# 1. JDK 8前的一些新特性

## 1.1 静态导入（import static）

使用import时，还能使用import static直接导入其他类中的静态方法，这样，调用时直接写方法名即可，不用加上所在的类名。例如：

|  |
| --- |
| **package** com.zhang.test;  *// 静态带入Math类的pow方法* **import static** java.lang.Math.*pow*;  **public class** Demo {  **public static void** main(String[] args) {  *// 计算2的3次方* System.***out***.println(*pow*(2, 3));  } } |

使用静态导入的前提是方法不能与本类静态方法同名，这和不能导入多个相同类名是一致的，因为这样JVM会不清楚究竟使用的是哪个类或者方法。这种情况只能写上类的全名。

一般情况下，也不会使用这种静态导入的方式。

## 1.2 增强for循环（for each）

引入增强for循环是为了简化数组和集合的遍历。for each的格式如下：

|  |
| --- |
| for(数组或集合中元素数据类型 变量A : 需遍历的数组或即可) {  此for循环中直接使用A元素，就是数组或集合中的单个变量  } |

例子：用for循环遍历数组和集合。

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.ArrayList;  **public class** Demo {  **public static void** main(String[] args) {  String[] strArr = {**"hello"**, **"world"**, **"java"**};  *// 遍历* **for**(String str : strArr) {  System.***out***.println(str);  }   ArrayList<String> strList = **new** ArrayList<>();  strList.add(**"hello"**);  strList.add(**"world"**);  strList.add(**"java"**);  *// foreach遍历* **for**(String str : strList) {  System.***out***.println(str);  }  } } |

可以看到使用for each后，遍历变得很简单。再比如Map集合的遍历：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.HashMap; **import** java.util.Map;  **public class** Demo {  **public static void** main(String[] args) {  Map<String, String> map = **new** HashMap<>();  map.put(**"2016"**, **"张三"**);  map.put(**"2017"**, **"李四"**);  map.put(**"2018"**, **"王五"**);  *// for遍历。实际上遍历的是entrySet()返回的Set集合。* **for**(Map.Entry<String, String> entry : map.entrySet()) {  System.***out***.println(**"键："** + entry.getKey());  System.***out***.println(**"值："** + entry.getValue());  }  } } |

注意的是，在for each中，也不能对集合元素进行删除修改等操作，否则同样会出现并发修改异常。遍历过程中，对变量A进行修改的操作是不会对集合或数组生效的。

## 1.3 可变参数

定义方法时，若不能确定方法参数的个数，那么就可以使用可变参数。比如现在有个方法，用于求n个数的和，那么这个参数就是不确定的，可用可变参数。可变参数的格式是：

修饰符 返回值类型 方法名(形参类型... 形参变量名)

实际上可变参数就是用数组实现的，可变参数的形参就可当数组使用。只是传递时，既能接收多个参数，也能接收数组形式，比较方便。例如上述加法的实现：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *// 可变参数的使用* System.***out***.println(*add*(1, 2, 3));  System.***out***.println(*add*(1, 2, 3, 4, 5));  System.***out***.println(*add*(**new int**[]{1, 2, 3}));  }   **public static int** add(**int** ... nums) {  **int** sum = 0;  **for**(**int** num : nums) {  sum += num;  }  **return** sum;  } } |

需要注意的是，一个方法中只能有一个可变参数，并且可变参数必须是方法的最后一个参数。这是符合逻辑的，如果可变参数在其他位置，就无法判断可变参数长度截止到哪里。

实际上我们在Arrays类里学的asList方法就是可变参数，即：

public static <T> List<T> asList(T... a)。把数组转换成集合。

## 1.4 自动装箱和拆箱

这点在讲集合时已经提过。

## 1.5 枚举类型

在枚举类中可以列举出此枚举类包含的值，这样有利于代码的规范，避免接收错误数据。

比如在“性别”这个枚举类中定义了“男”、“女”和“其他”三个枚举值，那么我们定义一个方法设置一个人的性别时，让该方法只接收“性别”这个枚举类型，那么，传递进方法的数据只能是“男”、“女”或者“其他”，不可能是别的数据。

