Java 8 new features in practice

Guide Java New Java Features About 6351 words About 21 minutes

> This article comes from cowbi 's contribution~

Oracle released Java 8 (JDK 1.8) in 2014, and for many reasons, it has become the most widely used JDK version on the market. Although it's been nearly seven years since its release, many programmers still don't fully understand its new features, especially those who are used to versions prior to Java 8, like me.

To avoid straying too far from the mainstream, it's worth summarizing these new features. Compared to JDK 7, there are many changes or optimizations. For example, interfaces can contain static methods with bodies, a completely new concept; `java.util.HashMap` red-black trees have been added to the data structure; and lambda expressions are well-known. This article cannot cover all the new features one by one, so we will only list the more commonly used ones and provide detailed explanations. For more information, please see the official website's introduction to new features in Java 8 .

Interface

The original intention of designing interfaces was to provide abstraction and improve extensibility. This also leaves a bit of a quirk: when an interface is modified, the class that implements it must also be modified.

In order to solve the problem of incompatibility between interface modification and existing implementation, the method of the new interface can be modified with `default` or `static` , so that it can have a method body and the implementation class does not need to rewrite this method.

There can be multiple methods in an interface that are modified by them. The difference between these two modifiers is mainly the difference between ordinary methods and static methods.

1. `default` The modified method is an ordinary instance method that can be `this` called and can be inherited and overridden by subclasses.
2. `static` The modified method is used in the same way as a general class static met However, it cannot be inherited by subclasses and can only be `Interface` called.

Let's look at a practical example.

```java
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");
    }
}
```

If a class implements both `InterfaceNew` the interface and `InterfaceNew1` the interface `def()` , and `InterfaceNew` the interface and `InterfaceNew1` the interface have no inheritance relationship, then it must be rewritten `def()` . Otherwise, an error will be reported during compilation.

```java
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() {
    }
}
```

In Java 8, what is the difference between interfaces and abstract classes?

Many people think: "Since interfaces can also have their own method implementations, it seems that there is not much difference between them and abstract classes."

In fact, they are still different

1. The difference between interface and class seems to be nonsense, mainly:
    - Multiple implementations of interfaces, single inheritance of classes
    - The interface method is public abstract, and the variable is public static final. The abstract class can use other modifiers.
2. The interface method is more like an extension plug-in, while the abstract class method is inherited.

As mentioned earlier, the addition `default` and `static` modification of interface methods are intended to address incompatibilities between interface modifications and existing implementations, not to replace them `abstract class` . In practice, abstract classes should continue to be used where they are appropriate, and avoid replacing them simply because of new features in interfaces.

Remember that an interface is never the same as a class.

functional interface

Definition : Also known as SAM interface, that is, Single Abstract Method interfaces, an interface that has one and only one abstract method but can have multiple non-abstract methods.

In Java 8, there is a special package for functional interfaces `java.util.function` . All interfaces under this package have `@FunctionalInterface` annotations to provide functional programming.

Functional interfaces exist in other packages, some of which lack `@FunctionalInterface` annotations. However, as long as they conform to the definition of a functional interface, `@FunctionalInterface` they are considered functional interfaces, regardless of whether they have annotations. Annotations only enforce the definition at compile time. They are widely used in lambda expressions.

Lambda expressions

Next, let's talk about the well-known lambda expression. It is the most important new feature that drove the release of Java 8. It is the biggest change since generics `Generics` and annotations . `Annotation`

Using Lambda expressions can make the code more concise and compact, allowing Java to support simple *functional programming* .

> A lambda expression is an anonymous function that Java 8 allows you to pass functions as parameters into methods.

Syntax

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

Lambda in Action

We use common examples to experience the convenience brought by Lambda

Alternative to anonymous inner classes

In the past, the only way to pass dynamic parameters to methods was to use inner classes. For example

1. `Runnable` Interface

```java
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. Comparator Interface

```java
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> comparator = (Integer o1, Integer o2) -> o1 - o2;
Collections.sort(strings, comparator);
```

3. Listener Interface

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

4. Custom interface

The three examples above are common in development, and they demonstrate the convenience and efficiency of Lambda. It retains only the code that's actually needed, omitting any unnecessary code. So, are there any interface requirements? We've observed that the anonymous inner classes only override a single interface method, and of course, only method needs to be overridden. This is **the functional interface** we mentioned above . This means that any method parameter that is a functional interface can use a Lambda expression.

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

@FunctionalInterface
public interface Runnable{}
```

We customize a functional interface

```java
@FunctionalInterface
public interface LambdaInterface {
 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){
        i.f();
    }
}
```

Collection Iteration

```java
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));
}
```

Method references

Java 8 allows the use of `::` the keyword to pass a method or constructor reference. However, the type returned by the expression must be a functional-interface.

