我，一个10年老程序员，最近才开始用 Java 8 新特性

> 本文来自cowbi的投稿~

Oracle 于 2014 发布了 Java8（jdk1.8），诸多原因使它成为目前市场上使用最多的 jdk 版本。虽然发布距今已将近 7 年，但很多程序员对其新特性还是不够了解，尤其是用惯了 java8 之前版本的老程序员，比如我。

为了不脱离队伍太远，还是有必要对这些新特性做一些总结梳理。它较 jdk.7 有很多变化或者说是优化，比如 interface 里可以有静态方法，并且可以有方法体，这一点就颠覆了之前的认知；`java.util.HashMap` 数据结构里增加了红黑树；还有众所周知的 Lambda 表达式等等。本文不能把所有的新特性都给大家一一分享，只列出比较常用的新特性给大家做详细讲解。更多相关内容请看官网关于 Java8 的新特性的介绍。

Interface

interface 的设计初衷是面向抽象，提高扩展性。这也留有一点遗憾，Interface 修改的时候，实现它的类也必须跟着改。

为了解决接口的修改与现有的实现不兼容的问题。新 interface 的方法可以用`default` 或 `static`修饰，这样就可以有方法体，实现类也不必重写此方法。

一个 interface 中可以有多个方法被它们修饰，这 2 个修饰符的区别主要也是普通方法和静态方法的区别。

1. `default`修饰的方法，是普通实例方法，可以用`this`调用，可以被子类继承、重写。
2. `static`修饰的方法，使用上和一般类静态方法一样。但它不能被子类继承，只能用`Interface`调用。

我们来看一个实际的例子。

```
public interface InterfaceNew {
    static void sm() {
        System.out.println("interface提供的方式实现");
    }
    static void sm2() {
        System.out.println("interface提供的方式实现");
    }

    default void def() {
        System.out.println("interface default方法");
    }
    default void def2() {
        System.out.println("interface default2方法");
    }
    //须要实现类重写
    void f();
}

public interface InterfaceNew1 {
    default void def() {
        System.out.println("InterfaceNew1 default方法");
    }
}
```

如果有一个类既实现了 `InterfaceNew` 接口又实现了 `InterfaceNew1`接口，它们都有`def()`，并且 `InterfaceNew` 接口和 `InterfaceNew1`接口没有继承关系的话，这时就必须重写`def()`。不然的话，编译的时候就会报错。

```
public class InterfaceNewImpl implements InterfaceNew , InterfaceNew1{
    public static void main(String[] args) {
        InterfaceNewImpl interfaceNew = new InterfaceNewImpl();
        interfaceNew.def();
    }

    @Override
    public void def() {
        InterfaceNew1.super.def();
    }

    @Override
    public void f() {
    }
}
```

在 Java 8， 接口和抽象类有什么区别的?

很多小伙伴认为：“既然 interface 也可以有自己的方法实现，似乎和 abstract class 没多大区别了。”

其实它们还是有区别的

1. interface 和 class 的区别，好像是废话，主要有：
    - 接口多实现，类单继承
    - 接口的方法是 public abstract 修饰，变量是 public static final 修饰。 abstract class 可以用其他修饰符
2. interface 的方法是更像是一个扩展插件。而 abstract class 的方法是要继承的。

开始我们也提到，interface 新增`default`，和`static`修饰的方法，为了解决接口的修改与现有的实现不兼容的问题，并不是为了要替代`abstract class`。在使用上，该用 abstract class 的地方还是要用 abstract class，不要因为 interface 的新特性而降之替换。

记住接口永远和类不一样。

functional interface 函数式接口

定义：也称 SAM 接口，即 Single Abstract Method interfaces，有且只有一个抽象方法，但可以有多个非抽象方法的接口。

在 java 8 中专门有一个包放函数式接口`java.util.function`，该包下的所有接口都有 `@FunctionalInterface` 注解，提供函数式编程。

在其他包中也有函数式接口，其中一些没有`@FunctionalInterface` 注解，但是只要符合函数式接口的定义就是函数式接口，与是否有

`@FunctionalInterface`注解无关，注解只是在编译时起到强制规范定义的作用。其在 Lambda 表达式中有广泛的应用。

Lambda 表达式

接下来谈众所周知的 Lambda 表达式。它是推动 Java 8 发布的最重要新特性。是继泛型(`Generics`)和注解(`Annotation`)以来最大的变化。