定义枚举类的格式（使用enum关键字）：

|  |
| --- |
| 修饰符 enum 枚举类型名{  枚举项1,  枚举项2,  ...  枚举项n;  } |

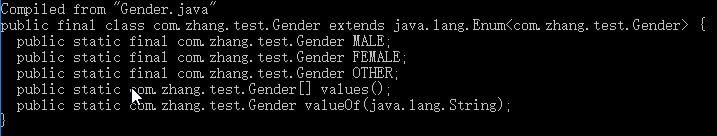
例如，定义的枚举类Gender为：

|  |
| --- |
| **package** com.zhang.test;  **public enum** Gender {  ***MALE***,  ***FEMALE***,  ***OTHER***; } |

然后主类中有个方法接受此枚举类型并输出。main方法调用此方法：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *showGender*(Gender.***MALE***);  }   **public static void** showGender(Gender gender) {  System.***out***.println(gender);  } } |

那么输出的直接是枚举项的字符串形式，即MALE。我们通过javap命令反编译Gender字节码文件，看看其中到底是什么。发现：



即枚举类其实也被是一个类，继承自Enum类，每个枚举值都是本类具体的对象，只不过是静态常量。并且枚举类提供了一些方法。

实际上我们可以在枚举类中写构造方法和成员。只是构造方法必须是私有的，在本类中使用，可以借此给枚举值设置一些信息。普通方法能在外界由各项枚举值使用，以此实现多种功能。

例如：

（1）修改的Gender枚举（各枚举项必须写在枚举类的开头位置）：

|  |
| --- |
| **package** com.zhang.test;  **public enum** Gender {  *// 各项枚举值必须在开头位置* ***MALE***(**"男"**),  ***FEMALE***(**"女"**),  ***OTHER***;  *// 成员变量，表示各项枚举值的名字* **private** String **name**;  **private** Gender(){} *// 无参构造  // 带参构造* **private** Gender(String name) {  **this**.**name** = name;  }  *// get set方法* **public** String getName() {  **return name**;  }  **public void** setName(String name) {  **this**.**name** = name;  } } |

（2）主类：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *showGender*(Gender.***MALE***);  }   **public static void** showGender(Gender gender) {  System.***out***.println(gender);  *// 调用其中getName方法* System.***out***.println(gender.getName());  } } |

枚举类中可以有抽象方法，但是要由具体的枚举类实现。即要写成这样（匿名内部类形式），比如：

|  |
| --- |
| **package** com.zhang.test;  **public enum** Gender {  ***MALE***() {  @Override  **public** String getInfo() {  **return "男"**;  }  },  ***FEMALE***() {  @Override  **public** String getInfo() {  **return "女"** }  },  ***OTHER***() {  @Override  **public** String getInfo() {  **return "其他"**;  }  };  **public abstract** String getInfo(); } |

然后主类也能调用枚举项的getInfo()方法。

枚举类继承了Enum类，查看Enum类API，发现以下实例方法很好用，可以被枚举项使用：

public final int ordinal()：返回该枚举类常量的序数，序数从0开始。

public final int compareTo(E o)：比较此枚举与指定对象的顺序。若相同返回0。只能与相同枚举类型的其他枚举常量进行比较。

当然，以前的equals()、toString()方法还是可用的。

Enum类还有一个静态方法valueOf：

public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)，此方法用于将字符串转换成枚举常量。比如：Gender male = Enum.valueOf(Gender.class, “MALE”)，此时male就是Gender.MALE枚举项。此方法的第二个参数就是传递Gender类的字节码。

JVM还会自动对枚举类插入两个静态方法：

（1）values()：获得此枚举类的所有枚举项的数组。比如：Gender[] gs = Gender.values()。

（2）valueOf(String name)：也是根据字符串获得对应的枚举项，只不过比Enum提供的方法简便，直接用Gender调用方法，不用传递Gender的字节码文件。即：Gender g = Gender.valueOf(“MALE”)。如果传递了枚举没有的字符串，那么这两个valueOf()方法都会抛出异常。

此外，枚举还能用在switch语句中。

## 1.6 数字字面量下划线

JDK 7后，数字的字面量中可以有下划线，此下划线作用是增强大数值的可阅读性，对实际数据值没有影响。例如：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  **int** binary = 0b1000\_0101\_1010; *// 二进制字面量* **int** hex = 0x1031\_FC21; *// 十六进制字面量* **double** decimal = 1000\_3213\_5.67; *// 十进制字面量* System.***out***.println(binary);  System.***out***.println(hex);  System.***out***.println(decimal);  } } |

