

Java新特新

主讲人：图灵学院-无忌老师

JDK版本	名称	发布时间
1.0	Oak(橡树)	1996-01-23
1.1		1997-02-19
1.2	Playground (运动场)	1998-12-04
1.3	Kestrel (美洲红隼)	2000-05-08
1.4.0	Merlin (灰背隼)	2002-02-13
Java SE 5.0 / 1.5	Tiger (老虎)	2004-09-30
Java SE 6.0 / 1.6	Mustang (野马)	2006-04
Java SE 7.0 / 1.7	Dolphin (海豚)	2011-07-28
Java SE 8.0 / 1.8	Spider (蜘蛛)	2014-03-18
Java SE 9.0		2017-09-21
Java SE 10.0		2018-03-21
Java SE 11.0		2018-09-25

JDK1.4 正则表达式，异常链，NIO，日志类，XML解析器，XLST转换器

JDK1.5 自动装箱、泛型、动态注解、枚举、可变长参数、遍历循环

JDK1.6 提供动态语言支持、提供编译API和卫星HTTP服务器API，改进JVM的锁，同步垃圾回收，类加载

JDK1.7 提供GI收集器、加强对非Java语言的调用支持 (JSR-292,升级类加载架构)

JDK1.8 Lambda 表达式、方法引用、默认方法、新工具、Stream API、Date Time API、Optional 类、Nashorn, JavaScript 引擎

一、接口的默认方法

Java 8允许我们给接口添加一个非抽象的方法实现，只需要使用 default关键字即可，这个特征又叫做扩展方法，示例如下：

```
interface Formula {
    double calculate(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

Formula接口在拥有calculate方法之外同时还定义了sqrt方法，实现了Formula接口的子类只需要实现一个calculate方法，默认方法sqrt将在子类上可以直接使用。

```
Formula formula = new Formula() {
    @Override
    public double calculate(int a) {
        return sqrt(a * 100);
    }
};

formula.calculate(100);    // 100.0
formula.sqrt(16);        // 4.0
```

文中的formula被实现为一个匿名类的实例，该代码非常容易理解，6行代码实现了计算 $\text{sqrt}(a * 100)$ 。在下一节中，我们将会看到实现单方法接口的更简单的做法。

译者注：在Java中只有单继承，如果要让一个类赋予新的特性，通常是使用接口来实现，在C++中支持多继承，允许一个子类同时具有多个父类的接口与功能，在其他语言中，让一个类同时具有其他的可复用代码的方法叫做mixin。新的Java 8的这个特新在编译器实现的角度上来说更加接近Scala的trait。在C#中也有名为扩展方法的概念，允许给已存在的类型扩展方法，和Java 8的这个在语义上有差别。

二、Lambda 表达式

首先看看在老版本的Java中是如何排列字符串的：

```
List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");

Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
});
```

只需要给静态方法 `Collections.sort` 传入一个List对象以及一个比较器来按指定顺序排列。通常做法都是创建一个匿名的比较器对象然后将其传递给sort方法。

在Java 8 中你就没必要使用这种传统的匿名对象的方式了，Java 8提供了更简洁的语法，lambda表达式：

```
Collections.sort(names, (String a, String b) -> {
    return b.compareTo(a);
});
```

看到了吧，代码变得更短且更具有可读性，但是实际上还可以写得更短：

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

对于函数体只有一行代码的，你可以去掉大括号{}以及return关键字，但是你还可以写得更短点：

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Java编译器可以自动推导出参数类型，所以你可以不用再写一次类型。接下来我们看看lambda表达式还能作出什么更方便的东西来

三、函数式接口

Lambda表达式是如何在java的类型系统中表示的呢？每一个lambda表达式都对应一个类型，通常是接口类型。而“函数式接口”是指仅仅只包含一个抽象方法的接口，每一个该类型的lambda表达式都会被匹配到这个抽象方法。因为 默认方法 不算抽象方法，所以你也可以给你的函数式接口添加默认方法。

我们可以将lambda表达式当作任意只包含一个抽象方法的接口类型，确保你的接口一定达到这个要求，你只需要给你的接口添加 `@FunctionalInterface` 注解，编译器如果发现你标注了这个注解的接口有多于一个抽象方法的时候会报错的。

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}
Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
Integer converted = converter.convert("123");
System.out.println(converted);    // 123
```