使用 Lambda 表达式可以使代码变的更加简洁紧凑。让 java 也能支持简单的*函数式编程*。

> Lambda 表达式是一个匿名函数，java 8 允许把函数作为参数传递进方法中。

语法格式

```
(parameters) -> expression 或
(parameters) ->{ statements; }
```

Lambda 实战

我们用常用的实例来感受 Lambda 带来的便利

替代匿名内部类

过去给方法传动态参数的唯一方法是使用内部类。比如

1.`Runnable` 接口

```
new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("The runable now is using!");
            }
}).start();
//用lambda
new Thread(() -> System.out.println("It's a lambda function!")).start();
```

2.`Comperator` 接口

```
List<Integer> strings = Arrays.asList(1, 2, 3);

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

```
});

//Lambda
Collections.sort(strings, (Integer o1, Integer o2) -> o1 - o2);
//分解开
Comparator<Integer> comperator = (Integer o1, Integer o2) -> o1 - o2;
Collections.sort(strings, comperator);
```

3.`Listener` 接口

```
JButton button = new JButton();
button.addItemListener(new ItemListener() {
@Override
public void itemStateChanged(ItemEvent e) {
   e.getItem();
}
});
//lambda
button.addItemListener(e -> e.getItem());
```

4.自定义接口

上面的 3 个例子是我们在开发过程中最常见的，从中也能体会到 Lambda 带来的便捷与清爽。它只保留实际用到的代码，把无用代码全部省略。那它对接口有没有要求呢？我们发现这些匿名内部类只重写了接口的一个方法，当然也只有一个方法须要重写。这就是我们上文提到的**函数式接口**，也就是说只要方法的参数是函数式接口都可以用 Lambda 表达式。

```
@FunctionalInterface
public interface Comparator<T>{}

@FunctionalInterface
public interface Runnable{}
```

我们自定义一个函数式接口

```
@FunctionalInterface
public interface LambdaFunctionalInterface {
 void f();
}
//使用
public class LambdaClass {
    public static void forEg() {
        lambdaInterfaceDemo(()-> System.out.println("自定义函数式接口"));
    }
    //函数式接口参数
    static void lambdaInterfaceDemo(LambdaInterface i){
        System.out.println(i);
```

```
    }
}
```

集合迭代

```
void lamndaFor() {
        List<String> strings = Arrays.asList("1", "2", "3");
        //传统foreach
        for (String s : strings) {
            System.out.println(s);
        }
        //Lambda foreach
        strings.forEach((s) -> System.out.println(s));
        //or
        strings.forEach(System.out::println);
                //map
        Map<Integer, String> map = new HashMap<>();
        map.forEach((k,v)->System.out.println(v));
}
```

方法的引用

Java 8 允许使用 :: 关键字来传递方法或者构造函数引用，无论如何，表达式返回的类型必须是 functional-interface。

```
public class LambdaClassSuper {
    LambdaInterface sf(){
        return null;
    }
}

public class LambdaClass {
    public static LambdaInterface staticF() {
        return null;
    }

    public LambdaInterface f() {
        return null;
    }

    void show() {
        //1.调用静态函数，返回类型必须是functional-interface
        LambdaInterface t = LambdaClass::staticF;

        //2.实例方法调用
        LambdaClass lambdaClass = new LambdaClass();
        LambdaInterface lambdaInterface = lambdaClass::f;
```

```
        //3.超类上的方法调用
        LambdaInterface superf = super::sf;

        //4. 构造方法调用
        LambdaInterface tt = LambdaClassSuper::new;
}
```

访问变量

```
int i = 0;
Collections.sort(strings, (Integer o1, Integer o2) -> o1 - i);
//i =3;
```

lambda 表达式可以引用外边变量，但是该变量默认拥有 final 属性，不能被修改，如果修改，编译时就报错。

Stream

java 新增了 `java.util.stream` 包，它和之前的流大同小异。之前接触最多的是资源流，比如 `java.io.FileInputStream`，通过流把文件从一个地方输入到另一个地方，它只是内容搬运工，对文件内容不做任何*CRUD*。

`Stream`依然不存储数据，不同的是它可以检索(Retrieve)和逻辑处理集合数据、包括筛选、排序、统计、计数等。可以想象成是 Sql 语句。

它的源数据可以是 `Collection`、`Array` 等。由于它的方法参数都是函数式接口类型，所以一般和 Lambda 配合使用。

流类型

1. stream 串行流
2. parallelStream 并行流，可多线程执行

常用方法

接下来我们看`java.util.stream.Stream`常用方法

