Java8 实战

1. 行为参数化

- **@FunctionalInterface**
  - 此注解会检查接口是不是函数式接口，如果不是，则会报错："*Multiple non-overriding abstract methods found in interface Foo*"
- **倒序**
  - list.sort(comparing(Apple::getWeight). `reversed` ());
    - 比较 `reverseOrder()` 方法
- **比较器链**
  - list.sort(comparing(Apple::getWeight) . `thenComparing` (Apple::getCountry));
- **谓语复合**
  - `Predicate` 接口的 `negate` 、 `and` 和 `or` 方法
    - 不是红苹果，eg: Predicate notRedApple = redApple. `negate` ()
- **函数复合**
  - `andThen` 和 `compose` ，参考

```
Function<Integer, Integer> times2 = e -> e * 2;
Function<Integer, Integer> squared = e -> e * e;

times2.compose(squared).apply(4);
// Returns 32   先compose中的函数，再执行times2
times2.andThen(squared).apply(4);
// Returns 64   先执行times2，再执行compose中的函数
```

2. 方法引用

- `方法引用` 就是 `Lambda` 表达式的快捷写法
  - eg：(Apple a) -> a.getWeight() 等效 Apple::getWeight()
- 方法引用有三类
  - `静态方法` ： `Integer` 的 `parseInt` 方法，写作 `Integer::parseInt`
  - `参数类型的实例方法` ： `(Apple a) -> a.getWeight()` 可以换成 `Apple::getWeight`
  - `Lambda外的对象的实例方法` ：

```
package com.example.demo.multi.springBoot;

import com.example.demo.multi.springBoot.entity.User;
import org.junit.Test;
import java.util.function.Consumer;

public class EasyTester {

    @Test
    public void test() {
        EasyTester easyTester = new EasyTester();  // Lambda外的对象easyTester
        apply(easyTester::printUserName);  // 方法引用，引用easyTester对象的printUserName方法
    }

    public void apply(Consumer<? super User> function){
        User user = new User();
```

```
        user.setName("Tom");
        user.setAge(10);
        function.accept(user);
    }

    public void printUserName(User user) {
        System.out.println("The user's name is " + user.getName());
    }
}
```

- 其他方法引用类型
  - 构造方法引用：`ClassName::new`，与指向静态方法的引用类似

3. Stream

- `Guava`、`Apache`和`lambdaj`等库也都实现了类似的流功能
- 流的源可以是`集合`、`数组`或`输入/输出资源`，从有序集合生成流时会保留原有的顺序
- 流的操作方法：`filter`、`map`、`reduce`、`find`、`match`、`sort`、`flatMap`
- 关于`flatMap`

```
List<String> uniqueCharacters = words.stream()
        .map(w -> w.split(""))
        .flatMap(Arrays::stream)
        .distinct()
        .collect(Collectors.toList());
```

  `flatMap`类似于降维；

- `集合`讲的是数据，`流`讲的是计算
  - *流和集合的区别*就好比`DVD`和`流媒体`的区别：集合着急着创建数据，然后再处理；流则是边创建边使用；
- 流只能被消费一次：

```
Stream<String> s = list.stream();
s.forEach(System.out::println);
s.forEach(System.out::println);  // 这里会报错
```

- 流是`内部迭代`，而集合是`外部迭代`；
- 流的操作类型
  - 中间操作：filter、map、limit、reduce、match、sort、distinct、skip、flatMap（流的扁平化）等
  - 终端操作：collect、forEach、count、reduce
- 流的查找与匹配
  - allMatch、anyMatch、noneMatch、findFirst、findAny
- `limit`操作与`短路`技巧、`匹配`操作与`短路`

> 有些操作不需要处理整个流就能得到结果。例如，假设你需要对一个用 and 连起来的大布 尔表达式求值。不管表达式有多长，你只需找到一个表达式为 false，就可以推断整个表达式 将返回 false，所以用不着计算整个表达式。这就是短路。
>
> 对于流而言，某些操作（例如 allMatch、anyMatch、noneMatch、findFirst 和 findAny） 不用处理整个流就能得到结果。只要找到一个元素，就可以有结果了。同样，limit 也是一个 短路操作：它只需要创建一个给定大小的流，而用不着处理流中所有的元素。在碰到无限大小 的流的时候，这种操作就有用了：它们可以把无限流变成有限流。

