

阿里云 开发者社区

Java 开发手册 泰山版

灵魂13问

一线大厂怎么用Java?
全网千万阅读量技术博主深入解读
挖掘阿里巴巴一线团队开发思维

作者: Hollis



阿里云开发者电子书系列



扫码下载
最新版《Java 开发手册》



阿里云开发者“藏经阁”
海量免费电子书下载



扫码关注 Hollis
一个对 Coding 有着独特追求的人

目录

《新版 Java 开发手册》提到的三目运算符的空指针问题到底是个怎么回事？	4
为什么阿里巴巴建议初始化 HashMap 的容量大小？	15
Java 开发手册建议创建 HashMap 时设置初始化容量，但是多少合适呢？	27
为什么阿里巴巴禁止使用 Executors 创建线程池？	31
为什么阿里巴巴要求谨慎使用 ArrayList 中的 subList 方法？	37
为什么阿里巴巴不建议在 for 循环中使用“+”进行字符串拼接？	44
为什么阿里巴巴禁止在 foreach 循环里进行元素的 remove/add 操作？	54
为什么阿里巴巴禁止工程师直接使用日志系统 (Log4j、Logback) 中的 API？	66
为什么阿里巴巴禁止把 SimpleDateFormat 定义成 static 变量？	74
为什么阿里巴巴禁止开发人员使用 isSuccess 作为变量名？	85
为什么阿里巴巴禁止开发人员修改 serialVersionUID 字段的值？	97
为什么阿里巴巴建议开发者谨慎使用继承？	109
为什么阿里巴巴禁止使用 count(列名) 或 count(常量) 来替代 count(*)？	111

《新版 Java 开发手册》提到的三目运算符的空指针问题到底是个怎么回事？

作者 | Hollis

作者简介：Hollis (个人公众号 Id: Hollis)，一个对 Coding 有着独特追求的人，现任阿里巴巴技术专家，个人技术博主，技术文章全网阅读量数千万，《程序员的三门课》联合作者。

最近，阿里巴巴 Java 开发手册发布了最新版——泰山版，这个名字起的不错，一览众山小。

新版新增了 30+ 规约，其中有一条规约引起了作者的关注，那就是手册中提到在三目运算符使用过程中，需要注意自动拆箱导致的 NullPointerException (后文简称：NPE) 问题：

4. **【强制】**三目运算符 condition? 表达式 1：表达式 2 中，高度注意表达式 1 和 2 在类型对齐时，可能抛出因自动拆箱导致的 NPE 异常。

说明：以下两种场景会触发类型对齐的拆箱操作：

- 1) 表达式 1 或表达式 2 的值只要有一个是原始类型。
- 2) 表达式 1 或表达式 2 的值的类型不一致，会强制拆箱升级成表示范围更大的那个类型。

反例：

```
Integer a = 1;
Integer b = 2;
Integer c = null;
Boolean flag = false;
// a*b 的结果是 int 类型，那么 c 会强制拆箱成 int 类型，抛出 NPE 异常
Integer result = (flag? a*b : c);
```

因为这个问题我很久之前 (2015 年) 遇到过，曾经在博客中也记录过，刚好最新的开发手册再次提到了这个知识点，于是把之前的文章内容翻出来并重新整理了一下，带大家一起回顾下这个知识点。

可能有些人看过我之前那篇文章，本文并不是单纯的“旧瓶装新酒”，在重新梳理这个知识点的时候，作者重新翻阅了《The Java Language Specification》，并且对比了 Java SE 7 和 Java SE 8 之后的相关变化，希望可以帮助大家更加全面的理解这个问题。

基础回顾

在详细展开介绍之前，先简单介绍下本文要涉及到的几个重要概念，分别是“三目运算符”、“自动拆装箱”等，如果大家对于这些历史知识有所掌握的话，可以先跳过本段内容，直接看问题重现部分即可。

三目运算符

在《The Java Language Specification》中，三目运算符的官方名称是 `Conditional Operator ? :`，我一般称呼他为条件表达式，详细介绍在 JLS 15.25 中，这里简单介绍下其基本形式和用法：

三目运算符是 Java 语言中的重要组成部分，它也是唯一有 3 个操作数的运算符。形式为：

```
< 表达式 1> ? < 表达式 2> : < 表达式 3>
```

以上，通过 `?`、`:` 组合的形式得到一个条件表达式。其中 `?` 运算符的含义是：先求表达式 1 的值，如果为真，则执行并返回表达式 2 的结果；如果表达式 1 的值为假，则执行并返回表达式 3 的结果。

值得注意的是，一个条件表达式从不会既计算 `< 表达式 2>`，又计算 `< 表达式 3>`。条件运算符是右结合的，也就是说，从右向左分组计算。例如，`a?b:c?d:e` 将按 `a?b:(c?d:e)` 执行。

自动装箱与自动拆箱

介绍过了三目运算符（条件表达式）之后，我们再来简单介绍下 Java 中的自动装箱相关知识。

每一个 Java 开发者一定都对 Java 中的基本数据类型不陌生，Java 中共有 8 种基本数据类型，这些基础数据类型带来一个好处就是他们直接在栈内存中存储，不会在堆上分配内存，使用起来更加高效。

但是，Java 语言是一个面向对象的语言，而基本数据类型不是对象，导致在实际使用过程中有诸多不便，如集合类要求其内部元素必须是 Object 类型，基本数据类型就无法使用。

所以，相对应的，Java 提供了 8 种包装类型，更加方便在需要对象的地方使用。

有了基本数据类型和包装类，带来了一个麻烦就是需要在他们之间进行转换。在 Java SE5 中，为了减少开发人员的工作，Java 提供了自动拆箱与自动装箱功能。

自动装箱：就是将基本数据类型自动转换成对应的包装类。

自动拆箱：就是将包装类自动转换成对应的基本数据类型。

```
Integer i = 10; // 自动装箱

int b = i;      // 自动拆箱
```

我们可以简单理解为，当我们自己写的代码符合装（拆）箱规范的时候，编译器就会自动帮我们拆（装）箱。

自动装箱都是通过包装类的 `valueOf()` 方法来实现的，自动拆箱都是通过包装类对象的 `xxxValue()` 来实现的（如 `booleanValue()`、`longValue()` 等）。

问题重现

在最新版的开发手册中给出了一个例子，提示我们在三目运算符的过程中，可能会进行自动拆箱而导致 NPE 问题。

原文中的例子相对复杂一些，因为他还涉及到多个 Integer 相乘的结果是 int 的问题，我们举一个相对简单的一点的例子先来重现下这个问题：

```
boolean flag = true; // 设置成 true，保证条件表达式的表达式二一定可以执行

boolean simpleBoolean = false; // 定义一个基本数据类型的 boolean 变量

Boolean nullBoolean = null; // 定义一个包装类对象类型的 Boolean 变量，值为 null

boolean x = flag ? nullBoolean : simpleBoolean; // 使用三目运算符并给 x 变量赋值
```

以上代码，在运行过程中，会抛出 NPE：

```
Exception in thread "main" java.lang.NullPointerException
```

而且，这个和你使用的 JDK 版本是无关的，作者分别在 JDK 6、JDK 8 和 JDK 14 上做了测试，均会抛出 NPE。

为了一探究竟，我们尝试对以上代码进行反编译，使用 jad 工具进行反编译后，得到以下代码：

```
boolean flag = true;

boolean simpleBoolean = false;

Boolean nullBoolean = null;

boolean x = flag ? nullBoolean.booleanValue() : simpleBoolean;
```

可以看到，反编译后的代码的最后一行，编译器帮我们做了一次自动拆箱，而就是因为这次自动拆箱，导致代码出现对于一个 null 对象 (`nullBoolean.booleanValue()`) 的调用，导致了 NPE。

那么，为什么编译器会进行自动拆箱呢？什么情况下需要进行自动拆箱呢？

原理分析

关于为什么编辑器会在代码编译阶段对于三目运算符中的表达式进行自动拆箱，其实在《The Java Language Specification》(后文简称 JLS) 的第 15.25 章节中是有相关介绍的。

在不同版本的 JLS 中，关于这部分描述虽然不尽相同，尤其在 Java 8 中有了大幅度的更新，但是其核心内容和原理是不变的。我们直接看 Java SE 1.7 JLS 中关于这部分的描述 (因为 1.7 的表述更加简洁一些)：

The type of a conditional expression is determined as follows: • If the second and third operands have the same type (which may be the null type), then that is the type of the conditional expression. • If one of the second and third operands is of primitive type T, and the type of the other is the result of applying boxing conversion (§ 5.1.7) to T, then the type of the conditional expression is T.

简单的来说就是：当第二位和第三位操作数的类型相同时，则三目运算符表达式的结果和这两位操作数的类型相同。当第二，第三位操作数分别为基本类型 and 该基本类型对应的包装类型时，那么该表达式的结果的类型要求是基本类型。

为了满足以上规定，又避免程序员过度感知这个规则，所以在编译过程中编译器如果发现三目操作符的第二位和第三位操作数的类型分别是基本数据类型 (如 boolean) 以及该基本类型对应的包装类型 (如 Boolean) 时，并且需要返回表达式为包装类型，那么就需要对该包装类进行自动拆箱。

在 Java SE 1.8 JLS 中，关于这部分描述又做了一些细分，再次把表达式区分成布尔型条件表达式 (Boolean Conditional Expressions)、数值型条件表达式 (Numeric Conditional Expressions) 和引用类型条件表达式 (Reference

Conditional Expressions)。

并且通过表格的形式明确的列举了第二位和第三位分别是不同类型时得到的表达式结果值应该是什么，感兴趣的大家可以去翻阅一下。

其实简单总结下，就是：**当第二位和第三位表达式都是包装类型的时候，该表达式的结果才是该包装类型，否则，只要有一个表达式的类型是基本数据类型，则表达式得到的结果都是基本数据类型。如果结果不符合预期，那么编译器就会进行自动拆箱。**(即 Java 开发手册中总结的：只要表达式 1 和表达式 2 的类型有一个是基本类型，就会做触发类型对齐的拆箱操作，只不过如果都是基本类型也就不需要拆箱了。)

如下 3 种情况是我们熟知该规则，在声明表达式的结果的类型时刻意和规则保持一致的情况(为了帮助大家理解，我备注了注释和反编译后的代码)：

```
boolean flag = true;

boolean simpleBoolean = false;

Boolean objectBoolean = Boolean.FALSE;

// 当第二位和第三位表达式都是对象时，表达式返回值也为对象；

Boolean x1 = flag ? objectBoolean : objectBoolean;

// 反编译后代码为: Boolean x1 = flag ? objectBoolean : objectBoolean;

// 因为 x1 的类型是对象，所以不需要做任何特殊操作。


// 当第二位和第三位表达式都为基本类型时，表达式返回值也为基本类型；

boolean x2 = flag ? simpleBoolean : simpleBoolean;

// 反编译后代码为: boolean x2 = flag ? simpleBoolean : simpleBoolean;

// 因为 x2 的类型也是基本类型，所以不需要做任何特殊操作。
```

```
// 当第二位和第三位表达式中有一个为基本类型时，表达式返回值也为基本类型；

boolean x3 = flag ? objectBoolean : simpleBoolean;

// 反编译后代码为: boolean x3 = flag ? objectBoolean.booleanValue() : simpleBoolean;

// 因为 x3 的类型是基本类型，所以需要对其中的包装类进行拆箱。
```

