[SCA] Java代码规范

- 性能提升

需要 Map 的主键和取值时，应该迭代 entrySet()

当循环中只需要 Map 的主键时，迭代 keySet() 是正确的。但是，当需要主键和取值时，迭代 entrySet() 才是更高效的做法，比先迭代 keySet() 后再去 get 取值性能更佳。

反例：

```
Map<String, String> map = ...;
for (String key : map.keySet()) {
    String value = map.get(key);
    ...
}
```

正例：

```
Map<String, String> map = ...;
for (Map.Entry<String, String> entry : map.entrySet()) {
    String key = entry.getKey();
    String value = entry.getValue();
    ...
}
```

应该使用Collection.isEmpty()检测空

使用 Collection.size() 来检测空逻辑上没有问题，但是使用 Collection.isEmpty()使得代码更易读，并且可以获得更好的性能。任何 Collection.isEmpty() 实现的时间复杂度都是 O(1)，但是某些 Collection.size() 实现的时间复杂度可能是 O(n)。

反例：

```
if (collection.size() == 0) {
    ...
}
```

正例：

```
if (collection.isEmpty()) {
    ...
}
```

如果需要还需要检测 null，可采用CollectionUtils.isEmpty(collection)和CollectionUtils.isNotEmpty(collection)。

集合初始化尽量指定大小

java 的集合类用起来十分方便，但是看源码可知，集合也是有大小限制的。每次扩容的时间复杂度很有可能是 O(n)，所以尽量指定可预知的集合大小，能减少集合的扩容次数。

反例：

```
int[] arr = new int[]{1, 2, 3};
List<Integer> list = new ArrayList<>();
for (int i : arr) {
    list.add(i);
}
```

正例：

```
int[] arr = new int[]{1, 2, 3};
List<Integer> list = new ArrayList<>(arr.length);
for (int i : arr) {
    list.add(i);


}
```

字符串拼接使用 StringBuilder

一般的字符串拼接在编译期 java 会进行优化，但是在循环中字符串拼接， java 编译期无法做到优化，所以需要使用 StringBuilder 进行替换。

反例：

```
String s = "";
for (int i = 0; i < 10; i++) {
    s += i;
}
```

正例：

```
String a = "a";
String b = "b";
String c = "c";
String s = a + b + c; // java
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10; i++) {
    sb.append(i);  // javaStringBuilder
}
```

List 的随机访问

大家都知道数组和链表的区别：数组的随机访问效率更高。当调用方法获取到 List 后，如果想随机访问其中的数据，并不知道该数组内部实现是链表还是数组，怎么办呢？可以判断它是否实现 RandomAccess 接口。

正例：

```
// list
List<Integer> list = otherService.getList();
if (list instanceof RandomAccess) {
    //
    System.out.println(list.get(list.size() - 1));
} else {
    //
}
```

频繁调用 Collection.contains 方法请使用 Set

在 java 集合类库中，List 的 contains 方法普遍时间复杂度是 O(n)，如果在代码中需要频繁调用 contains 方法查找数据，可以先将 list 转换成 HashSet 实现，将 O(n) 的时间复杂度降为 O(1)。

反例：

```
ArrayList<Integer> list = otherService.getList();
for (int i = 0; i <= Integer.MAX_VALUE; i++) {
    // O(n)
    list.contains(i);
}
```

正例：

```
ArrayList<Integer> list = otherService.getList();
Set<Integer> set = new HashSet(list);
for (int i = 0; i <= Integer.MAX_VALUE; i++) {
    // O(1)
    set.contains(i);
}
```

- 代码规范

不要使用集合实现来赋值静态成员变量

对于集合类型的静态成员变量，不要使用集合实现来赋值，应该使用静态代码块赋值。

反例：

```
private static Map<String, Integer> map = new HashMap<String, Integer>() {
    {
        put("a", 1);
        put("b", 2);
    }
};


private static List<String> list = new ArrayList<String>() {
    {
        add("a");
        add("b");
    }
};
```

正例:

```
private static Map<String, Integer> map = new HashMap<>();
static {
    map.put("a", 1);
    map.put("b", 2);
};


private static List<String> list = new ArrayList<>();
static {
    list.add("a");
    list.add("b");
};
```

建议使用 try-with-resources 语句

Java 7 中引入了 try-with-resources 语句，该语句能保证将相关资源关闭，优于原来的 try-catch-finally 语句，并且使程序代码更安全更简洁。

反例:

```
private void handle(String fileName) {
    BufferedReader reader = null;
    try {
        String line;
        reader = new BufferedReader(new FileReader(fileName));
        while ((line = reader.readLine()) != null) {
            ...
        }
    } catch (Exception e) {
        ...
    } finally {
        if (reader != null) {
            try {
                reader.close();
            } catch (IOException e) {
                ...
            }
        }
    }
}
```

正例:

```
private void handle(String fileName) {
    try (BufferedReader reader = new BufferedReader(new FileReader(fileName))) {
        String line;
        while ((line = reader.readLine()) != null) {
            ...
        }
    } catch (Exception e) {
        ...
    }
}
```

