# 操作Excel

<http://www.cnblogs.com/huajiezh/p/5467821.html>

Java操作Excel文件有两种常用的方式：jxl和poi。

poi和jxl的区别：

网上有人做过测试，当数据量较小时（<30000），jxl的处理速度由于poi方式；当数据量较大时，poi由于jxl。但是jxl只能处理xls的文件，现已停止更新，而poi方式可以处理xls跟xlsx文件。

综上，使用Java从操作Excel时选择poi方式更好。

## POI介绍

POI是Apache软件基金会为Java操作Microsoft Office提供支持的开放API。它是一个开源项目。通过它，Java开发人员可以实现对Office文件的新建、读写等操作。

### POI包结构

1. HSSF 提供读写Microsoft Excel XLS格式档案的功能。
2. XSSF 提供读写Microsoft Excel OOXML XLSX格式档案的功能。
3. HWPF 提供读写Microsoft Word DOC格式档案的功能。
4. HSLF 提供读写Microsoft PowerPoint格式档案的功能。
5. HDGF 提供读Microsoft Visio格式档案的功能。
6. HPBF 提供读Microsoft Publisher格式档案的功能。
7. HSMF 提供读Microsoft Outlook格式档案的功能。

其中，操作Excel需要HSSF和XSSF模块。

## HSSF模块

HSSF模块主要用于操作.xls格式的文档。它里面包括了xls文档的各种结构：

1、HSSFWorkbook excel文档对象

2、HSSFSheet excel的sheet

3、HSSFRow excel的行

4、HSSFCell excel的单元格

5、HSSFFont excel字体

6、HSSFName 名称

7、HSSFDataFormat 日期格式

8、HSSFHeader sheet 头

9、HSSFFooter sheet 尾

10、HSSFCellStyle cell样式

11、HSSFDateUtil 日期

12、HSSFPrintSetup 打印

13、HSSFErrorConstants 错误信息表

### HSSFWorkbook

HSSFWorkbook用于创建一个Excel文本对象，同时能够设置Excel的一些文本级属性。比如设置Excel中的sheet属性，设置文档包含几个sheet，那个sheet为显示状态等。

HSSFWorkbook提供的方法：

1. 构造

#### 构造方法

HSSFWorkbook提供了多个构造方法用于创建一个Excel文档。这里介绍几种常用的构造方式。

使用构造方法创建一个HSSFWorkbook实例后，还需要通过HSSFWorkbook提供的write方法将创建的Excel文档输出到本地。此时，write方法接收一个输出流参数，可以在这里指定文档的生成位置。

1. 创建一个空的Excel文档

@Test  
**public void** testExcel() **throws** IOException{  
 HSSFWorkbook workbook = **new** HSSFWorkbook();  
 OutputStream os = **null**;  
 **try** {  
 os = **new** FileOutputStream(**"C:\\Users\\hand\\Desktop\\samle.xls"**);  
 workbook.write(os);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*结束\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 } **finally** {  
 os.close();  
 }  
}

1. POIFSFileSystem类是专门用来读取excel表格的，它可以把Excel文件作为数据流来进行传入传出。

# 注解

注解是JDK 1.5及以后版本引入的一个特性。与类、接口、枚举在同一层次。

注解也叫元数据，用于在代码的源文件、编译文件和运行时提供一些注释、说明。我们可以根据这些注释对代码进行相应的处理。

## 元注解

元注解的主要作用就是注解其他注解。

也就是在定义注解的时候，可以再使用元注解进行注解，指定自定义注解的作用对象等属性。

Java5提供了四个标准的元注解：

1、@Target

2、@Retention

3、@Documented

4、@Inherited

### @Target

@Target注解用于设置注解可修饰对象。也就是可以在哪个地方使用该注解。

@Target可以设置值：

1. @Target(ElementType.CONSTRUCTOR) 用于修饰构造器，也就是注解只能在构造器方法上声明
2. @Target(ElementType.METHOD) 用于修饰方法
3. @Target(ElementType.FIELD) 用于修饰字段、枚举常量
4. @Target(ElementType.PARAMETER) 用于修饰方法中参数
5. @Target(ElementType.PACKAGE) 用于修饰包
6. @Target(ElementType.LOCAL\_VARIABLE) 用于修饰局部变量
7. @Target(ElementType.TYPE) 用于修饰类、接口、注解、枚举
8. @Target(ElementType.ANNOTATION\_TYPE) 用于修饰注解7

@Documented  
@Retention(RetentionPolicy.***RUNTIME***)  
@Target(ElementType.***ANNOTATION\_TYPE***)  
**public** @**interface** Target {  
 *//可以设置多个值，格式为*{ElementType.***METHOD***, ElementType.***TYPE***}ElementType[] value();  
}

### @Retention

@Retention用于定义注解的保留策略，也就是注解存在于代码的哪个阶段，比如源码阶段、编译阶段、运行阶段。（注解的生命周期）

@Retention注解提供了三种保留策略：

1、@Retention(RetentionPolicy.SOURCE) 注解仅存在于源码中，在class字节码文件中不包含

2、@Retention(RetentionPolicy.CLASS) 默认的保留策略，注解会在class字节码文件中存在，但运行时无法获得

3、@Retention(RetentionPolicy.RUNTIME) 注解在源码和class字节码文件中都存在，在运行时可以通过反射获取到

通过上面介绍可知，三种保留策略的声明周期长度：SOURCE < CLASS < RUNTIME

前者能起作用的地方，后者也能起作用。一般如果我们需要在运行时解析注解，则必须用RUNTIME类型；如果要在编译时进行一些预处理操作，比如生成一些辅助代码（如 ButterKnife），就用 CLASS注解；如果只是做一些检查性的操作，比如 @Override 和 @SuppressWarnings，则可选用 SOURCE 注解。