注意：下划线不能出现在进制标识和数值之间、不能出现在数值开头和结尾、不能出现在小数点旁边。

## 1.7 try-with-resources（自动资源管理）

Java 7 增加了一个新特性，该特性提供了一种能自动关闭资源的资源管理方式。这个特性也被称为自动资源管理(Automatic Resource Management, ARM)， 该特性是基于try 语句进行的扩展，即使用try-with-resources的方式管理资源。

try-with-resources的使用格式：

|  |
| --- |
| try(需要自动释放的对象) {  在此作用域中使用资源对象...  } catch(异常) {  ........  } finally {  } |

也就是说，try-with-resources能用于对某些资源的自动释放，即把需要自动关闭的资源定义在try后的括号中（如果有多个资源，用分号分隔），然后只在try的大括号作用域中使用资源对象。这样，所使用的资源对象就能自动释放，而不需要开发者手动关闭。例如在IO流操作中，执行IO完毕后都需要手动调用close方法释放流对象，如果采用try-with-resources特性，流对象使用完后就可以由程序自动释放。例如：

|  |
| --- |
| **package** com.zhang.demo;  **import** java.io.\*;  **public class** Demo {  **public static void** main(String[] args) {  *copyFile*(**new** File(**"1.txt"**), **new** File(**"2.txt"**));  }   */\*\*  \* 实现文件的拷贝。  \** ***@param src*** *原文件  \** ***@param target*** *目标文件  \*/* **public static void** copyFile(File src, File target) {  **try** (FileInputStream fis = **new** FileInputStream(src);  FileOutputStream fos = **new** FileOutputStream(target)) {  **byte**[] buff = **new byte**[1024 \* 8];  **int** len = 0;  **while** ( (len = fis.read(buff)) != -1 ) {  fos.write(buff, 0, len);  }  *// 无需关闭资源* } **catch** (IOException e) {  **throw new** RuntimeException(e);  }  } } |

自动资源管理主要用于，当不再需要文件（或其他资源）时，可以防止无意中忘记释放它们。当try 代码块结束时，自动释放资源。因此不需要显示的调用 close() 方法。该形式也称为“带资源的 try 语句”。

主要注意的是：

（1）必须是java.lang.AutoCloseable的实现类对象才能进行自动释放。也就是说，try小括号中的资源对象所在类必须实现了AutoCloseable接口（或其子接口）。实际上，JDK中的流对象基本都实现了此接口（可以查看JDK 7及其以后的API文档），因此可以使用此特性进行自动释放资源。后续在Java EE中学习的数据库连接对象Connection也能用这种方式释放资源。

（2）try小括号中声明的资源会被隐式地声明为“final”修饰的对象，不能改变资源对象，且资源的作用域局限在try语句中。

说明：另外，关于NIO流的新特性将在第四部分教程中讲解。

# JDK 8中的新特性

## 2.1 Java 8的主要内容

Java 8的主要内容有：

（1）接口中的默认方法和静态方法

（2）新的时间日期API

（3）Optional类

（4）并行流

（5）函数式编程

其中，新特性中最为核心的就是函数式编程，其中包含了Lambda表达式、函数式接口、Stream API等内容，这部分内容将在下个文档中详细介绍。本章会讲解上面的1-4的特性。

当然Java 8还拥有很多变化，例如添加了对Base64编码的支持、改进了ConcurrentHashMap类实现的数据结构、改进了JVM等等，这些会在以后的开发中逐渐掌握，不再单独提出。

## 2.2. 接口中的默认方法和静态方法

Java 8中允许接口中包含拥有具体实现的默认方法和静态方法，分别用default和static关键字来修饰方法。理所当然，此时方法必须具体实现且不能使用abstract关键字。

默认方法的作用是给接口的实现类默认实现该具体方法（前提是实现类不重写该方法），当实现类对象调用该默认方法时，就会执行接口中定义的默认方法。

而对于静态方法，直接可通过“接口名.方法名”在程序中调用，显得更加灵活。

例子：

（1）DemoInterface.java接口代码：

|  |
| --- |
| **package** com.zhang.test;  **public interface** DemoInterface {  *// 静态方法* **public static void** staticShow() {  System.***out***.println(**"这是DemoInterface的静态方法"**);  }  *// 默认方法* **public default void** show() {  System.***out***.println(**"这是默认方法"**);  } } |

