

# JDK 8新特性

## 课程介绍

1. Java SE的发展历史
2. 了解Open JDK 和 Oracle JDK
3. JDK 8新特性
  - **Lambda表达式**
  - **集合之Stream流式操作**
  - 接口的增强
  - 并行数组排序
  - Optional中避免Null检查
  - 新的时间和日期 API
  - 可重复注解

## 适合人群

在校大学生，学习过Java，渴望了解Java前沿技术的同学。

正在从事Java编程工作。

Java 8发布于2014-03-18，发布至今已经5年了，是目前企业中使用最广泛的一个版本。Java 8是一次重大的版本升级，带来了很多的新特性。在本教程中我们将会学习以下这些新特性。

## Lambda表达式介绍

### 目标

了解使用匿名内部类存在的问题

体验Lambda

### 使用匿名内部类存在的问题

当需要启动一个线程去完成任务时，通常会通过 `Runnable` 接口来定义任务内容，并使用 `Thread` 类来启动该线程。

传统写法,代码如下:

```
public class Demo01LambdaIntro {  
    public static void main(String[] args) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("新线程任务执行!");  
            }  
        }).start();  
    }  
}
```

由于面向对象的语法要求，首先创建一个 `Runnable` 接口的匿名内部类对象来指定线程要执行的任务内容，再将其交给一个线程来启动。

### 代码分析:

对于 `Runnable` 的匿名内部类用法，可以分析出几点内容：

- `Thread` 类需要 `Runnable` 接口作为参数，其中的抽象 `run` 方法是用来指定线程任务内容的核心
- 为了指定 `run` 的方法体，**不得不**需要 `Runnable` 接口的实现类
- 为了省去定义一个 `Runnable` 实现类的麻烦，**不得不**使用匿名内部类
- 必须覆盖重写抽象 `run` 方法，所以方法名称、方法参数、方法返回值**不得不再**写一遍，且不能写错
- 而实际上，**似乎只有方法体才是关键所在。**

## Lambda体验

Lambda是一个**匿名函数**，可以理解为一小段可以传递的代码。

### Lambda表达式写法,代码如下:

借助Java 8的全新语法，上述 `Runnable` 接口的匿名内部类写法可以通过更简单的Lambda表达式达到相同的效果

```
public class Demo01LambdaIntro {  
    public static void main(String[] args) {  
        new Thread(() -> System.out.println("新线程任务执行!")).start(); // 启动线程  
    }  
}
```

这段代码和刚才的执行效果是完全一样的，可以在JDK 8或更高的编译级别下通过。从代码的语义中可以看出：我们启动了一个线程，而线程任务的内容以一种更加简洁的形式被指定。

我们只需要将要执行的代码放到一个Lambda表达式中，不需要定义类，不需要创建对象。

## Lambda的优点

简化匿名内部类的使用，语法更加简单。

## 小结

了解了匿名内部类语法冗余，体验了Lambda表达式的使用，发现Lambda是简化匿名内部类的简写

## Lambda的标准格式

## 目标

掌握Lambda的标准格式

练习无参数无返回值的Lambda

练习有参数有返回值的Lambda

## Lambda的标准格式

Lambda省去面向对象的条条框框，Lambda的标准格式由**3个部分**组成：

```
(参数类型 参数名称) -> {  
    代码体;  
}
```

格式说明：

- (参数类型 参数名称)：参数列表
- {代码体}：方法体
- ->：箭头，分隔参数列表和方法体

### Lambda与方法的对比

匿名内部类

```
public void run() {  
    System.out.println("aa");  
}
```

Lambda

```
() -> System.out.println("bb! ")
```

## 练习无参数无返回值的Lambda

掌握了Lambda的语法，我们来通过一个案例熟悉Lambda的使用。

```
interface Swimmable {  
    public abstract void swimming();  
}
```

```
package com.itheima.demo01lambda;  
  
public class Demo02LambdaUse {  
    public static void main(String[] args) {  
        goSwimming(new Swimmable() {
```

```
@Override
public void swimming() {
    System.out.println("匿名内部类游泳");
}

});

goSwimming(() -> {
    System.out.println("Lambda游泳");
});

}

public static void goSwimming(Swimmable swimmable) {
    swimmable.swimming();
}

}
```

## 练习有参数有返回值的Lambda

下面举例演示 `java.util.Comparator<T>` 接口的使用场景代码，其中的抽象方法定义为：

- `public abstract int compare(T o1, T o2);`

当需要对一个对象集合进行排序时，`Collections.sort` 方法需要一个 `Comparator` 接口实例来指定排序的规则。

### 传统写法

如果使用传统的代码对 `ArrayList` 集合进行排序，写法如下：

```
public class Person {
    private String name;
    private int age;
    private int height;
    // 省略其他
}
```

```
package com.itheima.demo01lambda;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;

public class Demo03LambdaUse {
    public static void main(String[] args) {
        ArrayList<Person> persons = new ArrayList<>();
        persons.add(new Person("刘德华", 58, 174));
        persons.add(new Person("张学友", 58, 176));
        persons.add(new Person("刘德华", 54, 171));
        persons.add(new Person("黎明", 53, 178));
```

```
Collections.sort(persons, new Comparator<Person>() {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.getAge() - o2.getAge();
    }
});

for (Person person : persons) {
    System.out.println(person);
}
}
```

这种做法在面向对象的思想中，似乎也是“理所当然”的。其中 `Comparator` 接口的实例（使用了匿名内部类）代表了“按照年龄从小到大”的排序规则。

### Lambda写法

```
package com.itheima.demo01lambda;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;

public class Demo03LambdaUse {
    public static void main(String[] args) {
        ArrayList<Person> persons = new ArrayList<>();
        persons.add(new Person("刘德华", 58, 174));
        persons.add(new Person("张学友", 58, 176));
        persons.add(new Person("刘德华", 54, 171));
        persons.add(new Person("黎明", 53, 178));

        Collections.sort(persons, (o1, o2) -> {
            return o1.getAge() - o2.getAge();
        });

        for (Person person : persons) {
            System.out.println(person);
        }

        System.out.println("-----");

        List<Integer> list = Arrays.asList(11, 22, 33, 44);

        list.forEach(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) {
                System.out.println(integer);
            }
        });
    }
}
```

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

        list.forEach((s) -> {
            System.out.println(s);
        });
    }
}
```

## 小结

首先学习了Lambda表达式的标准格式

```
(参数列表) -> {
    方法体;
}
```

以后我们调用方法时,看到参数是接口就可以考虑使用Lambda表达式,Lambda表达式相当于是对接口中抽象方法的重写

## 了解Lambda的实现原理

### 目标

了解Lambda的实现原理



我们现在已经会使用Lambda表达式了。现在同学们肯定很好奇Lambda是如何实现的，现在我们就来探究Lambda表达式的底层实现原理。

```
@FunctionalInterface
interface Swimmable {
    public abstract void swimming();
}
```

```
public class Demo04LambdaImpl {
    public static void main(String[] args) {
        goSwimming(new Swimmable() {
            @Override
            public void swimming() {
                System.out.println("使用匿名内部类实现游泳");
            }
        });
    }

    public static void goSwimming(Swimmable swimmable) {
        swimmable.swimming();
    }
}
```

我们可以看到匿名内部类会在编译后产生一个类： Demo04LambdaImpl\$1.class

 Demo04LambdaImpl\$1.class  
 Demo04LambdaImpl.class

使用Xjad反编译这个类，得到如下代码：

```
package com.itheima.demo01lambda;

import java.io.PrintStream;

// Referenced classes of package com.itheima.demo01lambda:
//      Swimmable, Demo04LambdaImpl

static class Demo04LambdaImpl$1 implements Swimmable {
    public void swimming()
    {
        System.out.println("使用匿名内部类实现游泳");
    }

    Demo04LambdaImpl$1() {
    }
}
```

我们再来看看Lambda的效果，修改代码如下：

```
public class Demo04LambdaImpl {
    public static void main(String[] args) {
        goSwimming() -> {
            System.out.println("Lambda游泳");
        });
    }

    public static void goSwimming(Swimmable swimmable) {
        swimmable.swimming();
    }
}
```

运行程序，控制台可以得到预期的结果，但是并没有出现一个新的类，也就是说Lambda并没有在编译的时候产生一个新的类。使用Xjad对这个类进行反编译，发现Xjad报错。使用了Lambda后Xjad反编译工具无法反编译。我们使用JDK自带的一个工具： `javap`，对字节码进行反汇编，查看字节码指令。

在DOS命令行输入：

```
javap -c -p 文件名.class
```

-c: 表示对代码进行反汇编  
-p: 显示所有类和成员



反汇编后效果如下：

```
C:\Users\>javap -c -p Demo04LambdaImpl.class
Compiled from "Demo04LambdaImpl.java"
public class com.itheima.demo01lambda.Demo04LambdaImpl {
    public com.itheima.demo01lambda.Demo04LambdaImpl();
        Code:
            0: aload_0
            1: invokespecial #1                  // Method java/lang/Object."<init>":()V
            4: return

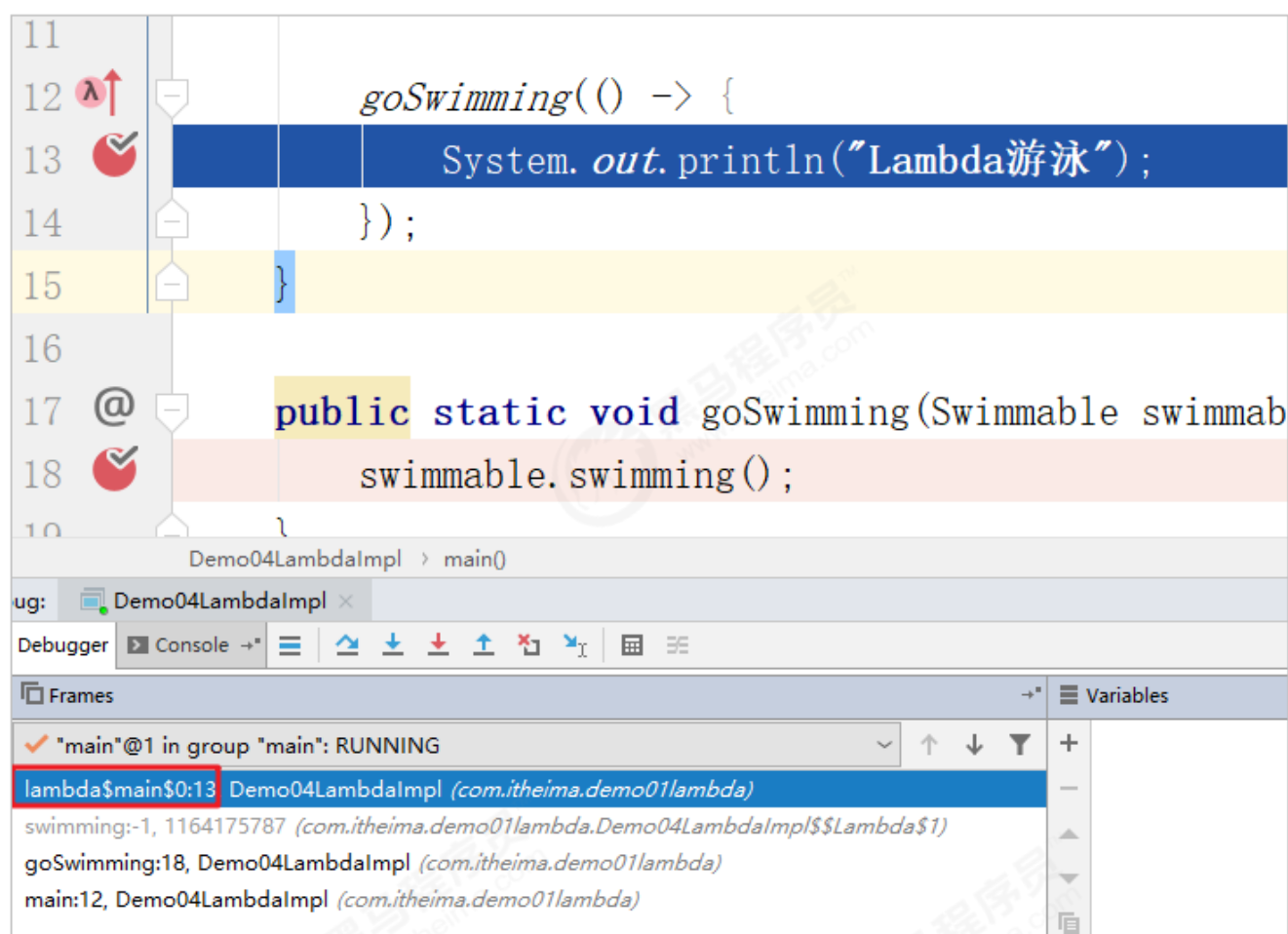
    public static void main(java.lang.String[]);
        Code:
            0: invokedynamic #2, 0              // Invokedynamic #0:swimming:
            (Lcom/itheima/demo01lambda/Swimmable;
            5: invokestatic #3                  // Method goSwimming:
            (Lcom/itheima/demo01lambda/Swimmable;)V
            8: return

    public static void goSwimming(com.itheima.demo01lambda.Swimmable);
        Code:
            0: aload_0
            1: invokeinterface #4, 1            // InterfaceMethod
            com/itheima/demo01lambda/Swimmable.swimming:()V
            6: return

    private static void lambda$main$0();
        Code:
            0: getstatic      #5                // Field
            java/lang/System.out:Ljava/io/PrintStream;
            3: ldc           #6                  // String Lambda游泳
            5: invokevirtual #7                // Method java/io/PrintStream.println:
            (Ljava/lang/String;)V
            8: return
}
```

可以看到在类中多出了一个私有的静态方法 `lambda$main$0`。这个方法里面放的是什么呢？我们通过断点调试来看看：





可以确认 `lambda$main$0` 里面放的就是Lambda中的内容，我们可以这么理解 `lambda$main$0` 方法：

```
public class Demo04LambdaImpl {  
    public static void main(String[] args) {  
        ...  
    }  
  
    private static void lambda$main$0() {  
        System.out.println("Lambda游泳");  
    }  
}
```

关于这个方法 `lambda$main$0` 的命名：以 `lambda` 开头，因为是在 `main()` 函数里使用了 `lambda` 表达式，所以带有 `$main` 表示，因为是第一个，所以 `$0`。

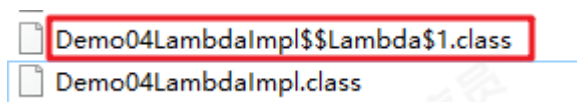
如何调用这个方法呢？其实 `Lambda` 在运行的时候会生成一个内部类，为了验证是否生成内部类，可以在运行时加上 `-Djdk.internal.lambda.dumpProxyClasses`，加上这个参数后，运行时会将生成的内部类 `class` 码输出到一个文件中。使用 `java` 命令如下：

```
java -Djdk.internal.lambda.dumpProxyClasses 要运行的包名.类名
```

根据上面的格式，在命令行输入以下命令：

```
C:\Users\>java -Djdk.internal.lambda.dumpProxyClasses
com.itheima.demo01lambda.Demo04LambdaImpl
Lambda游泳
```

执行完毕，可以看到生成一个新的类，效果如下：



反编译 Demo04LambdaImpl\$\$Lambda\$1.class 这个字节码文件，内容如下：

```
// Referenced classes of package com.itheima.demo01lambda:
//      Swimmable, Demo04LambdaImpl

final class Demo04LambdaImpl$$Lambda$1 implements Swimmable {

    public void swimming()
    {
        Demo04LambdaImpl.lambda$main$0();
    }

    private Demo04LambdaImpl$$Lambda$1()
    {
    }
}
```

可以看到这个匿名内部类实现了 `Swimmable` 接口，并且重写了 `swimming` 方法，`swimming` 方法调用 `Demo04LambdaImpl.lambda$main$0()`，也就是调用Lambda中的内容。最后可以将Lambda理解为：

```
public class Demo04LambdaImpl {
    public static void main(String[] args) {

        goSwimming(new Swimmable() {
            public void swimming() {
                Demo04LambdaImpl.lambda$main$0();
            }
        });

        private static void lambda$main$0() {
            System.out.println("Lambda表达式游泳");
        }

        public static void goSwimming(Swimmable swimmable) {
            swimmable.swimming();
        }
}
```

## 小结

匿名内部类在编译的时候会生成一个class文件

Lambda在程序运行的时候形成一个类

1. 在类中新增一个方法,这个方法的方法体就是Lambda表达式中的代码
2. 还会形成一个匿名内部类,实现接口,重写抽象方法
3. 在接口的重写方法中会调用新生成的方法.

## Lambda省略格式

### 目标

掌握Lambda省略格式

在Lambda标准格式的基础上，使用省略写法的规则为：

1. 小括号内参数的类型可以省略
2. 如果小括号内有**且仅有一个参数**，则小括号可以省略
3. 如果大括号内有**且仅有一个语句**，可以同时省略大括号、return关键字及语句分号

