

## 11 | 空值处理：分不清楚的null和恼人的空指针

2020-04-02 朱晔

Java 业务开发常见错误 100 例

[进入课程 >](#)



**讲述：王少泽**

时长 23:36 大小 21.62M



你好，我是朱晔。今天，我要和你分享的主题是，空值处理：分不清楚的 null 和恼人的空指针。

有一天我收到一条短信，内容是“尊敬的 null 你好，XXX”。当时我就笑了，这是程序员都能 Get 的笑点，程序没有获取到我的姓名，然后把空格式化为 null。很明显，这是没处理好 null。哪怕把 null 替换为贵宾、顾客，也不会引发这样的笑话。

程序中的变量是 null，就意味着它没有引用指向或者说没有指针。这时，我们对这个变量进行任何操作，都必然会引发空指针异常，在 Java 中就是 `NullPointerException`。那么，空指针异常容易在哪些情况下出现，又应该如何修复呢？



空指针异常虽然恼人但好在容易定位，更麻烦的是要弄清楚 null 的含义。比如，客户端给服务端的一个数据是 null，那么其意图到底是给一个空值，还是没提供值呢？再比如，数据库中字段的 NULL 值，是否有特殊的含义呢，针对数据库中的 NULL 值，写 SQL 需要注意什么呢？

今天，就让我们带着这些问题开始 null 的踩坑之旅吧。

## 修复和定位恼人的空指针问题

**NullPointerException 是 Java 代码中最常见的异常，我将其最可能出现的场景归为以下 5 种：**

参数值是 Integer 等包装类型，使用时因为自动拆箱出现了空指针异常；


字符串比较出现空指针异常；

诸如 ConcurrentHashMap 这样的容器不支持 Key 和 Value 为 null，强行 put null 的 Key 或 Value 会出现空指针异常；

A 对象包含了 B，在通过 A 对象的字段获得 B 之后，没有对字段判空就级联调用 B 的方法出现空指针异常；

方法或远程服务返回的 List 不是空而是 null，没有进行判空就直接调用 List 的方法出现空指针异常。

为模拟说明这 5 种场景，我写了一个 wrongMethod 方法，并一个 wrong 方法来调用它。wrong 方法的入参 test 是一个由 0 和 1 构成的、长度为 4 的字符串，第几位设置为 1 就代表第几个参数为 null，用来控制 wrongMethod 方法的 4 个入参，以模拟各种空指针情况：

 复制代码

```
1 private List<String> wrongMethod(FooService fooService, Integer i, String s, S
2     log.info("result {} {} {} {}", i + 1, s.equals("OK"), s.equals(t),
3         new ConcurrentHashMap<String, String>().put(null, null));
4     if (fooService.getBarService().bar().equals("OK"))
5         log.info("OK");
6     return null;
7 }
8
9 @GetMapping("wrong")
10 public int wrong(@RequestParam(value = "test", defaultValue = "1111") String t
```

```

11     return wrongMethod(test.charAt(0) == '1' ? null : new FooService(),
12        test.charAt(1) == '1' ? null : 1,
13        test.charAt(2) == '1' ? null : "OK",
14        test.charAt(3) == '1' ? null : "OK").size();
15 }
16
17 class FooService {
18     @Getter
19     private BarService barService;
20
21 }
22
23 class BarService {
24     String bar() {
25         return "OK";
26     }
27 }

```

很明显，这个案例出现空指针异常是因为变量是一个空指针，尝试获得变量的值或访问变量的成员会获得空指针异常。但，这个异常的定位比较麻烦。

在测试方法 `wrongMethod` 中，我们通过一行日志记录的操作，在一行代码中模拟了 4 处空指针异常：

对入参 `Integer i` 进行 `+1` 操作；

对入参 `String s` 进行比较操作，判断内容是否等于"OK"；

对入参 `String s` 和入参 `String t` 进行比较操作，判断两者是否相等；

对 `new` 出来的 `ConcurrentHashMap` 进行 `put` 操作，`Key` 和 `Value` 都设置为 `null`。

输出的异常信息如下：

 复制代码

```

1 java.lang.NullPointerException: null
2     at org.geekbang.time.commonmistakes.nullvalue.demo2.AvoidNullPointerException.
3     at org.geekbang.time.commonmistakes.nullvalue.demo2.AvoidNullPointerException.