### @Documented

@Document是一个没有成员的标记注解。被它修饰的注解可以被Javadoc等工具文档化，也就是生成API。

### @Inherited

@Inherited 元注解是一个标记注解。设置注解是否可继承。

一个注解如果使用了@Inherited修饰，则这个注解是可继承的。那么当这个注解修饰一个类时，这个类的子类也会自动继承该注解。

@Target(ElementType.***METHOD***)  
@Retention(RetentionPolicy.***RUNTIME***)  
@Documented  
@Inherited  
**public** @**interface** MyAnnocation {  
 String funName(); *// 如果没有设置默认值，则使用该注解时必须设置该成员值* String methodName() **default "METHOD"**;  
 String className() **default "CLASS"**;  
 **boolean** isSuit() **default true**;  
}

使用注解：

@MyAnnocation(funName = **"printVal"**)  
**public void** printVal(String val){  
 System.***out***.println(**"hello world"**+val);  
}

## 内置注解

JDK 自身提供的注解称为内置注解。、

JDK 5.0提供了三个常用注解：

1、@Deprecated 废弃，表示被修饰的资源已经或即将废弃，不建议使用这些资源

2、@Override 覆盖，主要用于子类继承或实现接口后，重写方法时使用

3、@SuppressWarnings 取消警告，忽略某些代码产生的特殊警告

### @Deprecated

是一个标注注解。

当我们想要告诉编译器一个方法或字段已经被弃用时，可以使用该注解修饰。这样当其他人引用该注解修饰的资源时会发出警告。

@Deprecated 注解一般和javadoc配合使用。在javadoc中告诉其他人这个弃用方法的替代。也就是哪个新的方法可以实现它的功能及如何使用。

@Documented  
@Retention(RetentionPolicy.***RUNTIME***)  
@Target(value={***CONSTRUCTOR***, ***FIELD***, ***LOCAL\_VARIABLE***, ***METHOD***, ***PACKAGE***, ***PARAMETER***, ***TYPE***})  
**public** @**interface** Deprecated {  
}

### @Override

是一个标注注解。

当我们重写父类方法时，可以使用该注解告诉编译器我们正在覆盖父类的一个方法。这样当父类的这个方法被删除或修改时，编译器就会提示错误信息。

@Target(ElementType.***METHOD***)  
@Retention(RetentionPolicy.***SOURCE***)  
**public** @**interface** Override {  
}

### @SuppressWarnings

这个注解仅仅是告知编译器，忽略它们产生了特殊警告，比如：在java泛型中使用原始类型。它的保持性策略(retention policy)是SOURCE，在编译器中将被丢弃。

@Target({***TYPE***, ***FIELD***, ***METHOD***, ***PARAMETER***, ***CONSTRUCTOR***, ***LOCAL\_VARIABLE***})  
@Retention(RetentionPolicy.***SOURCE***)  
**public** @**interface** SuppressWarnings {

String[] value();

}

## 自定义注解

创建自定义注解与编写接口比较相似，只是必须使用@interface关键字声明注解。

在定义注解时，不能继承其他的注解或接口。@interface用来声明一个注解，注解中的每一个方法实际上是声明了一个配置参数。方法的名称就是参数的名称，返回值类型就是参数的类型（返回值类型只能是基本类型、Class、String、enum）。可以通过default来声明参数的默认值。

@Target(ElementType.***METHOD***)  
@Retention(RetentionPolicy.***RUNTIME***)  
@Documented  
@Inherited  
**public** @**interface** MyAnnocation {  
  
 String funName(); *// 如果没有设置默认值，则使用该注解时必须设置该成员值* String methodName() **default "METHOD"**;  
 String className() **default "CLASS"**;  
 **boolean** isSuit() **default true**;  
  
}

注意：

1. 自定义注解必须通过@interface声明
2. 自定义注解中对的每一个方法都类似于对象的一个参数，方法名是参数名，返回类型是参数值类型。
3. 自定义注解中的方法可以通过 default 关键字设置默认值。如果没有设置默认值，在使用该注解时，必须为注解的该方法设值。
4. 自定义注解中方法的返回值必须是基本类型：

4.1、所有基本数据类型（int,float,boolean,byte,double,char,long,short)

4.2、String类型

4.3、Class类型

4.4、enum类型

4.5、以上所有类型的数组

## 通过映射解析注解

解析注解：即通过反射获取类、函数或成员上的运行时注解信息，从而实现动态控制程序运行的逻辑。

注解解析首先针对的是运行时注解，通过元注解@Retention控制。其次，运行时注解是可以控制程序的执行逻辑的。比如Spring中的@RequestMapping注解，就可以根据url选择不同的方法处理请求。

Java中的Class.java文件中只提供了isAnnotation方法用于判断当前类是不是注解，但Class实现了AnnotatedElement接口，这个接口中提供了大量与注解相关的方法，这为通过反射解析运行时注解提供了基础。