需要注意如果`@FunctionalInterface`如果没有指定，上面的代码也是对的。

译者注 将lambda表达式映射到一个单方法的接口上，这种做法在Java 8之前就有别的语言实现，比如Rhino JavaScript解释器，如果一个函数参数接收一个单方法的接口而你传递的是一个function，Rhino解释器会自动做一个单接口的实例到function的适配器，典型的应用场景有org.w3c.dom.events.EventTarget的addEventListener 第二个参数EventListener。

四、方法与构造函数引用

前一节中的代码还可以通过静态方法引用来表示：

```
Converter<String, Integer> converter = Integer::valueOf;
Integer converted = converter.convert("123");
System.out.println(converted);    // 123
```

Java 8 允许你使用 `::` 关键字来传递方法或者构造函数引用，上面的代码展示了如何引用一个静态方法，我们也可以引用一个对象的方法：

```
converter = something::startsWith;
String converted = converter.convert("Java");
System.out.println(converted);    // "J"
```

接下来看看构造函数是如何使用`::`关键字来引用的，首先我们定义一个包含多个构造函数的简单类：

```
class Person {
    String firstName;
    String lastName;

    Person() {}

    Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

接下来我们指定一个用来创建Person对象的对象工厂接口：

```
interface PersonFactory<P extends Person> {
    P create(String firstName, String lastName);
}
```

这里我们使用构造函数引用来将他们关联起来，而不是实现一个完整的工厂：

```
PersonFactory<Person> personFactory = Person::new;
Person person = personFactory.create("Peter", "Parker");
```

我们只需要使用 `Person::new` 来获取Person类构造函数的引用，Java编译器会自动根据 `PersonFactory.create` 方法的签名来选择合适的构造函数。

五、Lambda 作用域

在lambda表达式中访问外层作用域和老版本的匿名对象中的方式很相似。你可以直接访问标记了final的外层局部变量，或者实例的字段以及静态变量。

六、访问局部变量

我们可以直接在lambda表达式中访问外层的局部变量：

```
final int num = 1;
Converter<Integer, String> stringConverter =
    (from) -> String.valueOf(from + num);

stringConverter.convert(2);    // 3
```

但是和匿名对象不同的是，这里的变量num可以不用声明为final，该代码同样正确：

```
int num = 1;
Converter<Integer, String> stringConverter =
    (from) -> String.valueOf(from + num);

stringConverter.convert(2);    // 3
```

不过这里的num必须不可被后面的代码修改（即隐性的具有final的语义），例如下面的就无法编译：

```
int num = 1;
Converter<Integer, String> stringConverter =
    (from) -> String.valueOf(from + num);
num = 3;
```

在lambda表达式中试图修改num同样是不允许的。

七、访问对象字段与静态变量

和本地变量不同的是，lambda内部对于实例的字段以及静态变量是即可读又可写。该行为和匿名对象是一致的：

```
class Lambda4 {
    static int outerStaticNum;
    int outerNum;

    void testScopes() {
        Converter<Integer, String> stringConverter1 = (from) -> {
            outerNum = 23;
            return String.valueOf(from);
        };

        Converter<Integer, String> stringConverter2 = (from) -> {
            outerStaticNum = 72;
            return String.valueOf(from);
        };
    }
}
```

八、访问接口的默认方法

还记得第一节中的formula例子么，接口Formula定义了一个默认方法sqrt可以直接被formula的实例包括匿名对象访问到，但是在lambda表达式中这个是不行的。

Lambda表达式中是无法访问到默认方法的，以下代码将无法编译：