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

public class LambdaClass extends LambdaClassSuper {
    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;
    }
}
```

Accessing variables

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

Lambda expressions can reference external variables, but these variables have the final attribute by default and cannot be modified. If modified, an error will be reported during compilation.

Stream

Java has added a new `java.util.stream` package, which is similar to the previous stream. Previously, the most common use case was resource streams, for example , using streams to move files from one location to another. These streams were simply content movers and didn't perform any *CRUD* `java.io.FileInputStream` operations on the file contents .

`Stream` It still does not store data, but it can retrieve and logically process collection data, including filtering, sorting, statistics, counting, etc. It can be imagined as a SQL statement.

Its source data can be `Collection` , `Array` etc. Since its method parameters are all functional interface types, it is generally used in conjunction with Lambda.

Stream Type

1. stream serial stream
2. parallelStream parallel stream, multi-threaded execution

Common methods

Next, let's look at `java.util.stream.Stream` the common methods

```java
/**
* 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();
```

```java
/**
* 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)
```

Actual combat

This article lists `Stream` the usage of representative methods. For more usage methods, please refer to the API.

```java
@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);
}
```

Delayed execution

When executing `Stream` a method that returns , it doesn't execute immediately. Instead, it waits until a `Stream` method that returns something other than is executed. Because `Stream` can't be used directly, it needs to be processed into a regular type. The here `Stream` can be thought of as a binary stream (two completely different things), which is difficult to understand even if you get it.

Let's break down `filter` the method below.

```java
@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
```

According to the execution order, "Executed" should be printed four times `Predicate.test` before " `count` Executed". However, the actual result is exactly the opposite. This indicates that the method in the filter is not executed immediately, but only `count()` after the method is called.

The examples above all use serial `Stream` operations. Parallel operations `parallelStream` are used in the same way as serial operations. The main difference is `parallelStream` multi-threaded execution, which is implemented using the ForkJoin framework. You can learn more about this `ForkJoin` framework and its implementation when you have time `ForkJoinPool` . For simplicity, it's implemented using a thread pool, which raises issues like thread safety and thread consumption. Let's explore the multi-threaded execution of parallel streams through code.

```java
@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
```

From the results we can see that for-each uses multithreading.

summary

From the source code and examples, we can summarize some characteristics of stream

1. Through simple chain programming, it is easy to reprocess the data after traversal.
2. Method parameters are all functional interface types
3. A Stream can only be operated once and is closed after the operation. Continuing to use this stream will result in an error.
4. Stream does not save data and does not change the data source

Optional

In the introduction to Optional in Alibaba Development Manual , it is written as follows:

> Preventing NPE is a basic skill for programmers. Pay attention to the scenarios where NPE occurs:
> 1) When the return type is a primitive data type and an object of a packaged data type is returned, automatic unboxing may cause an NPE.
> Counterexample: public int f() { return Integer object}, if it is null, it will automatically unbox and throw NPE.
> 2) The query result of the database may be null.
> 3) Even if the elements in the collection are not empty, the data elements retrieved may be null.
> 4) When a remote call returns an object, a null pointer check is always required to prevent NPE.
> 5) For data obtained from the Session, it is recommended to perform NPE checks to avoid null pointers.
> 6) Cascade calls to obj.getA().getB().getC(); a series of calls can easily cause NPE.
> Positive example: Use JDK8's Optional class to prevent NPE issues.

He suggested using `Optional` to solve the NPE (`java.lang.NullPointerException`) problem. It was created specifically for NPEs and can contain null or non-null values. Let's gradually uncover `Optional` the hidden secrets of through the source code.

Suppose there is a `Zoo` class with an attribute `Dog` that needs to be `Dog` obtained `age` .

```java
class Zoo {
   private Dog dog;
}

class Dog {
   private int age;
}
```

The traditional solutions to NPE are as follows:

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

Some people say that layering the check for null objects is ugly and inelegant, but I don't think so. On the contrary, I find it neat, easy to read, and understand. What do you think?

`Optional` This is how it is implemented:

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

Isn't it a lot simpler?

How to create an Optional

The above example `Optional.ofNullable` shows one way to create an Optional. Let's first look at its meaning and other source code methods for creating Optional.