```

这段信息确实提示了这行代码出现了空指针异常，但我们很难定位出到底是哪里出现了空指针，可能是把入参 `Integer` 拆箱为 `int` 的时候出现的，也可能是入参的两个字符串任意一个

为 null，也可能是因为把 null 加入了 ConcurrentHashMap。

你可能会想到，要排查这样的问题，只要设置一个断点看一下入参即可。但，在真实的业务场景中，空指针问题往往是在特定的入参和代码分支下才会出现，本地难以重现。如果要排查生产上出现的空指针问题，设置代码断点不现实，通常是要么把代码进行拆分，要么增加更多的日志，但都比较麻烦。

在这里，我推荐使用阿里开源的 Java 故障诊断神器 [Arthas](#)。Arthas 简单易用功能强大，可以定位出大多数的 Java 生产问题。

接下来，我就和你演示下如何在 30 秒内知道 wrongMethod 方法的入参，从而定位到空指针到底是哪个入参引起的。如下截图中有三个红框，我先和你分析第二和第三个红框：

第二个红框表示，Arthas 启动后被附加到了 JVM 进程；

第三个红框表示，通过 watch 命令监控 wrongMethod 方法的入参。



到这里，如果是简单的业务逻辑的话，你就可以定位到空指针异常了；如果是分支复杂的业务逻辑，你需要再借助 stack 命令来查看 wrongMethod 方法的调用栈，并配合 watch 命令查看各方法的入参，就可以很方便地定位到空指针的根源了。

下图演示了通过 stack 命令观察 wrongMethod 的调用路径：

```
[arthas@15231]$ stack org.geekbang.time.commonmistakes.nullvalue.demo2.AvoidNullPointerExceptionController.wrongMethod
Press Q or Ctrl+C to abort.
Affect(class-cnt:1, method-cnt:1) cost in 46 ms.
ts=2020-01-04 21:21:48;thread_name=http-nio-45678-exec-4;id=3e;is_daemon=true;priority=5;TCL=org.springframework.boot.web.embedded.tomcat.TomcatEmbeddedWebappClassLoader@733c464f
f
  @org.geekbang.time.commonmistakes.nullvalue.demo2.AvoidNullPointerExceptionController.wrongMethod()
    at org.geekbang.time.commonmistakes.nullvalue.demo2.AvoidNullPointerExceptionController.wrong(AvoidNullPointerExceptionController.java:20)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(NativeMethodAccessorImpl.java:-2)
```

如果你想了解 Arthas 各种命令的详细使用方法，可以 [点击这里查看](#)。

接下来，我们看看如何修复上面出现的 5 种空指针异常。

其实，对于任何空指针异常的处理，最直白的方式是先判空后操作。不过，这只能让异常不再出现，我们还是要找到程序逻辑中出现的空指针究竟是来源于入参还是 Bug：

如果是来源于入参，还要进一步分析入参是否合理等；

如果是来源于 Bug，那空指针不一定是纯粹的程序 Bug，可能还涉及业务属性和接口调用规范等。

在这里，因为是 Demo，所以我们只考虑纯粹的空指针判空这种修复方式。如果要先判空后处理，大多数人会想到使用 if-else 代码块。但，这种方式既增加代码量又会降低易读性，我们可以尝试利用 Java 8 的 Optional 类来消除这样的 if-else 逻辑，使用一行代码进行判空和处理。

修复思路如下：


对于 Integer 的判空，可以使用 Optional.ofNullable 来构造一个 Optional，然后使用 orElse(0) 把 null 替换为默认值再进行 +1 操作。

