关于**Optional**的应用

解答课内容：

1：分享Optional的应用和使用
2：分析面向对象的静态成员和非静态成员
3：分析程序执行和Java基础的关系
4：实战开发Stream流的树形菜单
5：实战开发面试题的评论开发与实战
6：实现Redis和数据量的流量次数的累计

概述

- jdk1.7 - 资源的自动释放
- jdk1.8 - Stream
- jdk1.8 - Lambda
- jdk1.8 - 函数接口
- jdk1.8 - Optional
- jdk1.8 -日期处理

01、**Optional**的应用和使用

官方参考网址：https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

一：简介

以前一直不懂Optional有啥用，感觉太无语了，Java8还把它当做一个噱头来宣传，最近终于发现它的用处了，当然不用函数式编程的话，是没感觉的；
举例：一个非洲的Zoo，提供add一个animal进来的功能，可是有可能是用Future来捕捉动物的，也就是说可能没有catch到animal，但是为了防止以后使用的时候，有NPE错误，只有做判断；
个人觉得Optional实现的功能，有很多替代方案，if-else、三目等都可以；但Optional是用于函数式的一个整体中的一环，让函数式更流畅

NullPointerException相信每个JAVA程序员都不陌生，是JAVA应用程序中最常见的异常。之前，

Google Guava项目曾提出用Optional类来包装对象从而解决NullPointerException。受此影响，JDK8的类中也引入了Optional类，在新版的SpringData Jpa和Spring Redis Data中都已实现了对该方法的支持。

