Java 泛型

1. 什么是泛型

泛型不只是 Java 语言所特有的特性，泛型是程序设计语言的一种特性。允许程序员在强类型的程序设计语言中编写代码时定义一些可变部分，那些部分在使用前必须做出声明。

Java 中的集合类是支持泛型的，例如：

```
public class test {
    public static void main(String[] args) {
        ArrayList< > integers = new ArrayList<>();
    }
}
```

代码中的 `<Integer>` 就是泛型，我们**把类型像参数一样传递**，尖括号中间就是数据类型，我们可以称之为**实际类型参数**，这里实际类型参数的数据类型只能为引用数据类型。

2. 为什么需要泛型

我们在使用 `ArrayList` 实现类的时候，如果**没有指定泛型**，`IDEA` 会给出警告，代码也是可以顺利运行的。请看如下实例：

```
import java.util.ArrayList;

public class GenericsDemo1 {

    public static void main(String[] args) {
        ArrayList arrayList = new ArrayList();
        arrayList.add("Hello");
        String str = (String) arrayList.get(0);
        System.out.println("str=" + str);
    }

}
```

运行结果：

```
str=Hello
```

虽然运行时没有发生任何异常，但这样做有两个缺点：

1. **需要强制类型转换**： 由于 `ArrayList` 内部就是一个 `Object[]` 数组，在 `get()` 元素的时候，返回的是 `Object` 类型，所以在 `ArrayList` 外获取该对象，需要强制类型转换。其它的 `Collection`、`Map` 如果不使用泛型，也存在这个问题；
2. 可向集合中添加任意类型的对象，存在类型不安全风险。例如如下代码中，我们向列表中既添加了 `Integer` 类型，又添加了 `String` 类型：

```
import java.util.ArrayList;

public class GenericsDemo2 {
    public static void main(String[] args) {
        ArrayList arrayList = new ArrayList();
        arrayList.add(123);
        arrayList.add("Hello");
        String str = (String) arrayList.get(0);
        System.out.println("element=" + str);
    }
}
```

运行结果：

```
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer
cannot be cast to class java.lang.String (java.lang.Integer and java.lang.String
are in module java.base of loader 'bootstrap')
    at GenericsDemo2.main(GenericsDemo2.java:8)
```

上面列表第 1 个元素实际上是整型，但被我们强制转换为字符串类型，这是行不通的，因此会抛出 `ClassCastException` 异常。

使用泛型可以解决这些问题。泛型有如下优点：

1. 可以减少类型转换的次数，代码更加简洁；
2. 程序更加健壮：只要编译期没有警告，运行期就不会抛出 `ClassCastException` 异常；
3. 提高了代码的可读性：编写集合的时候，就限定了集合中能存放的类型。

3. 如何使用泛型

3.1 泛型使用

在代码中，这样使用泛型：

```
List<String> list = new ArrayList<String>();
// Java 7 及以后的版本中，构造方法中可以省略泛型类型：
List<String> list = new ArrayList<>();
```

要注意的是，变量声明的类型必须与传递给实际对象的类型保持一致，下面是错误的例子：

```
List<Object> list = new ArrayList<String>();
List<Number> numbers = new ArrayList(Integer);
```

3.2 自定义泛型类

3.2.1 Java 源码中泛型的定义

`java.util.ArrayList`是下面这样定义的：

类名后面的`<E>`就是泛型的定义，`E`不是 Java 中的一个具体的类型，它是 Java 泛型的通配符（注意是大写的，实际上就是`Element`的含义），可将其理解为一个占位符，**将其定义在类上，使用时才确定类型**。此处的命名不受限制，但最好有一定含义，例如`java.lang.HashMap`的泛型定义为`HashMap<K,V>`，`K`表示`Key`，`V`表示`Value`。

3.2.2 自定义泛型类实例1

下面我们来自定义一个泛型类，自定义泛型按照约定俗成可以叫`<T>`，具有`Type`的含义，实例如下：

实例演示

```
public class NumberGeneric<T> { // 把泛型定义在类上

    private T number; // 定义在类上的泛型，在类内部可以使用

    public T getNumber() {
        return number;
    }

    public void setNumber(T number) {
        this.number = number;
    }

    public static void main(String[] args) {
        // 实例化对象，指定元素类型为整型
        NumberGeneric<Integer> integerNumberGeneric = new NumberGeneric<>();
        // 分别调用set、get方法
        integerNumberGeneric.setNumber(123);
        System.out.println("integerNumber=" + integerNumberGeneric.getNumber());

        // 实例化对象，指定元素类型为长整型
        NumberGeneric<Long> longNumberGeneric = new NumberGeneric<>();
        // 分别调用set、get方法
        longNumberGeneric.setNumber(20L);
        System.out.println("longNumber=" + longNumberGeneric.getNumber());

        // 实例化对象，指定元素类型为双精度浮点型
        NumberGeneric<Double> doubleNumberGeneric = new NumberGeneric<>();
        // 分别调用set、get方法
        doubleNumberGeneric.setNumber(4000.0);
        System.out.println("doubleNumber=" + doubleNumberGeneric.getNumber());
    }

}
```

运行结果：

```
integerNumber=123
longNumber=20
doubleNumber=4000.0
```

我们在类的定义处也定义了泛型：`NumberGeneric<T>`；在类内部定义了一个`T`类型的`number`变量，并且为其添加了`setter`和`getter`方法。

对于泛型类的使用也很简单，在主方法中，创建对象的时候指定`T`的类型分别为`Integer`、`Long`、`Double`，类就可以自动转换成对应的类型了。

3.2.3 自定义泛型类实例2

对于含有多个泛型的类

`HashMap`类是如下这么定义的。如下是 Java 源码的截图：

参照`HashMap<K,V>`类的定义，下面我们来看看如何定义含有两个泛型的类，实例如下：

实例演示

```
public class KeyValueGeneric<K,V> { // 把两个泛型K、V定义在类上

    /**
     * 类型为K的key属性
     */
    private K key;

    /**
     * 类型为V的value属性
     */
    private V value;

    public K getKey() {
        return key;
    }

    public void setKey(K key) {
        this.key = key;
    }

    public V getValue() {
        return value;
    }

    public void setValue(V value) {
        this.value = value;
    }

    public static void main(String[] args) {
        // 实例化对象，分别指定元素类型为整型、长整型
        KeyValueGeneric<Integer, Long> integerLongKeyValueGeneric = new
KeyValueGeneric<>();
        // 调用setter、getter方法
        integerLongKeyValueGeneric.setKey(200);
```

```
        integerLongKeyValueGeneric.setValue(300L);
        System.out.println("key=" + integerLongKeyValueGeneric.getKey());
        System.out.println("value=" + integerLongKeyValueGeneric.getValue());

        // 实例化对象，分别指定元素类型为浮点型、字符串类型
        KeyValueGeneric<Float, String> floatStringKeyValueGeneric = new
KeyValueGeneric<>();
        // 调用setter、getter方法
        floatStringKeyValueGeneric.setKey(0.5f);
        floatStringKeyValueGeneric.setValue("零点五");
        System.out.println("key=" + floatStringKeyValueGeneric.getKey());
        System.out.println("value=" + floatStringKeyValueGeneric.getValue());
    }
}
```

运行结果：

```
key=200
value=300
key=0.5
value=零点五
```

3.3 自定义泛型方法

在类上定义的泛型，在方法中也可以使用。

泛型方法不一定写在泛型类当中。当类的调用者总是关心类中的某个泛型方法，不关心其他属性，这个时候就没必要再整个类上定义泛型了。

实例演示：

```
public class GenericMethod {

    /**
     * 泛型方法show
     * @param t 要打印的参数
     * @param <T> T
     */
    public <T> void show(T t) {
        System.out.println(t);
    }

    public static void main(String[] args) {
        // 实例化对象
        GenericMethod genericMethod = new GenericMethod();
        // 调用泛型方法show，传入不同类型的参数
        genericMethod.show("Java");
        genericMethod.show(222);
        genericMethod.show(222.0);
        genericMethod.show(222L);
    }
```

```
}
```

