

Howard_14
码龄6年  暂无认证

80
原创

7万+
周排名

80万+
总排名

51万+
访问

等级

3984
积分

106
粉丝

295
获赞

79
评论

593
收藏



私信

关注

搜博主文章



- 热门文章
- 实时查看linux下tomcat运行日志

 75319
- 浅谈ArrayList动态扩容

 59688
- 数据库三范式的简单理解

 40649
- jq中获取某ul下所有li的文本值

 34780
- java中的fail-fast(快速失败)机制

 32850

分类专栏

	redis	2篇
	j2ee	28篇
	异常	6篇
	Linux	24篇
	数据库	16篇
	算法	12篇



- 最新评论
- 浅谈ArrayList动态扩容

wwwangwei194910: 你知道什么时候会触发了么, 加企鹅82137三三久二聊聊
- centos6.5安装mysql5.5–yum安装

peixiake1: 码住, 求博主联系方式, 我的微信cto51shequ, 在线等回复
- mybatis实现saveOrUpdate

买房动力十足: 我觉得这个方法最好不要用 主要是涉及权限这块的 可能是会有点问...
- java中的fail-fast(快速失败)机制

SpriteDxy: 使用索引遍历集合, 在遍历过程中进行修改也不会发生异常
- linux下卸载自带jdk以及安装jdk

雪落梅花: 非常感谢! 🍻

您愿意向朋友推荐 “博客详情页” 吗?

java中的fail-fast(快速失败)机制

原创

Howard_14

2017-10-30 16:38:06

 32942

 收藏 166

版权

分类专栏:

java

文章标签:

java

编程

基础

迭代

fail-fast

 java 专栏收录该内容

0 订阅

32 篇文章

订阅专栏

引入

在前面介绍 [ArrayList的扩容问题](#)时对于modCount的操作没有详细说明，该变量的操作在add，remove等操作中都会发生改变。那么该变量到底有什么作用呢？

简介

fail-fast 机制，即快速失败机制，是java集合(Collection)中的一种[错误检测机制](#)。当在迭代集合的过程中该集合在[结构上](#)发生改变的时候，就[有可能](#)会发生fail-fast，即抛出 ConcurrentModificationException异常。fail-fast机制并不保证在不同步的修改下一定会抛出异常，它只是尽最大努力去抛出，所以这种机制一般仅用于检测bug。

fail-fast的出现场景

在我们常见的java集合中就可能出现fail-fast机制,比如ArrayList，HashMap。在多线程和单线程环境下都有可能出现快速失败。

1、单线程环境下的fail-fast:

ArrayList发生fail-fast例子：

```
1      public static void main(String[] args) {
2          List<String> list = new ArrayList<>();
3          for (int i = 0 ; i < 10 ; i++ ) {
4              list.add(i + "");
5          }
6          Iterator<String> iterator = list.iterator();
7          int i = 0 ;
8          while(iterator.hasNext()) {
9              if (i == 3) {
10                 list.remove(3);
11             }
12             System.out.println(iterator.next());
13             i ++;
14         }
15     }
```

该段代码定义了一个Arraylist集合，并使用迭代器遍历，在遍历过程中，刻意在某一步迭代中remove一个元素，这个时候，就会发生fail-fast。

```
0
1
2
Exception in thread "main" java.util.ConcurrentModificationException
at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)
at java.util.ArrayList$Itr.next(ArrayList.java:851)
at com.howard.test.FailFastTest2.main(FailFastTest2.java:19)
```

HashMap发生fail-fast：

```
1      public static void main(String[] args) {
2          Map<String, String> map = new HashMap<>();
3          for (int i = 0 ; i < 10 ; i ++ ) {
4              map.put(i+"", i+"");
5          }
6          Iterator<Entry<String, String>> it = map.entrySet().iterator();
7          int i = 0;
8          while (it.hasNext()) {
```

强烈不推荐不推荐一般般推荐强烈推荐

最新文章

redis内存回收机制

redis的数据类型

浅谈HashMap

2019年 3篇

2018年 3篇

2017年 58篇

2016年 27篇

PARATERA 并行

已为100,000科技工作者提供计算服务

给力金秋 要任性

免费赠送

5000 核时CPU计算资源

500 元卡时GPU计算资源

5000核时CPU计算资源使用AM-D/Intel最新处理器，可折算为先导一号10000核时CPU计算资源。

/ 活动时间：9月1日-9月30日 /

立即参与

```
11         }12         Entry<String, String> entry = it.next();
13         System.out.println("key= " + entry.getKey() + " and value= " + entry.getValue
14         i++;
15     }
16 }
```