- 出现：Guava

二：认识使用

- 简化程序的逻辑
- 可以修复程序代码中的逻辑判断的问题

三、Optional类

```
1
2 package java.util;
3
4 import java.util.function.Consumer;
5 import java.util.function.Function;
```

```
6  import java.util.function.Predicate;
7  import java.util.function.Supplier;
8
9  public final class Optional<T> {
10     /**
11      * Common instance for {@code empty()}.
12      */
13     private static final Optional<?> EMPTY
   = new Optional<>();
14
15     /**
16      * If non-null, the value; if null,
   indicates no value is present
17      */
18     private final T value;
19
20     /**
21      * Constructs an empty instance.
22      *
23      * @implNote Generally only one empty
   instance, {@link Optional#EMPTY},
24      * should exist per VM.
25      */
26     private Optional() {
27         this.value = null;
28     }
29
30     /**
31      * Returns an empty {@code Optional}
   instance.  No value is present for this
32      * Optional.
33      *
34      * @apiNote Though it may be tempting
   to do so, avoid testing if an object
35      * is empty by comparing with {@code
   ==} against instances returned by
36      * {@code Option.empty()}. There is no
   guarantee that it is a singleton.
```

```
     * Instead, use {@link #isPresent()}.
     *
     * @param <T> Type of the non-existent
value
     * @return an empty {@code Optional}
     */
    public static<T> Optional<T> empty() {
        @SuppressWarnings("unchecked")
        Optional<T> t = (Optional<T>)
EMPTY;
        return t;
    }

    /**
     * Constructs an instance with the
value present.
     *
     * @param value the non-null value to
be present
     * @throws NullPointerException if
value is null
     */
    private Optional(T value) {
        this.value =
Objects.requireNonNull(value);
    }

    /**
     * Returns an {@code Optional} with the
specified present non-null value.
     *
     * @param <T> the class of the value
     * @param value the value to be
present, which must be non-null
     * @return an {@code Optional} with the
value present
     * @throws NullPointerException if
value is null
```

```
65      */
66     public static <T> Optional<T> of(T
value) {
67         return new Optional<>(value);
68     }
69 
70     /**
71      * Returns an {@code Optional}
describing the specified value, if non-
null,
72      * otherwise returns an empty {@code
Optional}.
73      *
74      * @param <T> the class of the value
75      * @param value the possibly-null value
to describe
76      * @return an {@code Optional} with a
present value if the specified value
77      * is non-null, otherwise an empty
{@code Optional}
78      */
79     public static <T> Optional<T>
ofNullable(T value) {
80         return value == null ? empty() :
of(value);
81     }
82 
83     /**
84      * If a value is present in this {@code
Optional}, returns the value,
85      * otherwise throws {@code
NoSuchElementException}.
86      *
87      * @return the non-null value held by
this {@code Optional}
88      * @throws NoSuchElementException if
there is no value present
89      *
```

```
90      * @see Optional#isPresent()
91      */
92     public T get() {
93         if (value == null) {
94             throw new
NoSuchElementException("No value present");
95         }
96         return value;
97     }
98 
99     /**
100      * Return {@code true} if there is a
value present, otherwise {@code false}.
101      *
102      * @return {@code true} if there is a
value present, otherwise {@code false}
103      */
104     public boolean isPresent() {
105         return value != null;
106     }
107 
108     /**
109      * If a value is present, invoke the
specified consumer with the value,
110      * otherwise do nothing.
111      *
112      * @param consumer block to be executed
if a value is present
113      * @throws NullPointerException if
value is present and {@code consumer} is
114      * null
115      */
116     public void ifPresent(Consumer<? super
T> consumer) {
117         if (value != null)
118             consumer.accept(value);
119     }
120 
```

```
121     /**
122      * If a value is present, and the value
    matches the given predicate,
123      * return an {@code Optional}
    describing the value, otherwise return an
124      * empty {@code Optional}.
125      *
126      * @param predicate a predicate to
    apply to the value, if present
127      * @return an {@code Optional}
    describing the value of this {@code
    Optional}
128      * if a value is present and the value
    matches the given predicate,
129      * otherwise an empty {@code Optional}
130      * @throws NullPointerException if the
    predicate is null
131      */
132     public Optional<T> filter(Predicate<?
    super T> predicate) {
133         Objects.requireNonNull(predicate);
134         if (!isPresent())
135             return this;
136         else
137             return predicate.test(value) ?
    this : empty();
138     }
139
140     /**
141      * If a value is present, apply the
    provided mapping function to it,
142      * and if the result is non-null,
    return an {@code Optional} describing the
143      * result.  Otherwise return an empty
    {@code Optional}.
144      *
145      * @apiNote This method supports post-
    processing on optional values, without
```

```
146      * the need to explicitly check for a
    return status.  For example, the
147      * following code traverses a stream of
    file names, selects one that has
148      * not yet been processed, and then
    opens that file, returning an
149      * {@code Optional<FileInputStream>}:
150      *
151      * <pre>{@code
152      *     Optional<FileInputStream> fis =
153      *         names.stream().filter(name -
    > !isProcessedYet(name))
154      *                       .findFirst()
155      *                       .map(name ->
    new FileInputStream(name));
156      * }</pre>
157      *
158      * Here, {@code findFirst} returns an
    {@code Optional<String>}, and then
159      * {@code map} returns an {@code
    Optional<FileInputStream>} for the desired
160      * file if one exists.
161      *
162      * @param <U> The type of the result of
    the mapping function
163      * @param mapper a mapping function to
    apply to the value, if present
164      * @return an {@code Optional}
    describing the result of applying a mapping
165      * function to the value of this {@code
    Optional}, if a value is present,
166      * otherwise an empty {@code Optional}
167      * @throws NullPointerException if the
    mapping function is null
168      */
169     public<U> Optional<U> map(Function<?
    super T, ? extends U> mapper) {
170         Objects.requireNonNull(mapper);
```

```
171         if (!isPresent())
172             return empty();
173         else {
174             return
    Optional.ofNullable(mapper.apply(value));
175         }
176     }
177 
178     /**
179      * If a value is present, apply the
    provided {@code Optional}-bearing
180      * mapping function to it, return that
    result, otherwise return an empty
181      * {@code Optional}.  This method is
    similar to {@link #map(Function)},
182      * but the provided mapper is one whose
    result is already an {@code Optional},
183      * and if invoked, {@code flatMap} does
    not wrap it with an additional
184      * {@code Optional}.
185      *
186      * @param <U> The type parameter to the
    {@code Optional} returned by
187      * @param mapper a mapping function to
    apply to the value, if present
188      *           the mapping function
189      * @return the result of applying an
    {@code Optional}-bearing mapping
190      * function to the value of this {@code
    Optional}, if a value is present,
191      * otherwise an empty {@code Optional}
192      * @throws NullPointerException if the
    mapping function is null or returns
193      * a null result
194      */
195     public<U> Optional<U>
    flatMap(Function<? super T, Optional<U>>
    mapper) {
```

```
196         Objects.requireNonNull(mapper);
197         if (!isPresent())
198             return empty();
199         else {
200             return
    Objects.requireNonNull(mapper.apply(value))
    ;
201         }
202     }
203 
204     /**
205      * Return the value if present,
    otherwise return {@code other}.
206      *
207      * @param other the value to be
    returned if there is no value present, may
208      * be null
209      * @return the value, if present,
    otherwise {@code other}
210      */
211     public T orElse(T other) {
212         return value != null ? value :
    other;
213     }
214 
215     /**
216      * Return the value if present,
    otherwise invoke {@code other} and return
217      * the result of that invocation.
218      *
219      * @param other a {@code Supplier}
    whose result is returned if no value
220      * is present
221      * @return the value if present
    otherwise the result of {@code other.get()}
222      * @throws NullPointerException if
    value is not present and {@code other} is
223      * null
```

```
224         */
225         public T orElseGet(Supplier<? extends
    T> other) {
226             return value != null ? value :
    other.get();
227         }
228 
229         /**
230          * Return the contained value, if
    present, otherwise throw an exception
231          * to be created by the provided
    supplier.
232          *
233          * @apiNote A method reference to the
    exception constructor with an empty
234          * argument list can be used as the
    supplier. For example,
235          * {@code IllegalStateException::new}
236          *
237          * @param <X> Type of the exception to
    be thrown
238          * @param exceptionSupplier The
    supplier which will return the exception to
239          * be thrown
240          * @return the present value
241          * @throws X if there is no value
    present
242          * @throws NullPointerException if no
    value is present and
243          * {@code exceptionSupplier} is null
244          */
245         public <X extends Throwable> T
    orElseThrow(Supplier<? extends X>
    exceptionSupplier) throws X {
246             if (value != null) {
247                 return value;
248             } else {
249                 throw exceptionSupplier.get();
```

```
250         }
251     }
252 
253     /**
254      * Indicates whether some other object
    is "equal to" this Optional. The
255      * other object is considered equal if:
256      * <ul>
257      * <li>it is also an {@code Optional}
    and;
258      * <li>both instances have no value
    present or;
259      * <li>the present values are "equal
    to" each other via {@code equals()}.
260      * </ul>
261      *
262      * @param obj an object to be tested
    for equality
263      * @return {code true} if the other
    object is "equal to" this object
264      * otherwise {@code false}
265      */
266     @Override
267     public boolean equals(Object obj) {
268         if (this == obj) {
269             return true;
270         }
271 
272         if (!(obj instanceof Optional)) {
273             return false;
274         }
275 
276         Optional<?> other = (Optional<?>)
    obj;
277         return Objects.equals(value,
    other.value);
278     }
279 
```

```
280     /**
281      * Returns the hash code value of the
    present value, if any, or 0 (zero) if
282      * no value is present.
283      *
284      * @return hash code value of the
    present value or 0 if no value is present
285      */
286     @Override
287     public int hashCode() {
288         return Objects.hashCode(value);
289     }
290 
291     /**
292      * Returns a non-empty string
    representation of this Optional suitable
    for
293      * debugging. The exact presentation
    format is unspecified and may vary
294      * between implementations and
    versions.
295      *
296      * @implSpec If a value is present the
    result must include its string
297      * representation in the result. Empty
    and present Optionals must be
298      * unambiguously differentiable.
299      *
300      * @return the string representation of
    this instance
301      */
302     @Override
303     public String toString() {
304         return value != null
305             ? String.format("Optional[%s]",
    value)
306             : "Optional.empty";
307     }
```

```
308 }
309
```