（2）实现类

|  |
| --- |
| **package** com.zhang.test;  **public class** DemoImpl **implements** DemoInterface {  *// 这里可什么也不实现。  // 当然也可以重写默认方法* } |

（3）主类

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *// 可直接调用接口静态方法* DemoInterface.*staticShow*();  DemoInterface di = **new** DemoImpl();  di.show(); *// 调用的是默认方法* } } |

需要注意的是，接口默认方法有“类优先”的原则：若一个类继承了某类和一个某接口，且该父类中有方法和该接口中的默认方法同名，那么执行时会选择父类的方法执行（子类未覆盖的情况下）。如果一个类实现了A和B两个接口，且A和B接口都有同名默认方法，那么该实现类需要重写方法来解决冲突。

## 2.3. Java 8的日期时间API

### 2.3.1 原有日期时间API存在的问题

JDK自1.0开始提供了Date类，自1.1开始提供了Calendar类。但实际上这些API并不是十分好用。并且，DateFormat是线程不安全的，这会导致在高并发时使用SimplaDateFormat类有线程安全问题。具体的请仔细阅读博客：

http://www.cnblogs.com/peida/archive/2013/05/31/3070790.html

其中详细解释了SimpleDateFormat的线程安全问题和解决办法。

下面我们将学习Java 8中的日期时间API，该全新的日期时间API是为了解决上述问题设计的，一方面使用比较简洁，另一方面，Java 8的日期时间对象是线程安全的，可在实际中进行应用。

### 2.3.2 使用LocalXXX和Instant

Java 8中提供了“LocalXXX”系列来表示日期和时间，他们分别是LocalDate、LocalTime和LocalDateTime。他们的用法相同，只不过LocalDate只能表示日期、LocalTime只能表示时间，而LocalDateTime能表示日期和时间。

LocalXXX使用ISO-8601日历系统标准。正如前面所说，LocalXXX类的实例是不可变的对象，是线程安全的。

现在我们以LocalDateTime讲解LocalXXX的使用。

（1）now()和of()静态方法来创建LocalXXX对象，分别根据当前时间和指定时间创建对象：

|  |
| --- |
| @Test **public void** test() {  *// 1. 使用now静态方法，根据当前时间创建LocalXXX对象* LocalDateTime localDateTime1 = LocalDateTime.*now*();  System.***out***.println(localDateTime1);  *// 上述直接输出结果为：2017-07-08T22:00:48.722  // 这就是ISO-8601标准格式   // 2. 使用of静态方法，根据指定日期/时间创建LocalXXX对象* LocalDateTime localDateTime2 = LocalDateTime.*of*(2016, 7, 8, 22, 3, 23);  System.***out***.println(localDateTime2); *// 输出结果：2016-07-08T22:03:23* } |

（2）plusXxx()和minusXxx()分别用于增加和减少某字段的时间。他们的返回值还是LocalXXX对象。即，对LocalXXX执行的大部分操作方法，还是会返回LocalXXX日期时间的实例。

|  |
| --- |
| @Test **public void** test() {  LocalDateTime nowTime = LocalDateTime.*now*();  *// 在当前时间上增加1小时* LocalDateTime newTime1 = nowTime.plusHours(1);  System.***out***.println(newTime1);   *// 在nowTime时间上减少1年* LocalDateTime newTime2 = nowTime.minusYears(1);  System.***out***.println(newTime2); } |

（3）用getXxx()和withXxx()分别获取和设置某日期时间属性。特别注意下面示例的withXxx()的用法。

|  |
| --- |
| @Test **public void** test() {  LocalDateTime localDateTime = LocalDateTime.*now*();  *// 1. getXxx()方法  // 获得当前是几日* **int** day = localDateTime.getDayOfMonth();  System.***out***.println(day);  *// 获得当前月份。* **int** month = localDateTime.getMonthValue();  *// 也可使用 int month = localDateTime.getMonth().getValue();  // 因为getMonth()得到的是Month对象。再对该对象进行操作。* System.***out***.println(month);   *// 2. with方法  // 设置小时、分钟...* localDateTime.withHour(20);  localDateTime.withMinute(50);  System.***out***.println(localDateTime);  *// 注意：上述直接“设置”是不行的，还是输出当前时间  // 因为之间就说过，LocalXXX是不可变的！！！  // 因此我们应该每次都接收新的LocalXXX对象！！！  // withXxx都会返回新的LocalXXX对象* LocalDateTime newLDT = localDateTime.withHour(20).withMinute(50);  System.***out***.println(newLDT); } |