```java
/**
* 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` The only difference between the two `of` methods is that when value is null, `ofNullable` the return value is `EMPTY` , while of will throw `NullPointerException` an exception. If you need to `NullPointerException` expose , use it `of` ; otherwise, use `ofNullable` .

`map()` `flatMap()` What is the difference between and ?

`map` Both `flatMap` apply a function to each element in a collection, but the difference is that `map` a new collection is returned, `flatMap` which maps each element to a collection and then flattens the collection.

In actual application scenarios, if `map` an array is returned, the final result is a two-dimensional array. The purpose of using it `flatMap` is to flatten this two-dimensional array into a one-dimensional array.

```java
public class MapAndFlatMapExample {
    public static void main(String[] args) {
        List<String[]> listOfArrays = Arrays.asList(
                new String[]{"apple", "banana", "cherry"},
                new String[]{"orange", "grape", "pear"},
                new String[]{"kiwi", "melon", "pineapple"}
        );

        List<String[]> mapResult = listOfArrays.stream()
                .map(array ->
Arrays.stream(array).map(String::toUpperCase).toArray(String[]::new))
                .collect(Collectors.toList());

        System.out.println("Using map:");
        mapResult.forEach(arrays->
System.out.println(Arrays.toString(arrays)));

        List<String> flatMapResult = listOfArrays.stream()
                .flatMap(array ->
Arrays.stream(array).map(String::toUpperCase))
                .collect(Collectors.toList());

        System.out.println("Using flatMap:");
        System.out.println(flatMapResult);
    }
}
```

Run results:

```plain
Using map:
[[APPLE, BANANA, CHERRY], [ORANGE, GRAPE, PEAR], [KIWI, MELON,
PINEAPPLE]]

Using flatMap:
[APPLE, BANANA, CHERRY, ORANGE, GRAPE, PEAR, KIWI, MELON,
PINEAPPLE]
```

The simplest understanding is that the results `flatMap()` can be expanded. `map()`

Inside `Optional` , when used `map()` , if the mapping function returns a normal value, it will wrap the value in a new one `Optional` . When used `flatMap` , if the mapping function returns a one , it will flatten `Optional` the returned one and no longer wrap it into a nested one . `Optional` `Optional`

Here is a sample code for comparison:

```java
public static void main(String[] args) {
        int userId = 1;

        // 使用flatMap的代码
        String cityUsingFlatMap = getUserById(userId)
                .flatMap(OptionalExample::getAddressByUser)
                .map(Address::getCity)
                .orElse("Unknown");

        System.out.println("User's city using flatMap: " +
cityUsingFlatMap);

        // 不使用flatMap的代码
        Optional<Optional<Address>> optionalAddress =
getUserById(userId)
                .map(OptionalExample::getAddressByUser);

        String cityWithoutFlatMap;
        if (optionalAddress.isPresent()) {
            Optional<Address> addressOptional =
optionalAddress.get();
            if (addressOptional.isPresent()) {
                Address address = addressOptional.get();
                cityWithoutFlatMap = address.getCity();
            } else {
                cityWithoutFlatMap = "Unknown";
            }
        } else {
            cityWithoutFlatMap = "Unknown";
        }

        System.out.println("User's city without flatMap: " +
cityWithoutFlatMap);
    }
```

Proper use in `Stream` and can reduce a lot of unnecessary code. `Optional` `flatMap`

Determine whether value is null

java

```java
/**
* 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);
}
```

Get value

java

```java
/**
* 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;
}
```

Filter Values

```java
/**
* 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();
}
```

summary

After reading `Optional` the source code, `Optional` the method is really very simple. It is worth noting that if you don't want to see it `NPE` , don't use `of()` , `get()` , `flatMap(..)` . Finally, use `Optional` the high-frequency method of .

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

Date-Time API

This is a `java.util.Date` powerful addition that solves most of the pain points of the Date class:

1. Not thread safe
2. Time zone handling troubles
3. Various formatting and time calculations are cumbersome
4. The design is flawed, the Date class contains both date and time; there is also a java.sql.Date, which is confusing.

