# JAVA高效编程

## 函数式编程

### 1.1函数编程演变史

需求：根据不同条件，筛选出指定条件的商品列表

策略模式-->实体类算法-->匿名内部类算法-->lamba

|  |
| --- |
| */\*\*  \* Sku选择谓词接口  \*/* **public interface** SkuPredicate {  */\*\*  \* 选择判断标准  \** ***@param sku*** *\** ***@return*** *\*/* **boolean** test(Sku sku); } |

|  |
| --- |
| */\*\*  \* V4.0.0  \* 根据不同的Sku判断标准，对Sku列表进行过滤  \** ***@param cartSkuList*** *\** ***@param predicate*** *不同的Sku判断标准策略  \** ***@return*** *\*/* **public static** List<Sku> filterSkus(List<Sku> cartSkuList, SkuPredicate predicate){  List<Sku> result = **new** ArrayList<Sku>();  **for**(Sku sku:cartSkuList){  *// 根据不同的Sku判断标准策略，对Sku进行过滤* **if** (predicate.test(sku)){  result.add(sku);  }  }  **return** result; } |

|  |
| --- |
| @Test **public void** testFilterSkus1(){  List<Sku> cartSkuList = CartService.*getCartSkuList*();  List<Sku> result = CartService.*filterSkus*(  cartSkuList, **new** SkuPredicate() {  **public boolean** test(Sku sku) {  **return** sku.getTotalPrice() > 2000;  }  });   System.***out***.println(JSON.*toJSONString*(  result, **true**)); } |

|  |
| --- |
| @Test **public void** testFilterSkus2(){  List<Sku> cartSkuList = CartService.*getCartSkuList*();  List<Sku> result = CartService.*filterSkus*(  cartSkuList, (Sku sku) -> 2000 < sku.getTotalPrice());   System.***out***.println(JSON.*toJSONString*(  result, **true**)); } |

## 第2节 流式编程

### 2.1集合与流程操作对比

需求：分别使用集合操作及Stream流操作，完成对实际应用场景中的数据处理。直观感受流操作带来的便捷。

|  |
| --- |
| */\*\*  \* 以Stream流方式实现需求  \*/* @Test **public void** newCartHandle(){  AtomicReference<Double> money = **new** AtomicReference<>(Double.*valueOf*(0.0));   List<String> resultSkuNameList = CartService.*getCartSkuList*()  .stream()  *// 打印商品信息* .peek(sku -> System.***out***.println(JSON.*toJSONString*(sku,**true**)))  *// 过滤图书商品* .filter(sku -> !SkuCategoryEnum.***BOOKS***.equals(sku.getSkuCategory()))  *// 排序，默认从大到小* .sorted(Comparator.*comparing*(Sku::getTotalPrice).reversed())  *// TOP 2* .limit(2)  *// 累加商品总金额* .peek(sku -> money.set(money.get() + sku.getTotalPrice()))  *// 获取商品名称* .map(sku -> sku.getSkuName())  *// 收集结果* .collect(Collectors.*toList*());   System.***out***.println(JSON.*toJSONString*(resultSkuNameList, **true**));  System.***out***.println(**"商品总价："** + money);  } |