- `《Java8实战》`*P105*页 - 一个优秀的求勾股数的实例，以及优化实例；

3.1 Optional 类

- Optional 类（java.util.Optional）是一个容器类，代表一个值存在或不存在。
- 关键 API
  - isPresent()
  - ifPresent(Consumer block)
  - T get() - 值存在时返回值，否则抛出一个 NoSuchElement 异常
  - T orElse(T other) - 值存在时返回值，否则返回一个默认值

3.2 归约

- reduce

```
int sum = numbers.stream().reduce(0, Integer::sum);
// 无初始值
Optional<Integer> sum = numbers.stream().reduce((a, b) -> (a + b));  // 因为，如果没有初始值，并且数组为空的话，不知道返回什么，所以，这里返回Optional类
```

扩展

- **归约方法的优势与并行化**、

```
// 如果使用For-each实现并行，那简直就是折磨
// 但要并行执行这段代码也要满足一定的条件：传递给reduce的Lambda不能更改状态（如实例变量），而且操作必须满足结合律才可以按任意顺序执行。
int sum = numbers.parallelStream().reduce(0, Integer::sum);
```

3.3 流的状态

- `sort`或`distinct`等操作，从流中排序和删除重复项时都需要知道*先前的历史*，我们把这些操作叫作**有状态操作**；

3.4 有界和无界

- 无界
  - 排序要求所有元素都放入缓冲区后才能给输出流加入一个项目，这一操作的存储要求是无界的；
- 有界
  - 如 reduce、sum、max 等操作需要内部状态来累积结果。在我们的例子里就是一个 int 或 double，而 int 只需要 4 个字节，double 只需要 8 个字节；
- 有界无界 - 帮助理解：

> 如果一个操作为了能够获取结果需要添加内部状态来进行记录，那么这个操作就是有状态的，比如求和操作，需要一个内部状态来进行累加，而像 filter，map 等操作是完全不需要额外内部状态的，因此他们是无状态的。如果一个操作是有状态的，且这个操作需要访问的流的长度是可确定的，比如 max，min 等，访问长度为 1，那么这个操作又是有界的，如果需要访问的流的长度不能确定(随着流的长度变化而变化)，比如 distinct，sort 等操作，这有可能需要访问所有流元素后，才能得出结果，比如一个质数流倒序，这将是不能完成的，所以这类操作是无界的。对于无状态操作来说，并行是十分容易的，对于有状态有界的操作来说，精心设计小心使用也是可以在并行的情况下工作良好的，但是对于有状态且无界的操作来说，并行将变得比较困难且不可控。

3.5 流操作总结

操作	类型	返回类型
filter	中间	Stream<T>

distinct	中间 (有状态-无界)	Stream<T>
skip	中间 (有状态-有界)	Stream<T>
limit	中间 (有状态-有界)	Stream<T>
map	中间	Stream<T>
flatMap	中间	Stream<T>
sorted	中间 (有状态-无界)	Stream<T>
anyMatch	终端	boolean
noneMatch	终端	boolean
allMatch	终端	boolean
findAny	终端	Optional
findFirst	终端	Optional
forEach	终端	void
collect	终端	R
reduce	终端 (有状态-有界)	Optional
count	终端	long

3.6 流中基本类型装箱问题

- `IntStream`、`DoubleStream`和`LongStream`

> IntStream、DoubleStream 和 LongStream，分别将流中的元素特化为 int、long 和 double，从而避免了暗含的装箱成本。每 个接口都带来了进行常用数值归约的新方法，比如对数值流求和的 sum，找到最大元素的 max。 此外还有在必要时再把它们转换回对象流的方法 。

- 将流转换为特化版本的常用方法是 mapToInt、mapToDouble 和 mapToLong
- 特化流转非特化流 - intStream.boxed()
- **特化流 API**
  - IntStream.range(1, 100)
  - IntStream.rangeClosed(1, 100)

