链滴

Java8 Lambda 场景示例

作者：crick77

原文链接：https://ld246.com/article/1545321970124

来源网站：链滴

所有代码及完整测试用例在 https://github.com/wangyuheng/java8-lambda-sample

收集整理了一些实际业务场景下, 对Java8函数的使用.

预警: 代码较多, 比较无趣

阅读前提

1. 了解java8的一些新特性, 如: Optional、 Stream
2. 对函数式编程感兴趣

背景

一切的一切，源于不喜欢在代码中写太多的 if else.

scene & solution

1. Optional

Optional应该可以算作对null的一种优雅封装, 比如如下场景

我有一个枚举, 并且提供了一个**parse**方法, 可以根据key返回对应的枚举值.

1.1 enum parse

```
enum BooleanEnum {
    // 封装boolean
    TRUE(1, "是"),
    FALSE(0, "否");

    private Integer key;
    private String label;

    BooleanEnum(Integer key, String label) {
        this.key = key;
        this.label = label;
    }

    BooleanEnum parse(Integer key) {
        ...
        return ...
    }
}
```

但事实可能没那么美好, 如果这个parse方法传入了一个非法的key, 比如"-999", 那么应该如何处理?

1. 返回null
2. 返回default
3. 抛出IllegalException

可能性多了, 在调用不同方法时, 就需要考虑这个方法会选择哪种方案. 比如通过方法是否声明了Illegal xception异常进行判断、对每个结果进行 null!= result 判断. 这时, 方法的提供者就应该考虑, 返回一个 ptional对象, 方便调用方进行下一步的判断处理.

```
public static Optional<BooleanEnum> parse(Integer key) {
    return Arrays.stream(values()).filter(i -> i.getKey().equals(key)).findAny();
}
```

1.2 nested isPresent()

基于上述思考, 我们会将方法的返回值通过Optional进行封装, 可能就出现了如下代码

```
private Optional<User> login_check_by_if(String username, String password) {
    Optional<UserPO> optionalUserPO = findByUsername(username);
    if (optionalUserPO.isPresent()) {
        UserPO po = optionalUserPO.get();
        if (password.equals(po.getPassword())) {
            return Optional.of(convert(po));
        } else {
            return Optional.empty();
        }
    } else {
        return Optional.empty();
    }
}
```

这说明使用者知道了Optional可以方便进行非空判断, 但是远不止于此. Optional已经为我们开启了个流, 所以基于**惰性求值**的原则, 我们可以将上述代码简化.

```
private Optional<User> login_check_by_map(String username, String password) {
    return findByUsername(username)
            .filter(po -> Objects.equals(po.getPassword(), password))
            .map(this::convert);
}
```

1.3 .get().get().get()

我们在使用ServerWebExchange获取session值的时候, 为了保证健壮性, 需要针对每一级进行非空判 , 如

```
private String getTreasureIf(Exchange exchange) {
    if (null != exchange) {
        Session session = exchange.getSession();
        if (null != session) {
            Request request = session.getRequest();
            if (null != request) {
                return request.getTreasure();
            } else {
                return null;
            }
        } else {
            return null;
        }
```

```
    } else {
        return null;
    }
}
```

基于上述原则, 同样可以通过Optional进行简化

```
private Optional<String> getTreasureOptionalMap(Exchange exchange) {
    return Optional.ofNullable(exchange)
            .map(Exchange::getSession)
            .map(Session::getRequest)
            .map(Request::getTreasure);
}
```

去掉了 if else 之后, 世界是不是清爽了很多?

2. Distinct field

在**mysql**中, 针对field去重, 我们可以直接使用distinct. 那么在java中如何对一个list进行去重呢? 可能这样

```
public static <T> List<T> removeDuplication(List<T> list, String... keys) {
    if (list == null || list.isEmpty()) {
        System.err.println("List is empty.");
        return list;
    }

    if (keys == null || keys.length < 1) {
        System.err.println("Missing parameters.");
        return list;
    }

    for (String key : keys) {
        if (StringUtils.isBlank(key)) {
            System.err.println("Key is empty.");
            return list;
        }
    }

    List<T> newList = new ArrayList<T>();
    Set<String> keySet = new HashSet<String>();

    for (T t : list) {
        StringBuffer logicKey = new StringBuffer();
        for (String keyField : keys) {
            try {
                logicKey.append(BeanUtils.getProperty(t, keyField));
                logicKey.append(SEPARATOR);
            } catch (Exception e) {
                e.printStackTrace();
                return list;
            }
        }
        if (!keySet.contains(logicKey.toString())) {
```

```
            keySet.add(logicKey.toString());
            newList.add(t);
        } else {
            System.err.println(logicKey + " has duplicated.");
        }
    }

    return newList;
}
```

其实我们只是借助了Set值不可重复的特性, 为了工具类的通用性, 我们根据 key 和**反射**进行匹配.