```
(int a) -> {  
    return new Person();  
}
```

省略后

```
a -> new Person()
```

## Lambda的前提条件

### 目标

掌握Lambda的前提条件

Lambda的语法非常简洁，但是Lambda表达式不是随便使用的，使用时有几个条件要特别注意：

1. 方法的参数或局部变量类型必须为接口才能使用Lambda
2. 接口中有且仅有一个抽象方法

```
public interface Flyable {  
    public abstract void flying();  
}
```

```
package com.itheima.demo01lambda;
```

```
public class Demo05LambdaCondition {  
    public static void main(String[] args) {  
        test01(() -> {  
        });  
  
        Flyable s = new Flyable() {  
            @Override  
            public void flying() {  
            }  
        };  
  
        Flyable s2 = () -> {  
        };  
    }  
  
    public static void test01(Flyable fly) {  
        fly.flying();  
    }  
}
```

## 小结

Lambda表达式的前提条件:

1. 方法的参数或变量的类型是接口
2. 这个接口中只能有一个抽象方法

## 函数式接口

函数式接口在Java中是指：**有且仅有一个抽象方法的接口**。

函数式接口，即适用于函数式编程场景的接口。而Java中的函数式编程体现就是Lambda，所以函数式接口就是可以适用于Lambda使用的接口。只有确保接口中有且仅有一个抽象方法，Java中的Lambda才能顺利地进行推导。

FunctionalInterface注解

与 `@Override` 注解的作用类似，Java 8中专门为函数式接口引入了一个新的注解：`@FunctionalInterface`。该注解可用于一个接口的定义上：

```
@FunctionalInterface  
public interface Operator {  
    void myMethod();  
}
```

一旦使用该注解来定义接口，编译器将会强制检查该接口是否确实有且仅有一个抽象方法，否则将会报错。不过，即使不使用该注解，只要满足函数式接口的定义，这仍然是一个函数式接口，使用起来都一样。

## Lambda和匿名内部类对比

### 目标

了解Lambda和匿名内部类在使用上的区别

1. 所需的类型不一样

匿名内部类,需要的类型可以是类,抽象类,接口

Lambda表达式,需要的类型必须是接口

2. 抽象方法的数量不一样

匿名内部类所需的接口中抽象方法的数量随意

Lambda表达式所需的接口只能有一个抽象方法

3. 实现原理不同

匿名内部类是在编译后会形成class

Lambda表达式是在程序运行的时候动态生成class

## 小结

当接口中只有一个抽象方法时,建议使用Lambda表达式,其他其他情况还是需要使用匿名内部类

## JDK 8接口新增的两个方法

### 目标

了解JDK 8接口新增的两个方法

掌握接口默认方法的使用

掌握接口静态方法的使用

## JDK 8接口增强介绍

JDK 8以前的接口:

```
interface 接口名 {  
    静态常量;  
    抽象方法;  
}
```

JDK 8对接口的增强, 接口还可以有**默认方法**和**静态方法**

JDK 8的接口:



```
interface 接口名 {  
    静态常量;  
    抽象方法;  
    默认方法;  
    静态方法;  
}
```

## 接口引入默认方法的背景

在JDK 8以前接口中只能有抽象方法。存在以下问题：

如果给接口新增抽象方法，所有实现类都必须重写这个抽象方法。不利于接口的扩展。

```
interface A {  
    public abstract void test1();  
    // 接口新增抽象方法,所有实现类都需要去重写这个方法,非常不利于接口的扩展  
    public abstract void test2();  
}  
  
class B implements A {  
    @Override  
    public void test1() {  
        System.out.println("BB test1");  
    }  
  
    // 接口新增抽象方法,所有实现类都需要去重写这个方法  
    @Override  
    public void test2() {  
        System.out.println("BB test2");  
    }  
}  
  
class C implements A {  
    @Override  
    public void test1() {  
        System.out.println("CC test1");  
    }  
  
    // 接口新增抽象方法,所有实现类都需要去重写这个方法  
    @Override  
    public void test2() {  
        System.out.println("CC test2");  
    }  
}
```

例如，JDK 8 时在Map接口中增加了 `forEach` 方法：

```
public interface Map<K, V> {  
    ...  
    abstract void forEach(BiConsumer<? super K, ? super V> action);  
}
```

通过API可以查询到Map接口的实现类如：

### Interface Map<K,V>

所有已知实现类：

AbstractMap, Attributes, AuthProvider, ClipboardContent, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, Headers, IdentityHashMap, LinkedHashMap, MapBinding, MapExpression, MapProperty, MapPropertyBase, MultiMapResult, PrinterStateReasons, Properties, Provider, ReadOnlyMapProperty, ReadOnlyMapPropertyBase, ReadOnlyMapWrapper, RenderingHints, ScriptObjectMirror, SimpleBindings, SimpleMapProperty, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

如果在Map接口中增加一个抽象方法，所有的实现类都需要去实现这个方法，那么工程量是巨大的。

因此，在JDK 8时为接口新增了默认方法，效果如下：

```
public interface Map<K, V> {  
    ...  
    default void forEach(BiConsumer<? super K, ? super V> action) {  
        ...  
    }  
}
```

接口中的默认方法实现类不必重写，可以直接使用，实现类也可以根据需要重写。这样就方便接口的扩展。

## 接口默认方法的定义格式

```
interface 接口名 {  
    修饰符 default 返回值类型 方法名() {  
        代码;  
    }  
}
```

## 接口默认方法的使用

方式一：实现类直接调用接口默认方法

方式二：实现类重写接口默认方法

```
package com.itheima.demo02interfaceupgrade;
```



```
public class Demo02UserDefaultFunction {
    public static void main(String[] args) {
        BB b = new BB();
        // 方式一：实现类直接调用接口默认方法
        b.test02();

        CC c = new CC();
        // 调用实现类重写接口默认方法
        c.test02();
    }
}

interface AA {
    public abstract void test1();

    public default void test02() {
        System.out.println("AA test02");
    }
}

class BB implements AA {
    @Override
    public void test1() {
        System.out.println("BB test1");
    }
}

class CC implements AA {
    @Override
    public void test1() {
        System.out.println("CC test1");
    }

    // 方式二：实现类重写接口默认方法
    @Override
    public void test02() {
        System.out.println("CC实现类重写接口默认方法");
    }
}
```

## 接口静态方法

为了方便接口扩展，JDK 8为接口新增了静态方法。

### 接口静态方法的定义格式



```
interface 接口名 {  
    修饰符 static 返回值类型 方法名() {  
        代码;  
    }  
}
```

## 接口静态方法的使用

直接使用接口名调用即可：接口名.静态方法名();

代码

```
public class Demo04UseStaticFunction {  
    public static void main(String[] args) {  
        // 直接使用接口名调用即可：接口名.静态方法名();  
        AAA.test01();  
    }  
}  
  
interface AAA {  
    public static void test01() {  
        System.out.println("AAA 接口的静态方法");  
    }  
}  
  
class BBB implements AAA {  
    /*    @Override 静态方法不能重写  
    public static void test01() {  
        System.out.println("AAA 接口的静态方法");  
    }*/  
}
```

## 接口默认方法和静态方法的区别

1. 默认方法通过实例调用，静态方法通过接口名调用。
2. 默认方法可以被继承，实现类可以直接使用接口默认方法，也可以重写接口默认方法。
3. 静态方法不能被继承，实现类不能重写接口静态方法，只能使用接口名调用。

## 小结

接口中新增的两种方法：

默认方法和静态方法

如何选择呢？如果这个方法需要被实现类继承或重写，使用默认方法，如果接口中的方法不需要被继承就使用静态方法

## 常用内置函数式接口

### 目标

了解内置函数式接口由来

了解常用内置函数式接口

### 内置函数式接口来由来

我们知道使用Lambda表达式的前提是需要有函数式接口。而Lambda使用时不关心接口名，抽象方法名，只关心抽象方法的参数列表和返回值类型。因此为了让我们使用Lambda方便，JDK提供了大量常用的函数式接口。

```
package com.itheima.demo03functionalinterface;

import java.util.List;

public class Demo01UserFunctionalInterface {
    public static void main(String[] args) {
        // 调用函数式接口中的方法
        method((arr) -> {
            int sum = 0;
            for (int n : arr) {
                sum += n;
            }
            return sum;
        });
    }

    // 使用自定义的函数式接口作为方法参数
    public static void method(Operator op) {
        int[] arr = {1, 2, 3, 4};
        int sum = op.getSum(arr);
        System.out.println("sum = " + sum);
    }
}

@FunctionalInterface
interface Operator {
    public abstract int getSum(int[] arr);
}
```

### 常用内置函数式接口介绍

它们主要在 `java.util.function` 包中。下面是最常用的几个接口。

#### 1. Supplier接口

```
@FunctionalInterface
public interface Supplier<T> {
    public abstract T get();
}
```

## 2. Consumer接口

```
@FunctionalInterface
public interface Consumer<T> {
    public abstract void accept(T t);
}
```

## 3. Function接口

```
@FunctionalInterface
public interface Function<T, R> {
    public abstract R apply(T t);
}
```

## 4. Predicate接口

```
@FunctionalInterface
public interface Predicate<T> {
    public abstract boolean test(T t);
}
```

Predicate接口用于做判断,返回boolean类型的值

# Supplier接口

`java.util.function.Supplier<T>` 接口, 它意味着"供给", 对应的Lambda表达式需要**“对外提供”**一个符合泛型类型的对象数据。

```
@FunctionalInterface
public interface Supplier<T> {
    public abstract T get();
}
```

供给型接口, 通过Supplier接口中的get方法可以得到一个值, 无参有返回的接口。

### 使用Lambda表达式返回数组元素最大值

使用 `Supplier` 接口作为方法参数类型, 通过Lambda表达式求出int数组中的最大值。提示: 接口的泛型请使用 `java.lang.Integer` 类。

代码示例:



```
public class Demo05Supplier {
    public static void main(String[] args) {
        printMax(() -> {
            int[] arr = {10, 20, 100, 30, 40, 50};
            // 先排序,最后就是最大的
            Arrays.sort(arr);
            return arr[arr.length - 1]; // 最后就是最大的
        });
    }

    private static void printMax(Supplier<Integer> supplier) {
        int max = supplier.get();
        System.out.println("max = " + max);
    }
}
```

## Consumer接口

`java.util.function.Consumer<T>` 接口则正好相反，它不是生产一个数据，而是**消费**一个数据，其数据类型由泛型参数决定。

```
@FunctionalInterface
public interface Consumer<T> {
    public abstract void accept(T t);
}
```

### 使用Lambda表达式将一个字符串转成大写和小写的字符串

Consumer消费型接口，可以拿到accept方法参数传递过来的数据进行处理，有参无返回的接口。基本使用如：

```
import java.util.function.Consumer;

public class Demo06Consumer {
    public static void main(String[] args) {
        // Lambda表达式
        test((String s) -> {
            System.out.println(s.toLowerCase());
        });
    }

    public static void test(Consumer<String> consumer) {
        consumer.accept("HelloWorld");
    }
}
```

### 默认方法：andThen

如果一个方法的参数和返回值全都是 `Consumer` 类型，那么就可以实现效果：消费一个数据的时候，首先做一个操作，然后再做一个操作，实现组合。而这个方法就是 `Consumer` 接口中的default方法 `andThen`。下面是JDK的源代码：

```
default Consumer<T> andThen(Consumer<? super T> after) {  
    Objects.requireNonNull(after);  
    return (T t) -> { accept(t); after.accept(t); };  
}
```

备注：`java.util.Objects` 的 `requireNonNull` 静态方法将会在参数为null时主动抛出 `NullPointerException` 异常。这省去了重复编写if语句和抛出空指针异常的麻烦。

要想实现组合，需要两个或多个Lambda表达式即可，而 `andThen` 的语义正是“一步接一步”操作。例如两个步骤组合的情况：

```
public class Demo07ConsumerAndThen {  
    public static void main(String[] args) {  
        // Lambda表达式  
        test((String s) -> {  
            System.out.println(s.toLowerCase());  
        }, (String s) -> {  
            System.out.println(s.toUpperCase());  
        });  
  
        // Lambda表达式简写  
        test(s -> System.out.println(s.toLowerCase()), s ->  
            System.out.println(s.toUpperCase()));  
    }  
  
    public static void test(Consumer<String> c1, Consumer<String> c2) {  
        String str = "Hello World";  
        // c1.accept(str); // 转小写  
        // c2.accept(str); // 转大写  
  
        // c1.andThen(c2).accept(str);  
        c2.andThen(c1).accept(str);  
    }  
}
```

运行结果将会首先打印完全大写的HELLO，然后打印完全小写的hello。当然，通过链式写法可以实现更多步骤的组合。

## Function接口

`java.util.function.Function<T,R>` 接口用来根据一个类型的数据得到另一个类型的数据，前者称为前置条件，后者称为后置条件。有参数有返回值。

```
@FunctionalInterface  
public interface Function<T, R> {  
    public abstract R apply(T t);  
}
```

### 使用Lambda表达式将字符串转成数字

Function转换型接口，对apply方法传入的T类型数据进行处理，返回R类型的结果，有参有返回的接口。使用的场景例如：将 String 类型转换为 Integer 类型。

```
public class Demo08Function {
    public static void main(String[] args) {
        // Lambda表达式
        test((String s) -> {
            return Integer.parseInt(s); // 10
        });
    }

    public static void test(Function<String, Integer> function) {
        Integer in = function.apply("10");
        System.out.println("in: " + (in + 5));
    }
}
```

### 默认方法：andThen

Function 接口中有一个默认的 andThen 方法，用来进行组合操作。JDK源代码如：

```
default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
    Objects.requireNonNull(after);
    return (T t) -> after.apply(apply(t));
}
```

该方法同样用于“先做什么，再做什么”的场景，和 Consumer 中的 andThen 差不多：

```
public class Demo09FunctionAndThen {
    public static void main(String[] args) {
        // Lambda表达式
        test((String s) -> {
            return Integer.parseInt(s);
        }, (Integer i) -> {
            return i * 10;
        });
    }

    public static void test(Function<String, Integer> f1, Function<Integer, Integer> f2) {
        // Integer in = f1.apply("66"); // 将字符串解析成为int数字
        // Integer in2 = f2.apply(in); // 将上一步的int数字乘以10
        Integer in3 = f1.andThen(f2).apply("66");

        System.out.println("in3: " + in3); // 660
    }
}
```

第一个操作是将字符串解析成为int数字，第二个操作是乘以10。两个操作通过 andThen 按照前后顺序组合到了一起。

请注意，Function的前置条件泛型和后置条件泛型可以相同。

## Predicate接口

有时候我们需要对某种类型的数据进行判断，从而得到一个boolean值结果。这时可以使用 `java.util.function.Predicate<T>` 接口。

```
@FunctionalInterface
public interface Predicate<T> {
    public abstract boolean test(T t);
}
Predicate接口用于做判断,返回boolean类型的值
```

### 使用Lambda判断一个人名如果超过3个字就认为是很长的名字

对test方法的参数T进行判断，返回boolean类型的结果。用于条件判断的场景：

```
public class Demo10Predicate {
    public static void main(String[] args) {
        test(s -> s.length() > 3, "迪丽热巴");
    }

    private static void test(Predicate<String> predicate, String str) {
        boolean veryLong = predicate.test(str);
        System.out.println("名字很长吗: " + veryLong);
    }
}
```

条件判断的标准是传入的Lambda表达式逻辑，只要名称长度大于3则认为很长。

### 默认方法：and

既然是条件判断，就会存在与、或、非三种常见的逻辑关系。其中将两个 Predicate 条件使用“与”逻辑连接起来实现“并且”的效果时，可以使用default方法 `and`。其JDK源码为：

```
default Predicate<T> and(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) && other.test(t);
}
```

### 使用Lambda表达式判断一个字符串中即包含W,也包含H

### 使用Lambda表达式判断一个字符串中包含W或者包含H

### 使用Lambda表达式判断一个字符串中即不包含W

如果要判断一个字符串既要包含大写“H”，又要包含大写“W”，那么：