3.7 流的创建

- Stream.of()

```
Stream<String> stream = Stream.of("Java 8 ", "Lambdas ", "In ", "Action");
```

- 创建空流

```
Stream<String> emptyStream = Stream.empty();
```

- 数组创建流

```
Arrays.stream(numbers)
```

- 由文件创建流

```
// Files.lines得到一个流，其中的每个元素都是给定文件中的一行
Stream<String> lines = Files.lines(Paths.get("data.txt"), Charset.defaultCharset())

// 读取classpath下的文件内容
Files.lines(Paths.get(ClassLoader.getSystemResource("banner.txt").toURI())).forEach(line ->
System.out.println(line));
```

- `Stream.iterate` 和 `Stream.generate`

> Stream API 提供了两个静态方法来从函数生成流：Stream.iterate 和 Stream.generate。 这两个操作可以创建所谓的无限流：不像从固定集合创建的流那样有固定大小的流。由 iterate 和 generate 产生的流会用给定的函数按需创建值，因此可以无穷无尽地计算下去！一般来说， 应该使用 limit(n)来对这种流加以限制，以避免打印无穷多个值。

```
// iterate
Stream.iterate(0, n -> n + 2)
        .limit(10)
        .forEach(System.out::println);
// generate
Stream.generate(Math::random)
        .limit(5)
        .forEach(System.out::println);
```

  - 《Java8 实战》P109 - 副作用的例子 —— *匿名类和 Lambda 的区别*

> 相比之下，使用 iterate 的方法则是纯粹不变的：它没有修改现有状态， 但在每次迭代时会创建新的元组。你将在第 7 章了解到，你应该始终采用不变的方法

3.8 终端方法

- `collect`

```
// 分组
Map<Currency, List<Transaction>> transactionsByCurrencies =
transactions.stream().collect(groupingBy(Transaction::getCurrency));
```

- `groupingBy`

  - 一级分组

```
menu.stream().collect(groupingBy(Dish::getType));
```

  - 二级分组

```
// 结果是个两级Map
menu.stream().collect(groupingBy(Dish::getType,
        groupingBy(dish -> {
                if (dish.getCalories() <= 400)
                        return CaloricLevel.DIET;
                else if (dish.getCalories() <= 700)
                        return CaloricLevel.NORMAL;
                else
                        return CaloricLevel.FAT;
        })));
```

> Collectors.groupingBy 工厂方法创 建的收集器，它除了普通的分类函数之外，还可以接受 collector 类型的第二个参数；所以，双参数的 groupingBy 的第二个参数不一定要是 groupingBy，也可以是 Collectors.counting 收集器，而结果就是 *{MEAT=3, FISH=2, OTHER=4}*

    - 普通的单参数 groupingBy(f)（其中 f 是分类函数）实际上是 groupingBy(f, toList())的简便写法。

- `partitioningBy` 是 `groupingBy` 的特殊情况
- `Collectors.characteristics` 方法决定是否可以使用并行方式处理流 - *《Java8 实战》P133*
- 想策底明白终端操作，就必须明白 `Collector` 接口中的 5 个方法；

3.9 自定义收集器

以《Java8 实战》中的举例为展示示例：位置-查看书籍的 `6.5.1 理解Collector 接口声明的方法`

3.9.1 实现类签名

public interface Collector <T, A, R>

- T - 流中元素的类型
- A - 用于累积部分结果的对象类型
- R - collect 操作最终结果的类型

例： *《Java8 实战》- p137*

```
public class PrimeNumbersCollector implements Collector<Integer, Map<Boolean, List<Integer>>, Map<Boolean,
List<Integer>>> {

}
```

3.9.2 实现归约

3.9.2.1 supplier 方法

提供收集器对象：Supplier<A> supplier()

```
public Supplier<Map<Boolean, List<Integer>>> supplier() {
 return () -> new HashMap<Boolean, List<Integer>>() {{ // 这里的初始化也很有意思
     put(true, new ArrayList<Integer>());
     put(false, new ArrayList<Integer>());
 }};
}
```

