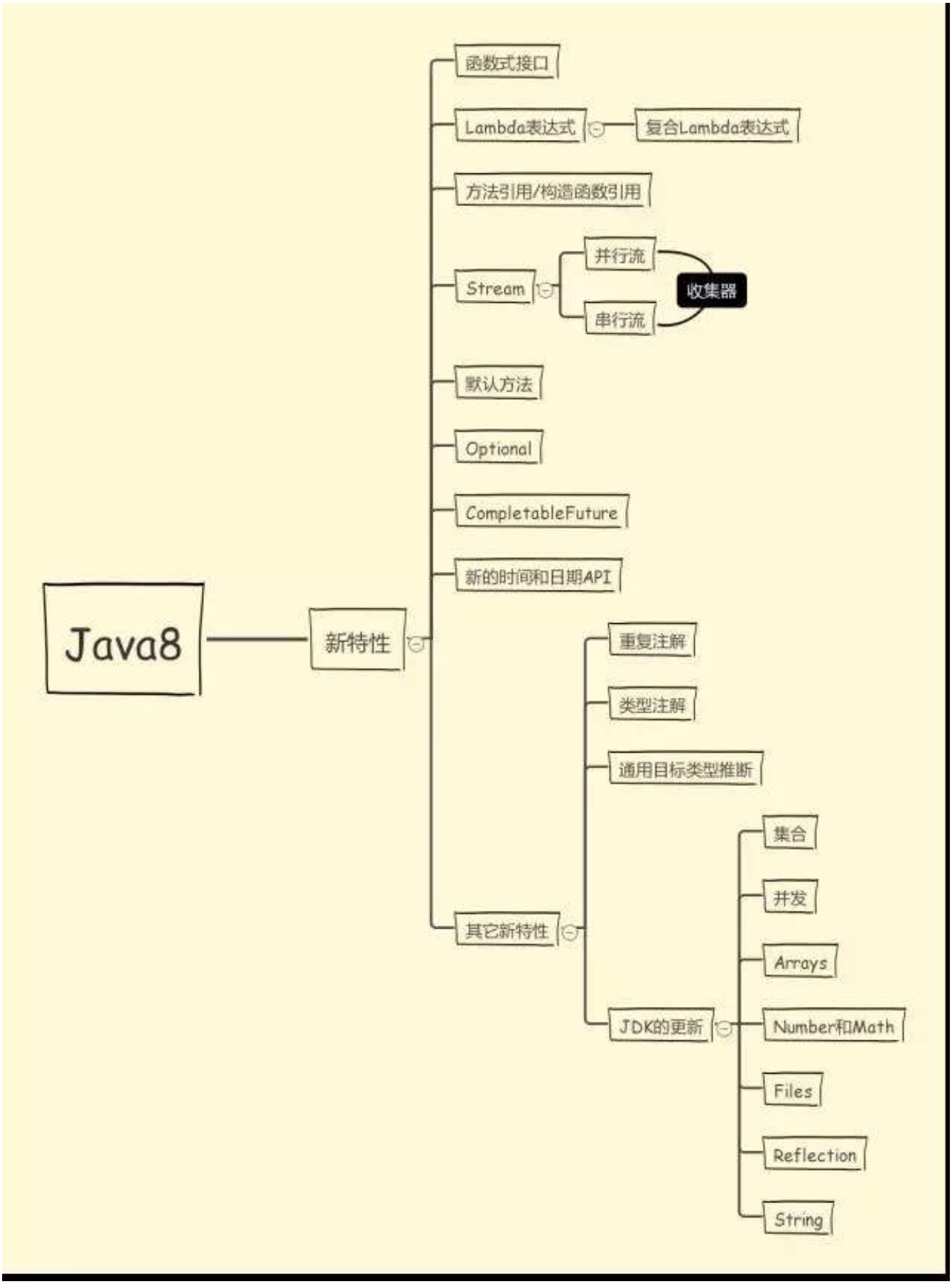
Java8 学习笔记

2017-04-25 ImportNew

> 来源：Listen，
> listenzhangbin.com/post/2017/01/java8-learning-notes/
> 如有好文章投稿，请点击 → 这里了解详情

