实战小技巧，可用于实际编码过程中的 `code snippets` 带你再日常得编码中写得更加顺畅

实战1：字符串占位替换

字符串占位替换，相信没有小伙伴是陌生的，这东西可以说是伴随着我们所有的项目工程，编码过程；别不相信，如

- String.format
- sql参数拼接的占位
- log日志输出

接下来我们看一下在我们的日常工作生涯中，经常涉及到的几种占位替换方式

1. String.format

这种可以说是最原始最基础的方式了，基本上在最开始学习java这门语言的时候就会涉及到，语法也比较简单

举例如下

```
String.format("hello %s", "一灰灰blog”);
```

使用`%`来表示占位，后面跟上不同的标识符，用于限定这个占位处的参数类型

这种使用姿势，由jdk原生提供支持，下表为不同的转换符对应的说明

转换符	说明	参数实例
%s	字符串替换	"一灰灰"
%c	字符类型	'a'
%b	布尔类型	true/false
%d	整数，十进制	10
%x	整数，十六进制	0x12
%o	整数，八进制	012
%f	浮点	0.12f
%e	指数	2e2
%g	通用浮点型	
%h	散列	
%%	百分比	
%n	换行	
%tx	日期与时间类型（x代表不同的日期与时间转换符	

虽然上面表中列出了很多，但实际使用时，`%s`，`%d`，`%f` 这三个就足以应付绝大部分的场景了；使用姿势和上面的实例参不多，第一个参数为字符串模板，后面的可变参数为待替换的值

下面是在实际使用过程中的注意事项

1.1 类型不匹配

上面的表中介绍了不同的转换符，要求的参数类型，如果没有对应上，会怎样

`%s`，传入非字符串类型

```
@Test
public void testFormat() {
    System.out.println(String.format("hello %s", 120));
    System.out.println(String.format("hello %s", true));
    System.out.println(String.format("hello %s", new int[]{1,2, 3}));
    System.out.println(String.format("hello %s", Arrays.asList(1, 2, 3)));
    System.out.println(String.format("hello %s", 0x12));
}
```

输出如下

```
hello 120
hello true
hello [I@3d82c5f3
hello [1, 2, 3]
hello 18
```

也就是说，`%s`的占位标记，传参如果不是String类型，那么实际替换的是`arg.toString()`(所以数组输出的是地址，而list输出了内容)

`%d`，传入非整数

与字符串的不一样的是，如果我们定义要求替换的参数类型为整数，那么传参不是整数，就会抛异常

```
System.out.println(String.format("hello %d", 1.0F));
System.out.println(String.format("hello %d", "10"));
```

上面这两个，一个传入的参数为浮点，一个传入的是字符串，在实际替换的时候，可不会调用`Integer.valufOf(String.valueOf(xxx))`来强转，而是采用更直接的方式，抛异常

关键的提示信息如下

```
java.util.IllegalFormatConversionException: d != java.lang.Float
java.util.IllegalFormatConversionException: d != java.lang.String
```

因此在实际使用这种方式进行替换时，推荐选择`%s`，毕竟兼容性更好

1.2 参数个数不匹配

我们会注意到，`String.format`接收的参数是不定长的，那么就可能存在字符串模板中预留的占位与实际传入的参数个数不匹配的场景，那么出现这种场景时，会怎样

参数缺少

```
System.out.println(String.format("hello %s %s", "yihui"));
```

上面的例子中，模板要求两个，实际只传入一个参数，会直接抛异常`MissingFormatArgumentException`

```
java.util.MissingFormatArgumentException: Format specifier '%s'
```

参数过多

```
System.out.println(String.format("hello %s", "yihuihui", "blog"));
```

执行正常，多余的参数不会被替换

因此，我们在使用`String.format`进行字符串替换时，请确保传参不要少于实际定义的参数个数；多了还好，少了就会抛异常

2. MessageFormat

上面介绍的String.format虽说简单好用，但我们用多之后，自然会遇到，一个参数，需要替换模板中多个占位的场景，针对这种场景，更友好的方式是`MessageFormat`，这个也是jdk原生提供的

我们来简单看一下它的使用姿势

```
String ans = MessageFormat.format("hello {0}, wechart site {0}{1}", "一灰灰", "blog");
```

使用`{数字}`来表示占位，其中数字对应的是传参的下标，因此当一个参数需要复用时，使用MessageFormat就可以比较简单的实现了，上面就是一个实例，替换之后的字符串为

```
hello 一灰灰, wechart site 一灰灰blog
```

接下来说一下它使用时的注意事项

2.1 {}成对出现

如果字符串中，只出现一个`{`，而没有配套的`}`，会抛异常

```
System.out.println(MessageFormat.format("hello }", 123));
System.out.println(MessageFormat.format("hello { world",  456));
```

注意上面两种case，上面一个是有`}`而缺少`{`，这样是没有问题的；而下面那个则会抛异常

```
java.lang.IllegalArgumentException: Unmatched braces in the pattern.
```

如果字符串中却是希望输出`{`，可以使用单引号来处理

```
System.out.println(MessageFormat.format("hello '{' world",  456));
```

2.2 单引号

上面提到需要转移时，可以用单引号进行处理，在字符串模板的定义中，如果有单引号，需要各位注意

只有一个单引号，会导致后面所有占位都不生效

```
System.out.println(MessageFormat.format("hello {0}, I'm {1}", "一灰灰", "blog"));
```

上面这个输出结果可能和我们实际希望的不一致

```
hello 一灰灰, Im {1}
```

要解决上面这个，就是使用两个单引号

```
System.out.println(MessageFormat.format("hello {0}, I''m {1}", "一灰灰", "blog"));
```

这样输出的就是我们预期的

```
hello 一灰灰, I'm blog
```

2.3 序号省略

上面的定义中，已经明确要求我们在`{}`中指定参数的序号，如果模板中没有指定会怎样?

```
System.out.println(messageFormat.format("hello {}, world", "yihuihui"));
```

直接抛异常

```
java.lang.IllegalArgumentException: can't parse argument number:
```

3. 小结

本文介绍的实战小技巧属于是jdk原生提供的两种实现字符串占位替换的方式，除了这两个之外，我们日常开发中还会遇到其他的占位替换方式

比如sql的`?`替换，mybatis中sql参数组装使用`${paramName}`，或者logback日志输出中的`{}`来表示占位，spring的@Value注解声明的配置注入方式`${name:defaultValue}`，这些也都属于占位替换的范畴，那么它们又是怎么实现的呢?

实战2：数组与list互转

这个考题比较常见，也比较简单，难道就这也有什么可以说到的门路不成?

接下来本文好好的说一说它的几种实现姿势，总有一款你喜欢的

1.数组转List

1.1. Array.asList

这个考题太简单了，直接使用`Array.asList`不就完事了么，比如

```
@Test
public void ary2list() {
    String[] ary = new String[]{ "1", "a"};
    List<String> list = Arrays.asList((ary);
    System.out.println(list);
}
```

数组转list，so easy!!!

真的就这么简单么？？？

且看下面这一段代码

```
public void ary2list() {
    String[] ary = new String[]{ "1", "a"};
    List<String> list = Arrays.asList((ary);
    System.out.println(list);

    list.add("c");
    System.out.println(list);
}
```

直接抛出了异常`java.lang.UnsupportedOperationException`

有兴趣的小伙伴可以看一下源码实现方式，通过`Arrays.asList`创建的List，虽说也命名是`ArrayList`，但是它的全路径为`java.util.Arrays.ArrayList`，不支持`add`，`remove`等操作（所以下次再有面试官问ArrayList的知识点时，就可以反问一句，老哥你指的是哪个ArrayList😝，逼格是不是立马拉起来）

知识点

- 通过`Arrays.asList`创建的列表，不允许新增，删除元素；但是可以更新列表中元素的值

1.2. new ArrayList

上面的数组转list方式虽然是最简单的，但不一定是合适的，特别是当我们可能对转换后的list进行操作时，可能埋坑（而且这种坑还非常隐晦，代码层面上很难发现）

为了减少在代码里面下毒的可能性，不妨使用下面这种方式`new ArrayList<>(Arrays.asList(ary))`

```
String[] ary = new String[]{ "1", "a"};
List<String> out = new ArrayList<>(Arrays.asList(ary));
out.add("hello");
System.out.println(out);
```

通过上面这种方式创建的List，就是我们熟知的`ArrayList`了

避雷预警

看到上面这个使用姿势，就很容易想到一个常见的踩雷点，比如我们的应用中，有一个全局共享的配置列表，张三需要拿id为奇数的配置，李四拿id为偶数的配置，然后他们都是这么做的

```
list.removeIf(s -> s.id % 2 == 0);
```

然后跑了一次之后发现这个全局的列表清空了，这就是典型的没有做好资源隔离的case了，针对这种场景，要么是限制使用方，直接针对全局的资源进行修改，要么就是使用方拿到的是一个隔离的备份

禁止修改：

- 使用不可变的容器，如前面提到的`java.util.Arrays.ArrayList` ()
- 使用`Collections.unmodifiableList`创建

```
List<String> unModifyList = Collections.unmodifiableList(out);
```

列表拷贝

```
new ArrayList<>(Arrays.asList(ary));
```

（上面这种属于深拷贝的实现，具体可以看一下jdk的源码实现）

1.3. Collections.addAll

第三种方式借助jdk提供的容器工具类`Collections`来实现

```
@Test
public void ary2listV3() {
    String[] ary = new String[]{ "1", "a"};
    // 创建列表，并指定长度，避免可能产生的扩容
    List<String> out = new ArrayList<>(ary.length);
    // 实现数组添加到列表中
    Collections.addAll(out, ary);

    // 因为列表为我们定义的ArrayList，因此可以对它进行增删改
    out.add("hello");
    System.out.println(out);
}
```

原则上是比较推荐这种方式来实现的，至于为啥？看下源码实现

```
public static <T> boolean addAll(Collection<? super T> c, T... elements) {
    boolean result = false;
    for (T element : elements)
        result |= c.add(element);
    return result;
}
```

这段代码的实现是不是非常眼熟，如果让我们自己来写，也差不多会写成这样吧，简单直观高效，完美

2. 列表转数组

不同于数组转列表的几种玩法，列表转数组就简单多了，直接调用`List.toArray`

```
List<String> list = Arrays.asList("a", "b", "c");
// 返回的是Object[] 数组
Object[] cell = list.toArray();

// 如果需要指定数组类型，可以传一个指定各类型的空的数组
// 也可以传一个与目标列表长度相等的数组，这样会将列表中的元素拷贝到这个数组中
String[] strCell = list.toArray(new String[]{});
```

3. 小结

今天的博文主题是数组与列表的互转，虽说题目简单，但是实现方式也是多种，需要搞清楚它们之间的本质区别，一不小心就可能采坑，而最简单的地方掉坑里，往往是最难发现和爬出来的

核心知识点小结如下

数组转list：

- `Arrays.asList(xxx)`：创建的是不可变列表，不能删除和新增元素
- `new ArrayList<>(Arrays.asList(xxx)`：相当于用列表创建列表，属于深拷贝的一种表现，获取到的列表支持新增、删除
- 推荐写法 `Collections.addAll()`

列表转数组

- `list.toArray`：如果需要指定数组类型，则传参指定

实战3：字符串与Collection的互转

将字符串转换为List，这种业务场景可以说非常非常常见了，实现方式也比较简单

```
public List<String> str2list(String str, String split) {
    String[] cells = str.split(split);
    return Arrays.asList(cells);
}
```