对于 String 和字面量的比较，可以把字面量放在前面，比如 "OK".equals(s)，这样即使 s 是 null 也不会出现空指针异常；而对于两个可能为 null 的字符串变量的 equals 比较，可以使用 Objects.equals，它会做判空处理。

对于 ConcurrentHashMap，既然其 Key 和 Value 都不支持 null，修复方式就是不要把 null 存进去。HashMap 的 Key 和 Value 可以存入 null，而 ConcurrentHashMap 看似是 HashMap 的线程安全版本，却不支持 null 值的 Key 和 Value，这是容易产生误区的一个地方。

对于类似 fooService.getBarService().bar().equals(“OK”) 的级联调用，需要判空的地方有很多，包括 fooService、getBarService() 方法的返回值，以及 bar 方法返回的字符串。如果使用 if-else 来判空的话可能需要好几行代码，但使用 Optional 的话一行代码就够了。

对于 rightMethod 返回的 List，由于不能确认其是否为 null，所以在调用 size 方法获得列表大小之前，同样可以使用 Optional.ofNullable 包装一下返回值，然后通过.orElse(Collections.emptyList()) 实现在 List 为 null 的时候获得一个空的 List，最后再调用 size 方法。

 复制代码

```
1 private List<String> rightMethod(FooService fooService, Integer i, String s, S
2     log.info("result {} {} {} {}", Optional.ofNullable(i).orElse(0) + 1, "OK".
3     Optional.ofNullable(fooService)
4         .map(FooService::getBarService)
5         .filter(barService -> "OK".equals(barService.bar()))
6         .ifPresent(result -> log.info("OK"));
7     return new ArrayList<>();
8 }
9
10 @GetMapping("right")
11 public int right(@RequestParam(value = "test", defaultValue = "1111") String t
12     return Optional.ofNullable(rightMethod(test.charAt(0) == '1' ? null : new
13         test.charAt(1) == '1' ? null : 1,
14         test.charAt(2) == '1' ? null : "OK",
15         test.charAt(3) == '1' ? null : "OK"))
16         .orElse(Collections.emptyList()).size();
17 }
```

经过修复后，调用 right 方法传入 1111，也就是给 rightMethod 的 4 个参数都设置为 null，日志中也看不到任何空指针异常了：

 复制代码

```
1 [21:43:40.619] [http-nio-45678-exec-2] [INFO ] [.AvoidNullPointerExceptionCont
```



但是，如果我们修改 right 方法入参为 0000，即传给 rightMethod 方法的 4 个参数都不可能是 null，最后日志中也无法出现 OK 字样。这又是为什么呢，BarService 的 bar 方法不是返回了 OK 字符串吗？

我们还是用 Arthas 来定位问题，使用 watch 命令来观察方法 rightMethod 的入参，-x 参数设置为 2 代表参数打印的深度为 2 层：

```
[arthas@15231]$ watch org.geekbang.time.commonmistakes.nullvalue.demo2.AvoidNullPointerExceptionController rightMethod params -x 2
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 38 ms.
ts=2020-01-04 21:43:06; [cost=3.482664ms] result=@Object[]
  @FooService[
    barService=null,
    this$0=@AvoidNullPointerException[org.geekbang.time.commonmistakes.nullvalue.demo2.AvoidNullPointerExceptionController@455da438],
  ],
  @Integer[1],
  @String[OK],
  @String[OK],
```

可以看到，FooService 中的 barService 字段为 null，这也就可以理解为什么最终出现这个 Bug 了。

这又引申出一个问题，**使用判空方式或 Optional 方式来避免出现空指针异常，不一定是解决问题的最好方式，空指针没出现可能隐藏了更深的 Bug**。因此，解决空指针异常，还是要真正 case by case 地定位分析案例，然后再去做判空处理，而处理时也并不只是判断非空然后进行正常业务流程这么简单，同样需要考虑为空的时候是应该出异常、设默认值还是记录日志等。