**public final class** Class<T> **implements** java.io.Serializable,  
 GenericDeclaration,  
 Type,  
 AnnotatedElement {

**public boolean** isAnnotation()

}

### AnnotatedElement接口

AnnotatedElement表示正在运行的程序中的一个注释元素。当一个类被运行时注解修饰时，就可以通过该接口提供的方法解析运行时注解。

方法介绍：

#### isAnnotationPresent

isAnnotationPresent方法判断当前元素上是否存在指定类型的注释。存在则返回true，不存在则false。

**default boolean** isAnnotationPresent(Class<? **extends** Annotation> annotationClass)

#### getAnnotation

getAnnotation方法用于获取当前元素上是否含有指定类型的注释。如果存在，则返回该注释实例，不存在则返回null。

<T **extends** Annotation> T getAnnotation(Class<T> annotationClass);

#### getAnnotations

返回当前元素上的所有注释实例。

Annotation[] getAnnotations();

#### getDeclaredAnnotation

返回直接存在于此元素上的指定类型的注解（不包括父类的注解）

<T **extends** Annotation> T getDeclaredAnnotation(Class<T> annotationClass)

#### getDeclaredAnnotations

返回直接存在于此元素上的所有注解（不包括父类的注解），调用者可以随意修改返回的数组；这不会对其他调用者返回的数组产生任何影响，没有则返回长度为0的数组

Annotation[] getDeclaredAnnotations();

### 解析注解实例

通过反射解析注解，我们可以获取注解提供的辅助信息，然后根据获取的信息对注解修饰类执行不同的处理逻辑。

创建一个普通Java类，使用MyAnnotation自定义注解修饰。

@MyAnnocation(funName = **"this is not a fun"**, className = **"ScanHe.class"**)  
**public class** ScanHe {  
  
 @MyAnnocation(funName = **"printVal"**)  
 **public void** printVal(String val){  
 System.***out***.println(**"hello world"**+val);  
 }  
}

1. 获取ScanHe类上的所有注解，然后获取注解信息

Annotation[] annotations = ScanHe.**class**.getAnnotations();  
**for**(Annotation annotation:annotations){  
 **if**(annotation **instanceof** MyAnnocation){  
 System.***out***.println(((MyAnnocation) annotation).className());  
 System.***out***.println(((MyAnnocation) annotation).funName());  
 System.***out***.println(((MyAnnocation) annotation).isSuit());  
 }  
}

1. 判断ScanHe类有没有被MyAnnotation注解修饰，如果有，就获取注解并解析

**if**(ScanHe.**class**.isAnnotationPresent(MyAnnocation.**class**)){  
 MyAnnocation annocation = ScanHe.**class**.getAnnotation(MyAnnocation.**class**);  
 System.***out***.println(annocation.className());  
 System.***out***.println(annocation.funName());  
 System.***out***.println(annocation.isSuit());  
}

1. 获取ScanHe中的所有方法，并对方法上的注解解析

Method[] methods = ScanHe.**class**.getMethods();  
**for**(Method method:methods){  
 **if**(method.isAnnotationPresent(MyAnnocation.**class**)){  
 MyAnnocation annocation = method.getAnnotation(MyAnnocation.**class**);  
 System.***out***.println(annocation.className());  
 System.***out***.println(annocation.funName());  
 System.***out***.println(annocation.isSuit());  
 }  
}

# Java8新特性

## 接口默认/静态方法

在Java8之前，接口中只能定义抽象方法。即只有方法定义，不能有方法实现。一个实体类如果要实现一个接口，就必须实现接口中所有方法。

Java8提供了对接口的默认方法和静态方法的支持。在接口的默认方法和静态方法中都可以有方法实现。

接口的静态方法只能在接口中使用，也就是子类不能调用其实现接口的静态方法。

接口中默认方法，子类可以直接调用而不必实现。当然，如果有特殊需求，接口中的默认方法不能满足，也可以在子类中重写。

默认方法的应用场景：

1. 对旧的接口进行扩展。在旧的接口中新增一个方法，在Java8以前，需要在该接口的所有实现类中都添加新增方法，比较麻烦。此时可以将新增方法声明为默认方法。
2. 如果接口中提供一个比较通用的方法，其子类可以直接调用，没必要每个实现类都重写该方法。

**void** test();  
  
**default void** concatClassName(){  
 *printStaticMethod*();  
 System.***out***.println(**"this is a default method"**);  
}  
  
**static void** printStaticMethod(){  
 System.***out***.println(**"this is a static method"**);  
}

在实现类中可以直接调用：

@Override  
**public void** test() {  
 System.***out***.println(**"this is a common method"**);  
}  
  
**public static void** main(String[] args){  
 **new** DefaultMethod().test();  
 **new** DefaultMethod().concatClassName();  
}

## 函数式接口

函数式接口在本质上还是一个接口；特殊的一点是函数式接口是SAM类型的接口（Single Abstract Method），接口中只有一个抽象方法。使用@FunctionalInterface进行修饰。

因为函数式接口只有一个抽象方法，使得在以其作为参数的方法中，可以使用Lambda表达式代替。Lambda表达式会自动覆盖函数式接口的抽象方法（个人理解）

Java8内置的四大核心型函数式接口：

1. 函数型接口 Function<T, R>
2. 断言型接口 Predicate<T>
3. 消费型接口 Consumer<T>
4. 供给型接口 Supplier<T>

