

点击关注[公众号](#)及时获取笔主最新更新文章，并可免费领取本文档配套的《Java面试突击》以及Java工程师必备学习资源。

随着 Java 8 的普及度越来越高，很多人都提到面试中关于Java 8 也是非常常问的知识点。应各位要求和需要，我打算对这部分知识做一个总结。本来准备自己总结的，后面看到Github 上有一个相关的仓库，地址：<https://github.com/winterbe/java8-tutorial>。这个仓库是英文的，我对其进行了翻译并添加和修改了部分内容，下面是正文了。

- Java 8 Tutorial
 - 接口的默认方法(Default Methods for Interfaces)
 - Lambda表达式(Lambda expressions)
 - 函数式接口(Functional Interfaces)
 - 方法和构造函数引用(Method and Constructor References)
 - Lamda 表达式作用域(Lambda Scopes)
 - 访问局部变量
 - 访问字段和静态变量
 - 访问默认接口方法
 - 内置函数式接口(Built-in Functional Interfaces)
 - Predicate
 - Function
 - Supplier
 - Consumer
 - Comparator
 - Optional
 - Streams(流)
 - Filter(过滤)
 - Sorted(排序)
 - Map(映射)
 - Match(匹配)
 - Count(计数)
 - Reduce(规约)
 - Parallel Streams(并行流)
 - Sequential Sort(串行排序)
 - Parallel Sort(并行排序)
 - Maps
 - Date API(日期相关API)
 - Clock
 - Timezones(时区)
 - LocalTime(本地时间)
 - LocalDate(本地日期)
 - LocalDateTime(本地日期时间)
 - Annotations(注解)
 - Where to go from here?

Java 8 Tutorial

欢迎阅读我对Java 8的介绍。本教程将逐步指导您完成所有新语言功能。在简短的代码示例的基础上，您将学习如何使用默认接口方法，lambda表达式，方法引用和可重复注释。在本文的最后，您将熟悉最新的 API 更改，如流，函数式接口(Functional Interfaces)，Map 类的扩展和新的 Date API。没有大段枯燥的文字，只有一堆注释的代码片段。

接口的默认方法(Default Methods for Interfaces)

Java 8使我们能够通过使用 `default` 关键字向接口添加非抽象方法实现。此功能也称为[虚拟扩展方法](#)。

第一个例子：

```
interface Formula{

    double calculate(int a);

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

}
```

Formula 接口中除了抽象方法计算接口公式还定义了默认方法 `sqrt`。实现该接口的类只需要实现抽象方法 `calculate`。默认方法`sqrt`可以直接使用。当然你也可以直接通过接口创建对象，然后实现接口中的默认方法就可以了，我们通过代码演示一下这种方式。

```
public class Main {

    public static void main(String[] args) {
        // 通过匿名内部类方式访问接口
        Formula formula = new Formula() {
            @Override
            public double calculate(int a) {
                return sqrt(a * 100);
            }
        };

        System.out.println(formula.calculate(100));    // 100.0
        System.out.println(formula.sqrt(16));         // 4.0

    }

}
```

formula 是作为匿名对象实现的。该代码非常容易理解，6行代码实现了计算 `sqrt(a * 100)`。在下一节中，我们将会看到在 Java 8 中实现单个方法对象有一种更好更方便的方法。

译者注：不管是抽象类还是接口，都可以通过匿名内部类的方式访问。不能通过抽象类或者接口直接创建对象。对于上面通过匿名内部类方式访问接口，我们可以这样理解：一个内部类实现了接口里的抽象方法并且返

回一个内部类对象，之后我们让接口的引用来指向这个对象。

Lambda表达式(Lambda expressions)