### 2.2流、简介、与集合区别

·JDK1.8引入的新成员，以声明式处理集合数据

·将基础操作链接起来，完成复杂的数据处理流水线

·提供透明的并行处理

·从支持数据处理操作的源生成的元素序列 ——Java8实战

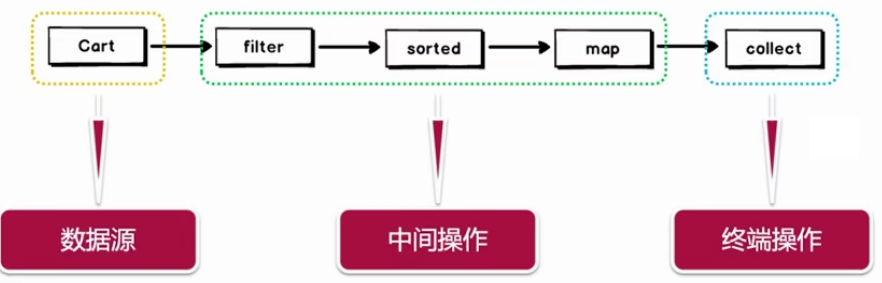
·时间与空间：集合--存储；流--计算

·集合可以遍历多次，流只能遍历一次

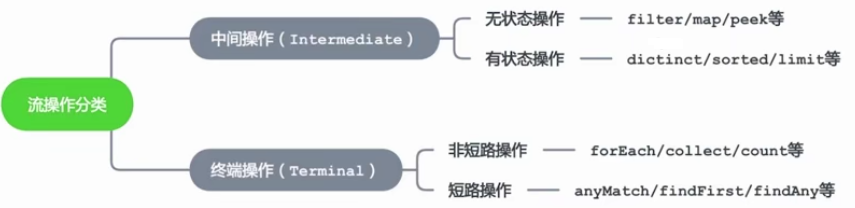
·外部迭代与内部迭代

### 2.3流的组成、分类

·流的组成



·流的分类



·流的分类



2.4流的使用

|  |
| --- |
| */\*\*  \* 演示流的各种操作  \*/* **public class** StreamOperator {  List<Sku> **list**;   @Before  **public void** init(){  **list** = CartService.*getCartSkuList*();  }   */\*\*  \* filter使用：过滤到不符合断言判断的数据  \*/* @Test  **public void** filterTest(){  **list**.stream()  .filter(sku -> SkuCategoryEnum.***BOOKS***.equals(sku.getSkuCategory()))  .forEach(item -> System.***out***.println(JSON.*toJSONString*(item,**true**)));  }   */\*\*  \* map使用：将一个元素转换为另一个元素  \*/* @Test  **public void** mapTest(){  **list**.stream()  .map(sku -> sku.getSkuName())  .forEach(item -> System.***out***.println(JSON.*toJSONString*(item, **true**)));  }   */\*\*  \* flatMap使用：将一个元素转换为另外一个流  \*/* @Test  **public void** filterMap(){  **list**.stream()  .flatMap(sku -> Arrays.*stream*(sku.getSkuName().split(**""**)))  .forEach(item-> System.***out***.println(JSON.*toJSONString*(item, **true**)));  }   */\*\*  \* peek:对流中的元素进行遍历操作，与foreach类似，但不会销毁流元素  \*/* @Test  **public void** peek(){  **list**.stream()  .peek(sku -> System.***out***.println(sku.getSkuName()))  .forEach(item-> System.***out***.println(JSON.*toJSONString*(item, **true**)));  }   */\*\*  \* sort：对流总元素进行排序，可选择自然排序或指定排序规则  \* 有状态操作  \*/* @Test  **public void** sortTest(){  **list**.stream()  .peek(sku -> System.***out***.println(sku.getSkuName()))  *// 有状态操作，peek就无状态需要先完成* .sorted(Comparator.*comparing*(Sku::getTotalPrice))  .forEach(item-> System.***out***.println(JSON.*toJSONString*(item, **true**)));  }   */\*\*  \* distinct：对流元素进行去重  \* 有状态操作  \*/* @Test  **public void** distinctTest(){  **list**.stream()  .map(sku -> sku.getSkuCategory())  .distinct()  .forEach(item-> System.***out***.println(JSON.*toJSONString*(item,**true**)));  }   */\*\*  \* skip:跳过前n条数据  \* 有状态操作  \*/* @Test  **public void** skipTest(){  **list**.stream()  .sorted(Comparator.*comparing*(Sku::getTotalPrice))  .skip(3)  .forEach(item-> System.***out***.println(JSON.*toJSONString*(item,**true**)));  }   */\*\*  \* limit：截断前n条数据  \* 有状态操作  \*/* @Test  **public void** limitTest(){  **list**.stream()  .sorted(Comparator.*comparing*(Sku::getTotalPrice))  .skip(1 \* 3)  .limit(3)  .forEach(item-> System.***out***.println(JSON.*toJSONString*(item, **true**)));  }   */\*\*  \* allMatch使用：终端操作，短路操作，所有元素都匹配返回true  \*/* @Test  **public void** allMatchTest(){  **boolean** match = **list**.stream()  .peek(sku -> System.***out***.println(sku.getSkuName()))  .allMatch(sku -> sku.getTotalPrice() > 100);  System.***out***.println(match);  }   */\*\*  \* anyMatch:任何元素匹配返回true  \*/* @Test  **public void** anyMatchTest(){  **boolean** match = **list**.stream()  .peek(sku -> System.***out***.println(sku.getSkuName()))  .anyMatch(sku -> sku.getTotalPrice() > 100);  System.***out***.println(match);  }   */\*\*  \* noneMatch：任何元素都不匹配返回true  \*/* @Test  **public void** noneMatchTest(){  **boolean** match = **list**.stream()  .peek(sku -> System.***out***.println(sku.getSkuName()))  .noneMatch(sku -> sku.getTotalPrice() > 10000);  System.***out***.println(match);  }   */\*\*  \* findFirst：查找第一个  \* 短路操作  \*/* @Test  **public void** findFirstTest(){  Optional<Sku> optional = **list**.stream()  .findFirst();  System.***out***.println(JSON.*toJSONString*(optional.get(),**true**));  }   */\*\*  \* 查找任意一个  \* 串行操作时，与findFirst没有区别  \*/* @Test  **public void** findAnyTest(){  Optional<Sku> optional = **list**.stream()  .findAny();  System.***out***.println(JSON.*toJSONString*(optional.get(),**true**));  }   */\*\*  \* max使用  \*/* @Test  **public void** maxTest(){  OptionalDouble optionalDouble = **list**.stream()  .mapToDouble(Sku::getTotalPrice)  .max();  System.***out***.println(optionalDouble.getAsDouble());  }   */\*\*  \* min使用  \*/* @Test  **public void** minTest(){  OptionalDouble optionalDouble = **list**.stream()  .mapToDouble(Sku::getTotalPrice)  .min();  System.***out***.println(optionalDouble.getAsDouble());  }   */\*\*  \* 计数  \*/* @Test  **public void** countTest(){  **long** count = **list**.stream()  .count();  System.***out***.println(count);  } } |