那么除了上面这种实现方式之外，还有其他的么？

1. 字符串转列表

上面的实现姿势相当于字符串先转数组，然后在通过数组转列表，所以可以沿用前一篇字数组转list的几种方式

1.1. jdk支持方式

借助`Collections.addAll`来实现

```
public List<String> str2list2(String str, String split) {
    List<String> list = new ArrayList<>();
    Collections.addAll(list, str.split(split));
    return list;
}
```

上面这种方式适用于输出String的列表，如果我希望转成int列表呢？可以采用下面的方式

```
public List<Integer> str2intList(String str, String split) {
    return Stream.of(str.split(split))
            .map(String::trim)
            .filter(s -> !s.isEmpty())
            .map(Integer::valueOf).collect(Collectors.toList());
}
```

直接将数组转换为流，然后基于jdk8的特性，来实现转换为int列表

1.2. guava方式

引入依赖

```
<!-- https://mvnrepository.com/artifact/com.google.guava/guava -->
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>30.1-jre</version>
</dependency>
```

除了使用jdk原生的方式之外，借助guava也是非常常见的的case了，主要通过Splitter来实现，写法看起来非常秀

```
public List<String> str2list2(String str, String split) {
    return Splitter.on(split).splitToList(str);
}
```

简单直接的一行代码搞定，如果我们希望是对输出的列表类型进行指定，也可以如下操作

```
public List<Integer> str2intListV2(String str, String split) {
    return Splitter.on(split).splitToStream(str)
            .map(String::trim).filter(s -> !s.isEmpty())
            .map(Integer::valueOf).collect(Collectors.toList());
}
```

1.3. apache-commons

引入依赖

```
 <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-collections4</artifactId>
    <version>4.4</version>
</dependency>
```

上面流的方式就很赞了，但是注意它是有jdk版本限制的，虽说现在基本上都是1.8以上的环境进行开发，但也不排除有上古的代码，比如我现在手上的项目，spring还是3...

如果我们不能使用流的方式，那么有什么简单的方式来实现字符串转换为指定类型的列表么？

```
public List<Integer> str2intListV3(String str, String split) {
    List<Integer> result = new ArrayList<>();
    CollectionUtils.collect(Arrays.asList(str.split(split)), new Transformer<String,
Integer>() {
        @Override
        public Integer transform(String s) {
            return Integer.valueOf(s);
        }
    }, result);
    return result;
}
```

上面这个实现也没有多优雅，不过这里有个编程小技巧可以学习，`new Transformer(){}`的传参方式，这种实现方式有点像回调的写法，虽然他们有本质的区别，此外就是jdk8之后的函数方法，就充分的体现这种设计思路，比如上面的换成jdk8的写法，直接简化为

```
public List<Integer> str2intListV3(String str, String split) {
    List<Integer> result = new ArrayList<>();
    CollectionUtils.collect(Arrays.asList(str.split(split)), Integer::valueOf,
result);
    return result;
}
```

2. 列表转字符串

2.1. StringBuilder

最容易想到的，直接使用StringBuilder来实现拼接

```
public String list2str(List<String> list, String split) {
    StringBuilder builder = new StringBuilder();
    for (String str: list) {
        builder.append(str).append(split);
    }
    return builder.substring(0, builder.length() - 1);
}
```

注意两点：

- 使用StringBuilder而不是StringBuffer (why?)
- 注意最后一个拼接符号不要

2.2. String.join

一个更简单的实现方式如下

```
public String list2str2(List<String> list, String split) {
    return String.join(split, list);
}
```

当然上面这个的缺点就是列表必须是字符串列表，如果换成int列表，则不行

2.3. gauva

guava也提供了列表转String的方式，同样很简单，而且还没有列表类型的限制

```
public <T> String list2str3(List<T> list, String split) {
    return Joiner.on(split).join(list);
}
```

3. 小结

本文的考题也非常常见，列表与字符串的互转，这里介绍了多种实现方式，有jdk原生的case（如果没有什么限制，推荐使用它，`String.split`除外，原因后面再说），如果有更高级的定制场景，如非String类型类表，则可以考虑guava的Splitter/Joinner来实现

在上面的实现中，也提供了几种有意思的编程方式

- Stream: 流，jdk8之后非常常见了
- 函数方法，回调写法case

实战4：字符串拼接

相信没有小伙伴没有写过这样的代码，比如说现在让我们来实现一个字符串拼接的场景，怎样的实现才算是优雅的呢?

以将int数组转为英文逗号分隔的字符串为例进行演示

1. 实现

1.1. 普通写法

直接使用StringBuilder来拼接

```
public String join(List<Integer> list) {
    StringBuilder builder = new StringBuilder();
    for(Integer sub: list) {
        builder.append(sub).append(",");
    }
    return builder.substring(0, builder.length() - 1);
}
```

上面这种写法相信比较常见，相对来说不太顺眼的地方就是最后的toString，需要将最后的一个英文逗号给干掉

当然也可以用下面这种事前判断方式，避免最终的字符串截取

```
public String join2(List<Integer> list) {
    StringBuilder builder = new StringBuilder();
    boolean first = true;
    for (Integer sub: list) {
        if (first) {
            first = false;
        } else {
            builder.append(",");
        }
        builder.append(sub);
    }
    return builder.toString();
}
```

1.2. StringJoiner

上面实现中，干掉最后的一个分隔符实在不是很优雅，那么有更好一点的用法么，接下来看一下使用`StringJoiner`的方式

```
public String join3(List<Integer> list) {
    StringJoiner joiner = new StringJoiner(",");
    for (Integer s : list) {
        joiner.add(String.valueOf(s));
    }
    return joiner.toString();
}
```

StringJoiner由jdk1.8提供，除了上面的基础玩法之外，结合jdk1.8带来的流操作方式，可以更简洁的实现

```
return list.stream().map(String::valueOf).collect(Collectors.joining(","));
```

怎么样，上面这个实现比起前面的代码是不是要简洁多了，一行代码完事

1.3. guava joiner

如果使用的jdk还不是1.8版本，不能使用上面的StringJoiner，没关系，还有guava的Joiner也可以实现

```
public String join5(List<Integer> list) {
    return Joiner.on(",").join(list);
}
```

注意

- 接收的参数类型为: 数组/Iterable/Iterator/可变参数, 基本上可以覆盖我们日常的业务场景

2. 小结

本篇文章的主题是一个非常非常常见的字符串拼接，一般来讲，我们在做字符串拼接时，最麻烦的事情就是分隔符的处理，要么就是分隔符前置添加，每次循环都需要判断是否为开头；要么就是后置，最后取字符串时，干掉最后一个分隔符

本文提供了一个非常使用的方式`StringJoiner`，完全解决了上面的分隔符问题，它的使用有两种场景

- 简单的容器转String：直接借助Stream的`Collectors.joining`来实现
- for循环 （这种场景一般是for循环内的逻辑不仅仅包括字符串拼接，还包括其他的业务逻辑）： 循环内直接执行`stringJoiner.add()`添加

对于jdk1.8及以上的版本，优先推荐使用上面说的StringJoiner来实现字符串拼接；至于jdk1.8之下，那么Guava就是一个不错的选择了，使用姿势也很很简单

实战5：驼峰与下划线划转

这个考题非常实用，特别是对于我们这些号称只需要CURD的后端开发来说，驼峰与下划线互转，这不是属于日常任务么；一般来讲db中的列名，要求是下划线格式（why? 阿里的数据库规范是这么定义的，就我感觉驼峰也没毛病），而java实体命名则是驼峰格式，所以它们之间的互转，就必然存在一个驼峰与下划线的互转

今天我们就来看一下，这两个的互转支持方式

1.实现

1.1. Gauva

一般来讲遇到这种普适性的问题，大部分都是有现成的工具类可以来直接使用的；在java生态中，说到好用的工具百宝箱，guava可以说是排列靠前的

接下来我们看一下如何使用Gauva来实现我们的目的

```
// 驼峰转下划线
String ans = CaseFormat.LOWER_CAMEL.to(CaseFormat.LOWER_UNDERSCORE, "helloWorld");
System.out.println(ans);

// 下划线转驼峰
String ans2 = CaseFormat.LOWER_UNDERSCORE.to(CaseFormat.LOWER_CAMEL, "hello_world");
System.out.println(ans2);
```

在这里主要使用的是`CaseFormat`来实现互转，guava的CaseFormat还提供了其他几种方式

上面这个虽然可以实现互转，但是如果我们有一个字符串为`helloWorld_Case`

将其他转换输出结果如下:

- 下划线：`hello_world__case`
- 驼峰：`helloworldCase`

这种输出，和标准的驼峰/下划线不太一样了（当然原因是由于输入也不标准）

1.2. Hutool

除了上面的guava，hutool的使用也非常广，其中包含很多工具类，其`StrUtil`也提供了下划线与驼峰的互转支持

```
String ans = StrUtil.toCamelCase("hello_world");
System.out.println(ans);
String ans2 = StrUtil.toUnderlineCase("helloWorld");
System.out.println(ans2);
```

同样的我们再来看一下特殊的case

```
System.out.println(StrUtil.toCamelCase("helloWorld_Case"));
System.out.println(StrUtil.toUnderlineCase("helloWorld_Case"));
```

输出结果如下

- 驼峰：`helloworldCase`
- 下划线: `hello_world_case`

相比较上面的guava的场景，下划线这个貌似还行

1.3. 自定义实现

接下来为了满足我们希望转换为标砖的驼峰/下划线输出方式的需求，我们自己来手撸一个

下划线转驼峰:

- 关键点就是找到下划线，然后去掉它，下一个字符转大写续上（如果下一个还是下划线，那继续找下一个）

根据上面这个思路来实现，如下

```
private static final char UNDER_LINE = '_';

/**
 * 下划线转驼峰
 *
 * @param name
 * @return
 */
public static String toCamelCase(String name) {
    if (null == name || name.length() == 0) {
        return null;
    }

    int length = name.length();
    StringBuilder sb = new StringBuilder(length);
    boolean underLineNextChar = false;

    for (int i = 0; i < length; ++i) {
        char c = name.charAt(i);
        if (c == UNDER_LINE) {
            underLineNextChar = true;
        } else if (underLineNextChar) {
            sb.append(Character.toUpperCase(c));
            underLineNextChar = false;
        } else {
            sb.append(c);
        }
    }

    return sb.toString();
}
```

驼峰转下划线

- 关键点：大写的，则前位补一个下划线，当前字符转小写（如果前面已经是一个下划线了，那前面不补，直接转小写即可）

```
public static String toUnderCase(String name) {
    if (name == null) {
        return null;
    }

    int len = name.length();
    StringBuilder res = new StringBuilder(len + 2);
    char pre = 0;
    for (int i = 0; i < len; i++) {
        char ch = name.charAt(i);
        if (Character.isUpperCase(ch)) {
            if (pre != UNDER_LINE) {
                res.append(UNDER_LINE);
            }
            res.append(Character.toLowerCase(ch));
        } else {
            res.append(ch);
        }
        pre = ch;
    }
    return res.toString();
}
```

再次测试`helloWorld_Case`，输出如下

- 驼峰：`helloWorldCase`
- 下划线: `hello_world_case`

实战6：枚举的特殊用法

难道我们日常使用的枚举还有什么特殊的玩法不成？没错，还真有，本文主要介绍枚举的两种不那么常见的使用姿势

- 利用枚举来实现单例模式
- 利用枚举来实现策略模式

1. 使用场景

1.1. 单例模式

单例模式可以说是每个java开发者必须掌握的一个设计模式了，通常我们说它的实现，有饱汉式和饿汉式，也有经常说的双重判断，今天我们介绍另外一种方式，借助枚举来实现

```
public enum SingleEnum {
    INSTANCE;

    public void print(String word) {
        System.out.println(word);
    }
}

@Test
public void testSingle() {
    SingleEnum.INSTANCE.print("hello world");
}
```

使用枚举来实现单例模式非常非常简单，将类声明为枚举，内部只定义一个值即可

为什么可以这样做?