```
/**
* 返回一个串行流
*/
default Stream<E> stream()

/**
* 返回一个并行流
*/
default Stream<E> parallelStream()

/**
* 返回T的流
```

```
*/
public static<T> Stream<T> of(T t)

/**
* 返回其元素是指定值的顺序流。
*/
public static<T> Stream<T> of(T... values) {
    return Arrays.stream(values);
}


/**
* 过滤，返回由与给定predicate匹配的该流的元素组成的流
*/
Stream<T> filter(Predicate<? super T> predicate);

/**
* 此流的所有元素是否与提供的predicate匹配。
*/
boolean allMatch(Predicate<? super T> predicate)

/**
* 此流任意元素是否有与提供的predicate匹配。
*/
boolean anyMatch(Predicate<? super T> predicate);

/**
* 返回一个 Stream的构建器。
*/
public static<T> Builder<T> builder();

/**
* 使用 Collector对此流的元素进行归纳
*/
<R, A> R collect(Collector<? super T, A, R> collector);

/**
 * 返回此流中的元素数。
*/
long count();

/**
* 返回由该流的不同元素（根据 Object.equals(Object) ）组成的流。
*/
Stream<T> distinct();

/**
 * 遍历
*/
void forEach(Consumer<? super T> action);

/**
* 用于获取指定数量的流，截短长度不能超过 maxSize 。
*/
```

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

/**
* 用于映射每个元素到对应的结果
*/
<R> Stream<R> map(Function<? super T, ? extends R> mapper);

/**
* 根据提供的 Comparator进行排序。
*/
Stream<T> sorted(Comparator<? super T> comparator);

/**
* 在丢弃流的第一个 n元素后，返回由该流的 n元素组成的流。
*/
Stream<T> skip(long n);

/**
* 返回一个包含此流的元素的数组。
*/
Object[] toArray();

/**
* 使用提供的 generator函数返回一个包含此流的元素的数组，以分配返回的数组，以及分区执行或调整大小可能需要的任何其他数组。
*/
<A> A[] toArray(IntFunction<A[]> generator);

/**
* 合并流
*/
public static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
```

实战

本文列出 `Stream` 具有代表性的方法之使用，更多的使用方法还是要看 Api。

```
@Test
public void test() {
  List<String> strings = Arrays.asList("abc", "def", "gkh", "abc");
    //返回符合条件的stream
    Stream<String> stringStream = strings.stream().filter(s -> "abc".equals(s));
    //计算流符合条件的流的数量
    long count = stringStream.count();

    //forEach遍历->打印元素
    strings.stream().forEach(System.out::println);

    //limit 获取到1个元素的stream
    Stream<String> limit = strings.stream().limit(1);
    //toArray 比如我们想看这个limitStream里面是什么，比如转换成String[],比如循环
```

```
        String[] array = limit.toArray(String[]::new);

        //map 对每个元素进行操作返回新流
        Stream<String> map = strings.stream().map(s -> s + "22");

        //sorted 排序并打印
        strings.stream().sorted().forEach(System.out::println);

        //Collectors collect 把abc放入容器中
        List<String> collect = strings.stream().filter(string ->
"abc".equals(string)).collect(Collectors.toList());
        //把list转为string，各元素用，号隔开
        String mergedString = strings.stream().filter(string ->
!string.isEmpty()).collect(Collectors.joining(","));

        //对数组的统计，比如用
        List<Integer> number = Arrays.asList(1, 2, 5, 4);

        IntSummaryStatistics statistics = number.stream().mapToInt((x) ->
x).summaryStatistics();
        System.out.println("列表中最大的数 : "+statistics.getMax());
        System.out.println("列表中最小的数 : "+statistics.getMin());
        System.out.println("平均数 : "+statistics.getAverage());
        System.out.println("所有数之和 : "+statistics.getSum());

        //concat 合并流
        List<String> strings2 = Arrays.asList("xyz", "jqx");
        Stream.concat(strings2.stream(),strings.stream()).count();

        //注意 一个Stream只能操作一次，不能断开，否则会报错。
        Stream stream = strings.stream();
        //第一次使用
        stream.limit(2);
        //第二次使用
        stream.forEach(System.out::println);
        //报错 java.lang.IllegalStateException: stream has already been operated upon
or closed

        //但是可以这样，连续使用
        stream.limit(2).forEach(System.out::println);
    }
```