该段代码定义了一个hashmap对象并存放了10个键值对，在迭代遍历过程中，使用map的remove方法移除了一个元素，导致抛出了 ConcurrentModificationException异常：

```
key= 0 and value= 0
key= 1 and value= 1
key= 2 and value= 2
Exception in thread "main" java.util.ConcurrentModificationException
at java.util.HashMap$HashIterator.nextNode(HashMap.java:1429)
at java.util.HashMap$EntryIterator.next(HashMap.java:1463)
at java.util.HashMap$EntryIterator.next(HashMap.java:1461)
at com.howard.test.FailFastTest2.main(FailFastTest2.java:39)
```

2、多线程环境下：

```
1 public class FailFastTest {
2     public static List<String> list = new ArrayList<>();
3
4     private static class MyThread1 extends Thread {
5         @Override
6         public void run() {
7             Iterator<String> iterator = list.iterator();
8             while(iterator.hasNext()) {
9                 String s = iterator.next();
10                System.out.println(this.getName() + ":" + s);
11                try {
12                    Thread.sleep(1000);
13                } catch (InterruptedException e) {
14                    e.printStackTrace();
15                }
16            }
17            super.run();
18        }
19    }
20
21    private static class MyThread2 extends Thread {
22        int i = 0;
23        @Override
24        public void run() {
25            while (i < 10) {
26                System.out.println("thread2:" + i);
27                if (i == 2) {
28                    list.remove(i);
29                }
30                try {
31                    Thread.sleep(1000);
32                } catch (InterruptedException e) {
33                    e.printStackTrace();
34                }
35                i ++;
36            }
37        }
38    }
39
40    public static void main(String[] args) {
41        for(int i = 0 ; i < 10;i++){
42            list.add(i+"");
43        }
44        MyThread1 thread1 = new MyThread1();
45        MyThread2 thread2 = new MyThread2();
```

```
49         thread2.start();
50     }
51 }
```

启动两个线程，分别对其中一个对list进行迭代，另一个在线程1的迭代过程中去remove一个元素，结果也是抛出了java.util.ConcurrentModificationException

```
thread2:0
thread1:0
thread2:1
thread1:1
thread2:2
thread1:2
thread2:3
Exception in thread "thread1" java.util.ConcurrentModificationException
    at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)
    at java.util.ArrayList$Itr.next(ArrayList.java:851)
    at com.howard.test.FailFastTest$MyThread1.run(FailFastTest.java:15)
```

fail-fast的原理

fail-fast是如何抛出ConcurrentModificationException异常的，又是在什么情况下才会抛出？

我们知道，对于集合如list，map类，我们都可以通过迭代器来遍历，而Iterator其实只是一个接口，具体的实现还是要看具体的集合类中的内部类去实现Iterator并实现相关方法。这里我们就以ArrayList类为例。在

ArrayList中，当调用list.iterator()时，其源码是：

```
1     public Iterator<E> iterator() {
2         return new Itr();
3     }
```

即它会返回一个新的Itr类，而Itr类是ArrayList的内部类，实现了Iterator接口，下面是该类的源码：

```
1     /**
2      * An optimized version of AbstractList.Itr
3      */
4     private class Itr implements Iterator<E> {
5         int cursor;        // index of next element to return
6         int lastRet = -1; // index of last element returned; -1 if no such
7         int expectedModCount = modCount;
8
9         public boolean hasNext() {
10             return cursor != size;
11         }
12
13         @SuppressWarnings("unchecked")
14         public E next() {
15             checkForComodification();
16             int i = cursor;
17             if (i >= size)
18                 throw new NoSuchElementException();
19             Object[] elementData = ArrayList.this.elementData;
20             if (i >= elementData.length)
21                 throw new ConcurrentModificationException();
22             cursor = i + 1;
23             return (E) elementData[lastRet = i];
24         }
25
26         public void remove() {
27             if (lastRet < 0)
28                 throw new IllegalStateException();
29             checkForComodification();
30
31             try {
32                 ArrayList.this.remove(lastRet);
33             }
```