## POJO 中属性的 null 到底代表了什么？

在我看来，相比判空避免空指针异常，更容易出错的是 null 的定位问题。对程序来说，null 就是指针没有任何指向，而结合业务逻辑情况就复杂得多，我们需要考虑：

DTO 中字段的 null 到底意味着什么？是客户端没有传给我们这个信息吗？

既然空指针问题很讨厌，那么 DTO 中的字段要设置默认值么？

如果数据库实体中的字段有 null，那么通过数据访问框架保存数据是否会覆盖数据库中的既有数据？

如果不能明确地回答这些问题，那么写出的程序逻辑很可能会混乱不堪。接下来，我们看一个实际案例吧。



有一个 User 的 POJO，同时扮演 DTO 和数据库 Entity 角色，包含用户 ID、姓名、昵称、年龄、注册时间等属性：

 复制代码

```
1 @Data
2 @Entity
3 public class User {
4     @Id
5     @GeneratedValue(strategy = IDENTITY)
6     private Long id;
7     private String name;
8     private String nickname;
9     private Integer age;
10    private Date createDate = new Date();
11 }
```

有一个 Post 接口用于更新用户数据，更新逻辑非常简单，根据用户姓名自动设置一个昵称，昵称的规则是“用户类型 + 姓名”，然后直接把客户端在 RequestBody 中使用 JSON 传过来的 User 对象通过 JPA 更新到数据库中，最后返回保存到数据库的数据。


 复制代码

```
1 @Autowired
2 private UserRepository userRepository;
3
4 @PostMapping("wrong")
5 public User wrong(@RequestBody User user) {
6     user.setNickname(String.format("guest%s", user.getName()));
7     return userRepository.save(user);
8 }
9
10 @Repository
11 public interface UserRepository extends JpaRepository<User, Long> {
12 }
```

首先，在数据库中初始化一个用户，age=36、name=zhuye、create\_date=2020 年 1 月 4 日、nickname 是 NULL：

	id	age	create_date	name	nickname
	1	36	2020-01-04 09:58:11.000000	zhuye	(NULL)

然后，使用 cURL 测试一下用户信息更新接口 Post，传入一个 id=1、name=null 的 JSON 字符串，期望把 ID 为 1 的用户姓名设置为空：

 复制代码

```
1 curl -H "Content-Type:application/json" -X POST -d '{"id":1, "name":null}' ht
2
3 {"id":1,"name":null,"nickname":"guestnull","age":null,"createDate":"2020-01-05"}
```

接口返回的结果和数据库中记录一致：

	id	age	create_date	name	nickname
	1	(NULL)	2020-01-05 02:01:03.784000	(NULL)	guestnull

可以看到，这里存在如下三个问题：

调用方只希望重置用户名，但 age 也被设置为了 null；

nickname 是用户类型加姓名，name 重置为 null 的话，访客用户的昵称应该是 guest，而不是 guestnull，重现了文首提到的那个笑点；

用户的创建时间原来是 1 月 4 日，更新了用户信息后变为了 1 月 5 日。

归根结底，这是如下 5 个方面的问题：

明确 DTO 种 null 的含义。对于 JSON 到 DTO 的反序列化过程，null 的表达是有歧义的，客户端不传某个属性，或者传 null，这个属性在 DTO 中都是 null。但，对于用户信息更新操作，不传意味着客户端不需要更新这个属性，维持数据库原先的值；传了 null，意味着客户端希望重置这个属性。因为 Java 中的 null 就是没有这个数据，无法区分这两种表达，所以本例中的 age 属性也被设置为了 null，或许我们可以借助 Optional 来解决这个问题。

POJO 中的字段有默认值。如果客户端不传值，就会赋值为默认值，导致创建时间也被更新到了数据库中。

注意字符串格式化时可能会把 null 值格式化为 null 字符串。比如昵称的设置，我们只是进行了简单的字符串格式化，存入数据库变为了 guestnull。显然，这是不合理的，也