### 接口声明方式

接口的声明有两种方式，一种是匿名内部类形式；一种是Lambda表达式形式；

#### **匿名内部类方式**

Stream.*of*(**"abc"**,**"dec"**,**"hid"**,**"yhc"**).filter(**new** Predicate <String>() {  
 @Override  
 **public boolean** test(String s) {  
 **return** s.contains(**"c"**);  
 }  
}).forEach(System.***out***::println);

在方法中直接new一个函数式接口对象，实现其抽象方法，进行逻辑处理。

#### Lambda表达式方式

Stream.*of*(**"abc"**,**"dec"**,**"hid"**,**"yhc"**).filter(

s -> s.contains(**"c"**)).forEach(System.***out***::println);

Lambda表达式中的前面部分是方法的参数，后半部分是方法体。如果只有一条语句，可以不用{}；如果有多条语句，需要将方法体用{}括起来。

注意：

Lambda表达式会自动覆盖函数式接口的抽象方法，其抽象方法是什么返回类型，Lambda表达式的方法体重就需要返回什么类型。

比如上面代码，Predicate接口的抽象方法test()返回的是一个boolean型值，则使用Lambda表达式时，方法体也必须返回一个boolean型。如果Lambda表达式后面部分是一条语句，则同等于return s.contains(**"c"**)；如果多条语句，则在{}中要使用return主动返回一个boolean型。

Stream.*of*(**"abc"**,**"dec"**,**"hid"**,**"yhc"**).filter(s ->   
 {**boolean** flag = s.contains(**"c"**);  
 **return** flag;}  
).forEach(System.***out***::println);

### Function接口

Function接口中的唯一抽象方法是apply()。

R apply(T t);

apply()方法接收一个泛型参数，在方法体中进行逻辑处理后再返回一个泛型。它的参数和返回值类型可以相同，可以不同。

Function接口中的apply方法和Consumer接口中的accept接口相比，apply方法可以返回一个泛型。

**new** Function<OmAccounts, Object>() {  
 @Override  
 **public** Object apply(OmAccounts o) {  
 **return null**;  
 }  
}

实现Function接口时需要指定两个参数类型，其中第一个是apply方法参数类型，第二个是apply方法返回值类型。

### Predicate接口

Predicate接口中有五个方法，但只有一个test()方法是抽象方法，是需要实现的。其他四个都是接口默认方法。

**boolean** test(T t);

test()方法返回一个boolean型。所以在使用Predicate接口时，方法体中返回值必须是一个boolean型数据。

IntStream.*of*(1,3,5,6,8,3,9).filter(**new** IntPredicate() {  
 @Override  
 **public boolean** test(**int** value) {  
 **return** value < 5;  
 }  
}).forEach(System.***out***::println);

IntStream.*of*(1,3,5,6,8,3,9).filter(item -> item<5).forEach(System.***out***::println);

上面两种写法是一样的，这个方法是过滤出流中数字小于5的元素，并打印出来。

### Consumer接口

Consumer接口中有2个方法，其中一个抽象方法accept，另一个是默认方法。

Consumer接口是消费型接口，它接收一个参数，且没有返回值。类似与生活中的消费行为。

**void** accept(T t);

accept()方法接收一个泛型参数，在方法体中对参数进行处理，没有返回值。

网上的例子：

**class** Student{  
 String **firstName**;  
 String **lastName**;  
 Double **grade**;  
 Double **feeDiscount** = 0.0;  
 Double **baseFee** = 20000.0;  
  
 **public** Student(String firstName, String lastName, Double grade) {  
 **this**.**firstName** = firstName;  
 **this**.**lastName** = lastName;  
 **this**.**grade** = grade;  
 }  
  
 **public void** printFee(){  
 Double newFee = **baseFee** - ((**baseFee** \* **feeDiscount**) / 100);  
 System.***out***.println(**"The fee after discount: "** + newFee);  
 }  
}

**public static** Student updateStudentFee(Student student, Predicate<Student> predicate, Consumer<Student> consumer){  
 **if** ( predicate.test(student)){  
 *//Use the consumer to update the discount value.* consumer.accept(student);  
 }  
 **return** student;  
}

**public static void** main(String[] args){  
 Student student1 = **new** Student(**"Ashok"**,**"Kumar"**, 9.5);  
 student1 = *updateStudentFee*(student1,  
 student -> student.**grade** > 8.5,  
 student -> student.**feeDiscount** = 30.0);  
 student1.printFee();  
 Student student2 = **new** Student(**"Rajat"**,**"Verma"**, 8.0);  
 student2 = *updateStudentFee*(student2,  
 student -> student.**grade** >= 8,  
 student -> student.**feeDiscount** = 20.0);  
 student2.printFee();  
}

### Supplier接口

Supplier接口中的唯一抽象方法是get()方法。可以在get方法中定义一个数据生成逻辑，然后调用get方法，就会获得一个新的数据。Stream的generate方法的参数就是一个Supplier接口，返回一个无限长的流。

Supplier接口是供给型接口，它的get方法不需要接收参数，但会返回一个泛型，类似于生活中的生产行为。

T get();

