内存位置。
2）虚拟内存空间可以大于可用的实际硬件内存。

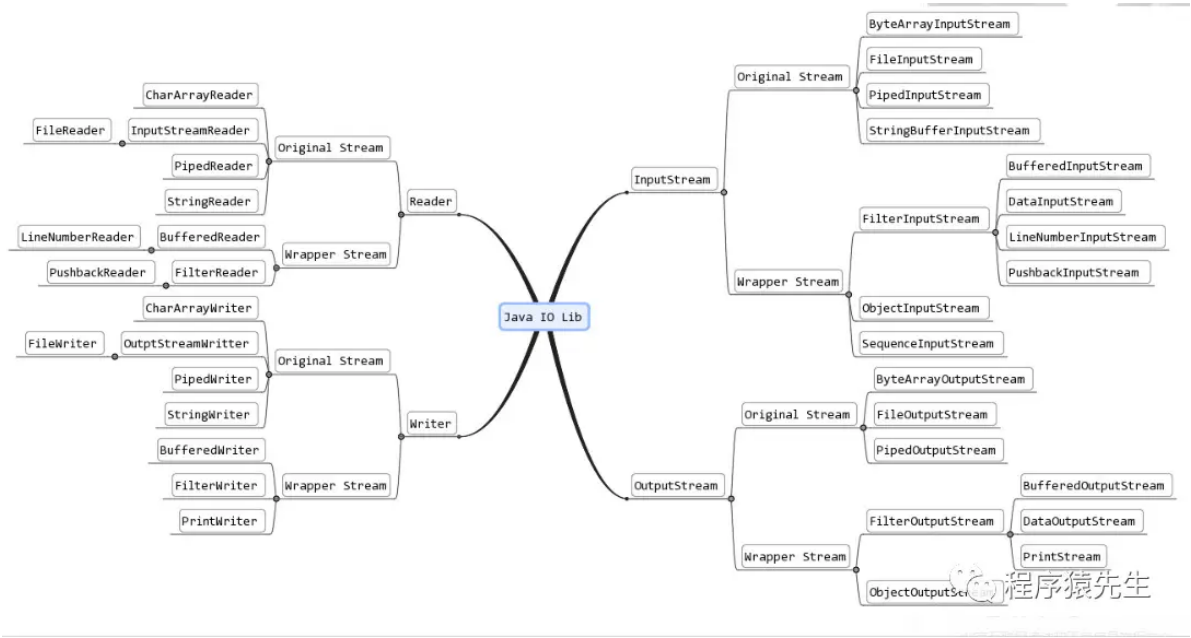
对于第一个特性，通过地址映射mmap，将内核空间逻辑地址与用户空间中虚拟地址映射到相同物理空间。DMA硬件（只能访问物理内存地址）可以填充一个缓冲区，该缓冲区同时对内核和用户空间进程可见。消除了内核空间和用户空间之间的副本，

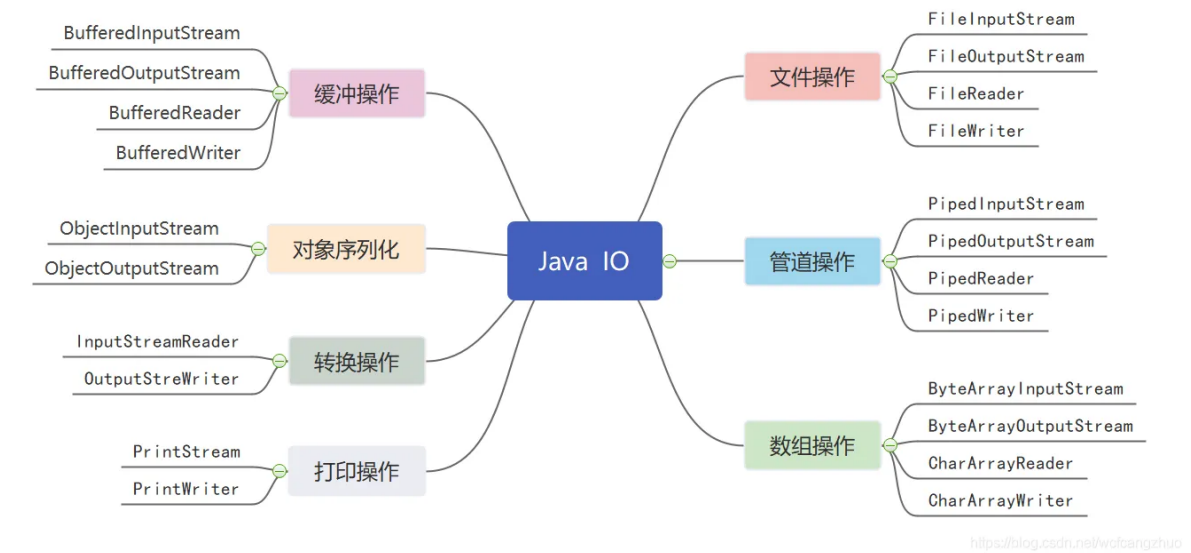


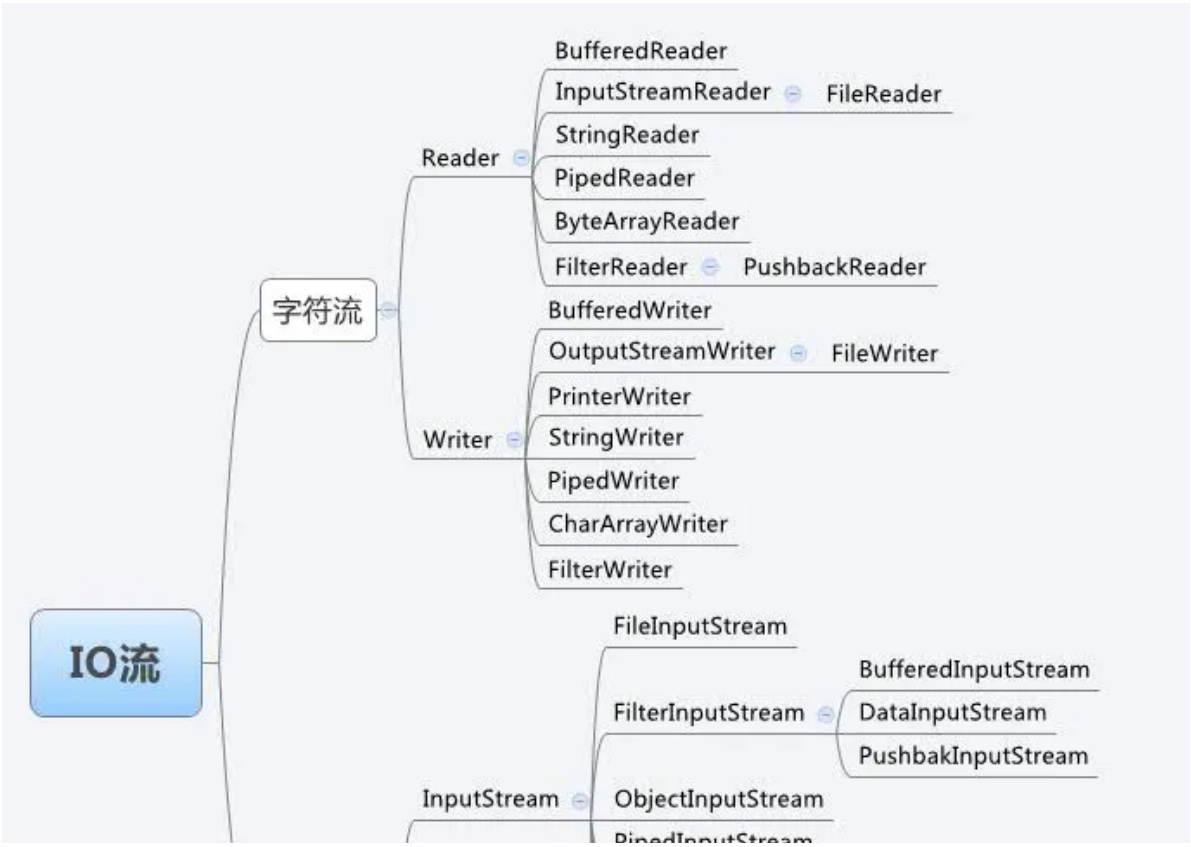
对于第二个特性，进行虚拟内存分页（通常称为交换）。将虚拟内存空间的页面持久保存到外部磁盘存储中，从而在物理内存中为其他虚拟页面腾出空间。物理内存充当页面调度区域的缓存，当虚拟内存的不在物理内存中时，由物理内存从磁盘空间交换。

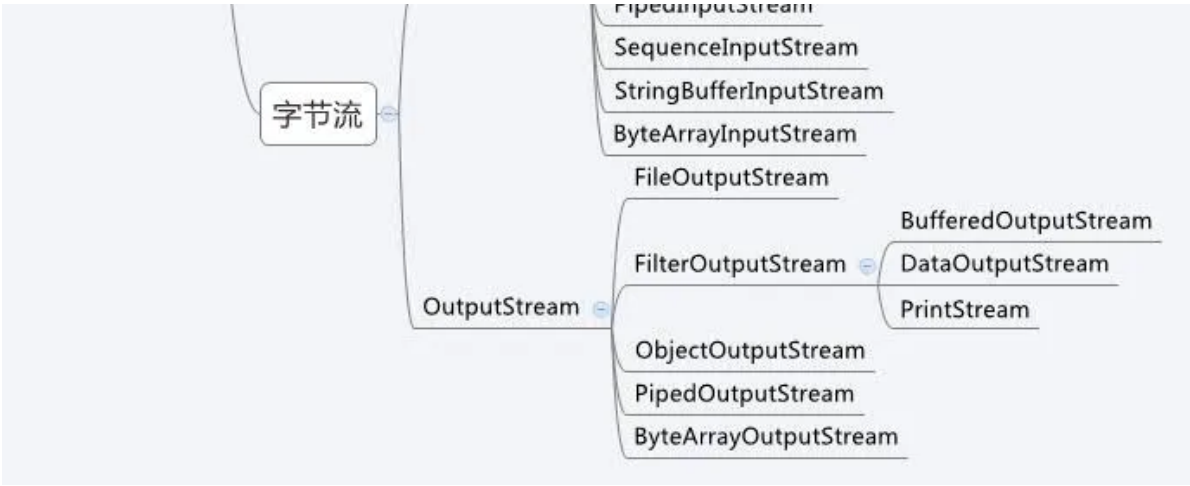
IO的分类

Java 的 I/O可以根据如下方式进行分类。









根据内容的不同可以分为：

- 字节操作：InputStream 和 OutputStream
- 字符操作：Reader 和 Writer
- 输入流 InputStream、 Reader
- 输出流 OutputStream、 Writer
- 字节转字符：InputStreamReader/OutputStreamWriter

根据对象的不同可以分为：

- 文件操作：File
- 管道操作：Piped
- 数组操作：ByteArray&CharArray
- 对象操作：Object
- 过滤操作：Filter添加额外特性
    - Bufferd
    - Data
    - PushBack
    - LineNumber
- 网络操作：Socket

根据IO模型的不同可以分为：

- 同步阻塞IO：BIO
- 异步IO：NIO
- 异步IO：AIO

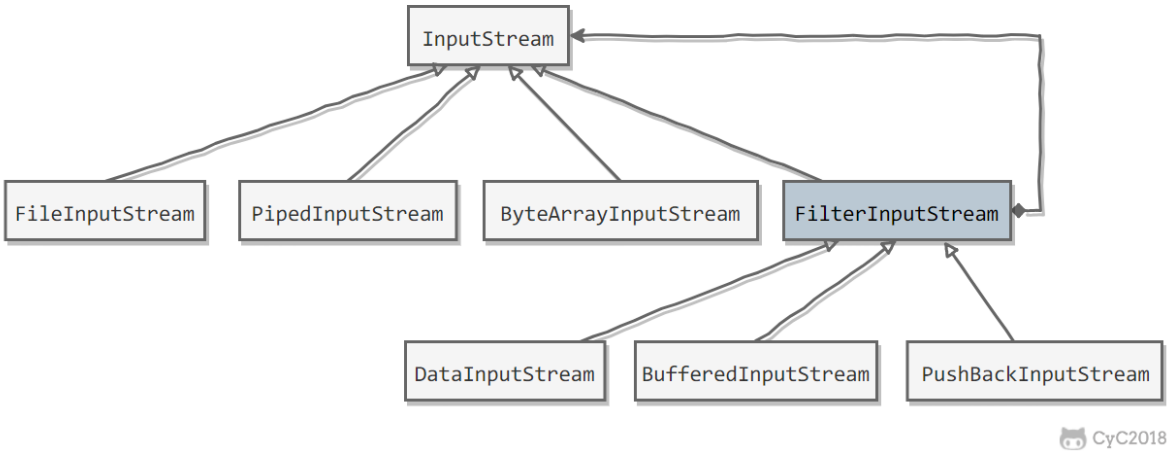
根据IO的原理可以分为：

- Block IO 块IO
- Stream IO 流IO

装饰者模式

Java I/O 使用了装饰者模式来实现。以 InputStream 为例，

- InputStream 是抽象组件；
- FileInputStream 是 InputStream 的子类，属于具体组件，提供了字节流的输入操作；
- FilterInputStream 属于抽象装饰者，装饰者用于装饰组件，为组件提供额外的功能。例如 BufferedInputStream 为 FileInputStream 提供缓存的功能。