### 2.4流的构建

·由值创建

·由数组创建流

·由文件创建流

·由函数创建流（无限流）

|  |
| --- |
| */\*\*  \* 流的四种构建形式  \*/* **public class** StreamConstructor {  */\*\*  \* 由数值直接构建流  \*/* @Test  **public void** streamFromValue(){  Stream stream = Stream.*of*(1,2,3,4,5);  stream.forEach(System.***out***::println);  }   */\*\*  \* 通过数组生成流  \*/* @Test  **public void** streamFromArray(){  **int**[] numbers = {1,2,3,4,5};  IntStream stream = Arrays.*stream*(numbers);  stream.forEach(System.***out***::println);  }   */\*\*  \* 通过文件生成流  \** ***@throws*** *IOException  \*/* @Test  **public void** streamFromFile() **throws** IOException {  Stream<String> stream = Files.*lines*(  Paths.*get*(**"G:\\Effective\_Code\\effective\_code\\src\\test\\java\\com\\byf\\effective\\stream\\StreamConstructor.java"**)  );  stream.forEach(System.***out***::println);  }   */\*\*  \* 通过函数生成流：无限流  \*/* @Test  **public void** streamFromFunction(){  Stream stream = Stream.*iterate*(0,n-> n+2);  Stream stream1 = Stream.*generate*(Math::*random*);  stream1.limit(100).forEach(System.***out***::println);   } } |

### 2.5收集器

·将流中的元素积累成一个结果

·作用于一个终端操作collect()上

·collect/Collector/Collectors

预定义收集器功能

·将流元素规约汇总为一个值

·将流元素分组；

·将流元素分区；

### 2.6总结

·流操作

·中间操作（无状态）

·filter

·map

·flatMap

·peek

·中间操作（有状态）

·distinct

·skip

·limit

·sorted

·终端操作（短路）

·allMatch

·anyMatch

·noneMatch

·findFirst

·findAny

·终端操作（非短路）

·max

·min

·forEach

·reduce

·collect

·count

·流构建

·值

·数组

·文件

·函数（无限流）

·收集器

·将流元素规约和汇总为一个值

·分组

·分区

## 第3节 资源关闭

### 3.1资源关闭对比

垃圾回收特点：

·垃圾回收机制只负责回收堆内存资源，不会回收任何物理资源

·程序无法精确控制垃圾回收动作的具体发生时间

·在垃圾回收之前总先调用它的finalize方法

常见手动释放的物理资源

·文件/流资源

·socket套接字

·数据库连接资源

·nio缓冲

不释放问题

·资源被长时间无效占用

·超过最大限制后，将无资源可用

·导致系统无法使用

Try-with-resource 使用

·多资源自动关闭你

·实现AutoCloseable接口

·避免异常屏蔽

|  |
| --- |
| */\*\*  \* JDK1.7之后，实现正确关闭流资源方法  \* try - with - resource  \*/* **public class** NewFileCopyTest {  @Test  **public void** copyFile(){  *// 定义输入输出路径* String oriUrl = **"origin/NewFileCopyTest.java"**;  String targetUrl = **"origin/target/target1.txt"**;   *// 初始化输入输出流对象* **try**(  FileInputStream originalFIS = **new** FileInputStream(oriUrl);  FileOutputStream targetFOS = **new** FileOutputStream(targetUrl);  ){  **int** content;  **while**((content = originalFIS.read()) != -1){  targetFOS.write(content);  }  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  }   } } |

资源关闭顺序

·先开后关

·从外到内

·地城资源单独声明原则

资源关闭特殊情况

·资源对象被return的情况下，由调用方关闭

·ByteArrayInputStream等不需要检查关闭的资源对象

·使用Socket获取的InputStream和OutputStream对象不需要关闭，socket关闭时底层输入输出流也就关闭了

## 第4节 工具类Guava

### 4.1使用Optional避免null

Guava引入Optional<T>标明可能为null的T类型引用。Optional实例可能包含非null的引用（引用存在），也可能什么也不包括（引用确实）。

|  |
| --- |
| */\*\*  \* JAVA8中的Optional使用方法  \*/* **public class** OptionalTest {  @Test  **public void** test() **throws** Throwable {  */\*\*  \* 三种创建Optional对象方式  \*/  // 1.创建空的Optional对象* Optional.*empty*();   *// 2.使用非null值创建Optional对象* Optional.*of*(**""**);   *// 3.使用任意值创建Optional对象* Optional optional = Optional.*ofNullable*(**null**);   */\*\*  \* 判断是否引用确实的方法  \*/* optional.isPresent();   */\*\*  \* 当optional引用存在时执行  \* 类似的方法：map filter flatMap  \*/* optional.ifPresent(System.***out***::println);   */\*\*  \* 当optional引用缺失时执行  \*/* optional.orElse(**"引用缺失"**);  optional.orElseGet(()->{  **return "自定义引用缺失"**;  });  optional.orElseThrow(()->{  **throw new** RuntimeException(**"引用缺失异常"**);  });  }   **public static void** stream(List<String> list){  *//list.stream().forEach(System.out::println);* Optional.*ofNullable*(list)  .map(List::stream)  .orElseGet(Stream::*empty*)  .forEach(System.***out***::println);  }   **public static void** main(String[] args) {  *stream*(**null**);  } } |