3.9.2.2 accumulator - 收集器

```
public BiConsumer<Map<Boolean, List<Integer>>, Integer> accumulator() {
 return (Map<Boolean, List<Integer>> acc, Integer candidate) -> {
        acc.get(isPrime(acc.get(true), candidate))
                .add(candidate);
 };
}

/* 判断是否是质数，注意此方法局限性：此方法是在求取连续0-n中质数列时使用到的； */
public static boolean isPrime(List<Integer> primes, int candidate){
 int candidateRoot = (int) Math.sqrt((double) candidate);
 return takeWhile(primes, i -> i <= candidateRoot)
     .stream()
     .noneMatch(p -> candidate % p == 0);
}
```

3.9.3 多线程支持 - combiner

```
public BinaryOperator<Map<Boolean, List<Integer>>> combiner() {
        return (Map<Boolean, List<Integer>> map1, Map<Boolean, List<Integer>> map2) -> {
         map1.get(true).addAll(map2.get(true));
         map1.get(false).addAll(map2.get(false));
         return map1;
    };
}
```

> 实际上这个收集器是不能并行使用的，因为该算法本身是顺序的。这意味着永远都不会调用 combiner 方法，你可以把它的实现留空（更好的做法是抛出一个 UnsupportedOperationException 异常）。为了让这个例子完整，我们还是决定实现它。

3.10 并行与顺序

3.10.1 并行方法

parallel - 对流进行并行执行

> 并行流内部使用了默认的 ForkJoinPool(分支/合并框架)，

- 并行线程数
  - 默认的线程数量就是你的**处理器数量**，这个值是由 `Runtime.getRuntime().availableProcessors()` 得到的
  - 可以通过系统属性 `java.util.concurrent.ForkJoinPool.common.parallelism` 来改变线程池大小：

    > System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism","12");
    > 这是一个全局设置，因此它将影响代码中所有的并行流。反过来说，目前还无法专为某个并行流指定这个值。一般而言，让 ForkJoinPool 的大小等于处理器数量是个不错的默认值，除非你有很好的理由，否则我们强烈建议你不要修改它。

3.10.2 顺序方法

sequential - 对并行流调用 sequential 方法就可以把它变成顺序流

3.10.3 并行/顺序的选择

重要的一点就是：要保证在内核中并行执行工作的时间比在内核之间传输数据的时间长；

反例 1：

```
public static long sideEffectSum(long n) {
     Accumulator accumulator = new Accumulator();
     LongStream.rangeClosed(1, n).forEach(accumulator::add);
     return accumulator.total;
}
public class Accumulator {
     public long total = 0;
     public void add(long value) { total += value; }
}
```

> 那这种代码又有什么问题呢？不幸的是，它真的无可救药，因为它在本质上就是顺序的。每次访问 total 都会出现数据竞争

反例 2

```
public static long sideEffectParallelSum(long n) {
     Accumulator accumulator = new Accumulator();
     LongStream.rangeClosed(1, n).parallel().forEach(accumulator::add);
```

```
    return accumulator.total;
}

System.out.println("SideEffect parallel sum done in: " +
 measurePerf(ParallelStreams::sideEffectParallelSum, 10_000_000L) +"
 msecs" );
// 结果：
//Result: 5959989000692
//Result: 7425264100768
//Result: 6827235020033
//Result: 7192970417739
//Result: 6714157975331
//Result: 7497810541907
//Result: 6435348440385
//Result: 6999349840672
//Result: 7435914379978
//Result: 7715125932481
//SideEffect parallel sum done in: 49 msecs
```

> 此示例的问题是每次执行都会返回不同的结果，都离正确值 50000005000000 差很远。这是由于多个线程在同时访问累加器，执行 total += value，而这一句虽然看似简单，却不是一个原子操作。问题的根源在于，forEach 中调用的方法有副作用，它会改变多个线程共享的对象的可变状态