运行结果：

```
Java
222
222.0
222
```

实例中，使用 `<T>` 来定义 `show` 方法的泛型，它接收一个泛型的参数变量并在方法体打印；调用泛型方法也很简单，在主方法中实例化对象，调用对象下的泛型方法，可传入不同类型的参数。

4. 泛型类的子类

泛型类也是一个 Java 类，它也具有继承的特性。

泛型类的继承可分为两种情况：

1. 子类明确泛型类的类型参数变量；
2. 子类不明确泛型类的类型参数变量。

下面我们来分别看一下这两种情况。

4.1 明确类型参数变量

例如，有一个泛型接口：

```
public interface GenericInterface<T> { // 在接口上定义泛型
    void show(T t);
}
```

泛型接口的实现类如下：

```
public class GenericInterfaceImpl implements GenericInterface<String> { // 明确泛型类型为String类型
    @Override
    public void show(String s) {
        System.out.println(s);
    }
}
```

子类实现明确了泛型的参数变量为 `String` 类型。因此方法 `show()` 的重写也将 `T` 替换为了 `String` 类型。

4.2 不明确类型参数变量

当实现类不确定泛型类的参数变量时，实现类需要定义类型参数变量，调用者使用子类时，也需要传递类型参数变量。

如下是`GenericInterface`接口的另一个实现类：

```
public class GenericInterfaceImpl1<T> implements GenericInterface<T> { // 实现类也需要定义泛型参数变量
    @Override
    public void show(T t) {
        System.out.println(t);
    }
}
```

在主方法中调用实现类的`show()`方法：

```
    public static void main(String[] args) {
        GenericInterfaceImpl1<Float> floatGenericInterfaceImpl1 = new GenericInterfaceImpl1<>();
        floatGenericInterfaceImpl1.show(100.1f);
    }
```

5. 类型通配符

我们先来看一个泛型作为方法参数的实例：

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

public class GenericDemo3 {
    /**
     * 遍历并打印集合中的每一个元素
     * @param list 要接收的集合
     */
    public void printListElement(List<Object> list) {
        for (Object o : list) {
            System.out.println(o);
        }
    }
}
```

观察上面的代码，参数`list`的限定的泛型类型为`Object`， 也就是说，这个方法只能接收元素为`Object`类型的集合，如果我们想传递其他元素类型的集合，是行不通的。例如，如果传递装载`Integer`元素的集合，程序在编译阶段就会报错：

```java
public class GenericDemo3 {
    public void printListElement(List<Object> list) {  1 usage
        for (Object o : list) {
            System.out.println(o);
        }
    }

    public static void main(String[] args) {
        ArrayList<Integer> integers = new ArrayList<>();
        integers.add(1);
        integers.add(2);
        integers.add(3);
        GenericDemo3 demo1 = new GenericDemo3();
        demo1.printListElement(integers);
    }
}
```

运行报错：

```
java: 不兼容的类型: java.util.ArrayList<java.lang.Integer>无法转换为
java.util.List<java.lang.Object>
```

5.1 无限定通配符

想要解决这个问题，使用类型通配符即可，修改方法参数处的代码，将`<>`中间的Object改为`?`即可：

```java
public void printListElement(List<?> list) {}
```

此处的`?`就是类型通配符，表示可以匹配任意类型，因此调用方可以传递任意泛型类型的列表。

完整实例如下：

实例演示

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

public class GenericDemo3 {
    /**
     * 遍历并打印集合中的每一个元素
     * @param list 要接收的集合
     */
    public void printListElement(List<?> list) {
        for (Object o : list) {
            System.out.println(o);
```

```
        }
    }

    public static void main(String[] args) {
        // 实例化一个整型的列表
        List<Integer> integers = new ArrayList<>();
        // 添加元素
        integers.add(1);
        integers.add(2);
        integers.add(3);
        GenericDemo3 genericDemo3 = new GenericDemo3();
        // 调用printListElement()方法
        genericDemo3.printListElement(integers);

        // 实例化一个字符串类型的列表
        List<String> strings = new ArrayList<>();
        // 添加元素
        strings.add("Hello");
        strings.add("test");
        // 调用printListElement()方法
        genericDemo3.printListElement(strings);
    }
}
```

运行结果：

```
1
2
3
Hello
test
```

5.2 extends 通配符

`extends`通配符用来限定泛型的上限。什么意思呢？依旧以上面的实例为例，我们希望方法接收的`List`集合限定在数值类型内（float、integer、double、byte 等），不希望其他类型可以传入（比如字符串）。此时，可以改写上面的方法定义，设定上界通配符：

```
public void printListElement(List<? extends Number> list) {}
```

这样的写法的含义为：`List`集合装载的元素只能是`Number`自身或其子类（`Number`类型是所有数值类型的父类），完整实例如下：

实例演示

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

public class GenericDemo4 {
    /**
```

```
     * 遍历并打印集合中的每一个元素
     * @param list 要接收的集合
     */
    public void printListElement(List<? extends Number> list) {
        for (Object o : list) {
            System.out.println(o);
        }
    }

    public static void main(String[] args) {
        // 实例化一个整型的列表
        List<Integer> integers = new ArrayList<>();
        // 添加元素
        integers.add(1);
        integers.add(2);
        integers.add(3);
        GenericDemo4 genericDemo3 = new GenericDemo4();
        // 调用printListElement()方法
        genericDemo3.printListElement(integers);

    }
}
```

运行结果：

```
1
2
3
```

5.3 super 通配符

上面是如何设定通配符上界，也就不难理解通配符的下界了，可以限定传递的参数只能是某个类型的父类。

语法如下：

```
<? super Type>
```

Java 反射

1. 什么是反射

通常情况下，我们想调用一个类内部的属性或方法，需要先实例化这个类，然后通过对象去调用类内部的属性和方法；通过 Java 的反射机制，我们就可以在程序的运行状态中，动态获取类的信息，注入类内部的属性和方法，完成对象的实例化等操作。

在编写代码时直接通过 `new` 的方式就可以实例化一个对象，访问其属性和方法，为什么偏偏要绕个弯子，通过反射机制来进行这些操作呢？下面我们就来看一下反射的使用场景。

2. 反射的使用场景

Java 的反射机制，主要用来**编写一些通用性较高的代码**或者**编写框架**的时候使用。

3. Class 类

3.1 获取 Class 对象的方法

想要使用反射，就要获取某个 `class` 文件对应的 `Class` 对象，我们有 3 种方法：

1. **类名.class**：即通过一个 `Class` 的静态变量 `class` 获取，实例如下：

```
Class cls = Student.class;
```

1. **对象.getClass ()**：前提是有该类的对象实例，该方法由 `java.lang.Object` 类提供，实例如下：

```
Student Student = new Student("小花");
Class Student.getClass();
```

1. **Class.forName (“包名。类名”)**：如果知道一个类的完整包名，可以通过 `Class` 类的静态方法 `forName()` 获得 `Class` 对象，实例如下：

```
class cls = Class.forName("java.util.ArrayList");
```

3.2 实例

```
package com.rp.reflect;

public class Student {
    // 无参构造方法
    public Student() {
    }

    // 有参构造方法
    public Student(String nickname) {
        this.nickname = nickname;
    }

    // 昵称
    private String nickname;
```

```
    // 定义getter和setter方法
    public String getNickname() {
        return nickname;
    }

    public void setNickname(String nickname) {
        this.nickname = nickname;
    }

    public static void main(String[] args) throws ClassNotFoundException {
        // 方法1：类名.class
        Class cls1 = Student.class;

        // 方法2：对象.getClass()
        Student student = new Student();
        Class cls2 = student.getClass();

        // 方法3：Class.forName("包名.类名")
        Class cls3 = Class.forName("com.rp.reflect.Student");
    }