### 4.2不可变集合

·创建对象的不可变拷贝是一项很好的防御性编程技巧；

·Guava为所有JDK标准集合类型和Guava新集合类型都提供了简单易用的不可变版本；

优点：

·当对象被不可信的库调用时，不可变形式是安全的

·不可变对象被多个线程调用，不存在竞态条件问题

·不可变集合不需要考虑变化，因此可以节省空间和实际；

·不可变对象因为有固定不变，可以作为常量来安全使用；

JDK提供的unmodifiableXXX方法

·笨重且累赘

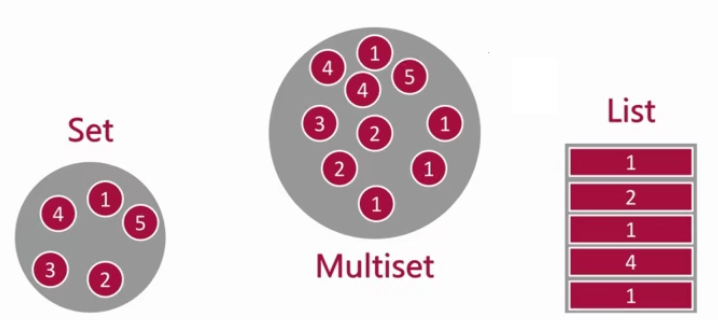
·不安全

·低效

|  |
| --- |
| */\*\*  \* 不可变集合用法  \*/* **public class** ImmutableTest {  **public static void** test(List<Integer> list){  list.remove(0);  }   **public static void** main(String[] args) {  List<Integer> list = **new** ArrayList<>();  list.add(1);  list.add(2);  list.add(3);  List<Integer> newNlist = Collections.*unmodifiableList*(list);  *test*(newNlist);  System.***out***.println(newNlist);  }   */\*\*  \* 不可变对象的三种使用方式  \*/* **public void** immutable(){  List<Integer> list = **new** ArrayList<>();  list.add(1);  list.add(2);  list.add(3);  *// 1 通过已经存在的集合创建* ImmutableSet.*copyOf*(list);   *// 2 通过初始值，直接创建不可变集合* ImmutableSet immutableSet = ImmutableSet.*of*(1,2,3);   *// 3 以builder方式创建* ImmutableSet.*builder*()  .add(1)  .addAll(Sets.*newHashSet*(2,3))  .add(4)  .build();  } } |

### 4.3新集合类型

·Guava引入了很多JDK没有的、但明显有用的新集合类型。这些新类型是为了和JDK集合框架共存，而没有往JDK集合抽象中硬塞其他概念。



没有元素顺序限制的ArrayList<E>

·add(E)：添加单个给定元素

·iterator()：返回一个迭代器，包含Multiset所有元素（包含重复）；

Map<E,Integer>，键为元素，值为计数

·count(Object)：返回元素的计数

·entrySet()：返回Set<Multiset.Entry<E>>,和Map的entrySet类似

·elementSet()：返回所有不重复元素的Set<E>，和Map的keySet类似；

Multiset与Map的区别

·元素计数只能是正数

·multiset.size()返回集合大小；

·multiset.iterator()会迭代重复元素

·multiset支持设置元素的计数

·没有的元素，multiset.count(E)为0



|  |
| --- |
| */\*\*  \* 不可变集合用法  \*/* **public class** ImmutableTest {  **public static void** test(List<Integer> list){  list.remove(0);  }   **public static void** main(String[] args) {  List<Integer> list = **new** ArrayList<>();  list.add(1);  list.add(2);  list.add(3);  List<Integer> newNlist = Collections.*unmodifiableList*(list);  *test*(newNlist);  System.***out***.println(newNlist);  }   */\*\*  \* 不可变对象的三种使用方式  \*/* **public void** immutable(){  List<Integer> list = **new** ArrayList<>();  list.add(1);  list.add(2);  list.add(3);  *// 1 通过已经存在的集合创建* ImmutableSet.*copyOf*(list);   *// 2 通过初始值，直接创建不可变集合* ImmutableSet immutableSet = ImmutableSet.*of*(1,2,3);   *// 3 以builder方式创建* ImmutableSet.*builder*()  .add(1)  .addAll(Sets.*newHashSet*(2,3))  .add(4)  .build();  } } |