注：记住要避免共享可变状态

3.10.3.1 选择原则

- **测试** - 手动测试，哪种方式运行性能好；
- **留意装箱** - 自动装箱和拆箱操作会大大降低性能
- **有些操作本身在并行流上的性能就比顺序流差**

  > limit 和 findFirst 等依赖于元素顺序的操作，它们在并行流上执行的代价非常大。例如，findAny 会比 findFirst 性能好，因为它不一定要按顺序来执行。你总是可以调用 unordered 方法来把有序流变成无序流。那么，如果你需要流中的 n 个元素而不是专门要前 n 个的话，对无序并行流调用 limit 可能会比单个有序流（比如数据源是一个 List）更高效。

- **总计算成本**

  > 设 N 是要处理的元素的总数，Q 是一个元素通过流水线的大致处理成本，则 N*Q 就是这个对成本的一个粗略的定性估计。Q 值较高就意味着使用并行流时性能好的可能性比较大。

  个人理解：应该是 N*Q 的值越大，意味着使用并行时性能越好；

- **较小的数据量应避免使用并行**

  因为并行化也是有代价的，进程构建和进程间通信会有性能开销；

- **数据结构是否易于分解**

  如，ArrayList 的拆分效率比 LinkedList 高得多，因为前者用不着遍历就可以平均拆分，而后者则必须遍历。另外，用 range 工厂方法创建的原始类型流也可以快速分解；

源	分解性
ArrayList	极佳
LinkedList	差
IntStream.range	极佳

Stream.iterate	差
HashSet	好
TreeSet	好

- **流自身的特点**

  流水线中的中间操作修改流的方式，也可能会改变分解过程的性能；

- **合并步骤的代价**

3.11 有序变无序

`unordered` 方法

3.12 并行拆分策略

接口：Spliterator - 查看 **《Java8 实战》** - P157

3.13 peek

peek 的设计初衷就是在流的每个元素恢复运行之前，插入执行一个动作。

```
List<Integer> result =
        numbers.stream()
     .peek(x -> System.out.println("from stream: " + x))
     .map(x -> x + 17)
     .peek(x -> System.out.println("after map: " + x))
     .filter(x -> x % 2 == 0)
     .peek(x -> System.out.println("after filter: " + x))
     .limit(3)
     .peek(x -> System.out.println("after limit: " + x))
     .collect(toList());
```

4. Fork/Join 框架

4.0 原理

> **ForkJoinPool**由 ForkJoinTask 数组和 ForkJoinWorkerThread 数组组成，ForkJoinTask 数组负责存放程序提交给 ForkJoinPool 的任务，而 ForkJoinWorkerThread 数组负责执行这些任务。

4.0.1 ForkJoinTask 的 fork 方法实现原理

> 当我们调用 ForkJoinTask 的**fork**方法时，程序会调用 ForkJoinWorkerThread 的**pushTask**方法异步的执行这个任务，然后立即返回结果。
>
> pushTask 方法把当前任务存放在 ForkJoinTask 数组 queue 里，然后再调用 ForkJoinPool 的 signalWork()方法唤醒或创建一个工作线程来执行任务。

4.0.2 ForkJoinTask 的 join 方法实现原理

> Join 方法的主要作用是阻塞当前线程并等待获取结果。

4.1 ExecutorService 接口

4.2 RecursiveTask

> 要把任务提交到这个池（ForkJoinPool），必须创建 RecursiveTask 的一个子类，其中 R 是并行化任务（以及所有子任务）产生的结果类型，或者如果任务不返回结果，则是 RecursiveAction 类型（当然它可能会更新其他非局部机构）。

4.2.1 主要方法

- protected abstract R compute();

  这个方法同时定义了将任务拆分成子任务的逻辑，以及无法再拆分或不方便再拆分时，生成单个子任务结果的逻辑

- **异常处理**

  > ForkJoinTask 在执行的时候可能会抛出异常，但是我们没办法在主线程里直接捕获异常，所以 ForkJoinTask 提供了 isCompletedAbnormally()方法来检查任务是否已经抛出异常或已经被取消了，并且可以通过 ForkJoinTask 的 getException 方法获取异常。
  >
  > getException 方法返回 Throwable 对象，如果任务被取消了则返回 CancellationException。如果任务没有完成或者没有抛出异常则返回 null。