是开头我们说的笑话的来源，还需要进行判断。

**DTO 和 Entity 共用了一个 POJO。**对于用户昵称的设置是程序控制的，我们不应该把它们暴露在 DTO 中，否则很容易把客户端随意设置的值更新到数据库中。此外，创建时间最好让数据库设置为当前时间，不用程序控制，可以通过在字段上设置 columnDefinition 来实现。

**数据库字段允许保存 null，会进一步增加出错的可能性和复杂度。**因为如果数据真正落地的时候也支持 NULL 的话，可能就有 NULL、空字符串和字符串 null 三种状态。这一点我会在下一小节展开。如果所有属性都有默认值，问题会简单一点。

按照这个思路，我们对 DTO 和 Entity 进行拆分，修改后代码如下所示：

UserDto 中只保留 id、name 和 age 三个属性，且 name 和 age 使用 Optional 来包装，以区分客户端不传数据还是故意传 null。

在 UserEntity 的字段上使用 @Column 注解，把数据库字段 name、nickname、age 和 createDate 都设置为 NOT NULL，并设置 createDate 的默认值为 CURRENT\_TIMESTAMP，由数据库来生成创建时间。

使用 Hibernate 的 @DynamicUpdate 注解实现更新 SQL 的动态生成，实现只更新修改后的字段，不过需要先查询一次实体，让 Hibernate 可以“跟踪”实体属性的当前状态，以确保有效。

 复制代码

```
1  @Data
2  public class UserDto {
3      private Long id;
4      private Optional<String> name;
5      private Optional<Integer> age;
6  };
7
8  @Data
9  @Entity
10 @DynamicUpdate
11 public class UserEntity {
12     @Id
13     @GeneratedValue(strategy = IDENTITY)
14     private Long id;
15     @Column(nullable = false)
16     private String name;
17     @Column(nullable = false)
18     private String nickname;
```

```
19     @Column(nullable = false)
20     private Integer age;
21     @Column(nullable = false, columnDefinition = "TIMESTAMP DEFAULT CURRENT_TII
22     private Date createDate;
23 }
```

在重构了 DTO 和 Entity 后，我们重新定义一个 right 接口，以便对更新操作进行更精细化的处理。首先是参数校验：

对传入的 UserDto 和 ID 属性先判空，如果为空直接抛出 IllegalArgumentException。

根据 id 从数据库中查询出实体后进行判空，如果为空直接抛出 IllegalArgumentException。


然后，由于 DTO 中已经巧妙使用了 Optional 来区分客户端不传值和传 null 值，那么业务逻辑实现上就可以按照客户端的意图来分别实现逻辑。如果不传值，那么 Optional 本身为 null，直接跳过 Entity 字段的更新即可，这样动态生成的 SQL 就不会包含这个列；如果传了值，那么进一步判断传的是不是 null。

下面，我们根据业务需要分别对姓名、年龄和昵称进行更新：

对于姓名，我们认为客户端传 null 是希望把姓名重置为空，允许这样的操作，使用 Optional 的 orElse 方法一键把空转换为空字符串即可。

对于年龄，我们认为如果客户端希望更新年龄就必须传一个有效的年龄，年龄不存在重置操作，可以使用 Optional 的 orElseThrow 方法在值为空的时候抛出 IllegalArgumentException。

对于昵称，因为数据库中姓名不可能为 null，所以可以放心地把昵称设置为 guest 加上数据库取出来的姓名。

 复制代码

```
1 @PostMapping("right")
2 public UserEntity right(@RequestBody UserDto user) {
3     if (user == null || user.getId() == null)
4         throw new IllegalArgumentException("用户Id不能为空");
5
6     UserEntity userEntity = userRepository.findById(user.getId())
7         .orElseThrow(() -> new IllegalArgumentException("用户不存在"));
8 }
```

```


9     if (user.getName() != null) {
10         userEntity.setName(user.getName().orElse(""));
11     }
12     userEntity.setNickname("guest" + userEntity.getName());
13     if (user.getAge() != null) {
14         userEntity.setAge(user.getAge().orElseThrow(() -> new IllegalArgumentException("Age cannot be null")));
15     }
16     return userEntityRepository.save(userEntity);
17 }