Howard_14

关注

61

11

166



专栏目录



举报

```
36         } catch (IndexOutOfBoundsException ex) {
37             throw new ConcurrentM
38         }
39     }
40
41     @Override
42     @SuppressWarnings("unchecked")
43     public void forEachRemaining(Consumer<? super E> consumer) {
44         Objects.requireNonNull(consumer);
45         final int size = ArrayList.this.size;
46         int i = cursor;
47         if (i >= size) {
48             return;
49         }
50         final Object[] elementData = ArrayList.this.elementData;
51         if (i >= elementData.length) {
52             throw new ConcurrentModificationException();
53         }
54         while (i != size && modCount == expectedModCount) {
55             consumer.accept((E) elementData[i++]);
56         }
57         // update once at end of iteration to reduce heap write traffic
58         cursor = i;
59         lastRet = i - 1;
60         checkForComodification();
61     }
62
63     final void checkForComodification() {
64         if (modCount != expectedModCount)
65             throw new ConcurrentModificationException();
66     }
67 }
```

其中，有三个属性：

```
1  int cursor;           // index of next element to return
2  int lastRet = -1; // index of last element returned; -1 if no such
3  int expectedModCount = modCount;
```

cursor是指集合遍历过程中的即将遍历的元素的索引，lastRet是cursor -1，默认为-1，即不存在上一个时，为-1，它主要用于记录刚刚遍历过的元素的索引。expectedModCount这个就是fail-fast判断的关键变量了，它初始值就为ArrayList中的modCount。（modCount是抽象类AbstractList中的变量，默认为0，而ArrayList继承了AbstractList，所以也有这个变量，modCount用于记录集合操作过程中作的修改次数，与size还是有区别的，并不一定等于size）

我们一步一步来看：

```
1     public boolean hasNext() {
2         return cursor != size;
3     }
```

迭代器迭代结束的标志就是hasNext()返回false，而该方法就是用cursor游标和size(集合中的元素数目)进行对比，当cursor等于size时，表示已经遍历完成。

接下来看看最关心的next()方法，看看为什么在迭代过程中，如果有线程对集合结构做出改变，就会发生fail-fast：

```
1     @SuppressWarnings("unchecked")
2     public E next() {
3         checkForComodification();
4         int i = cursor;
5         if (i >= size)
```



举报


```
8         if (i >= elementData.length) g |           throw new ConcurrentModificationE
10         cursor = i + 1;
11         return (E) elementData[lastRet = i];
12     }
```

从源码知道，每次调用next()方法，在实际访问元素前，都会调用checkForComodification方法，该方法源码如下：

```
1         final void checkForComodification() {
2             if (modCount != expectedModCount)
3                 throw new ConcurrentModificationException();
4         }
```

可以看出，该方法才是判断是否抛出ConcurrentModificationException异常的关键。在该段代码中，当modCount != expectedModCount时，就会抛出该异常。但是在一开始的时候，expectedModCount初始值默认等于modCount，为什么会出现modCount != expectedModCount，很明显expectedModCount在整个迭代过程除了一开始赋予初始值modCount外，并没有再发生改变，所以可能发生改变的就只有modCount，在前面关于ArrayList扩容机制的分析中，可以知道在ArrayList进行add，remove，clear等涉及到修改集合中的元素个数的操作时，modCount就会发生改变(modCount ++),所以当另一个线程(并发修改)或者同一个线程遍历过程中，调用相关方法使集合的个数发生改变，就会使modCount发生变化，这样在checkForComodification方法中就会抛出ConcurrentModificationException异常。类似的，hashMap中发生的原理也是一样的。

避免fail-fast

了解了fail-fast机制的产生原理，接下来就看看如何解决fail-fast

方法1

在单线程的遍历过程中，如果要进行remove操作，可以调用迭代器的remove方法而不是集合类的remove方法。看看ArrayList中迭代器的remove方法的源码：

```
1         public void remove() {
2             if (lastRet < 0)
3                 throw new IllegalStateException();
4             checkForComodification();
5
6             try {
7                 ArrayList.this.remove(lastRet);
8                 cursor = lastRet;
9                 lastRet = -1;
10                expectedModCount = modCount;
11            } catch (IndexOutOfBoundsException ex) {
12                throw new ConcurrentModificationException();
13            }
14        }
```