因为我们熟知三目运算符的规则，所以我们就会按照以上方式去定义 x1、x2 和 x3 的类型。

但是，并不是所有人都熟知这个规则，所以在实际应用中，还会出现以下三种定义方式：

```
// 当第二位和第三位表达式都是对象时，表达式返回值也为对象；

boolean x4 = flag ? objectBoolean : objectBoolean;

// 反编译后代码为: boolean x4 = (flag ? objectBoolean : objectBoolean).booleanValue();

// 因为 x4 的类型是基本类型，所以需要表达式结果进行自动拆箱。


// 当第二位和第三位表达式都为基本类型时，表达式返回值也为基本类型；

Boolean x5 = flag ? simpleBoolean : simpleBoolean;

// 反编译后代码为: Boolean x5 = Boolean.valueOf(flag ? simpleBoolean : simpleBoolean);

// 因为 x5 的类型是对象类型，所以需要表达式结果进行自动装箱。


// 当第二位和第三位表达式中有一个为基本类型时，表达式返回值也为基本类型；

Boolean x6 = flag ? objectBoolean : simpleBoolean;

// 反编译后代码为: Boolean x6 = Boolean.valueOf(flag ? objectBoolean.booleanValue() : simpleBoolean);

// 因为 x6 的类型是对象类型，所以需要表达式结果进行自动装箱。
```

所以，日常开发中就有可能出现以上 6 种情况。聪明的读者们读到这里也一定想到了，在以上 6 种情况中，如果是涉及到自动拆箱的，一旦对象的值为 null，就必然会发生 NPE。

举例验证，我们把以上的 x3、x4 以及 x6 中的对象类型设置成 null，分别执行下代码：

```
Boolean nullBoolean = null;

boolean x3 = flag ? nullBoolean : simpleBoolean;

boolean x4 = flag ? nullBoolean : objectBoolean;

Boolean x6 = flag ? nullBoolean : simpleBoolean;
```

以上三种情况，都会在执行时发生 NPE。

其中 x3 和 x6 是三目运算符运算过程中，根据 JLS 的规则确定类型的过程中要做自动拆箱而导致的 NPE。由于使用了三目运算符，并且第二、第三位操作数分别是基本类型和对象。就需要对对象进行拆箱操作，由于该对象为 null，所以在拆箱过程中调用 null.booleanValue() 的时候就报了 NPE。

而 x4 是因为三目运算符运算结束后根据规则他得到的是一个对象类型，但是在给变量赋值过程中进行自动拆箱所导致的 NPE。

小结

如前文介绍，在开发过程中，如果涉及到三目运算符，那么就要高度注意其中的自动拆装箱问题。

最好的做法就是保持三目运算符的第二位和第三位表达式的类型一致，并且如果要把三目运算符表达式给变量赋值的时候，也尽量保持变量的类型和他们保持一致。并且，做好单元测试！！

所以，Java 开发手册中提到要高度注意第二位和第三位表达式的类型对齐过程中由于自动拆箱发生的 NPE 问题，其实还需要注意使用三目运算符表达式给变量赋值的时候由于自动拆箱导致的 NPE 问题。

至此，我们已经介绍完了 Java 开发手册中关于三目运算符使用过程中可能会导致 NPE 的问题。

如果一定要给出一个方法论去避免这个问题的话，那么在使用的过程中，无论是三目运算符中的三个表达式，还是三目运算符表达式要赋值的变量，最好都使用包装类型，可以减少发生错误的概率。

正文内容已完，如果大家对这个问题还有更深的兴趣的话，接下来部分内容是扩展内容，也欢迎学习，不过这部分涉及到很多 JLS 的规范，如果实在看不懂也没关系～

扩展思考

为了方便大家理解，我使用了简单的布尔类型的例子说明了 NPE 的问题。但是实际在代码开发中，遇到的场景可能并没有那么简单，比如说以下代码，大家猜一下能否正常执行：

```
Map<String,Boolean> map = new HashMap<String, Boolean>();  
  
Boolean b = (map!=null ? map.get("Hollis") : false);
```

如果你的答案是 "不能，这里会抛 NPE" 那么说明你看懂了本文的内容，但是，我只能说你只是答对了一半。

因为以上代码，在小于 JDK 1.8 的版本中执行的结果是 NPE，在 JDK 1.8 及以后的版本中执行结果是 null。

之所以会出现这样的不同，这个就说来话长了，我挑其中的重点内容简单介绍下

吧，以下内容主要内容还是围绕 Java 8 的 JLS 。

JLS 15 中对条件表达式（三目运算符）做了细分之后分为三种，区分方式：

- 如果表达式的第二个和第三个操作数都是布尔表达式，那么该条件表达式就是布尔表达式
- 如果表达式的第二个和第三个操作数都是数字型表达式，那么该条件表达式就是数字型表达式
- 除了以上两种以外的表达式就是引用表达式

因为 `Boolean b = (map!=null ? map.get("Hollis") : false);` 表达式中，第二位操作数为 `map.get("test")`，虽然 Map 在定义的时候规定了其值为 Boolean，但是在编译过程中泛型是会被擦除的（泛型的类型擦除），所以，其结果就是 Object。那么根据以上规则判断，这个表达式就是引用表达式。

又跟据 JLS15.25.3 中规定：

- 如果引用条件表达式出现在赋值上下文或调用上下文中，那么条件表达式就是合成表达式

因为，`Boolean b = (map!=null ? map.get("Hollis") : false);` 其实就是一个赋值上下文（关于赋值上下文相见 JLS 5.2），所以 `map!=null ? map.get("Hollis") : false;` 就是合成表达式。

那么 JLS15.25.3 中对合成表达式的操作数类型做了约束：

- 合成的引用条件表达式的类型与其目标类型相同

所以，因为有了这个约束，编译器就可以推断（Java 8 中类型推断，详见 JLS 18）出该表达式的第二个操作数和第三个操作数的结果应该都是 Boolean 类型。

所以，在编译过程中，就可以分别把他们都转成 Boolean 即可，那么以上代码在 Java 8 中反编译后内容如下：

```
Boolean b = maps == null ? Boolean.valueOf(false) : (Boolean)maps.  
get("Hollis");
```

但是在 Java 7 中可没有这些规定 (Java 8 之前的类型推断功能还很弱), 编译器只知道表达式的第二位和第三位分别是基本类型和包装类型, 而无法推断最终表达式类型。

那么他就会先根据 JLS 15.25 的规定, 把返回值结果转换成基本类型。然后在进行变量赋值的时候, 再转换成包装类型:

```
Boolean b = Boolean.valueOf(maps == null ? false : ((Boolean)maps.  
get("Hollis")).booleanValue());
```

所以, 相比 Java 8 中多了一步自动拆箱, 所以会导致 NPE。

参考资料:

《Java 开发手册——泰山版》

<http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.25>

<http://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.25>

<https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.2>

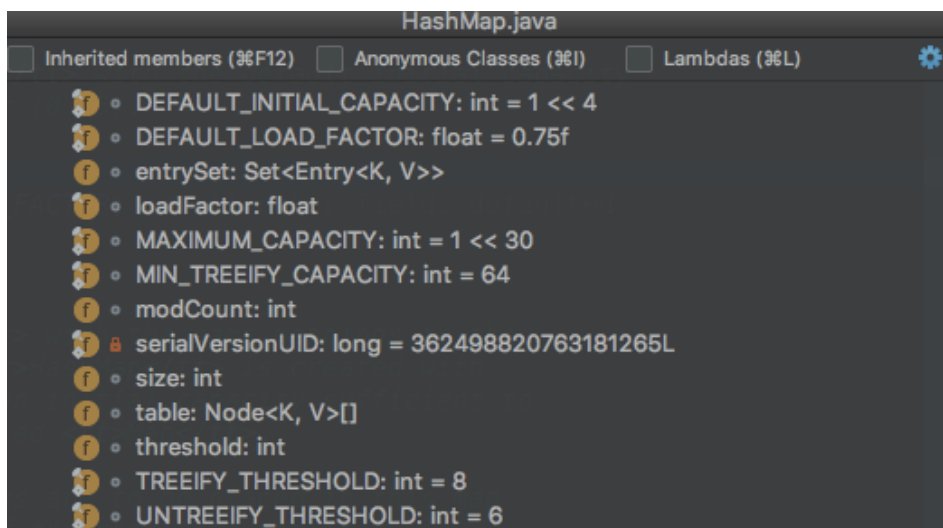
<https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.12.2.7>

<https://docs.oracle.com/javase/specs/jls/se8/html/jls-18.html>

为什么阿里巴巴建议初始化 HashMap 的容量大小？

很多人在通过阅读源码的方式学习 Java，这是个很好的方式。而 JDK 的源码自然是首选。在 JDK 的众多类中，我觉得 HashMap 及其相关的类是设计的比较好的。很多人读过 HashMap 的代码，不知道你们有没有和我一样，觉得 HashMap 中关于容量相关的参数定义的太多了，傻傻分不清楚。

先来看一下，HashMap 中都定义了哪些成员变量。



上面是一张 HashMap 中主要的成员变量的图，其中有一个是我们本文主要关注的：`size`、`loadFactor`、`threshold`、`DEFAULT_LOAD_FACTOR` 和 `DEFAULT_INITIAL_CAPACITY`。

我们先来简单解释一下这些参数的含义，然后再分析他们的作用。

HashMap 类中有以下主要成员变量:

- transient int size;
 - 记录了 Map 中 KV 对的个数
- loadFactor
 - 装载因子, 用来衡量 HashMap 满的程度。loadFactor 的默认值为 0.75f
(`static final float DEFAULT_LOAD_FACTOR = 0.75f;`)。
- int threshold;
 - 临界值, 当实际 KV 个数超过 threshold 时, HashMap 会将容量扩容,
threshold = 容量 * 加载因子
- 除了以上这些重要成员变量外, HashMap 中还有一个和他们紧密相关的概念: capacity
 - 容量, 如果不指定, 默认容量是 16(`static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;`)

可能看完了你还是有点蒙, size 和 capacity 之间有啥关系? 为啥要定义这两个变量。loadFactor 和 threshold 又是干啥的?

size 和 capacity

HashMap 中的 size 和 capacity 之间的区别其实解释起来也挺简单的。我们知道, HashMap 就像一个“桶”, 那么 capacity 就是这个桶“当前”最多可以装多少元素, 而 size 表示这个桶已经装了多少元素。来看下以下代码:

```
Map<String, String> map = new HashMap<String, String>();
map.put("hollis", "hollischuang");

Class<?> mapType = map.getClass();
Method capacity = mapType.getDeclaredMethod("capacity");
capacity.setAccessible(true);
System.out.println("capacity : " + capacity.invoke(map));

Field size = mapType.getDeclaredField("size");
size.setAccessible(true);
```

```
System.out.println("size : " + size.get(map));
```