}
```

代码中，我们在 `com.rp.reflect` 包下定义了一个 `Student` 类，并在主方法中，使用了 3 种方法获取 `Class` 的实例对象，其 `forName()` 方法会抛出一个 `ClassNotFoundException`。

3.3 调用构造方法

- `Constructor getConstructor(Class...)`：获取某个 `public` 的构造方法；
- `Constructor getDeclaredConstructor(Class...)`：获取某个构造方法；
- `Constructor[] getConstructors()`：获取所有 `public` 的构造方法；
- `Constructor[] getDeclaredConstructors()`：获取所有构造方法。

3.4 获取字段

- `Field getField(name)`：根据属性名获取某个 `public` 的字段（包含父类继承）；
- `Field getDeclaredField(name)`：根据属性名获取当前类的某个字段（不包含父类继承）；
- `Field[] getFields()`：获得所有的 `public` 字段（包含父类继承）；
- `Field[] getDeclaredFields()`：获取当前类的所有字段（不包含父类继承）。

```
// 类名.class 方式获取 Class 实例
 Class cls1 = Student1.class;
// 获取 public 的字段 position
 Field position = cls1.getField("position");
// 获取字段 balance
 Field balance = cls1.getDeclaredField("balance");
```

3.5 调用方法

获取方法的目的就是调用方法，调用方法也就是让方法执行。

通常情况下，我们是这样调用对象下的实例方法（以 `String` 类的 `replace()` 方法为例）：

```
String name = new String("Colorful");
String result = name.replace("ful", "");
```

改写成通过反射方法调用：

实例演示

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class ReflectionDemo1 {
    public static void main(String[] args) throws NoSuchMethodException,
InvocationTargetException, IllegalAccessException {
        // 实例化字符串对象
        String name = new String("Colorful");
        // 获取 method 对象
        Method method = String.class.getMethod("replace", CharSequence.class,
CharSequence.class);
        // 调用 invoke() 执行方法
        String result = (String) method.invoke(name,  "ful", "");
        System.out.println(result);
    }
}
```

运行结果：

```
Color
```

代码中，调用 `Method` 实例的 `invoke(Object obj, Object...args)` 方法，就是通过反射来调用了该方法。

其中 `invoke()` 方法的第一个参数为对象实例，紧接着的可变参数就是要调用方法的参数，参数要保持一致。

3.6 反射应用

反射似乎离我们的实际开发非常遥远，实际情况也的确是这样的。因为我们在实际开发中基本不会用到反射。

Lambda 表达式

1. 什么是 Lambda 表达式

Lambda 表达式是一个匿名函数。使用它可以写出简洁、灵活的代码。作为一种更紧凑的代码风格，使 Java 语言的表达能力得到了提升。

2. 为什么需要 Lambda 表达式

在 Java 8 之前，编写一个匿名内部类的代码很冗长、可读性很差，如下：

```
public class LambdaDemo1 {

    public static void main(String[] args) {

        // 实例化一个 Runnable 接口的匿名实现类对象
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hello, 匿名内部类");
            }
        };
        // 执行匿名内部类的 run() 方法
        runnable.run();
    }

}
```

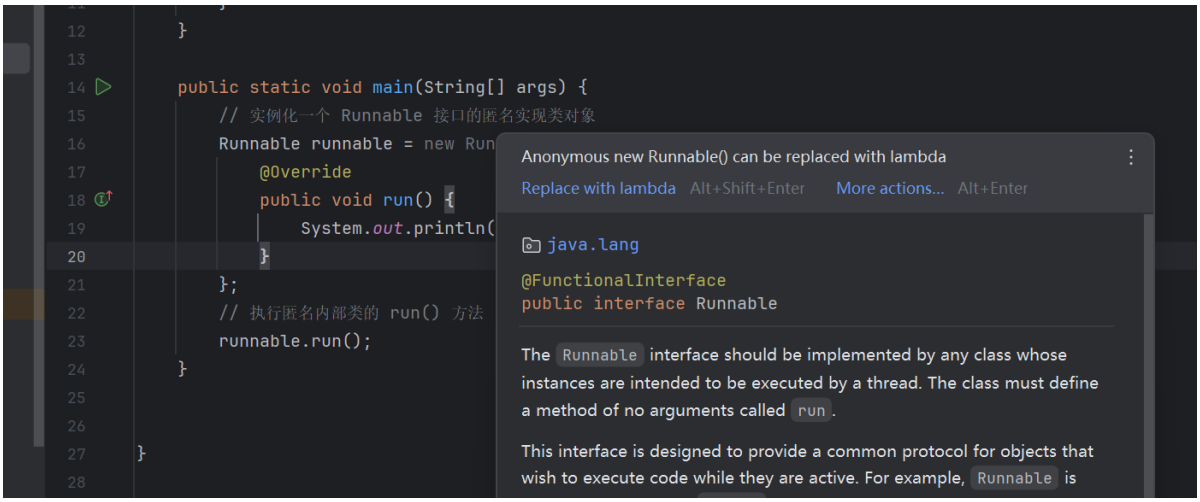
运行结果：

```
Hello, 匿名内部类
```

Lambda 表达式的应用则使代码变得更加紧凑；另外，Lambda 表达式使并行操作大集合变得很方便，

下面我们使用 `Lambda` 表达式改写上面的代码。

如果你使用 `IDEA` 编写代码，可以直接一键智能修改，首先，将鼠标光标移动到灰色的 `new Runnable()` 代码处，此时会弹出一个提示框，提示可以使用 `Lambda` 表达式替换，点击 `Replace with lambda` 按钮即可完成代码替换，截图如下：



修改后如下：

```
public class LambdaDemo1 {

    public static void main(String[] args) {

        // 实例化一个 Runnable 接口的匿名实现类对象
        Runnable runnable = () -> System.out.println("Hello, 匿名内部类");
        // 执行匿名内部类的 run() 方法
        runnable.run();
    }

}
```

运行结果：

```
Hello, 匿名内部类
```

通过对比，使用 `lambda` 表达式实现了与匿名内部类同样的功能，并且仅仅用了一行代码，代码变得更加简洁了。对于这样的写法，你可能还非常疑惑，但别担心，我们马上就来详细讲解基础语法。

3. 基础语法

Lambda 表达式由三个部分组成：

- **一个括号内用逗号分隔的形式参数列表**：实际上就是接口里面抽象方法的参数；
- **一个箭头符号**：->，这个箭头我们又称为 Lambda 操作符或箭头操作符；
- **方法体**：可以是表达式和代码块，是重写的方法的方法体。语法如下：

1. 方法体为表达式，该表达式的值作为返回值返回。

```
(parameters) -> expression
```

1. 方法体为代码块，必须用 {} 来包裹起来，且需要一个 return 返回值，但若函数式接口里面方法返回值是 void，则无需返回值。

```
(parameters) -> {
    statement1;
    statement2;
}
```

4. 使用实例

Lambda 表达式本质上就是接口实现类的对象，它简化了之前匿名内部类的冗长代码的编写。

关于 Lambda 表达式的具体使用如下。

4.1 无参数无返回值

无参数无返回值，指的是接口实现类重写的方法是无参数无返回值的，我们一开始提到的 `Runnable` 接口匿名内部类就属于此类：

实例演示

```
public class LambdaDemo2 {

    public static void main(String[] args) {
        // 通过匿名内部类实例实例化一个 Runnable 接口的实现类
        Runnable runnable1 = new Runnable() {
            @Override
            public void run() {  // 方法无形参列表，也无返回值
                System.out.println("Hello, 匿名内部类");
            }
        };
        // 执行匿名内部类的 run() 方法
        runnable1.run();

        // 无参数无返回值，通过 lambda 表达式来实例化 Runnable 接口的实现类
        Runnable runnable2 = () -> System.out.println("Hello, Lambda");
        // 执行通过 lambda 表达式实例化的对象下的 run() 方法
        runnable2.run();
    }

}
```

运行结果：

```
Hello, 匿名内部类
Hello, Lambda
```

4.2 单参数无返回值

无参数无返回值，指的是接口实现类重写的方法是单个参数，返回值为 `void` 的，实例如下：

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

public class LambdaDemo3 {

    public static void main(String[] args) {

        // 单参数无返回值
        Consumer<String> consumer1 = new Consumer<String>() {
            @Override
            public void accept(String s) {
                System.out.println(s);
            }
        };
        consumer1.accept("Hello World!");