延迟执行

在执行返回 `Stream` 的方法时，并不立刻执行，而是等返回一个非 `Stream` 的方法后才执行。因为拿到 `Stream` 并不能直接用，而是需要处理成一个常规类型。这里的 `Stream` 可以想象成是二进制流（2 个完全不一样的东东），拿到也看不懂。

我们下面分解一下 `filter` 方法。

```
@Test
public void laziness(){
  List<String> strings = Arrays.asList("abc", "def", "gkh", "abc");
  Stream<Integer> stream = strings.stream().filter(new Predicate() {
      @Override
      public boolean test(Object o) {
        System.out.println("Predicate.test 执行");
        return true;
        }
      });

   System.out.println("count 执行");
   stream.count();
}
/*-------执行结果--------*/
count 执行
Predicate.test 执行
Predicate.test 执行
Predicate.test 执行
Predicate.test 执行
```

按执行顺序应该是先打印 4 次「Predicate.test 执行」，再打印「count 执行」。实际结果恰恰相反。说明 filter 中的方法并没有立刻执行，而是等调用count()方法后才执行。

上面都是串行 Stream 的实例。并行 parallelStream 在使用方法上和串行一样。主要区别是 parallelStream 可多线程执行，是基于 ForkJoin 框架实现的，有时间大家可以了解一下 ForkJoin 框架和 ForkJoinPool。这里可以简单的理解它是通过线程池来实现的，这样就会涉及到线程安全，线程消耗等问题。下面我们通过代码来体验一下串行流的多线程执行。

```
@Test
public void parallelStreamTest(){
   List<Integer> numbers = Arrays.asList(1, 2, 5, 4);
   numbers.parallelStream() .forEach(num-
>System.out.println(Thread.currentThread().getName()+">>"+num));
}
//执行结果
main>>5
ForkJoinPool.commonPool-worker-2>>4
ForkJoinPool.commonPool-worker-11>>1
ForkJoinPool.commonPool-worker-9>>2
```

从结果中我们看到，for-each 用到的是多线程。

小结

从源码和实例中我们可以总结出一些 stream 的特点

1. 通过简单的链式编程，使得它可以方便地对遍历处理后的数据进行再处理。
2. 方法参数都是函数式接口类型

3. 一个 Stream 只能操作一次，操作完就关闭了，继续使用这个 stream 会报错。
4. Stream 不保存数据，不改变数据源

Optional

在阿里巴巴开发手册关于 Optional 的介绍中这样写到：

> 防止 NPE，是程序员的基本修养，注意 NPE 产生的场景：
>
> 1） 返回类型为基本数据类型，return 包装数据类型的对象时，自动拆箱有可能产生 NPE。
>
> 反例：public int f() { return Integer 对象}， 如果为 null，自动解箱抛 NPE。
>
> 2） 数据库的查询结果可能为 null。
>
> 3） 集合里的元素即使 isNotEmpty，取出的数据元素也可能为 null。
>
> 4） 远程调用返回对象时，一律要求进行空指针判断，防止 NPE。
>
> 5） 对于 Session 中获取的数据，建议进行 NPE 检查，避免空指针。
>
> 6） 级联调用 obj.getA().getB().getC()；一连串调用，易产生 NPE。
>
> 正例：使用 JDK8 的 Optional 类来防止 NPE 问题。

他建议使用 `Optional` 解决 NPE（`java.lang.NumberFormatException`）问题，它就是为 NPE 而生的，其中可以包含空值或非空值。下面我们通过源码逐步揭开 `Optional` 的红盖头。