4.3 工作窃取

参考博客

4.3.1 主要特性终结

- 一个线程一个队列；
- 队列是双向的，方便窃取线程从尾端窃取任务；

4.3.2 疑问

- 是否存在多个线程窃取统一个线程的队列中的任务？
- 如果总任务数量是 n，单线程最大任务数为 s，那么，整个计算过程中创建的线程总数是多少？

4.4 使用注意

- 对一个任务调用 join 方法会阻塞调用方，直到该任务做出结果。因此，有必要在两个子任务的计算都开始之后再调用它。否则，你得到的版本会比原始的顺序算法更慢更复杂，因为每个子任务都必须等待另一个子任务完成才能启动；
- 不应该在 RecursiveTask 内部使用 ForkJoinPool 的 invoke 方法。相反，你应该始终直接调用 compute 或 fork 方法，只有顺序代码才应该用 invoke 来启动并行计算；
- 调试使用分支/合并框架的并行计算可能有点棘手。特别是你平常都在你喜欢的 IDE 里面看栈跟踪（stack trace）来找问题，但放在分支 Ĥ 合并计算上就不行了，因为调用 compute 的线程并不是概念上的调用方，后者是调用 fork 的那个；
- 就像任何其他 Java 代码一样，分支/合并框架需要“预热”或者说要执行几遍才会被 JIT 编译器优化；这就是为什么在测量性能之前跑几遍程序很重要；

5. 重构

5.1 匿名类和 Lambda

5.1.1 this 和 super

- **this**

- 匿名类中 - this 代表的是类自身；
- Lambda 中 - this 代表的是包含类；

- 变量**作用域**

  匿名类可以屏蔽包含类的变量，而 Lambda 表达式不能（它们会导致编译错误）：

```
int a = 10;
Runnable r1 = () -> {
     int a = 2;  // 编译错误
     System.out.println(a);
};
```

- **类型**

  匿名类的类型是在初始化时确定的，而 Lambda 的类型取决于它的上下文，例：

```
interface Task{
 public void execute();
}
public static void doSomething(Runnable r){ r.run(); }
public static void doSomething(Task a){ a.execute(); }

// 当调用doSomething时，难以区分参数类型是Runnable还是Task
doSomething(() -> System.out.println("Danger danger!!"));

// 可以尝试使用显式的类型转换来解决这种模棱两可的情况
doSomething((Task)() -> System.out.println("Danger danger!!"));
```

5.2 方法引用

```
inventory.sort((Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
//  使用方法引用
inventory.sort(comparing(Apple::getWeight));
```