## 第5节 线程池

### 5.1线程池概述

线程池带来的好处

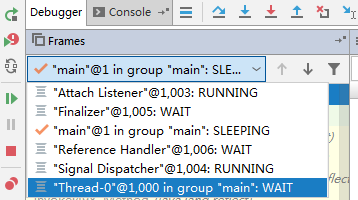
·降低资源消耗

·提高响应速度

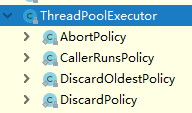
·提高线程的可管理性

### 5.2队列特性

|  |
| --- |
| **public class** QueueTest {  */\*\*  \* 基于数组的有界阻塞队列，队列容量10  \** ***@throws*** *InterruptedException  \*/* @Test  **public void** testArrayBlockingQueue() **throws** InterruptedException {  ArrayBlockingQueue queue = **new** ArrayBlockingQueue<Integer>(10);  **for** (**int** i = 0; i < 20; i++) {  queue.put(i);  System.***out***.println(**"put : "** + i);  }  }   */\*\*  \* 基于链表的有界 / 无界阻塞队列  \** ***@throws*** *InterruptedException  \*/* @Test  **public void** testLinkedBlockingQueue() **throws** InterruptedException {  LinkedBlockingQueue queue = **new** LinkedBlockingQueue<Integer>();  **for** (**int** i = 0; i < 20; i++) {  queue.put(i);  System.***out***.println(**"put : "** + i);  }  }   */\*\*  \* 同步移交阻塞队列，提供生成消费一对一的缓冲作用  \*/* @Test  **public void** test() **throws** InterruptedException {  SynchronousQueue queue = **new** SynchronousQueue<Integer>();  **new** Thread(()->{  **try** {  queue.put(1);  System.***out***.println(**"put : "**);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }).start();  */\*new Thread(()->{  try {  queue.take();  System.out.println("remove : ");  } catch (InterruptedException e) {  e.printStackTrace();  }  }).start();\*/* Thread.*sleep*(1000L \* 20);  } } |



### 5.3线程池可选择饱和策略

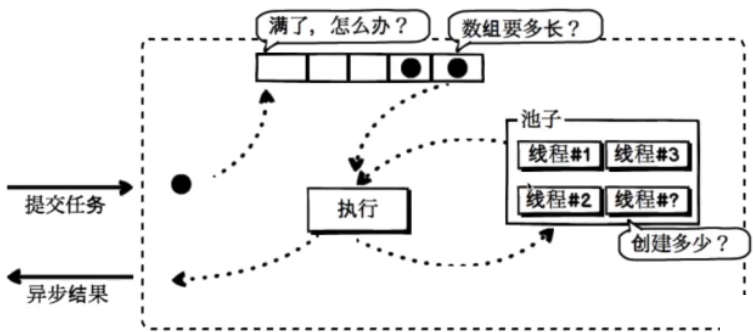


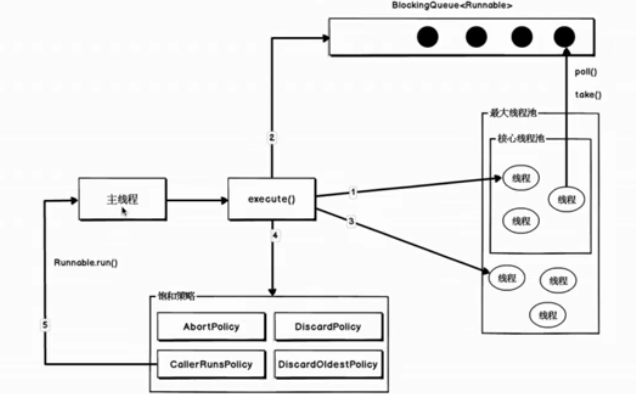
|  |
| --- |
| */\*\*  \* A handler for rejected tasks that throws a  \* {****@code*** *RejectedExecutionException}.  \*/* **public static class** AbortPolicy **implements** RejectedExecutionHandler {  */\*\*  \* Creates an {****@code*** *AbortPolicy}.  \*/* **public** AbortPolicy() { }   */\*\*  \* Always throws RejectedExecutionException.  \*  \** ***@param r*** *the runnable task requested to be executed  \** ***@param e*** *the executor attempting to execute this task  \** ***@throws*** *RejectedExecutionException always  \*/* **public void** rejectedExecution(Runnable r, ThreadPoolExecutor e) {  **throw new** RejectedExecutionException(**"Task "** + r.toString() +  **" rejected from "** +  e.toString());  } } |

|  |
| --- |
| */\*\*  \* A handler for rejected tasks that runs the rejected task  \* directly in the calling thread of the {****@code*** *execute} method,  \* unless the executor has been shut down, in which case the task  \* is discarded.  \*/* **public static class** CallerRunsPolicy **implements** RejectedExecutionHandler {  */\*\*  \* Creates a {****@code*** *CallerRunsPolicy}.  \*/* **public** CallerRunsPolicy() { }   */\*\*  \* Executes task r in the caller's thread, unless the executor  \* has been shut down, in which case the task is discarded.  \*  \** ***@param r*** *the runnable task requested to be executed  \** ***@param e*** *the executor attempting to execute this task  \*/* **public void** rejectedExecution(Runnable r, ThreadPoolExecutor e) {  **if** (!e.isShutdown()) {  r.run();  }  } } |