工具类应该屏蔽构造函数

工具类是一堆静态字段和函数的集合，不应该被实例化。但是，Java 为每个没有明确定义构造函数的类添加了一个隐式公有构造函数。所以，为了避免 java "小白"使用有误，应该显式定义私有构造函数来屏蔽这个隐式公有构造函数。

反例:

```
public class MathUtils {
    public static final double PI = 3.1415926D;
    public static int sum(int a, int b) {
        return a + b;
    }
}
```

正例：

```
public class MathUtils {
    public static final double PI = 3.1415926D;
    private MathUtils() {}
    public static int sum(int a, int b) {
        return a + b;
    }
}
```

不要用NullPointerException判断空

空指针异常应该用代码规避（比如检测不为空），而不是用捕获异常的方式处理。

反例：

```
public String getUserName(User user) {
    try {
        return user.getName();
    } catch (NullPointerException e) {
        return null;
    }
}
```

正例：

```
public String getUserName(User user) {
    if (Objects.isNull(user)) {
        return null;
    }
    return user.getName();
}
```

使用String.valueOf(value)代替""+value

当要把其它对象或类型转化为字符串时，使用 String.valueOf(value) 比""+value 的效率更高。

反例：

```
int i = 1;
String s = "" + i;
```

正例：

```
int i = 1;
String s = String.valueOf(i);
```

过时代码添加 @Deprecated 注解

当一段代码过时，但为了兼容又无法直接删除，不希望以后有人再使用它时，可以添加 @Deprecated 注解进行标记。在文档注释中添加 @deprecated 来进行解释，并提供可替代方案。

正例：

```
/** *  * * @deprecated {@link newSave()} */
@Deprecated
public void save(){
    // do something
}
```

返回空数组和空集合而不是 null

返回 null，需要调用方强制检测 null，否则就会抛出空指针异常。返回空数组或空集合，有效地避免了调用方因为未检测 null 而抛出空指针异常，还可以删除调用方检测 null 的语句使代码更简洁。

反例:

```
public static Result[] getResults() {
    return null;
}


public static List<Result> getResultList() {
    return null;
}


public static Map<String, Result> getResultMap() {
    return null;
}


public static void main(String[] args) {
    Result[] results = getResults();
    if (results != null) {
        for (Result result : results) {
            ...
        }
    }


    List<Result> resultList = getResultList();
    if (resultList != null) {
        for (Result result : resultList) {
            ...
        }
    }


    Map<String, Result> resultMap = getResultMap();
    if (resultMap != null) {
        for (Map.Entry<String, Result> resultEntry : resultMap) {
            ...
        }
    }
}
```

正例:

```
public static Result[] getResults() {
    return new Result[0];
}


public static List<Result> getResultList() {
    return Collections.emptyList();
}


public static Map<String, Result> getResultMap() {
    return Collections.emptyMap();
}


public static void main(String[] args) {
    Result[] results = getResults();
    for (Result result : results) {
        ...
    }


    List<Result> resultList = getResultList();
    for (Result result : resultList) {
        ...
    }


    Map<String, Result> resultMap = getResultMap();
    for (Map.Entry<String, Result> resultEntry : resultMap) {
        ...
    }
}
```

优先使用常量或确定值来调用 equals 方法

对象的 equals 方法容易抛空指针异常，应使用常量或确定有值的对象来调用 equals 方法。当然，使用 java.util.Objects.equals() 方法是最佳实践。

反例：

```
public void isFinished(OrderStatus status) {
    return status.equals(OrderStatus.FINISHED); //
}
```

正例：

```
public void isFinished(OrderStatus status) {
    return OrderStatus.FINISHED.equals(status);
}


public void isFinished(OrderStatus status) {
    return Objects.equals(status, OrderStatus.FINISHED);
}
```

枚举的属性字段必须是私有不可变

枚举通常被当做常量使用，如果枚举中存在公共属性字段或设置字段方法，那么这些枚举常量的属性很容易被修改。理想情况下，枚举中的属性字段是私有的，并在私有构造函数中赋值，没有对应的 Setter 方法，最好加上 final 修饰符。

反例：

```
public enum UserStatus {
    DISABLED(0, ""),
    ENABLED(1, "");


    public int value;
    private String description;


    private UserStatus(int value, String description) {
        this.value = value;
        this.description = description;
    }


    public String getDescription() {
        return description;
    }


    public void setDescription(String description) {
        this.description = description;
    }
}
```

正例：

```
public enum UserStatus {
    DISABLED(0, ""),
    ENABLED(1, "");


    private final int value;
    private final String description;


    private UserStatus(int value, String description) {
        this.value = value;
        this.description = description;
    }


    public int getValue() {
        return value;
    }


    public String getDescription() {
        return description;
    }
}
```

小心String.split(String regex)

字符串 String 的 split 方法，传入的分隔字符串是正则表达式！部分关键字（比如.[]()| 等）需要转义。

反例：

```
"a.ab.abc".split("."); // []
"a|ab|abc".split("|"); // ["a", "|", "a", "b", "|", "a", "b", "c"]
```

正例：

```
"a.ab.abc".split("\\."); // ["a", "ab", "abc"]
"a|ab|abc".split("\\|"); // ["a", "ab", "abc"]
```