        Consumer<String> consumer2 = (String s) -> {
            System.out.println(s);
        };
        consumer2.accept("你好，世界！");
    }

}
```

运行结果：

```
Hello World!
你好，世界！
```

4.3 省略数据类型

使用 `Lambda` 表达式可以省略掉括号中的类型，实例代码如下：

```
// 省略类型前代码
Consumer<String> consumer2 = (String s) -> {
    System.out.println(s);
};
consumer2.accept("你好，世界！");

// 省略类型后代码
Consumer<String> consumer3 = (s) -> {
    System.out.println(s);
};
consumer3.accept("你好，世界！");
```

> **Tips**：之所以能够省略括号中的数据类型，是因为我们在 `Comsumer<String>` 处已经指定了泛型，编译器可以推断出类型，后面就不用指定具体类型了。称为**类型推断**。

4.4 省略参数的小括号

当我们写的 `Lambda` 表达式只需要一个参数时，参数的小括号就可以省略，改写上面实例的代码：

```
// 省略小括号前代码
Consumer<String> consumer3 = (s) -> {
    System.out.println(s);
};
consumer3.accept("你好，世界！");
// 省略小括号后代码
Consumer<String> consumer4 = s -> {
    System.out.println(s);
};
consumer3.accept("你好，世界！");
```

观察实例代码，之前的 `(s) ->` 可以改写成 `s ->`，这样写也是合法的。

4.5 省略 return 和大括号

当 `Lambda` 表达式体只有一条语句时，如果有返回，则 return 和大括号都可以省略，实例代码如下：

实例演示

```
import java.util.Comparator;

public class LambdaDemo4 {

    public static void main(String[] args) {

        // 省略 return 和 {} 前代码
        Comparator<Integer> comparator1 = (o1, o2) -> {
            return o1.compareTo(o2);
        };
        System.out.println(comparator1.compare(12, 12));

        // 省略 return 和 {} 后代码
        Comparator<Integer> comparator2 = (o1, o2) -> o1.compareTo(o2);
        System.out.println(comparator2.compare(12, 23));

    }
}
```

运行结果：

```
0
-1
```

函数式接口

1. 什么是函数式接口

函数式接口（`Functional Interface`）的定义非常容易理解：只有一个抽象方法的接口，就是函数式接口。可以通过`Lambda`表达式来创建函数式接口的对象。

我们来看一个在之前我们就经常使用的`Runnable`接口，`Runnable`接口就是一个函数式接口。Runnable`接口中只包含一个抽象的`run()`方法，并且在接口上标注了一个`@FuncationInterface`注解，此注解就是 Java 8 新增的注解，用来标识一个函数式接口。

2. 为什么需要函数式接口

在 Java 中，`Lambda`表达式的类型是对象而不是函数，

他们必须依赖于一种特别的对象类型——函数式接口。所以说，Java 中的`Lambda`表达式就是一个函数式接口的对象。我们之前使用匿名实现类表示的对象，都可以使用`Lambda`表达式来表示。

3. 自定义函数式接口

想要自定义一个函数式接口也非常简单，在接口上做两件事即可：

1. **定义一个抽象方法**：注意，接口中只能有一个抽象方法；
2. **在接口上标记`@FunctionalInterface`注解**：当然也可以不标记，但是如果错写了多个方法，编辑器就不能自动检测你定义的函数式接口是否有问题了，所以建议还是写上吧。

```
/**
 * 自定义函数式接口
 * @author colorful@TaleLin
 */
@FunctionalInterface
public interface FunctionalInterfaceDemo {

    void run();

}
```

由于标记了`@FunctionalInterface`注解，下面接口下包含两个抽象方法的这种错误写法，编译器就会给出提示。

4.创建函数式接口对象

我们可以使用匿名内部类来创建该接口的对象，实例代码如下：

```
/**
 * 测试创建函数式接口对象
 * @author colorful@TaleLin
 */
```

```
public class Test {

    public static void main(String[] args) {
        // 使用匿名内部类方式创建函数式接口
        FunctionalInterfaceDemo functionalInterfaceDemo = new
FunctionalInterfaceDemo() {
            @Override
            public void run() {
                System.out.println("匿名内部类方式创建函数式接口");
            }
        };
        functionalInterfaceDemo.run();
    }

}
```

运行结果：

```
匿名内部类方式创建函数式接口
```

现在，我们学习了`Lambda`表达式，也可以使用`Lambda`表达式来创建，这种方法相较匿名内部类更加简洁，也更推荐这种做法。实例代码如下：

```
/**
 * 测试创建函数式接口对象
 * @author colorful@TaleLin
 */
public class Test {

    public static void main(String[] args) {
        // 使用 Lambda 表达式方式创建函数式接口
        FunctionalInterfaceDemo functionalInterfaceDemo = () ->
System.out.println("Lambda 表达式方式创建函数式接口");
        functionalInterfaceDemo.run();
    }

}
```

运行结果：

```
Lambda 表达式方式创建函数式接口
```

还有一种更笨的方法，写一个接口的实现类，通过实例化实现类来创建对象。由于比较简单，而且不符合我们学习函数式接口的初衷，

5. 内置的函数式接口介绍

Java 中内置了丰富的函数式接口，位于`java.util.function`包下。

Java 内置了 4 个核心函数式接口：

1. `Comsumer<T>` **消费型接口**：表示接受单个输入参数但不返回结果的操作，包含方法：`void accept(T t)`，可以理解为消费者，只消费（接收单个参数）、不返回（返回为`void`）；
2. `Supplier<T>` **供给型接口**：表示结果的供给者，包含方法`T get()`，可以理解为供给者，只提供（返回`T`类型对象）、不消费（不接受参数）；
3. `Function<T, R>` **函数型接口**：表示接受一个`T`类型参数并返回`R`类型结果的对象，包含方法`R apply(T t)`；
4. `Predicate<T>` **断言型接口**：确定`T`类型的对象是否满足约束，并返回`boolean`值，包含方法`boolean test(T t)`。

方法引用

1. 什么是方法引用

方法引用（`Method References`）是一种**语法糖**，它本质上就是`Lambda`表达式，我们知道`Lambda`表达式是函数式接口的实例，所以说方法引用也是函数式接口的实例。

我们来回顾一个之前学过的实例：

实例演示

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

public class MethodReferencesDemo1 {

    public static void main(String[] args) {
        Consumer<String> consumer = s -> System.out.println(s);
        consumer.accept("只消费，不返回");
    }

}
```

运行结果：

```
只消费，不返回
```

上面是 Java 内置函数式接口中的**消费型接口**，如果你使用`idea`编写代码，`System.out.println(s)`这个表达式可以一键替换为方法引用，将鼠标光标放置到语句上，会弹出提示框，再点击`Replace lambda with method reference`按钮即可完成一键替换。

替换为方法引用后的实例代码：

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

public class MethodReferencesDemo1 {

    public static void main(String[] args) {
        Consumer<String> consumer = System.out::println;
        consumer.accept("只消费，不返回");
    }

}
```

运行结果：

```
只消费，不返回
```

我们看到`System.out.println(s)`这个表达式被替换成了`System.out::println`，同样成功执行了代码。

2. 语法

方法引用使用一对冒号（`::`）来引用方法，格式如下：

```
类或对象 :: 方法名
```

上面实例中方法引用的代码为：

```
System.out::println
```

其中`System.out`就是`PrintStream`类的对象，`println`就是方法名。

3. 使用场景和使用条件

方法引用的使用场景为：当要传递给`Lambda`体的操作，已经有实现的方法了，就可以使用方法引用。

方法引用的使用条件为：接口中的抽象方法的形参列表和返回值类型与方法引用的方法形参列表和返回值相同。

4. 方法引用的分类

对于方法引用的使用，通常可以分为以下 4 种情况：

1. `对象 :: 非静态方法`：对象引用非静态方法，即实例方法；
2. `类 :: 静态方法`：类引用静态方法；
3. `类 :: 非静态方法`：类引用非静态方法；
4. `类 :: new`：构造方法引用。

下面是根据以上几种情况来看几个实例。

4.1 对象引用实例方法

对象引用实例方法，`System.out`就是对象，而`println`就是实例方法。

4.2 类引用静态方法

类引用静态方法，请查看以下实例：

实例演示