|  |
| --- |
| */\*\*  \* A handler for rejected tasks that discards the oldest unhandled  \* request and then retries {****@code*** *execute}, unless the executor  \* is shut down, in which case the task is discarded.  \*/* **public static class** DiscardOldestPolicy **implements** RejectedExecutionHandler {  */\*\*  \* Creates a {****@code*** *DiscardOldestPolicy} for the given executor.  \*/* **public** DiscardOldestPolicy() { }   */\*\*  \* Obtains and ignores the next task that the executor  \* would otherwise execute, if one is immediately available,  \* and then retries execution of task r, unless the executor  \* is shut down, in which case task r is instead discarded.  \*  \** ***@param r*** *the runnable task requested to be executed  \** ***@param e*** *the executor attempting to execute this task  \*/* **public void** rejectedExecution(Runnable r, ThreadPoolExecutor e) {  **if** (!e.isShutdown()) {  e.getQueue().poll();  e.execute(r);  }  } } |

|  |
| --- |
| */\*\*  \* A handler for rejected tasks that silently discards the  \* rejected task.  \*/* **public static class** DiscardPolicy **implements** RejectedExecutionHandler {  */\*\*  \* Creates a {****@code*** *DiscardPolicy}.  \*/* **public** DiscardPolicy() { }   */\*\*  \* Does nothing, which has the effect of discarding task r.  \*  \** ***@param r*** *the runnable task requested to be executed  \** ***@param e*** *the executor attempting to execute this task  \*/* **public void** rejectedExecution(Runnable r, ThreadPoolExecutor e) {  } } |





## 第6节 实用工具Lombok

ProjectLombok是一个java库，可以自动插入编译器并构建工具，为您的java增添色彩。

永远不要再写另一个getter或setter或equals方法，使用一个注解，您的类具有一个功能齐全的构建器，自动化为您的日志记录变量等。

### 6.1Lombok实现原理

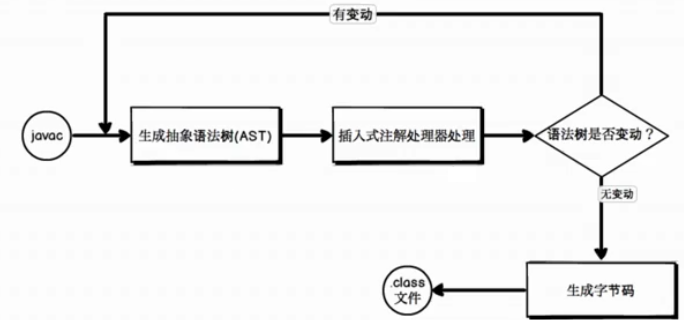
·运行时解析（AOP）

·编译时解析

编译时解析的两种机制

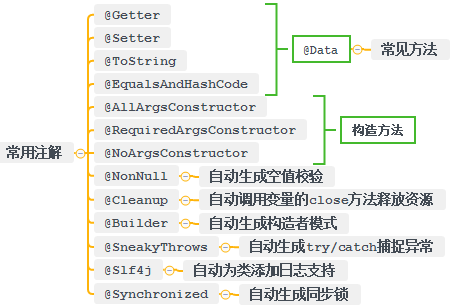
·Annotation Procession Tool（注解处理器，JDK1.8移除）