```
public class Demo10Predicate_And_Or_Negate {
    public static void main(String[] args) {
        // Lambda表达式
        test((String s) -> {
            return s.contains("H");
        }, (String s) -> {
```



```
        return s.contains("w");
    });
}

public static void test(Predicate<String> p1, Predicate<String> p2) {
    String str = "HelloWorld";
    boolean b1 = p1.test(str); // 判断包含大写“H”
    boolean b2 = p2.test(str); // 判断包含大写“W”
    //    if (b1 && b2) {
    //        System.out.println("即包含w,也包含H");
    //    }

    boolean bb = p1.and(p2).test(str);
    if (bb) {
        System.out.println("即包含w,也包含H");
    }
}
}
```

### 默认方法：or

#### 使用Lambda表达式判断一个字符串中包含W或者包含H

与 and 的“与”类似，默认方法 or 实现逻辑关系中的“或”。JDK源码为：

```
default Predicate<T> or(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) || other.test(t);
}
```

如果希望实现逻辑“字符串包含大写H或者包含大写W”，那么代码只需要将“and”修改为“or”名称即可，其他都不变：

```
public class Demo10Predicate_And_Or_Negate {
    public static void main(String[] args) {
        // Lambda表达式
        test((String s) -> {
            return s.contains("H");
        }, (String s) -> {
            return s.contains("w");
        });
    }

    public static void test(Predicate<String> p1, Predicate<String> p2) {
        String str = "HelloWorld";
        boolean b1 = p1.test(str); // 判断包含大写“H”
        boolean b2 = p2.test(str); // 判断包含大写“W”

        //    if (b1 || b2) {
        //        System.out.println("有H,或者w");
        //    }
        boolean bbb = p1.or(p2).test(str);
        if (bbb) {
            System.out.println("有H,或者w");
        }
    }
}
```



```
}  
}  
}
```

## 默认方法：negate

### 使用Lambda表达式判断一个字符串中即不包含W

“与”、“或”已经了解了，剩下的“非”（取反）也会简单。默认方法 negate 的JDK源代码为：

```
default Predicate<T> negate() {  
    return (t) -> !test(t);  
}
```

从实现中很容易看出，它是执行了test方法之后，对结果boolean值进行“!”取反而已。一定要在 test 方法调用之前调用 negate 方法，正如 and 和 or 方法一样：

```
import java.util.function.Predicate;  
  
public class Demo10Predicate_And_Or_Negate {  
    public static void main(String[] args) {  
        // Lambda表达式  
        test((String s) -> {  
            return s.contains("H");  
        }, (String s) -> {  
            return s.contains("W");  
        });  
    }  
  
    public static void test(Predicate<String> p1, Predicate<String> p2) {  
        String str = "HelloWorld";  
        boolean b1 = p1.test(str); // 判断包含大写“H”  
        boolean b2 = p2.test(str); // 判断包含大写“W”  
  
        // 没有H,就打印  
        // if (!b1) {  
        //     System.out.println("没有H");  
        // }  
        boolean test = p1.negate().test(str);  
        if (test) {  
            System.out.println("没有H");  
        }  
    }  
}
```

## 小结

### 1. Supplier接口

```
@FunctionalInterface
public interface Supplier<T> {
    public abstract T get();
}
```

## 2. Consumer接口

```
@FunctionalInterface
public interface Consumer<T> {
    public abstract void accept(T t);
}
```

## 3. Function接口

```
@FunctionalInterface
public interface Function<T, R> {
    public abstract R apply(T t);
}
```

## 4. Predicate接口

```
@FunctionalInterface
public interface Predicate<T> {
    public abstract boolean test(T t);
}
```

Predicate接口用于做判断,返回boolean类型的值

# 介绍方法引用

## 目标

了解Lambda的冗余场景

掌握方法引用的格式

了解常见的方法引用方式

## Lambda的冗余场景

使用Lambda表达式求一个数组的和

```
public class Demo11MethodRefIntro {
    public static void getMax(int[] arr) {
        int sum = 0;
        for (int n : arr) {
```

```
        sum += n;
    }
    System.out.println(sum);
}

public static void main(String[] args) {
    printMax((int[] arr) -> {
        int sum = 0;
        for (int n : arr) {
            sum += n;
        }
        System.out.println(sum);
    });
}

private static void printMax(Consumer<int[]> consumer) {
    int[] arr = {10, 20, 30, 40, 50};
    consumer.accept(arr);
}
}
```

## 体验方法引用简化Lambda

如果我们在Lambda中所指定的功能，已经有其他方法存在相同方案，那是否还有必要再写重复逻辑？可以直接“引用”过去就好了：

```
public class DemoPrintRef {
    public static void getMax(int[] arr) {
        int sum = 0;
        for (int n : arr) {
            sum += n;
        }
        System.out.println(sum);
    }

    public static void main(String[] args) {
        printMax(Demo11MethodRefIntro::getMax);
    }

    private static void printMax(Consumer<int[]> consumer) {
        int[] arr = {10, 20, 30, 40, 50};
        consumer.accept(arr);
    }
}
```

请注意其中的双冒号 `::` 写法，这被称为“**方法引用**”，是一种新的语法。

## 方法引用的格式

符号表示：`::`

**符号说明：**双冒号为方法引用运算符，而它所在的表达式被称为**方法引用**。

**应用场景：**如果Lambda所要实现的方案，已经有其他方法存在相同方案，那么则可以使用方法引用。

## 常见引用方式

方法引用在JDK 8中使用方式相当灵活，有以下几种形式：

1. `instanceName::methodName` 对象::方法名
2. `ClassName::staticMethodName` 类名::静态方法
3. `ClassName::methodName` 类名::普通方法
4. `ClassName::new` 类名::new 调用的构造器
5. `TypeName[]::new` `String[]::new` 调用数组的构造器

## 小结

首先了解Lambda表达式的冗余情况,体验了方法引用,了解常见的方法引用方式

## 对象名::引用成员方法

这是最常见的一种用法，与上例相同。如果一个类中已经存在了一个成员方法，则可以通过对象名引用成员方法，代码为：

```
// 对象::实例方法
@Test
public void test01() {
    Date now = new Date();
    Supplier<Long> supp = () -> {
        return now.getTime();
    };

    System.out.println(supp.get());

    Supplier<Long> supp2 = now::getTime;
    System.out.println(supp2.get());
}
```

方法引用的注意事项

1. 被引用的方法，参数要和接口中抽象方法的参数一样
2. 当接口抽象方法有返回值时，被引用的方法也必须有返回值

## 类名::引用静态方法

由于在 `java.lang.System` 类中已经存在了静态方法 `currentTimeMillis`，所以当我们需要通过Lambda来调用该方法时，可以使用方法引用，写法是：

```
// 类名::静态方法
@Test
public void test02() {
    Supplier<Long> supp = () -> {
        return System.currentTimeMillis();
    };
    System.out.println(supp.get());

    Supplier<Long> supp2 = System::currentTimeMillis;
    System.out.println(supp2.get());
}
```

## 类名::引用实例方法

Java面向对象中，类名只能调用静态方法，类名引用实例方法是有前提的，实际上是拿第一个参数作为方法的调用者。

```
// 类名::实例方法
@Test
public void test03() {
    Function<String, Integer> f1 = (s) -> {
        return s.length();
    };
    System.out.println(f1.apply("abc"));

    Function<String, Integer> f2 = String::length;
    System.out.println(f2.apply("abc"));

    BiFunction<String, Integer, String> bif = String::substring;
    String hello = bif.apply("hello", 2);
    System.out.println("hello = " + hello);
}
```

## 类名::new引用构造器

由于构造器的名称与类名完全一样。所以构造器引用使用 类名称::new 的格式表示。首先是一个简单的 Person 类：

```
public class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

要使用这个函数式接口，可以通过方法引用传递：

```
// 类名::new
@Test
public void test04() {
    Supplier<Person> sup = () -> {
        return new Person();
    };
    System.out.println(sup.get());

    Supplier<Person> sup2 = Person::new;
    System.out.println(sup2.get());

    BiFunction<String, Integer, Person> fun2 = Person::new;
    System.out.println(fun2.apply("张三", 18));
}
```

## 数组::new 引用数组构造器

数组也是 `Object` 的子类对象，所以同样具有构造器，只是语法稍有不同。

```
// 类型[]::new
@Test
public void test05() {
    Function<Integer, String[]> fun = (len) -> {
        return new String[len];
    };
    String[] arr1 = fun.apply(10);
    System.out.println(arr1 + ", " + arr1.length);

    Function<Integer, String[]> fun2 = String[]::new;
    String[] arr2 = fun2.apply(5);
    System.out.println(arr2 + ", " + arr2.length);
}
```

## 小结

方法引用是对Lambda表达式符合特定情况下的一种缩写，它使得我们的Lambda表达式更加的精简，也可以理解为Lambda表达式的缩写形式，不过要注意的是方法引用只能"引用"已经存在的方法！

## Stream流介绍

### 目标

了解集合的处理数据的弊端

理解Stream流思想和作用

## 集合处理数据的弊端

当我们需要对集合中的元素进行操作的时候，除了必需的添加、删除、获取外，最典型的的就是集合遍历。我们来体验集合操作数据的弊端，需求如下：

一个ArrayList集合中存储有以下数据：张无忌，周芷若，赵敏，张强，张三丰  
需求：1.拿到所有姓张的 2.拿到名字长度为3个字的 3.打印这些数据

代码如下：

```
public static void main(String[] args) {  
    // 一个ArrayList集合中存储有以下数据：张无忌，周芷若，赵敏，张强，张三丰  
    // 需求：1.拿到所有姓张的 2.拿到名字长度为3个字的 3.打印这些数据  
    ArrayList<String> list = new ArrayList<>();  
    Collections.addAll(list, "张无忌", "周芷若", "赵敏", "张强", "张三丰");  
  
    // 1.拿到所有姓张的  
    ArrayList<String> zhangList = new ArrayList<>(); // {"张无忌", "张强", "张三丰"}  
    for (String name : list) {  
        if (name.startsWith("张")) {  
            zhangList.add(name);  
        }  
    }  
  
    // 2.拿到名字长度为3个字的  
    ArrayList<String> threeList = new ArrayList<>(); // {"张无忌", "张三丰"}  
    for (String name : zhangList) {  
        if (name.length() == 3) {  
            threeList.add(name);  
        }  
    }  
  
    // 3.打印这些数据  
    for (String name : threeList) {  
        System.out.println(name);  
    }  
}
```

### 循环遍历的弊端

这段代码中含有三个循环，每一个作用不同：

1. 首先筛选所有姓张的人；
2. 然后筛选名字有三个字的人；
3. 最后进行对结果进行打印输出。

每当我们需要对集合中的元素进行操作的时候，总是需要进行循环、循环、再循环。这是理所当然的么？**不是**。循环是做事情的方式，而不是目的。每个需求都要循环一次，还要搞一个新集合来装数据，如果希望再次遍历，只能再使用另一个循环从头开始。

那Stream能给我们带来怎样更加优雅的写法呢？

### Stream的更优写法

下面来看一下借助Java 8的Stream API，修改后的代码：

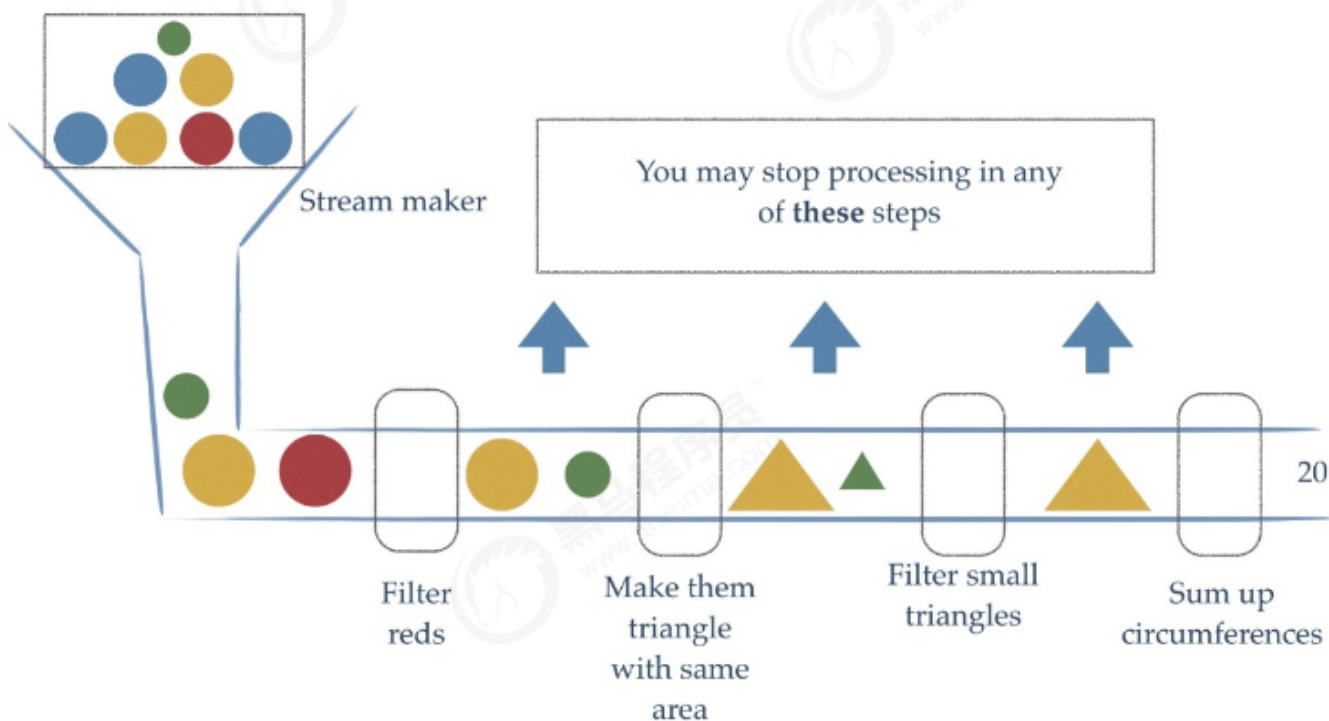
```
public class Demo03StreamFilter {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<>();  
        list.add("张无忌");  
        list.add("周芷若");  
        list.add("赵敏");  
        list.add("张强");  
        list.add("张三丰");  
  
        list.stream()  
            .filter(s -> s.startsWith("张"))  
            .filter(s -> s.length() == 3)  
            .forEach(System.out::println);  
    }  
}
```