```

假设数据库中已经有这么一条记录，id=1、age=36、create\_date=2020 年 1 月 4 日、name=zhuye、nickname=guestzhuye：

id	age	create_date	name	nickname
1	36	2020-01-04 11:09:20	zhuye	guestzhuye

使用相同的参数调用 right 接口，再来试试是否解决了所有问题。传入一个 id=1、name=null 的 JSON 字符串，期望把 id 为 1 的用户姓名设置为空：

 复制代码

```

1 curl -H "Content-Type:application/json" -X POST -d '{"id":1, "name":null}' ht
2
3 {"id":1,"name":"","nickname":"guest","age":36,"createDate":"2020-01-04T11:09:20

```

结果如下：

id	age	create_date	name	nickname
1	36	2020-01-04 11:09:20		guest

可以看到，right 接口完美实现了仅重置 name 属性的操作，昵称也不再有 null 字符串，年龄和创建时间字段也没被修改。

通过日志可以看到，Hibernate 生成的 SQL 语句只更新了 name 和 nickname 两个字段：

[复制代码](#)

```
1 Hibernate: update user_entity set name=?, nickname=? where id=?
```

接下来，为了测试使用 Optional 是否可以有效区分 JSON 中没传属性还是传了 null，我们在 JSON 中设置了一个 null 的 age，结果是正确得到了年龄不能为空的错误提示：

[复制代码](#)

```
1 curl -H "Content-Type:application/json" -X POST -d '{"id":1, "age":null}' http|
2
3 {"timestamp":"2020-01-05T03:14:40.324+0000","status":500,"error":"Internal Ser
```

## 小心 MySQL 中有关 NULL 的三个坑

前面提到，数据库表字段允许存 NULL 除了会让我们困惑外，还容易有坑。这里我会结合 NULL 字段，和你着重说明 sum 函数、count 函数，以及 NULL 值条件可能踩的坑。

为方便演示，首先定义一个只有 id 和 score 两个字段的实体：

[复制代码](#)

```
1 @Entity
2 @Data
3 public class User {
4     @Id
5     @GeneratedValue(strategy = IDENTITY)
6     private Long id;
7     private Long score;
8 }
```

程序启动的时候，往实体初始化一条数据，其 id 是自增列自动设置的 1，score 是 NULL：

[复制代码](#)

```
1 @Autowired
2 private UserRepository userRepository;
3
4 @PostConstruct
5 public void init() {
6     userRepository.save(new User());
7 }
```


```
7 }
```

然后，测试下面三个用例，来看看结合数据库中的 null 值可能会出现的坑：

通过 sum 函数统计一个只有 NULL 值的列的总和，比如 SUM(score);


select 记录数量，count 使用一个允许 NULL 的字段，比如 COUNT(score);

使用 =NULL 条件查询字段值为 NULL 的记录，比如 score=null 条件。

 复制代码

```
1 @Repository
2 public interface UserRepository extends JpaRepository<User, Long> {
3     @Query(nativeQuery=true,value = "SELECT SUM(score) FROM `user`")
4     Long wrong1();
5     @Query(nativeQuery = true, value = "SELECT COUNT(score) FROM `user`")
6     Long wrong2();
7     @Query(nativeQuery = true, value = "SELECT * FROM `user` WHERE score=null")
8     List<User> wrong3();
9 }
```

得到的结果，分别是 null、0 和空 List：

 复制代码

```
1 [11:38:50.137] [http-nio-45678-exec-1] [INFO ] [t.c.nullvalue.demo3.DbNullCont
```