·Pluggable Annotation Procesion API（JSR269插入式注解处理）



### 6.2常用注解

7 常用注解



@Data

常见方法：

@Getter

@Setter

@ToString

@EqualsAndHashCode

构造方法

@AllArgsConstructor

@RequiredArgsConstructor

@NoArgsConstructor

@NonNull

自动生成空值校验

@Cleanup

自动调用变量的close方法释放资源

@Builder

自动生成构造者模式

@SneakyThrows

自动生成try/catch捕捉异常

@Slf4j

自动为类添加日志支持

@Synchronized

自动生成同步锁

### 6.3优缺点

优点：

·通过注解自动生成样板代码，提高开发效率

·代码简洁，只关注相关属性

·新增属性后，无需刻意修改相关方法

缺点：

·降低源代码可读性和完整性

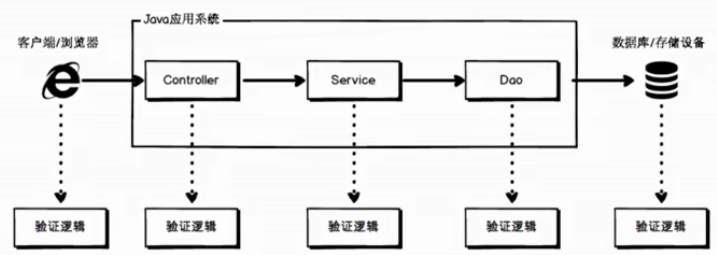
·加大对问题排查难度

·需要IDE的相关插件

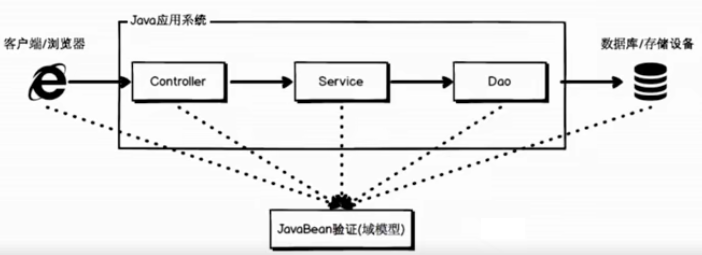
## 第7节验证框架

### 7.1分层验证与JavaBean验证

·分层验证



·JavaBean验证



### 7.2 JCP，JSR简介



JSR303/JSR349/JSR380

·Bean Validation 1.0 ： 2009年Java对象bean验证

·Bean Validation 1.1 ： 2013年在1.0基础上添加了方法级验证，错误消息支持EL表达式

·Bean Validation 2.0 ： 2017年支持JavaEE8容器类型级联验证，和新的约束注解

Bean Validation与Hibernate Validator



Hibernate Validator与Spring Validation

Spring Validation在Hibernate Validator的基础上，对其进行了二次封装，以满足在Spring环境中更简单、高效的对数据进行验证。

### 7.3初级约束注解

|  |
| --- |
| */\*\*  \* 待验证对象实体类  \* 用户信息类‘  \*  \*  \*/* @Data **public class** UserInfo {   *// 用户ID* @NotNull(message = **"用户ID不能为空"**)  **private** String **userId**;   *//用户名* @NotEmpty(message = **"用户名不能为空"**)  **private** String **userName**;   */\*\*  \* 用户密码  \* NotBlank 自动去掉字符串前后的空格后验证是否为空  \*/* @NotBlank(message = **"用户密码不能为空"**)  @Length(min = 6, max = 20, message = **"密码长度不能少于6位，不能多于20位"**)  **private** String **password**;   *// 邮箱* @Email(message = **"邮箱格式不正确"**)  **private** String **email**;   *// 手机号* **private** String **phone**;   *// 生日* @Past(message = **"生日不能为未来时间点"**)  **private** Date **birthday**;   *// 年龄* @Min(value = 18, message = **"年龄不能小于18岁"**)  @Max(value = 60, message = **"年龄不能大于60岁"**)  **private** Integer **age**;   *// 好友列表* @Size(min = 1, message = **"不能少于1个好友"**)  **private** List<UserInfo> **friends**;  } |

|  |
| --- |
| */\*\*  \* 验证测试类  \*/* **public class** ValidationTest {  *// 验证器对象* **private** Validator **validator**;  *// 待验证对象* **private** UserInfo **userInfo**;  *//验证结果集合* **private** Set<ConstraintViolation<UserInfo>> **set**;    */\*\*  \* 初始化操作  \*/* @Before  **public void** init(){  **validator** = Validation.*buildDefaultValidatorFactory*().getValidator();  **userInfo** = **new** UserInfo();  **userInfo**.setUserId(**"123456"**);  **userInfo**.setUserName(**" "**);  **userInfo**.setPassword(**"123456"**);  **userInfo**.setEmail(**"baiyifan@163.com"**);  **userInfo**.setAge(18);  Calendar calendar = Calendar.*getInstance*();  calendar.set(2019,1,1);  **userInfo**.setBirthday(calendar.getTime());  **userInfo**.setFriends(**new** ArrayList(){{add(**new** UserInfo());}});  }   @Test  **public void** test(){  **set** = **validator**.validate(**userInfo**);  }   @After  **public void** print(){  **set**.stream().forEach(item -> {  System.***out***.println(item.getMessage());  });  } } |

### 7.4中级约束注解

级联

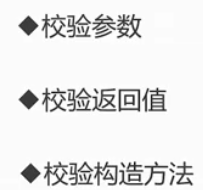
分组

序列

|  |
| --- |
| */\*\*  \* 级联验证测试  \*/* @Test **public void** graphValidation(){  **set** = **validator**.validate(**userInfo**); }  */\*\*  \* 分组验证  \*/* @Test **public void** groupValidation(){  **set** = **validator**.validate(**userInfo**,  UserInfo.RegisterGroup.**class**,  UserInfo.LoginGroup.**class**); }  */\*\*  \* 组序列验证  \*/* @Test **public void** groupSequenceValidation(){  **set** = **validator**.validate(**userInfo**, UserInfo.Group.**class**); } |

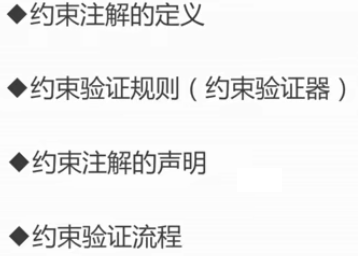
|  |
| --- |
| */\*\*  \* 待验证对象实体类  \* 用户信息类‘  \*  \*  \*/* @Data **public class** UserInfo {   *// 登录场景* **public interface** LoginGroup {}   *// 注册场景* **public interface** RegisterGroup {}   *// 排序场景* @GroupSequence({  LoginGroup.**class**,  RegisterGroup.**class**,  Default.**class** *//属于默认属性，即没有指明验证组的属性* })  **public interface** Group {}  *// 用户ID* @NotNull(message = **"用户ID不能为空"**,  groups = LoginGroup.**class**)  **private** String **userId**;   *//用户名* @NotEmpty(message = **"用户名不能为空"**)  **private** String **userName**;   */\*\*  \* 用户密码  \* NotBlank 自动去掉字符串前后的空格后验证是否为空  \*/* @NotBlank(message = **"用户密码不能为空"**)  @Length(min = 6, max = 20, message = **"密码长度不能少于6位，不能多于20位"**)  **private** String **password**;   *// 邮箱* @NotNull(groups = RegisterGroup.**class**,  message = **"邮箱不能为空"**)  @Email(message = **"邮箱格式不正确"**)  **private** String **email**;   *// 手机号* **private** String **phone**;   *// 生日* @Past(message = **"生日不能为未来时间点"**)  **private** Date **birthday**;   *// 年龄* @Min(value = 18, message = **"年龄不能小于18岁"**)  @Max(value = 60, message = **"年龄不能大于60岁"**)  **private** Integer **age**;   */\*\*  \* 好友列表  \** ***@Valid*** *级联验证  \*/* @Size(min = 1, message = **"不能少于1个好友"**)  **private** List<@Valid UserInfo> **friends**;  } |