通过源码分析，Optional是一个final类，

- 说明不能被继承
- 它的构造函数是私有的，是单列的对象
- 提供一系列的方法，对其java程序逻辑的判断和优化处理

序号	方法	方法说明
1	`private Optional()`	无参构造，构造一个空Optional
2	`private Optional(T value)`	根据传入的非空value构建Optional
3	`public static<T> Optional<T> empty()`	返回一个空的Optional，该实例的value为空
4	`public static <T> Optional<T> of(T value)`	根据传入的非空value构建Optional，与Optional(T value)方法作用相同
5	`public static <T> Optional<T> ofNullable(T value)`	与of(T value)方法不同的是，ofNullable(T value)允许你传入一个空的value，当传入的是空值时其创建一个空Optional，当传入的value非空时，与of()作用相同
6	`public T get()`	返回Optional的值，如果容器为空，则抛出NoSuchElementException异常

序号	方法说明	
7	`public boolean isPresent()`	判断当家Optional是否已设置了值
8	`public void ifPresent(Consumer<? super T> consumer)`	判断当家Optional是否已设置了值，如果有值，则调用Consumer函数式接口进行处理
9	`public Optional<T> filter(Predicate<? super T> predicate)`	如果设置了值，且满足Predicate的判断条件，则返回该Optional，否则返回一个空的Optional
10	`public<U> Optional<U> map(Function<? super T, ? extends U> mapper)`	如果Optional设置了value，则调用Function对值进行处理，并返回包含处理后值的Optional，否则返回空Optional
11	`public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper)`	与map()方法类型，不同的是它的mapper结果已经是一个Optional，不需要再对结果进行包装
12	`public T orElse(T other)`	如果Optional值不为空，则返回该值，否则返回other
13	`public T orElseGet(Supplier<? extends T> other)`	如果Optional值不为空，则返回该值，否则根据other另外生成一个
14	`public <X extends Throwable> T orElseThrow(Supplier<? extends X>`	如果Optional值不为空，则返回该值，否则通过supplier抛出一个异常

序号	extends X> exceptionSupplier)throws X	方法说明

四、Optional的作用

- 简化程序的逻辑判断

05、isPresent

判断当家Optional是否已设置了值

- 也就如果被Optional使用的对象，如果不是null,就返回:true
- 也就如果被Optional使用的对象，如果是null,就返回:false

```
User user = null;
Optional<User> optional = Optional.ofNullable(user);
if(  !  optional.isPresent()){
    throw  new RuntimeException("用户找不到!!!");
}
```