假设有一个 `Zoo` 类，里面有个属性 `Dog`，需求要获取 `Dog` 的 `age`。

```
class Zoo {
    private Dog dog;
}

class Dog {
    private int age;
}
```

传统解决 NPE 的办法如下：

```
Zoo zoo = getZoo();
if(zoo != null){
    Dog dog = zoo.getDog();
    if(dog != null){
        int age = dog.getAge();
        System.out.println(age);
    }
}
```

层层判断对象分空，有人说这种方式很丑陋不优雅，我并不这么认为。反而觉得很整洁，易读，易懂。你们觉得呢?

`Optional` 是这样的实现的:

```
Optional.ofNullable(zoo).map(o -> o.getDog()).map(d -> d.getAge()).ifPresent(age -
>
    System.out.println(age)
);
```

是不是简洁了很多呢?

如何创建一个 Optional

上例中`Optional.ofNullable`是其中一种创建 Optional 的方式。我们先看一下它的含义和其他创建 Optional 的源码方法。

```
/**
* Common instance for {@code empty()}. 全局EMPTY对象
*/
private static final Optional<?> EMPTY = new Optional<>();

/**
* Optional维护的值
*/
private final T value;

/**
* 如果value是null就返回EMPTY，否则就返回of(T)
*/
public static <T> Optional<T> ofNullable(T value) {
   return value == null ? empty() : of(value);
}
/**
* 返回 EMPTY 对象
*/
public static<T> Optional<T> empty() {
   Optional<T> t = (Optional<T>) EMPTY;
   return t;
}
/**
* 返回Optional对象
*/
public static <T> Optional<T> of(T value) {
    return new Optional<>(value);
}
/**
* 私有构造方法，给value赋值
*/
private Optional(T value) {
```

```
    this.value = Objects.requireNonNull(value);
}
/**
* 所以如果of(T value) 的value是null，会抛出NullPointerException异常，这样貌似就没处理
NPE问题
*/
public static <T> T requireNonNull(T obj) {
    if (obj == null)
        throw new NullPointerException();
    return obj;
}
```

ofNullable 方法和of方法唯一区别就是当 value 为 null 时，ofNullable 返回的是EMPTY，of 会抛出 NullPointerException 异常。如果需要把 NullPointerException 暴漏出来就用 of，否则就用 ofNullable。

map()相关方法。

```
/**
* 如果value为null，返回EMPTY，否则返回Optional封装的参数值
*/
public<U> Optional<U> map(Function<? super T, ? extends U> mapper) {
        Objects.requireNonNull(mapper);
        if (!isPresent())
            return empty();
        else {
            return Optional.ofNullable(mapper.apply(value));
        }
}
/**
* 如果value为null，返回EMPTY，否则返回Optional封装的参数值，如果参数值返回null会抛
NullPointerException
*/
public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {
        Objects.requireNonNull(mapper);
        if (!isPresent())
            return empty();
        else {
            return Objects.requireNonNull(mapper.apply(value));
        }
}
```

map() 和 flatMap() 有什么区别的?

1.参数不一样，map 的参数上面看到过，flatMap 的参数是这样

```
class ZooFlat {
        private DogFlat dog = new DogFlat();
```

```
        public DogFlat getDog() {
            return dog;
        }
    }

class DogFlat {
        private int age = 1;
        public Optional<Integer> getAge() {
            return Optional.ofNullable(age);
        }
}

ZooFlat zooFlat = new ZooFlat();
Optional.ofNullable(zooFlat).map(o -> o.getDog()).flatMap(d ->
d.getAge()).ifPresent(age ->
    System.out.println(age)
);
```

2.`flatMap()` 参数返回值如果是 null 会抛 `NullPointerException`，而 `map()` 返回`EMPTY`。

判断 value 是否为 null

```
/**
* value是否为null
*/
public boolean isPresent() {
    return value != null;
}
/**
* 如果value不为null执行consumer.accept
*/
public void ifPresent(Consumer<? super T> consumer) {
   if (value != null)
    consumer.accept(value);
}
```