我们定义了一个新的 HashMap，并向其中 put 了一个元素，然后通过反射的方式打印 capacity 和 size。输出结果为: **capacity : 16、size : 1**

默认情况下，一个 HashMap 的容量 (capacity) 是 16，设计成 16 的好处我在[《全网把 Map 中的 hash\(\) 分析的最透彻的文章，别无二家》](#)中也简单介绍过，主要是可以使用按位与替代取模来提升 hash 的效率。

为什么我刚刚说 capacity 就是这个桶“当前”最多可以装多少元素呢？当前怎么理解呢。其实，HashMap 是具有扩容机制的。在一个 HashMap 第一次初始化的时候，默认情况下他的容量是 16，当达到扩容条件的时候，就需要进行扩容了，会从 16 扩容成 32。

我们知道，HashMap 的重载的构造函数中，有一个是支持传入 initialCapacity 的，那么我们尝试着设置一下，看结果如何。

```
Map<String, String> map = new HashMap<String, String>(1);

Class<?> mapType = map.getClass();
Method capacity = mapType.getDeclaredMethod("capacity");
capacity.setAccessible(true);
System.out.println("capacity : " + capacity.invoke(map));

Map<String, String> map = new HashMap<String, String>(7);

Class<?> mapType = map.getClass();
Method capacity = mapType.getDeclaredMethod("capacity");
capacity.setAccessible(true);
System.out.println("capacity : " + capacity.invoke(map));

Map<String, String> map = new HashMap<String, String>(9);

Class<?> mapType = map.getClass();
Method capacity = mapType.getDeclaredMethod("capacity");
capacity.setAccessible(true);
System.out.println("capacity : " + capacity.invoke(map));
```

分别执行以上 3 段代码，分别输出：**capacity : 2**、**capacity : 8**、**capacity : 16**。

也就是说，默认情况下 HashMap 的容量是 16，但是，如果用户通过构造函数指定了一个数字作为容量，那么 Hash 会选择大于该数字的第一个 2 的幂作为容量。(1->1、7->8、9->16)

这里有一个小建议：在初始化 HashMap 的时候，应该尽量指定其大小。尤其是当你已知 map 中存放的元素个数时。(《阿里巴巴 Java 开发规约》)

loadFactor 和 threshold

前面我们提到过，HashMap 有扩容机制，就是当达到扩容条件时会进行扩容，从 16 扩容到 32、64、128...

那么，这个扩容条件指的是什么呢？

其实，HashMap 的扩容条件就是当 HashMap 中的元素个数 (size) 超过临界值 (threshold) 时就会自动扩容。

在 HashMap 中， $\text{threshold} = \text{loadFactor} * \text{capacity}$ 。

loadFactor 是装载因子，表示 HashMap 满的程度，默认值为 0.75f，设置成 0.75 有一个好处，那就是 0.75 正好是 3/4，而 capacity 又是 2 的幂。所以，两个数的乘积都是整数。

对于一个默认的 HashMap 来说，默认情况下，当其 size 大于 12 ($16 * 0.75$) 时就会触发扩容。

验证代码如下：

```
Map<String, String> map = new HashMap<>();
map.put("hollis1", "hollischuang");
map.put("hollis2", "hollischuang");
```

```

map.put("hollis3", "hollischuang");
map.put("hollis4", "hollischuang");
map.put("hollis5", "hollischuang");
map.put("hollis6", "hollischuang");
map.put("hollis7", "hollischuang");
map.put("hollis8", "hollischuang");
map.put("hollis9", "hollischuang");
map.put("hollis10", "hollischuang");
map.put("hollis11", "hollischuang");
map.put("hollis12", "hollischuang");
Class<?> mapType = map.getClass();

Method capacity = mapType.getDeclaredMethod("capacity");
capacity.setAccessible(true);
System.out.println("capacity : " + capacity.invoke(map));

Field size = mapType.getDeclaredField("size");
size.setAccessible(true);
System.out.println("size : " + size.get(map));

Field threshold = mapType.getDeclaredField("threshold");
threshold.setAccessible(true);
System.out.println("threshold : " + threshold.get(map));

Field loadFactor = mapType.getDeclaredField("loadFactor");
loadFactor.setAccessible(true);
System.out.println("loadFactor : " + loadFactor.get(map));

map.put("hollis13", "hollischuang");
Method capacity = mapType.getDeclaredMethod("capacity");
capacity.setAccessible(true);
System.out.println("capacity : " + capacity.invoke(map));

Field size = mapType.getDeclaredField("size");
size.setAccessible(true);
System.out.println("size : " + size.get(map));

Field threshold = mapType.getDeclaredField("threshold");
threshold.setAccessible(true);
System.out.println("threshold : " + threshold.get(map));

Field loadFactor = mapType.getDeclaredField("loadFactor");
loadFactor.setAccessible(true);
System.out.println("loadFactor : " + loadFactor.get(map));

```

输出结果:


```
capacity : 16
size : 12
threshold : 12
loadFactor : 0.75

capacity : 32
size : 13
threshold : 24
loadFactor : 0.75
```

当 HashMap 中的元素个数达到 13 的时候, capacity 就从 16 扩容到 32 了。

HashMap 中还提供了一个支持传入 initialCapacity, loadFactor 两个参数的方法, 来初始化容量和装载因子。不过, 一般不建议修改 loadFactor 的值。

总之, HashMap 中 size 表示当前共有多少个 KV 对, capacity 表示当前 HashMap 的容量是多少, 默认值是 16, 每次扩容都是成倍的。loadFactor 是装载因子, 当 Map 中元素个数超过 `loadFactor * capacity` 的值时, 会触发扩容。`loadFactor * capacity` 可以用 threshold 表示。

PS: 文中分析基于 JDK1.8.0_73

在上面的内容我们说明了 HashMap 中和容量相关的几个概念, 简单介绍了一下 HashMap 的扩容机制。

文中我们提到, 默认情况下 HashMap 的容量是 16, 但是, 如果用户通过构造函数指定了一个数字作为容量, 那么 Hash 会选择**大于该数字的第一个 2 的幂**作为容量。(3->4、7->8、9->16)

接下来我们再来深入学习下, 到底应不应该设置 HashMap 的默认容量? 如果真的要设置 HashMap 的初始容量, 我们应该设置多少?

为什么要设置 HashMap 的初始化容量

我们之前提到过, 《Java 开发手册》中建议我们设置 HashMap 的初始化容量。

9. 【推荐】集合初始化时，指定集合初始值大小。

说明：HashMap 使用 `HashMap(int initialCapacity)` 初始化，

那么，为什么要这么建议？你有想过没有。

我们先来写一段代码在 JDK 1.7 (jdk1.7.0_79) 下面来分别测试下，在不指定初始化容量和指定初始化容量的情况下性能情况如何。(jdk 8 结果会有所不同，我会在后面的文章中分析)

```
public static void main(String[] args) {
    int aHundredMillion = 10000000;

    Map<Integer, Integer> map = new HashMap<>();

    long s1 = System.currentTimeMillis();
    for (int i = 0; i < aHundredMillion; i++) {
        map.put(i, i);
    }
    long s2 = System.currentTimeMillis();

    System.out.println("未初始化容量, 耗时 : " + (s2 - s1));

    Map<Integer, Integer> map1 = new HashMap<>(aHundredMillion / 2);

    long s5 = System.currentTimeMillis();
    for (int i = 0; i < aHundredMillion; i++) {
        map1.put(i, i);
    }
    long s6 = System.currentTimeMillis();

    System.out.println("初始化容量 5000000, 耗时 : " + (s6 - s5));

    Map<Integer, Integer> map2 = new HashMap<>(aHundredMillion);

    long s3 = System.currentTimeMillis();
    for (int i = 0; i < aHundredMillion; i++) {
        map2.put(i, i);
    }
    long s4 = System.currentTimeMillis();

    System.out.println("初始化容量为 10000000, 耗时 : " + (s4 - s3));
}
```

以上代码不难理解，我们创建了 3 个 HashMap，分别使用默认的容量（16）、使用元素个数的一半（5 千万）作为初始容量、使用元素个数（一亿）作为初始容量进行初始化。然后分别向其中 put 一亿个 KV。

输出结果：

```
未初始化容量，耗时：14419
初始化容量 5000000，耗时：11916
初始化容量为 100000000，耗时：7984
```

从结果中，我们可以知道，在已知 HashMap 中将要存放的 KV 个数的时候，设置一个合理的初始化容量可以有效的提高性能。

当然，以上结论也是有理论支撑的。我们在上文介绍过，HashMap 有扩容机制，就是当达到扩容条件时会进行扩容。HashMap 的扩容条件就是当 HashMap 中的元素个数（size）超过临界值（threshold）时就会自动扩容。在 HashMap 中，`threshold = loadFactor * capacity`。

所以，如果我们没有设置初始容量大小，随着元素的不断增加，HashMap 会发生多次扩容，而 HashMap 中的扩容机制决定了每次扩容都需要重建 hash 表，是非常影响性能的。

从上面的代码示例中，我们还发现，同样是设置初始化容量，设置的数值不同也会影响性能，那么当我们已知 HashMap 中即将存放的 KV 个数的时候，容量设置成多少为好呢？

HashMap 中容量的初始化

在之前，我们通过代码实例其实介绍过，默认情况下，当我们设置 HashMap 的初始化容量时，实际上 HashMap 会采用第一个大于该数值的 2 的幂作为初始化容量。

来看下面的例子

```
Map<String, String> map = new HashMap<String, String>(1);
map.put("hahaha", "hollischuang");

Class<?> mapType = map.getClass();
Method capacity = mapType.getDeclaredMethod("capacity");
capacity.setAccessible(true);
System.out.println("capacity : " + capacity.invoke(map));
```

初始化容量设置成 1 的时候，输出结果是 2。在 jdk1.8 中，如果我们传入的初始化容量为 1，实际上设置的结果也为 1，上面代码输出结果为 2 的原因是代码中 `map.put("hahaha", "hollischuang");` 导致了扩容，容量从 1 扩容到 2。

那么，话题再说回来，当我们通过 `HashMap(int initialCapacity)` 设置初始容量的时候，HashMap 并不一定会直接采用我们传入的数值，而是经过计算，得到一个新值，目的是提高 hash 的效率。(1→1、3→4、7→8、9→16)

在 Jdk 1.7 和 Jdk 1.8 中，HashMap 初始化这个容量的时机不同。jdk1.8 中，在调用 HashMap 的构造函数定义 HashMap 的时候，就会进行容量的设定。而在 Jdk 1.7 中，要等到第一次 put 操作时才进行这一操作。

不管是 Jdk 1.7 还是 Jdk 1.8，计算初始化容量的算法其实是如出一辙的，主要代码如下：

```
int n = cap - 1;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
```