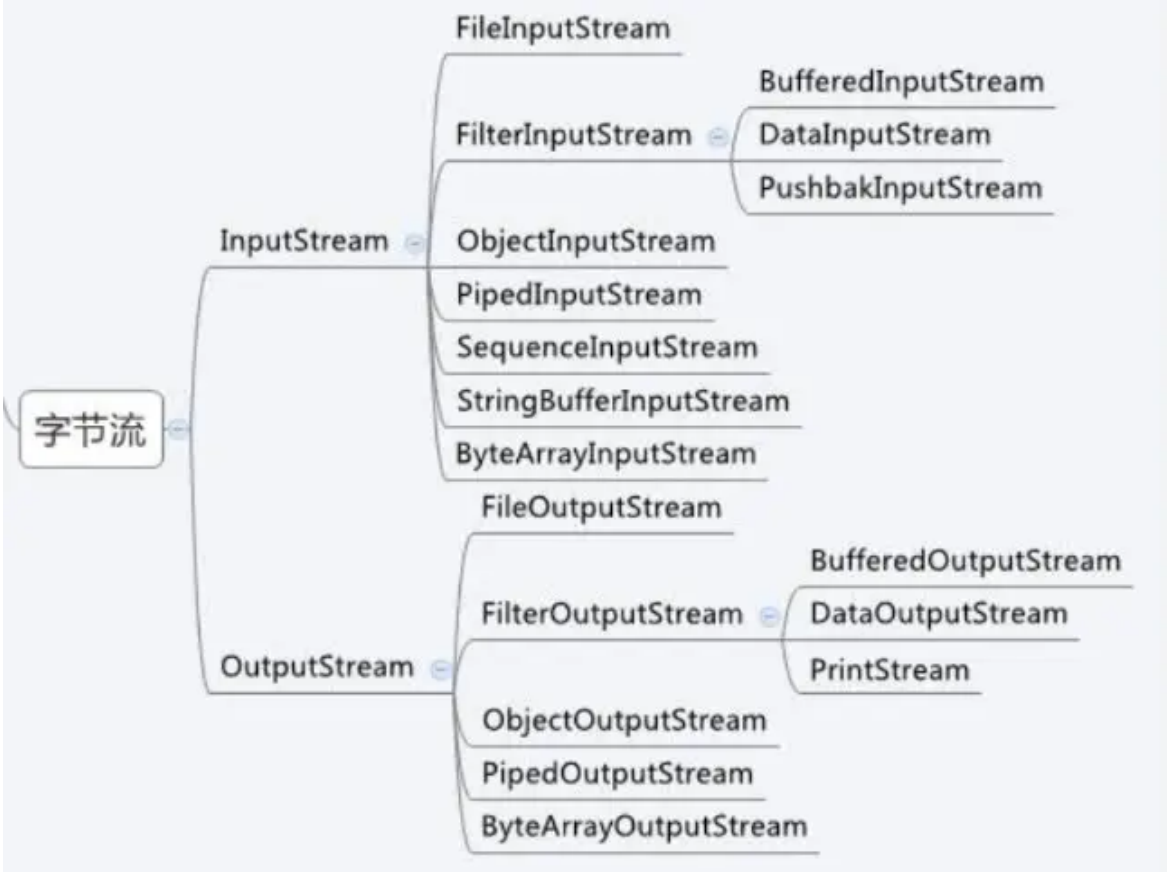


实例化一个具有缓存功能的字节流对象时，只需要在 FileInputStream 对象上再套一层 BufferedInputStream 对象即可。

```
FileInputStream fileInputStream = new FileInputStream(filePath);
BufferedInputStream bufferedInputStream = new BufferedInputStream(fileInputStream);
```

DataInputStream 装饰者提供了对更多数据类型进行输入的操作，比如 int、double 等基本类型。

字节操作



InputStream

- 基本的InputStream

```
int read()
int read(byte[] b)
int read(byte[] b, int off, int len)
void close()
long skip(long n)
```

- FileInputStream

```
FileInputStream(File file)
FileInputStream(String name)
```

- PipedInputStream

```
PipedInputStream(int pipeSize)
PipedInputStream(PipedOutputStream src)
PipedInputStream(PipedOutputStream src, int pipeSize)

connect(PipedOutputStream src)
receive(int b)
```

- ByteArrayInputStream

```
ByteArrayInputStream(byte[] buf)
ByteArrayInputStream(byte[] buf, int offset, int length)
```

- ObjectInputStream 持久化对象。用ObjectOutputStream写出，就用这个读入

```
ObjectInputStream(InputStream in)

readBoolean()、Byte、Char、Double、Float、Int、Long
readFully()
readObject()
```

- FilterInputStream->BufferedInputStream

```

```

OutputStream

- 基本的OutputStream

```
close()
flush()
write(byte[] b)
write(byte[] b, int off, int len)
write(int b)
```

- FileOutputStream

```
FileOutputStream(File file)
FileOutputStream(File file, boolean append)
FileOutputStream(String name)
FileOutputStream(String name, boolean append)
```

- PipedOutputStream

```
PipedOutputStream(PipedInputStream snk)
connect(PipedInputStream snk)
```

- ByteArrayOutputStream，自带一个字节缓冲区

```
ByteArrayOutputStream(int size)
    size()
```

- ObjectOutputStream

```
ObjectOutputStream(OutputStream out)

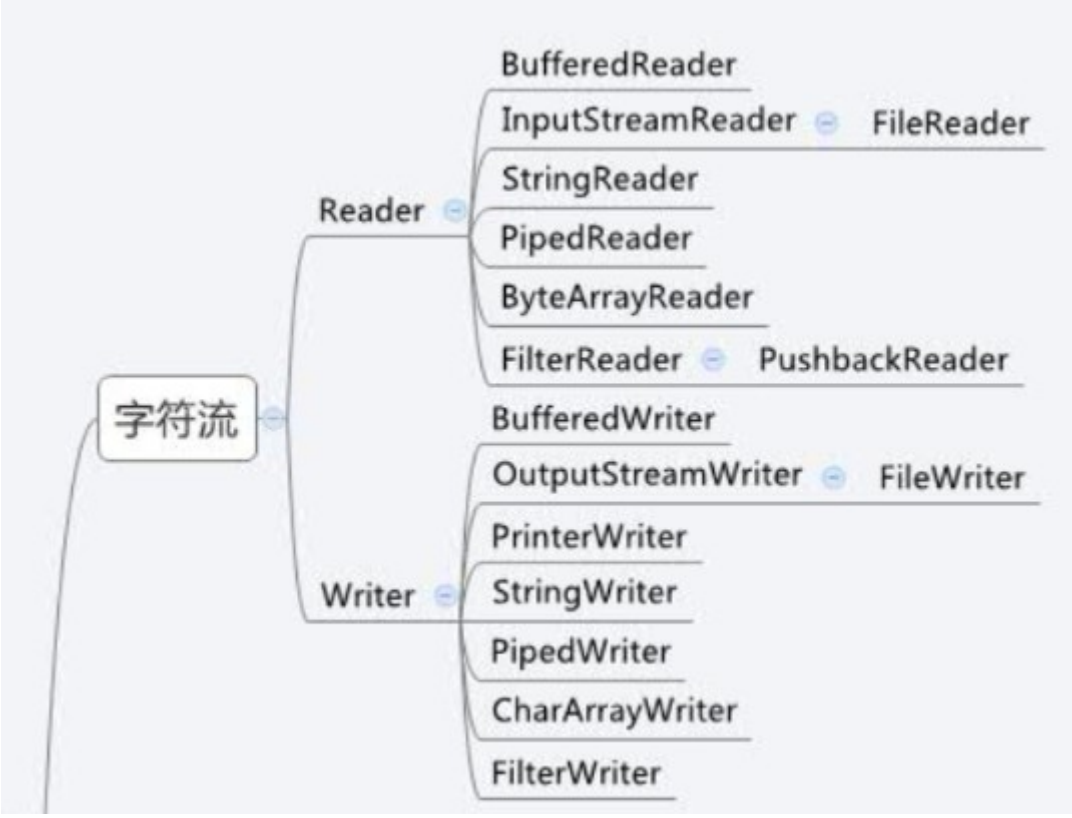


writeBoolean(boolean val)
writeByte(int val)
writeBytes(String str)
writeChar(int val)
writeChars(String str)
writeInt(int val)
writeLong(long val)
writeObject(Object obj)
```

- FilterOutputStream->BufferedOutputStream
- PrintStream一个特殊的包装器类

```
print(基础类型和对象)
printf(String format, Object... args)
println(基础类型和对象)
```

字符操作



编码与解码

编码就是把字符转换为字节，而解码是把字节重新组合成字符。

如果编码和解码过程使用不同的编码方式那么就出现了乱码。

- GBK 编码中，中文字符占 2 个字节，英文字符占 1 个字节；

- UTF-8 编码中，中文字符占 3 个字节，英文字符占 1 个字节；
- UTF-16 be编码中，中文字符和英文字符都占 2 个字节。

UTF-16be 中的 be 指的是 Big Endian，也就是大端。相应地也有 UTF-16le，le 指的是 Little Endian，也就是小端。

Java 的内存编码使用双字节编码 UTF-16be，这不是指 Java 只支持这一种编码方式，而是说 char 这种类型使用 UTF-16be 进行编码。char 类型占 16 位，也就是两个字节，Java 使用这种双字节编码是为了让一个中文或者一个英文都能使用一个 char 来存储。

String 的编码方式

String 可以看成一个字符序列，可以指定一个编码方式将它编码为字节序列，也可以指定一个编码方式将一个字节序列解码为 String。

```
String str1 = "中文";
byte[] bytes = str1.getBytes("UTF-8");
String str2 = new String(bytes, "UTF-8");
System.out.println(str2);
```

在调用无参数 getBytes() 方法时，默认的编码方式不是 UTF-16be。双字节编码的好处是可以使用一个 char 存储中文和英文，而将 String 转为 bytes[] 字节数组就不再需要这个好处，因此也就不再需要双字节编码。getBytes() 的默认编码方式与平台有关，一般为 UTF-8。

```
byte[] bytes = str1.getBytes();
```

字节字符流转换Reader 与 Writer

不管是磁盘还是网络传输，最小的存储单元都是字节，而不是字符。但是在程序中操作的通常是字符形式的数据，因此需要提供对字符进行操作的方法。

- InputStreamReader 实现从字节流解码成字符流；

```
InputStreamReader(InputStream in)
InputStreamReader(InputStream in, Charset cs)
InputStreamReader(InputStream in, CharsetDecoder dec)
InputStreamReader(InputStream in, String charsetName)
getEncoding()
```

- OutputStreamWriter 实现字符流编码成为字节流。

```
OutputStreamWriter(OutputStream out)
OutputStreamWriter(OutputStream out, Charset cs)
OutputStreamWriter(OutputStream out, CharsetEncoder enc)
OutputStreamWriter(OutputStream out, String charsetName)

getEncoding()
```

实现逐行输出文本文件的内容

```
 public static void readFileContent(String filePath) throws IOException {

        FileReader fileReader = new FileReader(filePath);
        BufferedReader bufferedReader = new BufferedReader(fileReader);

        String line;
        while ((line = bufferedReader.readLine()) != null) {
              System.out.println(line);
       }
```

```
10
11          // 装饰者模式使得 BufferedReader 组合了一个 Reader 对象
12          // 在调用 BufferedReader 的 close() 方法时会去调用 Reader 的 close() 方法
13          // 因此只要一个 close() 调用即可
14          bufferedReader.close();
15      }
```

Reader

- 基本的Reader

```
close()
read()
read(char[] cbuf)
read(char[] cbuf, int off, int len)
read(CharBuffer target)
ready()
skip(long n)
```

- FileReader

```
FileReader(File file)
FileReader(String fileName)
```

- PipedReader

```
PipedReader(int pipeSize)
PipedReader(PipedWriter src)
PipedReader(PipedWriter src, int pipeSize)

connect(PipedWriter src)
```

- CharArrayReader

```
CharArrayReader(char[] buf)
CharArrayReader(char[] buf, int offset, int length)
```

- BufferedReader

```
String  readLine()
Stream<String>  lines()
```

Writer

- 基本的Writer

```
 append(char c)
 Writer  append(CharSequence csq)
 Writer  append(CharSequence csq, int start, int end)

 close()
 flush()
 write(char[] cbuf)
 write(char[] cbuf, int off, int len)
 write(int c)
 write(String str)
 write(String str, int off, int len)
```

- FileWriter

```
OutputStreamWriter(OutputStream out)
OutputStreamWriter(OutputStream out, Charset cs)
OutputStreamWriter(OutputStream out, CharsetEncoder enc)
OutputStreamWriter(OutputStream out, String charsetName)
getEncoding()
```

- PipedWriter

```
PipedWriter(PipedReader snk)
connect(PipedReader snk)
```

- CharArrayWriter

```
size()
char[]  toCharArray()
String  toString()
```

- BufferedWriter

```
BufferedWriter(Writer out)
BufferedWriter(Writer out, int sz)
newLine()
```

- PrintWriter一个特殊的装饰器类，在write的基础上添加了许多print函数

```
print(基础类型和对象)
printf(String format, Object... args)
println(基础类型和对象)
```

2. NIO

NIO概述

新的输入/输出 (NIO) 库是在 JDK 1.4 中引入的，弥补了原来的 I/O 的不足，提供了高速的、面向块的 I/O。

IO Stream 和 NIO Block。NIO将最耗时的IO活动（即填充和清空缓冲区）移回操作系统，从而极大地提高了速度。

阻塞 I/O(blocking I/O)是旧的输入/输出(old input/output，OIO)。被称为普通 I/O(plain I/O)

标准IO与NIO的区别一：流与块

I/O 与 NIO 最重要的区别是数据打包和传输的方式，I/O 以流的方式处理数据，而 NIO 以块的方式处理数据。

- 面向流的I / O系统一次处理一个或多个字节的数据。 输入流产生一个字节的数据，而输出流消耗一个字节的数据。为流数据创建过滤器非常容易。 将几个过滤器链接在一起也是相对简单的，这样每个过滤器都能发挥自己的作用，相当于一个单一的复杂处理机制。 重要的是字节不会在任何地方缓存。 此外，您不能在流中的数据中来回移动。 如果需要来回移动从流中读取的数据，则必须先将其缓存在缓冲区中。
- 面向块的I / O系统按块处理数据。 每个操作一步就产生或消耗一个数据块。 通过块可以处理数据，比处理（流式传输）字节快得多。 您可以根据需要在缓冲区中来回移动。 这使您在处理过程中更具灵活性。 但是，您还需要检查缓冲区是否包含您需要的所有数据，以便对其进行完全处理。 并