```
import java.util.Comparator;

public class MethodReferencesDemo2 {

    public static void main(String[] args) {
        // 使用 Lambda 表达式
        Comparator<Integer> comparator1 = (t1, t2) -> Integer.compare(t1, t2);
        System.out.println(comparator1.compare(11, 12));

        // 使用方法引用，类 :: 静态方法（ compare() 为静态方法）
        Comparator<Integer> comparator2 = Integer::compare;
        System.out.println(comparator2.compare(12, 11));
    }

}
```

运行结果：

```
-1
1
```

查看 Java 源码，可观察到`compare()`方法是静态方法。

4.3 类引用实例方法

类引用实例方法：

```
import java.util.Comparator;

public class MethodReferencesDemo4 {

    public static void main(String[] args) {
        // 使用 Lambda 表达式
        Comparator<String> comparator1 = (s1, s2) -> s1.compareTo(s2);
        int compare1 = comparator1.compare("Hello", "Java");
        System.out.println(compare1);
```

```
        // 使用方法引用，类 :: 实例方法（ compareTo() 为实例方法）
        Comparator<String> comparator2 = String::compareTo;
        int compare2 = comparator2.compare("Hello", "Hello");
        System.out.println(compare2);
    }

}
```

运行结果：

```
-2
0
```

`Comparator` 接口中的 `compare(T t1, T t2)` 抽象方法，有两个参数，但是 `String` 类下的实例方法 `compareTo(String anotherString)` 只有 1 个参数，为什么这种情况下还能使用方法引用呢？这属于一个特殊情况，当函数式接口中的抽象方法有两个参数时，已实现方法的第 1 个参数作为方法调用者时，也可以使用方法引用。此时，就可以使用类来**引用**实例方法了（即实例中的 `String::compareTo`）。

4.4 类引用构造方法

类引用构造方法，可以直接使用 `类名 :: new`，请查看如下实例：

实例演示

```
import java.util.function.Function;
import java.util.function.Supplier;

public class MethodReferencesDemo5 {

    static class Person {
        private String name;

        public Person() {
            System.out.println("无参数构造方法执行了");
        }

        public Person(String name) {
            System.out.println("单参数构造方法执行了");
            this.name = name;
        }

        public String getName() {
            return name;
        }

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

```
    }

    public static void main(String[] args) {

        // 使用 Lambda 表达式，调用无参构造方法
        Supplier<Person> supplier1 = () -> new Person();
        supplier1.get();

        // 使用方法引用，引用无参构造方法
        Supplier<Person> supplier2 = Person::new;
        supplier2.get();

        // 使用 Lambda 表达式，调用单参构造方法
        Function<String, Person> function1 = name -> new Person(name);
        Person person1 = function1.apply("小花");
        System.out.println(person1.getName());


        // 使用方法引用，引用单参构造方法
        Function<String, Person> function2 = Person::new;
        Person person2 = function1.apply("小明");
        System.out.println(person2.getName());
    }

}
```

运行结果：

```
无参数构造方法执行了
无参数构造方法执行了
单参数构造方法执行了
小花
单参数构造方法执行了
小明
```

在实例中，我们使用了`Lambda`表达式和方法引用两种方式，分别调用了静态内部类`Person`的无参和单参构造方法。函数式接口中的抽象方法的形参列表与构造方法的形参列表一致，抽象方法的返回值类型就是构造方法所属的类型。

Java 流式操作

流式操作，是`Java 8`除了`Lambda`表达式外的又一重大改变。是存在于java.util.stream`包下的`API`，我们称之为`Stream API，它把真正的函数式编程引入到了 Java 中。

1. 什么是 Stream

`Stream`是数据渠道，用于操作数据源所生成的元素序列，它可以实现对集合（`Collection`）的复杂操作，例如查找、替换、过滤和映射数据等操作。

我们这里说的`Stream`不同于`java`的输入输出流。另外，Collection 是一种静态的**数据结构**，存储在内存中，而`Stream`是用于计算的，通过`CPU`实现计算。注意不要混淆。

> **Tips**：`Stream`自己不会存储数据；`Stream`不会改变源对象，而是返回一个新的持有结果的`Stream`（不可变性）；`Stream`操作是延迟执行的。

2. 为什么使用 Stream API

我们在实际开发中，项目中的很多数据都来源于关系型数据库（例如 MySQL 数据库），我们使用`SQL`的条件语句就可以实现对数据的筛选、过滤等等操作；

但也有很多数据来源于非关系型数据库（`Redis`、`MongoDB`等），想要处理这些数据，往往需要在 Java 层面去处理。

使用`Stream API`对集合中的数据进行操作，就类似于 SQL 执行的数据库查询。也可以使用`Stream API`来执行并行操作。简单来说，`Stream API`提供了一种高效且易于使用的处理数据的方式。

3. 流式操作的执行流程

流式操作通常分为以下 3 个步骤：

1. **创建`Stream`对象**：通过一个数据源（例如集合、数组），获取一个流；
2. **中间操作**：一个中间的链式操作，对数据源的数据进行处理（例如过滤、排序等），直到执行终止操作才执行；
3. **终止操作**：一旦执行终止操作，就执行中间的链式操作，并产生结果。

4. Stream 对象的创建

有 4 种方式来创建`Stream`对象。

4.1 通过集合创建 Stream

Java 8 的`java.util.Collection`接口被扩展，提供了两个获取流的默认方法：

- `default Stream<E> stream()`：返回一个串行流（顺序流）；
- `default Stream<E> parallelStream()`：返回一个并行流。

实例如下：

```
// 创建一个集合，并添加几个元素
List<String> stringList = new ArrayList<>();
stringList.add("hello");
stringList.add("world");
stringList.add("java");

// 通过集合获取串行 stream 对象
Stream<String> stream = stringList.stream();
// 通过集合获取并行 stream 对象
Stream<String> personStream = stringList.parallelStream();
```

串行流并行流的区别是：串行流从集合中取数据是按照集合的顺序的；而并行流是并行操作的，获取到的数据是无序的。

4.2 通过数组创建 Stream

Java 8 中的 `java.util.Arrays` 的静态方法 `stream()` 可以获取数组流：

- `static <T> Stream<T> stream(T[] array)`：返回一个数组流。

此外，`stream()` 还有几个重载方法，能够处理对应的基本数据类型的数组：

- `public static IntStream stream(int[] array)`：返回以指定数组作为其源的连续 `IntStream`；
- `public static LongStream stream(long[] array)`：返回以指定数组作为其源的连续 `LongStream`；
- `public static DoubleStream stream(double[] array)`：返回以指定数组作为其源的连续 `DoubleStream`。

实例如下：

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

public class StreamDemo1 {

    public static void main(String[] args) {
        // 初始化一个整型数组
        int[] arr = new int[]{1,2,3};
        // 通过整型数组，获取整形的 stream 对象
        IntStream stream1 = Arrays.stream(arr);

        // 通过字符串类型的数组，获取泛型类型为 String 的 stream 对象
        String[] stringArr = new String[]{"Hello", "World"};
        Stream<String> stream2 = Arrays.stream(stringArr);
    }
}
```

4.3 通过 Stream 的 `of()` 方法

可以通过 `Stream` 类下的 `of()` 方法来创建 Stream 对象，实例如下：

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

public class StreamDemo1 {

    public static void main(String[] args) {
        // 通过 Stream 类下的 of() 方法，创建 stream 对象、
        Stream<Integer> stream = Stream.of(1, 2, 3);
    }
}
```

4.4 创建无限流

可以使用`Stream`类下的静态方法`iterate()`以及`generate()`创建无限流：

- `public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)`：遍历；
- `public static<T> Stream<T> generate(Supplier<T> s)`：生成。