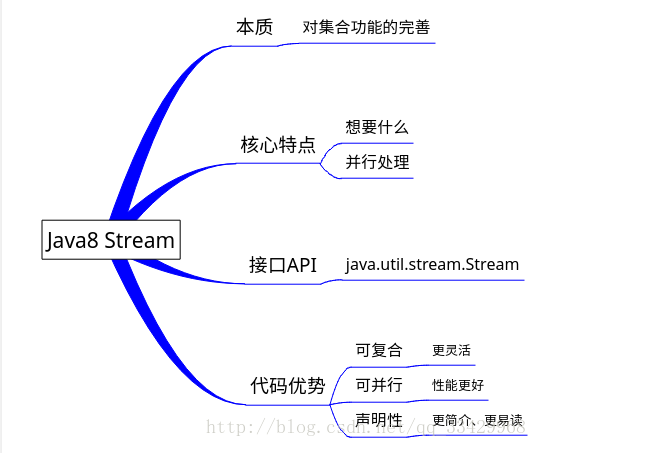
Stream.*generate*(**new** Supplier <Character>() {  
 **char c** = 65;  
 @Override  
 **public** Character get() {  
 **c** =(**char**)(**c**+1);  
 **return c**;  
 }  
}).limit(10).forEach(System.***out***::println);

## Stram流

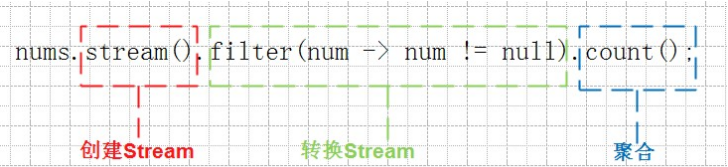
在Java中集合具有非常广泛的应用，当涉及到复杂的业务逻辑处理时，需要借助大量迭代器对象对集合进行多次遍历，才能完成操作。而Java8中Stream的流出现大大简化了Java代码对集合的操作过程。

### 流的定义

* 流是元素的集合。
* 它不是数据结构，不会保存数据，只是操作集合的过程中数据的一种存在形式。
* 在Java8之前，对集合的操作通过迭代对象Iterator完成。如果我们想要集合中的元素执行某个操作，比如选出集合中数字小于10的元素；需要通过迭代器Iterator对集合进行遍历，然后对遍历的每个元素执行函数调用。而对于Stream，如果我们想要对流中的元素执行某个函数，只需要把函数交给Stream即可，具体它是如何遍历元素，如何调用函数的，不需要我们关心。



### Stream语法



关于流的操作基本可以分为以下三个过程：

1. 创建流（Stream）对象，Stream生命开始的地方
2. 转换Stream，也就是流中元素的逻辑处理，每次转换原有Stream对象不改变，返回一个新的Stream对象
3. Stram聚合操作（Reduce和Collect方法）

### 获取Stream实例

最常用的创建Stream对象的两种方法：

1. 通过Stream接口的静态工厂方法（注意：Java8里接口可以带静态方法）；
2. 通过Collection接口的默认方法（默认方法：Default method，也是Java8中的一个新特性，就是接口中的一个带有实现的方法）–stream()，把一个Collection对象转换成Stream

#### Stream接口的静态工厂方法

**Stream.of()**方法接受两种方式的参数，一种是多个值；一种接受单一值；

Stream stream = Stream.*of*(**"abc"**,**"dfc"**,**"hig"**);  
Stream stream1 = Stream.*of*(**"abcde"**);

单一值中可以传递一个字符串，也可以传递一个数组。

**Stream.generator()**方法接受一个Supplier对象，生成一个无限长度的Stream，其元素的生成是通过给定的Supplier（这个接口可以看成一个对象的工厂，每次调用返回一个给定类型的对象）

Stream stream = Stream.*generate*(**new** Supplier <Object>() {  
 **int i** = 1; //自定义元素生成规则，也可以使用Math.random()生成随机数  
 @Override  
 **public** Object get() {  
 **i** = **i** + 2;  
 **return i**;  
 }  
});

Stream的generator方法会生成一个无限长度的Stream，不过这个无限Stream是懒加载的，也就是不调用的时候并不会主动生成元素。

这种无限长Stream一般和Stream的limit()方法配合使用。

Stream.*generate*(**new** Supplier <Object>() {  
 @Override  
 **public** Object get() {  
 **return** Math.*random*();  
 }  
});  
Stream.*generate*(() -> Math.*random*());  
Stream.*generate*(Math::*random*);//方法引用的语法只能在静态方法中使用，generator是静态方法

三条语句的作用都是一样的，只是使用了lambda表达式和方法引用的语法来简化代码。

**Stream.iterate()**方法也是生成无限长度的Stream，和generator不同的是，iterate方法接受两个参数，第一个参数作为种子值，第二个参数是一个函数接口UnaryOperator（可以用Lambda定义），其元素的生成规则是重复对给定的种子值(seed)调用UnaryOperator函数。Stream中的元素可以认为是：seed，f(seed),f(f(seed))无限循环。

Stream.*iterate*(1, num -> num+2).limit(10).forEach(System.***out***::println);

上面这个函数的逻辑就是生成一个包含10个元素的从1（种子值）开始，每次递加2的Stream流，然后输出流中的元素。

#### Collection子类获取Stream

Collection接口有一个stream方法，所以其所有子类都都可以获取对应的Stream对象。如List，Arrays等对象。

Arrays.*asList*(**"abc"**,**"hig"**,**"igc"**).stream();