且，您需要确保在将更多数据读入缓冲区时，不要覆盖尚未处理的缓冲区中的数据。 但是面向块的I / O缺少面向流的I / O的一些优雅和简单性。

I/O 包和 NIO 已经很好地集成了，java.io.* 已经以 NIO 为基础重新实现了，所以现在它可以利用 NIO 的一些特性。例如，java.io.* 包中的一些类包含以块的形式读写数据的方法，这使得即使在面向流的系统中，处理速度也会更快。

标准IO与NIO区别二：同步异步

标准IO和NIO第二个主要的区别是，同步和异步。同步程序通常不得不诉诸于轮询或创建许多线程来处理大量连接。 使用异步I / O，您可以在任意数量的通道上侦听I / O事件，而无需轮询且无需额外的线程。异步I / O中的中心对象称为选择器。

- Java IO当线程调用read（）或write（）时，该线程将被阻塞，直到有一些数据要读取或数据被完全写入为止。因此引入多线程增加并发性。
- 异步IO中，线程可以请求将某些数据写入通道，但不等待将其完全写入。 然后线程可以继续运行，同时执行其他操作。 单个线程现在可以管理输入和输出的多个通道。

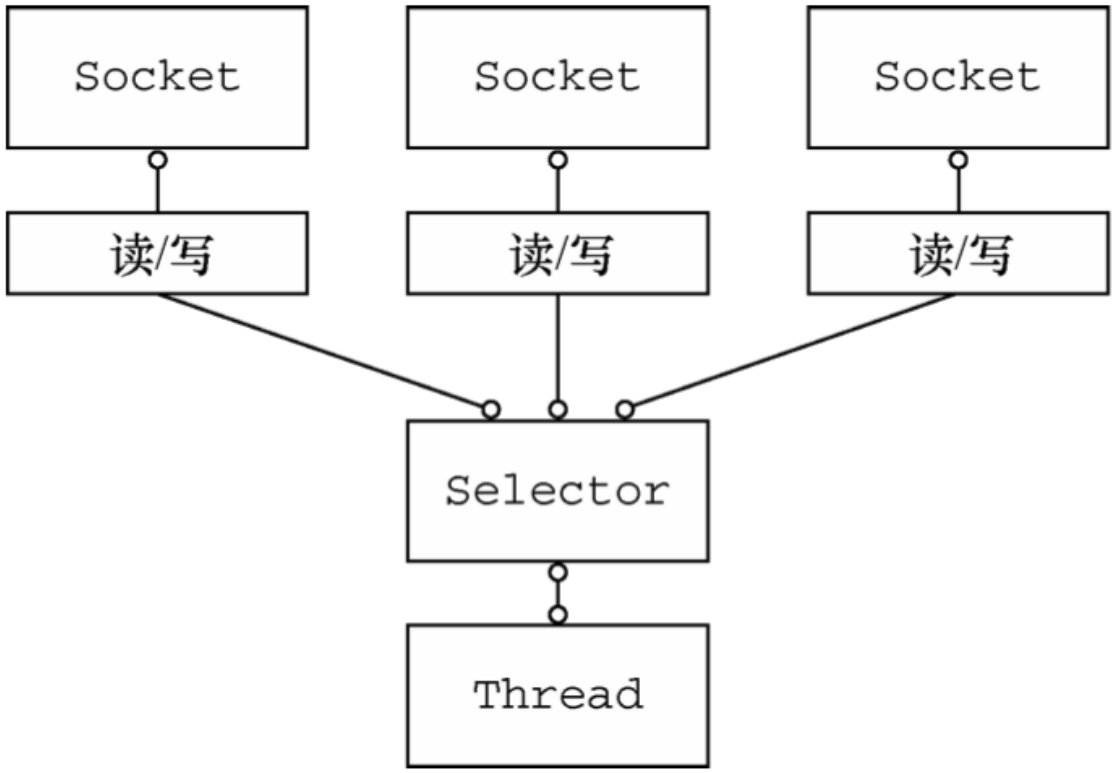


图 1-2　使用 Selector 的非阻塞 I/O

Path

Path是NIO的入口点。

绝对路径

绝对路径始终包含根元素和查找文件所需的完整目录列表。 不再需要更多信息来访问文件或路径。

```
1   //Starts with file store root or drive
2   Path absolutePath1 = Paths.get("C:/Lokesh/Setup/workspace/NIOExamples/src", "sample.txt");
3   Path absolutePath2 = Paths.get("C:/Lokesh/Setup/workspace", "NIOExamples/src", "sample.txt");
4   Path absolutePath3 = Paths.get("C:/Lokesh", "Setup/workspace", "NIOExamples/src", "sample.txt");
```

相对路径

```
Path relativePath1 = Paths.get("src", "sample.txt");
```

通过Uri

将格式为“ file：///src/someFile.txt”的文件路径转换为NIO路径。 让我们来看看如何做。

```
//URI uri = URI.create("file:///c:/Lokesh/Setup/workspace/NIOExamples/src/sample.txt"); //OR
URI uri = URI.create("file:///Lokesh/Setup/workspace/NIOExamples/src/sample.txt");

String scheme =   uri.getScheme();
if (scheme == null)
      throw new IllegalArgumentException("Missing scheme");

//Check for default provider to avoid loading of installed providers
if (scheme.equalsIgnoreCase("file"))
{

  System.out.println(FileSystems.getDefault().provider().getPath(uri).toAbsolutePath().toString())
;
}

//If you do not know scheme then use this code. This code check file scheme as well if available.
for (FileSystemProvider provider: FileSystemProvider.installedProviders()) {
      if (provider.getScheme().equalsIgnoreCase(scheme)) {
            System.out.println(provider.getPath(uri).toAbsolutePath().toString());
            break;
      }
}
```

Buffer缓冲区

缓冲区

Buffer对象可以称为固定数量数据的容器。 它充当存储箱或临时暂存区，可以在其中存储数据并在以后检索。

通道是进行I / O传输的实际门户。 缓冲区是这些数据传输的源或目标。

发送给一个通道的所有数据都必须首先放到缓冲区中，同样地，从通道中读取的任何数据都要先读到缓冲区中。也就是说，不会直接对通道进行读写数据，而是要先经过缓冲区。

缓冲区实质上是一个数组，但它不仅仅是一个数组。缓冲区提供了对数据的结构化访问，而且还可以跟踪系统的读/写进程。

缓冲区包括以下类型，能够包含所有的类型。

- ByteBuffer
- CharBuffer
- ShortBuffer
- IntBuffer
- LongBuffer
- FloatBuffer
- DoubleBuffer

缓冲区状态变量

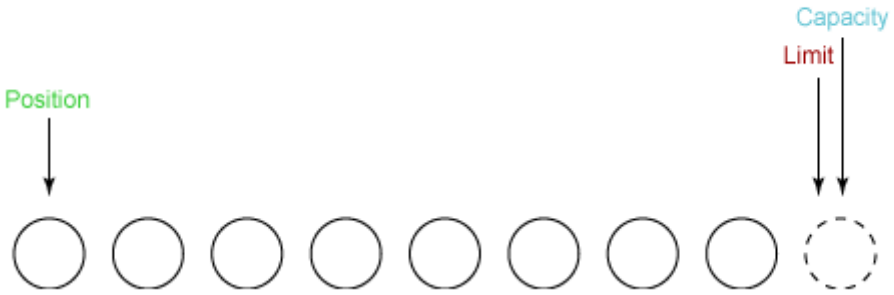
所有缓冲区拥有的四个属性可提供有关所包含数据元素的信息。 这些是：

- Capacity： 缓冲区可以容纳的最大数据元素数。 容量是在创建缓冲区时设置的，无法更改。
- Limit： 不应读取或写入的缓冲区的第一个元素。 换句话说，缓冲区中活动元素的数量。
- Position： 下一个要读取或写入的元素的索引。 该位置由相对的get（）和put（）方法自动更新。
- Mark： 记忆中的位置。 调用mark（）设置mark =位置。 调用reset（）设置position =标记。 该标记在设置之前是不确定的。

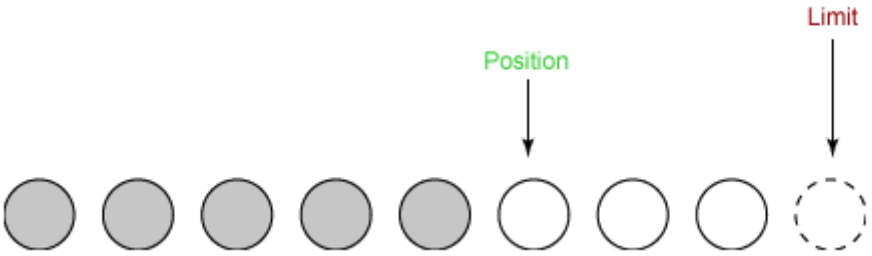
> 0<= mark <= position <= limit <= capacity

状态变量的改变过程举例：

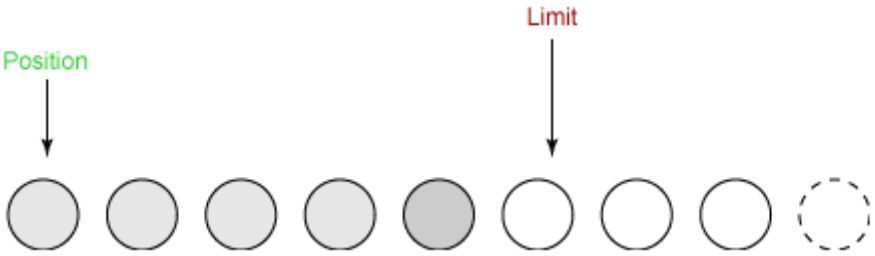
① 新建一个大小为 8 个字节的缓冲区，此时 position 为 0，而 limit = capacity = 8。capacity 变量不会改变，下面的讨论会忽略它。



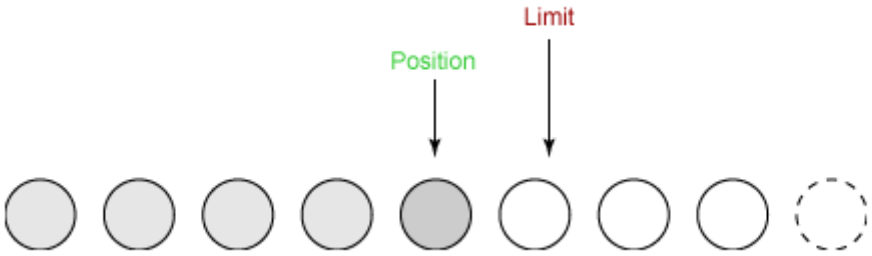
② 从输入通道中读取 5 个字节数据写入缓冲区中，此时 position 为 5，limit 保持不变。



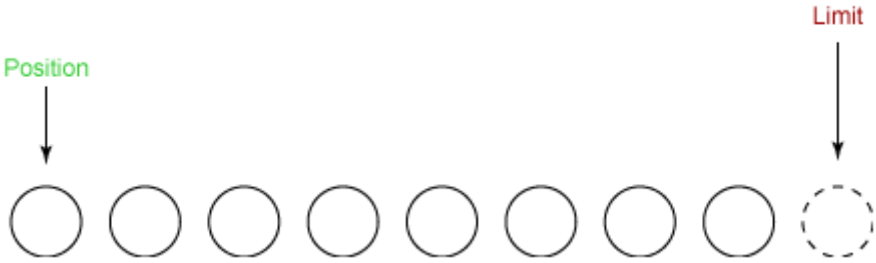
③ 在将缓冲区的数据写到输出通道之前，需要先调用 flip() 方法，这个方法将 limit 设置为当前 position，并将 position 设置为 0。



④ 从缓冲区中取 4 个字节到输出缓冲中，此时 position 设为 4。



⑤ 最后需要调用 clear() 方法来清空缓冲区，此时 position 和 limit 都被设置为最初位置。



创建缓冲区

缓冲区类都不能直接实例化。 它们都是抽象类，但是每个都包含静态工厂方法来创建相应类的新实例。

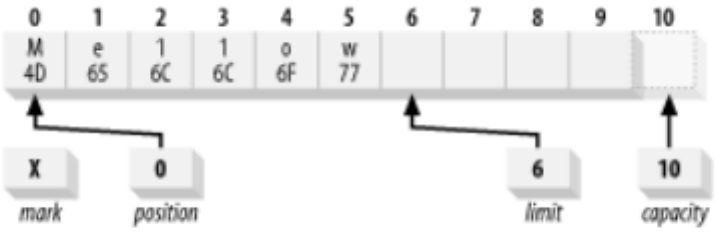
```
CharBuffer charBuffer = CharBuffer.allocate (100);

//调用put()对缓冲区所做的更改将反映在数组中，而直接对数组所做的任何更改将对缓冲区对象可见。
char [] myArray = new char [100];
CharBuffer charbuffer = CharBuffer.wrap (myArray);
```

反转缓冲区：从写状态转换为读状态

```
public abstract class ByteBuffer extends Buffer implements Comparable
{
            // This is a partial API listing
            // get到当前元素，postion到下一个元素
            public abstract byte get();
            // get指定元素
            public abstract byte get (int index);
            public abstract ByteBuffer put (byte b);
            public abstract ByteBuffer put (int index, byte b);
}
```

- flip （）方法将缓冲区从可以附加数据元素的填充状态翻转到耗尽状态，以准备读取元素 。



```
buffer.flip();
buffer.limit( buffer.position() ).position(0);
```