上面的代码挺有意思的，一个简单的容量初始化，Java 的工程师也有很多考虑在里面。

上面的算法目的挺简单，就是：根据用户传入的容量值（代码中的 cap），通过计算，得到第一个比他大的 2 的幂并返回。

聪明的读者们，如果让你设计这个算法你准备如何计算？如果你想到二进制的
话，那就很简单了。举几个例子看一下：

5	0000 0000 0000 0101	19	0000 0000 0001 0011
7	0000 0000 0000 0111	31	0000 0000 0001 1111
8	0000 0000 0000 1000	32	0000 0000 0010 0000
HollisChuang			
9	0000 0000 0000 1001	37	0000 0000 0010 0101
15	0000 0000 0000 1111	63	0000 0000 0011 1111
16	0000 0000 0001 0000	64	0000 0000 0100 0000

请关注上面的几个例子中，蓝色字体部分的变化情况，或许你会发现些规律。
5->8、9->16、19->32、37->64 都是主要经过了两个阶段。

Step 1, 5->7

Step 2, 7->8

Step 1, 9->15

Step 2, 15->16

Step 1, 19->31

Step 2, 31->32

对应到以上代码中，Step1:

```
n |= n >>> 1;
n |= n >>> 2;
n |= n >>> 4;
n |= n >>> 8;
```

```
n |= n >>> 16;
```

对应到以上代码中，Step2:

```
return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
```

Step 2 比较简单，就是做一下极限值的判断，然后把 Step 1 得到的数值 +1。

Step 1 怎么理解呢？其实是对一个二进制数依次向右移位，然后与原值取或。其目的对于一个数字的二进制，从第一个不为 0 的位开始，把后面的所有位都设置成 1。

随便拿一个二进制数，套一遍上面的公式就发现其目的了：

```
1100 1100 1100 >>>1 = 0110 0110 0110
1100 1100 1100 | 0110 0110 0110 = 1110 1110 1110
1110 1110 1110 >>>2 = 0011 1011 1011
1110 1110 1110 | 0011 1011 1011 = 1111 1111 1111
1111 1111 1111 >>>4 = 1111 1111 1111
1111 1111 1111 | 1111 1111 1111 = 1111 1111 1111
```

通过几次无符号右移和按位或运算，我们把 1100 1100 1100 转换成了 1111 1111 1111，再把 1111 1111 1111 加 1，就得到了 1 0000 0000 0000，这就是大于 1100 1100 1100 的第一个 2 的幂。

好了，我们现在解释清楚了 Step 1 和 Step 2 的代码。就是可以把一个数转化成第一个比他自身大的 2 的幂。（可以开始佩服 Java 的工程师们了，使用无符号右移和按位或运算大大提升了效率。）

但是还有一种特殊情况套用以上公式不行，这些数字就是 2 的幂自身。如果数字 4 套用公式的话。得到的会是 8：

```
Step 1:
0100 >>>1 = 0010
0100 | 0010 = 0110
0110 >>>1 = 0011
```

```
0110 | 0011 = 0111
Step 2:
0111 + 0001 = 1000
```

为了解决这个问题，JDK 的工程师把所有用户传进来的数在进行计算之前先 -1，就是源码中的第一行：

```
int n = cap - 1;
```

至此，再来回过头看看这个设置初始容量的代码，目的是不是一目了然了：

```
int n = cap - 1;
n |= n >>> 1;
n |= n >>> 2;
n |= n >>> 4;
n |= n >>> 8;
n |= n >>> 16;
return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
```

总结

当我们想要在代码中创建一个 HashMap 的时候，如果我们已知这个 Map 中即将存放的元素个数，给 HashMap 设置初始容量可以在一定程度上提升效率。

但是，JDK 并不会直接拿用户传进来的数字当做默认容量，而是会进行一番运算，最终得到一个 2 的幂。原因在 ([全网把 Map 中的 hash\(\) 分析的最透彻的文章，别无二家。](#)) 介绍过，得到这个数字的算法其实是使用了使用无符号右移和按位或运算来提升效率。

Java 开发手册建议创建 HashMap 时设置初始化容量，但是多少合适呢？

集合是 Java 开发日常开发中经常会使用到的，而作为一种典型的 K-V 结构的数据结构，HashMap 对于 Java 开发者一定不陌生。

关于 HashMap，很多人都对他有一些基本的了解，比如他和 hashtable 之间的区别、他和 concurrentHashMap 之间的区别等。这些都是比较常见的，关于 HashMap 的一些知识点和面试题，想来大家一定了熟于心了，并且在开发中也能有效的应用上。

但是，作者在很多次 CodeReview 以及面试中发现，有一个比较关键的小细节经常被忽视，那就是 HashMap 创建的时候，要不要指定容量？如果要指定的话，多少是合适的？为什么？

要设置 HashMap 的初始化容量

在《[HashMap 中傻傻分不清楚的那些概念](#)》中我们曾经有过以下结论：

HashMap 有扩容机制，就是当达到扩容条件时会进行扩容。HashMap 的扩容条件就是当 HashMap 中的元素个数 (size) 超过临界值 (threshold) 时就会自动扩容。在 HashMap 中， $\text{threshold} = \text{loadFactor} * \text{capacity}$ 。

所以，如果我们没有设置初始容量大小，随着元素的不断增加，HashMap 会发生多次扩容，而 HashMap 中的扩容机制决定了每次扩容都需要重建 hash 表，是非常影响性能的。

所以，首先可以明确的是，我们建议开发者在创建 HashMap 的时候指定初始化

容量。并且《阿里巴巴开发手册》中也是这么建议的：

9. 【推荐】集合初始化时，指定集合初始值大小。

说明：HashMap 使用 `HashMap(int initialCapacity)` 初始化，

HashMap 初始化容量设置多少合适

那么，既然建议我们集合初始化的时候，要指定初始值大小，那么我们创建 HashMap 的时候，到底指定多少合适呢？

有些人会自然想到，我准备塞多少个元素我就设置成多少呗。比如我准备塞 7 个元素，那就 `new HashMap(7)`。

但是，这么做不仅不对，而且以上方式创建出来的 Map 的容量也不是 7。

因为，当我们使用 `HashMap(int initialCapacity)` 来初始化容量的时候，HashMap 并不会使用我们传进来的 `initialCapacity` 直接作为初始容量。

JDK 会默认帮我们计算一个相对合理的值当做初始容量。所谓合理值，其实是找到第一个比用户传入的值大的 2 的幂。

也就是说，当我们 `new HashMap(7)` 创建 HashMap 的时候，JDK 会通过计算，帮我们创建一个容量为 8 的 Map；当我们 `new HashMap(9)` 创建 HashMap 的时候，JDK 会通过计算，帮我们创建一个容量为 16 的 Map。

但是，这个值看似合理，实际上并不尽然。因为 HashMap 在根据用户传入的 `capacity` 计算得到的默认容量，并没有考虑到 `loadFactor` 这个因素，只是简单机械的计算出第一个大约这个数字的 2 的幂。

`loadFactor` 是负载因子，当 HashMap 中的元素个数 (`size`) 超过 `threshold = loadFactor * capacity` 时，就会进行扩容。

也就是说，如果我们设置的默认值是 7，经过 JDK 处理之后，HashMap 的容量会被设置成 8，但是，这个 HashMap 在元素个数达到 $8 \times 0.75 = 6$ 的时候就会进行一次扩容，这明显是我们不希望见到的。

那么，到底设置成什么值比较合理呢？

这里我们可以参考 JDK8 中 putAll 方法中的实现的，这个实现在 guava (21.0 版本) 也被采用。

这个值的计算方法就是：

```
return (int) ((float) expectedSize / 0.75F + 1.0F);
```

比如我们计划向 HashMap 中放入 7 个元素的时候，我们通过 $\text{expectedSize} / 0.75F + 1.0F$ 计算， $7 / 0.75 + 1 = 10$ ，10 经过 JDK 处理之后，会被设置成 16，这就大大的减少了扩容的几率。

当 HashMap 内部维护的哈希表的容量达到 75% 时 (默认情况下)，会触发 rehash，而 rehash 的过程是比较耗费时间的。所以初始化容量要设置成 $\text{expectedSize} / 0.75 + 1$ 的话，可以有效的减少冲突也可以减小误差。(大家结合这个公式，好好理解下这句话)

所以，我们可以认为，当我们明确知道 HashMap 中元素的个数的时候，把默认容量设置成 $\text{expectedSize} / 0.75F + 1.0F$ 是一个在性能上相对好的选择，但是，同时也会牺牲些内存。

这个算法在 guava 中有实现，开发的时候，可以直接通过 Maps 类创建一个 HashMap：

```
Map<String, String> map = Maps.newHashMapWithExpectedSize(7);
```

其代码实现如下：

```
public static <K, V> HashMap<K, V> newHashMapWithExpectedSize(int expectedSize) {
    return new HashMap(capacity(expectedSize));
}

static int capacity(int expectedSize) {
    if (expectedSize < 3) {
        CollectPreconditions.checkNotNull(expectedSize, "expectedSize");
        return expectedSize + 1;
    } else {
        return expectedSize < 1073741824 ? (int)((float)expectedSize / 0.75F
+ 1.0F) : 2147483647;
    }
}
```

但是，以上的操作是一种用内存换性能的做法，真正使用的时候，要考虑到内存的影响。但是，大多数情况下，我们还是认为内存是一种比较富裕的资源。

但是话又说回来了，有些时候，我们到底要不要设置 HashMap 的初识值，这个值又设置成多少，真的有那么大影响吗？其实也不见得！

可是，大的性能优化，不就是一个一个的优化细节堆叠出来的吗？

再不济，以后你写代码的时候，使用 `Maps.newHashMapWithExpectedSize(7);` 的写法，也可以让同事和老板眼前一亮。

或者哪一天你碰到一个面试官问你一些细节的时候，你也能有个印象，或者某一天你也可以拿这个出去面试问其他人～！啊哈哈哈。

为什么阿里巴巴禁止使用 Executors 创建线程池？

在《[深入源码分析 Java 线程池的实现原理](#)》这篇文章中，我们介绍过了 Java 中线程池的常见用法以及基本原理。

在文中有这样一段描述：

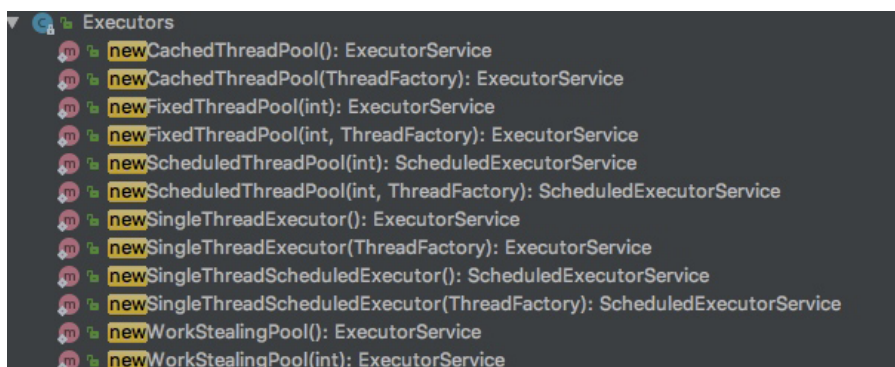
可以通过 Executors 静态工厂构建线程池，但一般不建议这样使用。

关于这个问题，在那篇文章中并没有深入的展开。作者之所以这么说，是因为这种创建线程池的方式有很大的隐患，稍有不慎就有可能导致线上故障，如：[一次 Java 线程池误用引发的血案和总结](#)。

本文我们就来围绕这个问题来分析一下为什么 JDK 自身提供的构建线程池的方式并不建议使用？到底应该如何创建一个线程池呢？

Executors

Executors 是一个 Java 中的工具类。提供工厂方法来创建不同类型的线程池。



从上图中也可以看出, Executors 的创建线程池的方法, 创建出来的线程池都实现了 ExecutorService 接口。常用方法有以下几个:

`newFixedThreadPool(int Threads)`: 创建固定数目线程的线程池。

`newCachedThreadPool()`: 创建一个可缓存的线程池, 调用 `execute` 将重用以前构造的线程 (如果线程可用)。如果没有可用的线程, 则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

`newSingleThreadExecutor()` 创建一个单线程化的 Executor。

`newScheduledThreadPool(int corePoolSize)` 创建一个支持定时及周期性的任务执行的线程池, 多数情况下可用来替代 Timer 类。

类看起来功能还是比较强大的, 又用到了工厂模式、又有比较强的扩展性, 重要的是用起来还比较方便, 如:

```
ExecutorService executor = Executors.newFixedThreadPool(nThreads) ;
```

即可创建一个固定大小的线程池。

但是为什么我说不建议大家使用这个类来创建线程池呢?