6. Java8 与设计模式

设计模式 - 对设计经验的归纳总结被称为设计模式；

6.1 策略模式 - 《Java8 实战》P171

定义：策略模式代表了解决一类算法的通用解决方案，你可以在运行时选择使用哪种策略方；

6.1.1 模式结构

策略模式包含三部分内容：

- 一个代表某个算法的接口；
- 一个或多个该接口的具体实现，它们代表了算法的多种实现，表示不同的策略；

6.2 模板方法 - 《Java8 实战》P172

通俗解释：模板方法模式在你“希望使用这个算法，但是需要对其中的某些行进行改进，才能达到希望的效果”时是非常有用的；

个人理解：先提供一个通用的模板类，模板类中实现了通用的方法逻辑，之后，每个不同的业务对象只要继承模板类，修改自己业务对应的方法逻辑，剩下的方法继承父类的实现；

6.3 观察者模式 - 《Java8 实战》 P173

观察者模式是一种比较常见的方案，某些事件发生时（比如状态转变），如果一个对象（通常我们称之为主题）需要自动地通知其他多个对象（称为观察者），就会采用该方案；

6.4 责任链模式 - 《Java8 实战》 P175

一个处理对象可能需要在完成一些工作之后，将结果传递给另一个对象，这个对象接着做一些工作，再转交给下一个处理对象，以此类推；

- 在 Java8 中，可以使用**Function#andThen**函数代替；

6.5 工厂模式 - 《Java8 实战》 P177

7. Java8 接口默认方法

7.1 接口的扩展

- Java 8 允许在接口内声明静态方法；
- 默认方法；

7.2 jdk8 中的接口和抽象类的区别：

《Java 8 实战》 P191

1. 一个类只能继承一个抽象类，但是一个类可以实现多个接口。
2. 一个抽象类可以通过实例变量（字段）保存一个通用状态，而接口是不能有实例变量的。

7.3 可选方法

用户实现接口的时候可能有些方法不想实现，jdk8 之前，用户会在实现类中提供一个空的方法，而在 jdk8 中，如果接口中提供了默认方法，用户在不想实现某些方法时，就可以省略方法的实现；

7.4 多继承 - 新模式

《Java 8 实战》 P192

```
public interface A {
     void hello() {
         System.out.println("Hello from A");
     }
}
public interface B {
     void hello() {
         System.out.println("Hello from B");
     }
}
public class C implements B, A { }
public class C implements B, A {
     void hello(){
        B.super.hello();
     }
}
```

7.5 菱形继承问题

《Java 8 实战》 P200

8. Optional

8.1 flatMap

flatMap 方法，类似流的 `map` 方法，源码如下：

```
public <U> Optional<U> flatMap(Function<? super T,
```

```
    ? extends Optional<? extends U>> mapper) {
    Objects.requireNonNull(mapper);
    if (!isPresent()) {
        return empty();
    } else {
        @SuppressWarnings("unchecked")
        Optional<U> r = (Optional<U>) mapper.apply(value);
        return Objects.requireNonNull(r);
    }
}
```

8.2 Optional 与序列化

> 由于 Optional 类设计时就没特别考虑将其作为类的字段使用，所以它也并未实现 Serializable 接口。由于这个原因，如果你的应用使用了某些要求序列化的库或者框架，在域模型中使用 Optional，有可能引发应用程序故障

8.3 旧代码改造

8.3.1 异常与 Optional 的对比

《Java 8 实战》 P217 - 10.4.2 异常与 Optional 的对比

OptionalUtility

8.4 基础类型的 Optional

比如：OptionalInt、OptionalLong 以及 OptionalDouble；

> 不推荐大家使用基础类型的 Optional，因为基础类型的 Optional 不支持 map、flatMap 以及 filter 方法，而这些却是 Optional 类最有用的方法。

9. 组合式异步编程

9.1 Future

9.1.1 注意点

- 比起 `future.get()`，其实更推荐使用 `get (long timeout, TimeUnit unit)` 方法，设置了超时时间可以防止程序无限制的等待 future 的结果。

9.2 CompletableFuture

博客 中描述的很详细了，就暂时不再赘述了；

CompletableFuture 能够将回调放到与任务不同的线程中执行，也能将回调作为继续执行的同步函数，在与任务相同的线程中执行。它避免了传统回调最大的问题，那就是能够将控制流分离到不同的事件处理器中。

> CompletableFuture 弥补了 Future 模式的缺点。在异步的任务完成后，需要用其结果继续操作时，无需等待。可以直接通过 thenAccept、thenApply、thenCompose 等方式将前面异步处理的结果交给另外一个异步事件处理线程来处理。

10. 优化进阶

10.1 声明式编程

《Java 8 实战》 P265

10.2 函数式编程

> 当谈论“函数式”时，我们想说的其实是“像数学函数那样——没有副作用”。

- **纯粹的函数式编程**与**函数式编程** - *《Java 8 实战》 P266*
- 要被称为函数式，函数或者方法不应该抛出任何异常 - *《Java 8 实战》 P267*

> 从实际操作的角度出发，你可以选择在本地局部地使用异常，避免通过接口将结果暴露给其他方法，这种方式既取得了函数式的优点，又不会过度膨胀代码。

> 作为函数式的程序，你的函数或方法调用的库函数如果有副作用，你必须设法隐藏它们的非函数式行为，否则就不能调用这些方法（换句话说，你需要确保它们对数据结构的任何修改对于调用者都是不可见的，你可以通过首次复制，或者捕获任何可能抛出的异常实现这一目的）