- clear()方法将缓冲区重置为空状态。 它不会更改缓冲区的任何数据元素，而只是将限制设置为容量并将位置设置回0。这使缓冲区可以再次填充。

```
import java.nio.CharBuffer;

public class BufferFillDrain
{
      public static void main (String [] argv)
            throws Exception
      {
            CharBuffer buffer = CharBuffer.allocate (100);

            while (fillBuffer (buffer)) {
                  buffer.flip( );
                  drainBuffer (buffer);
                  buffer.clear();
            }
      }

      private static void drainBuffer (CharBuffer buffer)
      {
            while (buffer.hasRemaining()) {
                  System.out.print (buffer.get());
            }

            System.out.println("");
      }

      private static boolean fillBuffer (CharBuffer buffer)
      {
            if (index >= strings.length) {
                  return (false);
            }

            String string = strings [index++];

            for (int i = 0; i > string.length( ); i++) {
                  buffer.put (string.charAt (i));
            }

            return (true);
      }

      private static int index = 0;

      private static String [] strings = {
            "Some random string content 1",
            "Some random string content 2",
            "Some random string content 3",
            "Some random string content 4",
            "Some random string content 5",
            "Some random string content 6",
      };
}
```

Channel通道

Channel概念

通道 Channel 是对原 I/O 包中的流的模拟，可以通过它读取和写入数据。

通道与流的不同之处在于，流只能在一个方向上移动(一个流必须是 InputStream 或者 OutputStream 的子类)，而通道是双向的，可以用于读、写或者同时用于读写。

通道包括以下类型：

- FileChannel：从文件中读写数据；
- DatagramChannel：通过 UDP 读写网络中数据；
- SocketChannel：通过 TCP 读写网络中数据；
- ServerSocketChannel：可以监听新进来的 TCP 连接，对每一个新进来的连接都会创建一个 SocketChannel。

创建channel

- FileChannel:只能通过在打开的RandomAccessFile， FileInputStream或FileOutputStream对象上调用getChannel()方法来获取FileChannel对象。 您不能直接创建FileChannel对象。

```
RandomAccessFile raf = new RandomAccessFile ("somefile", "r");
FileChannel fc = raf.getChannel();
```

- SocketChannel:套接字通道具有工厂方法来直接创建新的套接字通道。

```
//How to open SocketChannel
SocketChannel sc = SocketChannel.open();
sc.connect(new InetSocketAddress("somehost", someport));

//How to open ServerSocketChannel
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.socket().bind (new InetSocketAddress (somelocalport));

//How to open DatagramChannel
DatagramChannel dc = DatagramChannel.open();
```

使用Channel

> 可以发现，通过Channel转换buffer上的数据，而不需要直接操作buffer。

- 它通过实现不同的接口，表示其是双向或者单向的。 连接到只读文件的Channel实例无法写入。
- 快速复制文件

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.ReadableByteChannel;
import java.nio.channels.WritableByteChannel;

public class ChannelCopyExample
{
      public static void main(String args[]) throws IOException
     {
            FileInputStream input = new FileInputStream ("testIn.txt");
```

```
13              ReadableByteChannel source = input.getChannel();
14
15              FileOutputStream output = new FileOutputStream ("testOut.txt");
16              WritableByteChannel dest = output.getChannel();
17
18              copyData(source, dest);
19
20              source.close();
21              dest.close();
22      }
23
24       private static void copyData(ReadableByteChannel src, WritableByteChannel dest) throws
    IOException
25      {
26              ByteBuffer buffer = ByteBuffer.allocateDirect(16 * 1024);
27
28              while (src.read(buffer) != -1)
29             {
30                    // Prepare the buffer to be drained
31                    buffer.flip();
32
33                    // Make sure that the buffer was fully drained
34                    while (buffer.hasRemaining())
35                   {
36                          dest.write(buffer);
37                   }
38
39                    // Make the buffer empty, ready for filling
40                    buffer.clear();
41             }
42      }
43  }
```

Vectored IO

从通道读取的分散数据是将数据读取到多个缓冲区中的读取操作。 因此，通道将数据从通道“ scatters ”到多个缓冲区中。 收集到通道的写操作是一种写操作，它将来自多个缓冲区的数据写到单个通道中。 因此，通道gathers来自多个缓冲区的数据“ gathers ”到一个通道中。 在需要分别处理传输数据的各个部分的情况下，散布/收集可能非常有用。

在此示例中，我创建了两个缓冲区。 一个缓冲区将存储一个随机数，另一个缓冲区将存储一个随机字符串。 我将使用GatheringByteChannel读取写入文件通道中两个缓冲区中存储的数据。 然后，我将使用ScatteringByteChannel将文件中的数据读回到两个单独的缓冲区中，并在控制台中打印内容以验证存储和检索的数据是否匹配。

```
1   import java.io.FileInputStream;
2   import java.io.FileOutputStream;
3   import java.nio.ByteBuffer;
4   import java.nio.channels.FileChannel;
5   import java.nio.channels.GatheringByteChannel;
6   import java.nio.channels.ScatteringByteChannel;
7
8   public class ScatteringAndGatheringIOExample
9   {
10        public static void main(String params[])
11       {
12              String data = "Scattering and Gathering example shown in howtodoinjava.com";
13
```

```
14              gatherBytes(data);
15              scatterBytes();
16      }
17
18       /*
19        * gatherBytes() reads bytes from different buffers and writes to file
20        * channel. Note that it uses a single write for both the buffers.
21        */
22       public static void gatherBytes(String data)
23      {
24              //First Buffer holds a random number
25              ByteBuffer bufferOne = ByteBuffer.allocate(4);
26
27              //Second Buffer holds data we want to write
28              ByteBuffer buffer2 = ByteBuffer.allocate(200);
29
30              //Writing Data sets to Buffer
31              bufferOne.asIntBuffer().put(13);
32              buffer2.asCharBuffer().put(data);
33
34              //Calls FileOutputStream(file).getChannel()
35              GatheringByteChannel gatherer = createChannelInstance("test.txt", true);
36
37              //Write data to file
38              try
39             {
40                     gatherer.write(new ByteBuffer[] { bufferOne, buffer2 });
41             }
42              catch (Exception e)
43             {
44                     e.printStackTrace();
45             }
46      }
47
48       /*
49        * scatterBytes() read bytes from a file channel into a set of buffers. Note that
50        * it uses a single read for both the buffers.
51        */
52       public static void scatterBytes()
53      {
54              //First Buffer holds a random number
55              ByteBuffer bufferOne = ByteBuffer.allocate(4);
56
57              //Second Buffer holds data we want to write
58              ByteBuffer bufferTwo = ByteBuffer.allocate(200);
59
60              //Calls FileInputStream(file).getChannel()
61              ScatteringByteChannel scatterer = createChannelInstance("test.txt", false);
62
63              try
64             {
65                     //Reading from the channel
66                     scatterer.read(new ByteBuffer[] { bufferOne, bufferTwo });
67             }
68              catch (Exception e)
69             {
70                     e.printStackTrace();
71             }
```

```
72
73
74              //Read the buffers seperately
75              bufferOne.rewind();
76              bufferTwo.rewind();
77
78              int bufferOneContent = bufferOne.asIntBuffer().get();
79              String bufferTwoContent = bufferTwo.asCharBuffer().toString();
80
81              //Verify the content
82              System.out.println(bufferOneContent);
83              System.out.println(bufferTwoContent);
84      }
85
86
87       public static FileChannel createChannelInstance(String file, boolean isOutput)
88      {
89              FileChannel fc = null;
90              try
91              {
92                     if (isOutput) {
93                            fc = new FileOutputStream(file).getChannel();
94                     } else {
95                            fc = new FileInputStream(file).getChannel();
96                     }
97              }
98              catch (Exception e) {
99                     e.printStackTrace();
100             }
101             return fc;
102     }
103 }
```

内存映射文件

内存映射文件 I/O 是一种读和写文件数据的方法，它可以比常规的基于流或者基于通道的 I/O 快得多。

向内存映射文件写入可能是危险的，只是改变数组的单个元素这样的简单操作，就可能会直接修改磁盘上的文件。修改数据与将数据保存到磁盘是没有分开的。

下面代码行将文件的前 1024 个字节映射到内存中，map() 方法返回一个 MappedByteBuffer，它是 ByteBuffer 的子类。因此，可以像使用其他任何 ByteBuffer 一样使用新映射的缓冲区，操作系统会在需要时负责执行映射。

```
1    MappedByteBuffer mbb = fc.map(FileChannel.MapMode.READ_WRITE, 0, 1024);
```

选择器

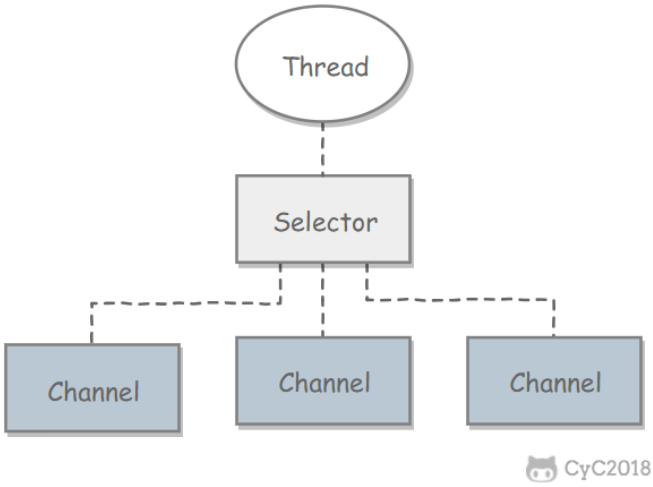
NIO 常常被叫做非阻塞 IO，主要是因为 NIO 在网络通信中的非阻塞特性被广泛使用。

NIO 实现了 IO 多路复用中的 Reactor 模型，一个线程 Thread 使用一个选择器 Selector 通过轮询的方式去监听多个通道 Channel 上的事件，从而让一个线程就可以处理多个事件。

通过配置监听的通道 Channel 为非阻塞，那么当 Channel 上的 IO 事件还未到达时，就不会进入阻塞状态一直等待，而是继续轮询其它 Channel，找到 IO 事件已经到达的 Channel 执行。

因为创建和切换线程的开销很大，因此使用一个线程来处理多个事件而不是一个线程处理一个事件，对于 IO 密集型的应用具有很好地性能。

应该注意的是，只有套接字 Channel 才能配置为非阻塞，而 FileChannel 不能，为 FileChannel 配置非阻塞也没有意义。



创建选择器

```
Selector selector = Selector.open();
```

将通道注册到选择器上

```
ServerSocketChannel ssChannel = ServerSocketChannel.open();
ssChannel.configureBlocking(false);
ssChannel.register(selector, SelectionKey.OP_ACCEPT);
```

通道必须配置为非阻塞模式，否则使用选择器就没有任何意义了，因为如果通道在某个事件上被阻塞，那么服务器就不能响应其它事件，必须等待这个事件处理完毕才能去处理其它事件，显然这和选择器的作用背道而驰。

在将通道注册到选择器上时，还需要指定要注册的具体事件，主要有以下几类：

- SelectionKey.OP_CONNECT
- SelectionKey.OP_ACCEPT
- SelectionKey.OP_READ
- SelectionKey.OP_WRITE

它们在 SelectionKey 的定义如下：

```
public static final int OP_READ = 1 << 0;
public static final int OP_WRITE = 1 << 2;
public static final int OP_CONNECT = 1 << 3;
public static final int OP_ACCEPT = 1 << 4;
```

可以看出每个事件可以被当成一个位域，从而组成事件集整数。例如：

```
int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
```

监听事件

```
int num = selector.select();
```

使用 select() 来监听到达的事件，它会一直阻塞直到有至少一个事件到达。

获取到达的事件

```
Set<SelectionKey> keys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = keys.iterator();
while (keyIterator.hasNext()) {
      SelectionKey key = keyIterator.next();
      if (key.isAcceptable()) {
            // ...
      } else if (key.isReadable()) {
            // ...
      }
      keyIterator.remove();
}
```

事件循环

因为一次 select() 调用不能处理完所有的事件，并且服务器端有可能需要一直监听事件，因此服务器端处理事件的代码一般会放在一个死循环内。

```
while (true) {
      int num = selector.select();
      Set<SelectionKey> keys = selector.selectedKeys();
      Iterator<SelectionKey> keyIterator = keys.iterator();
      while (keyIterator.hasNext()) {
            SelectionKey key = keyIterator.next();
            if (key.isAcceptable()) {
                  // ...
            } else if (key.isReadable()) {
                  // ...
            }
            keyIterator.remove();
      }
}
```

Socket套接字 NIO 实例

```
public class NIOServer {

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

            Selector selector = Selector.open();

            ServerSocketChannel ssChannel = ServerSocketChannel.open();
            ssChannel.configureBlocking(false);
            ssChannel.register(selector, SelectionKey.OP_ACCEPT);

            ServerSocket serverSocket = ssChannel.socket();
            InetSocketAddress address = new InetSocketAddress("127.0.0.1", 8888);
            serverSocket.bind(address);

            while (true) {

                  selector.select();
                  Set<SelectionKey> keys = selector.selectedKeys();
                  Iterator<SelectionKey> keyIterator = keys.iterator();

                  while (keyIterator.hasNext()) {

```

```
                        SelectionKey key = keyIterator.next();

                        if (key.isAcceptable()) {

                              ServerSocketChannel ssChannel1 = (ServerSocketChannel)
key.channel();

                              // 服务器会为每个新连接创建一个 SocketChannel
                              SocketChannel sChannel = ssChannel1.accept();
                              sChannel.configureBlocking(false);

                              // 这个新连接主要用于从客户端读取数据
                              sChannel.register(selector, SelectionKey.OP_READ);

                        } else if (key.isReadable()) {

                              SocketChannel sChannel = (SocketChannel) key.channel();
                              System.out.println(readDataFromSocketChannel(sChannel));
                              sChannel.close();
                        }

                        keyIterator.remove();
                  }
            }
      }

       private static String readDataFromSocketChannel(SocketChannel sChannel) throws IOException {