直接阅读代码的字面意思即可完美展示无关逻辑方式的语义：**获取流、过滤姓张、过滤长度为3、逐一打印**。我们真正要做的事情内容被更好地体现在代码中。

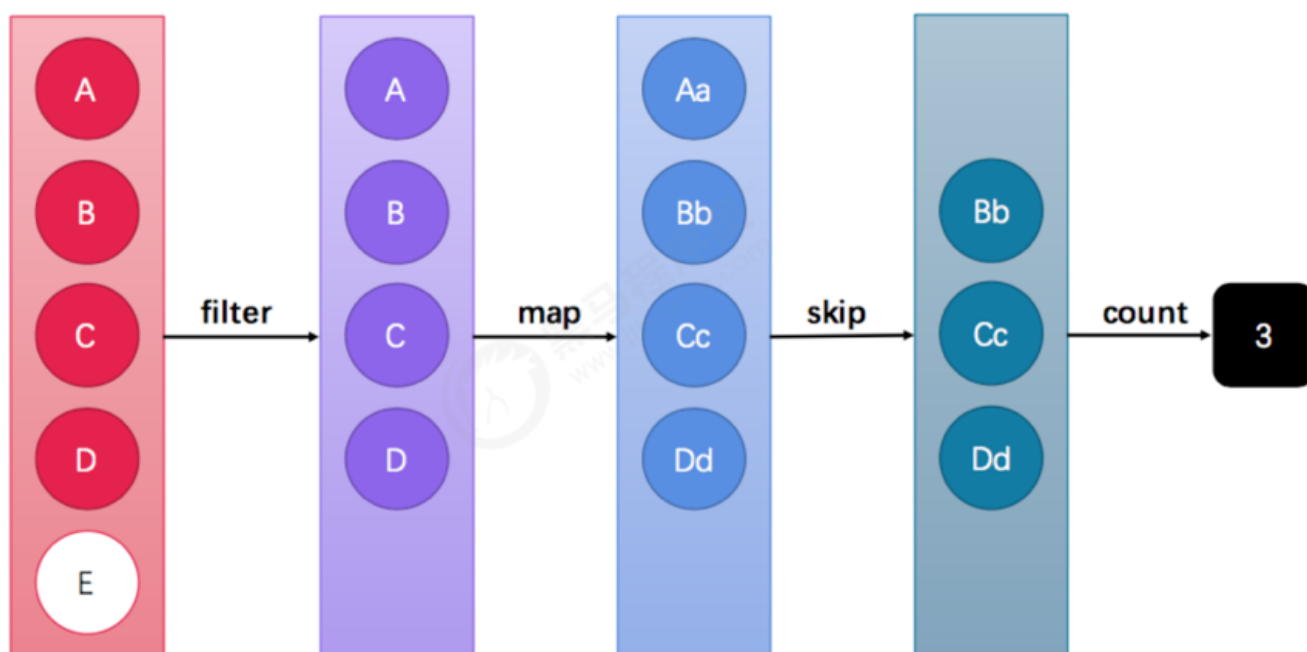
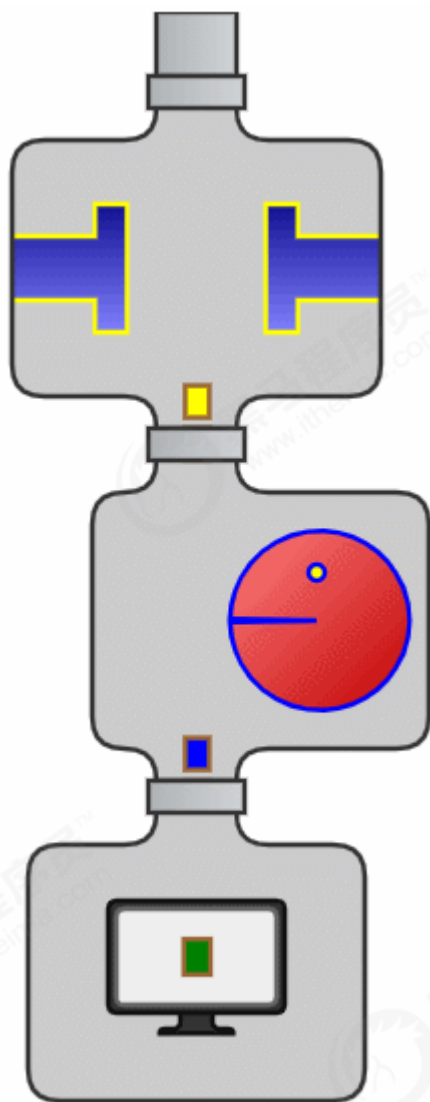
## Stream流式思想概述

**注意：Stream和IO流(InputStream/OutputStream)没有任何关系，请暂时忘记对传统IO流的固有印象！**

Stream流式思想类似于工厂车间的“**生产流水线**”，Stream流不是一种数据结构，不保存数据，而是对数据进行加工处理。Stream可以看作是流水线上的一个工序。在流水线上，通过多个工序让一个原材料加工成一个商品。







Stream API能让我们快速完成许多复杂的操作，如筛选、切片、映射、查找、去除重复，统计，匹配和归约。

## 小结

首先我们了解了集合操作数据的弊端,每次都需要循环遍历,还要创建新集合,很麻烦

Stream是流式思想,相当于工厂的流水线,对集合中的数据进行加工处理

## 获取Stream流的两种方式

### 目标

掌握根据Collection获取流

掌握Stream中的静态方法of获取流

`java.util.stream.Stream<T>` 是JDK 8新加入的流接口。

获取一个流非常简单，有以下几种常用的方式：

- 所有的 `Collection` 集合都可以通过 `stream` 默认方法获取流；
- `Stream` 接口的静态方法 `of` 可以获取数组对应的流。

#### 方式1：根据Collection获取流

首先，`java.util.Collection` 接口中加入了default方法 `stream` 用来获取流，所以其所有实现类均可获取流。

```
public interface Collection {  
    default Stream<E> stream()  
}
```

```
import java.util.*;  
import java.util.stream.Stream;  
  
public class Demo04GetStream {  
    public static void main(String[] args) {  
        // 集合获取流  
        // Collection接口中的方法: default Stream<E> stream() 获取流  
        List<String> list = new ArrayList<>();  
        // ...  
        Stream<String> stream1 = list.stream();  
  
        Set<String> set = new HashSet<>();  
        // ...  
        Stream<String> stream2 = set.stream();  
  
        Vector<String> vector = new Vector<>();  
        // ...  
        Stream<String> stream3 = vector.stream();  
    }  
}
```

java.util.Map 接口不是 Collection 的子接口，所以获取对应的流需要分key、value或entry等情况：

```
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Stream;

public class Demo05GetStream {
    public static void main(String[] args) {
        // Map获取流
        Map<String, String> map = new HashMap<>();
        // ...
        Stream<String> keyStream = map.keySet().stream();
        Stream<String> valueStream = map.values().stream();
        Stream<Map.Entry<String, String>> entryStream = map.entrySet().stream();
    }
}
```

## 方式2：Stream中的静态方法of获取流

由于数组对象不可能添加默认方法，所以 Stream 接口中提供了静态方法 of，使用很简单：

```
import java.util.stream.Stream;

public class Demo06GetStream {
    public static void main(String[] args) {
        // Stream中的静态方法：static Stream of(T... values)
        Stream<String> stream6 = Stream.of("aa", "bb", "cc");

        String[] arr = {"aa", "bb", "cc"};
        Stream<String> stream7 = Stream.of(arr);

        Integer[] arr2 = {11, 22, 33};
        Stream<Integer> stream8 = Stream.of(arr2);

        // 注意：基本数据类型的数组不行
        int[] arr3 = {11, 22, 33};
        Stream<int[]> stream9 = Stream.of(arr3);
    }
}
```

备注：of 方法的参数其实是一个可变参数，所以支持数组。

## 小结

学习了两种获取流的方式：

1. 通过Collection接口中的默认方法Stream stream()
2. 通过Stream接口中的静态of方法

## Stream常用方法和注意事项

## 目标

了解Stream常用方法的分类

掌握Stream注意事项

## Stream常用方法

Stream流模型的操作很丰富，这里介绍一些常用的API。这些方法可以被分成两种：

方法名	方法作用	返回值类型	方法种类
count	统计个数	long	终结
forEach	逐一处理	void	终结
filter	过滤	Stream	函数拼接
limit	取用前几个	Stream	函数拼接
skip	跳过前几个	Stream	函数拼接
map	映射	Stream	函数拼接
concat	组合	Stream	函数拼接

- **终结方法**：返回值类型不再是 `Stream` 类型的方法，不再支持链式调用。本小节中，终结方法包括 `count` 和 `forEach` 方法。
- **非终结方法**：返回值类型仍然是 `Stream` 类型的方法，支持链式调用。（除了终结方法外，其余方法均为非终结方法。）

备注：本小节之外的更多方法，请自行参考API文档。

## Stream注意事项(重要)

1. Stream只能操作一次
2. Stream方法返回的是新的流
3. Stream不调用终结方法，中间的操作不会执行

## 小结

我们学习了Stream的常用方法,我们知道Stream这些常用方法可以分成两类,终结方法,函数拼接方法

Stream的3个注意事项:

1. Stream只能操作一次
2. Stream方法返回的是新的流
3. Stream不调用终结方法，中间的操作不会执行

## Stream流的forEach方法

forEach 用来遍历流中的数据

```
void forEach(Consumer<? super T> action);
```

该方法接收一个 `Consumer` 接口函数，会将每一个流元素交给该函数进行处理。例如：

```
@Test
public void testForEach() {
    List<String> one = new ArrayList<>();
    Collections.addAll(one, "迪丽热巴", "宋远桥", "苏星河", "老子", "庄子", "孙子");

    /*one.stream().forEach((String s) -> {
        System.out.println(s);
    });*/

    // 简写
    // one.stream().forEach(s -> System.out.println(s));

    one.stream().forEach(System.out::println);
}
```

## Stream流的count方法

Stream流提供 `count` 方法来统计其中的元素个数：

```
long count();
```

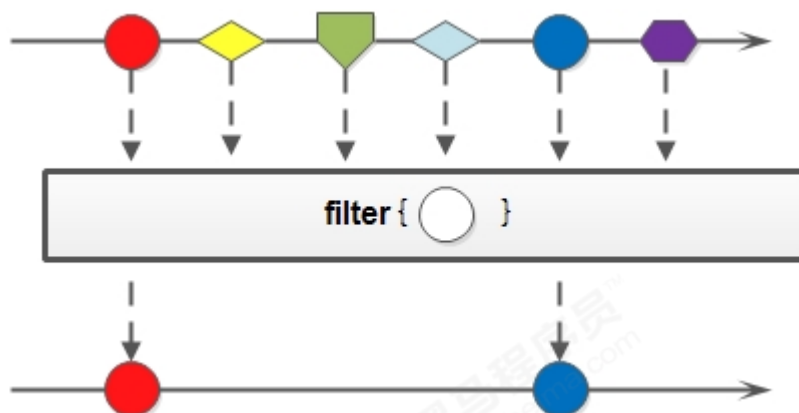
该方法返回一个long值代表元素个数。基本使用：

```
@Test
public void testCount() {
    List<String> one = new ArrayList<>();
    Collections.addAll(one, "迪丽热巴", "宋远桥", "苏星河", "老子", "庄子", "孙子");

    System.out.println(one.stream().count());
}
```

## Stream流的filter方法

`filter`用于过滤数据，返回符合过滤条件的数据



可以通过 `filter` 方法将一个流转换成另一个子集流。方法声明：

```
Stream<T> filter(Predicate<? super T> predicate);
```

该接口接收一个 `Predicate` 函数式接口参数（可以是一个Lambda或方法引用）作为筛选条件。

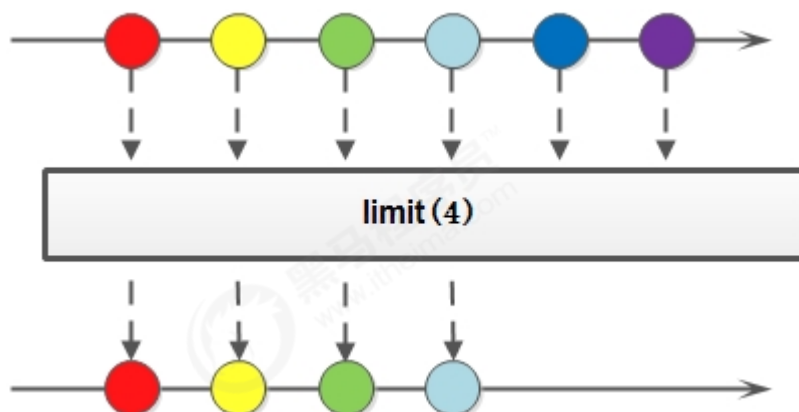
Stream流中的 `filter` 方法基本使用的代码如下：

```
@Test
public void testFilter() {
    List<String> one = new ArrayList<>();
    Collections.addAll(one, "迪丽热巴", "宋远桥", "苏星河", "老子", "庄子", "孙子");

    one.stream().filter(s -> s.length() == 2).forEach(System.out::println);
}
```

在这里通过Lambda表达式来指定了筛选的条件：姓名长度为2个字。

## Stream流的limit方法



`limit` 方法可以对流进行截取，只取用前n个。方法签名：

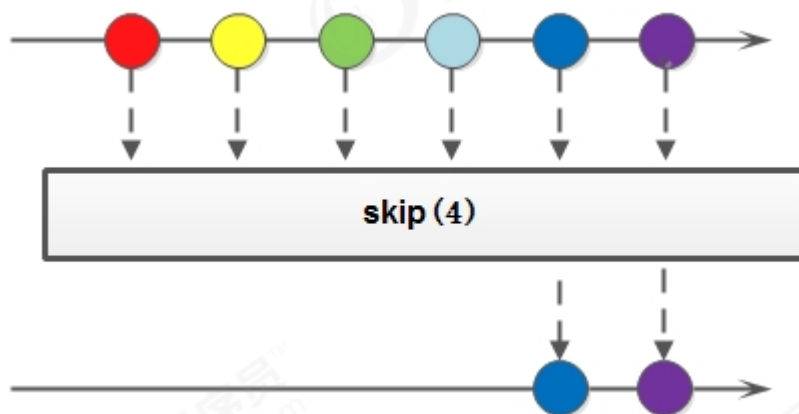
```
Stream<T> limit(long maxSize);
```

参数是一个long型，如果集合当前长度大于参数则进行截取。否则不进行操作。基本使用：

```
@Test
public void testLimit() {
    List<String> one = new ArrayList<>();
    Collections.addAll(one, "迪丽热巴", "宋远桥", "苏星河", "老子", "庄子", "孙子");

    one.stream().limit(3).forEach(System.out::println);
}
```

## Stream流的skip方法



如果希望跳过前几个元素，可以使用 `skip` 方法获取一个截取之后的新流：

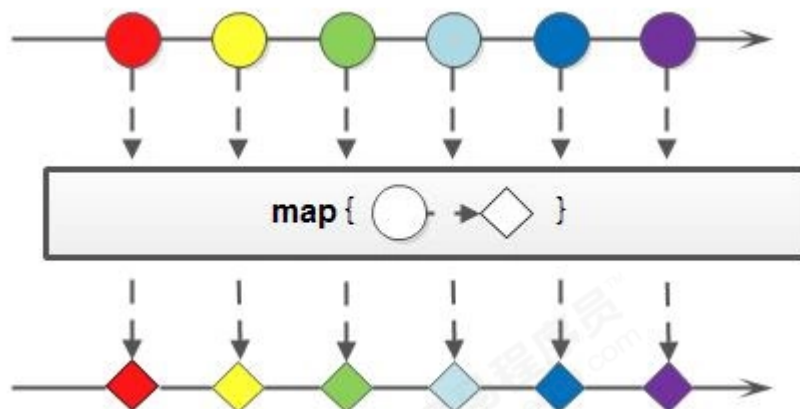
```
Stream<T> skip(long n);
```

如果流的当前长度大于`n`，则跳过前`n`个；否则将会得到一个长度为0的空流。基本使用：

```
@Test
public void testSkip() {
    List<String> one = new ArrayList<>();
    Collections.addAll(one, "迪丽热巴", "宋远桥", "苏星河", "老子", "庄子", "孙子");

    one.stream().skip(2).forEach(System.out::println);
}
```

## Stream流的map方法



如果需要将流中的元素映射到另一个流中，可以使用 `map` 方法。方法签名：

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

该接口需要一个 `Function` 函数式接口参数，可以将当前流中的T类型数据转换为另一种R类型的流。

Stream流中的 `map` 方法基本使用的代码如：

```
@Test
public void testMap() {
    Stream<String> original = Stream.of("11", "22", "33");
    Stream<Integer> result = original.map(Integer::parseInt);
    result.forEach(s -> System.out.println(s + 10));
}
```

这段代码中，`map` 方法的参数通过方法引用，将字符串类型转换成为了int类型（并自动装箱为 `Integer` 类对象）。

## Stream流的sorted方法

如果需要将数据排序，可以使用 `sorted` 方法。方法签名：

```
Stream<T> sorted();
Stream<T> sorted(Comparator<? super T> comparator);
```

### 基本使用

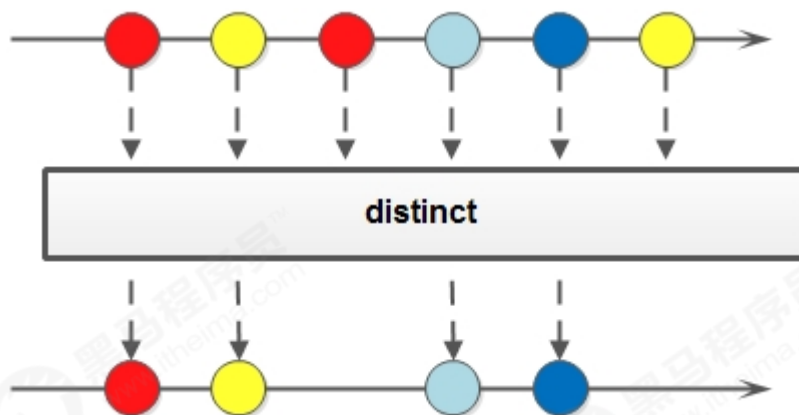
Stream流中的 `sorted` 方法基本使用的代码如：



```
@Test
public void testSorted() {
    // sorted(): 根据元素的自然顺序排序
    // sorted(Comparator<? super T> comparator): 根据比较器指定的规则排序
    Stream.of(33, 22, 11, 55)
        .sorted()
        .sorted((o1, o2) -> o2 - o1)
        .forEach(System.out::println);
}
```