可以看到，该remove方法并不会修改modCount的值，并且不会对后面的遍历造成影响，因为该方法remove不能指定元素，只能remove当前遍历过的那个元素，所以调用该方法并不会发生fail-fast现象。该方法有局限性。例子：

```
1         public static void main(String[] args) {
2             List<String> list = new ArrayList<>();
3             for (int i = 0 ; i < 10 ; i++ ) {
4                 list.add(i + "");
5             }
6             Iterator<String> iterator = list.iterator();
7             int i = 0 ;
8             while(iterator.hasNext()) {
```



Howard_14

关注

61

11

166



专栏目录

方法



举报

```
11         }12         System.out.println(iterator.next());
13         i ++;
14     }
15 }
```

方法2

使用java并发包(java.util.concurrent)中的类来代替 ArrayList 和hashMap。

比如使用 CopyOnWriterArrayList代替 ArrayList， CopyOnWriterArrayList在是使用上跟 ArrayList几乎一样， CopyOnWriter是写时复制的容器(COW)，在读写时是线程安全的。该容器在对add和remove等操作时，并不是在原数组上进行修改，而是将原数组拷贝一份，在新数组上进行修改，待完成后，才将指向旧数组的引用指向新数组，所以对于 CopyOnWriterArrayList在迭代过程并不会发生fail-fast现象。但 CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。

对于HashMap，可以使用ConcurrentHashMap， ConcurrentHashMap采用了锁机制，是线程安全的。在迭代方面，ConcurrentHashMap使用了一种不同的迭代方式。在这种迭代方式中，当iterator被创建后集合再发生改变就不再是抛出ConcurrentModificationException，取而代之的是在改变时new新的数据从而不影响原有的数据， iterator完成后再将头指针替换为新的数据，这样iterator线程可以使用原来老的数据，而写线程也可以并发的完成改变。即迭代不会发生fail-fast，但不保证获取的是最新的数据。

参考链接：

<http://www.jb51.net/article/84468.htm>

http://www.cnblogs.com/ccgjava/p/6347425.html?utm_source=itdadao&utm_medium=referral

听说看了这份Java学习路线的同学，毕业都拿到了大厂offer m0_60719736的博客 199
前言自学/学习路线这样的一期我想写很久了，因为一直想写的全一点硬核一点所以拖到了现在，我相信这一期对不管是还在...

常见容错机制：failover、failfast、fallback、failsafe 青鱼入云的博客 2万+
1.failover：失效转移 Fail-Over的含义为“失效转移”，是一种备份操作模式，当主要组件异常时，其功能转移到备份组件。其...

 优质评论可以帮助作者获得更高权重

 评论

 白鹭立雪: 调用了集合类的remove()方法，modCount在remove()中是被重新赋值了的。expectedModCount在迭

代器的remove()方法里被重新赋值ModCount保证的相等成立 1 年前 回复 ...

 5

 会飞的瓜牛: 博主里面有一个错误哦 方法1那里：该remove方法并不会修改modCount的值，并且不会对后面的

遍历造成影响。它的modCount改变了，只是expectedModCount被重新赋值了 1 年前 回复 ...

 1

 反披风 回复： 会修改modCount的值把 因为它调用的不就是ArrayList的remove方法，增删都会修改mod的

值。 7 月前 回复 ...

 1

 weixin_41282995 回复 zsq201619918:

1 ArrayList.this.remove(lastRet);

在这个语句里面修改了modCount的值。 8 月前 回复 ...

 zsq201619918 回复： 我也没看到代码里面有修改modCount的值 1 年前 回复 ...

 1

 SpriteDxy: 使用索引遍历集合，在遍历过程中进行修改也不会发生异常 6 月前 回复 ...

 1

 jiuliezuo8655: 写得很好 以为只有多线程才需要考虑 1 年前 回复 ...

 1

 规则固态长方体物质空间移动工程师: 博主你好 我想转载一下，我会注明出处的 1 年前 回复 ...

 1

 hancoder: nb 1 年前 回复 ...

 1


 AlenHY: 萌新看懂了 1 年前 回复 ...

 1

 Selenium.: 写的好，收益匪浅 1 年前 回复 ...

 1

java fail 方法_Java集合中的fail-fast(快速失败)机制详解 8-9
简介我们知道Java中Collection接口下的很多集合都是线程不安全的,比如 java.util.ArrayList不是线程安全的,因此如果在使...

 Howard_14 关注

 61  11  166



专栏目录

7-11

l.ArrayList不是线程安全的,因此如果在使用...



举报