显然，这三条 SQL 语句的执行结果和我们的期望不同：

虽然记录的 score 都是 NULL，但 sum 的结果应该是 0 才对；

虽然这条记录的 score 是 NULL，但记录总数应该是 1 才对；

使用 =NULL 并没有查询到 id=1 的记录，查询条件失效。

原因是：


**MySQL 中 sum 函数没统计到任何记录时，会返回 null 而不是 0**，可以使用 IFNULL 函数把 null 转换为 0；



**MySQL 中 count 字段不统计 null 值**，COUNT(\*) 才是统计所有记录数量的正确方式。

**MySQL 中 =NULL 并不是判断条件而是赋值**，对 NULL 进行判断只能使用 IS NULL 或者 IS NOT NULL。

修改一下 SQL：

 复制代码

```
1 @Query(nativeQuery = true, value = "SELECT IFNULL(SUM(score),0) FROM `user`")
2 Long right1();
3 @Query(nativeQuery = true, value = "SELECT COUNT(*) FROM `user`")
4 Long right2();
5 @Query(nativeQuery = true, value = "SELECT * FROM `user` WHERE score IS NULL")
6 List<User> right3();
```

可以得到三个正确结果，分别为 0、1、[User(id=1, score=null)]：

 复制代码

```
1 [14:50:35.768] [http-nio-45678-exec-1] [INFO ] [t.c.nullvalue.demo3.DbNullCont
```

## 重点回顾

今天，我和你讨论了做好空值处理需要注意的几个问题。

我首先总结了业务代码中 5 种最容易出现空指针异常的写法，以及相应的修复方式。针对判空，通过 Optional 配合 Stream 可以避免大多数冗长的 if-else 判空逻辑，实现一行代码优雅判空。另外，要定位和修复空指针异常，除了可以通过增加日志进行排查外，在生产上使用 Arthas 来查看方法的调用栈和入参会更快捷。

在我看来，业务系统最基本的标准是不能出现未处理的空指针异常，因为它往往代表了业务逻辑的中断，所以我建议每天查询一次生产日志来排查空指针异常，有条件的话建议订阅空指针异常报警，以便及时发现及时处理。

POJO 中字段的 null 定位，从服务端的角度往往很难分清楚，到底是客户端希望忽略这个字段还是有意传了 null，因此我们尝试用 Optional 类来区分 null 的定位。同时，为避免把

空值更新到数据库中，可以实现动态 SQL，只更新必要的字段。

最后，我分享了数据库字段使用 NULL 可能会带来的三个坑（包括 sum 函数、count 函数，以及 NULL 值条件），以及解决方式。

总结来讲，null 的正确处理以及避免空指针异常，绝不是判空这么简单，还要根据业务属性从前到后仔细考虑，客户端传入的 null 代表了什么，出现了 null 是否允许使用默认值替代，入库的时候应该传入 null 还是空值，并确保整个逻辑处理的一致性，才能尽量避免 Bug。

为处理好 null，作为客户端的开发者，需要和服务端对齐字段 null 的含义以及降级逻辑；而作为服务端的开发者，需要对入参进行前置判断，提前挡掉服务端不可接受的空值，同时在整个业务逻辑过程中进行完善的空值处理。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [🔗 这个链接](#) 查看。

## 思考与讨论

1. ConcurrentHashMap 的 Key 和 Value 都不能为 null，而 HashMap 却可以，你知道这么设计的原因是什么吗？TreeMap、Hashtable 等 Map 的 Key 和 Value 是否支持 null 呢？
2. 对于 Hibernate 框架可以使用 @DynamicUpdate 注解实现字段的动态更新，对于 MyBatis 框架如何实现类似的动态 SQL 功能，实现插入和修改 SQL 只包含 POJO 中的非空字段？