获取 value

```
/**
* Return the value if present, otherwise invoke {@code other} and return
* the result of that invocation.
* 如果value != null 返回value，否则返回other的执行结果
*/
public T orElseGet(Supplier<? extends T> other) {
    return value != null ? value : other.get();
}

/**
* 如果value != null 返回value，否则返回T
```

```
*/
public T orElse(T other) {
    return value != null ? value : other;
}

/**
* 如果value != null 返回value，否则抛出参数返回的异常
*/
public <X extends Throwable> T orElseThrow(Supplier<? extends X>
exceptionSupplier) throws X {
        if (value != null) {
            return value;
        } else {
            throw exceptionSupplier.get();
        }
}
/**
* value为null抛出NoSuchElementException，不为空返回value。
*/
public T get() {
  if (value == null) {
      throw new NoSuchElementException("No value present");
  }
  return value;
}
```

过滤值

```
/**
* 1. 如果是empty返回empty
* 2. predicate.test(value)==true 返回this，否则返回empty
*/
public Optional<T> filter(Predicate<? super T> predicate) {
        Objects.requireNonNull(predicate);
        if (!isPresent())
            return this;
        else
            return predicate.test(value) ? this : empty();
}
```

小结

看完 `Optional` 源码，`Optional` 的方法真的非常简单，值得注意的是如果坚决不想看见 `NPE`，就不要用 `of()` 、 `get()` 、`flatMap(..)`\。最后再综合用一下 `Optional` 的高频方法。

```
Optional.ofNullable(zoo).map(o -> o.getDog()).map(d -> d.getAge()).filter(v-
>v==1).orElse(3);
```

Date-Time API

这是对`java.util.Date`强有力的补充，解决了 Date 类的大部分痛点：

1. 非线程安全
2. 时区处理麻烦
3. 各种格式化、和时间计算繁琐
4. 设计有缺陷，Date 类同时包含日期和时间；还有一个 java.sql.Date，容易混淆。

我们从常用的时间实例来对比 java.util.Date 和新 Date 有什么区别。用`java.util.Date`的代码该改改了。

java.time 主要类

`java.util.Date` 既包含日期又包含时间，而 `java.time` 把它们进行了分离

```
LocalDateTime.class //日期+时间 format: yyyy-MM-ddTHH:mm:ss.SSS
LocalDate.class //日期 format: yyyy-MM-dd
LocalTime.class //时间 format: HH:mm:ss
```

格式化

Java 8 之前:

```
public void oldFormat(){
        Date now = new Date();
    //format yyyy-MM-dd HH:mm:ss
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    String date  = sdf.format(now);
    System.out.println(String.format("date format : %s", date));

    //format HH:mm:ss
    SimpleDateFormat sdft = new SimpleDateFormat("HH:mm:ss");
    String time = sdft.format(now);
    System.out.println(String.format("time format : %s", time));

    //format yyyy-MM-dd HH:mm:ss
    SimpleDateFormat sdfdt = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    String datetime = sdfdt.format(now);
    System.out.println(String.format("dateTime format : %s", datetime));
}
```

Java 8 之后:

```
public void newFormat(){
    //format yyyy-MM-dd
    LocalDate date = LocalDate.now();
    System.out.println(String.format("date format : %s", date));
```

```
    //format HH:mm:ss
    LocalTime time = LocalTime.now().withNano(0);
    System.out.println(String.format("time format : %s", time));

    //format yyyy-MM-dd HH:mm:ss
    LocalDateTime dateTime = LocalDateTime.now();
    DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");
    System.out.println(String.format("dateTime format : %s", dateTimeStr));
}
```

字符串转日期格式

Java 8 之前:

```
//已弃用
Date date = new Date("2021-01-26");
//替换为
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
Date date1 = sdf.parse("2021-01-26");
```

Java 8 之后:

```
LocalDate date = LocalDate.of(2021, 1, 26);
LocalDate.parse("2021-01-26");

LocalDateTime dateTime = LocalDateTime.of(2021, 1, 26, 12, 12, 22);
LocalDateTime.parse("2021-01-26 12:12:22");

LocalTime time = LocalTime.of(12, 12, 22);
LocalTime.parse("12:12:22");
```

Java 8 之前 转换都需要借助 `SimpleDateFormat` 类，而**Java 8 之后**只需要 `LocalDate`、`LocalTime`、`LocalDateTime`的 `of` 或 `parse` 方法。