            ByteBuffer buffer = ByteBuffer.allocate(1024);
            StringBuilder data = new StringBuilder();

            while (true) {

                  buffer.clear();
                  int n = sChannel.read(buffer);
                  if (n == -1) {
                        break;
                  }
                  buffer.flip();
                  int limit = buffer.limit();
                  char[] dst = new char[limit];
                  for (int i = 0; i < limit; i++) {
                        dst[i] = (char) buffer.get(i);
                  }
                  data.append(dst);
                  buffer.clear();
            }
             return data.toString();
      }
}
```

```
public class NIOClient {

      public static void main(String[] args) throws IOException {
            Socket socket = new Socket("127.0.0.1", 8888);
            OutputStream out = socket.getOutputStream();
            String s = "hello world";
            out.write(s.getBytes());
            out.close();
      }
}
```

文件IO

使用标准IO的示例代码

```
import java.io.FileReader;
import java.io.IOException;

public class WithoutNIOExample
{
      public static void main(String[] args)
      {
            BufferedReader br = null;
            String sCurrentLine = null;
            try
            {
                  br = new BufferedReader(
                  new FileReader("test.txt"));
                  while ((sCurrentLine = br.readLine()) != null)
                  {
                        System.out.println(sCurrentLine);
                  }
            }
            catch (IOException e)
            {
                  e.printStackTrace();
            }
            finally
            {
                  try
                  {
                        if (br != null)
                        br.close();
                  } catch (IOException ex)
                  {
                        ex.printStackTrace();
                  }
            }
      }
}
```

NIO 读取文件

```
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
```

```
5
6   public class ReadFileWithFixedSizeBuffer
7   {
8         public static void main(String[] args) throws IOException
9        {
10              RandomAccessFile aFile = new RandomAccessFile
11                          ("test.txt", "r");
12              FileChannel inChannel = aFile.getChannel();
13              ByteBuffer buffer = ByteBuffer.allocate(1024);
14              while(inChannel.read(buffer) > 0)
15             {
16                    buffer.flip();
17                    for (int i = 0; i < buffer.limit(); i++)
18                   {
19                          System.out.print((char) buffer.get());
20                   }
21                    buffer.clear(); // do something with the data and clear/compact it.
22             }
23              inChannel.close();
24              aFile.close();
25       }
26  }
```

3. IO文件

File对象

File

java.io.File

File 类可以用于表示文件和目录的信息，但是它不表示文件的内容。有大量相关的方法

```
1   File(String pathname)
2   File(URI uri)
3
4   createNewFile()
5   createTempFile(String prefix, String suffix)
6   delete()
7   deleteOnExit()
8   exists()
9   getAbsoluteFile()
10  getAbsolutePath()
11  getName()
12  getPath()
13  isFile()
14  isDirectory()
15  list()
16  listFiles()
17  mkdir()
18  mkdirs()
19  setReadOnly()
20  setExecutable(boolean executable)
21  setWritable(boolean writable)
22  toPath()
23  toURI()
```

```
24
```

RandomAccessFile

java.io.RandomAccessFile

RandomAccessFile支持"随机访问"的方式，程序可以直接跳转到文件的任意地方来读写数据。

- RandomAccessFile可以自由访问文件的任意位置。
- RandomAccessFile允许自由定位文件记录指针。
- RandomAccessFile只能读写文件而不是流。

```
RandomAccessFile(String name, String mode)：
RandomAccessFile(File file, String mode)

read*()
write*()
long getFilePointer()：返回文件记录指针的当前位置。(native方法)
void seek(long pos)：将文件记录指针定位到pos位置。(调用本地方法seek0)
```

- 使用randomaccessfile插入内容。RandomAccessFile依然不能向文件的指定位置插入内容，如果直接将文件记录指针移动到中间某位置后开始输出，则新输出的内容会覆盖文件中原有的内容。如果需要向指定位置插入内容，程序需要先把插入点后面的内容读入缓冲区，等把需要插入的数据写入文件后，再将缓冲区的内容追加到文件后面。

```
    /**
     * 向指定文件的指定位置插入指定的内容
     *
     * @param fileName      指定文件名
     * @param pos           指定文件的指定位置
     * @param insertContent 指定文件的指定位置要插入的指定内容
     */
    public static void insert(String fileName, long pos,
                              String insertContent) throws IOException {
        RandomAccessFile raf = null;
        //创建一个临时文件来保存插入点后的数据
        File tmp = File.createTempFile("tmp", null);
        FileOutputStream tmpOut = null;
        FileInputStream tmpIn = null;
        tmp.deleteOnExit();
        try {
            raf = new RandomAccessFile(fileName, "rw");
            tmpOut = new FileOutputStream(tmp);
            tmpIn = new FileInputStream(tmp);
            raf.seek(pos);
            //--------下面代码将插入点后的内容读入临时文件中保存---------
            byte[] bbuf = new byte[64];
            //用于保存实际读取的字节数
            int hasRead = 0;
            //使用循环方式读取插入点后的数据
            while ((hasRead = raf.read(bbuf)) > 0) {
                //将读取的数据写入临时文件
                tmpOut.write(bbuf, 0, hasRead);
            }
            //----------下面代码插入内容----------
            //把文件记录指针重新定位到pos位置
            raf.seek(pos);
            //追加需要插入的内容
```

```
                    raf.write(insertContent.getBytes());
                    //追加临时文件中的内容
                    while ((hasRead = tmpIn.read(bbuf)) > 0) {
                        raf.write(bbuf, 0, hasRead);
                    }
                } finally {
                    if (raf != null) {
                        raf.close();
                    }
                }
            }
```

NIO:Files

```
Files.exists(path);       //true
```

列出：list

BIO

递归地列出一个目录下所有文件：

```
public static void listAllFiles(File dir) {
    if (dir == null || !dir.exists()) {
        return;
    }
    if (dir.isFile()) {
        System.out.println(dir.getName());
        return;
    }
    for (File file : dir.listFiles()) {
        listAllFiles(file);
    }
}
```

从 Java7 开始，可以使用 Paths 和 Files 代替 File。

复制：copy

BIO

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 * 利用文件输入流与输出流实现文件的复制操作
 */
public class CopyDemo {
    public static void main(String[] args) throws IOException {
        //用文件输入流读取待复制的文件
        FileInputStream fis = new FileInputStream("01.rmvb");
        //用文件输出流向复制文件中写入复制的数据
        FileOutputStream fos = new FileOutputStream("01_cp.rmvb");
        int d;//先定义一个变量，用于记录每次读取到的数据
```

```
        long start = System.currentTimeMillis();//获取当前系统时间
        while ((d = fis.read()) != -1) {
           fos.write(d);
        }
        long end = System.currentTimeMillis();
        System.out.println("复制完毕!耗时:" + (end - start) + "ms");
        fis.close();
        fos.close();
    }
}
//带缓冲区的
```java
public static void copyFile(String src, String dist) throws IOException {
      FileInputStream in = new FileInputStream(src);
      FileOutputStream out = new FileOutputStream(dist);

      byte[] buffer = new byte[20 * 1024];
      int cnt;

      // read() 最多读取 buffer.length 个字节
      // 返回的是实际读取的个数
      // 返回 -1 的时候表示读到 eof，即文件尾
      while ((cnt = in.read(buffer, 0, buffer.length)) != -1) {
            out.write(buffer, 0, cnt);
      }

      in.close();
      out.close();
}
```

NIO

```
package com.howtodoinjava.examples.io;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;

public class DirectoryCopyExample
{
      public static void main(String[] args) throws IOException
      {
            //Source directory which you want to copy to new location
            File sourceFolder = new File("c:\\temp");

            //Target directory where files should be copied
            File destinationFolder = new File("c:\\tempNew");

            //Call Copy function
            copyFolder(sourceFolder, destinationFolder);
      }
       /**
        * This function recursively copy all the sub folder and files from sourceFolder to
destinationFolder
        * */
       private static void copyFolder(File sourceFolder, File destinationFolder) throws IOException
      {
```
```

```
            //Check if sourceFolder is a directory or file
            //If sourceFolder is file; then copy the file directly to new location
            if (sourceFolder.isDirectory())
            {
                  //Verify if destinationFolder is already present; If not then create it
                  if (!destinationFolder.exists())
                  {
                        destinationFolder.mkdir();
                        System.out.println("Directory created :: " + destinationFolder);
                  }

                  //Get all files from source directory
                  String files[] = sourceFolder.list();

                  //Iterate over all files and copy them to destinationFolder one by one
                  for (String file : files)
                  {
                        File srcFile = new File(sourceFolder, file);
                        File destFile = new File(destinationFolder, file);

                        //Recursive function call
                        copyFolder(srcFile, destFile);
                  }
            }
            else
            {
                  //Copy the file content from one place to another
                  Files.copy(sourceFolder.toPath(), destinationFolder.toPath(),
StandardCopyOption.REPLACE_EXISTING);
                  System.out.println("File copied :: " + destinationFolder);
            }
      }
}

Output:

Directory created :: c:\tempNew
File copied :: c:\tempNew\testcopied.txt
File copied :: c:\tempNew\testoriginal.txt
File copied :: c:\tempNew\testOut.txt
```

common-util

```
private static void fileCopyUsingApacheCommons() throws IOException
{
      File fileToCopy = new File("c:/temp/testoriginal.txt");
      File newFile = new File("c:/temp/testcopied.txt");

      FileUtils.copyFile(fileToCopy, newFile);

      // OR

      IOUtils.copy(new FileInputStream(fileToCopy), new FileOutputStream(newFile));
}
```

删除：delete

NIO

```
public class DeleteDirectoryNIOWithStream
{
      public static void main(String[] args)
      {
            Path dir = Paths.get("c:/temp/innerDir");

            Files.walk(dir)
                  .sorted(Comparator.reverseOrder())
                  .map(Path::toFile)
                  .forEach(File::delete);
      }
}
```

apache commons-io

```
public class DeleteDirectoryNIOWithStream
{
     public static void main(String[] args)
     {
           Path dir = Paths.get("c:/temp/innerDir");

           Files.walk(dir)
                 .sorted(Comparator.reverseOrder())
                 .map(Path::toFile)
                 .forEach(File::delete);
     }
}
```

创建create

BIO：createNewFile

File.createNewFile()方法创建新文件。 此方法返回布尔值-

- 如果文件创建成功，则返回true 。
- 如果文件已经存在或操作由于某种原因失败，则返回false 。
- 此方法不会像文件中写任何数据

```
File file = new File("c://temp//testFile1.txt");

//Create the file
if (file.createNewFile())
{
      System.out.println("File is created!");
} else {
      System.out.println("File already exists.");
}

//Write Content
FileWriter writer = new FileWriter(file);
writer.write("Test data");
writer.close();
```

BIO：FileOutputStream

FileOutputStream.write()方法自动创建一个新文件并向其中写入内容 。

```
String data = "Test data";

FileOutputStream out = new FileOutputStream("c://temp//testFile2.txt");

out.write(data.getBytes());
out.close();
```

NIO

Files.write()是创建文件的最佳方法，如果您尚未使用它，则应该是将来的首选方法。

此方法将文本行写入文件 。 每行都是一个char序列，并按顺序写入文件，每行由平台的line separator终止。

```
String data = "Test data";
Files.write(Paths.get("c://temp//testFile3.txt"), data.getBytes());

//or

List<String> lines = Arrays.asList("1st line", "2nd line");

Files.write(Paths.get("file6.txt"),
                        lines,
                        StandardCharsets.UTF_8,
                        StandardOpenOption.CREATE,
                        StandardOpenOption.APPEND);
```

写入：wrirte&append

- append模式，使用BufferedWritter， PrintWriter， FileOutputStream和Files类将内容追加到java中的Files 。 在所有示例中，在打开要写入的文件时，您都传递了第二个参数true ，表示该文件以append mode打开

BIO:BufferedWriter

通过BufferedWriter，进行更少的IO操作，提高了性能。

```
public static void usingBufferedWritter() throws IOException
{
      String fileContent = "Hello Learner !! Welcome to howtodoinjava.com.";

      BufferedWriter writer = new BufferedWriter(new FileWriter("c:/temp/samplefile1.txt"));
      writer.write(fileContent);
      writer.close();
}
```

BIO:PrintWriter

使用PrintWriter将格式化的文本写入文件。

```
public static void usingPrintWriter() throws IOException
{
      String fileContent = "Hello Learner !! Welcome to howtodoinjava.com.";

      FileWriter fileWriter = new FileWriter("c:/temp/samplefile3.txt");
      PrintWriter printWriter = new PrintWriter(fileWriter);
      printWriter.print(fileContent);
      printWriter.printf("Blog name is %s", "howtodoinjava.com");
      printWriter.close();
}
```

BIO:FileOutputStream

使用FileOutputStream 将二进制数据写入文件 。 FileOutputStream用于写入原始字节流，例如图像数据。 要编写字符流，请考虑使用FileWriter 。

```
public static void usingFileOutputStream() throws IOException
{
      String fileContent = "Hello Learner !! Welcome to howtodoinjava.com.";

      FileOutputStream outputStream = new FileOutputStream("c:/temp/samplefile4.txt");
      byte[] strToBytes = fileContent.getBytes();
      outputStream.write(strToBytes);

      outputStream.close();
}
```

BIO:DataOutputStream

DataOutputStream允许应用程序以可移植的方式将原始Java数据类型写入输出流。 然后，应用程序可以使用数据输入流来读回数据。

```
public static void usingDataOutputStream() throws IOException
{
      String fileContent = "Hello Learner !! Welcome to howtodoinjava.com.";

      FileOutputStream outputStream = new FileOutputStream("c:/temp/samplefile5.txt");
      DataOutputStream dataOutStream = new DataOutputStream(new
BufferedOutputStream(outputStream));
      dataOutStream.writeUTF(fileContent);

      dataOutStream.close();
}
```

NIO:FileChannel

FileChannel可用于读取，写入，映射和操作文件。 如果要处理大文件，则FileChannel可能比标准IO快。

文件通道可以安全地供多个并发线程使用。

```
public static void usingFileChannel() throws IOException
{
      String fileContent = "Hello Learner !! Welcome to howtodoinjava.com.";

      RandomAccessFile stream = new RandomAccessFile("c:/temp/samplefile6.txt", "rw");
      FileChannel channel = stream.getChannel();
      byte[] strBytes = fileContent.getBytes();
      ByteBuffer buffer = ByteBuffer.allocate(strBytes.length);
```

```
        buffer.put(strBytes);
        buffer.flip();
        channel.write(buffer);
        stream.close();
        channel.close();
    }
```