### 转换Stream

#### filter

filter()方法，接收一个函数接口Predicate参数，对于Stream中包含的元素使用给定的过滤函数进行过滤操作，新生成的Stream只包含符合条件的元素

Predicate接口中有一个test方法，返回一个boolean值。所以，Lambda表达式中的逻辑处理部分必须返回的是boolean型。

Stream<String> stream = Arrays.*asList*(**"abc"**,**"hig"**,**"igc"**).stream();  
stream.filter(str -> str.contains(**"c"**)).forEach(System.***out***::println);

过滤掉流中不含有c的元素，然后将含有c的打印出来

#### map、flatMap

对于Stream中包含的元素使用给定的转换函数进行转换操作，新生成的Stream只包含转换生成的元素。

这个方法有三个对于原始类型的变种方法，分别是：mapToInt，mapToLong和mapToDouble。这三个方法也比较好理解，比如mapToInt就是把原始Stream转换成一个新的Stream，这个新生成的Stream中的元素都是int类型。之所以会有这样三个变种方法，可以免除自动装箱/拆箱的额外消耗；

同样，Stream提供了IntStream、LongStream、DoubleStream对象，避免自动装箱/拆箱的额外消耗；

Stream<String> stream = Arrays.*asList*(**"abc"**,**"hig"**,**"igc"**).stream();  
stream.map(str -> str.concat(**"\*\_\*"**)).forEach(System.***out***::println);

Stream的map方法执行的是流的转换逻辑，它接收的参数是一个函数，函数可以返回任何类型的值。

flatMap方法和map类似，不同的是其每个元素转换得到的是Stream对象，会把子Stream中的元素压缩到父集合中

#### peek

生成一个包含原Stream的所有元素的新Stream，同时会提供一个消费函数（Consumer实例），新Stream每个元素被消费的时候都会执行给定的消费函数；

Stream<T> peek(Consumer<? **super** T> action);

peek()方法接收一个Consumer实例，然后对流中的每个元素执行Consumer实例中的逻辑操作。

#### limit、skip

 limit()方法对一个Stream进行截断操作，获取其前N个元素，如果原Stream中包含的元素个数小于N，那就获取其所有的元素；

skip()方法也是对一个Stream进行截断操作，返回一个丢弃原Stream的前N个元素后剩下元素组成的新Stream，如果原Stream中包含的元素个数小于N，那么返回空Stream；

IntStream stream = IntStream.*iterate*(1, i -> i+1).limit(10);  
stream.limit(5);  
stream.skip(11);

#### 普通方法

forEach()方法用于遍历流中的元素。

distinct()方法用于对Stream中包含的元素进行去重操作（去重逻辑依赖元素的equals方法，如果是实体类对象，有时需要重新equals和hashCode方法），新生成的Stream中没有重复的元素；

IntStream stream = IntStream.*of*(1,2,4,5,2,5,6);  
stream.distinct();

Sort()方法用于对流中的元素进行排序

Stream stream = Stream.*of*(1,2,4,5,2,5,6);  
stream.sorted();

findFirst()方法用于获取Stream流中的第一个元素，如果Stream为空，则返回一个Optional对象。

Stream<Integer> stream = Stream.*of*();  
Optional<Integer> result = stream.findFirst();

anyMatch、allMatch、noneMatch方法都接受一个Predicate函数接口作为参数，返回一个boolean型，判断是否匹配成功。anyMath方法是只要流中存在满足匹配条件的元素，就返回true；allMatch方法要求流中所有的元素都必须满足匹配条件，才返回true；noneMatch方法要求流中的所有元素都不满足匹配条件，返回true。

Stream stream = Stream.*of*(1,2,4,5,2,5,6);  
stream.anyMatch(i -> (**int**)i > 5);  
stream.allMatch(i -> (**int**)i > 5);  
stream.noneMatch(i -> (**int**)i > 5);

### Stream汇聚

汇聚操作（也称为折叠）接受一个元素序列为输入，反复使用某个合并操作，把序列中的元素合并成一个汇总的结果。比如查找一个数字列表的总和或者最大值，或者把这些数字累积成一个List对象。

Stream接口有一些通用的汇聚操作，比如reduce()和collect()；也有一些特定用途的汇聚操作，比如sum(),max()和count()。

汇聚可以分成两部分来看：

1. 可变汇聚：通过反复修改可变对象，把输入的元素们累积到一个可变的容器中，比如Collection或者StringBuilder；
2. 其他汇聚：除了可变汇聚，剩余的都是其他汇聚。它是通过把前一次的汇聚结果当成下一次的入参，反复如此。比如reduce，count，allMatch；

#### collect

可变汇聚对应的只有一个方法：collect，正如其名字显示的，它可以把Stream中的要有元素收集到一个结果容器中（比如Collection）。先看一下最通用的collect方法的定义（还有其他override方法）：

<R> R collect(Supplier<R> supplier,  
 BiConsumer<R, ? super T> accumulator,  
 BiConsumer<R, R> combiner);

先来看看这三个参数的含义：Supplier supplier是一个工厂函数，用来生成一个新的容器；BiConsumer accumulator也是一个函数，用来把Stream中的元素添加到结果容器中；BiConsumer combiner还是一个函数，用来把中间状态的多个结果容器合并成为一个（并发的时候会用到）。