关于程序和数据库中的 null、空指针问题，你还遇到过什么坑吗？我是朱晔，欢迎在评论区与我留言分享，也欢迎你把这篇文章分享给你的朋友或同事，一起交流。

点击参与 

## 进入朱晔老师「读者群」带你攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 | 集合类：坑满地的List列表操作

下一篇 12 | 异常处理：别让自己在出问题的时候变为瞎子

### 精选留言 (10)

 写留言



Darren

2020-04-02

补充下：在MySQL的使用中，对于索引列，建议都设置为not null，因为如果有null的话，MySQL需要单独专门处理null值，会额外耗费性能。

回答下问题：

第一个问题：

从ConcurrentHashMap他自己的作者（Doug Lea）：

展开 

作者回复: 



 2

 5



Monday

2020-04-03

今天最大的惊喜就是arthas，以前听说过，但是从来没使用过，真心神器，感谢，感谢！！



2



美美

2020-04-02

有个规范我记得是说，不要在字段，方法参数，集合中使用Optional

作者回复: 其实主要是序列化和反序列化的麻烦（兼容问题），现在使用spring boot + jackson已经没问题了，自动引入了Jdk8Module，如果你使用其他序列化框架的话可能要考虑一下这种方式是否适合！其实很多时候最佳实践呢也是需要随着变迁不断调整的。



4



jiarupc

2020-04-02

这是我个人的一些工作经历。

以前尝试过Optional，但其他人反馈看不懂，最后还是换回了if-else。

得出结论，技术要考虑团队的接受程度。...

展开

作者回复: 也对



1



刘楠

2020-04-02

ConcurrentHashMap

```
final V putVal(K key, V value, boolean onlyIfAbsent) {  
    if (key == null || value == null) throw new NullPointerException();
```

因为直接对key进行了hashCode,要进行比较

...

展开



1



失火的夏天

2020-04-02

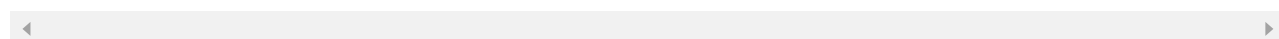
ConcurrentHashMap 的 Key 和 Value 都不能为 null，而 HashMap 却可以。

ConcurrentHashMap这个老爷子只说了value如果是空，会有二义性。就是在线程安全情况下，他到底是设置了一个null还是根本就没这玩意，key他老人家没说。。。老师可以说下理解吗？ ...

展开 ∨

作者回复: Key也是一样的道理，此外，我也更同意他的观点，就是普通的Map允许null是否是一个正确的做法也是值得商榷的，会增加犯错的可能

一个更好的类库和框架并不是把最佳实践写在文档中，而是直接编码在实现中



1



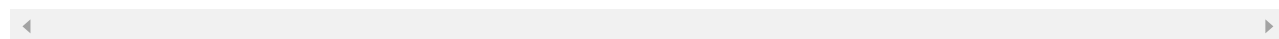
**终结者999号**

2020-04-02

在平常的开发中，对于DTO的值验证性校验也可以使用Hibernate Validator，也可以杜绝用户不按接口文档中所定义的格式输入，感觉也可以使用

展开 ∨

作者回复: 是



1



**Demon.Lee**

2020-04-07

谢谢老师。小伙伴们，我们这边UserDto都要求写成UserDTO，你们是哪种呢



**汝林外史**

2020-04-03

问题大家都答的很好，我就直接问问题吧。

老师，Hashtable的put会对value做null判断，key是在调用hashCode方法时报空指针，而ConcurrentHashMap是直接对key和value做null判断，是不是Hashtable的设计有问题？

展开 ∨

作者回复: Hashtable虽然这种写法不怎么好看，但结果都是一个NullPointerException，作者可能觉得 `int hash = key.hashCode();`

会抛空指针就没必要提前判断了吧



**pedro**

2020-04-02

第二个问题，mybatis 可以使用 if 标签来判断属性是否为 null 从而动态生成不含该属性的 sql。