这段代码中，`sorted` 方法根据元素的自然顺序排序，也可以指定比较器排序。

## Stream流的distinct方法



如果需要去除重复数据，可以使用 `distinct` 方法。方法签名：

```
Stream<T> distinct();
```

### 基本使用

Stream流中的 `distinct` 方法基本使用的代码如：

```
@Test
public void testDistinct() {
    Stream.of(22, 33, 22, 11, 33)
        .distinct()
        .forEach(System.out::println);
}
```

如果是自定义类型如何是否也能去除重复的数据呢？

```
@Test
public void testDistinct2() {
    Stream.of(
        new Person("刘德华", 58),
        new Person("张学友", 56),
        new Person("张学友", 56),
        new Person("黎明", 52))
        .distinct()
        .forEach(System.out::println);
}
```

```
public class Person {
    private String name;
    private int age;
    // 省略其他
}
```

自定义类型是根据对象的hashCode和equals来去除重复元素的。

## Stream流的match方法

如果需要判断数据是否匹配指定的条件，可以使用 Match 相关方法。方法签名：

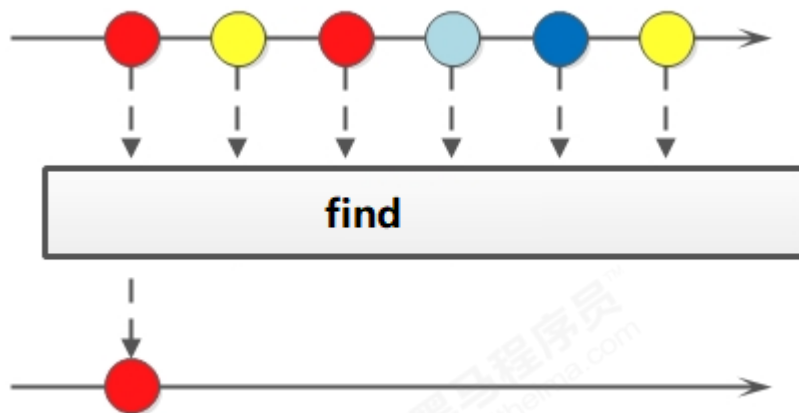
```
boolean allMatch(Predicate<? super T> predicate);
boolean anyMatch(Predicate<? super T> predicate);
boolean noneMatch(Predicate<? super T> predicate);
```

### 基本使用

Stream流中的 Match 相关方法基本使用的代码如：

```
@Test
public void testMatch() {
    boolean b = Stream.of(5, 3, 6, 1)
        // .allMatch(e -> e > 0); // allMatch: 元素是否全部满足条件
        // .anyMatch(e -> e > 5); // anyMatch: 元素是否任意有一个满足条件
        .noneMatch(e -> e < 0); // noneMatch: 元素是否全部不满足条件
    System.out.println("b = " + b);
}
```

## Stream流的find方法



如果需要找到某些数据，可以使用 `find` 相关方法。方法签名：

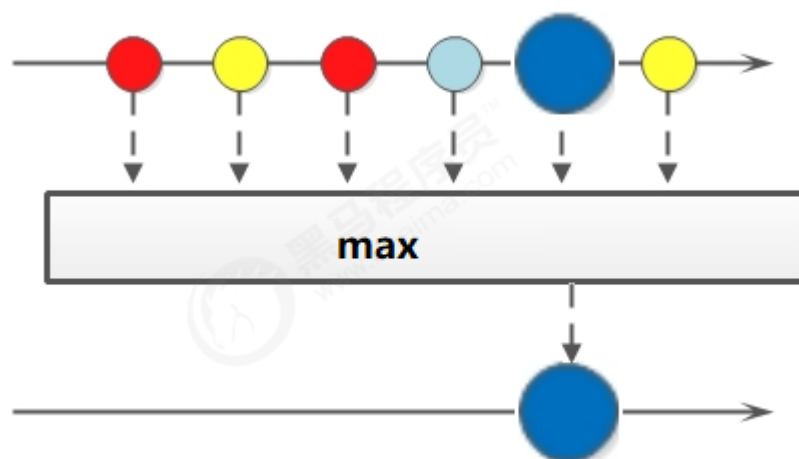
```
Optional<T> findFirst();  
Optional<T> findAny();
```

## 基本使用

Stream流中的 `find` 相关方法基本使用的代码如：

```
@Test  
public void testFind() {  
    Optional<Integer> first = Stream.of(5, 3, 6, 1).findFirst();  
    System.out.println("first = " + first.get());  
  
    Optional<Integer> any = Stream.of(5, 3, 6, 1).findAny();  
    System.out.println("any = " + any.get());  
}
```

## Stream流的max和min方法



如果需要获取最大和最小值，可以使用 `max` 和 `min` 方法。方法签名：

```
Optional<T> max(Comparator<? super T> comparator);  
Optional<T> min(Comparator<? super T> comparator);
```

## 基本使用

Stream流中的 max 和 min 相关方法基本使用的代码如：

```
@Test  
public void testMax_Min() {  
    Optional<Integer> max = Stream.of(5, 3, 6, 1).max((o1, o2) -> o1 - o2);  
    System.out.println("first = " + max.get());  
    Optional<Integer> min = Stream.of(5, 3, 6, 1).min((o1, o2) -> o1 - o2);  
    System.out.println("any = " + min.get());  
}
```

## Stream流的reduce方法



如果需要将所有数据归纳得到一个数据，可以使用 reduce 方法。方法签名：

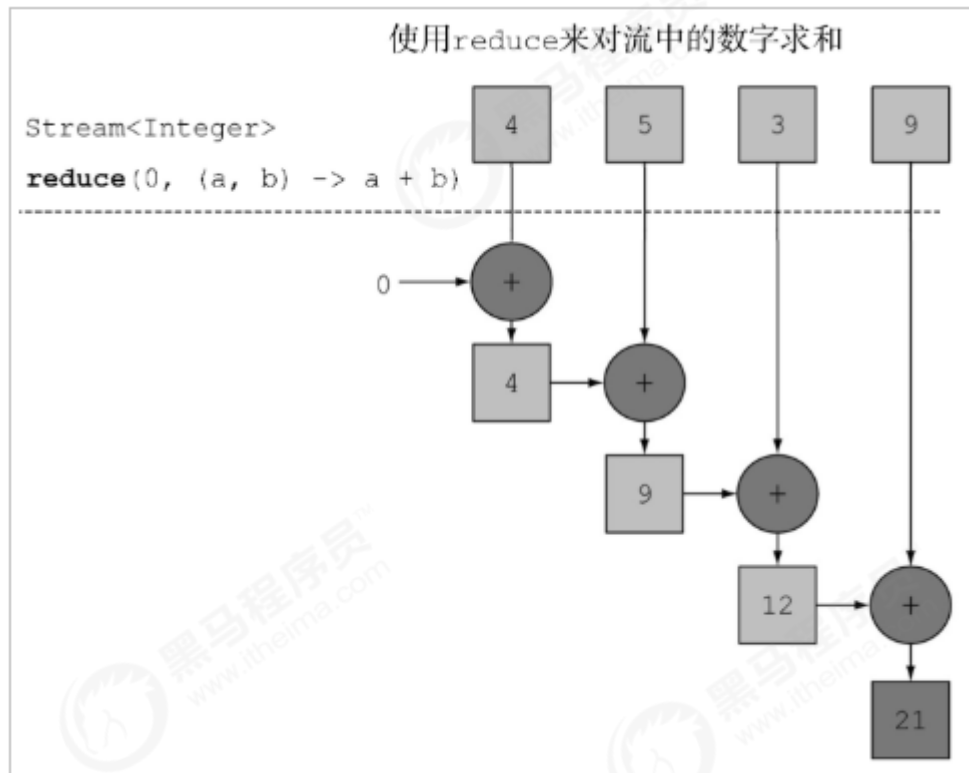
```
T reduce(T identity, BinaryOperator<T> accumulator);
```

## 基本使用

Stream流中的 reduce 相关方法基本使用的代码如：

```
@Test  
public void testReduce() {  
    int reduce = Stream.of(4, 5, 3, 9)  
        .reduce(0, (a, b) -> {  
            System.out.println("a = " + a + ", b = " + b);  
            return a + b;  
        });  
    // reduce:  
    // 第一次将默认做赋值给x，取出第一个元素赋值给y，进行操作  
    // 第二次，将第一次的结果赋值给x，取出二个元素赋值给y，进行操作  
    // 第三次，将第二次的结果赋值给x，取出三个元素赋值给y，进行操作  
    // 第四次，将第三次的结果赋值给x，取出四个元素赋值给y，进行操作  
    System.out.println("reduce = " + reduce);  
  
    int reduce2 = Stream.of(4, 5, 3, 9)  
        .reduce(0, (x, y) -> {  
            return Integer.sum(x, y);  
        });  
  
    int reduce3 = Stream.of(4, 5, 3, 9).reduce(0, Integer::sum);
```

```
int max = Stream.of(4, 5, 3, 9)
    .reduce(0, (x, y) -> {
        return x > y ? x : y;
    });
System.out.println("max = " + max);
}
```



## Stream流的map和reduce组合使用

```
@Test
public void testMapReduce() {
    // 求出所有年龄的总和
    int totalAge = Stream.of(
        new Person("刘德华", 58),
        new Person("张学友", 56),
        new Person("郭富城", 54),
        new Person("黎明", 52))
        .map((p) -> p.getAge())
        .reduce(0, (x, y) -> x + y);
    System.out.println("totalAge = " + totalAge);

    // 找出最大年龄
    int maxAge = Stream.of(
        new Person("刘德华", 58),
        new Person("张学友", 56),
        new Person("郭富城", 54),
```

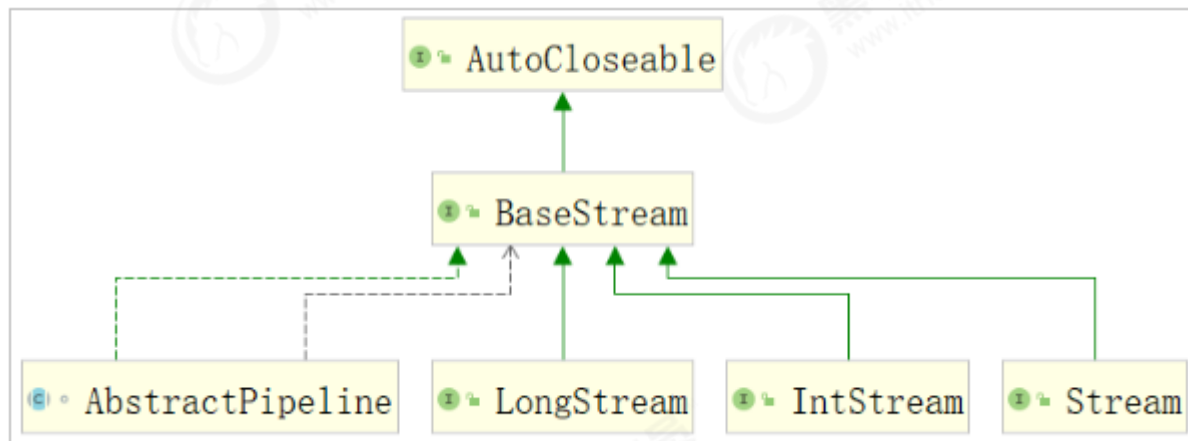
```
new Person("黎明", 52))
    .map((p) -> p.getAge())
    .reduce(0, (x, y) -> x > y ? x : y);
System.out.println("maxAge = " + maxAge);
```

```
// 统计 数字2 出现的次数
int count = Stream.of(1, 2, 2, 1, 3, 2)
    .map(i -> {
        if (i == 2) {
            return 1;
        } else {
            return 0;
        }
    })
    .reduce(0, Integer::sum);
System.out.println("count = " + count);
}
```

## Stream流的mapToInt

如果需要将Stream中的Integer类型数据转成int类型，可以使用 `mapToInt` 方法。方法签名：

```
IntStream mapToInt(ToIntFunction<? super T> mapper);
```



### 基本使用

Stream流中的 `mapToInt` 相关方法基本使用的代码如：

```
@Test
public void test1() {
    // Integer占用的内存比int多,在Stream流操作中会自动装箱和拆箱
    Stream<Integer> stream = Arrays.stream(new Integer[]{1, 2, 3, 4, 5});

    // 把大于3的和打印出来
    // Integer result = stream
    //     .filter(i -> i.intValue() > 3)
```

```
//          .reduce(0, Integer::sum);
// System.out.println(result);

// 先将流中的Integer数据转成int,后续都是操作int类型
IntStream intStream = stream.mapToInt(Integer::intValue);
int reduce = intStream
    .filter(i -> i > 3)
    .reduce(0, Integer::sum);
System.out.println(reduce);

// 将IntStream转化为Stream<Integer>
IntStream intStream1 = IntStream.rangeClosed(1, 10);
Stream<Integer> boxed = intStream1.boxed();
boxed.forEach(s -> System.out.println(s.getClass() + ", " + s));
}
```

## Stream流的concat方法

如果有两个流，希望合并成为一个流，那么可以使用 `Stream` 接口的静态方法 `concat`：

```
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
```

备注：这是一个静态方法，与 `java.lang.String` 当中的 `concat` 方法是不同的。

该方法的基本使用代码如下：

```
@Test
public void testConcat() {
    Stream<String> streamA = Stream.of("张三");
    Stream<String> streamB = Stream.of("李四");
    Stream<String> result = Stream.concat(streamA, streamB);
    result.forEach(System.out::println);
}
```

## Stream综合案例

现在有两个 `ArrayList` 集合存储队伍当中的多个成员姓名，要求使用传统的for循环（或增强for循环）依次进行以下若干操作步骤：

1. 第一个队伍只要名字为3个字的成员姓名；
2. 第一个队伍筛选之后只要前3个人；
3. 第二个队伍只要姓张的成员姓名；
4. 第二个队伍筛选之后不要前2个人；
5. 将两个队伍合并为一个队伍；
6. 根据姓名创建 `Person` 对象；
7. 打印整个队伍的 `Person` 对象信息。



两个队伍（集合）的代码如下：

```
public class DemoArrayListNames {  
    public static void main(String[] args) {  
        List<String> one = List.of("迪丽热巴", "宋远桥", "苏星河", "老子", "庄子", "孙子", "洪七公");  
        List<String> two = List.of("古力娜扎", "张无忌", "张三丰", "赵丽颖", "张二狗", "张天爱", "张三");  
  
        // ....  
    }  
}
```

而 `Person` 类的代码为：

```
public class Person {  
  
    private String name;  
  
    public Person() {}  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public String toString() {  
        return "Person{name='" + name + "'}";  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

## 传统方式

使用for循环，示例代码：

```
public class DemoArrayListNames {  
    public static void main(String[] args) {  
        List<String> one = List.of("迪丽热巴", "宋远桥", "苏星河", "老子", "庄子", "孙子", "洪七公");  
        List<String> two = List.of("古力娜扎", "张无忌", "张三丰", "赵丽颖", "张二狗", "张天爱", "张三");  
  
        // 第一个队伍只要名字为3个字的成员姓名;  
        List<String> oneA = new ArrayList<>();  
    }  
}
```





```
for (String name : one) {
    if (name.length() == 3) {
        oneA.add(name);
    }
}

// 第一个队伍筛选之后只要前3个人;
List<String> oneB = new ArrayList<>();
for (int i = 0; i < 3; i++) {
    oneB.add(oneA.get(i));
}

// 第二个队伍只要姓张的成员姓名;
List<String> twoA = new ArrayList<>();
for (String name : two) {
    if (name.startsWith("张")) {
        twoA.add(name);
    }
}

// 第二个队伍筛选之后不要前2个人;
List<String> twoB = new ArrayList<>();
for (int i = 2; i < twoA.size(); i++) {
    twoB.add(twoA.get(i));
}

// 将两个队伍合并为一个队伍;
List<String> totalNames = new ArrayList<>();
totalNames.addAll(oneB);
totalNames.addAll(twoB);

// 根据姓名创建Person对象;
List<Person> totalPersonList = new ArrayList<>();
for (String name : totalNames) {
    totalPersonList.add(new Person(name));
}