我提到的是「不建议」, 但是在阿里巴巴 Java 开发手册中也明确指出, 而且用的词是「不允许」使用 Executors 创建线程池。

4. 【强制】线程池不允许使用 Executors 去创建, 而是通过 ThreadPoolExecutor 的方式, 这样的处理方式让写的同学更加明确线程池的运行规则, 规避资源耗尽的风险。

说明: Executors 返回的线程池对象的弊端如下:

1) **FixedThreadPool** 和 **SingleThreadPool**:

允许的请求队列长度为 `Integer.MAX_VALUE`, 可能会堆积大量的请求, 从而导致 OOM。

2) **CachedThreadPool** 和 **ScheduledThreadPool**:

允许的创建线程数量为 `Integer.MAX_VALUE`, 可能会创建大量的线程, 从而导致 OOM。

Executors 存在什么问题

在阿里巴巴 Java 开发手册中提到, 使用 Executors 创建线程池可能会导致 OOM(OutOfMemory, 内存溢出), 但是并没有说明为什么, 那么接下来我们就来看一下到底为什么不允许使用 Executors?

我们先来一个简单的例子, 模拟一下使用 Executors 导致 OOM 的情况。

```
/**
 * @author Hollis
 */
public class ExecutorsDemo {
    private static ExecutorService executor = Executors.newFixedThreadPool(15);
    public static void main(String[] args) {
        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            executor.execute(new SubThread());
        }
    }
}

class SubThread implements Runnable {
    @Override
    public void run() {
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            //do nothing
        }
    }
}
```

通过指定 JVM 参数: `-Xmx8m -Xms8m` 运行以上代码, 会抛出 OOM:

```
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit
exceeded
    at java.util.concurrent.LinkedBlockingQueue.offer(LinkedBlockingQueue.
java:416)
    at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.
java:1371)
    at com.hollis.ExecutorsDemo.main(ExecutorsDemo.java:16)
```

以上代码指出, `ExecutorsDemo.java` 的第 16 行, 就是代码中的 `executor.execute(new SubThread());`。

Executors 为什么存在缺陷

通过上面的例子，我们知道了 Executors 创建的线程池存在 OOM 的风险，那么到底是什么原因导致的呢？我们需要深入 Executors 的源码来分析一下。

其实，在上面的报错信息中，我们是可以看出蛛丝马迹的，在以上的代码中其实已经说了，真正的导致 OOM 的其实是 `LinkedBlockingQueue.offer` 方法。

```
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
    at java.util.concurrent.LinkedBlockingQueue.offer(LinkedBlockingQueue.
java:416)
    at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.
java:1371)
    at com.hollis.ExecutorsDemo.main(ExecutorsDemo.java:16)
```

如果读者翻看代码的话，也可以发现，其实底层确实是通过 `LinkedBlockingQueue` 实现的：

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

如果读者对 Java 中的阻塞队列有所了解的话，看到这里或许就能够明白原因了。

Java 中的 `BlockingQueue` 主要有两种实现，分别是 `ArrayBlockingQueue` 和 `LinkedBlockingQueue`。

`ArrayBlockingQueue` 是一个用数组实现的有界阻塞队列，必须设置容量。

`LinkedBlockingQueue` 是一个用链表实现的有界阻塞队列，容量可以选择进行设置，不设置的话，将是一个无边界的阻塞队列，最大长度为 `Integer.MAX_VALUE`。

这里的问题就出在：不设置的话，将是一个无边界的阻塞队列，最大长度为

`Integer.MAX_VALUE`。也就是说，如果我们不设置 `LinkedBlockingQueue` 的容量的话，其默认容量将会是 `Integer.MAX_VALUE`。

而 `newFixedThreadPool` 中创建 `LinkedBlockingQueue` 时，并未指定容量。此时，`LinkedBlockingQueue` 就是一个无边界队列，对于一个无边界队列来说，是可以不断的向队列中加入任务的，这种情况下就有可能因为任务过多而导致内存溢出问题。

上面提到的问题主要体现在 `newFixedThreadPool` 和 `newSingleThreadExecutor` 两个工厂方法上，并不是说 `newCachedThreadPool` 和 `newScheduledThreadPool` 这两个方法就安全了，这两种方式创建的最大线程数可能是 `Integer.MAX_VALUE`，而创建这么多线程，必然就有可能导致 OOM。

创建线程池的正确姿势

避免使用 Executors 创建线程池，主要是避免使用其中的默认实现，那么我们可以自己直接调用 `ThreadPoolExecutor` 的构造函数来自己创建线程池。在创建的同时，给 `BlockQueue` 指定容量就可以了。

```
private static ExecutorService executor = new ThreadPoolExecutor(10, 10,
    60L, TimeUnit.SECONDS,
    new ArrayBlockingQueue(10));
```

这种情况下，一旦提交的线程数超过当前可用线程数时，就会抛出 `java.util.concurrent.RejectedExecutionException`，这是因为当前线程池使用的队列是有边界队列，队列已经满了便无法继续处理新的请求。但是异常 (Exception) 总比发生错误 (Error) 要好。

除了自己定义 `ThreadPoolExecutor` 外。还有其它方法。这个时候第一时间就应该想到开源类库，如 apache 和 guava 等。

作者推荐使用 guava 提供的 `ThreadFactoryBuilder` 来创建线程池。


```
public class ExecutorsDemo {

    private static ThreadFactory namedThreadFactory = new
    ThreadFactoryBuilder()
        .setNameFormat("demo-pool-%d").build();

    private static ExecutorService pool = new ThreadPoolExecutor(5, 200,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>(1024), namedThreadFactory, new
    ThreadPoolExecutor.
    AbortPolicy());

    public static void main(String[] args) {

        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            pool.execute(new SubThread());
        }
    }
}
```

通过上述方式创建线程时，不仅可以避免 OOM 的问题，还可以自定义线程名称，更加方便的出错的时候溯源。

思考题，文中作者说：发生异常 (Exception) 要比发生错误 (Error) 好，为什么这么说？

为什么阿里巴巴要求谨慎使用 ArrayList 中的 subList 方法？

集合是 Java 开发日常开发中经常会使用到的。

关于集合类，《阿里巴巴 Java 开发手册》中其实有一个规定：

2. 【强制】ArrayList 的 subList 结果不可强转成 ArrayList，否则会抛出 ClassCastException 异常，即 `java.util.RandomAccessSubList cannot be cast to java.util.ArrayList`。

说明：subList 返回的是 ArrayList 的内部类 SubList，并不是 ArrayList，而是 ArrayList 的一个视图，对于 SubList 子列表的所有操作最终会反映到原列表上。

本文就来分析一下为什么会有如此建议？其背后的原理是什么？

subList

subList 是 List 接口中定义的一个方法，该方法主要用于返回一个集合中的一段、可以理解为截取一个集合中的部分元素，他的返回值也是一个 List。

如以下代码：

```
public static void main(String[] args) {
    List<String> names = new ArrayList<String>() {{
        add("Hollis");
        add("hollischuang");
        add("H");
    }};

    List subList = names.subList(0, 1);
    System.out.println(subList);
}
```

以上代码输出结果为:

```
[Hollis]
```

如果我们改动下代码, 将 subList 的返回值强转成 ArrayList 试一下:

```
public static void main(String[] args) {  
    List<String> names = new ArrayList<String>() {{  
        add("Hollis");  
        add("hollischuang");  
        add("H");  
    }};  
  
    ArrayList subList = names.subList(0, 1);  
    System.out.println(subList);  
}
```

以上代码将抛出异常:

```
java.lang.ClassCastException: java.util.ArrayList$SubList cannot be cast to  
java.util.ArrayList
```

不只是强转成 ArrayList 会报错, 强转成 LinkedList、Vector 等 List 的实现类同样也都会报错。

那么, 为什么会发生这样的报错呢? 我们接下来深入分析一下。

底层原理

首先, 我们看下 subList 方法给我们返回的 List 到底是个什么东西, 这一点在 JDK 源码中注释是这样说的:

Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

也就是说 subList 返回是一个视图, 那么什么叫做视图呢?

我们看下 subList 的源码:

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

这个方法返回了一个 SubList，这个类是 ArrayList 中的一个内部类。

SubList 这个类中单独定义了 set、get、size、add、remove 等方法。

当我们调用 subList 方法的时候，会通过调用 SubList 的构造函数创建一个 SubList，那么看下这个构造函数做了哪些事情：

```
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;  
}
```

可以看到，这个构造函数中把原来的 List 以及该 List 中的部分属性直接赋值给自己的一些属性了。

也就是说，SubList 并没有重新创建一个 List，而是直接引用了原有的 List（返回了父类的视图），只是指定了一下他要使用的元素的范围而已（从 fromIndex（包含），到 toIndex（不包含））。

所以，为什么不能讲 subList 方法得到的集合直接转换成 ArrayList 呢？因为 SubList 只是 ArrayList 的内部类，他们之间并没有集成关系，故无法直接进行强制类型转换。

视图有什么问题

前面通过查看源码，我们知道，subList() 方法并没有重新创建一个 ArrayList，而是返回了一个 ArrayList 的内部类——SubList。

这个 SubList 是 ArrayList 的一个视图。

那么，这个视图又会带来什么问题呢？我们需要简单写几段代码看一下。

```
1、非结构性改变 SubList
2、public static void main(String[] args) {
3、    List<String> sourceList = new ArrayList<String>() {{
4、        add("H");
5、        add("O");
6、        add("L");
7、        add("L");
8、        add("I");
9、        add("S");
10、    }};
11、
12、    List subList = sourceList.subList(2, 5);
13、
14、    System.out.println("sourceList : " + sourceList);
15、    System.out.println("sourceList.subList(2, 5) 得到 List : ");
16、    System.out.println("subList : " + subList);
17、
18、    subList.set(1, "666");
19、
20、    System.out.println("subList.set(3,666) 得到 List : ");
21、    System.out.println("subList : " + subList);
22、    System.out.println("sourceList : " + sourceList);
23、
24、}
```