Stream<Integer> stream = Stream.*of*(1,2,4,5,2,5,6);  
stream.collect(() -> **new** ArrayList<Integer>(),  
 (list,item) -> list.add(item),  
 (list1, list2) -> list1.addAll(list2));

上面这段代码就是对一个元素是Integer类型的List，先过滤掉全部的null，然后把剩下的元素收集到一个新的List中。进一步看一下collect方法的三个参数，都是lambda形式的函数。

在上面代码中，collect()方法中的三个参数：

* 1、第一个函数生成一个新的ArrayList实例；
* 2、第二个函数接受两个参数，第一个是前面生成的ArrayList对象，二个是stream中包含的元素，函数体就是把stream中的元素加入ArrayList对象中。第二个函数被反复调用直到原stream的元素被消费完毕；
* 3、第三个函数也是接受两个参数，这两个都是ArrayList类型的，函数体就是把第二个ArrayList全部加入到第一个中；

collect()方法提供了多种接收参数的方式，上面这种参数接收过于麻烦，可以使用下面这种方式：

<R, A> R collect(Collector<? **super** T, A, R> collector);

在这个方法中，我们只需要传递一个Collector对象作为参数即可。另外，Collector一般不需要我们手动构造，Java8提供的Collectors工具类中提供了很多静态方法可以直接返回一个Collector对象，比如toList()、toSet()和toCollection()方法。

Stream<Integer> stream = Stream.*of*(1,2,4,5,2,5,6);  
stream.collect(Collectors.*toList*());

#### reduce

reduce()方法提供了三种实现方式，第一种是接收一个函数接口作为参数

Optional<T> reduce(BinaryOperator<T> accumulator);

这个函数接口接收两个参数

Stream<Integer> stream = Stream.*of*(1,2,4,5,2,5,6);  
Optional<Integer> result = stream.reduce((sum, item) -> sum+=item);  
System.***out***.print(result.get());

第一次执行时，第一个参数是Stream中的第一个元素，第二个参数是Stream中的第二个元素；然后将函数的返回结果赋给第一个参数，Stream流中接下来的元素依次赋给函数中的第二个参数。

第一次运算时：sum = 1；item=2；

第二次运算时：sum = 3；item=4；

……

T reduce(T identity, BinaryOperator<T> accumulator);

这种方式中，reduce接收两个参数，其中第一个参数作为Stream为空的时候的返回值，第二个参数接收一个函数接口。

Stream<Integer> stream = Stream.*of*(1,2,4,5,2,5,6);  
Integer result = stream.reduce(1, (sum, item) -> {System.***out***.println(sum + **" "**+item); sum+=item; **return** sum;});

函数仍旧接收两个参数，此时第一次执行时，第一个参数是reduce()中的第一个参数，第二个参数是Stream中的第一个元素；然后将函数的返回结果赋给第一个参数，Stream流中接下来的元素依次赋给函数中的第二个参数。

第一次运算时：sum = 1；item=1；

第二次运算时：sum = 2；item=2；

……

## Lambda表达式

Lambda表达式更像是一种函数定义方式，是一段带有输入参数的可执行代码块。

与普通函数定义相比，使用Lambda表达式可以让代码更简洁；

函数一般将方法与一个函数名绑定，Lambda表达式不需要与标识符绑定；

### Lambda语法

Lambda表达式的一般语法，

(Type1 param1, Type2 param2, ..., TypeN paramN) -> {  
 statement1;  
 statement2;  
 ......  
 **return** statmentM;  
}

上述是Lambda的一般语法，在实际使用时部分代码还可以省略。

Stream.*of*(**"A"**,**"B"**,**"C"**).map((String s) -> {s = s.concat(**"\_"**); **return** s;}).forEach(System.***out***::println);

这段代码是对Stream中的每个元素都拼接一个\_符号，然后打印。

Lambda表达式省略情形：

1、参数类型省略–绝大多数情况，编译器都可以从上下文环境中推断出lambda表达式的参数类型；

2、当lambda表达式的参数个数只有一个，可以省略小括号；

3、当lambda表达式只包含一条语句时，可以省略大括号、return和语句结尾的分号；

4、在静态方法中可以使用方法引用；

Stream.*of*(**"A"**,**"B"**,**"C"**).map(s -> s.concat(**"\_"**)).forEach(System.***out***::println);

### 访问外部变量

接口中定义的变量只能是final类型，也就是不可变。

接口可以访问接口的外部变量，Lambda表达式也可以，只是Lambda只能访问外部的final修饰的变量。即不可变变量。这里的不变是指引用不变。即如果一个final变量指向了一个对象，那么它将一直指向这个对象，引用不变，但它指向的对象可变。

String[] arry = **new** String[]{**"A"**, **"B"**, **"C"**};  
**for**(Integer i:Arrays.*asList*(1,2,3)){  
 Arrays.*stream*(arry).map(s -> s.concat(i.toString())).forEach(System.***out***::println);  
}

在Lambda表达式中引用了外部变量i，此时i引用不变，引用的对象值改变了。

String[] arry = **new** String[]{**"A"**, **"B"**, **"C"**};  
**for**(Integer i=1; i<=3; i++ ){  
 Arrays.*stream*(arry).map(s -> s.concat(i.toString())).forEach(System.***out***::println);  
}