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);
});
```

可以看出，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：

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

`List` 类本身就有个 `sort` 方法。并且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表达式还有什么其他用法。

函数式接口(Functional Interfaces)

译者注： 原文对这部分解释不太清楚，故做了修改！

Java 语言设计者们投入了大量精力来思考如何使现有的函数友好地支持Lambda。最终采取的方法是：增加函数式接口的概念。“函数式接口”是指仅仅只包含一个抽象方法，但是可以有多个非抽象方法(也就是上面提到的默认方法)的接口。像这样的接口，可以被隐式转换为lambda表达式。`java.lang.Runnable` 与 `java.util.concurrent.Callable` 是函数式接口最典型的两个例子。Java 8增加了一种特殊的注解 `@FunctionalInterface`，但是这个注解通常不是必须的(某些情况建议使用)，只要接口只包含一个抽象方法，

虚拟机会自动判断该接口为函数式接口。一般建议在接口上使用`@FunctionalInterface` 注解进行声明，这样的话，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，如下图所示

```
3 @FunctionalInterface
4 Invalid '@FunctionalInterface' annotation; Converter<F,T> is not a functional interface
5     T convert(F from);
6     void anotherAbstractMethod();
7 }
8
```

示例:

```
@FunctionalInterface
public interface Converter<F, T> {
    T convert(F from);
}
```

```
// TODO 将数字字符串转换为整数类型
Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
Integer converted = converter.convert("123");
System.out.println(converted.getClass()); //class java.lang.Integer
```

译者注：大部分函数式接口都不用我们自己写，Java8都给我们实现好了，这些接口都在`java.util.function`包里。

方法和构造函数引用(Method and Constructor References)

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.getClass()); //class java.lang.Integer
```

Java 8允许您通过`::`关键字传递方法或构造函数的引用。上面的示例显示了如何引用静态方法。但我们也可以引用对象方法：

```
class Something {
    String startsWith(String s) {
        return String.valueOf(s.charAt(0));
    }
}
```

```
Something something = new Something();
Converter<String, String> 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，而不是手动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` 方法的参数类型来选择合适的构造函数。

Lambda 表达式作用域(Lambda Scopes)

访问局部变量

我们可以直接在 lambda 表达式中访问外部的局部变量：

```
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。
```

访问字段和静态变量

与局部变量相比，我们对lambda表达式中的实例字段和静态变量都有读写访问权限。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);
        };
    }
}
```

访问默认接口方法

还记得第一节中的 formula 示例吗？Formula 接口定义了一个默认方法sqrt，可以从包含匿名对象的每个 formula 实例访问该方法。这不适用于lambda表达式。

无法从 lambda 表达式中访问默认方法,故以下代码无法编译：

```
Formula formula = (a) -> sqrt(a * 100);
```

内置函数式接口(Built-in Functional Interfaces)

JDK 1.8 API包含许多内置函数式接口。其中一些接口在老版本的 Java 中是比较常见的比如：[Comparator](#) 或 [Runnable](#)，这些接口都增加了[@FunctionalInterface](#)注解以便能用在 lambda 表达式上。

但是 Java 8 API 同样还提供了很多全新的函数式接口来让你的编程工作更加方便，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

Predicate 接口是只有一个参数的返回布尔类型值的 **断言型** 接口。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）：

译者注： Predicate 接口源码如下

```
package java.util.function;
import java.util.Objects;

@FunctionalInterface
public interface Predicate<T> {

    // 该方法是接受一个传入类型,返回一个布尔值.此方法应用于判断.
    boolean test(T t);

    //and方法与关系型运算符"&&"相似,两边都成立才返回true
    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }
    // 与关系运算符"!"相似,对判断进行取反
    default Predicate<T> negate() {
        return (t) -> !test(t);
    }
    //or方法与关系型运算符"||"相似,两边只要有一个成立就返回true
    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }
    // 该方法接收一个Object对象,返回一个Predicate类型.此方法用于判断第一个test的方法与第二个test方法相同(equal).
    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef)
            ? Objects::isNull
            : object -> targetRef.equals(object);
    }
}
```

示例：

```

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

Function 接口接受一个参数并生成结果。默认方法可用于将多个函数链接在一起（compose, andThen）：

译者注： Function 接口源码如下

```

package java.util.function;

import java.util.Objects;

@FunctionalInterface
public interface Function<T, R> {

    //将Function对象应用到输入的参数上，然后返回计算结果。
    R apply(T t);
    //将两个Function整合，并返回一个能够执行两个Function对象功能的Function对象。
    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }
    //
    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}

```