日期计算

下面仅以**一周后日期**为例，其他单位（年、月、日、1/2 日、时等等）大同小异。另外，这些单位都在 *java.time.temporal.ChronoUnit* 枚举中定义。

Java 8 之前:

```
public void afterDay(){
     //一周后的日期
     SimpleDateFormat formatDate = new SimpleDateFormat("yyyy-MM-dd");
     Calendar ca = Calendar.getInstance();
     ca.add(Calendar.DATE, 7);
```

```
        Date d = ca.getTime();
        String after = formatDate.format(d);
        System.out.println("一周后日期：" + after);

    //算两个日期间隔多少天，计算间隔多少年，多少月方法类似
        String dates1 = "2021-12-23";
    String dates2 = "2021-02-26";
        SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
        Date date1 = format.parse(dates1);
        Date date2 = format.parse(dates2);
        int day = (int) ((date1.getTime() - date2.getTime()) / (1000 * 3600 * 24));
        System.out.println(dates2 + "和" + dates2 + "相差" + day + "天");
        //结果：2021-12-23和2021-12-23相差300天
}
```

Java 8 之后:

```
public void pushWeek(){
        //一周后的日期
        LocalDate localDate = LocalDate.now();
        //方法1
        LocalDate after = localDate.plus(1, ChronoUnit.WEEKS);
        //方法2
        LocalDate after2 = localDate.plusWeeks(1);
        System.out.println("一周后日期：" + after);

        //算两个日期间隔多少天，计算间隔多少年，多少月
        LocalDate date1 = LocalDate.parse("2021-02-26");
        LocalDate date2 = LocalDate.parse("2021-12-23");
        Period period = Period.between(date1, date2);
        System.out.println("date1 到 date2 相隔：
                    + period.getYears() + "年"
                    + period.getMonths() + "月"
                    + period.getDays() + "天");
            //打印结果是“date1 到 date2 相隔：0年9月27天”
        //这里period.getDays()得到的天是抛去年月以外的天数，并不是总天数
        //如果要获取纯粹的总天数应该用下面的方法
        long day = date2.toEpochDay() - date1.toEpochDay();
        System.out.println(date2 + "和" + date2 + "相差" + day + "天");
        //打印结果：2021-12-23和2021-12-23相差300天
}
```

获取指定日期

除了日期计算繁琐，获取特定一个日期也很麻烦，比如获取本月最后一天，第一天。

Java 8 之前:

```
public void getDay() {

        SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
        //获取当前月第一天:
        Calendar c = Calendar.getInstance();
        c.add(Calendar.MONTH, 0);
        c.set(Calendar.DAY_OF_MONTH, 1);
        String first = format.format(c.getTime());
        System.out.println("first day:" + first);

        //获取当前月最后一天
        Calendar ca = Calendar.getInstance();
        ca.set(Calendar.DAY_OF_MONTH, ca.getActualMaximum(Calendar.DAY_OF_MONTH));
        String last = format.format(ca.getTime());
        System.out.println("last day:" + last);

        //当年最后一天
        Calendar currCal = Calendar.getInstance();
        Calendar calendar = Calendar.getInstance();
        calendar.clear();
        calendar.set(Calendar.YEAR, currCal.get(Calendar.YEAR));
        calendar.roll(Calendar.DAY_OF_YEAR, -1);
        Date time = calendar.getTime();
        System.out.println("last day:" + format.format(time));
}
```

Java 8 之后:

```
public void getDayNew() {
    LocalDate today = LocalDate.now();
    //获取当前月第一天:
    LocalDate firstDayOfThisMonth =
today.with(TemporalAdjusters.firstDayOfMonth());
    // 取本月最后一天
    LocalDate lastDayOfThisMonth = today.with(TemporalAdjusters.lastDayOfMonth());
    //取下一天:
    LocalDate nextDay = lastDayOfThisMonth.plusDays(1);
    //当年最后一天
    LocalDate lastday = today.with(TemporalAdjusters.lastDayOfYear());
    //2021年最后一个周日，如果用Calendar是不得烦死。
    LocalDate lastMondayOf2021 = LocalDate.parse("2021-12-
31").with(TemporalAdjusters.lastInMonth(DayOfWeek.SUNDAY));
}
```