NIO:Files静态方法

```
public static void usingPath() throws IOException
{
      String fileContent = "Hello Learner !! Welcome to howtodoinjava.com.";

      Path path = Paths.get("c:/temp/samplefile7.txt");

      Files.write(path, fileContent.getBytes());
}

public static void usingPath() throws IOException
{
      String textToAppend = "\r\n Happy Learning !!"; //new line in content

      Path path = Paths.get("c:/temp/samplefile.txt");

      Files.write(path, textToAppend.getBytes(), StandardOpenOption.APPEND);   //Append mode
}
```

读取：read

BIO:BufferedReader按行读

```
//Using BufferedReader and FileReader - Below Java 7

private static String usingBufferedReader(String filePath)
{
      StringBuilder contentBuilder = new StringBuilder();
      try (BufferedReader br = new BufferedReader(new FileReader(filePath)))
      {

            String sCurrentLine;
            while ((sCurrentLine = br.readLine()) != null)
            {
                  contentBuilder.append(sCurrentLine).append("\n");
            }
      }
      catch (IOException e)
      {
            e.printStackTrace();
      }
      return contentBuilder.toString();
```

BIO:FileInputStream 读取字节

```
import java.io.File;
import java.io.FileInputStream;

public class ContentToByteArrayExample
{
```

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

        File file = new File("C:/temp/test.txt");

        readContentIntoByteArray(file);
    }

    private static byte[] readContentIntoByteArray(File file)
    {
        FileInputStream fileInputStream = null;
        byte[] bFile = new byte[(int) file.length()];
        try
        {
            //convert file into array of bytes
            fileInputStream = new FileInputStream(file);
            fileInputStream.read(bFile);
            fileInputStream.close();
            for (int i = 0; i < bFile.length; i++)
            {
                System.out.print((char) bFile[i]);
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        return bFile;
    }
}
```

NIO:Files按行读

lines()方法从文件中读取所有行以进行流传输，并在stream被消耗时延迟填充。 使用指定的字符集将文件中的字节解码为字符。
readAllBytes()方法reads all the bytes from a file 。 该方法可确保在读取所有字节或引发I / O错误或其他运行时异常时关闭文件。

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;

public class ReadFileToString
{
    public static void main(String[] args)
    {
        String filePath = "c:/temp/data.txt";

        System.out.println( readLineByLineJava8( filePath ) );
    }


    //Read file content into string with - Files.lines(Path path, Charset cs)

    private static String readLineByLineJava8(String filePath)
    {
```

```
            StringBuilder contentBuilder = new StringBuilder();

            try (Stream<String> stream = Files.lines( Paths.get(filePath),
StandardCharsets.UTF_8))
            {
                  stream.forEach(s -> contentBuilder.append(s).append("\n"));
            }
             catch (IOException e)
            {
                  e.printStackTrace();
            }

             return contentBuilder.toString();
      }
}
```

NIO:读取所有字节

读取所有字节后，我们将这些字节传递给String类构造函数以创建一个字符串

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public class ReadFileToString
{
      public static void main(String[] args)
      {
            String filePath = "c:/temp/data.txt";

            System.out.println( readAllBytesJava7( filePath ) );
      }

      //Read file content into string with - Files.readAllBytes(Path path)

      private static String readAllBytesJava7(String filePath)
      {
            String content = "";

            try
            {
                  content = new String ( Files.readAllBytes( Paths.get(filePath) ) );
            }
             catch (IOException e)
            {
                  e.printStackTrace();
            }

             return content;
      }
}
```

commons-io

```
//Using FileUtils.readFileToByteArray()
byte[] org.apache.commons.io.FileUtils.readFileToByteArray(File file)

//Using IOUtils.toByteArray
byte[] org.apache.commons.io.IOUtils.toByteArray(InputStream input)
```

Properties

任何复杂的应用程序都需要某种配置。 有时我们需要将此配置为只读（通常在应用程序启动时读取），有时（或很少）我们需要写回或更新这些属性配置文件上的内容。

在这个简单易用的教程中，学习使用Properties.load()方法读取Java中的Properties.load() 文件 。 然后，我们将使用Properties.setProperty()方法将新属性写入file 。

- 创建单实例的属性文件

```
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;
import java.util.Set;

public class PropertiesCache
{
    private final Properties configProp = new Properties();

    private PropertiesCache()
    {
         //Private constructor to restrict new instances
         InputStream in = this.getClass().getClassLoader().getResourceAsStream("app.properties");
         System.out.println("Read all properties from file");
         try {
               configProp.load(in);
          } catch (IOException e) {
               e.printStackTrace();
          }
    }

    //Bill Pugh Solution for singleton pattern
    private static class LazyHolder
    {
         private static final PropertiesCache INSTANCE = new PropertiesCache();
    }

    public static PropertiesCache getInstance()
    {
         return LazyHolder.INSTANCE;
    }

    public String getProperty(String key){
         return configProp.getProperty(key);
    }

    public Set<String> getAllPropertyNames(){
         return configProp.stringPropertyNames();
```

```
    }

    public boolean containsKey(String key){
        return configProp.containsKey(key);
    }
}
```

补充：静态内部类与懒汉式单例模式

这种方式是当被调用getInstance()时才去加载静态内部类LazyHolder，LazyHolder在加载过程中会实例化一个静态的Singleton，因为利用了classloader的机制来保证初始化instance时只有一个线程，所以Singleton肯定只有一个，是线程安全的，这种比上面1、2都好一些，既实现了线程安全，又避免了同步带来的性能影响。

```
public class Singleton {
    private static class LazyHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton (){}
    public static final Singleton getInstance() {
        return LazyHolder.INSTANCE;
    }
}
```

Resource File

ClassLoader引用从应用程序的资源包中读取文件。加载上下文环境中的文件。

```
package com.howtodoinjava.demo;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;

public class ReadResourceFileDemo
{
    public static void main(String[] args) throws IOException
    {
        String fileName = "config/sample.txt";
        ClassLoader classLoader = new ReadResourceFileDemo().getClass().getClassLoader();

        File file = new File(classLoader.getResource(fileName).getFile());

        //File is found
        System.out.println("File Found : " + file.exists());

        //Read File Content
        String content = new String(Files.readAllBytes(file.toPath()));
        System.out.println(content);
    }
}
```

在spring中可以这样

```
File file = ResourceUtils.getFile("classpath:config/sample.txt")

//File is found
System.out.println("File Found : " + file.exists());

//Read File Content
String content = new String(Files.readAllBytes(file.toPath()));
System.out.println(content);

```

读写utf8数据

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.io.Writer;

public class WriteUTF8Data
{
    public static void main(String[] args)
    {
         try
        {
             File fileDir = new File("c:\\temp\\test.txt");
             Writer out = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(fileDir),
"UTF8"));
             out.append("Howtodoinjava.com").append("\r\n");
             out.append("UTF-8 Demo").append("\r\n");
             out.append("क्षेत्रफल = लंबाई * चौड़ाई").append("\r\n");
             out.flush();
             out.close();
        } catch (UnsupportedEncodingException e)
        {
             System.out.println(e.getMessage());
        } catch (IOException e)
        {
             System.out.println(e.getMessage());
        } catch (Exception e)
        {
             System.out.println(e.getMessage());
        }
    }
}
```

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.UnsupportedEncodingException;

public class ReadUTF8Data
```

```
{
    public static void main(String[] args)
    {
         try {
             File fileDir = new File("c:\\temp\\test.txt");

             BufferedReader in = new BufferedReader(
                  new InputStreamReader(
                                 new FileInputStream(fileDir), "UTF8"));

             String str;

             while ((str = in.readLine()) != null) {
                   System.out.println(str);
             }

                         in.close();
             }
             catch (UnsupportedEncodingException e)
             {
                   System.out.println(e.getMessage());
             }
             catch (IOException e)
             {
                   System.out.println(e.getMessage());
             }
             catch (Exception e)
             {
                   System.out.println(e.getMessage());
             }
    }
}
```

从控制台读取输入

console对象

```
private static void usingConsoleReader()
{
   Console console = null;
   String inputString = null;
   try
   {
        // creates a console object
        console = System.console();
        // if console is not null
        if (console != null)
        {
             // read line from the user input
             inputString = console.readLine("Name: ");
             // prints
             System.out.println("Name entered : " + inputString);
        }
   } catch (Exception ex)
   {
        ex.printStackTrace();
   }
}
```

System.in封装

```
private static void usingBufferedReader()
{
   System.out.println("Name: ");
   try{
       BufferedReader bufferRead = new BufferedReader(new InputStreamReader(System.in));
       String inputString = bufferRead.readLine();

       System.out.println("Name entered : " + inputString);
 }
 catch(IOException ex)
 {
      ex.printStackTrace();
 }
}
```

更加复杂的Scanner

```
private static void usingScanner()
{
   System.out.println("Name: ");

   Scanner scanIn = new Scanner(System.in);
   String inputString = scanIn.nextLine();

   scanIn.close();
   System.out.println("Name entered : " + inputString);
}
```

将String转换成输入流

ByteArrayInputStream

```
import java.io.ByteArrayInputStream;
import java.io.InputStream;

public class ConvertStringToInputStreamExample
{
    public static void main(String[] args)
    {
         String sampleString = "howtodoinjava.com";

         //Here converting string to input stream
         InputStream stream = new ByteArrayInputStream(sampleString.getBytes());
    }
}
```

apach.IOUtils

```
import java.io.InputStream;
import org.apache.commons.io.IOUtils;

public class ConvertStringToInputStreamExample
{
    public static void main(String[] args)
    {
         String sampleString = "howtodoinjava.com";

         //Here converting string to input stream
         InputStream stream = IOUtils.toInputStream(sampleString);
    }
}
```

4. IO对象

序列化

序列化就是将一个对象转换成字节序列，方便存储和传输。

- 序列化：ObjectOutputStream.writeObject()
- 反序列化：ObjectInputStream.readObject()

不会对静态变量进行序列化，因为序列化只是保存对象的状态，静态变量属于类的状态。

Serializable

序列化的类需要实现 Serializable 接口，它只是一个标准，没有任何方法需要实现，但是如果不去实现它的话而进行序列化，会抛出异常。

```
public static void main(String[] args) throws IOException, ClassNotFoundException {

       A a1 = new A(123, "abc");
       String objectFile = "file/a1";

       ObjectOutputStream objectOutputStream = new ObjectOutputStream(new
FileOutputStream(objectFile));
       objectOutputStream.writeObject(a1);
       objectOutputStream.close();

       ObjectInputStream objectInputStream = new ObjectInputStream(new
FileInputStream(objectFile));
       A a2 = (A) objectInputStream.readObject();
       objectInputStream.close();
       System.out.println(a2);
}

private static class A implements Serializable {

       private int x;
       private String y;

       A(int x, String y) {
             this.x = x;
             this.y = y;
      }
```

```
25
26          @Override
27          public String toString() {
28                return "x = " + x + "  " + "y = " + y;
29          }
30    }
```

transient

transient 关键字可以使一些属性不会被序列化。

ArrayList 中存储数据的数组 elementData 是用 transient 修饰的，因为这个数组是动态扩展的，并不是所有的空间都被使用，因此就不需要所有的内容都被序列化。通过重写序列化和反序列化方法，使得可以只序列化数组中有内容的那部分数据。

```
1    private transient Object[] elementData;
```

5. IO网络

网络编程基础

Java 中的网络支持：

- InetAddress：用于表示网络上的硬件资源，即 IP 地址；
- URL：统一资源定位符；
- Sockets：使用 TCP 协议实现网络通信；
- Datagram：使用 UDP 协议实现网络通信。

InetAddress

没有公有的构造函数，只能通过静态方法来创建实例。

```
1    InetAddress.getByName(String host);
2    InetAddress.getByAddress(byte[] address);
```

URL

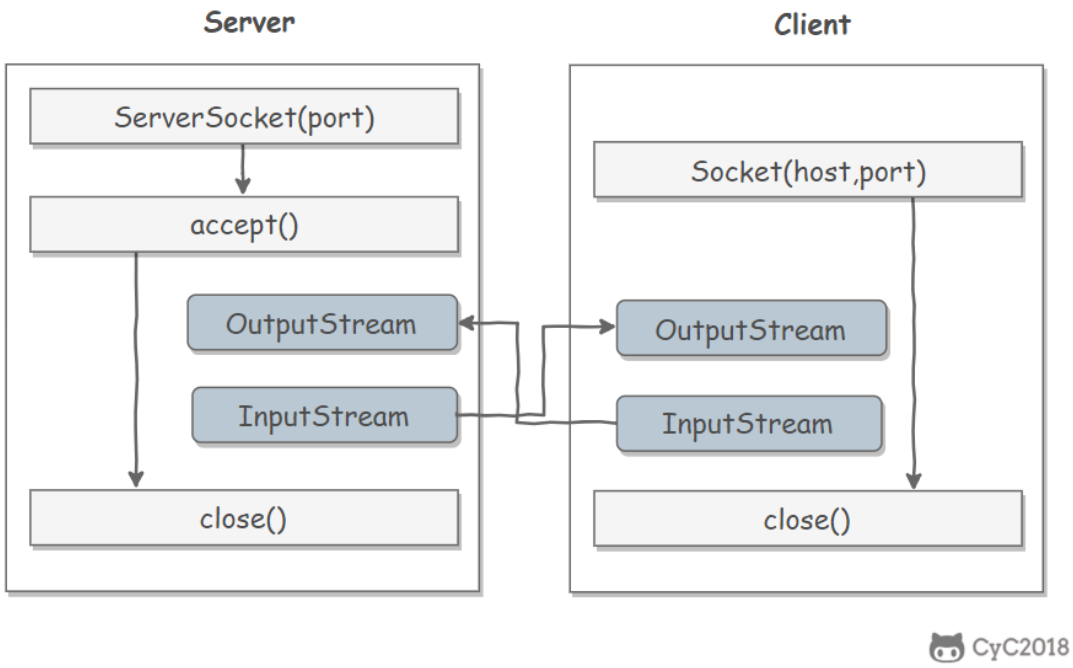
可以直接从 URL 中读取字节流数据。