```

Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String::valueOf);
backToString.apply("123");           // "123"

```


Supplier

Supplier 接口产生给定泛型类型的结果。与 Function 接口不同，Supplier 接口不接受参数。

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

Consumer

Consumer 接口表示要对单个输入参数执行的操作。

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

Comparator

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

Optional不是函数式接口，而是用于防止 NullPointerException 的漂亮工具。这是下一节的一个重要概念，让我们快速了解一下Optional的工作原理。

Optional 是一个简单的容器，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有时却什么也没有返回，而在Java 8中，你应该返回 Optional 而不是 null。

译者注：示例中每个方法的作用已经添加。

```
//of () : 为非null的值创建一个Optional
Optional<String> optional = Optional.of("bam");
//isPresent () : 如果值存在返回true, 否则返回false
optional.isPresent();           // true
//get(): 如果Optional有值则将其返回, 否则抛出NoSuchElementException
optional.get();                  // "bam"
//orElse () : 如果有值则将其返回, 否则返回指定的其它值
optional.orElse("fallback");    // "bam"
```

```
//ifPresent () : 如果Optional实例有值则为其调用consumer, 否则不做处理
optional.ifPresent((s) -> System.out.println(s.charAt(0)));    // "b"
```

推荐阅读: [\[Java8\]如何正确使用Optional](#)

Streams(流)

`java.util.Stream` 表示能应用在一组元素上一次执行的操作序列。Stream 操作分为中间操作或者最终操作两种, 最终操作返回一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, 首先创建实例代码需要用到的数据List:

```
List<String> stringList = new ArrayList<>();
stringList.add("ddd2");
stringList.add("aaa2");
stringList.add("bbb1");
stringList.add("aaa1");
stringList.add("bbb3");
stringList.add("ccc");
stringList.add("bbb2");
stringList.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**, 所以我们可以在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。

```
// 测试 Filter(过滤)
stringList
    .stream()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println); //aaa2 aaa1
```

forEach 是为 Lambda 而设计的, 保持了最紧凑的风格。而且 Lambda 表达式本身是可以重用的, 非常方便。

Sorted(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。

```
// 测试 Sort (排序)
stringList
    .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是会被修改的：

```
System.out.println(stringList);// 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来将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。

```
// 测试 Map 操作
stringList
    .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。

```
// 测试 Match (匹配)操作
boolean anyStartsWithA =
    stringList
        .stream()
        .anyMatch((s) -> s.startsWith("a"));
System.out.println(anyStartsWithA);    // true

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

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

```

        boolean noneStartsWithZ =
            stringList
                .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**。

```

//测试 Count (计数)操作
long startsWithB =
    stringList
        .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将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：

```

//测试 Reduce (规约)操作
Optional<String> reduced =
    stringList
        .stream()
        .sorted()
        .reduce((s1, s2) -> s1 + "#" + s2);

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

```

译者注：这个方法的主要作用是把 Stream 元素组合起来。它提供一个起始值（种子），然后依照运算规则（BinaryOperator），和前面 Stream 的第一个、第二个、第 n 个元素组合。从这个意义上说，字符串拼接、数值的 sum、min、max、average 都是特殊的 reduce。例如 Stream 的 sum 就相当于 `Integer sum = integers.reduce(0, (a, b) -> a+b)`；也有没有起始值的情况，这时会把 Stream 的前面两个元素组合起来，返回的是 Optional。

```

// 字符串连接, concat = "ABCD"
String concat = Stream.of("A", "B", "C", "D").reduce("", String::concat);
// 求最小值, minValue = -3.0
double minValue = Stream.of(-1.5, 1.0, -3.0, -2.0).reduce(Double.MAX_VALUE,
    Double::min);

```

```
// 求和, sumValue = 10, 有起始值
int sumValue = Stream.of(1, 2, 3, 4).reduce(0, Integer::sum);
// 求和, sumValue = 10, 无起始值
sumValue = Stream.of(1, 2, 3, 4).reduce(Integer::sum).get();
// 过滤, 字符串连接, concat = "ace"
concat = Stream.of("a", "B", "c", "D", "e", "F").
    filter(x -> x.compareTo("Z") > 0).
    reduce("", String::concat);
```