- 枚举类不能`new`，因此保证单例
- 枚举类不能被继承
- 类不加载时，不会实例化

使用枚举类创建的单例有一个好处，就是即使用反射，也无法打破它的单例性质，这是相比较于其他的实现方式的一个优点

那么，为啥在实际的项目中，不太常见这种写法?

- 就我个人的一点认知（不保证准确）：这个与我们对枚举的认知有一定关系，在《Effect in java》一书中，推荐我们使用这种方式来实现单例，但是在实际的项目开发中，我们更多的将枚举作为常量来使用，很少在枚举类中，添加复杂的业务逻辑

1.2. 策略模式

枚举除了很容易就实现上面的单例模式之外，还可以非常简单的实现策略模式

举一个简单的例子，我现在有一个接口，通过接受的参数，来决定最终的数据存在什么地方

如果按照正常的写法，可能就是很多的if/else

```
public void save(String type, Object data) {
    if ("db".equals(type) {
        // 保存到db
        saveInDb(data);
    } else if ("file".equals(type))
        // 保存在文件
        saveInFile(data);
    } else if ("oss".eqauls(type)) {
        // 保存在oss
        saveInOss(type);
    }
}
```

上面这种写法虽说简单直观，但是当type类型一多了之后，这个if/else的代码行数就会很多很多了，而且看起来也不美观

接下来我们介绍一种利用枚举，基于策略模式的思想来解决上面的if/else问题

```
public enum SaveStrategyEnum {
    DB("db") {
        @Override
        public void save(Object obj) {
            System.out.println("save in db:" + obj);
        }
    },
    FILE("file") {
        @Override
        public void save(Object obj) {
            System.out.println("save in file: " + obj);
        }
    },
    OSS("oss") {
        @Override
        public void save(Object obj) {
            System.out.println("save in oss: " + obj);
        }
    };

    private String type;

    SaveStrategyEnum(String type) {
        this.type = type;
    }

    public abstract void save(Object obj);

    public static SaveStrategyEnum typeOf(String type) {
        for (SaveStrategyEnum strategyEnum: values()) {
            if (strategyEnum.type.equalsIgnoreCase(type)) {
                return strategyEnum;
            }
        }
        return null;
    }
}

public void save(String type, Object data) {
    SaveStrategyEnum strategyEnum = SaveStrategyEnum.typeOf(type);
    if (strategyEnum != null) {
        strategyEnum.save(data);
    }
}
```

上面的实现，主要利用的是`抽象类 + 枚举`来完成不同的策略具体实现

这种实现方式，相比较与前面的单例模式，还是更常见一点，虽然整体看下来没有什么难度，但是仔细看一看，会发现几个知识点

- 抽象方法的使用 （在模板设计模式中，更能体会抽象方法的使用妙处）
- 利用枚举原生提供的`values()`，来实现遍历，找到目标

2. 小结

枚举虽然说是jdk原生提供的一个基础数据类型，但是它的使用姿势除了我们熟知的常量之外，还可以有效的运用在设计模式中，让我们的代码实现更优雅

比如使用枚举来实现单例模式，就不用再面对让人烦躁的双重判断/内部类的方式了

使用枚举的策略模式，也可以有效解决我们类中大量的if/else

实战7：排序比较要慎重

今天介绍的又是一个非常非常基本的基本知识点，为啥要单独拎出来？还是因为这个东西虽然非常简单，但是很容易掉坑，我已经遇到几次不严谨的写法了

1.排序

1.1. Comparator 与 Comparable

输掉排序，这两个接口好像不太容易绕过去，我们简单介绍下它们的区别

- 如果你有一个类，希望支持同类型的自定义比较策略，可以实现接口`Compareable`
- 如果某个类，没有实现`Compareable`接口，但是又希望对它进行比较，则可以自自定义一个`Comparator`，来定义这个类的比较规则

通过一个简单的实例进行演示说明

```
public static class Demo implements Comparable<Demo> {
    int code;
    int age;

    public Demo(int code, int age) {
        this.code = code;
        this.age = age;
    }

    @Override
    public int compareTo(Demo o) {
        if (code == o.code) {
            return 0;
        } else if (code < o.code) {
            return -1;
        } else {
            return 1;
        }
    }

    @Override
    public String toString() {
        return "Demo{" +
                "code=" + code +
                ", age=" + age +
                '}';
    }
}
```

上面的实现中，重点关注 Demo类，实现了`Comparable`接口，因此可以直接调用`list.sort(null)`来进行比较；

但是如果我们现在需求改变了，希望实现针对demo类的age字段，进行升序排列，那么就可以利用`Comparator`来实现了

```
@Test
public void testDemoSort() {
    List<Demo> list = new ArrayList<>();
    list.add(new Demo(10, 30));
    list.add(new Demo(12, 10));
    list.add(new Demo(11, 20));
    // 默认根据 code 进行升序比较
    list.sort(null);
    System.out.println("sort by code: " + list);

    list.sort(new Comparator<Demo>() {
        @Override
        public int compare(Demo o1, Demo o2) {
            if (o1.age == o2.age) {
                return 0;
            } else if (o1.age < o2.age) {
                return -1;
            } else {
                return 1;
            }
        }
    });
    System.out.println("sort by age: " + list);
}
```

输出结果如下

```
sort by code: [Demo{code=10, age=30}, Demo{code=11, age=20}, Demo{code=12, age=10}]
sort by age: [Demo{code=12, age=10}, Demo{code=11, age=20}, Demo{code=10, age=30}]
```

1.2. 踩坑预告

再上面的compare方法实现中，我们可以发现里面的实现有点不太美观，我们最终的目的是什么?

- 如果左边的小于右边的，返回 -1
- 如果左边的大于右边的，返回 0
- 如果左边的等于右边的，返回 1

基于此，经常可以看到的实现如下

```
list.sort(new Comparator<Demo>() {
    @Override
    public int compare(Demo o1, Demo o2) {
       return o1.age - o2.age;
    }
});
```

上面这个实现虽然简洁了，但是有一个致命的问题，可能溢出!!!

所以请注意，千万千万不要用上面这种写法

那么有没有更优雅的方式呢?

- 有，使用基础类的`compare`方法

```
list.sort(new Comparator<Demo>() {
    @Override
    public int compare(Demo o1, Demo o2) {
       return Integer.compare(o1.age, o2.age);
    }
});
```

上面这一段代码，再jdk1.8中，可以简化为下面一句

```
list.sort(Comparator.comparingInt(o -> o.age));
```

再扩展一下，如果希望倒排呢?

- 第一种实现方式，调换位置
- Jdk1.8方式，使用负数

```
list.sort(new Comparator<Demo>() {
    @Override
    public int compare(Demo o1, Demo o2) {
       return Integer.compare(o2.age, o1.age);
    }
});

list.sort(Comparator.comparingInt(o -> -o.age));
```

2. 小结

今天主要介绍的知识点是排序，再我们日常使用中，如果一个类希望支持排序，最好的方式就是让它实现`Comparable`接口，然后自定义排序方式

这样再容器中，如果需要排序，直接调用 `list.sort(null)` 或者 `CollectionUtils.sort(list)`

如果目标类没有实现排序接口，或者希望使用另外一种排序方式，则通过自定义的`Comparator`来实现

最后关于`compare`方法的实现，设计到两个类的比较，这种最终的落脚地，多半是基础类型的比较

- o1 与 o2 比较，返回负数，则最终的结果中o1再前面（即升序排列）
- 不要直接使用 `o1-o2`会溢出，推荐使用 `Integer.compare(o1, o2);`

实战8：容器的初始化大小指定

容器可以说是我们日常开发中，除了基本对象之外，使用最多的类了，那么平时在使用的时候，是否有主意到良好编程习惯的大佬，在创建容器的时候，一般会设置size；那么他们为什么要这么干呢？是出于什么进行考量的呢？

今天我们将针对最常见的List/Map/Set三种容器类型的初始化值选择，进行说明

1. 容器初始化

1.1. List

列表，在我们日常使用过程中，会接触到下面几个

- ArrayList: 最常见的数组列表
- LinkedList: 基于链表的列表
- CopyOnWriteArrayList: 线程安全的数组列表

接下来逐一进行说明

1.1.1 ArrayList

现在以ArrayList为例，进行源码分析，当我们不指定列表大小，直接创建时

```
public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
```

上面是内部实现，其中`elementData`就是列表中存数据的数组，初始化为默认数组

当我们第一次添加一个元素时，发现数组为默认值，会触发一次数组扩容，新的数组大小为10（详情看源码）

其次就是数组的库容机制，通过源码/网上分享知识点可以知道，这个扩容的实现如下

- 当新添加的元素，数组放不下时，实现扩容
- `扩容后的大小` = `扩容前大小` + max(`添加元素个数`, 1/2 * `扩容前大小`)

基于上面的知识点，大致可以得出指定列表长度的好处

- 节省空间（用多少申请多少，避免浪费）
- 减少扩容带来的拷贝（扩容一次就会带来一次数组拷贝，如果已知列表很大，结果还使用默认的10，这会产生很多可避免的扩容开销）

1.1.2 LinkedList

基于链表的列表，不同于上面的数组列表，它没有提供指定大小的构造方法，why?

因为链表本身的数据结构的特点，它就像糖葫芦一样，一个串一个，有数据，才有接上的可能，因此不需要指定大小

1.1.3 CopyOnWriteArrayList

这个又非常有意思，它同样不能指定大小，但是原因与前面不同，主要在于它保证线程安全的实现方式

- 每次新增/修改(加锁，保证单线程访问)，都是在拷贝的数组操作；完成之后，用新的替换旧的

所以说，每次变更，都会存在数组拷贝，因此就没有必要提前指定数组大小

那么它的初始化每次都使用默认的么?

并不是这样的，当我们已知这个列表中的值时，推荐使用下面这种方式

```
List<String> values= Arrays.asList("12", "220", "123");
List<String> cList = new CopyOnWriteArrayList<>(values);
```

- 将初始化值，放在一个普通的列表中，然后利用普通列表来初始化`CopyOnWriteArrayList`

1.2.Map

常见的map容器使用，大多是下面几个

- `HashMap`
- `LinkedHashMap`: 有序的hashmap
- `TreeMap`: 有序的hashmap
- `ConcurrentHashMap`: 线程安全的map

1.2.1 HashMap

HashMap的底层数据结构是`数组 + 链表/红黑树`，关于这个就不细说了

我们在初始化时，若不指定size，则数组的默认长度为8（请注意，Map的数组长度是2的倍数）

与ArrayList的扩容时机不一样的是，默认情况下，Map容量没满就会触发一次扩容

默认是数量达到`size * 0.75`(0.75为扩容因子，可以在创建时修改)，就会触发一次扩容

why?