`java.time.temporal.TemporalAdjusters` 里面还有很多便捷的算法，这里就不带大家看 Api 了，都很简单，看了秒懂。

JDBC 和 java8

现在 jdbc 时间类型和 java8 时间类型对应关系是

1. `Date` ---> `LocalDate`
2. `Time` ---> `LocalTime`
3. `TimesSamp` ---> `LocalDateTime`

而之前统统对应 `Date`，也只有 `Date`。

时区

> 时区：正式的时区划分为每隔经度 15° 划分一个时区，全球共 24 个时区，每个时区相差 1 小时。但为了行政上的方便，常将 1 个国家或 1 个省份划在一起，比如我国幅员宽广，大概横跨 5 个时区，实际上只用东八时区的标准时即北京时间为准。

`java.util.Date` 对象实质上存的是 1970 年 1 月 1 日 0 点（GMT）至 Date 对象所表示时刻所经过的毫秒数。也就是说不管在哪个时区 new Date，它记录的毫秒数都一样，和时区无关。但在使用上应该把它转换成当地时间，这就涉及到了时间的国际化。`java.util.Date` 本身并不支持国际化，需要借助 `TimeZone`。

```
//北京时间: Wed Jan 27 14:05:29 CST 2021
Date date = new Date();

SimpleDateFormat bjSdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
//北京时区
bjSdf.setTimeZone(TimeZone.getTimeZone("Asia/Shanghai"));
System.out.println("毫秒数:" + date.getTime() + ", 北京时间:" +
bjSdf.format(date));

//东京时区
SimpleDateFormat tokyoSdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
tokyoSdf.setTimeZone(TimeZone.getTimeZone("Asia/Tokyo"));  // 设置东京时区
System.out.println("毫秒数:" + date.getTime() + ", 东京时间:" +
tokyoSdf.format(date));

//如果直接print会自动转成当前时区的时间
System.out.println(date);
//Wed Jan 27 14:05:29 CST 2021
```

在新特性中引入了 `java.time.ZonedDateTime` 来表示带时区的时间。它可以看成是 `LocalDateTime + ZoneId`。

```
//当前时区时间
ZonedDateTime zonedDateTime = ZonedDateTime.now();
System.out.println("当前时区时间: " + zonedDateTime);

//东京时间
ZoneId zoneId = ZoneId.of(ZoneId.SHORT_IDS.get("JST"));
ZonedDateTime tokyoTime = zonedDateTime.withZoneSameInstant(zoneId);
System.out.println("东京时间: " + tokyoTime);

// ZonedDateTime 转 LocalDateTime
```

```
LocalDateTime localDateTime = tokyoTime.toLocalDateTime();
System.out.println("东京时间转当地时间: " + localDateTime);

//LocalDateTime 转 ZonedDateTime
ZonedDateTime localZoned = localDateTime.atZone(ZoneId.systemDefault());
System.out.println("本地时区时间: " + localZoned);

//打印结果
当前时区时间: 2021-01-27T14:43:58.735+08:00[Asia/Shanghai]
东京时间: 2021-01-27T15:43:58.735+09:00[Asia/Tokyo]
东京时间转当地时间: 2021-01-27T15:43:58.735
当地时区时间: 2021-01-27T15:53:35.618+08:00[Asia/Shanghai]
```

小结

通过上面比较新老 `Date` 的不同，当然只列出部分功能上的区别，更多功能还得自己去挖掘。总之 date-time-api 给日期操作带来了福利。在日常工作中遇到 date 类型的操作，第一考虑的是 date-time-api，实在解决不了再考虑老的 Date。

总结

我们梳理总结的 java 8 新特性有

- Interface & functional Interface
- Lambda
- Stream
- Optional
- Date time-api

这些都是开发当中比较常用的特征。梳理下来发现它们真香，而我却没有更早的应用。总觉得学习 java 8 新特性比较麻烦，一致使用老的实现方式。其实这些新特性几天就可以掌握，一但掌握，效率会有很大的提高。其实我们涨工资也是涨的学习的钱，不学习终究会被淘汰，35 岁危机会提前来临。