上面代码例如第一个示例的 `reduce()`, 第一个参数 (空白字符) 即为起始值, 第二个参数 (`String::concat`) 为 `BinaryOperator`。这类有起始值的 `reduce()` 都返回具体的对象。而对于第四个示例没有起始值的 `reduce()`, 由于可能没有足够的元素, 返回的是 `Optional`, 请留意这个区别。更多内容查看: [IBM: Java 8 中的 Streams API 详解](#)

Parallel Streams(并行流)

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());
}
```

我们分别用串行和并行两种方式对其进行排序, 最后看看所用时间的对比。

Sequential Sort(串行排序)

```
//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));
```

```
1000000
sequential sort took: 709 ms//串行排序所用的时间
```

Parallel Sort(并行排序)

```
//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));
```

```
1000000
parallel sort took: 475 ms//串行排序所用的时间
```

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

Maps

前面提到过，Map 类型不支持 streams，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接口本身没有可用的 `stream()` 方法，但是你可以在键，值上创建专门的流或者通过 `map.keySet().stream()`、`map.values().stream()` 和 `map.entrySet().stream()`。

此外,Maps 支持各种新的和有用的方法来执行常见任务。

```
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)); //val0 val1 val2 val3 val4 val5
val6 val7 val8 val9
```

`putIfAbsent` 阻止我们在null检查时写入额外的代码;`forEach`接受一个 consumer 来对 map 中的每个元素操作。

此示例显示如何使用函数在 map 上计算代码：

```
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，否则对原键对应的值做合并操作并重新插入到map中。

Date API(日期相关API)

Java 8在 `java.time` 包下包含一个全新的日期和时间API。新的Date API与Joda-Time库相似，但它们不一样。以下示例涵盖了此新 API 的最重要部分。译者对这部分内容参考相关书籍做了大部分修改。

译者注(总结):

- Clock 类提供了访问当前日期和时间的方法，Clock 是时区敏感的，可以用来取代 `System.currentTimeMillis()` 来获取当前的微秒数。某一个特定的时间点也可以使用 `Instant` 类来表示，`Instant` 类也可以用来创建旧版本的 `java.util.Date` 对象。
- 在新API中时区使用 `ZoneId` 来表示。时区可以很方便的使用静态方法 `of` 来获取到。抽象类 `ZoneId` (在 `java.time` 包中) 表示一个区域标识符。它有一个名为 `getAvailableZoneIds` 的静态方法，它返回所有区域标识符。

- jdk1.8中新增了 `LocalDate` 与 `LocalDateTime`等类来解决日期处理方法，同时引入了一个新的类 `DateTimeFormatter` 来解决日期格式化问题。可以使用`Instant`代替 `Date`，`LocalDateTime`代替 `Calendar`，`DateTimeFormatter` 代替 `SimpleDateFormat`。

Clock

`Clock` 类提供了访问当前日期和时间的方法，`Clock` 是时区敏感的，可以用来取代 `System.currentTimeMillis()` 来获取当前的微秒数。某一个特定的时间点也可以使用 `Instant` 类来表示，`Instant` 类也可以用来创建旧版本的`java.util.Date` 对象。

```
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();
System.out.println(millis);//1552379579043
Instant instant = clock.instant();
System.out.println(instant);
Date legacyDate = Date.from(instant); //2019-03-12T08:46:42.588Z
System.out.println(legacyDate);//Tue Mar 12 16:32:59 CST 2019
```

Timezones(时区)

在新API中时区使用 `ZoneId` 来表示。时区可以很方便的使用静态方法`of`来获取到。抽象类`ZoneId` (在 `java.time`包中) 表示一个区域标识符。它有一个名为`getAvailableZoneIds`的静态方法，它返回所有区域标识符。

```
//输出所有区域标识符
System.out.println(ZoneId.getAvailableZoneIds());

ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Brazil/East");
System.out.println(zone1.getRules());// ZoneRules[currentStandardOffset=+01:00]
System.out.println(zone2.getRules());// ZoneRules[currentStandardOffset=-03:00]
```