- 主要是为了减少hash冲突

同样的为了减少冲突，在初始化时，我们需要指定一个合适大小

比如我们

- 已知map的数量为2，这个时候Map的大小选择因该是4
- map数量为6，这个时候Map的大小选择是16

有时候让我们自己来计算这个值，就有些麻烦了，这个时候，可以直接使用Guava的工具类来完成这个目的

```
Map<String, String> map = Maps.newHashMapWithExpectedSize(6);
```

1.2.2 LinkedHashMap

初始化方式同上，略

1.2.3 ConcurrentHashMap

初始化方式同上，略

1.2.4 TreeMap

不同于上面几个的是treeMap，没有提供指定容器大小的构造方法

原因和前面说到的LinkedList有些类似，TreeMap的底层数据结构为Tree，所以新增数据是挂在树的一个节点下面，无需指定容量大小

1.3. Set

集合用的最多应该就是`HashSet`了，底层结构模型复用，所以初始化大小指定与HashMap一致，也不需要多说

2. 小结

今天这篇博文主要介绍的是三种常见的容器，在创建时，如何指定容量大小

首先明确一点，指定容量大小是为了

- 减少扩容带来的额外开销
- 指定容量代销，可以减少无效的内存开销

初始化值设置的关键点:

- ArrayList: 数据有多少个，初始化值就是多少
- HashMap: 考虑到扩容因子，初始化大小 = `(size / 0.75 + 1)`

实战9：List.subList使用不当StackOverflowError

相信每个小伙伴都使用过`List.subList`来获取子列表，日常使用可能没啥问题，但是，请注意，它的使用，很可能一不小心就可能导致oom

1.实例说明

1.1. subList

场景复现，如基于list实现一个小顶堆

```
public List<Integer> minStack(List<Integer> list, int value, int stackSzie) {
    list.add(value);
    if (list.size() < stackSzie) {
        return list;
    }
    list.sort(null);
    return list.subList(0, stackSzie);
}

@Test
public void testFix() {
    List<Integer> list = new ArrayList<>();
    for (int i = Integer.MAX_VALUE; i > Integer.MIN_VALUE; i--) {
        list.add(i);
        list = minStack(list, i, 5);
        System.out.println(list);
    }
}
```

上面这个执行完毕之后，居然出现栈溢出

```
// ....
[2147462802, 2147462803, 2147462804, 2147462805, 2147462806]
[2147462801, 2147462802, 2147462803, 2147462804, 2147462805]

java.lang.StackOverflowError
    at java.util.ArrayList$SubList.add(ArrayList.java:1057)
    at java.util.ArrayList$SubList.add(ArrayList.java:1057)
```

从实现来看，感觉也没啥问题啊， 我们稍微改一下上面的返回

```
public List<Integer> minStack(List<Integer> list, int value, int stackSzie) {
    list.add(value);
    if (list.size() < stackSzie) {
        return list;
    }
    list.sort(null);
    return new ArrayList<>(list.subList(0, stackSzie));
}
```

再次执行，却没有异常；所以关键点就在与

- list.subList的使用上

1.2. StackOverflowError分析

接下来我们主要看一下`list.subList`的实现

```
public List<E> subList(int fromIndex, int toIndex) {
    subListRangeCheck(fromIndex, toIndex, size);
    return new SubList(this, 0, fromIndex, toIndex);
}

private class SubList extends AbstractList<E> implements RandomAccess {
    private final AbstractList<E> parent;
    private final int parentOffset;
    private final int offset;
    int size;

    SubList(AbstractList<E> parent,
            int offset, int fromIndex, int toIndex) {
        this.parent = parent;
        this.parentOffset = fromIndex;
        this.offset = offset + fromIndex;
        this.size = toIndex - fromIndex;
        this.modCount = ArrayList.this.modCount;
    }
    ...
}
```

上面返回的子列表是ArrayList的一个内部类`SubList`，它拥有一个指向父列表的成员`parrent`

也就是说，从源头的ArryList开始，后面每次调用`subList`，这个指代关系就深一层

然后它的add方法也很有意思

```
public void add(int index, E e) {
    rangeCheckForAdd(index);
    checkForComodification();
    parent.add(parentOffset + index, e);
    this.modCount = parent.modCount;
    this.size++;
}
```

重点看 `parent.add(parentOffset + index, e);`，添加的数据实际上是加在最源头的ArrayList上的，也就是说，虽然你现在拿到的SubList，只有几个元素，但是它对应的数组，可能超乎你的想象

当然上面这个异常主要是以为调用栈溢出（一直往上找parent）

这里反应的另外一个重要问题则是内存泄漏，就不继续说了

如果需要解决上面这个问题，改造方法如下

```
public List<E> subList(int fromIndex, int toIndex) {
    subListRangeCheck(fromIndex, toIndex, size);
    return new ArrayList<>(new SubList(this, 0, fromIndex, toIndex));
}s
```

2. 小结

jdk提供的原生方法虽然非常好用，但是在使用的时候，也需要多家注意，一不小心就可能掉进坑里；这也告诉我们多看源码是有必要的

最后一句关键知识点小结：

- `ArrayList.subList` 返回的是内部类，与原ArrayList公用一个数组，只是限定了这个数组的起始下标和结束下标而已
- 在使用 `subList`，请注意是否会存在内存泄露和栈溢出的问题

实战10：不可变容器

不可变容器，看着好像在实际的业务中不怎么会用到，但实则不然，相信每个小伙伴都用过，或者看到过下面的代码

```
Collections.emptyList();
Collections.emptyMap();
```

今天我们来介绍一下如何使用不可变容器，以及使用时的注意事项

1. 不可变容器

1.1. JDK不可变容器

java原生提供了一些不可变容器，它们最大的特点就是不支持添加、删除、修改容器内的值

`Collections.emptyXxx`空容器

```
Collections.emptyMap();
Collections.emptyList();
Collections.emptySet();
```

上面三个是最常用的几个了，通常当我们一个方法的返回结果定义为容器类型时，可能为了避免npe，在返回空容器时，会如此使用

除了上面这几个空的不可变容器之外，还有

- `UnmodifiableList`
- `UnmodifiableMap`
- `UnmodifiableSet`

它们的使用姿势，通常是借助`Collections`来实现

```
List<Integer> list = Collections.unmodifiableList(Arrays.asList(1, 2, 3));
```

如上面创建的List，就不支持set/remove等修改操作

使用不可变容容器，最大的好处就是基于它的不可修改特性，来实现公用，且不会被污染

- 所以一个自然而然能想到的应用场景就是`全局共享的配置`

1.2. Guava不可变容器

上面是jdk提供的不可变容器，相比较与它们，在实际的项目中，使用Gauva的不可变容器的可能更多

- `ImmutableXxx`；不可变容器

```
List<Integer> list = ImmutableList.of(1, 2, 3);
Set<Integer> set = ImmutableSet.of(1, 2, 3);
Map<String, Integer> map = ImmutableMap.of("hello", 1, "world", 2);
```

上面是最常见的三个容器对应的不可变型

从使用角度来看，初始化非常方便（相比较与jdk版而言）

2. 注意事项

不可变容器虽好，但是使用不当也是很坑的；就我个人的一个观点

- 如果是应用内的接口方法，容器传参，返回容器时，尽量不要使用不可变容器；因为你没办法保证别人拿到你的返回容器之后，会对它进行什么操作
- 如果是对外提供返回结果，特别是null的场景，使用不可变的空容器优于返回null
- 不可变容器，用于全局公用资源，共享配置参数；多线程的数据传递时，属于比较合适的场景

实战11：Map转换Map的几种方式

在日常开发过程中，从一个Map转换为另外一个Map属于基本操作了，那么我们一般怎么去实现这种场景呢？有什么更简洁省事的方法么？

1.Map互转

1.1 实例场景

现在我们给一个简单的实例

希望将一个`Map<String, Integer>`转换成`Map<String, String>`，接下来看一下有哪些实现方式，以及各自的优缺点

首先提供一个创建Map的公共方法

```
private static <T> Map<String, T> newMap(String key, T val, Object... kv) {
    Map<String, T> ans = new HashMap<>(8);
    ans.put(key, val);
    for (int i = 0, size = kv.length; i < size; i += 2) {
        ans.put(String.valueOf(kv[i]), (T) kv[i + 1]);
    }
    return ans;
}
```

1.1.1 基本的for循环转换

这种方式是最容易想到和实现的，直接for循环来转换即可

```
@Test
public void forEachParse() {
    Map<String, Integer> map = newMap("k", 1, "a", 2, "b", 3);
    Map<String, String> ans = new HashMap<>(map.size());
    for (Map.Entry<String, Integer> entry: map.entrySet()) {
        ans.put(entry.getKey(), String.valueOf(entry.getValue()));
    }
    System.out.println(ans);
}
```

这种方式的优点很明显，实现容易，业务直观；

缺点就是可复用性较差，代码量多（相比于下面的case）

1.1.2 容器的流式使用

在jdk1.8提供了流式操作，同样也可以采用这种方式来实现转换

```
@Test
public void stream() {
    Map<String, Integer> map = newMap("k", 1, "a", 2, "b", 3);
    Map<String, String> ans = map.entrySet().stream().collect(
            Collectors.toMap(Map.Entry::getKey, s -> String.valueOf(s.getValue()), (a,
b) -> a));
    System.out.println(ans);
}
```

使用stream的方式，优点就是链式，代码量少；缺点是相较于上面的阅读体验会差一些（当然这个取决于个人，有些小伙伴就更习惯看这种链式的代码）

1.1.3 Guava的trasform方式

从代码层面来看，上面两个都不够直观，如果对guava熟悉的小伙伴对下面的代码可能就很熟悉了

```
@Test
public void transfer() {
    Map<String, Integer> map = newMap("k", 1, "a", 2, "b", 3);
    Map<String, String> ans =  Maps.transformValues(map, String::valueOf);
    System.out.println(ans);
}
```

核心逻辑就一行 `Maps.transformValues(map, String::valueOf)`，实现了我们的Map转换的诉求

很明显，这种方式的优点就是间接、直观；当然缺点就是需要引入guava，并且熟悉guava

1.2 最后一问，这篇文章目的是啥？

既然我们的标题是实战小技巧，本文除了给大家介绍可以使用guava的`Maps.transformValues`来实现map转换之外，更主要的一个目的是如果让我们自己来实现一个工具类，来支持这个场景，应该怎么做？

直接提供一个转换方法？

第一步：一个泛型的转换接口

```
public <K, T, V> Map<K, V> transform(Map<K, T> map) {
}
```

定义上面这个接口之后，自然而然想到的缺点就是差一个value的转换实现

第二步：value转换的定义

这里采用Function接口思想来定义转换类

```
public <K, T, V> Map<K, V> transform(Map<K, T> map, Function<T, V> func) {
}
```

当然到这里我们就需要注意jdk1.8以下是不支持函数编程的，那么我们可以怎么来实现呢?