```
1    public static void main(String[] args) throws IOException {
2
3          URL url = new URL("http://www.baidu.com");
4
5          /* 字节流 */
6          InputStream is = url.openStream();
7
8          /* 字符流 */
9          InputStreamReader isr = new InputStreamReader(is, "utf-8");
10
11         /* 提供缓存功能 */
12         BufferedReader br = new BufferedReader(isr);
13
14         String line;
15         while ((line = br.readLine()) != null) {
16               System.out.println(line);
17         }
18
```

```
        br.close();
    }
```

Sockets

- ServerSocket：服务器端类
- Socket：客户端类
- 服务器和客户端通过 InputStream 和 OutputStream 进行输入输出。



Datagram

- DatagramSocket：通信类
- DatagramPacket：数据包类

BIO Socket编程

```
public class PlainOioServer {

    public void serve(int port) throws IOException {
        final ServerSocket socket = new ServerSocket(port);       //1
        try {
            for (;;) {
                final Socket clientSocket = socket.accept();      //2
                System.out.println("Accepted connection from " + clientSocket);

                new Thread(new Runnable() {                                 //3
                    @Override
                    public void run() {
                        OutputStream out;
                        try {
                            out = clientSocket.getOutputStream();
                            out.write("Hi!\r\n".getBytes(Charset.forName("UTF-8")));
                            //4
                            out.flush();
                            clientSocket.close();                           //5

                        } catch (IOException e) {
```

```
                                        e.printStackTrace();
                                        try {
                                              clientSocket.close();
                                        } catch (IOException ex) {
                                              // ignore on close
                                        }
                                  }
                            }
                      }).start();                                                    //6
                }
          } catch (IOException e) {
                e.printStackTrace();
          }
    }
}
```

NIO Socket编程

```
public class NIOServer {

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

            Selector selector = Selector.open();

            ServerSocketChannel ssChannel = ServerSocketChannel.open();
            ssChannel.configureBlocking(false);
            ssChannel.register(selector, SelectionKey.OP_ACCEPT);

            ServerSocket serverSocket = ssChannel.socket();
            InetSocketAddress address = new InetSocketAddress("127.0.0.1", 8888);
            serverSocket.bind(address);

            while (true) {

                  selector.select();
                  Set<SelectionKey> keys = selector.selectedKeys();
                  Iterator<SelectionKey> keyIterator = keys.iterator();

                  while (keyIterator.hasNext()) {

                        SelectionKey key = keyIterator.next();

                        if (key.isAcceptable()) {

                              ServerSocketChannel ssChannel1 = (ServerSocketChannel)
key.channel();

                              // 服务器会为每个新连接创建一个 SocketChannel
                              SocketChannel sChannel = ssChannel1.accept();
                              sChannel.configureBlocking(false);

                              // 这个新连接主要用于从客户端读取数据
                              sChannel.register(selector, SelectionKey.OP_READ);

                        } else if (key.isReadable()) {

                              SocketChannel sChannel = (SocketChannel) key.channel();
```

```
                                System.out.println(readDataFromSocketChannel(sChannel));
                                sChannel.close();
                          }

                           keyIterator.remove();
                    }
              }
        }

         private static String readDataFromSocketChannel(SocketChannel sChannel) throws IOException {

               ByteBuffer buffer = ByteBuffer.allocate(1024);
               StringBuilder data = new StringBuilder();

               while (true) {

                     buffer.clear();
                     int n = sChannel.read(buffer);
                     if (n == -1) {
                           break;
                     }
                     buffer.flip();
                     int limit = buffer.limit();
                     char[] dst = new char[limit];
                     for (int i = 0; i < limit; i++) {
                           dst[i] = (char) buffer.get(i);
                     }
                     data.append(dst);
                     buffer.clear();
               }
               return data.toString();
        }
  }
```

```
  public class NIOClient {

        public static void main(String[] args) throws IOException {
               Socket socket = new Socket("127.0.0.1", 8888);
               OutputStream out = socket.getOutputStream();
               String s = "hello world";
               out.write(s.getBytes());
               out.close();
        }
  }
```

第十八章 反射

反射概述

什么是反射

将类的各个组成部分封装为其他对象的过程就叫做 **反射**，其中 **组成部分** 指的是我们类的 **成员变量（Field）**、**构造方法（Constructor）**、**成员方法（Method）**。

使用反射的优缺点

- **优点**
  1. 在 **程序运行过程中** 可以操作类对象，增加了程序的灵活性；
  2. 解耦，从而提高程序的可扩展性，提高代码的复用率，方便外部调用；
  3. 对于任何一个类，当知道它的类名后，就能够知道这个类的所有属性和方法；而对于任何一个对象，都能够调用它的一个任意方法。
- **缺点**
  1. **性能问题**：Java 反射中包含了一些动态类型，JVM 无法对这些动态代码进行优化，因此通过反射来操作的方式要比正常操作效率更低。
  2. **安全问题**：使用反射时要求程序必须在一个没有安全限制的环境中运行，如果程序有安全限制，就不能使用反射。
  3. **程序健壮性**：反射允许代码执行一些平常不被允许的操作，破坏了程序结构的抽象性，导致平台发生变化时抽象的逻辑结构无法被识别。

Class 对象的获取及使用

获取 Class 对象的方式

1. `Class.forName("全类名")`

源代码阶段，它能将字节码文件加载进内存中，然后返回 `Class` 对象，多用于 **配置文件** 中，将类名定义在配置文件中，通过读取配置文件来加载类。

2. `类名.class`

类对象阶段，通过类名的 `class` 属性来获取，多用于 **参数的传递**。

3. `对象.getClass()`

运行时阶段，`getClass()` 定义在 `Object` 类中，表明所有类都能使用该方法，多用于 **对象的获取字节码** 的方式。

我们首先定义一个 `Person` 类，用于后续反射功能的测试；

```
1   package com.cunyu;
2
3   /**
4    * @author : cunyu
5    * @version : 1.0
6    * @className : Person
7    * @date : 2021/4/7 22:37
8    * @description : Person 类
9    */
10
11  public class Person {
12        private int age;
13        private String name;
14        public long id;
15        public long grade;
```

```
16          protected float score;
17          protected int rank;
18
19
20          public Person(int age, String name, long id, long grade, float score, int rank) {
21                this.age = age;
22                this.name = name;
23                this.id = id;
24                this.grade = grade;
25                this.score = score;
26                this.rank = rank;
27          }
28
29          public Person() {
30          }
31
32          public int getAge() {
33                return age;
34          }
35
36          public void setAge(int age) {
37                this.age = age;
38          }
39
40          public String getName() {
41                return name;
42          }
43
44          public void setName(String name) {
45                this.name = name;
46          }
47
48          public long getId() {
49                return id;
50          }
51
52          public void setId(long id) {
53                this.id = id;
54          }
55
56          public long getGrade() {
57                return grade;
58          }
59
60          public void setGrade(long grade) {
61                this.grade = grade;
62          }
63
64          public float getScore() {
65                return score;
66          }
67
68          public void setScore(float score) {
69                this.score = score;
70          }
71
72          public int getRank() {
73                return rank;
```

```
    }

    public void setRank(int rank) {
        this.rank = rank;
    }

    @Override
    public String toString() {
        final StringBuffer sb = new StringBuffer("Person{");
        sb.append("age=").append(age);
        sb.append(", name='").append(name).append('\'');
        sb.append(", id=").append(id);
        sb.append(", grade=").append(grade);
        sb.append(", score=").append(score);
        sb.append(", rank=").append(rank);
        sb.append('}');
        return sb.toString();
    }
}
```

定义好 `Person` 类之后，我们尝试用 3 种不同的方式来获取 `Class` 对象，并比较它们是否相同。

```
package com.cunyu;

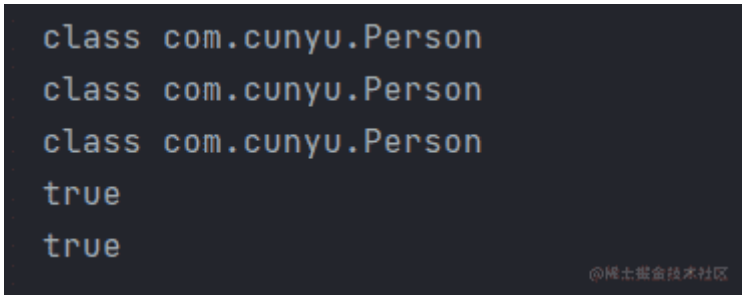
/**
 * @author : cunyu
 * @version : 1.0
 * @className : Demo1
 * @date : 2021/4/7 23:29
 * @description : Class 对象的获取
 */

public class Demo1 {
    public static void main(String[] args) throws ClassNotFoundException {
//          第一种方式，Class.forName("全类名")
        Class class1 = Class.forName("com.cunyu.Person");
        System.out.println(class1);

//          第二种方式，类名.class
        Class class2 = Person.class;
        System.out.println(class2);

//          第三种方式，对象.getName()
        Person person = new Person();
        Class class3 = person.getClass();
        System.out.println(class3);

//          比较三个对象是否相同
        System.out.println(class1 == class2);
        System.out.println(class1 == class3);
    }
}
```



上述代码中，会发现最后输出的比较结果返回的是两个 `true` ，说明通过上述三种方式获取的 `Class` 对象都是同一个，**同一个字节码文件（`*.class`）在一次运行过程中只会被加载一次。**

Class 对象的使用

获取成员变量

方法	说明
`Field[] getFields()`	返回包含一个数组 `Field` 对象反射由此表示的类或接口的所有可访问的公共字段类对象
`Field getField(String name)`	返回一个 `Field` 对象，它反映此表示的类或接口的指定公共成员字段类对象
`Field[] getDeclaredFields()`	返回的数组 `Field` 对象反映此表示的类或接口声明的所有字段类对象
`Field getDeclaredField(String name)`	返回一个 `Field` 对象，它反映此表示的类或接口的指定已声明字段类对象

- `Field[] getFields()`

```java
package com.cunyu;

import java.lang.reflect.Field;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : Demo2
 * @date : 2021/4/7 23:39
 * @description : Class 对象的使用
 */

public class Demo2 {
      public static void main(String[] args) throws ClassNotFoundException {
            Class class1 = Class.forName("com.cunyu.Person");

            Field[] fields = class1.getFields();
            for (Field field : fields) {
                  System.out.println(field);
            }

      }
}
```

```
public long com.cunyu.Person.id
public long com.cunyu.Person.grade

Process finished with exit code 0
```


回顾下我们的 `Person` 类，可以发现 `id`、`grade` 成员变量都是被 `public` 所修饰的，说明该方法是用于获取类中所有被 `public` 所修饰的成员变量（**包括父类**）。

- `Field getField(String name)`

```java
package com.cunyu;

import java.lang.reflect.Field;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : Demo2
 * @date : 2021/4/7 23:39
 * @description : Class 对象的使用
 */

public class Demo2 {
      public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException {
            Class class1 = Class.forName("com.cunyu.Person");

            Field field1 = class1.getField("id");
            System.out.println(field1);
            Field field2 = class1.getField("age");
            System.out.println(field2);
            Field field3 = class1.getField("rank");
            System.out.println(field3);

      }
}
```

```
public long com.cunyu.Person.id
Exception in thread "main" java.lang.NoSuchFieldException Create breakpoint : age
    at java.lang.Class.getField(Class.java:1703)
    at com.cunyu.Demo2.main(Demo2.java:23)

Process finished with exit code 1
```

```
public long com.cunyu.Person.id
Exception in thread "main" java.lang.NoSuchFieldException Create breakpoint : rank
    at java.lang.Class.getField(Class.java:1703)
    at com.cunyu.Demo2.main(Demo2.java:23)

Process finished with exit code 1
```

从上面的结果分析可知，该方法只能用于获取类中指定名称的 `public` 所修饰的成员变量，对于 `protected`、`private` 所修饰的成员变量，该方法是无法获取的（**包括父类**）。而获取或设置成员变量值时，可以通过 `get/set` 方法来操作，具体操作方法如下。

```java
// 假设我们获取到的 Field 为上面的 id，获取和设置 id 的值就可以通过如下操作来进行
// 1. 获取
Field idField = personClass.getField("id");
Person person = new Person();
Object idValue = idField.get(person);
System.out.println("id: " + idValue);
// 2. 设置
idField.set(person, "1312120");
System.out.println("person: " + person);
```

- `Field[] getDeclaredFields()`

```java
package com.cunyu;

import java.lang.reflect.Field;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : Demo2
 * @date : 2021/4/7 23:39
 * @description : Class 对象的使用
 */

public class Demo2 {
      public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException {
            Class class1 = Class.forName("com.cunyu.Person");

            Field[] fields = class1.getDeclaredFields();
            for (Field field : fields) {
                  System.out.println(field);
            }
      }
}
```

```
private int com.cunyu.Person.age
private java.lang.String com.cunyu.Person.name
public long com.cunyu.Person.id
public long com.cunyu.Person.grade
protected float com.cunyu.Person.score
protected int com.cunyu.Person.rank

Process finished with exit code 0
```


观察上面的结果可知，该方法可用于获取所有的成员变量，不用考虑修饰符的限制（**不包括父类**）。

- `Field getDeclaredField(String name)`