LocalTime(本地时间)

`LocalTime` 定义了一个没有时区信息的时间，例如 晚上10点或者 17:30:15。下面的例子使用前面代码创建的时区创建了两个本地时间。之后比较时间并以小时和分钟为单位计算两个时间的时间差：

```
LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);
System.out.println(now1.isBefore(now2)); // false

long hoursBetween = ChronoUnit.HOURS.between(now1, now2);
long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);

System.out.println(hoursBetween); // -3
System.out.println(minutesBetween); // -239
```


LocalTime 提供了多种工厂方法来简化对象的创建，包括解析时间字符串。

```
LocalTime late = LocalTime.of(23, 59, 59);
System.out.println(late);           // 23:59:59
DateTimeFormatter germanFormatter =
    DateTimeFormatter
        .ofLocalizedTime(FormatStyle.SHORT)
        .withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);
System.out.println(leetTime);      // 13:37
```

LocalDate(本地日期)

LocalDate 表示了一个确切的日期，比如 2014-03-11。该对象值是不可变的，用起来和LocalTime基本一致。下面的例子展示了如何给Date对象加减天/月/年。另外要注意的是这些对象是不可变的，操作返回的总是一个新实例。

```
LocalDate today = LocalDate.now(); //获取现在的日期
System.out.println("今天的日期: "+today); //2019-03-12
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
System.out.println("明天的日期: "+tomorrow); //2019-03-13
LocalDate yesterday = tomorrow.minusDays(2);
System.out.println("昨天的日期: "+yesterday); //2019-03-11
LocalDate independenceDay = LocalDate.of(2019, Month.MARCH, 12);
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
System.out.println("今天是周几: "+dayOfWeek); //TUESDAY
```

从字符串解析一个 LocalDate 类型和解析 LocalTime 一样简单,下面是使用 `DateTimeFormatter` 解析字符串的例子:


```
String str1 = "2014==04==12 01时06分09秒";
// 根据需要解析的日期、时间字符串定义解析所用的格式器
DateTimeFormatter fomatter1 = DateTimeFormatter
    .ofPattern("yyyy==MM==dd HH时mm分ss秒");

LocalDateTime dt1 = LocalDateTime.parse(str1, fomatter1);
System.out.println(dt1); // 输出 2014-04-12T01:06:09

String str2 = "2014$$$四月$$$13 20小时";
DateTimeFormatter fomatter2 = DateTimeFormatter
    .ofPattern("yyy$$$$MMM$$$$dd HH小时");
LocalDateTime dt2 = LocalDateTime.parse(str2, fomatter2);
System.out.println(dt2); // 输出 2014-04-13T20:00
```

再来看一个使用 `DateTimeFormatter` 格式化日期的示例

```
LocalDateTime rightNow=LocalDateTime.now();
String date=DateTimeFormatter.ISO_LOCAL_DATE_TIME.format(rightNow);
System.out.println(date);//2019-03-12T16:26:48.29
DateTimeFormatter formatter=DateTimeFormatter.ofPattern("YYYY-MM-dd HH:mm:ss");
System.out.println(formatter.format(rightNow));//2019-03-12 16:26:48
```