当然，如果用LocalDate对象，就只能操作年、月、日属性，而用LocalTime就只能操作时、分、秒、毫秒、纳秒等属性。

下面我们学习Instant。用Instant代表时间戳，时间戳就是指所表示时间相距Unix元年 1970年1月1日 00:00:00的毫秒值。Instant默认使用UTC时区。

使用示例：

|  |
| --- |
| @Test **public void** test() {  *// 获得当前时间的Instant对象。默认使用UTC时区* Instant ins = Instant.*now*();  System.***out***.println(ins);  *// 上述输出的是：2017-07-09T06:37:36.822Z  // 最后的字母Z表示是UTC时区。但我们现在本地的时间实际上是  // 2017-07-09T14:37:36.822。也就是相差8小时（我们在东8区，相差8个时区）   // 我们也可让ins产生8小时的时区偏移：* OffsetDateTime odt = ins.atOffset(ZoneOffset.*ofHours*(8));  System.***out***.println(odt);  *// 上述就获得了本地时间，输出格式是：2017-07-09T14:37:36.822+08:00   // 我们也可用ofEpochSecond()方法来自己设置时间戳获得Instant对象。* Instant instant = Instant.*ofEpochSecond*(1000000000L); *// 相距10亿秒* System.***out***.println(instant);  *// 输出时间是：2001-09-09T01:46:40Z* } |

### 2.3.3 Duration和Period

Duration用于计算两个“时间”的间隔，Period用于计算两个“日期”的间隔。例：

|  |
| --- |
| @Test **public void** test() {  *// Duration对象使用。可处理LocalXXX和Instant对象的比较* LocalDateTime ldt1 = LocalDateTime.*now*();  LocalDateTime ldt2 = LocalDateTime.*of*(2017, 8, 8, 15, 32, 5);  Duration duration = Duration.*between*(ldt1, ldt2);  System.***out***.println(duration); *// 输出的是ISO-8601格式：PT720H8M24.159S，P表示间隔，即间隔720小时...  // 使用toXxx()方法获得某个具体的间隔值* System.***out***.println(duration.toDays()); *// 间隔的天数* System.***out***.println(duration.toHours());   *// Period对象。只能处理LocalDate和Instant对象的比较。  // 不能处理LocalTime和LocalDateTime这样带有“时间”的对象* LocalDate lt1 = LocalDate.*now*();  LocalDate lt2 = LocalDate.*of*(2008, 8, 8);   Period period = Period.*between*(lt1, lt2);  System.***out***.println(period); *// 输出格式是：P-8Y-11M-1D。前面带有负号，表示lt1时间比lt2时间迟。  // 同样可得到具体的年月日，用getXxx方法* System.***out***.println(period.getYears());  System.***out***.println(period.getMonths());  System.***out***.println(period.getDays()); } |

### 2.3.4 日期的操纵

我们之前使用withXxx方法方便实现了对LocalXXX对象日期时间的操纵。在Java 8 中，提供了更方便的“时间校正器”TemporalAdjuster来“一键操纵”日期时间。

例如我们想将日期时间调整到“下个星期天”或者“我的结婚纪念日”等，就可使用TemporalAdjuster。TemporalAdjuster是一个函数式接口，也就是说我们可自定义Lambda来调整日期时间。

同样，Java 8也内置了一个TemporalAdjusters类，该类中的方法返回的就是JDK定义好的TemporalAdjuster实现，我们可以直接使用。

还是使用LocalXXX对象的with(TemporalAdjuster adjuster)方法调整时间，其中传入时间校正器即可。

例如：

|  |
| --- |
| @Test **public void** test() {  LocalDate localDate = LocalDate.*now*();  *// 获得下周一的日期* LocalDate newDate = localDate.with(TemporalAdjusters.*next*(DayOfWeek.***MONDAY***));  System.***out***.println(newDate); } |