Java8是2014年发布的，至今也已经有快三年的时间了，之前虽然有学习过，但是学的比较零散，不成系统，而且也没有覆盖到Java8所有的特性。 由于公司已经使用了JDK1.8，所以工作中能使用Java8的机会还是很多的，因此决定来系统地学习一下Java8的新特性，这是对我最近学习Java8的一些记录， 以备在有些细节记不太清的时候可以查询。



先来一个概览，上图是我整理的Java8中的新特性，总的来看，大致上可以分

成这么几个大块。

函数式接口

所谓的函数式接口就是只有一个抽象方法的接口，注意这里说的是抽象方法，因为Java8中加入了默认方法的特性，但是函数式接口是不关心接口中有没有默认方法的。 一般函数式接口可以使用@FunctionalInterface注解的形式来标注表示这是一个函数式接口，该注解标注与否对函数式接口没有实际的影响，不过一般还是推荐使用该注解，就像使用@Override注解一样。JDK1.8中提供了一些函数式接口如下：

函数式接口	函数描述符	原始类型特化
Predicate<T>	T -> boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	T -> void	IntConsumer, LongConsumer, DoubleConsumer
Function<T,R>	T -> R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>
Supplier<T>	() -> T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T,T) -> T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<L,R>	(L,R) -> boolean	
BiConsumer<T,U>	(T,U) -> void	ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>
BiFunction<T,U,R>	(T,U) -> R	ToIntBiFunction<T,U>, ToLongBiFunction<T,U>, ToDoubleBiFunction<T,U>

上表中的原始类型特化指的是为了消除自动装箱和拆箱的性能开销，JDK1.8提供的针对基本类型的函数式接口。

Lambda表达式和方法引用

有了函数式接口之后，就可以使用Lambda表达式和方法引用了。其实函数式接口的表中的函数描述符就是Lambda表达式，在函数式接口中Lambda表达式相当于匿名内部类的效果。 举个简单的例子：

```
public class TestLambda {

    public static void execute(Runnable runnable) {
        runnable.run();
    }

    public static void main(String[] args) {
        //Java8之前
        execute(new Runnable() {
            @Override
            public void run() {
                System.out.println("run");
            }
        });

        //使用Lambda表达式
        execute(() -> System.out.println("run"));
    }
}
```

可以看到，相比于使用匿名内部类的方式，Lambda表达式可以使用更少的代码但是有更清晰的表述。注意，Lambda表达式也不是完全等价于匿名内部类

的， 两者的不同点在于this的指向和本地变量的屏蔽上。

Lambda表达式还可以复合，把几个Lambda表达式串起来使用：

```
Predicate<Apple> redAndHeavyApple = redApple.and(a -> a.getWeight() > 150).or(a ->
  “green” .equals(a.getColor()));
```

上面这行代码把两个Lambda表达式串了起来，含义是选择重量大于150或者绿色的苹果。

方法引用可以看作Lambda表达式的更简洁的一种表达形式，使用::操作符，方法引用主要有三类：

1. 指向静态方法的方法引用(例如Integer的parseInt方法，写作Integer::parseInt)；
2. 指向任意类型实例方法的方法引用(例如String的length方法，写作String::length)；
3. 指向现有对象的实例方法的方法引用(例如假设你有一个本地变量localVariable用于存放Variable类型的对象，它支持实例方法getValue，那么可以写成localVariable::getValue)。

举个方法引用的简单的例子：

```
Function<String, Integer> stringToInteger = (String s) -> Integer.parseInt(s);


//使用方法引用
Function<String, Integer> stringToInteger = Integer::parseInt;
```

方法引用中还有一种特殊的形式，构造函数引用，假设一个类有一个默认的构造函数，那么使用方法引用的形式为：

```
Supplier<SomeClass> c1 = SomeClass::new;
SomeClass s1 = c1.get();

//等价于

Supplier<SomeClass> c1 = () -> new SomeClass();
SomeClass s1 = c1.get();
```

如果是构造函数有一个参数的情况：

```
Function<Integer, SomeClass> c1 = SomeClass::new;
SomeClass s1 = c1.apply(100);

//等价于

Function<Integer, SomeClass> c1 = i -> new SomeClass(i);
SomeClass s1 = c1.apply(100);
```