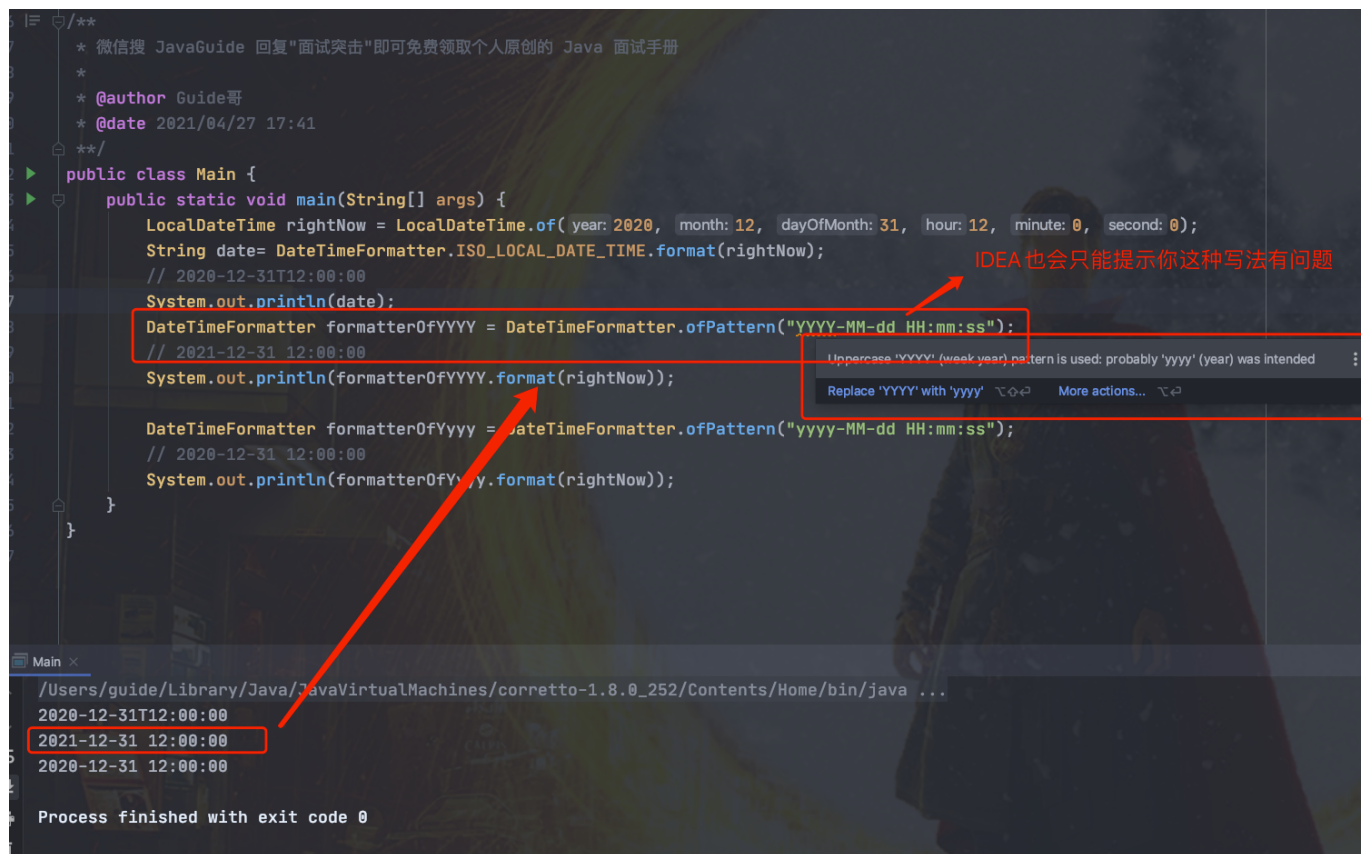
 **修正 (参见: [issue#1157](#))** : 使用 `YYYY` 显示年份时, 会显示当前时间所在周的年份, 在跨年周会有问题。一般情况下都使用 `yyyy`, 来显示准确的年份。

跨年导致日期显示错误示例:

```
LocalDateTime rightNow = LocalDateTime.of(2020, 12, 31, 12, 0, 0);
String date= DateTimeFormatter.ISO_LOCAL_DATE_TIME.format(rightNow);
// 2020-12-31T12:00:00
System.out.println(date);
DateTimeFormatter formatterOfYYYY = DateTimeFormatter.ofPattern("YYYY-MM-dd HH:mm:ss");
// 2021-12-31 12:00:00
System.out.println(formatterOfYYYY.format(rightNow));

DateTimeFormatter formatterOfYyyy = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
// 2020-12-31 12:00:00
System.out.println(formatterOfYyyy.format(rightNow));
```