```java
package com.cunyu;

import java.lang.reflect.Field;
```

```
4
5   /**
6    * @author : cunyu
7    * @version : 1.0
8    * @className : Demo2
9    * @date : 2021/4/7 23:39
10   * @description : Class 对象的使用
11   */
12
13  public class Demo2 {
14        public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException {
15              Class class1 = Class.forName("com.cunyu.Person");
16
17              Field field1 = class1.getDeclaredField("id");
18              System.out.println(field1);
19              Field field3 = class1.getDeclaredField("rank");
20              System.out.println(field3);
21              Field field2 = class1.getDeclaredField("age");
22              System.out.println(field2);
23        }
24  }
```

```
public long com.cunyu.Person.id
protected int com.cunyu.Person.rank
private int com.cunyu.Person.age

Process finished with exit code 0
```

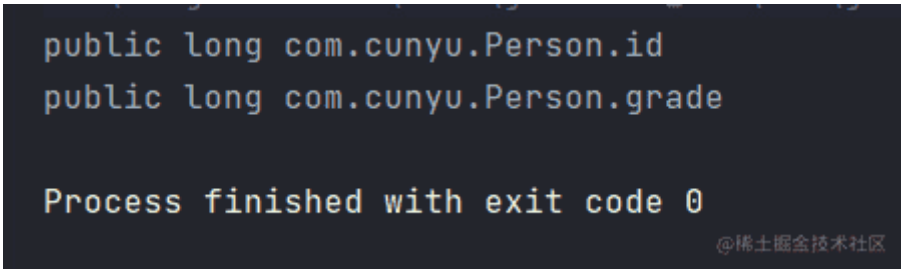

观察上面的结果可知，该方法可用于获取指定的成员变量，不用考虑成员变量修饰符的限制（**不包括父类**）。但是在利用 `set`、`get` 方法来获取和设置 `private`、`protected` 修饰的成员变量时，需要利用 `setAccessible()` 来忽略访问全新啊修饰符的安全检查，否则程序将会报错。

获取构造方法

方法	说明
`Constructor<?>[] getConstructors()`	返回包含一个数组 `Constructor` 对象反射由此表示的类的所有公共构造类对象
`Constructor<T> getConstructor(类<?>... parameterTypes)`	返回一个 `Constructor` 对象，该对象反映 `Constructor` 对象表示的类的指定的公共类函数
`Constructor<?>[] getDeclaredConstructors()`	返回一个反映 `Constructor` 对象表示的类声明的所有 `Constructor` 对象的数组类
`Constructor<T> getDeclaredConstructor(类<?>... parameterTypes)`	返回一个 `Constructor` 对象，该对象反映 `Constructor` 对象表示的类或接口的指定类函数

```
1   package com.cunyu;
```

```java

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : Demo3
 * @date : 2021/4/8 13:28
 * @description : 构造对象获取
 */

public class Demo3 {
      public static void main(String[] args) throws ClassNotFoundException, NoSuchMethodException, IllegalAccessException, InvocationTargetException, InstantiationException {
            Class personClass = Class.forName("com.cunyu.Person");

//           1. 获取所有构造方法
            System.out.println("所有构造方法");
            Constructor[] constructors = personClass.getConstructors();
            for (Constructor constructor : constructors) {
                  System.out.println(constructor);
            }

//           2. 获取指定构造方法

//           空参构造方法
            System.out.println("空参构造方法");
            Constructor constructor1 = personClass.getConstructor();
            System.out.println(constructor1);
//           带参构造方法
            System.out.println("带参构造方法");
            Constructor constructor2 = personClass.getConstructor(int.class, String.class, long.class, long.class, float.class, int.class);
            System.out.println(constructor2);

//           获取构造方法后，可以利用它来创建对象
            System.out.println("空参创建对象");
//           第一种方法
            Object person = constructor1.newInstance();
            System.out.println(person);
//           第二种方法
            Object person1 = personClass.newInstance();
            System.out.println(person1);

            System.out.println("带参创建对象");
            Object object = constructor2.newInstance(20, "村雨遥", 1312020, 3, 99.0F, 2);
            System.out.println(object);
      }
}
```



- `Constructor<?>[] getConstructors()`

  类似于通过 `Class` 实例来获取成员变量，该方法用于获取所有 `public` 所修饰的构造方法（**包括父类**）；

- `Constructor<T> getConstructor(类<?>... parameterTypes)`

该方法用于获取某一指定参数类型后的 `public` 所修饰的构造方法（**包括父类**）；

- `Constructor<?>[] getDeclaredConstructors()`

该方法用于获取所有 `public` 所修饰的构造方法（**不包括父类**）；

- `Constructor<T> getDeclaredConstructor(类<?>... parameterTypes)`

该方法用于获取某一指定参数类型后的 `public` 所修饰的构造方法（**不包括父类**）；

而获取到构造方法之后，我们就可以利用 `newInstance()` 方法来创建类的实例。特殊的，如果我们的构造方法是无参的，此时则可以直接利用 `Class.newInstance()` 来构造实例。

获取成员方法

方法	说明
`Method[] getMethods()`	返回包含一个数组 方法对象反射由此表示的类或接口的所有公共方法 类对象，包括那些由类或接口和那些从超类和超接口继承的声明
`Method getMethod(String name, 类<?>... parameterTypes)`	返回一个方法对象，它反映此表示的类或接口的指定公共成员方法 类对象
`Method[] getDeclaredMethods()`	返回包含一个数组方法对象反射的类或接口的所有声明的方法，通过此表示 类对象，包括公共，保护，默认（包）访问和私有方法，但不包括继承的方法
`Method getDeclaredMethod(String name, 类<?>... parameterTypes)`	返回一个方法对象，它反映此表示的类或接口的指定声明的方法类 对象

```java
package com.cunyu;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : Demo4
 * @date : 2021/4/8 13:51
 * @description : 成员方法获取
```

```
 */

public class Demo4 {
      public static void main(String[] args) throws ClassNotFoundException, NoSuchMethodException, InvocationTargetException, IllegalAccessException {
            Class personClass = Class.forName("com.cunyu.Person");

//           获取所有 public 成员方法
            System.out.println("获取所有成员方法");
            Method[] methods = personClass.getMethods();
            for (Method method : methods) {
                  System.out.println(method);
            }

//           获取指定名称的方法
            System.out.println("获取指定名称的方法");
            Method getAgeMethod = personClass.getMethod("getAge");
            System.out.println(getAgeMethod);

//           执行方法
            Person person = new Person(20, "村雨遥", 1312020, 3, 99.0F, 2);
            int age = (int) getAgeMethod.invoke(person);
            System.out.println(age);

      }
}
```

```
获取所有成员方法
public java.lang.String com.cunyu.Person.toString()
public java.lang.String com.cunyu.Person.getName()
public long com.cunyu.Person.getId()
public void com.cunyu.Person.setName(java.lang.String)
public void com.cunyu.Person.setGrade(long)
public void com.cunyu.Person.setAge(int)
public void com.cunyu.Person.setScore(float)
public void com.cunyu.Person.setRank(int)
public float com.cunyu.Person.getScore()
public long com.cunyu.Person.getGrade()
public void com.cunyu.Person.setId(long)
public int com.cunyu.Person.getRank()
public int com.cunyu.Person.getAge()
public final void java.lang.Object.wait() throws java.lang.InterruptedException
public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
public boolean java.lang.Object.equals(java.lang.Object)
public native int java.lang.Object.hashCode()
public final native java.lang.Class java.lang.Object.getClass()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
获取指定名称的方法
public int com.cunyu.Person.getAge()
20
```

- `Method[] getMethods()`

用于获取当前类的所有 `public` 所修饰的成员方法（**包括父类**）。

- `Method getMethod(String name, 类<?>... parameterTypes)`

用于获取当前类的某一个指定名称 `public` 所修饰的成员方法（**包括父类**）。

- `Method[] getDeclaredMethods()`

用于获取当前类的所有 `public` 所修饰的成员方法（**不包括父类**）。

- `Method getDeclaredMethod(String name, 类<?>... parameterTypes)`

用于获取当前类的某一个指定名称 `public` 所修饰的成员方法（**不包括父类**）。

而当我们获取到类的成员方法后，如果要执行某一个方法，可以使用 `invoke()` 方法来执行该方法。

获取类名

```
package com.cunyu;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : Demo5
 * @date : 2021/4/8 14:06
 * @description : 获取类名
 */

public class Demo5 {
      public static void main(String[] args) throws ClassNotFoundException {
            Person person = new Person();
            Class personClass = person.getClass();

            String className = personClass.getName();
            System.out.println(className);
      }
}
```

```
com.cunyu.Person

Process finished with exit code 0
```

- `String getName()`

从上述程序的结果可知，当我们获取到 `Class` 对象之后，如果不知道类的全名，就可以使用 `getName()` 来获取该类的全名。

反射实例

假设我们有如下需求：在不改变类的代码的前提下，我们能够创建任意类的对象，并执行其中的方法。

此时，我们可以通过 **配置文件 + 反射** 的方式来实现这一效果，而这也就是我们现在所用框架中的基础，当我们使用反射后，只需要通过修改配置文件中的内容就能够不用去改代码就实现对应的功能。

假设我们有两个类，一个 `Student`，一个 `Teacher`，两者的定义如下；

```
package com.cunyu;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : Teacher
 * @date : 2021/4/8 15:15
 * @description : 教师类
```

```
 */

public class Teacher {
      private String name;
      private int age;

      public void teach() {
            System.out.println("教书育人……");
      }
}
```

```
package com.cunyu;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : Student
 * @date : 2021/4/8 15:16
 * @description : 学生类
 */

public class Student {
      private String name;
      private float score;

      public void study() {
            System.out.println("好好学习，天天向上……");
      }
}
```

要实现我们的需求，通常需要如下步骤：

1. 将要创建对象的全类名和要执行的方法都配置在配置文件中；

定义的配置文件 `prop.properties` ，其中主要内容包括 `className` 和 `methodName` 两个属性，分别代表类的全类名和要调用方法的名字。一个具体实例如下，分别代表名为 `Student` 的类和名为 `study` 的方法。

```
className=com.cunyu.Student
methodName=study
```

2. 然后在主方法中加载读取配置文件；

```
//          创建配置文件对象
Properties properties = new Properties();
//          加载配置文件
ClassLoader classLoader = ReflectTest.class.getClassLoader();
InputStream inputStream = classLoader.getResourceAsStream("prop.properties");
properties.load(inputStream);

//          获取配置文件中定义的数据
String className = properties.getProperty("className");
String methodName = properties.getProperty("methodName");
```

3. 利用反射技术将类加载到内存中；

```
//          加载进内存
Class name = Class.forName(className);
```

4. 接着利用 `newInstance()` 方法创建对象；

```
//          创建实例
Object object = name.newInstance();
```

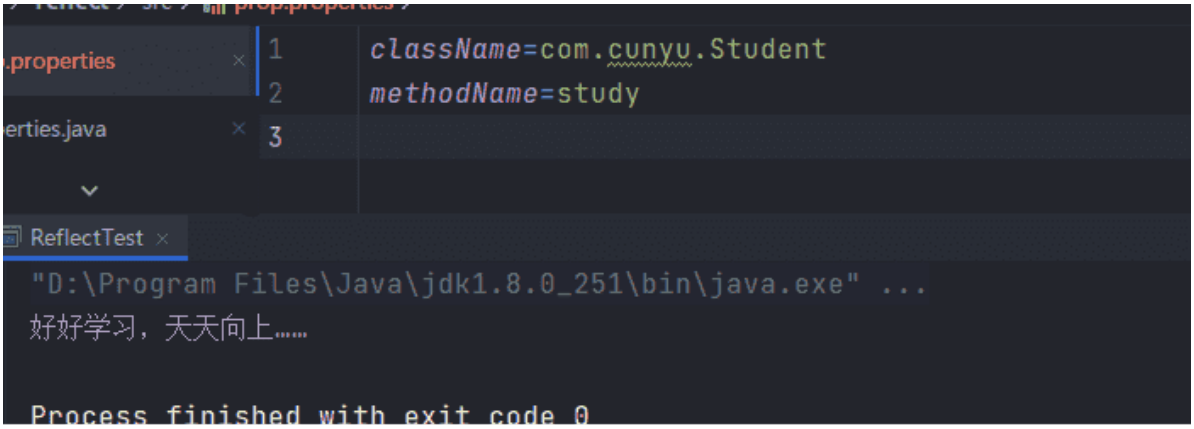
5. 最后则是利用 `invoke()` 方法来执行方法；

```
//          获取并执行方法
Method method = name.getMethod(methodName);
method.invoke(object);
```

将整个流程汇总起来就是：

```
package com.cunyu;

import java.io.IOException;
import java.io.InputStream;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Properties;

/**
 * @author : cunyu
 * @version : 1.0
 * @className : ReflectTest
 * @date : 2021/4/8 15:27
 * @description : 测试
 */

public class ReflectTest {
      public static void main(String[] args) throws IOException, ClassNotFoundException, IllegalAccessException, InstantiationException, InvocationTargetException, NoSuchMethodException {
//            创建配置文件对象
            Properties properties = new Properties();
//            加载配置文件
            ClassLoader classLoader = ReflectTest.class.getClassLoader();
            InputStream inputStream = classLoader.getResourceAsStream("prop.properties");
            properties.load(inputStream);

//            获取配置文件中定义的数据
            String className = properties.getProperty("className");
            String methodName = properties.getProperty("methodName");

//            加载进内存
            Class name = Class.forName(className);

//            创建实例
            Object object = name.newInstance();

//            获取并执行方法
            Method method = name.getMethod(methodName);
            method.invoke(object);
      }
}
```

此时，我们只需要改动配置文件 `prop.properties` 中的配置即可输出不同结果；


.properties
erties.java
1 className=com.cunyu.Student
2 methodName=study
3
ReflectTest
"D:\Program Files\Java\jdk1.8.0_251\bin\java.exe" ...
好好学习，天天向上.....
Process finished with exit code 0


D:\PersonalFiles\github\githu
Demo
1 className=com.cunyu.Teacher
2 methodName=teach
3
ReflectTest
"D:\Program Files\Java\jdk1.8.0_251\bin\java.exe" ...
教书育人.....
Process finished with exit code 0