```
Formula formula = (a) -> sqrt( a * 100);
Built-in Functional Interfaces
```

JDK 1.8 API包含了很多内建的函数式接口，在老Java中常用到的比如Comparator或者Runnable接口，这些接口都增加了@FunctionalInterface注解以便能用在lambda上。

Java 8 API同样还提供了很多全新的函数式接口来让工作更加方便，有一些接口是来自Google Guava库里的，即便你对这些很熟悉了，还是有必要看看这些是如何扩展到lambda上使用的。

Predicate接口

Predicate 接口只有一个参数，返回boolean类型。该接口包含多种默认方法来将Predicate组合成其他复杂的逻辑（比如：与，或，非）：

```
Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo");           // true
predicate.negate().test("foo");  // false

Predicate<Boolean> nonNull = Objects::nonNull;
Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isEmpty = isEmpty.negate();
```

Function 接口

Function 接口有一个参数并且返回一个结果，并附带了一些可以和其他函数组合的默认方法（compose, andThen）：

```
Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123");      // "123"
```

Supplier 接口

Supplier 接口返回一个任意范型的值，和Function接口不同的是该接口没有任何参数

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get();           // new Person
```

Consumer 接口

Consumer 接口表示执行在单个参数上的操作。

```
Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);
greeter.accept(new Person("Luke", "Skywalker"));
```

Comparator 接口

Comparator 是老Java中的经典接口，Java 8在此之上添加了多种默认方法：

```

Comparator<Person> comparator = (p1, p2) ->
p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe");
Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2);           // > 0
comparator.reversed().compare(p1, p2); // < 0

```

Optional 接口

Optional 不是函数是接口，这是个用来防止NullPointerException异常的辅助类型，这是下一届中将要用到的重要概念，现在先简单的看看这个接口能干什么：

Optional 被定义为一个简单的容器，其值可能是null或者不是null。在Java 8之前一般某个函数应该返回非空对象但是偶尔却可能返回了null，而在Java 8中，不推荐你返回null而是返回Optional。

```

Optional<String> optional = Optional.of("bam");

optional.isPresent();           // true
optional.get();                 // "bam"
optional.orElse("fallback");    // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0))); // "b"

```

Stream 接口

java.util.Stream 表示能应用在—组元素上—次执行的操作序列。Stream 操作分为中间操作或者最终操作两种，最终操作返回—特定类型的计算结果，而中间操作返回Stream本身，这样你就可以将多个操作依次串起来。Stream 的创建需要指定一个数据源，比如 java.util.Collection的子类，List或者Set，Map不支持。Stream的操作可以串行执行或者并行执行。

首先看看Stream是怎么用，首先创建实例代码的用到的数据List：

```

List<String> stringCollection = new ArrayList<>();
stringCollection.add("ddd2");
stringCollection.add("aaa2");
stringCollection.add("bbb1");
stringCollection.add("aaa1");
stringCollection.add("bbb3");
stringCollection.add("ccc");
stringCollection.add("bbb2");
stringCollection.add("ddd1");

```

Java 8扩展了集合类，可以通过 Collection.stream() 或者 Collection.parallelStream() 来创建一个Stream。下面几节将详细解释常用的Stream操作：

Filter 过滤

过滤通过一个predicate接口来过滤并只保留符合条件的元素，该操作属于中间操作，所以我们可以过滤后的结果来应用其他Stream操作（比如forEach）。forEach需要一个函数来对过滤后的元素依次执行。forEach是一个最终操作，所以我们不能在forEach之后来执行其他Stream操作。

```
stringCollection
  .stream()
  .filter((s) -> s.startsWith("a"))
  .forEach(System.out::println);

// "aaa2", "aaa1"
```

Sort 排序

排序是一个中间操作，返回的是排序好后的Stream。如果你不指定一个自定义的Comparator则会使用默认排序。

```
stringCollection
  .stream()
  .sorted()
  .filter((s) -> s.startsWith("a"))
  .forEach(System.out::println);

// "aaa1", "aaa2"
```

需要注意的是，排序只创建了一个排列好后的Stream，而不会影响原有的数据源，排序之后原数据stringCollection是不会被修改的。

```
System.out.println(stringCollection);
// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1
```