从下图可以更清晰的看到具体的错误, 并且 IDEA 已经智能地提示更倾向于使用 `yyyy` 而不是 `YYYY`。



LocalDateTime(本地日期时间)

LocalDateTime 同时表示了时间和日期，相当于前两节内容合并到一个对象上了。LocalDateTime 和 LocalTime 还有 LocalDate 一样，都是不可变的。LocalDateTime 提供了一些能访问具体字段的方法。

```
LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23, 59, 59);

DayOfWeek dayOfWeek = sylvester.getDayOfWeek();
System.out.println(dayOfWeek); // WEDNESDAY

Month month = sylvester.getMonth();
System.out.println(month); // DECEMBER

long minuteOfDay = sylvester.getLong(ChronoField.MINUTE_OF_DAY);
System.out.println(minuteOfDay); // 1439
```

只要附上时区信息，就可以将其转换为一个时间点Instant对象，Instant时间点对象可以很容易的转换为老式的java.util.Date。

```
Instant instant = sylvester
    .atZone(ZoneId.systemDefault())
    .toInstant();

Date legacyDate = Date.from(instant);
System.out.println(legacyDate); // Wed Dec 31 23:59:59 CET 2014
```

格式化LocalDateTime和格式化时间和日期一样的，除了使用预定义好的格式外，我们也可以自己定义格式：

```
DateTimeFormatter formatter =  
    DateTimeFormatter  
        .ofPattern("MMM dd, yyyy - HH:mm");  
LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter);  
String string = formatter.format(parsed);  
System.out.println(string);    // Nov 03, 2014 - 07:13
```

和java.text.NumberFormat不一样的是新版的DateTimeFormatter是不可变的，所以它是线程安全的。关于时间日期格式的详细信息在[这里](#)。

Annotations(注解)

在Java 8中支持多重注解了，先看个例子来理解一下是什么意思。首先定义一个包装类Hints注解用来放置一组具体的Hint注解：

```
@Retention(RetentionPolicy.RUNTIME)  
@interface Hints {  
    Hint[] value();  
}  
@Repeatable(Hints.class)  
@interface Hint {  
    String value();  
}
```

Java 8允许我们把同一个类型的注解使用多次，只需要给该注解标注一下@Repeatable即可。

例 1: 使用包装类当容器来存多个注解（老方法）

```
@Hints({@Hint("hint1"), @Hint("hint2")})  
class Person {}
```

例 2: 使用多重注解（新方法）

```
@Hint("hint1")  
@Hint("hint2")  
class Person {}
```

第二个例子里java编译器会隐性的帮你定义好@Hints注解，了解这一点有助于你用反射来获取这些信息：

```
Hint hint = Person.class.getAnnotation(Hint.class);  
System.out.println(hint);    // null  
Hints hints1 = Person.class.getAnnotation(Hints.class);
```

```
System.out.println(hints1.value().length); // 2

Hint[] hints2 = Person.class.getAnnotationsByType(Hint.class);
System.out.println(hints2.length); // 2
```

即便我们没有在 `Person` 类上定义 `@Hints` 注解，我们还是可以通过 `getAnnotation(Hints.class)` 来获取 `@Hints` 注解，更加方便的方法是使用 `getAnnotationsByType` 可以直接获取到所有的 `@Hint` 注解。另外Java 8 的注解还增加到两种新的target上了：

```
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
@interface MyAnnotation {}
```

Where to go from here?


关于Java 8的新特性就写到这了，肯定还有更多的特性等待发掘。JDK 1.8里还有很多很有用的东西，比如 `Arrays.parallelSort`, `StampedLock` 和 `CompletableFuture` 等等。

公众号

如果大家想要实时关注我更新的文章以及分享的干货的话，可以关注我的公众号。

《**Java面试突击**》：由本文档衍生的专为面试而生的《Java面试突击》V2.0 PDF 版本[公众号](#)后台回复 "**Java面试突击**" 即可免费领取！

Java工程师必备学习资源：一些Java工程师常用学习资源[公众号](#)后台回复关键字 "**1**" 即可免费无套路获取。

 我的公众号