实际中可自行查看TemporalAdjusters中适合的方法。

但是如果我们想要返回本年的“结婚纪念日”怎么办？那就需要自己写Lambda了，假设结婚纪念日是10月1日。那么代码为：

|  |
| --- |
| @Test **public void** test() {  LocalDate localDate = LocalDate.*now*();  *// 将时间调整为结婚纪念日* LocalDate marryDate = localDate.with((date) -> {  *// 强转为LocalDate* LocalDate temp = (LocalDate) date;  *// 返回本年结婚纪念日* **return** temp.withMonth(10).withDayOfMonth(1);  });  System.***out***.println(marryDate); } |

关于Lambda可参见下一章文档。

### 2.3.5 解析和格式化日期时间

我们使用DateTimeFormatter来解析和格式化Java 8的日期时间。DateTimeFormatter提供了静态方法ofPattern(String pattern)静态方法，该方法可通过传入的格式来获得DateTimeFormatter对象。

然后，利用LocalXXX的实例方法format()和静态方法parse()方法（要传入DateTimeFormatter对象）对日期时间进行格式化和解析，例：

|  |
| --- |
| @Test **public void** test() {  *// 当前时间* LocalDateTime localDateTime = LocalDateTime.*now*();  *// 获得DateTimeFormatter对象* DateTimeFormatter dtf = DateTimeFormatter.*ofPattern*(**"yyyy年MM月dd日 HH:mm:ss"**);  *// 格式化* String dateStr = localDateTime.format(dtf);  System.***out***.println(dateStr);   *// 也可解析* LocalDateTime parsedTime = LocalDateTime.*parse*(dateStr, dtf);  System.***out***.println(parsedTime); } |

使用DateTimeFormatter就没有线程安全问题。

### 2.3.6 时区的处理

Java8 中加入了对时区的支持，可用ZonedXXX类表示带时区的日期时间，即：

ZonedDate、ZonedTime和ZonedDateTime。每个时区都对应着 ID，地区ID都为 “{区域}/{城市}”的格式，例如：“Asia/Shanghai”等。

代码：

|  |
| --- |
| @Test **public void** test() {  ZonedDateTime zdt = ZonedDateTime.*now*(ZoneId.*of*(**"Asia/Shanghai"**));  System.***out***.println(zdt);  *// 输出格式：2017-07-09T20:24:33.791+08:00[Asia/Shanghai]* } |

可以用下面方法获得JDK支持的所有时区：

|  |
| --- |
| @Test **public void** test() {  Set<String> set = ZoneId.*getAvailableZoneIds*();  set.forEach(System.***out***::println); } |

## 2.4. Optional类

Optional<T> 类是一个容器类，代表一个值存在或不存在，可以较好地避免空指针异常。

常用方法：

（1）Optional.of(T t) : 创建一个 Optional 实例

（2）Optional.empty() : 创建一个空的 Optional 实例

（3）Optional.ofNullable(T t):若t不为 null，创建 Optional 实例，否则创建空实例

（4）isPresent() : 判断是否包含值

（5）orElse(T t) : 如果调用对象包含值，返回该值，否则返回t

（6）orElseGet(Supplier s) :如果调用对象包含值，返回该值，否则返回s获取的值

（7）map(Function f): 如果有值对其处理，并返回处理后的Optional，否则返回 Optional.empty()

（8）flatMap(Function mapper):与 map 类似，要求返回值必须是Optional

|  |
| --- |
| @Test **public void** test() {  Optional<Student> op = Optional.*of*(**new** Student());  System.***out***.println(op.get()); *// 其他方法自行使用* } |

## 2.5. 并行流与串行流

当执行数据量大、计算量大的任务时，一般采取并行计算。并行流就是把一个内容分成多个数据块，并用不同的线程分别处理每个数据块的流。

相较于串行流，并行流能有效利用CPU的多个核心，执行效率高，处理大数据花费的时间比串行流少（因为串行流只是用一个核心计算）。

并不是所有的任务都适合用并行流，因为要考虑任务是否可以拆分。若想要使用并行计算，那么需要拆分任务，使各个任务以并行的方式分别执行得到各个并行的结果，最后还要把各个结果合并为该任务的最终结果。

实际上在Java 7中，就提供了“Fork/Join”框架用于执行并行任务。Fork/Join框架就是在必要的情况下，将一个大任务拆分(fork)成若干个小任务（拆到不可再拆时），再将一个个的小任务运算的结果进行join汇总。