这个时候再对照一下guava的实现，然后再手撸一个，知识点就到手了

实战12：巧用函数方法实现二维数组遍历

对于数组遍历，基本上每个开发者都写过，遍历本身没什么好说的，但是当我们在遍历的过程中，有一些复杂的业务逻辑时，将会发现代码的层级会逐渐加深

如一个简单的case，将一个二维数组中的偶数找出来，保存到一个列表中

二维数组遍历，每个元素判断下是否为偶数，很容易就可以写出来，如

```
public void getEven() {
    int[][] cells = new int[][]{{1, 2, 3, 4}, {11, 12, 13, 14}, {21, 22, 23, 24}};
    List<Integer> ans = new ArrayList<>();
    for (int i = 0; i < cells.length; i ++) {
        for (int j = 0; j < cells[0].length; j++) {
            if ((cells[i][j] & 1) == 0) {
                ans.add(cells[i][j]);
            }
        }
    }
    System.out.println(ans);
}
```

上面这个实现没啥问题，但是这个代码的深度很容易就有三层了；当上面这个if中如果再有其他的判定条件，那么这个代码层级很容易增加了；二维数组还好，如果是三维数组，一个遍历就是三层；再加点逻辑，四层、五层不也是分分钟的事情么

那么问题来了，代码层级变多之后会有什么问题呢？

> 只要代码能跑，又能有什么问题呢？！

1. 函数方法消减代码层级

由于多维数组的遍历层级天然就很深，那么有办法进行消减么？

要解决这个问题，关键是要抓住重点，遍历的重点是什么？获取每个元素的坐标！那么我们可以怎么办？

> 定义一个函数方法，输入的就是函数坐标，在这个函数体中执行我们的遍历逻辑即可

基于上面这个思路，相信我们可以很容易写一个二维的数组遍历通用方法

```
public static void scan(int maxX, int maxY, BiConsumer<Integer, Integer> consumer) {
    for (int i = 0; i < maxX; i++) {
        for (int j = 0; j < maxY; j++) {
            consumer.accept(i, j);
        }
    }
}
```

主要上面的实现，函数方法直接使用了JDK默认提供的BiConsumer，两个传参，都是int 数组下表；无返回值

那么上面这个怎么用呢？

同样是上面的例子，改一下之后，如

```
public void getEven() {
    int[][] cells = new int[][]{{1, 2, 3, 4}, {11, 12, 13, 14}, {21, 22, 23, 24}};
    List<Integer> ans = new ArrayList<>();
    scan(cells.length, cells[0].length, (i, j) -> {
        if ((cells[i][j] & 1) == 0) {
            ans.add(cells[i][j]);
        }
    });
    System.out.println(ans);
}
```

相比于前面的，貌似也就少了一层而已，好像也没什么了不起的

但是，当数组变为三维、四维、无维时，这个改动的写法层级都不会变哦

2. 遍历中return支持

前面的实现对于正常的遍历没啥问题；但是当我们在遍历过程中，遇到某个条件直接返回，能支持么？

如一个遍历二维数组，我们希望判断其中是否有偶数，那么可以怎么整？

仔细琢磨一下我们的scan方法，希望可以支持return，主要的问题点就是这个函数方法执行之后，我该怎么知道是继续循环还是直接return呢？

很容易想到的就是执行逻辑中，添加一个额外的返回值，用于标记是否中断循环直接返回

基于此思路，我们可以实现一个简单的demo版本

定义一个函数方法，接受循环的下标 + 返回值

```
@FunctionalInterface
public interface ScanProcess<T> {
    ImmutablePair<Boolean, T> accept(int i, int j);
}
```

循环通用方法就可以相应的改成

```
public static <T> T scanReturn(int x, int y, ScanProcess<T> func) {
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
            ImmutablePair<Boolean, T> ans = func.accept(i, j);
            if (ans != null && ans.left) {
                return ans.right;
            }
        }
    }
    return null;
}
```

基于上面这种思路，我们的实际使用姿势如下

```
@Test
public void getEven() {
    int[][] cells = new int[][]{{1, 2, 3, 4}, {11, 12, 13, 14}, {21, 22, 23, 24}};
    List<Integer> ans = new ArrayList<>();
    scanReturn(cells.length, cells[0].length, (i, j) -> {
        if ((cells[i][j] & 1) == 0) {
            return ImmutablePair.of(true, i + "_" + j);
        }
        return ImmutablePair.of(false, null);
    });
    System.out.println(ans);
}
```

上面这个实现可满足我们的需求，唯一有个别扭的地方就是返回，总有点不太优雅；那么除了这种方式之外，还有其他的方式么？

既然考虑了返回值，那么再考虑一下传参呢？通过一个定义的参数来装在是否中断以及返回结果，是否可行呢？

基于这个思路，我们可以先定义一个参数包装类

```
public static class Ans<T> {
    private T ans;
    private boolean tag = false;

    public Ans<T> setAns(T ans) {
        tag = true;
        this.ans = ans;
        return this;
    }

    public T getAns() {
        return ans;
    }
}

public interface ScanFunc<T> {
    void accept(int i, int j, Ans<T> ans)
}
```

我们希望通过Ans这个类来记录循环结果，其中tag=true，则表示不用继续循环了，直接返回ans结果吧

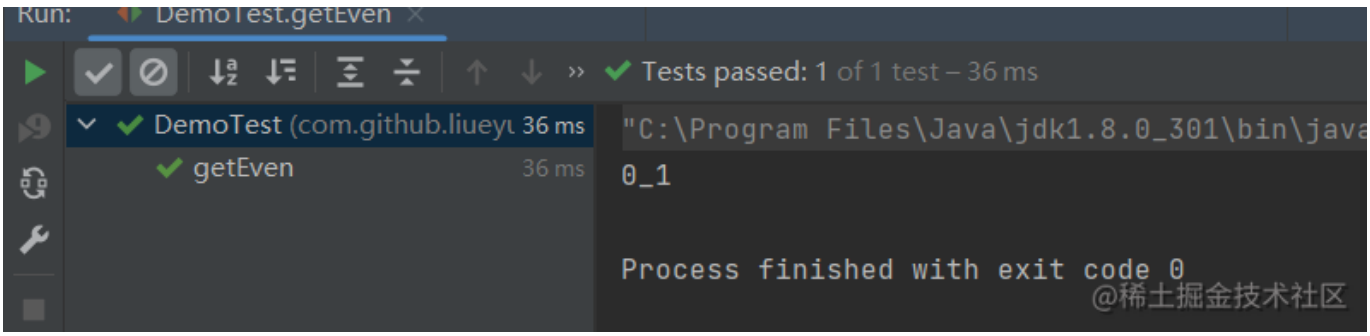
与之对应的方法改造及实例如下

```
public static <T> T scanReturn(int x, int y, ScanFunc<T> func) {
    Ans<T> ans = new Ans<>();
    for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
            func.accept(i, j, ans);
            if (ans.tag) {
                return ans.ans;
            }
        }
    }
    return null;
}

public void getEven() {
    int[][] cells = new int[][]{{1, 2, 3, 4}, {11, 12, 13, 14}, {21, 22, 23, 24}};
    String ans = scanReturn(cells.length, cells[0].length, (i, j, a) -> {
        if ((cells[i][j] & 1) == 0) {
            a.setAns(i + "_" + j);
        }
    });
    System.out.println(ans);
}
```

这样看起来就比前面的要好一点了

实际跑一下，看下输出是否和我们预期的一致；



实战13：List转Map List的几种姿势

今天介绍一个实用的小知识点，如何将List转为`Map<Object, List<Object>>`

1. 转换方式

1.1. 基本写法

最开始介绍的当然是最常见、最直观的写法，当然也是任何限制的写法

```
// 比如将下面的列表，按照字符串长度进行分组
List<String> list = new ArrayList<>();
list.add("hello");
list.add("word");
list.add("come");
list.add("on");
Map<Integer, List<String>> ans = new HashMap<>();

for(String str: list) {
    List<String> sub = ans.get(str.length());
    if(sub == null) {
        sub = new ArrayList<>();
        ans.put(str.length(), sub);
    }
    sub.add(str);
}
System.out.println(ans);
```

对于jdk8+，上面for循环中的内容可以利用`Map.computeIfAbsent`来替换，具体写法如下

```
for (String str : list) {
    ans.computeIfAbsent(str.length(), k -> new ArrayList<>()).add(str);
}
```

当然既然已经是jdk1.8了，借助Stream的流处理，可以将上面的更一步进行简化，如下

```
Map<Integer, List<String>> ans =
list.stream().collect(Collectors.groupingBy(String::length));
```

1.2. 通用方法

上面是针对特定的列表，针对业务进行开发转换，那么我们接下来尝试构建一个通用的工具类

这里我们主要借助的知识点就是泛型，一个重要的点就是如何获取Map中的key

对于jdk < 1.8的写法，通过接口来定义实现key的获取姿势

```
public static <K, V> Map<K, List<V>> toMapList(List<V> list, KeyFunc<V, K> keyFunc) {
    Map<K, List<V>> result = new HashMap<>();
    for (V item: list) {
        K key = keyFunc.getKey(item);
        if (!result.containsKey(key)) {
            result.put(key, new ArrayList<>());
        }
        result.get(key).add(item);
    }
    return result;
}

public static interface KeyFunc<T, K> {
    K getKey(T t);
}
```

使用demo如下

```
public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("hello");
    list.add("word");
    list.add("come");
    list.add("on");
    Map<Integer, List<String>> res = toMapList(list, new KeyFunc<String, Integer>() {
        @Override
        public Integer getKey(String s) {
            return s.length();
        }
    });
    System.out.println(res);
}
```

接下来再看一下jdk1.8之后的写法，结合stream + 函数方法来实现

```
public static <K, V> Map<K, List<V>> toMapList(List<V> list, Function<V, K> func) {
    return list.stream().collect(Collectors.groupingBy(func));
}
```

其对应的使用方式则如下

```
public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("hello");
    list.add("word");
    list.add("come");
    list.add("on");
    Map<Integer, List<String>> res = toMapList(list, (Function<String, Integer>)
String::length);
    System.out.println(res);
}
```

1.3. 工具类

上一节介绍了基于泛型 + jdk8 Stream + 函数方法来实现通用转换工具类的实现姿势，接下来我们小结一下，输出一个适用于1.8之后的工具类

```
/**
 * List<V>转换为Map<K, List<V>> 特点在于Map中的value，是个列表，且列表中的元素就是从原列表中的元素
 *
 * @param list
 * @param func 基于list#item生成Map.key的函数方法
 * @param <K>
 * @param <V>
 * @return
 */
public static <K, V> Map<K, List<V>> toMapList(List<V> list, Function<V, K> func) {
    return list.stream().collect(Collectors.groupingBy(func));
}

/**
 * List<I>转换为Map<K, List<V>> 特点在于Map中的value是个列表，且列表中的元素是由list.item转换而来
 *
 * @param list
 * @param keyFunc 基于list#item生成的Map.key的函数方法
 * @param valFunc 基于list#item转换Map.value列表中元素的函数方法
 * @param <K>
 * @param <I>
 * @param <V>
 * @return
 */
public static <K, I, V> Map<K, List<V>> toMapList(List<I> list, Function<I, K>
keyFunc, Function<I, V> valFunc) {
    return list.stream().collect(Collectors.groupingBy(keyFunc,
Collectors.mapping(valFunc, Collectors.toList())));
}
```

1.4.guava HashMultimap扩展知识点

最后再介绍一个扩展知识点，Gauva工具包中提供了一个`HashMultimap`的工具类，他的使用姿势和我们平常的Map并无差别，但是需要在注意的是，它的value是个集合

```
List<String> list = new ArrayList<>();
list.add("hello");
list.add("word");
list.add("come");
list.add("on");
list.add("on");
HashMultimap<Integer, String> map = HashMultimap.create();
for (String item: strList) {
    map.put(item.length(), item);
}
System.out.println(map);
```

实际输出如下，验证了value实际上是个集合（on只有一个，如果是我们上面的工具类，会输出两个）

```
{2=[on], 4=[word, come], 5=[hello]}
```

实战14：分页遍历得两种实现策略

在日常开发中，分页遍历迭代的场景可以说非常普遍了，比如扫表，每次捞100条数据，然后遍历这100条数据，依次执行某个业务逻辑；这100条执行完毕之后，再加载下一百条数据，直到扫描完毕