// 打印整个队伍的Person对象信息。
for (Person person : totalPersonList) {
    System.out.println(person);
}
}
```

运行结果为:

```
Person{name='宋远桥'}
Person{name='苏星河'}
Person{name='洪七公'}
Person{name='张二狗'}
Person{name='张天爱'}
Person{name='张三'}
```

## Stream方式

等效的Stream流式处理代码为：

```
public class DemoStreamNames {
    public static void main(String[] args) {
        List<String> one = List.of("迪丽热巴", "宋远桥", "苏星河", "老子", "庄子", "孙子", "洪七公");
        List<String> two = List.of("古力娜扎", "张无忌", "张三丰", "赵丽颖", "张二狗", "张天爱", "张三");

        // 第一个队伍只要名字为3个字的成员姓名;
        // 第一个队伍筛选之后只要前3个人;
        Stream<String> streamOne = one.stream().filter(s -> s.length() == 3).limit(3);

        // 第二个队伍只要姓张的成员姓名;
        // 第二个队伍筛选之后不要前2个人;
        Stream<String> streamTwo = two.stream().filter(s -> s.startsWith("张")).skip(2);

        // 将两个队伍合并为一个队伍;
        // 根据姓名创建Person对象;
        // 打印整个队伍的Person对象信息。
        Stream.concat(streamOne, streamTwo).map(Person::new).forEach(System.out::println);
    }
}
```

运行效果完全一样：

```
Person{name='宋远桥'}
Person{name='苏星河'}
Person{name='洪七公'}
Person{name='张二狗'}
Person{name='张天爱'}
Person{name='张三'}
```

## 收集Stream流中的结果

```
IntStream intStream = Stream.of(1, 2, 3, 4, 5).mapToInt(Integer::intValue);
intStream.filter(n -> n > 3).forEach(System.out::println);
intStream.filter(n -> n > 3).count();
intStream.filter(n -> n > 3).reduce(0, Integer::sum);
```

对流操作完成之后，如果需要将流的结果保存到数组或集合中，可以收集流中的数据

## 目标

掌握Stream流中的结果到集合中

掌握Stream流中的结果到数组中

## Stream流中的结果到集合中

Stream流提供 `collect` 方法，其参数需要一个 `java.util.stream.Collectors<T,A, R>` 接口对象来指定收集到哪种集合中。`java.util.stream.Collectors` 类提供一些方法，可以作为 `Collector` 接口的实例：

- `public static <T> Collector<T, ?, List<T>> toList()`：转换为 `List` 集合。
- `public static <T> Collector<T, ?, Set<T>> toSet()`：转换为 `Set` 集合。

下面是这两个方法的基本使用代码：

```
// 将流中数据收集到集合中
@Test
public void testStreamToCollection() {
    Stream<String> stream = Stream.of("aa", "bb", "cc");
    // List<String> list = stream.collect(Collectors.toList());
    // Set<String> set = stream.collect(Collectors.toSet());

    ArrayList<String> arrayList = stream.collect(Collectors.toCollection(ArrayList::new));
    HashSet<String> hashSet = stream.collect(Collectors.toCollection(HashSet::new));
}
```

## Stream流中的结果到数组中

Stream提供 `toArray` 方法来将结果放到一个数组中，返回值类型是 `Object[]` 的：

```
Object[] toArray();
```

其使用场景如：

```
@Test
public void testStreamToArray() {
    Stream<String> stream = Stream.of("aa", "bb", "cc");

    // Object[] objects = stream.toArray();
    // for (Object obj : objects) {
    //     System.out.println();
    // }

    String[] strings = stream.toArray(String[]::new);
    for (String str : strings) {
        System.out.println(str);
    }
}
```

## 对流中数据进行聚合计算



当我们使用Stream流处理数据后，可以像数据库的聚合函数一样对某个字段进行操作。比如获取最大值，获取最小值，求总和，平均值，统计数量。

```
@Test
public void testStreamToOther() {
    Stream<Student> studentStream = Stream.of(
        new Student("赵丽颖", 58, 95),
        new Student("杨颖", 56, 88),
        new Student("迪丽热巴", 56, 99),
        new Student("柳岩", 52, 77));

    // 获取最大值
    // Optional<Student> collect = studentStream.collect(Collectors.maxBy((o1, o2) ->
    o1.getSocre() - o2.getSocre()));

    // 获取最小值
    // Optional<Student> collect = studentStream.collect(Collectors.minBy((o1, o2) ->
    o1.getSocre() - o2.getSocre()));
    // System.out.println(collect.get());

    // 求总和
    // int sumAge = studentStream.collect(Collectors.summingInt(s -> s.getAge()));
    // System.out.println("sumAge = " + sumAge);

    // 平均值
    // double avgScore = studentStream.collect(Collectors.averagingInt(s -> s.getSocre()));
    // System.out.println("avgScore = " + avgScore);

    // 统计数量
    // Long count = studentStream.collect(Collectors.counting());
    // System.out.println("count = " + count);
}
```

## 对流中数据进行分组

当我们使用Stream流处理数据后，可以根据某个属性将数据分组：

```
// 分组
@Test
public void testGroup() {
    Stream<Student> studentStream = Stream.of(
        new Student("赵丽颖", 52, 95),
        new Student("杨颖", 56, 88),
        new Student("迪丽热巴", 56, 55),
        new Student("柳岩", 52, 33));

    // Map<Integer, List<Student>> map =
    studentStream.collect(Collectors.groupingBy(Student::getAge));

    // 将分数大于60的分为一组,小于60分成另一组
    Map<String, List<Student>> map = studentStream.collect(Collectors.groupingBy((s) ->
```



```
{
    if (s.getSocre() > 60) {
        return "及格";
    } else {
        return "不及格";
    }
});

map.forEach((k, v) -> {
    System.out.println(k + "::" + v);
});
}
```

效果:

不及格::[Student{name='迪丽热巴', age=56, socre=55}, Student{name='柳岩', age=52, socre=33}]  
及格::[Student{name='赵丽颖', age=52, socre=95}, Student{name='杨颖', age=56, socre=88}]

## 对流中数据进行多级分组

还可以对数据进行多级分组:

```
// 多级分组
@Test
public void testCustomGroup() {
    Stream<Student> studentStream = Stream.of(
        new Student("赵丽颖", 52, 95),
        new Student("杨颖", 56, 88),
        new Student("迪丽热巴", 56, 99),
        new Student("柳岩", 52, 77));

    Map<Integer, Map<String, List<Student>>> map =
        studentStream.collect(Collectors.groupingBy(s -> s.getAge(), Collectors.groupingBy(s -> {
            if (s.getSocre() >= 90) {
                return "优秀";
            } else if (s.getSocre() >= 80 && s.getSocre() < 90) {
                return "良好";
            } else if (s.getSocre() >= 80 && s.getSocre() < 80) {
                return "及格";
            } else {
                return "不及格";
            }
        })));

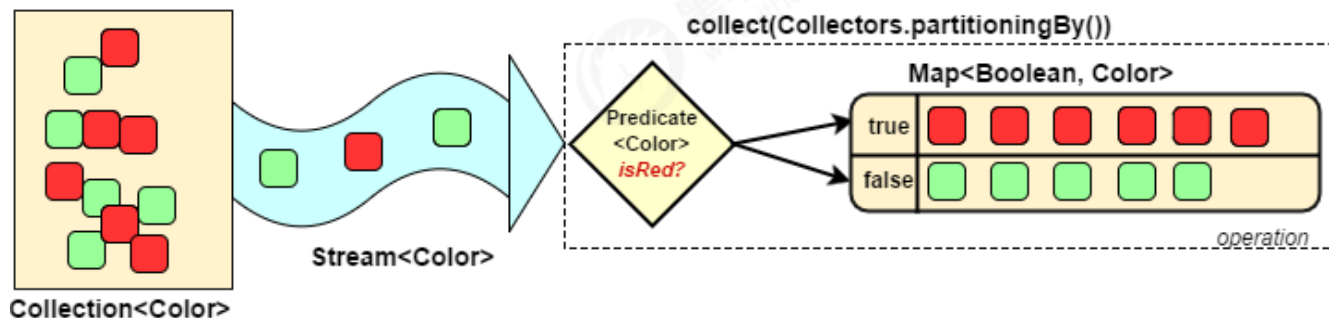
    map.forEach((k, v) -> {
        System.out.println(k + " == " + v);
    });
}
```

效果:

```
52 == {不及格=[Student{name='柳岩', age=52, socre=77}], 优秀=[Student{name='赵丽颖', age=52, socre=95}]}  
56 == {优秀=[Student{name='迪丽热巴', age=56, socre=99}], 良好=[Student{name='杨颖', age=56, socre=88}]}
```

## 对流中数据进行分区

`collectors.partitioningBy` 会根据值是否为true，把集合分割为两个列表，一个true列表，一个false列表。



```
// 分区  
@Test  
public void testPartition() {  
    Stream<Student> studentStream = Stream.of(  
        new Student("赵丽颖", 52, 95),  
        new Student("杨颖", 56, 88),  
        new Student("迪丽热巴", 56, 99),  
        new Student("柳岩", 52, 77));  
  
    // partitioningBy会根据值是否为true，把集合分割为两个列表，一个true列表，一个false列表。  
    Map<Boolean, List<Student>> map = studentStream.collect(Collectors.partitioningBy(s ->  
s.getSocre() > 90));  
  
    map.forEach((k, v) -> {  
        System.out.println(k + " == " + v);  
    });  
}
```

效果:

```
false == [Student{name='杨颖', age=56, socre=88}, Student{name='柳岩', age=52, socre=77}]  
true == [Student{name='赵丽颖', age=52, socre=95}, Student{name='迪丽热巴', age=56, socre=99}]
```

## 对流中数据进行拼接

`collectors.joining` 会根据指定的连接符，将所有元素连接成一个字符串。

```
// 拼接
@Test
public void testJoining() {
    Stream<Student> studentStream = Stream.of(
        new Student("赵丽颖", 52, 95),
        new Student("杨颖", 56, 88),
        new Student("迪丽热巴", 56, 99),
        new Student("柳岩", 52, 77));
    String collect = studentStream
        .map(Student::getName)
        .collect(Collectors.joining(">_<", "^_^", "^v^"));
    System.out.println(collect);
}
```

效果:

```
^_^赵丽颖>_<杨颖>_<迪丽热巴>_<柳岩^v^
```

## 小结

收集Stream流中的结果

到集合中: Collectors.toList()/Collectors.toSet()/Collectors.toCollection()

到数组中: toArray()/toArray(int[]::new)

聚合计算:

Collectors.maxBy/Collectors.minBy/Collectors.counting/Collectors.summingInt/Collectors.averagingInt

分组: Collectors.groupingBy

分区: Collectors.partitionBy

拼接: Collectors.joining

## 并行的Stream流

### 目标

了解串行的Stream流

掌握获取并行Stream流的两种方式

### 串行的Stream流

目前我们使用的Stream流是串行的，就是在一个线程上执行。

```
@Test
public void test0Serial() {
    long count = Stream.of(4, 5, 3, 9, 1, 2, 6)
        .filter(s -> {
            System.out.println(Thread.currentThread() + ", s = " + s);
            return true;
        })
        .count();

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

效果:

```
Thread[main,5,main], s = 4
Thread[main,5,main], s = 5
Thread[main,5,main], s = 3
Thread[main,5,main], s = 9
Thread[main,5,main], s = 1
Thread[main,5,main], s = 2
Thread[main,5,main], s = 6
```

## 并行的Stream流

parallelStream其实就是一个并行执行的流。它通过默认的ForkJoinPool，可能提高多线程任务的速度。

### 获取并行Stream流的两种方式

1. 直接获取并行的流
2. 将串行流转成并行流

```
@Test
public void testgetParallelStream() {
    ArrayList<Integer> list = new ArrayList<>();
    // 直接获取并行的流
    // Stream<Integer> stream = list.parallelStream();
    // 将串行流转成并行流
    Stream<Integer> stream = list.stream().parallel();
}
```

并行操作代码:



```
@Test
public void test0Parallel() {
    long count = Stream.of(4, 5, 3, 9, 1, 2, 6)
        .parallel() // 将流转成并发流,Stream处理的时候将才去
        .filter(s -> {
            System.out.println(Thread.currentThread() + ", s = " + s);
            return true;
        })
        .count();

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

效果:

```
Thread[ForkJoinPool.commonPool-worker-13,5,main], s = 3
Thread[ForkJoinPool.commonPool-worker-19,5,main], s = 6
Thread[main,5,main], s = 1
Thread[ForkJoinPool.commonPool-worker-5,5,main], s = 5
Thread[ForkJoinPool.commonPool-worker-23,5,main], s = 4
Thread[ForkJoinPool.commonPool-worker-27,5,main], s = 2
Thread[ForkJoinPool.commonPool-worker-9,5,main], s = 9
count = 7
```

## 小结

获取并行流有两种方式:

直接获取并行流: parallelStream()

将串行流转成并行流: parallel()

## 并行和串行Stream流的效率对比

### 目标

使用for循环，串行Stream流，并行Stream流来对5亿个数字求和。看消耗的时间。

```
public class Demo06 {
    private static long times = 50000000000L;
    private long start;
    @Before
    public void init() {
        start = System.currentTimeMillis();
    }

    @After
    public void destory() {
        long end = System.currentTimeMillis();
        System.out.println("消耗时间: " + (end - start));
    }
}
```

```
}

// 测试效率,parallelStream 120
@Test
public void parallelStream() {
    System.out.println("serialStream");
    LongStream.rangeClosed(0, times)
        .parallel()
        .reduce(0, Long::sum);
}

// 测试效率,普通Stream 342
@Test
public void serialStream() {
    System.out.println("serialStream");
    LongStream.rangeClosed(0, times)
        .reduce(0, Long::sum);
}

// 测试效率,正常for循环 421
@Test
public void forAdd() {
    System.out.println("forAdd");
    long result = 0L;
    for (long i = 1L; i < times; i++) {
        result += i;
    }
}
}
```

我们可以看到parallelStream的效率是最高的。

Stream并行处理的过程会分而治之，也就是将一个大任务切分成多个小任务，这表示每个任务都是一个操作。

## parallelStream线程安全问题

### 目标

解决parallelStream线程安全问题

```
// 并行流注意事项
@Test
public void parallelStreamNotice() {
    ArrayList<Integer> list = new ArrayList<Integer>();
    for (int i = 0; i < 1000; i++) {
        list.add(i);
    }