听起来和看起来都很复杂, 我们尝试进行优化, java8已经允许我们使用**函数作为入参**, 所以我们可以把取field的方法传入

```
@Test
public void should_return_2_because_distinct_by_age() {
    userList = userList.stream()
            .filter(distinctByKey(User::getName))
            .collect(Collectors.toList());
    userList.forEach(System.out::println);
    assertEquals(2, userList.size());
}

private static <T, R> Predicate<T> distinctByKey(Function<T, R> keyExtractor) {
    Set<R> seen = ConcurrentHashMap.newKeySet();
    return t -> seen.add(keyExtractor.apply(t));
}
```

我们传入一个Function, 返回一个Predicate. 而Predicate正好是Stream#filter的入参.

3. Predicate union predicate

如果需要一个工具类,用来判断url是否符合设定的pattern.

```
private static final AntPathMatcher ANT_PATH_MATCHER = new AntPathMatcher();

private boolean includedPathSelector(String antPattern, String urls) {
    if(StringUtils.isEmpty(urls) || StringUtils.isEmpty(antPattern)) {
        return false;
    } else {
        for (String url : urls.split(", ")) {
            if (Objects.nonNull(url)) {
                if (ANT_PATH_MATCHER.match(antPattern, url.trim())) {
                    return true;
                }
            }
        }
    }
    return false;
}
```

如果antPattern也是一个数组集合, 那么就需要进行嵌套循环进行判断. 复杂到不想实现.

所以我们通过reduce&Predicate#or进行判断

```
private static final AntPathMatcher ANT_PATH_MATCHER = new AntPathMatcher();

private Predicate<String> includedPathSelector(String urls) {
    if(StringUtils.isEmpty(urls)) {
        return x -> false;
    }

    return Pattern.compile(", ").splitAsStream(urls)
            .filter(Objects::nonNull)
            .map(String::trim)
            .map(this::ant)
            .reduce(p -> false, Predicate::or);
}

private Predicate<String> ant(final String antPattern) {
    if (null == antPattern) {
        return input -> false;
    }
    return input -> ANT_PATH_MATCHER.match(antPattern, input);
}
```

这里希望说明的是返回Predicate对比直接返回boolean有一个好处是可以通过语义更明确的链接进组合，如 and 、 or

```
@Test
public void should_return_true_when_one_of_split_item_match_predicate_or() {
    Predicate<String> predicate1 = this.includedPathSelector("/permission");
    Predicate<String> predicate2 = this.includedPathSelector("/config");
    assertTrue(predicate1.or(predicate2).test("/config"));
    assertTrue(predicate1.or(predicate2).test("/permission"));
}

@Test
public void should_return_false_when_one_of_split_item_match_predicate_and() {
    Predicate<String> predicate1 = this.includedPathSelector("/permission");
    Predicate<String> predicate2 = this.includedPathSelector("/config");
    assertFalse(predicate1.and(predicate2).test("/config"));
    assertFalse(predicate1.and(predicate2).test("/permission"));
}
```

4. map & group

Stream提供了丰富的分组、聚合功能. 比如业务中需要把List<Item>转换为一个Map<String, Item>, key为**item**的id, value 为**item**本身, 或者**item**中的某个属性. 又或者基于某个属性字段进行分组, 得到个Map<String, List<Item>>

```
@Test
public void map_item_arr_by_uid() {

    Map<String, List<Item>> uidMap = new HashMap<>();
    for (Item item : list) {
        if (uidMap.containsKey(item.getUid())) {
            uidMap.get(item.getUid()).add(item);
        } else {
```

```
            List<Item> itemList = new ArrayList<>();
            itemList.add(item);
            uidMap.put(item.getUid(), itemList);
        }
    }
    uidMap.forEach((k, v) -> System.out.println(String.format("key = %s : value = %s", k, v)));
}
```