那么要实现上面这种分页迭代遍历的场景，我们可以怎么做呢

本文将介绍两种使用姿势

- 常规的使用方法
- 借助Iterator的使用姿势

1. 实现方式

1.1. 数据查询模拟

首先mock一个分页获取数据的逻辑，直接随机生成数据，并且控制最多返回三页

```
public static int cnt = 0;

private static List<String> randStr(int start, int size) {
    ++cnt;
    if (cnt > 3) {
        return Collections.emptyList();
    } else if (cnt == 3) {
        cnt = 0;
        size -= 2;
    }

    System.out.println("====================== start to gen randList
====================");
    List<String> ans = new ArrayList<>(size);
    for (int i = 0; i < size; i++) {
        ans.add((start + i) + "_" + UUID.randomUUID().toString());
    }
    return ans;
}
```

1.2. 基本实现方式

针对这种场景，最常见也是最简单直观的实现方式

- while死循环
- 内部遍历

```
private static void scanByNormal() {
    int start = 0;
    int size = 5;
    while (true) {
        List<String> list = randStr(start, size);
        for (String str : list) {
            System.out.println(str);
        }

        if (list.size() < size) {
            break;
        }
        start += list.size();
    }
}
```

1.3. 迭代器实现方式

接下来介绍一种更有意思的方式，借助迭代器的遍历特性来实现，首先自定义一个通用分页迭代器

```java
public static abstract class MyIterator<T> implements Iterator<T> {
    private int start = 0;
    private int size = 5;

    private int currentIndex;
    private boolean hasMore = true;
    private List<T> list;

    public MyIterator() {
    }

    @Override
    public boolean hasNext() {
        if (list != null && list.size() > currentIndex) {
            return true;
        }

        // 当前的数据已经加载完毕，尝试加载下一批
        if (!hasMore) {
            return false;
        }

        list = load(start, size);
        if (list == null || list.isEmpty()) {
            // 没有加载到数据，结束
            return false;
        }

        if (list.size() < size) {
            // 返回条数小于限制条数，表示还有更多的数据可以加载
            hasMore = false;
        }

        currentIndex = 0;
        start += list.size();
        return true;
    }

    @Override
    public T next() {
        return list.get(currentIndex++);
    }

    public abstract List<T> load(int start, int size);
}
```

接下来借助上面的迭代器可以比较简单的实现我们的需求了

```
private static void scanByIterator() {
    MyIterator<String> iterator = new MyIterator<String>() {
        @Override
        public List<String> load(int start, int size) {
            return randStr(start, size);
        }
    };

    while (iterator.hasNext()) {
        String str = iterator.next();
        System.out.println(str);
    }
}
```

那么问题来了，上面这种使用方式比前面的优势体现再哪儿呢?

- 双层循环改为单层循环

接下来接入重点了，在jdk1.8引入了函数方法 + lambda之后，又提供了一个更简洁的使用姿势

```
public class IteratorTestForJdk18 {

    @FunctionalInterface
    public interface LoadFunc<T> {
        List<T> load(int start, int size);
    }

    public static class MyIterator<T> implements Iterator<T> {
        private int start = 0;
        private int size = 5;

        private int currentIndex;
        private boolean hasMore = true;
        private List<T> list;
        private LoadFunc<T> loadFunc;

        public MyIterator(LoadFunc<T> loadFunc) {
            this.loadFunc = loadFunc;
        }

        @Override
        public boolean hasNext() {
            if (list != null && list.size() > currentIndex) {
                return true;
            }

            // 当前的数据已经加载完毕，尝试加载下一批
            if (!hasMore) {
                return false;
            }

            list = loadFunc.load(start, size);
            if (list == null || list.isEmpty()) {
                // 没有加载到数据，结束
                return false;
            }

            if (list.size() < size) {
                // 返回条数小于限制条数，表示还有更多的数据可以加载
                hasMore = false;
            }

            currentIndex = 0;
            start += list.size();
            return true;
        }
```

```
        @Override
        public T next() {
            return list.get(currentIndex++);
        }
    }
}
```

在jdk1.8及之后的使用姿势，一行代码即可

```
private static void scanByIteratorInJdk8() {
    new MyIterator<>(IteratorTestForJdk18::randStr)
        .forEachRemaining(System.out::println);
}
```

这次对比效果是不是非常显眼了，从此以后分页迭代遍历再也不用冗长的双重迭代了

实战15：数组拷贝

说实话，在实际的业务开发中，基本上很少很少很少...会遇到数组拷贝的场景，甚至是我们一般都不怎么用数组，List它不香嘛，为啥要用数组

现在问题来了，要实现数组拷贝，怎么整?

1. 实现方式

1.1. 基础写法

最简单直接的写法，那就是新建一个数组，一个一个拷贝进去，不就完事了么

```
String[] data = new String[]{"1", "2", "3"};
String[] ans = new String[data.length];
for (int index = 0; index < data.length; index ++) {
    ans[index] = data[index];
}
```

1.2. 借用容器中转

数组用起来有点麻烦，还是用容器舒爽，借助List来实现数组的拷贝，也就几行代码

```
String[] data = new String[]{"1", "2", "3"};
List<String> list = Arrays.asList(data);
String[] out = new String[data.length];
list.toArray(out);
```

1.3. Array.copy

上面这个有点绕得远了， 直接使用Array.copy

```
String[] data = new String[]{"1", "2", "3"};
String[] out = Arrays.copyOf(data, data.length);
```

1.4. System.arraycopy

除了上面的，还可以使用更基础的用法

```
String[] data = new String[]{"1", "2", "3"};
String[] out = new String[data.length];
System.arraycopy(data, 0, out, 0, data.length);
```

如果有看过jdk源码的小伙伴，上面这个用法应该不会陌生，特别是在容器类，这种数组拷贝的方式比比可见

参数说明:

```
public static native void arraycopy(Object src,  int  srcPos,
        Object dest, int destPos,
        int length);
```

- src : 原数组
- srcPos: 原数组用于拷贝的起始下标
- dest: 拷贝后的数组
- destPos: 目标数组的小标
- length: 原数组中拷贝过去的数组长度

从上面的描述也能看出来，这个方法不仅能实现数组拷贝，还可以实现数组内指定片段的拷贝

实战16：判断类为基础类型or基础类型的包装类

判断一个类是否为基础类型属于常规操作了，一般我们遇到这种case，要怎么处理呢？

一个一个的if/else判断？还是其他的操作姿势？

1. 基础类型判断

基础类型可以借助class类的`isPrimitive`方法来实现判定，使用姿势也简单

```
obj.getClass().isPrimitive()
```

如果返回true，那么这个对象就是基本类型

- boolean
- char
- byte
- short
- int
- long
- float
- double
- void

但是请注意，对于封装类型，比如Long，访问isPrimitive返回的是false

2. 封装类型判断

那么封装类型可以怎么判断呢？难道一个一个的判定不成？

首先我们注意到`Class#isPrimitive`的方法签名，如下

```
/**
 * @see     java.lang.Boolean#TYPE
 * @see     java.lang.Character#TYPE
 * @see     java.lang.Byte#TYPE
 * @see     java.lang.Short#TYPE
 * @see     java.lang.Integer#TYPE
 * @see     java.lang.Long#TYPE
 * @see     java.lang.Float#TYPE
 * @see     java.lang.Double#TYPE
 * @see     java.lang.Void#TYPE
 * @since JDK1.1
 */
public native boolean isPrimitive();
```

上面的注释中，提到了Boolean#Type之类的静态成员，也就是说包装类型，都有一个TYPE的静态成员

比如boolean的是这个

```
@SuppressWarnings("unchecked")
public static final Class<Boolean> TYPE = (Class<Boolean>)
Class.getPrimitiveClass("boolean");
```

所以我们可以通过这个TYPE来判定，当前对象是否为封装对象

```
try {
    return ((Class) clz.getField("TYPE").get(null)).isPrimitive();
} catch (Exception e) {
    return false;
}
```

如果Class对象没有TYPE字段，那么就不是封装类，直接抛异常，返回false；当然这种通过异常的方式来判定，并不优雅；但是写法上比我们一个一个的if/else进行对比，要好得多了

实战17：Java对象内存地址输出

1. 输出对象地址

当一个对象没有重写`hascode`方法时，它返回的内存地址，当覆盖之后，我们有什么办法获取对象的内存地址么?

- 使用`System.identityHashCode()`输出内存地址

```
public static void main(String[] args) {
    BaseDo base = new BaseDo();
    base.name = "hello";
    int addr = System.identityHashCode(base);
    System.out.println(base.hashCode() + "|" + addr);
}

public static class BaseDo {
    String name;

    @Override
    public int hashCode() {
        return super.hashCode();
    }
}
```

输出结果如:

```
997608398|997608398
```

这个有啥用?

- 判断两个对象是否为同一个对象时，可以借用（我是在验证Mybatis的一级缓存的，判断返回的Entity是否确实是同一个的时候以此来判定的）

实战18：随机数生成怎么选

随机数生成，java中有一个专门的Random类来实现，除此之外，使用`Math.random`的也比较多，接下来我们简单学习下，随机数的使用姿势

1.随机数生成

1.1. Math.random

jdk提供的基础工具类Math中封装一些常用的基础方法，比如我们今天的主题，生成随机数，使用姿势如下

```
double val = Math.random();
```

使用起来比较简单，生成的是[0,1)之间的浮点数，但是不要以为它就真的只能生成0-1之间的随机数，举例如下

如果想利用它，生成一个 `[120, 500]` 这个区间的随机数，怎么整?

```
int ans = Double.valueOf(Math.ceil(Math.random() * 381 + 120)).intValue();
```

为啥上面的可行?

将上面的代码翻译一下，取值区间如

`Math.random() * 381 + 120` 取值范围如下

- [0, 1) * 381 + 120
- [0, 381) + 120
- [120, 501)

借助 `Math.ceil` 只取浮点数中的整数部分，这样我们的取值范围就是 [120, 500]了，和我们的预期一致

最后简单来看下， `Math.random()` 是怎么实现随机数的

```
private static final class RandomNumberGeneratorHolder {
    static final Random randomNumberGenerator = new Random();
}

public static double random() {
    return RandomNumberGeneratorHolder.randomNumberGenerator.nextDouble();
}
```

请注意上面的实现，原来底层依然使用的是 `Random` 类来生成随机数，而且上面这种写法属于非常经典的单例模式写法（不同于我们常见的双重判定方式，这种属于内部类的玩法，后面再说为啥

可以这么用)

1.2. Random

除了使用上面的Math.random来获取随机数之外，直接使用Random类也是很常见的case；接下来先简单看一下Random的使用姿势

创建Random对象

```
// 以当前时间戳作为随机种子
Random random = new Random();
// 以固定的数字作为随机种子，好处是每次执行时生成的随机数是一致的，便于场景复现
Random random2 = new Random(10);
```

生成随机数

```
// [0, max) 之间的随机整数
random.nextInt(max);

// 随机返回ture/false
random.nextBoolean()

// 随机长整数
random.nextLong()

// 随机浮点数
random.nextFloat()
random.nextDouble()
```

伪随机高斯分布双精度数

```
random.nextGaussian()
```

随机类的nextGaussian()方法返回下一个伪随机数，即与随机数生成器序列的平均值为0.0，标准差为1.0的高斯(正态)分布双精度值

这种使用场景可能用在更专业的场景，至少我接触过的业务开发中，没有用过这个😂

1.3. Math.random 与 Random如何选

上面两个都可以用来生成随机数，那么在实际使用的时候，怎么选择呢？

从前面的描述也可以知道，它们两没啥本质区别，底层都是用的Random类，在实际的运用过程中，如果我们希望可以场景复现，比如测试中奖概率的场景下，选择Random类，指定随机种子可能更友好；如果只是简单的随机数生成使用，那么选择`Math.random`即可，至少使用起来一行代码即可