Stream

Stream可以分成串行流和并行流，并行流是基于Java7中提供的ForkJoinPool来进行任务的调度，达到并行的处理的目的。 集合是我们平时在进行Java编程时非常常用的API，使用Stream可以帮助更好的来操作集合。Stream提供了非常丰富的操作，包括筛选、切片、映射、查找、匹配、归约等等， 这些操作又可以分为中间操作和终端操作，中间操作会返回一个流，因此我们可以使用多个中间操作来作链式的调用，当使用了终端操作之后，那么这个流就被认为是被消费了， 每个流只能有一个终端操作。

```
//筛选后收集到一个List中
List<Apple> vegetarianMenu = apples.stream().filter(Apple::isRed).collect(Collectors.toList());
```

```
//筛选加去重
List<Integer> numbers = Arrays.asList(1,2,1,3,3,2,4);
numbers.stream().filter(i -> i % 2 == 0).distinct().forEach(System.out::println);
```

以上都是一些简单的例子，Stream提供的API非常丰富，可以很好的满足我们的需求。

操作	类型	返回类型	使用的类型/函数式接口	函数描述符
filter	中间	Stream<T>	Predicate<T>	T -> boolean
distinct	中间	Stream<T>		
skip	中间	Stream<T>	long	
limit	中间	Stream<T>	long	
map	中间	Stream<R>	Function<T,R>	T -> R
flatMap	中间	Stream<R>	Function<T, Stream<R>>	T -> Stream<R>
sorted	中间	Stream<R>	Comparator<T>	(T,T) -> int
anyMatch	终端	boolean	Predicate<T>	T -> boolean
noneMatch	终端	boolean	Predicate<T>	T -> boolean
allMatch	终端	boolean	Predicate<T>	T -> boolean
findAny	终端	Optional<T>		
findFirst	终端	Optional<T>		
forEach	终端	void	Consumer<T>	T -> void
collect	终端	R	Collector<T,A,R>	
reduce	终端	Optional<T>	BinaryOperator<T>	(T,T) -> T
count	终端	long		

与函数式接口类似，Stream也提供了原始类型特化的流，比如说IntStream等：

```
//maoToInt转化为一个IntStream
```

```
int count = list.stream().mapToInt(list::getNumber).sum();
```

并行流与串行流的区别就在于将stream改成parallelStream，并行流会将流的操作拆分，放到线程池中去执行，但是并不是说使用并行流的性能一定好于串行的流， 恰恰相反，可能大多数时候使用串行流会有更好的性能，这是因为将任务提交到线程池，执行完之后再合并，这些本身都是有不小的开销的。关于并行流其实还有非常多的细节， 这里做一个抛砖引玉，有兴趣的同学可以在网上自行查找一些资料来学习。

默认方法

默认方法出现的原因是为了对原有接口的扩展，有了默认方法之后就不怕因改动原有的接口而对已经使用这些接口的程序造成的代码不兼容的影响。 在Java8中也对一些接口增加了一些默认方法，比如Map接口等等。一般来说，使用默认方法的场景有两个：可选方法和行为的多继承。

默认方法的使用相对来说比较简单，唯一要注意的点是如何处理默认方法的冲突。关于如何处理默认方法的冲突可以参考以下三条规则：

1. 类中的方法优先级最高。类或父类中声明的方法的优先级高于任何声明为默认方法的优先级。

2. 如果无法依据第一条规则进行判断，那么子接口的优先级更高：函数签名相同时，优先选择拥有最具体实现的默认方法的接口。即如果B继承了A，那么B就比A更具体。

3. 最后，如果还是无法判断，继承了多个接口的类必须通过显式覆盖和调用期望的方法，显式地选择使用哪一个默认方法的实现。那么如何显式地指定呢:

```
public class C implements B, A {
```

```
    public void hello() {
        B.super().hello();
    }
}
```

使用X.super.m(..)显式地调用希望调用的方法。

Optional

如果一个方法返回一个Object，那么我们在使用的时候总是要判断一下返回的结果是否为空，一般是这样的形式：

```
if (a != null) {
    //do something...
}
```

但是简单的情况还好，如果复杂的情况下每一个都要去检查非常麻烦，而且写出来的代码也不好看、很臃肿，但是如果不检查就很容易遇到NullPointerException， Java8中的Optional就是为此而设计的。

Optional一般使用在方法的返回值中，如果使用Optional来包装方法的返回值，这就表示方法的返回值可能为null，需要使用Optional提供的方法来检查，如果为null，还可以提供一个默认值。

```
//创建Optional对象
Optional<String> opt = Optional.empty();

//依据一个非空值创建Optional
Optional<String> opt = Optional.of("hello");

//可接受null的Optional
Optional<String> opt = Optional.ofNullable(null);
```