创建无限流的这种方式实际使用较少。

5. Stream 的中间操作

多个中间操作可以连接起来形成一个流水线，除非流水线上触发终止操作，否则中间操作不会执行任何的处理。在终止操作时会一次性全部处理这些中间操作，称为“惰性求值”。下面，我们来学习一下常用的中间操作方法。

5.1 筛选与切片

关于筛选和切片中间操作，有下面几个常用方法：

- `filter(Predicate p)`：接收`Lambda`，从流中清除某些元素；
- `distinct()`：筛选，通过流生成元素的`hashCode`和`equals()`方法去除重复元素；
- `limit(long maxSize)`：截断流，使其元素不超过给定数量；
- `skip(long n)`：跳过元素，返回一个扔掉了前`n`个元素的流。若流中元素不足`n`个，则返回一个空流。与`limit(n)`互补。

先来看一个过滤集合元素的实例：

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

public class StreamDemo2 {

    static class Person {
        private String name;
```

```
        private int age;

        public Person() { }

        public Person(String name, int age) {
            this.name = name;
            this.age = age;
        }

        public String getName() {
            return name;
        }

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

        public int getAge() {
            return age;
        }

        public void setAge(int age) {
            this.age = age;
        }

        @Override
        public String toString() {
            return "Person{" +
                    "name='" + name + '\'' +
                    ", age=" + age +
                    '}';
        }
    }

    /**
     * 创建一个Person的集合
     * @return List
     */
    public static List<Person> createPeople() {
        ArrayList<Person> people = new ArrayList<>();
        Person person1 = new Person("小明", 15);
        Person person2 = new Person("小芳", 20);
        Person person3 = new Person("小李", 18);
        Person person4 = new Person("小付", 23);
        Person person5 = new Person("大飞", 22);
        people.add(person1);
        people.add(person2);
        people.add(person3);
        people.add(person4);
        people.add(person5);
        return people;
    }

    public static void main(String[] args) {
        List<Person> people = createPeople();
        // 创建 Stream 对象
```

```
        Stream<Person> stream = people.stream();
        // 过滤年龄大于 20 的person
        Stream<Person> personStream = stream.filter(person -> person.getAge() >=
20);
        // 触发终止操作才能执行中间操作，遍历列表中元素并打印
        personStream.forEach(System.out::println);
    }
}
```

运行结果：

```
Person{name='小芳', age=20}
Person{name='小付', age=23}
Person{name='大飞', age=22}
```

实例中，有一个静态内部类 `Person` 以及一个创建 `Person` 的集合的静态方法 `createPeople()`，在主方法中，我们先调用该静态方法获取到一个 `Person` 列表，然后创建了 `Stream` 对象，再执行中间操作（即调用 `fliter()` 方法），这个方法的参数类型是一个**断言型的函数式接口**，接口下的抽象方法 `test()` 要求返回 `boolean` 结果，因此我们使用 `Lambda` 表达式，`Lambda` 体为 `person.getAge() >= 20`，其返回值就是一个布尔型结果，这样就实现了对年龄大于等于 20 的 `person` 对象的过滤。

由于必须触发终止操作才能执行中间操作，我们又调用了 `forEach(System.out::println)`，在这里记住它作用是遍历该列表并打印每一个元素即可，我们下面将会讲解。另外，`filter()` 等这些由于中间操作返回类型为 Stream，所以支持链式操作，我们可以将主方法中最后两行代码合并成一行：

```
stream.filter(person -> person.getAge() >= 20).forEach(System.out::println);
```

我们再来看一个截断流的使用实例：

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

public class StreamDemo3 {

    static class Person {
        private String name;
        private int age;

        public Person() { }

        public Person(String name, int age) {
            this.name = name;
            this.age = age;
        }

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

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

        public int getAge() {
            return age;
        }

        public void setAge(int age) {
            this.age = age;
        }

        @Override
        public String toString() {
            return "Person{" +
                    "name='" + name + '\'' +
                    ", age=" + age +
                    '}';
        }
    }

    /**
     * 创建一个Person的集合
     * @return List
     */
    public static List<Person> createPeople() {
        ArrayList<Person> people = new ArrayList<>();
        Person person1 = new Person("小明", 15);
        Person person2 = new Person("小芳", 20);
        Person person3 = new Person("小李", 18);
        Person person4 = new Person("小付", 23);
        Person person5 = new Person("大飞", 22);
        people.add(person1);
        people.add(person2);
        people.add(person3);
        people.add(person4);
        people.add(person5);
        return people;
    }

    public static void main(String[] args) {
        List<Person> people = createPeople();
        // 创建 Stream 对象
        Stream<Person> stream = people.stream();
        // 截断流，并调用终止操作打印集合中元素
        stream.limit(2).forEach(System.out::println);
    }
}
```

运行结果：

```
Person{name='小明', age=15}
Person{name='小芳', age=20}
```

根据运行结果显示，我们只打印了集合中的前两条数据。

跳过前 2 条数据的代码实例如下：

```
// 非完整代码
public static void main(String[] args) {
    List<Person> people = createPeople();
    // 创建 Stream 对象
    Stream<Person> stream = people.stream();
    // 跳过前两个元素，并调用终止操作打印集合中元素
    stream.skip(2).forEach(System.out::println);
}
```

运行结果：

```
Person{name='小李', age=18}
Person{name='小付', age=23}
Person{name='大飞', age=22}
```

`distinct()` 方法会根据 `equals()` 和 `hashCode()` 方法筛选重复数据，我们在 `Person` 类内部重写这两个方法，并且在 `createPerson()` 方法中，添加几个重复的数据 ， 实例如下：

实例演示

```
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;
import java.util.stream.Stream;

public class StreamDemo4 {

    static class Person {
        private String name;
        private int age;

        public Person() { }

        public Person(String name, int age) {
            this.name = name;
            this.age = age;
        }

        public String getName() {
            return name;
        }

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

        public int getAge() {
            return age;
        }
```

```
        public void setAge(int age) {
            this.age = age;
        }

        @Override
        public String toString() {
            return "Person{" +
                    "name='" + name + '\'' +
                    ", age=" + age +
                    '}';
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;
            Person person = (Person) o;
            return age == person.age &&
                    Objects.equals(name, person.name);
        }

        @Override
        public int hashCode() {
            return Objects.hash(name, age);
        }
    }

    /**
     * 创建一个Person的集合
     * @return List
     */
    public static List<Person> createPeople() {
        ArrayList<Person> people = new ArrayList<>();
        people.add(new Person("小明", 15));
        people.add(new Person("小芳", 20));
        people.add(new Person("小李", 18));
        people.add(new Person("小付", 23));
        people.add(new Person("小付", 23));
        people.add(new Person("大飞", 22));
        people.add(new Person("大飞", 22));
        people.add(new Person("大飞", 22));
        return people;
    }

    public static void main(String[] args) {
        List<Person> people = createPeople();
        // 创建 Stream 对象
        Stream<Person> stream = people.stream();

        System.out.println("去重前，集合中元素有：");
        stream.forEach(System.out::println);

        System.out.println("去重后，集合中元素有：");
        // 创建一个新流
        Stream<Person> stream1 = people.stream();
        // 截断流，并调用终止操作打印集合中元素
```

```
        stream1.distinct().forEach(System.out::println);
    }
}
```

运行结果：

```
去重前，集合中元素有：
Person{name='小明', age=15}
Person{name='小芳', age=20}
Person{name='小李', age=18}
Person{name='小付', age=23}
Person{name='小付', age=23}
Person{name='大飞', age=22}
Person{name='大飞', age=22}
Person{name='大飞', age=22}
去重后，集合中元素有：
Person{name='小明', age=15}
Person{name='小芳', age=20}
Person{name='小李', age=18}
Person{name='小付', age=23}
Person{name='大飞', age=22}
```

5.2 映射

关于映射中间操作，有下面几个常用方法：

- `map(Function f)`：接收一个方法作为参数，该方法会被应用到每个元素上，并将其映射成一个新的元素；
- `mapToDouble(ToDoubleFunction f)`：接收一个方法作为参数，该方法会被应用到每个元素上，产生一个新的`DoubleStream`；
- `mapToLong(ToLongFunction f)`：接收一个方法作为参数，该方法会被应用到每个元素上，产生一个新的`LongStream`；
- `flatMap(Function f)`：接收一个方法作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。