- 上面的结果其实就出现异常，因为user对象是null。所以`optional.isPresent()`结果就是false。所以出现异常

orElseThrow() （ if + throws）

```
1   public static User getUser(Integer userId){
2         User user = null;
3         user = Optional.ofNullable(user)
4                 .orElseThrow(()->new
RuntimeException("用户找不到!!!"));//简化 简化
if + throws / if/else
5         return user;
6     }
```

- 判断一个对象是否为null,如果为null，就会抛出异常 它可以简化：if+throws场景 + springmvc统一异常处理

orElse() + if + new (复默认值)

```
1 public static User getUser(Integer userId)  {
2     User user = null;
3     user =
Optional.ofNullable(user).orElse(new User());
4     return user;
5 }
```

- 判断一个对象是否为null,如果为null，给你默认对象：if+throws场景 + springmvc统一异常处理

oElseGet() + if + 加逻辑处理

在开发中，有些时候我们数据查询是有值的，有些是没有值，但是我统计出，那些错误访问数据。

```
user =
Optional.ofNullable(user).orElseGet(()->{
    User user1 = new User();
    // 增加各种处理和逻辑
    return user1;
});
```

ofNullable妙用（复制默认值）

```
User user = new User();
user.setNickname(Optional.ofNullable(user.get
Nickname()).orElse("学生"));
user.setPassword(Optional.ofNullable(user.get
Password()).orElse("123456"));
```

伪代码

```
// 保存用户
@GetMapping("/user/{userid}")
public User getUser(@PathVariable("userid")
Integer userId){
    User user = userService.getById(userId);
    user =
Optional.ofNullable(user).orElseThrow(()-
>new RuntimeException("用户找不到!!") );
    return user;
}

// 保存用户 mybatis
@PostMapping("/user/saveupdate")
public void saveupdateUser(@RequestBody User
user){
```

```
12      User user1 =
    userService.getById(user.getId());
13      user1 =
    Optional.ofNullable(user1).orElse(new
    User());
14
     if(!Optional.ofNullable(user1.getId()).isPr
    esent()){
15
     user1.setNickname(Optional.ofNullable(user.
    getNickname()).orElse("学生"));
16
     user1.setPassword(Optional.ofNullable(user.
    getPassword()).orElse("123456"));
17          userservice.save(user1);
18      }else{
19          userservice.update(user);
20      }
21  }
```

of 和offNullAble区别

```
1  Optional<String> fullName =
   Optional.of(null);
2  Optional<String> fullName2 =
   Optional.ofNullable(null);
```

- of()和ofNullable() 它们的目标都是一致
    - 两者都是去创建Optional对象
    - 两者都是给value成员变量赋值

- 为什么 of 如果传递null的时候，会报空指针异常呢？是因为底层在创建Optional对象的时候，如果value是null，就出现空指针异常。同时告诉你一个道理，of只能去处理那些非空对象。

```
 private Optional(T value) {
    this.value =
Objects.requireNonNull(value);
 }
```

```
public static <T> T requireNonNull(T obj)
{
    if (obj == null)
        throw new NullPointerException();
// 如果value是null，就出现空指针异常
    return obj;
}
```

Optional中filter ,map,flatmap的认识

- filter ,map,flatmap它们是属于java.util.Stream的中间方法。但是这里的filter方法和前面的stream流中间无关。
- 个人的理解不应该出现，因为会给很多的初学者造成错了现象。
    - 它在Optional中的存在，其实还是去解决一个程序开发过程中集合为空问题。
        - 可以达到，集合中的元素如果不为空，我就直接filter/map/flatmap处理，如果为空给要么抛出异常。要么给默认初始化

```
package com.kuangstudy.demo09;

import com.kuangstudy.demo08.User;

```

```
5  import java.util.ArrayList;
6  import java.util.List;
7  import java.util.Optional;
8  import java.util.function.Predicate;
9  import java.util.stream.Collectors;
10 import java.util.stream.Stream;
11
12 public class OptionnalDemo3 {
13
14     public static void main(String[] args) {
15         List<User> users = null;
16         users = Optional.ofNullable(users)
// 1:  创建一个Optional，同时给Optional的value赋值null
17                 .filter((u) ->
u.get(0).getId().equals(1))// 2: 如果你调用的Optional中filter，
18                 // 如果内部出现异常的话，直接放一个空的Optional对象
19                 .orElse(new ArrayList<>());
// 3: 如果出现
20         System.out.println(users);
21         //List<User> collect =
users.stream().filter(u -> u.getId() >
0).collect(Collectors.toList());
22
23     }
24
25
26 }
27
```