Map 映射

中间操作map会将元素根据指定的Function接口来依次将元素转成另外的对象，下面的示例展示了将字符串转换为大写字符串。你也可以通过map来讲对象转换成其他类型，map返回的Stream类型是根据你map传递进去的函数的返回值决定的。

```
stringCollection
  .stream()
  .map(String::toUpperCase)
  .sorted((a, b) -> b.compareTo(a))
  .forEach(System.out::println);

// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

Match 匹配

Stream提供了多种匹配操作，允许检测指定的Predicate是否匹配整个Stream。所有的匹配操作都是最终操作，并返回一个boolean类型的值。

```
boolean anyStartsWithA =
  stringCollection
```

```

        .stream()
        .anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA);    // true

boolean allStartsWithA =
    stringCollection
        .stream()
        .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA);    // false

boolean noneStartsWithZ =
    stringCollection
        .stream()
        .noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ);    // true

```

Count 计数

计数是一个最终操作，返回Stream中元素的个数，返回值类型是long。

```

long startswithB =
    stringCollection
        .stream()
        .filter((s) -> s.startsWith("b"))
        .count();

System.out.println(startswithB);    // 3

```

Reduce 规约

这是一个最终操作，允许通过指定的函数来讲stream中的多个元素规约为一个元素，规约后的结果是通过Optional接口表示的：

```

Optional<String> reduced =
    stringCollection
        .stream()
        .sorted()
        .reduce((s1, s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"

```

并行Streams

前面提到过Stream有串行和并行两种，串行Stream上的操作是在一个线程中依次完成，而并行Stream则是在多个线程上同时执行。

下面的例子展示了是如何通过并行Stream来提升性能：

首先我们创建一个没有重复元素的大表

```
int max = 1000000;
List<String> values = new ArrayList<>(max);
for (int i = 0; i < max; i++) {
    UUID uuid = UUID.randomUUID();
    values.add(uuid.toString());
}
```

然后我们计算一下排序这个Stream要耗时多久，

串行排序：

```
long t0 = System.nanoTime();

long count = values.stream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("sequential sort took: %d ms", millis));
```

// 串行耗时: 899 ms

并行排序：

```
long t0 = System.nanoTime();

long count = values.parallelStream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("parallel sort took: %d ms", millis));
```

// 并行排序耗时: 472 ms

上面两个代码几乎是一样的，但是并行版的快了50%之多，唯一需要做的改动就是将stream()改为parallelStream()。

Map

前面提到过，Map类型不支持stream，不过Map提供了一些新的有用的方法来处理一些日常任务。

```
Map<Integer, String> map = new HashMap<>();

for (int i = 0; i < 10; i++) {
    map.putIfAbsent(i, "val" + i);
}

map.forEach((id, val) -> system.out.println(val));
```

以上代码很容易理解，putIfAbsent 不需要我们做额外的存在性检查，而forEach则接收一个Consumer接口来对map里的每一个键值对进行操作。

下面的例子展示了map上的其他有用的函数：

```
map.computeIfPresent(3, (num, val) -> val + num);
map.get(3); // val33

map.computeIfPresent(9, (num, val) -> null);
map.containsKey(9); // false

map.computeIfAbsent(23, num -> "val" + num);
map.containsKey(23); // true

map.computeIfAbsent(3, num -> "bam");
map.get(3); // val33
```

接下来展示如何在Map里删除一个键值全都匹配的项

```
map.remove(3, "val3");
map.get(3); // val33

map.remove(3, "val33");
map.get(3); // null
```

另外一个有用的方法

```
map.getOrDefault(42, "not found"); // not found
```

对Map的元素做合并也变得很容易了：

```
map.merge(9, "val9", (value, newValue) -> value.concat(newValue));
map.get(9); // val9

map.merge(9, "concat", (value, newValue) -> value.concat(newValue));
map.get(9); // val9concat
```

Merge做的事情是如果键名不存在则插入，否则则对原键对应的值做合并操作并重新插入到map中。

九、Date API

Java 8 在包java.time下包含了一组全新的时间日期API。新的日期API和开源的Joda-Time库差不多，但又不完全一样，下面的例子展示了这组新API里最重要的一些部分：

Clock 时钟

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：<https://d.book118.com/308064110125006106>