下面我们采用Fork/Join实现数据累加的运算，该运算可拆分并行计算。

（1）首先定义一个类来描述计算任务，其中规定了如何计算、拆分和合并。该类需要继承RecursiveTask<T>。RecursiveTask表示该任务有返回值，如果某任务无返回值，则继承RecursiveAction即可。Recursive的意思是“递归”，代码中有所体现【递归拆分】。

|  |
| --- |
| **package** java8.demo;  **import** java.util.concurrent.RecursiveTask;  **public class** SumCalcTask **extends** RecursiveTask<Long> {  *// 定义开始和结束的拆分位置* **private long start**;  **private long** end;  *// 定义拆分的临界值(一百万)* **private static final long *THRESHOLD*** = 1000000L;   *// 用构造来传递当前要计算的结果* **public** SumCalcTask(**long** start, **long** end) {  **this**.**start** = start;  **this**.end = end;  }   *// 需要实现的计算方法* @Override  **protected** Long compute() {  **long** length = end - **start**;  **if** (length <= ***THRESHOLD***) {  *// 如果计算范围在临界值之内，就不拆分了，直接计算本范围的结果并返回。* **long** sum = 0;  **for** (**long** i = **start**; i <= end; i++) {  sum += i;  }  **return** sum;  } **else** {  *// 超过临界值，就拆分成两部分。根据start和end范围的中间值来拆分* **long** middle = (**start** + end) / 2;  *// 分成“左”和“右”两个任务，重新传入本类构造中进行运算。所以叫递归拆分。* SumCalcTask left = **new** SumCalcTask(**start**, middle);  left.fork(); *//fork拆分，并将该子任务压入线程队列* SumCalcTask right = **new** SumCalcTask(middle + 1, end);  right.fork();  *// 最后返回各分支的相加结果。  // 由于递归，实际可能拆分成n个任务并行，但这里只要考虑left和join即可，这也是递归的作用* **return** left.join() + right.join();  }  } } |

（2）测试：

|  |
| --- |
| @Test **public void** test() {  *// 用Java 8的时间对象计算时间* Instant start = Instant.*now*(); *// 开始时间* ForkJoinPool pool = **new** ForkJoinPool();  ForkJoinTask<Long> task = **new** SumCalcTask(1, 50000000000L);  **long** sum = pool.invoke(task);  Instant end = Instant.*now*(); *// 结束时间* System.***out***.println(Duration.*between*(start, end).toMillis()); *// 输出毫秒值* System.***out***.println(sum); } |

这样就完成了累加的并行计算。注意，当数据量越大、计算量越大时，上述的并行计算才会体现出优势，因为Fork/Join操作是需要耗费时间的，当数据量大时，Fork/Join时间远小于计算时间，因此有优势。

我们也可以写一段普通的串行代码比较他们各自花费的时间和CPU的利用效率。发现并行时，会充分利用各个CPU，CPU使用率能够达到90%以上，而使用串行流只会达到50%左右（没有看到只有一个CPU核心利用率达到90%以上的情况，那是因为操作系统也会进行调控）。因此并行计算提高了CPU利用率，节约了时间，适合大运算量。（同时，代码中的拆分临界值也会影响执行的速度，可以按需设置地大一些，太小的话会导致拆分太多）

但是由上我们也发现，使用Fork/Join框架比较复杂。Java 8中将并行进行了优化，我们可以很容易的对数据进行并行操作。 Stream API可以声明性地通过parallel()与sequential() 在并行流与顺序流之间切换。

下面是使用Java 8并行流完成的累加计算。

|  |
| --- |
| @Test **public void** test2() {  **long** sum = LongStream.*rangeClosed*(1L, 5000000000L)  .parallel() *// 并行流* .sum(); *// 求和* System.***out***.println(sum); } |

## 2.6 小扩展

（1）Java8改进的底层数据结构：例如ConcurrentHashMap改用了CAS算法、HashMap和HashSet中改用“位桶+链表+红黑树”实现底层数据结构（之前使用的是“位桶+链表”结构），以此改善了查询速度。具体参考网址： “ http://blog.csdn.net/yinbingqiu/article/details/60965080 ”。

（2）Java8改进JVM：Hotspot中不再有“永久区PermGen”（永久区用于存储加载的类信息，包括方法区也在永久区中），而改为使用“元空间Meta Space”，现在的元空间直接使用的是物理内存。这样，JVM调优参数也发生变化，不再有PermGenSize和MaxPermGenSize参数，取而代之的是MetaSpaceSize和MaxMetaSpaceSize。