请查看如下实例：

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

public class StreamDemo5 {

    public static void main(String[] args) {
        // 创建一个包含小写字母元素的字符串列表
        List<String> stringList = Arrays.asList("php", "js", "python", "java");
        // 调用 map() 方法，将String下的toUpperCase()方法作为参数，这个方法会被应用到每个元素上，映射成一个新元素，最后打印映射后的元素

 stringList.stream().map(String::toUpperCase).forEach(System.out::println);
    }

}
```

运行结果：

```
PHP
JS
PYTHON
JAVA
```

5.3 排序

关于排序中间操作，有下面几个常用方法：

- `sorted()`：产生一个新流，其中按照自然顺序排序；
- `sorted(Comparator com)`：产生一个新流，其中按照比较器顺序排序。

请查看如下实例：

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

public class StreamDemo6 {

    public static void main(String[] args) {
        List<Integer> integers = Arrays.asList(10, 12, 9, 8, 20, 1);
        // 调用sorted()方法自然排序，并打印每个元素
        integers.stream().sorted().forEach(System.out::println);
    }

}
```

运行结果：

```
1
8
9
10
12
20
```

上面实例中，我们调用`sorted()`方法对集合元素进行了从小到大的自然排序，那么如果想要实现从大到小排序，任何实现呢？此时就要用到`sorted(Comparator com)`方法定制排序，查看如下实例：

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

public class StreamDemo6 {

    public static void main(String[] args) {
        List<Integer> integers = Arrays.asList(10, 12, 9, 8, 20, 1);
        // 定制排序
        integers.stream().sorted(
                (i1, i2) -> -Integer.compare(i1, i2)
        ).forEach(System.out::println);
    }

}
```

运行结果：

```
20
12
10
9
8
1
```

实例中，`sorted()`方法接收的参数是一个函数式接口`Comparator`，因此使用`Lambda`表达式创建函数式接口实例即可，`Lambda`体调用整型的比较方法，对返回的整型值做一个取反即可。

6. Stream 的终止操作

执行终止操作会从流的**流水线**上生成结果，其结果可以是任何不是流的值，例如`List`、`String`、`void`。

在上面实例中，我们一直在使用`forEach()`方法来执行流的终止操作，下面我们看看还有哪些其他终止操作。

6.1 匹配与查找

关于匹配与查找的终止操作，有下面几个常用方法：

- `allMatch(Predicate p)`：检查是否匹配所有元素；
- `anyMatch(Predicate p)`：检查是否至少匹配一个元素；
- `noneMatch(Predicate p)`：检查是否没有匹配所有元素；
- `findFirst()`：返回第一个元素；
- `findAny()`：返回当前流中的任意元素；
- `count()`：返回流中元素总数；
- `max(Comparator c)`：返回流中最大值；
- `min(Comparator c)`：返回流中最小值；
- `forEach(Consumer c)`：内部迭代（使用 Collection 接口需要用户去做迭代，称为外部迭代；相反 `Stream API`使用内部迭代）。

如下实例，演示了几个匹配元素相关方法的使用：

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

public class StreamDemo7 {

    public static void main(String[] args) {
        // 创建一个整型列表
        List<Integer> integers = Arrays.asList(10, 12, 9, 8, 20, 1);
        // 使用 allMatch(Predicate p) 检查是否匹配所有元素，如果匹配，则返回 true；否则返回 false
        boolean b1 = integers.stream().allMatch(integer -> integer > 0);
        if (b1) {
            System.out.println(integers + "列表中所有的元素都大于0");
        } else {
            System.out.println(integers + "列表中不是所有的元素都大于0");
        }

        // 使用 anyMatch(Predicate p) 检查是否至少匹配一个元素
        boolean b2 = integers.stream().anyMatch(integer -> integer >= 20);
        if (b2) {
            System.out.println(integers + "列表中至少存在一个的元素都大于等于20");
        } else {
            System.out.println(integers + "列表中不存在任何一个大于等于20的元素");
        }

        // 使用 noneMath(Predicate p) 检查是否没有匹配所有元素
        boolean b3 = integers.stream().noneMatch(integer -> integer > 100);
        if (b3) {
            System.out.println(integers + "列表中不存在大于100的元素");
        } else {
            System.out.println(integers + "列表中存在大于100的元素");
        }
    }
```

```
}
```

运行结果：

```
[10, 12, 9, 8, 20, 1]列表中所有的元素都大于0
[10, 12, 9, 8, 20, 1]列表中至少存在一个的元素都大于等于20
[10, 12, 9, 8, 20, 1]列表中不存在大于100的元素
```

查找元素的相关方法使用实例如下：

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

public class StreamDemo8 {

    public static void main(String[] args) {
        // 创建一个整型列表
        List<Integer> integers = Arrays.asList(10, 12, 9, 8, 20, 1);

        // 使用 findFirst() 获取当前流中的第一个元素
        Optional<Integer> first = integers.stream().findFirst();
        System.out.println(integers + "列表中第一个元素为：" + first);

        // 使用 findAny() 获取当前流中的任意元素
        Optional<Integer> any = integers.stream().findAny();
        System.out.println("列表中任意元素：" + any);

        // 使用 count() 获取当前流中元素总数
        long count = integers.stream().count();
        System.out.println(integers + "列表中元素总数为" + count);

        // 使用 max(Comparator c) 获取流中最大值
        Optional<Integer> max = integers.stream().max(Integer::compare);
        System.out.println(integers + "列表中最大值为" + max);

        // 使用 min(Comparator c) 获取流中最小值
        Optional<Integer> min = integers.stream().min(Integer::compare);
        System.out.println(integers + "列表中最小值为" + min);
    }

}
```

运行结果：

```
[10, 12, 9, 8, 20, 1]列表中第一个元素为：Optional[10]
列表中任意元素：Optional[10]
[10, 12, 9, 8, 20, 1]列表中元素总数为6
[10, 12, 9, 8, 20, 1]列表中最大值为Optional[20]
[10, 12, 9, 8, 20, 1]列表中最小值为Optional[1]
```

实例中，我们观察到`findFirst()`、`findAny()`、`max()`等方法的返回值类型为`Optional`类型

6.2 归约

关于归约的终止操作，有下面几个常用方法：

- `reduce(T identity, BinaryOperator b)`：可以将流中的元素反复结合起来，得到一个值。返回 T；
- `reduce(BinaryOperator b)`：可以将流中的元素反复结合起来，得到一个值，返回`Optional<T>`。

归约相关方法的使用实例如下：

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

public class StreamDemo9 {

    public static void main(String[] args) {
        // 创建一个整型列表
        List<Integer> integers = Arrays.asList(10, 12, 9, 8, 20, 1);

        // 使用 reduce(T identity, BinaryOperator b) 计算列表中所有整数和
        Integer sum = integers.stream().reduce(0, Integer::sum);
        System.out.println(sum);

        // 使用 reduce(BinaryOperator b) 计算列表中所有整数和，返回一个 Optional<T>
        Optional<Integer> reduce = integers.stream().reduce(Integer::sum);
        System.out.println(reduce);
    }

}
```

运行结果：

```
60
Optional[60]
```

6.3 收集

`collect(Collector c)`：将流转换为其他形式。接收一个`Collector`接口的实现，用于给`Stream`中元素做汇总的方法。

实例如下：

实例演示

```
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class StreamDemo10 {

    public static void main(String[] args) {
        // 创建一个整型列表
        List<Integer> integers = Arrays.asList(10, 12, 9, 8, 20, 1, 10);
        Set<Integer> collect = integers.stream().collect(Collectors.toSet());
        System.out.println(collect);
    }

}
```

运行结果：

```
[1, 20, 8, 9, 10, 12]
```

Collector 接口中的实现决定了如何对流执行收集的操作（如收集到 List、Set、Map）。`java.util.stream.Collectors`类提供了很多静态方法，可以方便地创建常用收集器实例，常用静态方法如下：