    List<Integer> newList = new ArrayList<>();
    // 使用并行的流往集合中添加数据
```

```
list.parallelStream()
    .forEach(s -> {
        newList.add(s);
    });

system.out.println("newList = " + newList.size());
}
```

运行效果：

```
newList = 903
```

我们明明是往集合中添加1000个元素，而实际上只有903个元素。

解决方法： 加锁、使用线程安全的集合或者调用Stream的 `toArray()` / `collect()` 操作就是满足线程安全的了。

## parallelStream背后的技术

### 目标

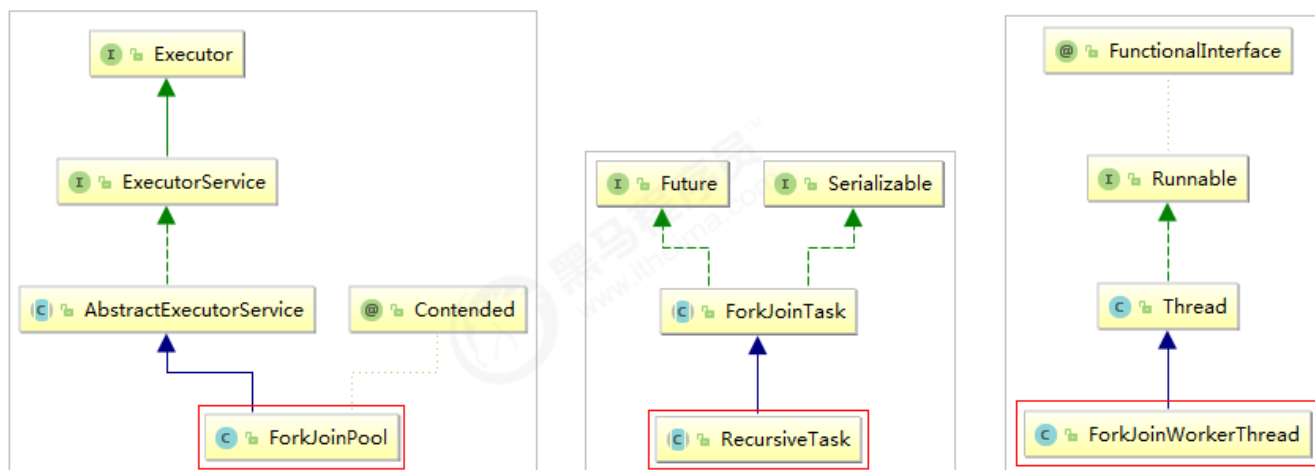
介绍Fork/Join框架

了解Fork/Join原理

### Fork/Join框架介绍

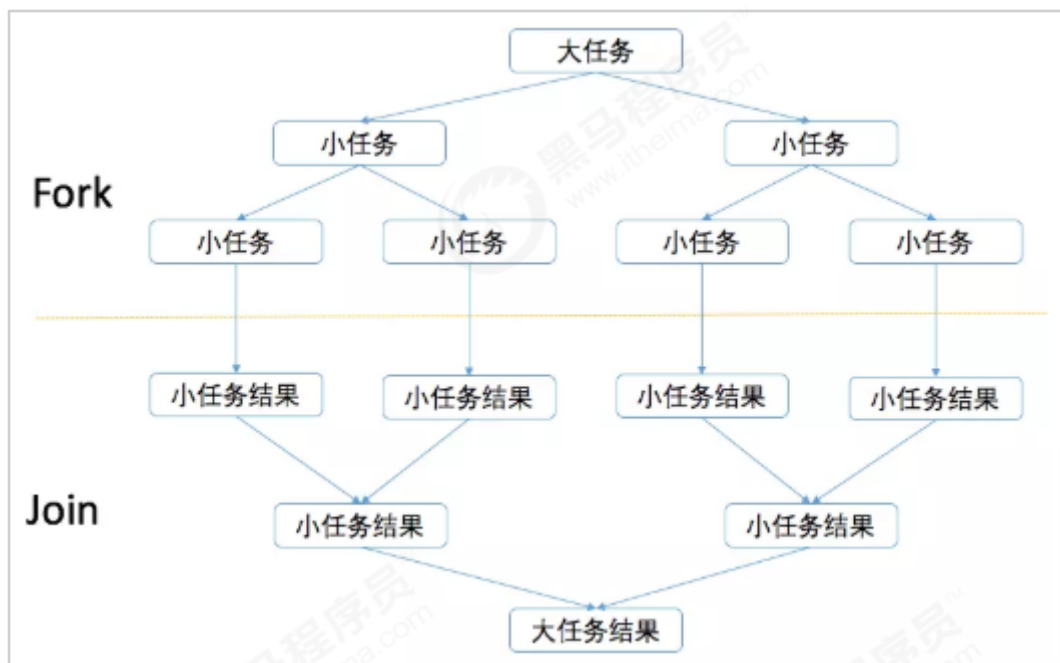
parallelStream使用的是Fork/Join框架。Fork/Join框架自JDK 7引入。Fork/Join框架可以将一个大任务拆分为很多小任务来异步执行。Fork/Join框架主要包含三个模块：

1. 线程池：ForkJoinPool
2. 任务对象：ForkJoinTask
3. 执行任务的线程：ForkJoinWorkerThread



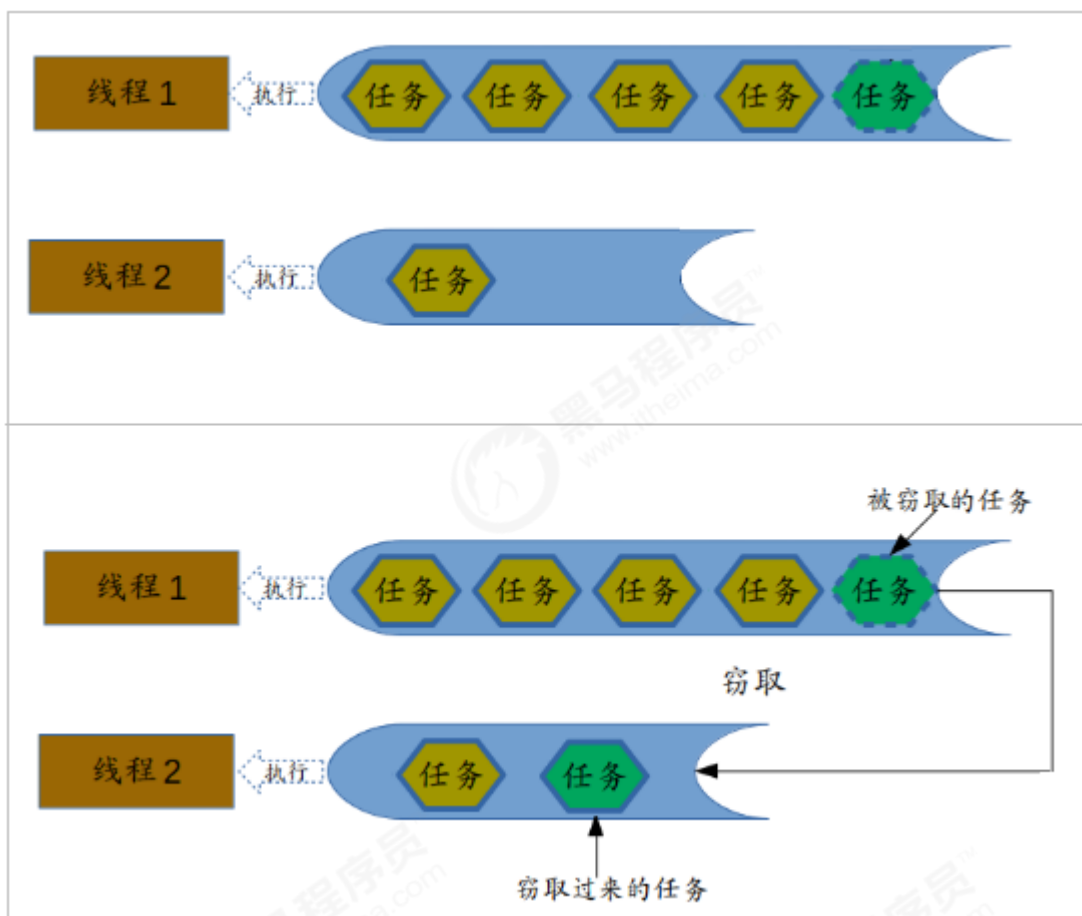
### Fork/Join原理-分治法

ForkJoinPool主要用来使用分治法(Divide-and-Conquer Algorithm)来解决问题。典型的应用比如快速排序算法，ForkJoinPool需要使用相对少的线程来处理大量的任务。比如要对1000万个数据进行排序，那么会将这个任务分割成两个500万的排序任务和一个针对这两组500万数据的合并任务。以此类推，对于500万的数据也会做出同样的分割处理，到最后会设置一个阈值来规定当数据规模到多少时，停止这样的分割处理。比如，当元素的数量小于10时，会停止分割，转而使用插入排序对它们进行排序。那么到最后，所有的任务加起来会有大概2000000+个。问题的关键在于，对于一个任务而言，只有当它所有的子任务完成之后，它才能够被执行。



## Fork/Join原理-工作窃取算法

Fork/Join最核心的地方就是利用了现代硬件设备多核，在一个操作时候会有空闲的cpu，那么如何利用好这个空闲的cpu就成了提高性能的关键，而这里我们要提到的工作窃取（work-stealing）算法就是整个Fork/Join框架的核心理念。Fork/Join工作窃取（work-stealing）算法是指某个线程从其他队列里窃取任务来执行。



那么为什么需要使用工作窃取算法呢？假如我们需要做一个比较大的任务，我们可以把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，于是把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应，比如A线程负责处理A队列里的任务。但是有的线程会先把自己队列里的任务干完，而其他线程对应的队列里还有任务等待处理。干完活的线程与其等着，不如去帮其他线程干活，于是它就去其他线程的队列里窃取一个任务来执行。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

工作窃取算法的优点是充分利用线程进行并行计算，并减少了线程间的竞争，其缺点是在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且消耗了更多的系统资源，比如创建多个线程和多个双端队列。

上文中已经提到了在Java 8引入了自动并行化的概念。它能够让一部分Java代码自动地以并行的方式执行，也就是我们使用了ForkJoinPool的ParallelStream。

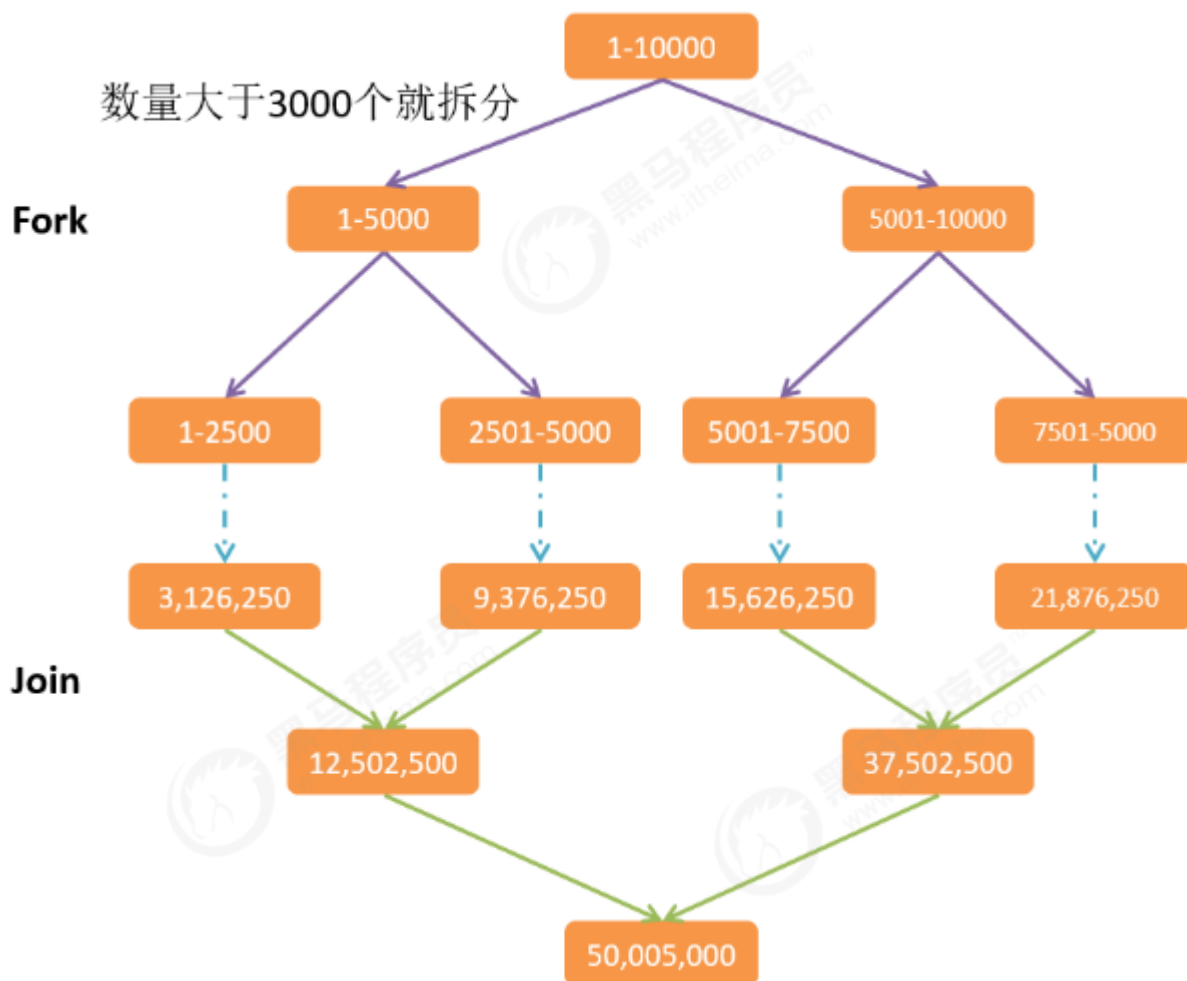
对于ForkJoinPool通用线程池的线程数量，通常使用默认值就可以了，即运行时计算机的处理器数量。可以通过设置系统属性：`java.util.concurrent.ForkJoinPool.common.parallelism=N`（N为线程数量），来调整ForkJoinPool的线程数量，可以尝试调整成不同的参数来观察每次的输出结果。

## 小结

1. 介绍了Fork/Join框架,他是JDK7推出的一套新的线程框架
2. Fork/Join框架-分治法，工作窃取算法

## Fork/Join案例

需求：使用Fork/Join计算1-10000的和，当一个任务的计算数量大于3000时拆分任务，数量小于3000时计算。



```
package com.itheima.demo05stream;

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import java.util.concurrent.RecursiveTask;

public class Demo07ForkJoin {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        ForkJoinPool pool = new ForkJoinPool();
        SumRecursiveTask task = new SumRecursiveTask(1, 10000L);
        Long result = pool.invoke(task);
        System.out.println("result = " + result);

        long end = System.currentTimeMillis();

        System.out.println("消耗的时间为: " + (end - start));
    }
}
```



```
}

class SumRecursiveTask extends RecursiveTask<Long> {
    private static final long THRESHOLD = 3000L;
    private final long start;
    private final long end;

    public SumRecursiveTask(long start, long end) {
        this.start = start;
        this.end = end;
    }

    @Override
    protected Long compute() {
        long length = end - start;

        if (length <= THRESHOLD) {
            // 任务不用再拆分了.可以计算了
            long sum = 0;
            for (long i = start; i <= end; i++) {
                sum += i;
            }
            System.out.println("计算: " + start + " -> " + end + ", 结果为: " + sum);
            return sum;
        } else {
            // 数量大于预定的数量,任务还需要再拆分
            long middle = (start + end) / 2;
            System.out.println("拆分: 左边 " + start + " -> " + middle + ", 右边 " + (middle + 1) + " -> " + end);

            SumRecursiveTask left = new SumRecursiveTask(start, middle);
            left.fork();
            SumRecursiveTask right = new SumRecursiveTask(middle + 1, end);
            right.fork();
            return left.join() + right.join();
        }
    }
}
```

## 小结

1. parallelStream是线程不安全的
2. parallelStream适用的场景是CPU密集型的，只是做到别浪费CPU，假如本身电脑CPU的负载很大，那还到处用并行流，那并不能起到作用
3. I/O密集型 磁盘I/O、网络I/O都属于I/O操作，这部分操作是较少消耗CPU资源，一般并行流中不适用于I/O密集型的操作，就比如使用并行流进行大批量的消息推送，涉及到了大量I/O，使用并行流反而慢了很多
4. 在使用并行流的时候是无法保证元素的顺序的，也就是即使你用了同步集合也只能保证元素都正确但无法保证其中的顺序

# 学习JDK 8新增的Optional类

## 目标

回顾以前对null的处理方式

介绍Optional类

掌握Optional的基本使用

掌握Optional的高级使用

## 以前对null的处理方式

```
@Test
public void test01() {
    String userName = "凤姐";
    // String userName = null;
    if (userName != null) {
        System.out.println("用户名为:" + userName);
    } else {
        System.out.println("用户名不存在");
    }
}
```

## Optional类介绍

Optional是一个没有子类的工具类，Optional是一个可以为null的容器对象。它的作用主要就是为了解决避免Null检查，防止NullPointerException。



## Optional的基本使用

Optional类的创建方式：

```
Optional.of(T t) : 创建一个 Optional 实例
Optional.empty() : 创建一个空的 Optional 实例
Optional.ofNullable(T t):若 t 不为 null,创建 Optional 实例,否则创建空实例
```





Optional类的常用方法：

isPresent() : 判断是否包含值,包含值返回true, 不包含值返回false  
get() : 如果Optional有值则将其返回, 否则抛出NoSuchElementException  
orElse(T t) : 如果调用对象包含值, 返回该值, 否则返回参数t  
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  
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()

```
@Test
public void test02() {
    // Optional<String> userNameO = Optional.of("凤姐");
    // Optional<String> userNameO = Optional.of(null);
    // Optional<String> userNameO = Optional.ofNullable(null);
    Optional<String> userNameO = Optional.empty();

    // isPresent() : 判断是否包含值,包含值返回true, 不包含值返回false。
    if (userNameO.isPresent()) {
        // get() : 如果Optional有值则将其返回, 否则抛出NoSuchElementException。
        String userName = userNameO.get();
        System.out.println("用户名为:" + userName);
    } else {
        System.out.println("用户名不存在");
    }
}
```

## Optional的高级使用