实战19：数字格式化

数字的格式化场景，更多的是在日志输出、金额计算相关的领域中会用到，平常我们可能更多使用`String.format`来格式化，但是请注意，数字格式化是有一个`DecimalFormat`，专门来针对数字进行格式化

今天我们的知识点就是DecimalFormat来实现数字格式化

1. 格式化

1.1. DecimalFormat使用说明

对于DecimalFormat的使用比较简单，主要是借助两个占位`0`与`#`，区别在于当格式化的占位数，多余实际数的时候，占位`0`的场景下，会用前缀0来补齐；而`#`则不需要补齐

上面这个可能不太好理解，举例说明如下

```
double num = 3.1415926;
System.out.println(new DecimalFormat("000", num));
System.out.println(new DecimalFormat("###", num));
```

上面两个都是只输出整数，但是输出结果不同，如下

```
003
3
```

简单来说，就是`0`，主要用于定长的输出，对于不足的，前缀补0

整数#小数

除了上面的基本姿势之外，更常见的是设置整数、小数的位数

```
System.out.println(new DecimalFormat("000.00", num));
System.out.println(new DecimalFormat("###.##", num));
```

输出结果如下

```
003.14
3.14
```

百分比

百分比的输出也属于常见的case，使用DecimalFormat就很简单

```
System.out.println(new DecimalFormat("000.00%", num));
System.out.println(new DecimalFormat("###.##%", num));
```

输出如下

```
314.16%
314.16%
```

科学计数

非专业场景下，科学技术的可能性比较小

```
System.out.println(new DecimalFormat("000.00E0", num));
System.out.println(new DecimalFormat("###.##E0", num));
```

输出结果如下

```
314.16E-2
3.1416E0
```

金钱样式输出

金融相关的钱输出时，非常有意思的是每三位加一个逗号分隔，如果想实现这个效果，也可以很简单完成

```
double num = 31415926
System.out.println(new DecimalFormat(",###", num));
```

输出结果如下

```
31,415,926
```

嵌入模板输出

格式化模板，除了基础的`000, ###`之外，还可以直接放在一个字符串中，实现类似`String.format`的效果

比如显示余额

```
double num = 31415926
System.out.println(new DecimalFormat("您的余额,###￥", num));
```

输出结果如下

```
您的余额31,415,926￥
```

实战20：进制转换很简单

进制转换，属于基本技能了，在java中要实现进制转换很简单，可以非常简单的实现，接下来我们来看下它的使用姿势

1. 进制转换

1.1. toString实现进制转换

Integer/Long#toString(int i, int radix) 可以将任一进制的整数，转换为其他任意进制的整数

- 第一个参数：待转换的数字
- 第二个参数：转换后的进制位

十六进制转十进制

```
Integer.toString(0x12, 10)
```

八进制转是十进制

```
Integer.toString(012, 10)
```

八进制转二进制

```
Integer.toString(012, 2)
```

1.2. 十进制转二进制

除了使用上面的姿势之外，可以直接使用`toBinaryString`来实现转二进制

```
Integer.toBinaryString(2)
Long.toBinaryString(2)
```

1.3. 十进制转八进制

`Integer/Long#toOctalString`：转八进制

```
Integer.toOctalString(9)
```

1.4. 十进制转十六进制

`Integer/Long#toHexString`：转十六进制

```
Integer.toHexString(10)
```

实战21：Properties配置文件

properties配置文件，相信各位小伙伴都不会太陌生，常用Spring的可能会经常看到它，虽说现在更推荐的是使用Yaml配置文件，但是properties配置文件的使用频率也不低

在jdk中有一个直接关连的类Properties，接下来我们来看一下它的用法

1. Properties配置类

1.1. 配置文件

properties文件的格式比较简单

- `key = value`: 等号左边的为配置key，右边的为配置value（value值会去除前后的空格）
- `#`： 以`#`来区分注释

一个基础的配置文件如下

```
# 测试
key = value
user.name = 一灰灰blog
user.age = 18
user.skill = java,python,js,shell
```

1.2. 配置文件加载

对于Properties配置文件，我们可以非常简单的借助`Properties`类，来实现配置的加载

```
public class PropertiesUtil {

    /**
     * 从文件中读取配置
     *
     * @param propertyFile
     * @return
     * @throws IOException
     */
    public static Properties loadProperties(String propertyFile) throws IOException {
        Properties config = new Properties();

config.load(PropertiesUtil.class.getClassLoader().getResourceAsStream(propertyFile));
        return config;
    }
}
```

直接使用`Properties#config`就可以读取配置文件内容，并赋值到java对象

重点注意：

重点看一下Properties类的继承关系，它的父类是Hashtable, 也就是说它的本质是Map对象

```
public
class Properties extends Hashtable<Object,Object> {
}
```

1.3. Properties对象使用

因为`Properties`是继承自Hashtable，而Hashtable是线程安全的Map容器，因此Properties也是线程安全的，同样的，在多线程并发获取配置的时候，它的性能表现也就不咋地了，why?

首先看一下配置获取

```
// 获取配置属性
public String getProperty(String key) {
    Object oval = super.get(key);
    String sval = (oval instanceof String) ? (String)oval : null;
    return ((sval == null) && (defaults != null)) ? defaults.getProperty(key) : sval;
}

// 获取配置属性，如果不存在，则返回默认值
public String getProperty(String key, String defaultValue) {
    String val = getProperty(key);
    return (val == null) ? defaultValue : val;
}
```

上面两个方法的使用频率很高，从签名上也很容易知道使用姿势；接下来需要看一下的为啥说并发效率很低

关键点就在第一个方法的`super.get()`，它对应的源码正是

```
public synchronized V get(Object key) {
  // ...
}
```

方法签名上有`synchronized`，所以为啥说并发环境下的性能表现不会特别好也就知道原因了

除了获取配置之外，另外一个常用的就是更新配置

```
public synchronized Object setProperty(String key, String value) {
    return put(key, value);
}
```

2. 小结

本文介绍的知识点主要是properties配置文件的处理，使用同名的java类来操作；需要重点注意的是Properties类属于Hashtable的子类，同样属于容器的范畴

最后提一个扩展的问题，在SpringBoot的配置自动装载中，可以将配置内容自动装载到配置类中，简单来讲就是支持配置到java bean的映射，如果现在让我们来实现这个，可以怎么整？

实战22：Properties配置文件自动装载JavaBean

SpringBoot的配置自动装载，使用起来还是很舒爽的，可以非常简单的将properties配置文件的内容，填充到Java bean对象中，如果我们现在是一个脱离于Springboot框架的项目，想实现上面这个功能，可以怎么来做呢?

1.配置封装

1.1. 配置文件自动装载

前面介绍了Properties文件的读取以及基本使用姿势，通过上篇博文已知Properties类的本质是一个Map，所以我们需要干的就是将Map容器的值，赋值到JavaBean的成员属性中

要实现这个功能，自然而然会想到的就是利用反射（考虑到我们赋值的通常为标准的java bean，使用内省是个更好的选择）

接下来我们需要实现的也比较清晰了，第一步获取成员属性，两种方式

- 内省: `BeanInfo bean = Introspector.getBeanInfo(clz); PropertyDescriptor[] propertyDescriptors = bean.getPropertyDescriptors();`
- 反射: `Field[] fields = clz.getDeclaredFields();`

第二步遍历成员属性，进行赋值

- 内省：借助前面获取的`PropertyDescriptor`对象，拿到set方法，进行赋值
  - `descriptor.getWriteMethod().invoke(obj, value)`
- 反射：适应`Field.set`来赋值
  - `field.set(obj, value);`

注意

- 上面的两种赋值方式，都要求我们传入的value对象类型与定义类型一直，否则会抛类型转换异常

为了避免复杂的类型转换与判定，我们这里介绍下apache的`commons-beanutils`来实现属性拷贝

```
<!-- https://mvnrepository.com/artifact/commons-beanutils/commons-beanutils -->
<dependency>
    <groupId>commons-beanutils</groupId>
    <artifactId>commons-beanutils</artifactId>
    <version>1.9.4</version>
</dependency>
```

接下来核心的实现逻辑如下

```
private static boolean isPrimitive(Class clz) {
    if (clz.isPrimitive()) {
        return true;
    }

    try {
        return ((Class) clz.getField("TYPE").get(null)).isPrimitive();
    } catch (Exception e) {
        return false;
    }
}

public static <T> T toBean(Properties properties, Class<T> type, String prefix) throws
IntrospectionException, IllegalAccessException, InstantiationException,
InvocationTargetException {
    if (prefix == null) {
        prefix = "";
    } else if (!prefix.isEmpty() && !prefix.endsWith(".")) {
        prefix += ".";
    }

    type.getDeclaredFields();

    // 内省方式来初始化
    T obj = type.newInstance();
    BeanInfo bean = Introspector.getBeanInfo(type);
    PropertyDescriptor[] propertyDescriptors = bean.getPropertyDescriptors();
    for (PropertyDescriptor descriptor : propertyDescriptors) {
        // 只支持基本数据类型的拷贝
        Class fieldType = descriptor.getPropertyType();
        if (fieldType == Class.class) {
            continue;
        }

        if (isPrimitive(fieldType) || fieldType == String.class) {
            // 支持基本类型的转换，如果使用 PropertyUtils，则不会实现基本类型 + String的
自动转换
            BeanUtils.setProperty(obj, descriptor.getName(),
properties.getProperty(prefix + descriptor.getName()));
        } else {
            BeanUtils.setProperty(obj, descriptor.getName(), toBean(properties,
fieldType, prefix + descriptor.getName()));
        }
    }
    return obj;
}
```

注意上面的实现，首先通过内省的方式获取所有的成员，然后进行遍历，借助`BeanUtils.setProperty`来实现属性值设置

这里面有两个知识点

- `BeanUtil`还是`PropertyUtil`
  - 它们两都有个设置属性的方法，但是BeanUtil支持简单类型的自动转换；而后者不行，要求类型完全一致
- 非简单类型
  - 对于非简单类型，上面采用了递归的调用方式来处理；请注意，这里并不完善，比如BigDecimal, Date, List, Map这些相对基础的类型，是不太适用的哦

1.2. 功能测试

最后针对上面的实现功能，简单的测试一下，是否可行

配置文件`mail.properties`

```
mail.host=localhost
mail.port=25
mail.smtp.auth=false
mail.smtp.starttlsEnable=false
mail.from=test@yhhblog.com
mail.username=user
mail.password=pwd
```

两个Java Bean

```
@Data
public static class MailProperties {
    private String host;
    private Integer port;
    private Smtp smtp;
    private String from;
    private String username;
    private String password;
}

@Data
public static class Smtp {
    private String auth;
    private String starttlsEnable;
}
```

转换测试类

```
public static Properties loadProperties(String propertyFile) throws IOException {
    Properties config = new Properties();

config.load(PropertiesUtil.class.getClassLoader().getResourceAsStream(propertyFile));
    return config;
}

@Test
public void testParse() throws Exception {
    Properties properties = loadProperties("mail.properties");
    MailProperties mailProperties = toBean(properties, MailProperties.class, "mail");
    System.out.println(mailProperties);
}
```

输出结果如下：

```
PropertiesUtil.MailProperties(host=localhost, port=25,
smtp=PropertiesUtil.Smtp(auth=false, starttlsEnable=false), from=test@yhhblog.com,
username=user, password=pwd)
```

实战23：基于引入包选择具体实现类

最近遇到一个需求场景，开源的工具包，新增了一个高级特性，会依赖json序列化工具，来做一些特殊操作；但是，这个辅助功能并不是必须的，也就是说对于使用这个工具包的业务方而言，正常使用完全不需要json相关的功能；如果我强引用某个json工具，一是对于不适用高级特性的用户而言没有必要；二则是我引入的json工具极有可能与使用者的不一致，会增加使用者的成本

因此我希望这个工具包对外提供时，并不会引入具体的json工具依赖；也就是说maven依赖中的`<scope>`设置为`provided`；具体的json序列化的实现，则取决于调用方自身引入了什么json工具包

那么可以怎么实现上面这个方式呢?