得到结果：

```
sourceList : [H, O, L, L, I, S]
sourceList.subList(2, 5) 得到 List :
subList : [L, L, I]
subList.set(3,666) 得到 List :
subList : [L, 666, I]
sourceList : [H, O, L, 666, I, S]
```

当我们尝试通过 set 方法，改变 subList 中某个元素的值得时候，我们发现，原来的那个 List 中对应元素的值也发生了改变。

同理，如果我们使用同样的方法，对 sourceList 中的某个元素进行修改，那么 subList 中对应的值也会发生改变。读者可以自行尝试一下。

```
1、结构性改变 SubList
2、public static void main(String[] args) {
3、    List<String> sourceList = new ArrayList<String>() {{
4、        add("H");
5、        add("O");
6、        add("L");
7、        add("L");
8、        add("I");
9、        add("S");
10、    }};
11、
12、    List subList = sourceList.subList(2, 5);
13、
14、    System.out.println("sourceList : " + sourceList);
15、    System.out.println("sourceList.subList(2, 5) 得到 List :");
16、    System.out.println("subList : " + subList);
17、
18、    subList.add("666");
19、
20、    System.out.println("subList.add(666) 得到 List :");
21、    System.out.println("subList : " + subList);
22、    System.out.println("sourceList : " + sourceList);
23、
24、}
```

得到结果:

```
sourceList : [H, O, L, L, I, S]
sourceList.subList(2, 5) 得到 List :
subList : [L, L, I]
subList.add(666) 得到 List :
subList : [L, L, I, 666]
sourceList : [H, O, L, L, I, 666, S]
```

我们尝试对 subList 的结构进行改变, 即向其追加元素, 那么得到的结果是 sourceList 的结构也同样发生了改变。

```
1、结构性改变原 List
2、public static void main(String[] args) {
3、    List<String> sourceList = new ArrayList<String>() {{
4、        add("H");
5、        add("O");
6、        add("L");
7、        add("L");
8、        add("I");
```

```
9、        add("S");
10、    }};
11、
12、    List subList = sourceList.subList(2, 5);
13、
14、    System.out.println("sourceList : " + sourceList);
15、    System.out.println("sourceList.subList(2, 5) 得到 List :");
16、    System.out.println("subList : " + subList);
17、
18、    sourceList.add("666");
19、
20、    System.out.println("sourceList.add(666) 得到 List :");
21、    System.out.println("sourceList : " + sourceList);
22、    System.out.println("subList : " + subList);
23、
24、 }
```

得到结果:

```
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.ArrayList$SubList.checkForComodification(ArrayList.java:1239)
    at java.util.ArrayList$SubList.listIterator(ArrayList.java:1099)
    at java.util.AbstractList.listIterator(AbstractList.java:299)
    at java.util.ArrayList$SubList.iterator(ArrayList.java:1095)
    at java.util.AbstractCollection.toString(AbstractCollection.java:454)
    at java.lang.String.valueOf(String.java:2994)
    at java.lang.StringBuilder.append(StringBuilder.java:131)
    at com.hollis.SubListTest.main(SubListTest.java:28)
```

我们尝试对 sourceList 的结构进行改变, 即向其追加元素, 结果发现抛出了 ConcurrentModificationException。关于这个异常, 我们在[《一不小心就踩坑的 fail-fast 是个什么鬼?》](#)中分析过, 这里原理相同, 就不再赘述了。

小结

我们简单总结一下, List 的 subList 方法并没有创建一个新的 List, 而是使用了原 List 的视图, 这个视图使用内部类 SubList 表示。

所以, 我们不能把 subList 方法返回的 List 强制转换成 ArrayList 等类, 因为他们之间没有继承关系。

另外，视图和原 List 的修改还需要注意几点，尤其是他们之间的相互影响：

1. 对父 (sourceList) 子 (subList)List 做的非结构性修改 (non-structural changes)，都会影响到彼此。
2. 对子 List 做结构性修改，操作同样会反映到父 List 上。
3. 对父 List 做结构性修改，会抛出异常 ConcurrentModificationException。

所以，阿里巴巴 Java 开发手册中有另外一条规定：

3. 【强制】在 subList 场景中，**高度注意**对原集合元素的增加或删除，均会导致子列表的遍历、增加、删除产生 ConcurrentModificationException 异常。

如何创建新的 List

如果需要对 subList 作出修改，又不想动原 list。那么可以创建 subList 的一个拷贝：

```
subList = Lists.newArrayList(subList);  
list.stream().skip(strart).limit(end).collect(Collectors.toList());
```

参考资料：

<https://www.jianshu.com/p/5854851240df>

<https://www.cnblogs.com/ljdblog/p/6251387.html>

为什么阿里巴巴不建议在 for 循环中使用 “+” 进行字符串拼接？

字符串，是 Java 中最常用的一个数据类型了。关于字符串的知识，作者已经发表过几篇文章介绍过很多，如：

[Java 7 源码学习系列（一）——String](#)

[该如何创建字符串，使用” “还是构造函数？](#)

[我终于搞清楚了和 String 有关的那点事儿](#)

[三张图彻底了解 Java 中字符串的不变性](#)

[为什么 Java 要把字符串设计成不可变的](#)

[三张图彻底了解 JDK 6 和 JDK 7 中 substring 的原理及区别](#)

[Java 中的 Switch 对整型、字符型、字符串型的具体实现细节](#)

本文，也是对于 Java 中字符串相关知识的一个补充，主要来介绍一下字符串拼接相关的知识。本文基于 jdk1.8.0_181。

字符串拼接

字符串拼接是我们在 Java 代码中比较经常要做的事情，就是把多个字符串拼接到一起。

我们都知道，**String 是 Java 中一个不可变的类**，所以他一旦被实例化就无法被修改。

不可变类的实例一旦创建，其成员变量的值就不能被修改。这样设计有很多好处，比如可以缓存 hashCode、使用更加便利以及更加安全等。

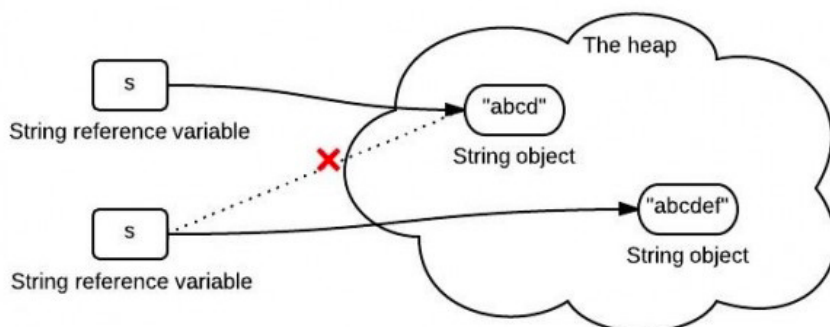
但是，既然字符串是不可变的，那么字符串拼接又是怎么回事呢？

字符串不变性与字符串拼接

其实，所有的所谓字符串拼接，都是重新生成了一个新的字符串。下面一段字符串拼接代码：

```
String s = "abcd";  
s = s.concat("ef");
```

其实最后我们得到的 s 已经是一个新的字符串了。如下图



s 中保存的是一个重新创建出来的 String 对象的引用。

那么，在 Java 中，到底如何进行字符串拼接呢？字符串拼接有很多种方式，这里简单介绍几种比较常用的。

使用 + 拼接字符串

在 Java 中，拼接字符串最简单的方式就是直接使用符号 + 来拼接。如：

```
String wechat = "Hollis";  
String introduce = " 每日更新 Java 相关文章 ";  
String hollis = wechat + "," + introduce;
```

这里要特别说明一点，有人把 Java 中使用 `+` 拼接字符串的功能理解为**运算符重载**。其实并不是，**Java 是不支持运算符重载的**。这其实只是 Java 提供的**一个语法糖**。后面再详细介绍。

运算符重载：在计算机程序设计中，运算符重载（英语：operator overloading）是多态的一种。运算符重载，就是对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型。

语法糖：语法糖（Syntactic sugar），也译为糖衣语法，是由英国计算机科学家彼得·兰丁发明的一个术语，指计算机语言中添加的某种语法，这种语法对语言的功能没有影响，但是更方便程序员使用。语法糖让程序更加简洁，有更高的可读性。

Concat

除了使用 `+` 拼接字符串之外，还可以使用 `String` 类中的方法 `concat` 方法来拼接字符串。如：

```
String wechat = "Hollis";  
String introduce = " 每日更新 Java 相关文章 ";  
String hollis = wechat.concat(",").concat(introduce);
```

StringBuffer

关于字符串，Java 中除了定义了一个可以用来定义**字符串常量**的 `String` 类以外，还提供了可以用来定义**字符串变量**的 `StringBuffer` 类，它的对象是可以扩充和修改的。

使用 `StringBuffer` 可以方便的对字符串进行拼接。如：

```
StringBuffer wechat = new StringBuffer("Hollis");  
String introduce = " 每日更新 Java 相关技术文章 ";  
StringBuffer hollis = wechat.append(", ").append(introduce);
```

StringBuilder

除了 `StringBuffer` 以外，还有一个类 `StringBuilder` 也可以使用，其用法和 `StringBuffer` 类似。如：

```
StringBuilder wechat = new StringBuilder("Hollis");  
String introduce = " 每日更新 Java 相关技术文章 ";  
StringBuilder hollis = wechat.append(", ").append(introduce);
```

StringUtils.join

除了 JDK 中内置的字符串拼接方法，还可以使用一些开源类库中提供的字符串拼接方法名，如 `apache.commons` 中提供的 `StringUtils` 类，其中的 `join` 方法可以拼接字符串。

```
String wechat = "Hollis";  
String introduce = " 每日更新 Java 相关技术文章 ";  
System.out.println(StringUtils.join(wechat, ",", introduce));
```

这里简单说一下，`StringUtils` 中提供的 `join` 方法，最主要的功能是：将数组或集合以某拼接符拼接到一起形成新的字符串，如：

```
String []list ={"Hollis"," 每日更新 Java 相关技术文章 "};  
String result= StringUtils.join(list,",");  
System.out.println(result);  
// 结果: Hollis, 每日更新 Java 相关技术文章
```

并且，Java8 中的 `String` 类中也提供了一个静态的 `join` 方法，用法和 `StringUtils.join` 类似。

以上就是比较常用的五种在 Java 中拼接字符串的方式，那么到底哪种更好用呢？为什么阿里巴巴 Java 开发手册中不建议在循环体中使用 “+” 进行字符串拼接呢？

17. 【推荐】循环体内，字符串的连接方式，使用 `StringBuilder` 的 `append` 方法进行扩展。

说明：反编译出的字节码文件显示每次循环都会 `new` 出一个 `StringBuilder` 对象，然后进行 `append` 操作，最后通过 `toString` 方法返回 `String` 对象，造成内存资源浪费。

反例：

```
String str = "start";
for (int i = 0; i < 100; i++) {
    str = str + "hello";
}
```