```
@Test
public void test03() {
    Optional<String> userNameO = Optional.of("凤姐");
    // Optional<String> userNameO = Optional.empty();

    // 存在做什么
    // userNameO.ifPresent(s -> System.out.println("用户名为" + s));

    // 存在做什么,不存在做什么
    userNameO.ifPresentOrElse(s -> System.out.println("用户名为" + s),
        () -> System.out.println("用户名不存在"));
}

@Test
public void test04() {
    // Optional<String> userNameO = Optional.of("凤姐");
    Optional<String> userNameO = Optional.empty();
    // 如果调用对象包含值, 返回该值, 否则返回参数t
    System.out.println("用户名为" + userNameO.orElse("null"));
}
```



```
// 如果调用对象包含值，返回该值，否则返回参数Supplier得到的值
String s1 = userName0.orElseGet(() -> {return "未知用户名";});
System.out.println("s1 = " + s1);
}
```

```
@Test
public void test05() {
    // User u = new User("凤姐", 18);
    // User u = new User(null, 18);
    // User u = null;
    // System.out.println(getUpperCaseUserName1(u));

    // 我们将可能会为null的变量构造Optional类型
    // User u = new User("凤姐", 18);
    User u = new User(null, 18);
    Optional<User> u0 = Optional.of(u);
    System.out.println(getUpperCaseUserName2(u0));
}

public String getUpperCaseUserName2(Optional<User> u0) {
    return u0.map(u -> u.getUserName())
        .map(name -> name.toUpperCase())
        .orElse("null");
}

/*public String getUpperCaseUserName1(User u) {
    if (u != null) {
        String userName = u.getUserName();
        if (userName != null) {
            return userName;
        } else {
            return null;
        }
    } else {
        return null;
    }
}*/
```

## 小结

Optional是一个可以为null的容器对象。orElse, ifPresent, ifPresentOrElse, map等方法避免对null的判断，写出更加优雅的代码。

## 学习JDK 8新的日期和时间 API

### 目标

了解旧版日期时间 API 存在的问题

## 新日期时间 API 介绍

掌握JDK 8的日期和时间类

掌握JDK 8的时间格式化与解析

掌握JDK 8的Instant时间戳

了解JDK 8的计算日期时间差类

了解JDK 8设置日期时间的时区

## 旧版日期时间 API 存在的问题

1. 设计很差：在java.util和java.sql的包中都有日期类，java.util.Date同时包含日期和时间，而java.sql.Date仅包含日期。此外用于格式化和解析的类在java.text包中定义。
2. 非线程安全：java.util.Date 是非线程安全的，所有的日期类都是可变的，这是Java日期类最大的问题之一。
3. 时区处理麻烦：日期类并不提供国际化，没有时区支持，因此Java引入了java.util.Calendar和java.util.TimeZone类，但他们同样存在上述所有的问题。

## 新日期时间 API 介绍

JDK 8中增加了一套全新的日期时间API，这套API设计合理，是线程安全的。新的日期及时间API位于 `java.time` 包中，下面是一些关键类。

`LocalDate`：表示日期，包含年月日，格式为 2019-10-16

`LocalTime`：表示时间，包含时分秒，格式为 16:38:54.158549300

`LocalDateTime`：表示日期时间，包含年月日，时分秒，格式为 2018-09-06T15:33:56.750

`DateTimeFormatter`：日期时间格式化类。

`Instant`：时间戳，表示一个特定的时间瞬间。

`Duration`：用于计算2个时间(LocalTime，时分秒)的距离

`Period`：用于计算2个日期(LocalDate，年月日)的距离

`ZonedDateTime`：包含时区的时间

Java中使用的历法是ISO 8601日历系统，它是世界民用历法，也就是我们所说的公历。平年有365天，闰年是366天。此外Java 8还提供了4套其他历法，分别是：

- `ThaiBuddhistDate`：泰国佛教历
- `MinguoDate`：中华民国历
- `JapaneseDate`：日本历
- `HijrahDate`：伊斯兰历

## JDK 8的日期和时间类



LocalDate、LocalTime、LocalDateTime类的实例是不可变的对象，分别表示使用 ISO-8601 日历系统的日期、时间、日期和时间。它们提供了简单的日期或时间，并不包含当前的时间信息，也不包含与时区相关的信息。

```
// LocalDate:获取日期时间的信息。格式为 2019-10-16
@Test
public void test01() {
    // 创建指定日期
    LocalDate fj = LocalDate.of(1985, 9, 23);
    System.out.println("fj = " + fj); // 1985-09-23

    // 得到当前日期
    LocalDate nowDate = LocalDate.now();
    System.out.println("nowDate = " + nowDate); // 2019-10-16

    // 获取日期信息
    System.out.println("年: " + nowDate.getYear());
    System.out.println("月: " + nowDate.getMonthValue());
    System.out.println("日: " + nowDate.getDayOfMonth());
    System.out.println("星期: " + nowDate.getDayOfWeek());
}

// LocalTime类: 获取时间信息。格式为 16:38:54.158549300
@Test
public void test02() {
    // 得到指定的时间
    LocalTime time = LocalTime.of(12,15, 28, 129_900_000);
    System.out.println("time = " + time);

    // 得到当前时间
    LocalTime nowTime = LocalTime.now();
    System.out.println("nowTime = " + nowTime);

    // 获取时间信息
    System.out.println("小时: " + nowTime.getHour());
    System.out.println("分钟: " + nowTime.getMinute());
    System.out.println("秒: " + nowTime.getSecond());
    System.out.println("纳秒: " + nowTime.getNano());
}

// LocalDateTime类: 获取日期时间信息。格式为 2018-09-06T15:33:56.750
@Test
public void test03() {
    LocalDateTime fj = LocalDateTime.of(1985, 9, 23, 9, 10, 20);
    System.out.println("fj = " + fj); // 1985-09-23T09:10:20

    // 得到当前日期时间
    LocalDateTime now = LocalDateTime.now();
    System.out.println("now = " + now); // 2019-10-16T16:42:24.497896800

    System.out.println(now.getYear());
    System.out.println(now.getMonthValue());
    System.out.println(now.getDayOfMonth());
}
```



```
System.out.println(now.getHour());  
System.out.println(now.getMinute());  
System.out.println(now.getSecond());  
System.out.println(now.getNano());  
}
```

对日期时间的修改，对已存在的LocalDate对象，创建它的修改版，最简单的方式是使用withAttribute方法。withAttribute方法会创建对象的一个副本，并按照需要修改它的属性。以下所有的方法都返回了一个修改属性的对象，他们不会影响原来的对象。

```
// LocalDateTime类：对日期时间的修改  
@Test  
public void test05() {  
    LocalDateTime now = LocalDateTime.now();  
    System.out.println("now = " + now);  
  
    // 修改日期时间  
    LocalDateTime setYear = now.withYear(2078);  
    System.out.println("修改年份: " + setYear);  
    System.out.println("now == setYear: " + (now == setYear));  
  
    System.out.println("修改月份: " + now.withMonth(6));  
    System.out.println("修改小时: " + now.withHour(9));  
    System.out.println("修改分钟: " + now.withMinute(11));  
  
    // 再当前对象的基础上加上或减去指定的时间  
    LocalDateTime localDateTime = now.plusDays(5);  
    System.out.println("5天后: " + localDateTime);  
    System.out.println("now == localDateTime: " + (now == localDateTime));  
    System.out.println("10年后: " + now.plusYears(10));  
    System.out.println("20月后: " + now.plusMonths(20));  
  
    System.out.println("20年前: " + now.minusYears(20));  
    System.out.println("5月前: " + now.minusMonths(5));  
    System.out.println("100天前: " + now.minusDays(100));  
}
```

## 日期时间的比较

```
// 日期时间的比较  
@Test  
public void test06() {  
    // 在JDK8中，LocalDate类中使用isBefore()、isAfter()、equals()方法来比较两个日期，可直接进行比较。  
    LocalDate now = LocalDate.now();  
    LocalDate date = LocalDate.of(2018, 8, 8);  
  
    System.out.println(now.isBefore(date)); // false  
    System.out.println(now.isAfter(date)); // true  
}
```

## JDK 8的时间格式化与解析

通过 `java.time.format.DateTimeFormatter` 类可以进行日期时间解析与格式化。

```
// 日期格式化
@Test
public void test04() {
    // 得到当前日期时间
    LocalDateTime now = LocalDateTime.now();

    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    // 将日期时间格式化为字符串
    String format = now.format(formatter);
    System.out.println("format = " + format);

    // 将字符串解析为日期时间
    LocalDateTime parse = LocalDateTime.parse("1985-09-23 10:12:22", formatter);
    System.out.println("parse = " + parse);
}
```

## JDK 8的 Instant 类

Instant 时间戳/时间线，内部保存了从1970年1月1日 00:00:00以来的秒和纳秒。

```
// 时间戳
@Test
public void test07() {
    Instant now = Instant.now();
    System.out.println("当前时间戳 = " + now);
    // 获取从1970年1月1日 00:00:00的秒
    System.out.println(now.getNano());
    System.out.println(now.getEpochSecond());
    System.out.println(now.toEpochMilli());
    System.out.println(System.currentTimeMillis());

    Instant instant = Instant.ofEpochSecond(5);
    System.out.println(instant);
}
```

## JDK 8的计算日期时间差类

Duration/Period类: 计算日期时间差。

1. Duration: 用于计算2个时间(LocalTime, 时分秒)的距离
2. Period: 用于计算2个日期(LocalDate, 年月日)的距离

```
// Duration/Period类：计算日期时间差
@Test
public void test08() {
    // Duration计算时间的距离
    LocalDateTime now = LocalDateTime.now();
    LocalDateTime time = LocalDateTime.of(14, 15, 20);
    Duration duration = Duration.between(time, now);
    System.out.println("相差的天数：" + duration.toDays());
    System.out.println("相差的小时数：" + duration.toHours());
    System.out.println("相差的分钟数：" + duration.toMinutes());
    System.out.println("相差的秒数：" + duration.toSeconds());

    // Period计算日期的距离
    LocalDate nowDate = LocalDate.now();
    LocalDate date = LocalDate.of(1998, 8, 8);
    // 让后面的时间减去前面的时间
    Period period = Period.between(date, nowDate);
    System.out.println("相差的年：" + period.getYears());
    System.out.println("相差的月：" + period.getMonths());
    System.out.println("相差的天：" + period.getDays());
}
```

## JDK 8的时间校正器

有时我们可能需要获取例如：将日期调整到“下一个月的第一天”等操作。可以通过时间校正器来进行。

- TemporalAdjuster：时间校正器。
- TemporalAdjusters：该类通过静态方法提供了大量的常用TemporalAdjuster的实现。

```
// TemporalAdjuster类：自定义调整时间
@Test
public void test09() {
    LocalDateTime now = LocalDateTime.now();

    // 得到下一个月的第一天
    TemporalAdjuster firstWeekDayOfNextMonth = temporal -> {
        LocalDateTime dateTime = (LocalDateTime) temporal;
        LocalDateTime nextMonth = dateTime.plusMonths(1).withDayOfMonth(1);
        System.out.println("nextMonth = " + nextMonth);
        return nextMonth;
    };

    LocalDateTime nextMonth = now.with(firstWeekDayOfNextMonth);
    System.out.println("nextMonth = " + nextMonth);
}
```

## JDK 8设置日期时间的时区



Java8 中加入了对时区的支持，LocalDate、LocalTime、LocalDateTime是不带时区的，带时区的日期时间类分别为：ZonedDateTime、ZoneTime、ZoneDateTime。

其中每个时区都对应着 ID，ID的格式为“区域/城市”。例如：Asia/Shanghai 等。

ZoneId：该类中包含了所有的时区信息。

```
// 设置日期时间的时区
@Test
public void test10() {
    // 1.获取所有的时区ID
    // ZoneId.getAvailableZoneIds().forEach(System.out::println);

    // 不带时间,获取计算机的当前时间
    LocalDateTime now = LocalDateTime.now(); // 中国使用的东八区的时区.比标准时间早8个小时
    System.out.println("now = " + now);

    // 2.操作带时区的类
    // now(Clock.systemUTC()): 创建世界标准时间
    ZonedDateTime bz = ZonedDateTime.now(Clock.systemUTC());
    System.out.println("bz = " + bz);

    // now(): 使用计算机的默认的时区,创建日期时间
    ZonedDateTime now1 = ZonedDateTime.now();
    System.out.println("now1 = " + now1); // 2019-10-
    19T16:19:44.007153500+08:00[Asia/Shanghai]

    // 使用指定的时区创建日期时间
    ZonedDateTime now2 = ZonedDateTime.now(ZoneId.of("America/Vancouver"));
    System.out.println("now2 = " + now2); // 2019-10-19T01:21:44.248794200-
    07:00[America/Vancouver]
}
```

## 小结

详细学习了新的日期是时间相关类，LocalDate表示日期,包含年月日,LocalTime表示时间,包含时分秒,LocalDateTime = LocalDate + LocalTime,时间的格式化和解析,通过DateTimeFormatter类型进行。

学习了Instant类,方便操作秒和纳秒,一般是给程序使用的.学习Duration/Period计算日期或时间的距离,还使用时间调整器方便的调整时间,学习了带时区的3个类ZoneDate/ZoneTime/ZoneDateTime

JDK 8新的日期和时间 API的优势:

1. 新版的日期和时间API中，日期和时间对象是不可变的。操纵的日期不会影响老值，而是新生成一个实例。
2. 新的API提供了两种不同的时间表示方式，有效地区分了人和机器的不同需求。
3. TemporalAdjuster可以更精确的操纵日期，还可以自定义日期调整器。
4. 是线程安全的



# 学习JDK 8重复注解与类型注解

## 目标

掌握重复注解的使用

掌握类型注解的使用

## 重复注解的使用

自从Java 5中引入注解以来，注解开始变得非常流行，并在各个框架和项目中被广泛使用。不过注解有一个很大的限制是：在同一个地方不能多次使用同一个注解。JDK 8引入了重复注解的概念，允许在同一个地方多次使用同一个注解。在JDK 8中使用@Repeatable注解定义重复注解。

重复注解的使用步骤：

### 1. 定义重复的注解容器注解

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyTests {
    MyTest[] value();
}
```

### 2. 定义一个可以重复的注解

```
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(MyTests.class)
@interface MyTest {
    String value();
}
```

### 3. 配置多个重复的注解

```
@MyTest("tbc")
@MyTest("tba")
@MyTest("tba")
public class Demo01 {
    @MyTest("mbc")
    @MyTest("mba")
    public void test() throws NoSuchMethodException {

    }
}
```

### 4. 解析得到指定注解

```
// 3. 配置多个重复的注解
@MyTest("tbc")
@MyTest("tba")
@MyTest("tba")
```



```
public class Demo01 {
    @Test
    @MyTest("mbc")
    @MyTest("mba")
    public void test() throws NoSuchMethodException {
        // 4.解析得到类上的指定注解
        MyTest[] tests = Demo01.class.getAnnotationsByType(MyTest.class);
        for (MyTest test : tests) {
            System.out.println(test.value());
        }

        // 得到方法上的指定注解
        Annotation[] tests1 =
        Demo01.class.getMethod("test").getAnnotationsByType(MyTest.class);
        for (Annotation annotation : tests1) {
            System.out.println("annotation = " + annotation);
        }
    }
}
```

## 类型注解的使用

JDK 8为@Target元注解新增了两种类型：TYPE\_PARAMETER，TYPE\_USE。

TYPE\_PARAMETER：表示该注解能写在类型参数的声明语句中。类型参数声明如：<T>、

TYPE\_USE：表示注解可以再任何用到类型的地方使用。

TYPE\_PARAMETER的使用

```
@Target(ElementType.TYPE_PARAMETER)
@interface TyptParam {
}
```

```
public class Demo02<@TyptParam T> {
    public static void main( String[] args) {

    }

    public <@TyptParam E> void test( String a) {

    }
}
```

TYPE\_USE的使用

```
@Target(ElementType.TYPE_USE)
@interface NotNull {
}
```

```
public class Demo02<@TyptParam T extends String> {
    private @NotNull int a = 10;
    public static void main(@NotNull String[] args) {
        @NotNull int x = 1;

        @NotNull String s = new @NotNull String();
    }

    public <@TyptParam E> void test( String a) {

    }
}
```

## 小结

通过@Repeatable元注解可以定义可重复注解，`TYPE_PARAMETER`可以让注解放在泛型上，`TYPE_USE`可以让注解放在类型的前面