- **引用透明性** - *《Java 8 实战》 P268*

> 如果一个函数只要传递同样的参数值，总是返回同样的结果，那这个函数就是引用透明的。

- **尾调/尾递** - *《Java 8 实战》 P272*

> 尾调用（Tail Call）是指在调用函数时直接将被调函数的返回值作为调用函数的返回值返回。如果这个调用是调用了调用函数本身（递归调用），这叫做尾递归（Tail Recursive）。使用尾调用的一个优点在于它不消耗额外的调用栈空间，它可以替换当前的栈帧。因此，使用尾调用来替换标准的调用被称作尾调用消除（Tail Call Elimination），或者尾调用优化（Tail Call Optimization）。

尾调定义 - 维基百科：

> **尾调用**是指一个函数里的最后一个动作是返回一个函数的调用结果的情形，即最后一步新调用的返回值直接被当前函数的返回结果。[1]此时，该尾部调用位置被称为**尾位置**。尾调用中有一种重要而特殊的情形叫做**尾递归**。经过适当处理，尾递归形式的函数的运行效率可以被极大地优化。[1]尾调用原则上都可以通过简化函数调用栈的结构而获得性能优化（称为“尾调用消除”），但是优化尾调用是否方便可行取决于运行环境对此类优化的支持程度如何。

注：Java 是不支持尾调/尾递的；

扩展

- **Spliterator**
- *Jdk8*中时间相关类 `LocalDateTime` 和 `DateTimeFormatter`；
  - `Joda-Time` — 优秀的时间 lib；参考
- **synchronized 降低性能的原因**
  - 多核 CPU 的每个处理器内核都有独立的高速缓存。加锁需要这些高速缓存同步运行，然而这又需要在内核间进行较慢的缓存一致性协议通信
- **变量**
  - 思考下面例子的输出结果

```
public class MeaningOfThis {
```

```
    public final int value = 4;   // value

    public void doIt() {
        int value = 6;   // value
        Runnable r = new Runnable() {
            public final int value = 5;   // value

            public void run() {
                int value = 10;   // value
                System.out.println(this.value);
            }
        };
        r.run();
    }

    public static void main(String... args) {
        MeaningOfThis m = new MeaningOfThis();
        m.doIt();
    }
}
```

- `实例变量` 都存储在 `堆` 中，而 `局部变量` 则保存在 `栈` 上
- 下面的代码会报错

```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);  // Variable used in lambda expression should be final or effectively final
portNumber = 31337;
```

  - 第一，实例变量和局部变量背后的实现有一 个关键不同。*实例变量都存储在堆中，而局部变量则保存在栈上*。如果 Lambda 可以直接访问局部变量，而且 Lambda 是在一个线程中使用的，则使用 Lambda 的线程，可能会在分配该变量的线程将这个变量收回之后，去访问该变量。因此，Java 在访问自由局部变量时，实际上是在访问它 的副本，而不是访问原始变量。如果局部变量仅仅赋值一次那就没有什么区别了——因此就有了 这个限制。
  - 第二，这一限制不鼓励你使用改变外部变量的典型命令式编程模式

- **类结构的变更**的影响的方面主要有（《Java 8 实战》 P190）：

  1. 二进制 - 现有的二进制执行文件能无缝持续链接（包括验证、准备和解析）和运行
  2. 源代码 - 引入变化之后，现有的程序依然能成功编译通过
  3. 函数行为 - 变更发生之后，程序接受同样的输入能得到同样的结果

- Java8 新语法 - **super**

  格式： X.super.m(...)

  意义：X 是你希望调用的 m 方法所在的父接口（不是父类，亲测过，父类不行）

  限制：

  - X 只能是接口；
  - 此语法必须存在于重写方法内；
  - 只能取到直接父接口中的 m 方法；

Questions

- 《Java8 实战》 - P168 ： 对“有条件的延迟执行”小节不能理解；