这种写法是错误的，此时i的引用改变了，不符合final变量定义。

### 方法引用

使用方法引用可以简化代码。但方法引用的使用有条件限制。**方法引用的唯一用途是支持Lambda的简写**

方法引用是用来直接访问类或者实例的已经存在的方法或者构造方法。方法引用提供了一种引用而不执行方法的方式，它需要由兼容的函数式接口构成的目标类型上下文。

当Lambda表达式中只是执行一个方法调用时，不用Lambda表达式，直接通过方法引用的形式可读性更高一些。方法引用是一种更简洁易懂的Lambda表达式。

1、引用静态方法   
ContainingClass::staticMethodName   
例子: String::valueOf，对应的Lambda：(s) -> String.valueOf(s)   
比较容易理解，和静态方法调用相比，只是把.换为::

2、引用特定对象的实例方法   
containingObject::instanceMethodName   
例子: x::toString，对应的Lambda：() -> this.toString()   
与引用静态方法相比，都换为实例的而已

3、引用构造函数   
ClassName::new   
例子: String::new，对应的Lambda：() -> new String()   
构造函数本质上是静态方法，只是方法名字比较特殊。

4、引用特定类型的任意对象的实例方法   
ContainingType::methodName   
例子: String::toString，对应的Lambda：(s) -> s.toString()   
太难以理解了。难以理解的东西，也难以维护。建议还是不要用该种方法引用。   
实例方法要通过对象来调用，方法引用对应Lambda，Lambda的第一个参数会成为调用实例方法的对象。

## Optional类

Optional是Java8提供的为了解决null安全问题的一个API。正确使用Optional可以使我们代码中很多繁琐、丑陋的设计变得十分优雅。

Optional是一个类，不是接口。

Optional是一个可以为null的容器对象，如果值在则isPresent()方法会返回true，调用get()方法会返回该对象。

### 获取Optional实例

Optional类中提供了三种方法用于获取Optional实例。

1、empty()方法

**private static final** Optional<?> ***EMPTY*** = **new** Optional<>();

**public static**<T> Optional<T> empty() {  
 @SuppressWarnings(**"unchecked"**)  
 Optional<T> t = (Optional<T>) ***EMPTY***;  
 **return** t;  
}

empty()方法返回的是一个新的Optional对象。但Optional中的值为空。

2、of()方法

**public static** <T> Optional<T> of(T value) {  
 **return new** Optional<>(value);  
}

of方法接收一个泛型对象作为参数，如果参数value为null，代码运行时会直接抛出空指针异常。因此，使用of方法获取Optional实例时，应该确保value不为null。

3、ofNullable()方法

**public static** <T> Optional<T> ofNullable(T value) {  
 **return** value == **null** ? *empty*() : *of*(value);  
}

ofNullable方法也是接收一个泛型对象作为参数，和of方法的区别是，如果参数value为null，会调用empty方法自动创建一个新的Optional对象

### Optional普通方法

isPresent()

isPresent方法用于判断Optional对象当中的值是否为null，是则返回false；否则返回true。

LambdaTest lambdaTest = **null**;  
System.***out***.println(Optional.*ofNullable*(lambdaTest).isPresent()); //false

get()

get方法用于获取Optional对象中的值，如果没值则抛出NoSuchElementException异常。

LambdaTest lambdaTest = **new** LambdaTest();  
Optional.*of*(lambdaTest).get();

ifParsent()

ifParsent方法接收一个消费型接口Consumer对象作为参数。

注意：

Lambda表达式方法参数是Optional对象中的值，不是Optional对象本身。

LambdaTest lambdaTest = **new** LambdaTest();  
Optional.*of*(lambdaTest).ifPresent(u -> System.***out***.println(u **instanceof** LambdaTest));

orElse()/orElseGet()/orElseThrow()

orElse方法，如果Optional实例有值则将其返回，否则返回orElse方法传入的参数

orElseGet与orElse方法类似，区别在于得到的默认值。orElse方法将传入的参数作为默认值，orElseGet方法可以接受[Supplier接口](http://blog.sanaulla.info/2013/04/02/supplier-interface-in-java-util-function-package-in-java-8/)的实现用来生成默认值。

在orElseThrow中我们可以传入一个lambda表达式或方法，如果值不存在来抛出异常。

在orElse和orElseGet方法中，传入的参数和Supplier接口生成的默认值都必须和Optional对象中的值是同一个类型。

LambdaTest lambdaTest = **new** LambdaTest();  
Optional.*of*(lambdaTest).orElse(**new** LambdaTest());  
Optional.*of*(lambdaTest).orElseGet(LambdaTest::**new**);  
Optional.*of*(lambdaTest).orElseThrow(NullPointerException::**new**);

### map()

map方法中接收一个函数型接口Function作为参数。在执行时，map会先判断Optional对象中的值是不是为空，如果值为空，调用empty方法新建一个空的Optional对象；如果不为空，则调用Function函数对Optional的值进行处理。

### flatMap()

flatMap方法也是接收一个函数型接口Function作为参数。它和map方法类似，也是用于对Optional对象进行处理，不同的是flatmap返回的是Optional对象，而map方法需要对返回值进行Optional封装。

### filter()

filter个方法通过传入限定条件对Optional实例的值进行过滤。然后返回满足条件的值。

## DateTime日期API

http://www.blogjava.net/4cai/archive/2014/04/29/413041.html