1.实现方式

1.1. 任务说明

上面的简单的说了一下我们需要做的事情，接下来我们重点盘一下，我们到底是要干什么

核心诉求相对清晰

1. 不强引入某个json工具
2. 若需要使用高级特性，则直接使用当前环境中已集成的json序列化工具；若没有提供，则抛异常，不支持

对于上面这个场景，常年使用Spring的我们估计不会陌生，Spring集成了很多的第三方开源组件，根据具体的依赖来选择最终的实现，比如日志，可以是logback，也可以是log4j；比如redis操作，可以是jedis，也可以是lettuce

那么Spring是怎么实现的呢?

1.2.具体实现

在Spring中有个注解名为`ConditionalOnClass`，表示当某个类存在时，才会干某些事情（如初始化bean对象）

它是怎么是实现的呢？（感兴趣的小伙伴可以搜索一下，或者重点关注下`SpringBootCondition`的实现）

这里且抛开Spring的实现姿势，我们采用传统的实现方式，直接判断是否有加载对应的类，来判断有没有引入相应的工具包

如需要判断是否引入了gson包，则判断ClassLoader是否有加载`com.google.gson.Gson`类

```
public static boolean exist(String name) {
    try {
        return JsonUtil.class.getClassLoader().loadClass(name) != null;
    } catch (Exception e) {
        return false;
    }
}
```

上面这种实现方式就可以达到我们的效果了；接下来我们参考下Spring的ClassUtils实现，做一个简单的封装，以判断是否存在某个类

```
// 这段代码来自Spring
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)
//

import java.lang.reflect.Array;
import java.util.HashMap;
import java.util.Map;

/**
 * @author Spring
 */
public abstract class ClassUtils {
    private static final Map<String, Class<?>> primitiveTypeNameMap = new HashMap(32);
    private static final Map<String, Class<?>> commonClassCache = new HashMap(64);

    private ClassUtils() {
    }

    public static boolean isPresent(String className) {
        try {
            forName(className, getDefaultClassLoader());
            return true;
        } catch (IllegalAccessError var3) {
            throw new IllegalStateException("Readability mismatch in inheritance
hierarchy of class [" + className + "]: " + var3.getMessage(), var3);
        } catch (Throwable var4) {
            return false;
        }
    }

    public static boolean isPresent(String className, ClassLoader classLoader) {
        try {
            forName(className, classLoader);
            return true;
        } catch (IllegalAccessError var3) {
            throw new IllegalStateException("Readability mismatch in inheritance
hierarchy of class [" + className + "]: " + var3.getMessage(), var3);
        } catch (Throwable var4) {
            return false;
        }
    }

    public static Class<?> forName(String name, ClassLoader classLoader) throws
ClassNotFoundException, LinkageError {
        Class<?> clazz = resolvePrimitiveClassName(name);
        if (clazz == null) {
```

```
            clazz = (Class) commonClassCache.get(name);
        }

        if (clazz != null) {
            return clazz;
        } else {
            Class elementClass;
            String elementName;
            if (name.endsWith("[]")) {
                elementName = name.substring(0, name.length() - "[]".length());
                elementClass = forName(elementName, classLoader);
                return Array.newInstance(elementClass, 0).getClass();
            } else if (name.startsWith("[L") && name.endsWith(";")) {
                elementName = name.substring("[L".length(), name.length() - 1);
                elementClass = forName(elementName, classLoader);
                return Array.newInstance(elementClass, 0).getClass();
            } else if (name.startsWith("[")) {
                elementName = name.substring("[".length());
                elementClass = forName(elementName, classLoader);
                return Array.newInstance(elementClass, 0).getClass();
            } else {
                ClassLoader clToUse = classLoader;
                if (classLoader == null) {
                    clToUse = getDefaultClassLoader();
                }

                try {
                    return Class.forName(name, false, clToUse);
                } catch (ClassNotFoundException var9) {
                    int lastDotIndex = name.lastIndexOf(46);
                    if (lastDotIndex != -1) {
                        String innerClassName = name.substring(0, lastDotIndex) + '$'
+ name.substring(lastDotIndex + 1);

                        try {
                            return Class.forName(innerClassName, false, clToUse);
                        } catch (ClassNotFoundException var8) {
                        }
                    }

                    throw var9;
                }
            }
        }
    }
```

```
    public static Class<?> resolvePrimitiveClassName(String name) {
        Class<?> result = null;
        if (name != null && name.length() <= 8) {
            result = (Class) primitiveTypeNameMap.get(name);
        }

        return result;
    }

    public static ClassLoader getDefaultClassLoader() {
        ClassLoader cl = null;

        try {
            cl = Thread.currentThread().getContextClassLoader();
        } catch (Throwable var3) {
        }

        if (cl == null) {
            cl = ClassUtils.class.getClassLoader();
            if (cl == null) {
                try {
                    cl = ClassLoader.getSystemClassLoader();
                } catch (Throwable var2) {
                }
            }
        }

        return cl;
    }
}
```

工具类存在之后，我们实现一个简单的json工具类，根据已有的json包来选择具体的实现

```
public class JsonUtil {
    private static JsonApi jsonApi;

    private static void initJsonApi() {
        if (jsonApi == null) {
            synchronized (JsonUtil.class) {
                if (jsonApi == null) {
                    if
(ClassUtils.isPresent("com.fasterxml.jackson.databind.ObjectMapper",
JsonUtil.class.getClassLoader())) {
                        jsonApi = new JacksonImpl();
                    } else if (ClassUtils.isPresent("com.google.gson.Gson",
JsonUtil.class.getClassLoader())) {
                        jsonApi = new GsonImpl();
                    } else if (ClassUtils.isPresent("com.alibaba.fastjson.JSONObject",
JsonUtil.class.getClassLoader())) {
                        jsonApi = new JacksonImpl();
                    } else {
                        throw new UnsupportedOperationException("no json framework to
deserialize string! please import jackson|gson|fastjson");
                    }
                }
            }
        }
    }

    /**
     * json转实体类，会根据当前已有的json框架来执行反序列化
     *
     * @param str
     * @param t
     * @param <T>
     * @return
     */
    public static <T> T toObj(String str, Class<T> t) {
        initJsonApi();
        return jsonApi.toObj(str, t);
    }

    public static <T> String toStr(T t) {
        initJsonApi();
        return jsonApi.toStr(t);
    }
}
```

上面的实现中，根据已有的json序列化工具，选择具体的实现类，我们定义了一个JsonApi接口，然后分别gson,jackson,fastjson给出默认的实现类

```
public interface JsonApi {
    <T> T toObj(String str, Class<T> clz);

    <T> String toStr(T t);
}

public class FastjsonImpl implements JsonApi {
    public <T> T toObj(String str, Class<T> clz) {
        return JSONObject.parseObject(str, clz);
    }

    public <T> String toStr(T t) {
        return JSONObject.toJSONString(t);
    }
}

public class GsonImpl implements JsonApi {
    private static final Gson gson = new Gson();

    public <T> T toObj(String str, Class<T> t) {
        return gson.fromJson(str, t);
    }

    public <T> String toStr(T t) {
        return gson.toJson(t);
    }
}

public class JacksonImpl implements JsonApi{
    private static final ObjectMapper jsonMapper = new ObjectMapper();

    public <T> T toObj(String str, Class<T> clz) {
        try {
            return jsonMapper.readValue(str, clz);
        } catch (Exception e) {
            throw new UnsupportedOperationException(e);
        }
    }

    public <T> String toStr(T t) {
        try {
            return jsonMapper.writeValueAsString(t);
        } catch (Exception e) {
            throw new UnsupportedOperationException(e);
        }
    }
```

```
}
```

最后的问题来了，如果调用方并没有使用上面三个序列化工具，而是使用其他的呢，可以支持么?

既然我们定义了一个JsonApi，那么是不是可以由用户自己来实现接口，然后自动选择它呢?

现在的问题就是如何找到用户自定义的接口实现了

1.3. 扩展机制

对于SPI机制比较熟悉的小伙伴可能非常清楚，可以通过在配置目录`META-INF/services/`下新增接口文件，内容为实现类的全路径名称，然后通过`ServiceLoader.load(JsonApi.class)`的方式来获取所有实现类

除了SPI的实现方式之外，另外一个策略则是上面提到的Spring的实现原理，借助字节码来处理（详情原理后面专文说明）

当然也有更容易想到的策略，扫描包路径下的class文件，遍历判断是否为实现类(额外注意jar包内的实现类场景)

接下来以SPI的方式来介绍下扩展实现方式，首先初始化JsonApi的方式改一下，优先使用用户自定义实现

```
private static void initJsonApi() {
    if (jsonApi == null) {
        synchronized (JsonUtil.class) {
            if (jsonApi == null) {
                ServiceLoader<JsonApi> loader = ServiceLoader.load(JsonApi.class);
                for (JsonApi value : loader) {
                    jsonApi = value;
                    return;
                }

                if
(ClassUtils.isPresent("com.fasterxml.jackson.databind.ObjectMapper",
JsonUtil.class.getClassLoader())) {
                    jsonApi = new JacksonImpl();
                } else if (ClassUtils.isPresent("com.google.gson.Gson",
JsonUtil.class.getClassLoader())) {
                    jsonApi = new GsonImpl();
                } else if (ClassUtils.isPresent("com.alibaba.fastjson.JSONObject",
JsonUtil.class.getClassLoader())) {
                    jsonApi = new JacksonImpl();
                } else{
                    throw new UnsupportedOperationException("no json framework to
deserialize string! please import jackson|gson|fastjson");
                }
            }
        }
    }
}
```

对于使用者而言，首先是实现接口

```
package com.github.hui.quick.plugin.test;

import com.github.hui.quick.plugin.qrcode.util.json.JsonApi;

public class DemoJsonImpl implements JsonApi {
    @Override
    public <T> T toObj(String str, Class<T> clz) {
        // ...
    }

    @Override
    public <T> String toStr(T t) {
        // ...
    }
}
```

接着就是实现定义，`resources/META-INF/services/` 目录下，新建文件名为 `com.github.hui.quick.plugin.qrcode.util.json.JsonApi`

内容如下

```
com.github.hui.quick.plugin.test.DemoJsonImpl
```

然后完工~

2. 小结

主要介绍一个小的知识点，如何根据应用已有的jar包来选择具体的实现类的方式；本文介绍的方案是通过ClassLoader来尝试加载对应的类，若能正常加载，则认为有；否则认为没有；这种实现方式虽然非常简单，但是请注意，它是有缺陷的，至于缺陷是啥...

除此之外，也可以考虑通过字节码的方式来判断是否有某个类，或者获取某个接口的实现；文中最后抛出了一个问题，如何获取接口的所有实现类

常见的方式有下面三类（具体介绍了SPI的实现姿势，其他的两种感兴趣的可以搜索一下）

- SPI定义方式
- 扫描包路径
- 字节码方式(如Spring，如Tomcat的 `@HandlesTypes`)