### 7.5高级约束注解



|  |
| --- |
| */\*\*  \* 对方法入参进行校验  \** ***@throws*** *NoSuchMethodException  \*/* @Test **public void** paramValidation() **throws** NoSuchMethodException {  ExecutableValidator executableValidator =  **validator**.forExecutables();  *// 待验证对象* UserInfoService service = **new** UserInfoService();  *// 待验证方法* Method method = service.getClass()  .getMethod(**"setUserInfo"**, UserInfo.**class**);  *// 方法输入参数* Object[] paramObjects = **new** Object[]{**new** UserInfo()};   *// 对方法的输入参数进行验证* **set1** = executableValidator.validateParameters(  service,  method,  paramObjects  ); }  */\*\*  \* 对返回值进行约束校验  \*/* @Test **public void** returnValueValidation() **throws** NoSuchMethodException, InvocationTargetException, IllegalAccessException {  *// 获取校验执行器* ExecutableValidator executableValidator = **validator**.forExecutables();  *// 构造验证的方法对象* UserInfoService service = **new** UserInfoService();  Method method = service.getClass()  .getMethod(**"getUserInfo"**);  *// 调用方法得到返回值* Object returnValue = method.invoke(service);  *// 校验方法返回值是否符合约束* **set1** = executableValidator.validateReturnValue(  service,  method,  returnValue);  }  */\*\*  \* 对构造函数输入进行校验  \** ***@throws*** *NoSuchMethodException  \*/* @Test **public void** constructorValidation() **throws** NoSuchMethodException {  *// 获取验证执行器* ExecutableValidator executableValidator = **validator**.forExecutables();  *// 获取构造函数* Constructor constructor = UserInfoService.**class**.getConstructor(UserInfo.**class**);  UserInfo userInfo = **new** UserInfo();  userInfo.setEmail(**"baiyifan@163.com@"**);  Object[] paramObjects = **new** Object[]{userInfo};  *// 校验构造函数* **set1** = executableValidator.validateConstructorParameters(  constructor,  paramObjects  ); } |

### 7.6验证框架实践



·自定义手机号约束注解



|  |
| --- |
| */\*\*  \* 自定义手机号约束注解关联验证器  \*/* **public class** PhoneValidator **implements** ConstraintValidator<Phone, String> {  */\*\*  \* 自定义验证逻辑  \** ***@param s*** *\** ***@param constraintValidatorContext*** *\** ***@return*** *\*/* @Override  **public boolean** isValid(String s, ConstraintValidatorContext constraintValidatorContext) {  *//手机号验证规则：187开头，后边随便* String check = **"187\\d{8}"**;  Pattern regex = Pattern.*compile*(check);  String phone = Optional.*ofNullable*(s).orElse(**""**);  Matcher matcher = regex.matcher(phone);  **return** matcher.matches();  } } |

|  |
| --- |
| */\*\*  \* 自定义手机号约束注解  \*/* @Documented *// 注解的作用目标* @Target({ElementType.***FIELD***}) *// 注解的保留策略* @Retention(RetentionPolicy.***RUNTIME***) *// 不同之处：与约束注解关联的验证器* @Constraint(validatedBy = PhoneValidator.**class**) **public** @**interface** Phone {  *// 约束注解验证时的输出信息* String message() **default "手机号校验错误"**;   *// 约束注解在验证时所属的组别* Class<?>[] groups() **default** {};   *// 注解的有效负载* Class<? **extends** Payload>[] payload() **default** {}; } |

|  |
| --- |
| @Before **public void** init(){  **validator** = Validation.*buildDefaultValidatorFactory*().getValidator();  **userInfo** = **new** UserInfo();  *//userInfo.setUserId("123456");* **userInfo**.setUserName(**"BYF"**);  **userInfo**.setPassword(**"123456"**);  **userInfo**.setEmail(**"baiyifan@163.com"**);  **userInfo**.setAge(18);  Calendar calendar = Calendar.*getInstance*();  calendar.set(2019,1,1);  **userInfo**.setBirthday(calendar.getTime());  **userInfo**.setPhone(**"18705193697"**);   UserInfo friend = **new** UserInfo();  *//friend.setUserId("234566");* friend.setUserName(**"HELLO"**);  friend.setPassword(**"222333444"**);  friend.setPhone(**"18705193697"**);  *//friend.setEmail("hello@163.com");* **userInfo**.setFriends(**new** ArrayList(){{add(friend);}}); } |
| @Test **public void** test(){  **set** = **validator**.validate(**userInfo**); } |