直接看如何利用Stream实现

```
@Data
@AllArgsConstructor
private static class Item {
    private Long id;
    private String uid;
}

private List<Item> list;

@Before
public void setUp() {
    long id = 0L;
    list = Arrays.asList(new Item(++id, "a"),
            new Item(++id, "a"),
            new Item(++id, "b"),
            new Item(++id, "b"),
            new Item(++id, "c"),
            new Item(++id, "d")
    );
    System.out.println("--------------------");
}

@Test
public void map_item_by_id() {
    Map<Long, Item> idMap = list.stream()
            .collect(Collectors.toMap(Item::getId, Function.identity()));
    idMap.forEach((k, v) -> System.out.println(String.format("key = %s : value = %s", k, v)));
}

@Test
public void map_item_arr_by_uid() {
    Map<String, List<Item>> uidMap = list.stream()
            .collect(Collectors.groupingBy(Item::getUid, Collectors.toList()));
    uidMap.forEach((k, v) -> System.out.println(String.format("key = %s : value = %s", k, v)));
}

@Test
public void map_uid_arr_by_uid() {
    Map<String, List<Long>> uidMapString = list.stream()
            .collect(Collectors.groupingBy(Item::getUid,
                    Collectors.mapping(Item::getId, Collectors.toList())));
    uidMapString.forEach((k, v) -> System.out.println(String.format("key = %s : value = %s", k, v
));
}
```

```
@Test
public void map_item_count_by_uid() {
    Map<String, Long> uidMapCount = list.stream()
            .collect(Collectors.groupingBy(Item::getUid,
                    Collectors.mapping(Function.identity(), Collectors.counting())));
    uidMapCount.forEach((k, v) -> System.out.println(String.format("key = %s : value = %s", k, v
));
}
```

5. concurrent & paged

另一个业务中常见的操作是因为原数据比较大, 所以我们将数据拆分为多页, 并设计多个线程并发的进相应的业务处理.

大致思路应该是

1. 设定每页period, 计算页数
2. 设定CountDownLatch
3. 创建线程池
4. 遍历list并通过subList拆分
5. 线程处理对应的subList

java8允许将函数作为入参, 所以第5步的操作, 我们可以把处理逻辑作为入参传递. 但是上面4步, 如何作呢?

```
private interface DivideHandler {
    default int getPeriod() {
        return 10;
    }

    default <T> void divideBatchHandler(List<T> dataList, Consumer<List<T>> consumer) {
        Optional.ofNullable(dataList).ifPresent(list ->
                IntStream.range(0, list.size())
                        .mapToObj(i -> new AbstractMap.SimpleImmutableEntry<>(i, list.get(i)))
                        .collect(Collectors.groupingBy(
                                e -> e.getKey() / getPeriod(),
                                Collectors.mapping(Map.Entry::getValue, Collectors.toList())))
                        .values()
                        .parallelStream()
                        .forEach(consumer)
        );
    }
}

class PrintDivideHandler implements DivideHandler {

    @Override
    public int getPeriod() {
        return 2;
    }
```

```
    private void batchPrint(List<String> dataList) {
        divideBatchHandler(dataList, System.out::println);
    }

}
```

parallelStream会调用**Fork/Join**框架进行并行处理. 如果需要在程序中显性的指定线程数, 可以通过 new ForkJoinPool(threadNum).submit(()->{}) 进行封装

6. chain return notNull & fail-fast

场景如下:

我们需要获取name值, 但是有多个方式可以获取, 也可能获取不到. 所以我们结合业务要求与获取效率行排序, 并且针对结果进行判断, 非空则返回.

听起来又是一连串的if.

针对这种case如何通过Stream优化呢? 依然是通过**惰性求值**以及**函数入参**.

我们先预设一个Stream逻辑, 其中顺序排列多个获取name值的函数, 然后统一执行.

```
private Optional<String> getOptionalName() {
    Stream<Supplier<String>> nameStream = Stream.of(
            this::getName1,
            this::getName2,
            this::getName3,
            this::getName4
    );
    return nameStream
            .map(Supplier::get)
            .filter(Objects::nonNull)
            .findAny();
}
```

如何复杂度再提升, 需要获取多个值构造一个bean, 其中任意一个值为空, 那么就终止stream, 不去进下一个值的获取操作.

此时可以利用flatMap进行连接

```
@Deprecated
private Optional<User> fillBuild() {
    Stream<String> nameStreamWrapper = buildStream(Arrays.asList(this::getName1, this::getName2, this::getName3, this::getName4));
    Stream<Integer> ageStreamWrapper = buildStream(Arrays.asList(this::getAge1, this::getAge2));

    BuildUser buildUser = (name, age) -> Stream.of(new User(name, age));

    return ageStreamWrapper.flatMap(age ->
            nameStreamWrapper.flatMap(name ->
                    buildUser.build(name, age)
            )
    ).findAny();
}
```

```
private <T> Stream<T> buildStream(List<Supplier<T>> suppliers) {
    return suppliers.stream()
            .map(Supplier::get)
            .filter(Objects::nonNull);
}

@FunctionalInterface
private interface BuildUser {
    Stream<User> build(String name, Integer age);
}
```

加上了@Deprecated标记是因为这么写看起来比较丑, 降低了语义和阅读性.

总结

1. 函数式编程带来更强的语义, 但是绝不仅是语法糖.
2. 尝试用函数式思维重构复杂逻辑, 会有意外收获

所有代码及完整测试用例在 https://github.com/wangyuheng/java8-lambda-sample