(阿里巴巴 Java 开发手册中关于字符串拼接的规约)

使用 + 拼接字符串的实现原理

前面提到过，使用 `+` 拼接字符串，其实只是 Java 提供的一个语法糖，那么，我们就来解一解这个语法糖，看看他的内部原理到底是如何实现的。

还是这样一段代码。我们把他生成的字节码进行反编译，看看结果。

```
String wechat = "Hollis";
String introduce = " 每日更新 Java 相关文章 ";
String hollis = wechat + "," + introduce;
```

反编译后的内容如下，反编译工具为 `jad`。

```
String wechat = "Hollis";
String introduce = "\u6BCF\u65E5\u66F4\u65B0Java\u76F8\u5173\u6280\u672F\u6587\u7AE0"; // 每日更新 Java 相关文章
String hollis = (new StringBuilder()).append(wechat).append(",").append(introduce).toString();
```

通过查看反编译以后的代码，我们可以发现，原来字符串常量在拼接过程中，是将 `String` 转成了 `StringBuilder` 后，使用其 `append` 方法进行处理。

那么也就是说，Java 中的 `+` 对字符串的拼接，其实现原理是使用 `StringBuilder.append`。

concat 是如何实现的

我们再来看一下 concat 方法的源代码，看一下这个方法又是如何实现的。

```
public String concat(String str) {
    int otherLen = str.length();
    if (otherLen == 0) {
        return this;
    }
    int len = value.length;
    char buf[] = Arrays.copyOf(value, len + otherLen);
    str.getChars(buf, len);
    return new String(buf, true);
}
```

这段代码首先创建了一个字符数组，长度是已有字符串和待拼接字符串的长度之和，再把两个字符串的值复制到新的字符数组中，并使用这个字符数组创建一个新的 String 对象并返回。

通过源码我们也可以看到，经过 concat 方法，其实是 new 了一个新的 String，这也就呼应到前面我们说的字符串的不变性问题上。

StringBuffer 和 StringBuilder

接下来我们看看 StringBuffer 和 StringBuilder 的实现原理。

和 String 类类似，StringBuilder 类也封装了一个字符数组，定义如下：

```
char[] value;
```

与 String 不同的是，它并不是 final 的，所以他是可以修改的。另外，与 String 不同，字符数组中不一定所有位置都被使用，它有一个实例变量，表示数组中已经使用的字符个数，定义如下：

```
int count;
```

其 append 源码如下：

```
public StringBuilder append(String str) {  
    super.append(str);  
    return this;  
}
```

该类继承了 `AbstractStringBuilder` 类，看下其 `append` 方法：

```
public AbstractStringBuilder append(String str) {  
    if (str == null)  
        return appendNull();  
    int len = str.length();  
    ensureCapacityInternal(count + len);  
    str.getChars(0, len, value, count);  
    count += len;  
    return this;  
}
```

`append` 会直接拷贝字符到内部的字符数组中，如果字符数组长度不够，会进行扩展。

`StringBuffer` 和 `StringBuilder` 类似，最大的区别就是 `StringBuffer` 是线程安全的，看一下 `StringBuffer` 的 `append` 方法。

```
public synchronized StringBuffer append(String str) {  
    toStringCache = null;  
    super.append(str);  
    return this;  
}
```

该方法使用 `synchronized` 进行声明，说明是一个线程安全的方法。而 `StringBuilder` 则不是线程安全的。

StringUtils.join 是如何实现的

通过查看 `StringUtils.join` 的源代码，我们可以发现，其实他也是通过 `StringBuilder` 来实现的。

```
public static String join(final Object[] array, String separator, final int  
    startIndex, final int endIndex) {
```

```
    if (array == null) {
        return null;
    }
    if (separator == null) {
        separator = EMPTY;
    }

    // endIndex - startIndex > 0:  Len = NofStrings *(len(firstString) +
    len(separator))
    //          (Assuming that all Strings are roughly equally long)
    final int noOfItems = endIndex - startIndex;
    if (noOfItems <= 0) {
        return EMPTY;
    }

    final StringBuilder buf = new StringBuilder(noOfItems * 16);

    for (int i = startIndex; i < endIndex; i++) {
        if (i > startIndex) {
            buf.append(separator);
        }
        if (array[i] != null) {
            buf.append(array[i]);
        }
    }
    return buf.toString();
}
```

效率比较

既然有这么多种字符串拼接的方法，那么到底哪一种效率最高呢？我们来简单对比一下。

```
long t1 = System.currentTimeMillis();
// 这里是初始字符串定义
for (int i = 0; i < 50000; i++) {
    // 这里是字符串拼接代码
}
long t2 = System.currentTimeMillis();
System.out.println("cost:" + (t2 - t1));
```

我们使用形如以上形式的代码，分别测试下五种字符串拼接代码的运行时间。得到结果如下：


```
+ cost:5119
StringBuilder cost:3
StringBuffer cost:4
concat cost:3623
StringUtils.join cost:25726
```

从结果可以看出，用时从短到长的对比是：

`StringBuilder` < `StringBuffer` < `concat` < `+` < `StringUtils.join`

`StringBuffer` 在 `StringBuilder` 的基础上，做了同步处理，所以在耗时上会相对多一些。

`StringUtils.join` 也是使用了 `StringBuilder`，并且其中还是有很多其他操作，所以耗时较长，这个也容易理解。其实 `StringUtils.join` 更擅长处理字符串数组或者列表的拼接。

那么问题来了，前面我们分析过，其实使用 `+` 拼接字符串的实现原理也是使用的 `StringBuilder`，那为什么结果相差这么多，高达 1000 多倍呢？

我们再把以下代码反编译下：

```
long t1 = System.currentTimeMillis();
String str = "hollis";
for (int i = 0; i < 50000; i++) {
    String s = String.valueOf(i);
    str += s;
}
long t2 = System.currentTimeMillis();
System.out.println("+ cost:" + (t2 - t1));
```

反编译后代码如下：

```
long t1 = System.currentTimeMillis();
String str = "hollis";
for(int i = 0; i < 50000; i++)
{
    String s = String.valueOf(i);
    str = (new StringBuilder()).append(str).append(s).toString();
}
```

```
long t2 = System.currentTimeMillis();
System.out.println((new StringBuilder()).append("+ cost:").append(t2 - t1).
toString());
```

我们可以看到，反编译后的代码，在 `for` 循环中，每次都是 `new` 了一个 `StringBuilder`，然后再把 `String` 转成 `StringBuilder`，再进行 `append`。

而频繁的新建对象当然要耗费很多时间了，不仅仅会耗费时间，频繁的创建对象，还会造成内存资源的浪费。

所以，阿里巴巴 Java 开发手册建议：循环体内，字符串的连接方式，使用 `StringBuilder` 的 `append` 方法进行扩展。而不要使用 `+`。

总结

本文介绍了什么是字符串拼接，虽然字符串是不可变的，但是还是可以通过新建字符串的方式来进行字符串的拼接。

常用的字符串拼接方式有五种，分别是使用 `+`、使用 `concat`、使用 `StringBuilder`、使用 `StringBuffer` 以及使用 `StringUtils.join`。

由于字符串拼接过程中会创建新的对象，所以如果要在一个循环体中进行字符串拼接，就要考虑内存问题和效率问题。

因此，经过对比，我们发现，直接使用 `StringBuilder` 的方式是效率最高的。因为 `StringBuilder` 天生就是设计来定义可变字符串和字符串的变化操作的。

但是，还要强调的是：

1. 如果不是在循环体中进行字符串拼接的话，直接使用 `+` 就好了。
2. 如果在并发场景中进行字符串拼接的话，要使用 `StringBuffer` 来代替 `StringBuilder`。

为什么阿里巴巴禁止在 foreach 循环里进行元素的 remove/add 操作？

在阿里巴巴 Java 开发手册中，有这样一条规定：

7. 【强制】不要在 foreach 循环里进行元素的 remove/add 操作。remove 元素请使用 Iterator 方式，如果并发操作，需要对 Iterator 对象加锁。

正例：

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String item = iterator.next();
    if (删除元素的条件) {
        iterator.remove();
    }
}
```

反例：

```
List<String> a = new ArrayList<String>();
list.add("1");
list.add("2");
for (String item : list) {
    if ("1".equals(item)) {
        list.remove(item);
    }
}
```

说明：以上代码的执行结果肯定会出乎大家的意料，那么试一下把“1”换成“2”，会是同样的结果吗？

但是手册中并没有给出具体原因，本文就来深入分析一下该规定背后的思考。

foreach 循环

Foreach 循环 (Foreach loop) 是计算机编程语言中的一种控制流程语句，通常用来循环遍历数组或集合中的元素。

Java 语言从 JDK 1.5.0 开始引入 foreach 循环。在遍历数组、集合方面，

foreach 为开发人员提供了极大的方便。

foreach 语法格式如下：

```
for( 元素类型 t 元素变量 x : 遍历对象 obj){  
    引用了 x 的 java 语句 ;  
}
```

以下实例演示了 普通 for 循环 和 foreach 循环使用：

```
public static void main(String[] args) {  
    // 使用 ImmutableList 初始化一个 List  
    List<String> userNames = ImmutableList.of("Hollis", "hollis",  
        "HollisChuang", "H");  
  
    System.out.println(" 使用 for 循环遍历 List");  
    for (int i = 0; i < userNames.size(); i++) {  
        System.out.println(userNames.get(i));  
    }  
  
    System.out.println(" 使用 foreach 遍历 List");  
    for (String userName : userNames) {  
        System.out.println(userName);  
    }  
}
```

以上代码运行输出结果为：

```
使用 for 循环遍历 List  
Hollis  
hollis  
HollisChuang  
H  
使用 foreach 遍历 List  
Hollis  
hollis  
HollisChuang  
H
```

可以看到，使用 foreach 语法遍历集合或者数组的时候，可以起到和普通 for 循环同样的效果，并且代码更加简洁。所以，foreach 循环也通常也被称为增强 for 循环。

但是，作为一个合格的程序员，我们不仅要知道什么是增强 for 循环，还需要知道增强 for 循环的原理是什么？

其实，增强 for 循环也是 Java 给我们提供的一个语法糖，如果将以上代码编译后的 class 文件进行反编译（使用 jad 工具）的话，可以得到以下代码：

```
Iterator iterator = userNames.iterator();
do
{
    if(!iterator.hasNext())
        break;
    String userName = (String)iterator.next();
    if(userName.equals("Hollis"))
        userNames.remove(userName);
} while(true);
System.out.println(userNames);
```

可以发现，原本的增强 for 循环，其实是依赖了 while 循环和 Iterator 实现的。（请记住这种实现方式，后面会用到！）

问题重现

规范中指出不让我们在 foreach 循环中对集合元素做 add/remove 操作，那么，我们尝试着做一下看看会发生什么问题。

```
// 使用双括弧语法 (double-brace syntax) 建立并初始化一个 List
List<String> userNames = new ArrayList<String>() {{
    add("Hollis");
    add("hollis");
    add("HollisChuang");
    add("H");
}};

for (int i = 0; i < userNames.size(); i++) {
    if (userNames.get(i).equals("Hollis")) {
        userNames.remove(i);
    }
}