除了以上这些方法外，Optional还提供了以下方法：

方法	描述
empty	返回一个空的Optional实例
filter	如果值存在并且满足提供的谓词，就返回包括该值的Optional对象；否则返回一个空的Optional对象
flatMap	如果值存在，就对该值执行提供的mapping函数调用，返回一个Optional类型的值，否则就返回一个空的Optional对象
get	如果该值存在，将该值用Optional封装返回，否则抛出一个NoSuchElementException异常
ifPresent	如果值存在，就执行使用该值的方法调用，否则返回false
isPresent	如果值存在就返回true，否则返回false
map	如果值存在，就对该值执行提供的mapping函数调用
of	将指定值用Optional封装之后返回，如果该值为null，抛出一个NullPointerException异常
ofNullable	将指定值用Optional封装之后返回，如果该值为null，则返回一个空的Optional对象
orElse	如果有值则将其返回，否则返回一个默认值
orElseGet	如果有值则将其返回，否则返回一个由指定的Supplier接口生成的值
orElseThrow	如果有值则将其返回，否则抛出一个由指定的Supplier接口生成的异常

CompletableFuture

在Java8之前，我们会使用JDK提供的Future接口来进行一些异步的操作，其实CompletableFuture也是实现了Future接口， 并且基于ForkJoinPool来执行任务，因此本质上来讲，CompletableFuture只是对原有API的封装， 而使用CompletableFuture与原来的Future的不同之处在于可以将两个Future组合起来，或者如果两个Future是有依赖关系的，可以等第一个执行完毕后再实行第二个等特性。

先来看看基本的使用方式：

```
public Future<Double> getPriceAsync(final String product) {
    final CompletableFuture<Double> futurePrice = new CompletableFuture<>();
    new Thread(() -> {
        double price = calculatePrice(product);
        futurePrice.complete(price);  //完成后使用complete方法，设置future的返回值
    }).start();
    return futurePrice;
}
```

得到Future之后就可以使用get方法来获取结果，CompletableFuture提供了一些工厂方法来简化这些API，并且使用函数式编程的方式来使用这些API，例如：

```
Fufure<Double> price = CompletableFuture.supplyAsync(() -> calculatePrice(product));
```

代码是不是一下子简洁了许多呢。之前说了，CompletableFuture可以组合多个Future，不管是Future之间有依赖的，还是没有依赖的。 如果第二个请求依赖于第一个请求的结果，那么可以使用thenCompose方法来组合两个Future

```
public List<String> findPriceAsync(String product) {
    List<CompletableFutute<String>> priceFutures = tasks.stream()
    .map(task -> CompletableFuture.supplyAsync(() -> task.getPrice(product),executor))
    .map(future -> future.thenApply(Work::parse))
    .map(future -> future.thenCompose(work -> CompletableFuture.supplyAsync(() ->
Count.applyCount(work), executor)))
    .collect(Collectors.toList());

    return priceFutures.stream().map(CompletableFuture::join).collect(Collectors.toList());
}
```

上面这段代码使用了thenCompose来组合两个CompletableFuture。

supplyAsync方法第二个参数接受一个自定义的Executor。 首先使用CompletableFuture执行一个任务，调用getPrice方法，得到一个Future，之后使用thenApply方法，将Future的结果应用parse方法， 之后再使用执行完parse之后的结果作为参数再执行一个applyCount方法，然后收集成一个CompletableFuture<String>的List， 最后再使用一个流，调用CompletableFuture的join方法，这是为了等待所有的异步任务执行完毕，获得最后的结果。

注意，这里必须使用两个流，如果在一个流里调用join方法，那么由于Stream的延迟特性，所有的操作还是会串行的执行，并不是异步的。

再来看一个两个Future之间没有依赖关系的例子：

```
Future<String> futurePriceInUsd = CompletableFuture.supplyAsync(() -> shop.getPrice(“price1”))
                        .thenCombine(CompletableFuture.supplyAsync(() -> shop.getPrice(“price2”)), (s1,
s2) -> s1 + s2);
```

这里有两个异步的任务，使用thenCombine方法来组合两个Future，thenCombine方法的第二个参数就是用来合并两个Future方法返回值的操作函数。