- `static List<T> toList()`：把流中元素收集到`List`；
- `static Set<T> toSet()`：把流中元素收集到`Set`；
- `static Collection<T> toCollection()`：把流中元素收集到创建的集合。

Optional 类

1. Optional 类概述

为了预防空指针异常，`Google`的`Guava`项目率先引入了`Optional`类，通过使用检查空值的方式来防止代码污染，受到`Guava`项目的启发，随后在`Java 8`中也引入了`Optional`类。

Optional 类位于`java.util`包下，是一个可以为`null`的**容器对象**，如果值存在则`isPresent()`方法会返回`true`，调用`get()`方法会返回该对象，可以有效避免空指针异常。下面我们来学习如何实例化这个类，以及这个类下提供了哪些常用方法。

2. 创建 Optional 对象

查看 `java.util.Optional` 类源码，可以发现其构造方法是私有的，因此不能通过 `new` 关键字来实例化：

我们可以通过如下几种方法，来创建Optional 对象：

- `Optional.of(T t)`：创建一个 Optional 对象，参数 `t` 必须非空；
- `Optional.empty()`：创建一个空的 `Optional` 实例；
- `Optional.ofNullable(T t)`：创建一个 `Optional` 对象，参数 `t` 可以为 `null`。

实例如下：

```
import java.util.Optional;

public class OptionalDemo1 {

    public static void main(String[] args) {
        // 创建一个 StringBuilder 对象
        StringBuilder string = new StringBuilder("我是一个字符串");

        // 使用 Optional.of(T t) 方法，创建 Optional 对象，注意 T 不能为空：
        Optional<StringBuilder> stringBuilderOptional = Optional.of(string);
        System.out.println(stringBuilderOptional);

        // 使用 Optional.empty() 方法，创建一个空的 Optional 对象：
        Optional<Object> empty = Optional.empty();
        System.out.println(empty);

        // 使用 Optional.ofNullable(T t) 方法，创建 Optional 对象，注意 t 允许为空：
        stringBuilderOptional = null;
        Optional<Optional<StringBuilder>> stringBuilderOptional1 =
Optional.ofNullable(stringBuilderOptional);
        System.out.println(stringBuilderOptional1);
    }

}
```

运行结果：

```
Optional[我是一个字符串]
Optional.empty
Optional.empty
```

3. 常用方法

`Optional<T>` 类提供了如下常用方法：

- `booean isPresent()`：判断是否包换对象；
- `void ifPresent(Consumer<? super T> consumer)`：如果有值，就执行 Consumer 接口的实现代码，并且该值会作为参数传递给它；
- `T get()`：如果调用对象包含值，返回该值，否则抛出异常；
- `T orElse(T other)`：如果有值则将其返回，否则返回指定的`other`对象；
- `T orElseGet(Supplier<? extends T other>)`：如果有值则将其返回，否则返回由`Supplier`接口实现提供的对象；
- `T orElseThrow(Supplier<? extends X> exceptionSupplier)`：如果有值则将其返回，否则抛出由`Supplier`接口实现提供的异常。

知道了如何创建`Optional`对象和常用方法，我们下面结合具体实例来看一下，`Optional`类是如何避免空指针异常的。

下面的实例，其在运行时会发生空指针异常：

```
import java.util.Optional;

public class OptionalDemo2 {

    static class Category {
        private String name;

        public Category(String name) {
            this.name = name;
        }

        public String getName() {
            return name;
        }

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

        @Override
        public String toString() {
            return "Category{" +
                    "name='" + name + '\'' +
                    '}';
        }
    }

    static class Goods {
        private String name;

        private Category category;

        public Goods() {

        }
```

```
        public Goods(String name, Category category) {
            this.name = name;
            this.category = category;
        }

        public String getName() {
            return name;
        }

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

        public Category getCategory() {
            return category;
        }

        public void setCategory(Category category) {
            this.category = category;
        }

        @Override
        public String toString() {
            return "Good{" +
                    "name='" + name + '\'' +
                    ", category=" + category +
                    '}';
        }
    }

    /**
     * 获取商品的分类名称
     * @param goods 商品
     * @return 分类名称
     */
    static String getGoodsCategoryName(Goods goods) {
        return goods.getCategory().getName();
    }

    public static void main(String[] args) {
        // 实例化一个商品类
        Goods goods = new Goods();
        // 获取商品的分类名称
        String categoryName = getGoodsCategoryName(goods);
        System.out.println(categoryName);
    }
}
```

运行结果：

```
Exception in thread "main" java.lang.NullPointerException
    at OptionalDemo2.getGoodsCategoryName(OptionalDemo2.java:73)
    at OptionalDemo2.main(OptionalDemo2.java:80)
```

实例中，由于在实例化`Goods`类时，我们没有给其下面的`Category`类型的属性`category`赋值，它就为`null`，在运行时，`null.getName()`就会抛出空指针异常。同理，如果`goods`实例为`null`，那么`null.getCategory()`也会抛出空指针异常。

在没有使用`Optional`类的情况下，想要优化代码，就不得不改写`getGoodsCategoryName()`方法：

```
static String getGoodsCategoryName(Goods goods) {
    if (goods != null) {
        Category category = goods.getCategory();
        if (category != null) {
            return category.getName();
        }
    }
    return "该商品无分类";
}
```

这也就是我们上面说的`null`检查逻辑代码，此处有两层`if`嵌套，如果有更深层次的级联属性，就要嵌套更多的层级。

下面我们将`Optional`类引入实例代码：

```
import java.util.Optional;

public class OptionalDemo3 {

    static class Category {
        private String name;

        public Category(String name) {
            this.name = name;
        }

        public String getName() {
            return name;
        }

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

        @Override
        public String toString() {
            return "Category{" +
                    "name='" + name + '\'' +
                    '}';
        }
    }

    static class Goods {
        private String name;
```

```
        private Category category;

        public Goods() {

        }

        public Goods(String name, Category category) {
            this.name = name;
            this.category = category;
        }

        public String getName() {
            return name;
        }

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

        public Category getCategory() {
            return category;
        }

        public void setCategory(Category category) {
            this.category = category;
        }

        @Override
        public String toString() {
            return "Good{" +
                    "name='" + name + '\'' +
                    ", category=" + category +
                    '}';
        }
    }

    /**
     * 获取商品的分类名称（使用 Optional 类包装）
     * @param goods 商品
     * @return 分类名称
     */
    static String getGoodsCategoryName(Goods goods) {
        // 将商品实例包装入 Optional 类，创建 Optional<Goods> 对象
        Optional<Goods> goodsOptional = Optional.ofNullable(goods);
        Goods goods1 = goodsOptional.orElse(new Goods("默认商品", new Category("默
认分类")));
        // 此时 goods1 一定是非空，不会产生空指针异常
        Category category = goods1.getCategory();

        // 将分类实例包装入 Optional 类，创建 Optional<Category> 对象
        Optional<Category> categoryOptional = Optional.ofNullable(category);
        Category category1 = categoryOptional.orElse(new Category("默认分类"));
        // 此时 category1 一定是非空，不会产生空指针异常
        return category1.getName();
    }
```

```
    public static void main(String[] args) {
        // 实例化一个商品类
        Goods goods = null;
        // 获取商品的分类名称
        String categoryName = getGoodsCategoryName(goods);
        System.out.println(categoryName);
    }
}
```

运行结果：

```
默认分类
```

实例中，我们使用`Optional`类的`ofNullable（T t）`方法分别包装了`goods`对象及其级联属性`category`对象，允许对象为空，然后又调用了其`ofElse(T t)`方法保证了对象一定非空。这样，空指针异常就被我们优雅地规避掉了。

4. 对于空指针异常的改进

Java 14 对于空指针异常有了一些改进，它提供了更明确异常堆栈打印信息，JVM 将精确地确定那个变量是`null`，不过空指针异常依然无法避免。明确的异常堆栈信息，能够帮助开发者快速定位错误发生的位置。