Let's compare the differences between java.util.Date and the new Date using common examples. `java.util.Date` The code used should be modified.

java.time Main Classes

`java.util.Date` Contains both date and time, but `java.time` separates them

```java
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
```

format

Before Java 8:

```java
public void oldFormat(){
    Date now = new Date();
    //format yyyy-MM-dd
    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));
}
```

After Java 8:

```java
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");
    String dateTimeStr = dateTime.format(dateTimeFormatter);
    System.out.println(String.format("dateTime format : %s",
dateTimeStr));
}
```

Convert string to date format

Before Java 8:

```java
//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");
```

After Java 8:

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

LocalDateTime dateTime = LocalDateTime.of(2021, 1, 26, 12, 12, 22);
LocalDateTime.parse("2021-01-26T12:12:22");

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

Before Java 8 , conversions required the use of `SimpleDateFormat` the class, while **after Java 8,** only the `LocalDate` , `LocalTime` , `LocalDateTime` or methods of `.of` `parse`

Date calculation

The following example uses **a date one week** from now . The other units (year, month, day, half-day, hour, etc.) are similar. These units are all defined in *the java.time.temporal.ChronoUnit* enumeration.

Before Java 8:

```java
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(dates1 + "和" + dates2 + "相差" + day + "天");
     //结果：2021-02-26和2021-12-23相差300天
}
```

After Java 8:

```java
public void pushWeek(){
     //一周后的日期
     LocalDate localDate = LocalDate.now();
     //方法1
     LocalDate after = localDate.plus(1, ChronoUnit.WEEKS);
```

```java
        //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(date1 + "和" + date2 + "相差" + day + "天");
        //打印结果: 2021-02-26和2021-12-23相差300天
    }

```

Get the specified date

In addition to the tedious date calculation, getting a specific date is also troublesome, such as getting the last day or first day of the month.

Before Java 8:

java

```java
public void getDay() {

        SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-
dd");
        //获取当前月第一天:
        Calendar c = Calendar.getInstance();
        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));
}
```

After Java 8:

```java
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` There are many convenient algorithms in it. I won't show you the API here. They are all very simple and you can understand them in seconds.

JDBC and Java 8

Now the correspondence between jdbc time type and java8 time type is

1. `Date ---> LocalDate`
2. `Time ---> LocalTime`

3. `Timestamp ---> LocalDateTime`

And all the previous ones corresponded `Date` only to this `Date` .

Time Zone

> Time Zones: Officially, time zones are divided every 15° of longitude, for a total of 24 time zones worldwide, each with a one-hour difference. However, for administrative convenience, countries or provinces are often grouped together. For example, my country, with its vast territory spanning approximately five time zones, actually uses only the standard time of the GMT+8 time zone, or Beijing Time.

`java.util.Date` The object actually stores the number of milliseconds that have elapsed since 12:00 AM (GMT) on January 1, 1970, to the time represented by the Date object. This means that regardless of the time zone in which the new Date is created, the milliseconds recorded remain the same, regardless of the time zone. However, when using it, it must be converted to local time, which involves internationalization of time. `java.util.Date` However, the object does not natively support internationalization and requires assistance `TimeZone` .

```java
//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
```

The new feature is introduced `java.time.ZonedDateTime` to represent time with time zone. It can be seen as `LocalDateTime + ZoneId` .

```java
//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]
```

summary

The above comparison of the new and old `Date` APIs only covers a few functional differences; you'll need to explore more. In short, the Date-Time API brings significant benefits to date manipulation. When encountering date-related operations in daily work, consider the Date-Time API first. If that doesn't work, consider the older Date API.

Summarize

The new features of Java 8 we have sorted out are

- Interface & functional interface

- Lambda
- Stream
- Optional
- Date time-api

These are all commonly used features in development. After reviewing them, I discovered how valuable they are, yet I haven't used them earlier. I've always found learning Java 8's new features cumbersome, so I've been using older implementations. In reality, these new features can be mastered in just a few days, and once mastered, efficiency will greatly improve. Salary increases are also paid for learning. If you don't learn, you'll eventually be eliminated, and the 35-year-old crisis will arrive sooner.

Recently Updated2025/2/13 19:17

Contributors: guide , Ma Zhaoming , HuYe , xiaoshuguo , anaer , huang.xinghui , Mzmba , lianzhy , 5ME , Cyunrei , Mr.Hope , jun , Guide , gemaxis , HaiBooLang