有时候，我们并不需要等待所有的异步任务结束，只需要其中的一个完成就可以了，CompletableFuture也提供了这样的方法：

```
//假设getStream方法返回一个Stream<CompletableFuture<String>>
CompletableFuture[] futures = getStream(“listen”).map(f ->
f.thenAccept(System.out::println)).toArray(CompletableFuture[]::new);
//等待其中的一个执行完毕
CompletableFuture.anyOf(futures).join();
```

使用anyOf方法来响应CompletableFuture的completion事件。

新的时间和日期API

Java8之前的时间和日期API并不好用，而且在线程安全性等方面也存在问题，一般会借助一些开源类库来解决时间处理的问题。在JDK1.8中新加入了时间和日期的API， 借助这些新的API基本可以不再需要开源类库的帮助来完成时间的处理了。

Java8中加入了LocalDateTime, LocalDate, LocalTime, Duration, Period, Instant, DateTimeFormatter等等API，来看一些使用这些API的简单的例子：

```
//创建日期
LocalDate date = LocalDate.of(2017,1,21); //2017-01-21
int year = date.getYear() //2017
Month month = date.getMonth(); //JANUARY
int day = date.getDayOfMonth(); //21
DayOfWeek dow = date.getDayOfWeek(); //SATURDAY
int len = date.lengthOfMonth(); //31(days in January)
boolean leap = date.isLeapYear(); //false(not a leap year)

//时间的解析和格式化
LocalDate date = LocalDate.parse(“2017-01-21”);
LocalTime time = LocalTime.parse(“13:45:20”);

LocalDateTime now = LocalDateTime.now();
now.format(DateTimeFormatter.BASIC_ISO_DATE);

//合并日期和时间
LocalDateTime dt1 = LocalDateTime.of(2017, Month.JANUARY, 21, 18, 7);
LocalDateTime dt2 = LocalDateTime.of(localDate, time);
LocalDateTime dt3 = localDate.atTime(13,45,20);
LocalDateTime dt4 = localDate.atTime(time);
LocalDateTime dt5 = time.atDate(localDate);
```

```
//操作日期
LocalDate date1 = LocalDate.of(2014,3,18); //2014-3-18
LocalDate date2 = date1.plusWeeks(1); //2014-3-25
LocalDate date3 = date2.minusYears(3); //2011-3-25
LocalDate date4 = date3.plus(6, ChronoUnit.MONTHS); //2011-09-25
```

可以发现，新的时间和日期API都是不可变的，并且是线程安全的，之前使用的比如SimpleDateFormat不是线程安全的， 现在可以使用DateTimeFormatter来代替，DateTimeFormatter是线程安全的。

以上只是Java8提供的新时间和日期API的一部分，更多的内容可以参考官网文档，有了这些API，相信完全可以不再依赖开源的类库来进行时间的处理。

小结

以上只是对Java8的新特性进行了一个非常简单的介绍，由于近年来函数式编程很火，Java8也受函数式编程思想的影响，吸收了函数式编程好的地方， 很多新特性都是按照函数式编程来设计的。关于Java8还有非常多的细节没有提到，这些需要我们自行去学习，推荐一本学习Java8非常好的书籍——《Java8实战》， 看完这本书对Java8的使用可以有一个比较清楚的了解。

现在已经是2017年了，据说今年会推出Java9，Java9会推出什么新特性，让我们拭目以待吧。

Read more Views 13326 26 Report

Top Comments

Write a comment

南浦云 9

看不懂拉姆达表达式，干了8年Java。

5 day(s) ago

编程界的小学生 6

StreamAPI和函数式接口几乎没讲，JAVA8核心都丢了，还是精品吗？

5 day(s) ago

是非 3

正在看《Java8实战》，确实写的非常好，尤其是其中的例子。

5 day(s) ago

Dube 2

期待更多java9内容

5 day(s) ago

韩武洽 1

我看了JAVA8的视频 但是真正遇到JAVA8能处理的问题 还是不太容易能解决的?
有没有JAVA8的Demo? 让我参考,顺便学习一下?
先谢谢了!

5 day(s) ago

Roman Wang 1

据说Java9会推出类似irb, repl的interactive console.

5 day(s) ago

Chandler钱 1

5 day(s) ago

Resurrect

学习了下函数式编程，感觉Java引入函数式编程非常有意义

4 day(s) ago

Most upvoted comments above

Learn about writing a valuable comment