System.out.println(userNames);
```

以上代码，首先使用双括弧语法 (double-brace syntax) 建立并初始化一个 List，其中包含四个字符串，分别是 Hollis、hollis、HollisChuang 和 H。

然后使用普通 for 循环对 List 进行遍历，删除 List 中元素内容等于 Hollis 的元素。然后输出 List，输出结果如下：

```
[hollis, HollisChuang, H]
```

以上是哪使用普通的 for 循环在遍历的同时进行删除，那么，我们再看下，如果使用增强 for 循环的话会发生什么：

```
List<String> userNames = new ArrayList<String>() {{
    add("Hollis");
    add("hollis");
    add("HollisChuang");
    add("H");
}};

for (String userName : userNames) {
    if (userName.equals("Hollis")) {
        userNames.remove(userName);
    }
}

System.out.println(userNames);
```

以上代码，使用增强 for 循环遍历元素，并尝试删除其中的 Hollis 字符串元素。运行以上代码，会抛出以下异常：

```
java.util.ConcurrentModificationException
```

同样的，读者可以尝试下在增强 for 循环中使用 add 方法添加元素，结果也会同样抛出该异常。

之所以会出现这个异常，是因为触发了一个 Java 集合的错误检测机制——fail-fast。

fail-fast

接下来，我们就来分析下在增强 for 循环中 add/remove 元素的时候会抛出 `java.util.ConcurrentModificationException` 的原因，即解释下到底什么是 fail-fast 进制，fail-fast 的原理等。

fail-fast，即快速失败，它是 Java 集合的一种错误检测机制。当多个线程对集合（非 fail-safe 的集合类）进行结构上的改变的操作时，有可能会产生 fail-fast 机制，这个时候就会抛出 `ConcurrentModificationException`（当方法检测到对象的并发修改，但不允许这种修改时就抛出该异常）。

同时需要注意的是，即使不是多线程环境，如果单线程违反了规则，同样也有可能抛出改异常。

那么，在增强 for 循环进行元素删除，是如何违反了规则的呢？

要分析这个问题，我们先将增强 for 循环这个语法糖进行解糖，得到以下代码：

```
public static void main(String[] args) {
    // 使用 ImmutableList 初始化一个 List
    List<String> userNames = new ArrayList<String>() {{
        add("Hollis");
        add("hollis");
        add("HollisChuang");
        add("H");
    }};

    Iterator iterator = userNames.iterator();
    do
    {
        if(!iterator.hasNext())
            break;
        String userName = (String)iterator.next();
        if(userName.equals("Hollis"))
            userNames.remove(userName);
    } while(true);
    System.out.println(userNames);
}
```

然后运行以上代码，同样会抛出异常。我们来看一下 `ConcurrentModificationException` 的完整堆栈：

```
7 public class ForEachDemo {
8
9 public static void main(String[] args) {
10 // 使用ImmutableList初始化一个List
11 List<String> userNames = new ArrayList<String>() {{
12     add("Hollis");
13     add("hollis");
14     add("HollisChuang");
15     add("H");
16 }};
17
18 Iterator iterator = userNames.iterator();
19 do
20 {
21     if(!iterator.hasNext())
22         break;
23     String userName = (String)iterator.next();
24     if(userName.equals("Hollis"))
25         userNames.remove(userName);
26 } while(true);
27 System.out.println(userNames);
28 }
29 }
30
```

ForEachDemo > main()

```
ForEachDemo x
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)
    at java.util.ArrayList$Itr.next(ArrayList.java:859)
    at com.hollis.ForEachDemo.main(ForEachDemo.java:23)
```

通过异常堆栈我们可以到，异常发生的调用链 `ForEachDemo` 的第 23 行，`Iterator.next` 调用了 `Iterator.checkForComodification` 方法，而异常就是 `checkForComodification` 方法中抛出的。

其实，经过 debug 后，我们可以发现，如果 `remove` 代码没有被执行过，`iterator.next` 这一行是一直没报错的。抛异常的时机也正是 `remove` 执行之后的的那一次 `next` 方法的调用。

我们直接看下 `checkForComodification` 方法的代码，看下抛出异常的原因：

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


代码比较简单, `modCount != expectedModCount` 的时候, 就会抛出 `ConcurrentModificationException`。

那么, 就来看一下, remove/add 操作是如何导致 modCount 和 expectedModCount 不相等的吧。

remove/add 做了什么

首先, 我们要搞清楚的是, 到底 modCount 和 expectedModCount 这两个变量都是个什么东西。

通过翻源码, 我们可以发现:

- modCount 是 ArrayList 中的一个成员变量。它表示该集合实际被修改的次数。
- expectedModCount 是 ArrayList 中的一个内部类——Itr 中的成员变量。expectedModCount 表示这个迭代器期望该集合被修改的次数。其值是在 ArrayList.iterator 方法被调用的时候初始化的。只有通过迭代器对集合进行操作, 该值才会改变。
- Itr 是一个 Iterator 的实现, 使用 ArrayList.iterator 方法可以获取到的迭代器就是 Itr 类的实例。

他们之间的关系如下:

```
class ArrayList{
    private int modCount;
    public void add();
    public void remove();
    private class Itr implements Iterator<E> {
        int expectedModCount = modCount;
    }
    public Iterator<E> iterator() {
        return new Itr();
    }
}
```

其实，看到这里，大概很多人都能猜到为什么 remove/add 操作之后，会导致 expectedModCount 和 modCount 不想等了。

通过翻阅代码，我们也可以发现，remove 方法核心逻辑如下：

```
/*
 * Private remove method that skips bounds checking and does not
 * return the value removed.
 */
private void fastRemove(int index) {
    modCount++;
    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                           numMoved);
    elementData[--size] = null; // clear to let GC do its work
}
```

可以看到，它只修改了 modCount，并没有对 expectedModCount 做任何操作。

简单总结一下，之所以会抛出 ConcurrentModificationException 异常，是因为我们的代码中使用了增强 for 循环，而在增强 for 循环中，集合遍历是通过 iterator 进行的，但是元素的 add/remove 却是直接使用的集合类自己的方法。这就导致 iterator 在遍历的时候，会发现有一个元素在自己不知不觉的情况下就被删除 / 添加了，就会抛出一个异常，用来提示用户，可能发生了并发修改！

正确姿势

至此，我们介绍清楚了不能在 foreach 循环体中直接对集合进行 add/remove 操作的原因。

但是，很多时候，我们是有需求需要过滤集合的，比如删除其中一部分元素，那么应该如何做呢？有几种方法可供参考：

1. 直接使用普通 for 循环进行操作

我们说不能在 foreach 中进行，但是使用普通的 for 循环还是可以的，因为普通 for 循环并没有用到 Iterator 的遍历，所以压根就没有进行 fail-fast 的检验。

```
List<String> userNames = new ArrayList<String>() {{
    add("Hollis");
    add("hollis");
    add("HollisChuang");
    add("H");
}};

for (int i = 0; i < 1; i++) {
    if (userNames.get(i).equals("Hollis")) {
        userNames.remove(i);
    }
}

System.out.println(userNames);
```

这种方案其实存在一个问题，那就是 remove 操作会改变 List 中元素的下标，可能存在漏删的情况。

2. 直接使用 Iterator 进行操作

除了直接使用普通 for 循环以外，我们还可以直接使用 Iterator 提供的 remove 方法。

```
List<String> userNames = new ArrayList<String>() {{
    add("Hollis");
    add("hollis");
    add("HollisChuang");
    add("H");
}};

Iterator iterator = userNames.iterator();

while (iterator.hasNext()) {
    if (iterator.next().equals("Hollis")) {
        iterator.remove();
    }
}

System.out.println(userNames);
```

如果直接使用 Iterator 提供的 remove 方法, 那么就可以修改到 expected-ModCount 的值。那么就不会再抛出异常了。其实现代码如下:

```
public void remove() {
    if (lastRet < 0)
        throw new IllegalStateException();
    checkForComodification();

    try {
        ArrayList.this.remove(lastRet);
        cursor = lastRet;
        lastRet = -1;
        expectedModCount = modCount;
    } catch (IndexOutOfBoundsException ex) {
        throw new ConcurrentModificationException();
    }
}
```

3. 使用 Java 8 中提供的 filter 过滤

Java 8 中可以把集合转换成流, 对于流有一种 filter 操作, 可以对原始 Stream 进行某项测试, 通过测试的元素被留下来生成一个新 Stream。

```
List<String> userNames = new ArrayList<String>() {{
    add("Hollis");
    add("hollis");
    add("HollisChuang");
    add("H");
}};

userNames = userNames.stream().filter(userName -> !userName.
equals("Hollis")).collect(Collectors.toList());
System.out.println(userNames);
```

4. 使用增强 for 循环其实也可以

如果, 我们非常确定在一个集合中, 某个即将删除的元素只包含一个的话, 比如对 Set 进行操作, 那么其实也是可以使用增强 for 循环的, 只要在删除之后, 立刻结束循环体, 不要再继续进行遍历就可以了, 也就是说不让代码执行到下一次的 next 方法。

```
List<String> userNames = new ArrayList<String>() {{
    add("Hollis");
    add("hollis");
    add("HollisChuang");
    add("H");
}};

for (String userName : userNames) {
    if (userName.equals("Hollis")) {
        userNames.remove(userName);
        break;
    }
}
System.out.println(userNames);
```

5. 直接使用 fail-safe 的集合类

在 Java 中，除了一些普通的集合类以外，还有一些采用了 fail-safe 机制的集合类。这样的集合容器在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。

由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发 `ConcurrentModificationException`。

```
ConcurrentLinkedDeque<String> userNames = new ConcurrentLinkedDeque<String>() {{
    add("Hollis");
    add("hollis");
    add("HollisChuang");
    add("H");
}};

for (String userName : userNames) {
    if (userName.equals("Hollis")) {
        userNames.remove();
    }
}
```

基于拷贝内容的优点是避免了 `ConcurrentModificationException`，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。

java.util.concurrent 包下的容器都是安全失败，可以在多线程下并发使用，并发修改。

总结

我们使用的增强 for 循环，其实是 Java 提供的语法糖，其实现原理是借助 Iterator 进行元素的遍历。

但是如果在遍历过程中，不通过 Iterator，而是通过集合类自身的方法对集合进行添加 / 删除操作。那么在 Iterator 进行下一次的遍历时，经检测发现有一次集合的修改操作并未通过自身进行，那么可能是发生了并发被其他线程执行的，这时候就会抛出异常，来提示用户可能发生了并发修改，这就是所谓的 fail-fast 机制。

当然还是有很多种方法可以解决这类问题的。比如使用普通 for 循环、使用 Iterator 进行元素删除、使用 Stream 的 filter、使用 fail-safe 的类等。

为什么阿里巴巴禁止工程师直接使用日志系统 (Log4j、Logback) 中的 API ?

作为 Java 程序员，我想很多人都知道日志对于一个程序的重要性，尤其是 Web 应用。很多时候，日志可能是我们了解应用程序如何执行的唯一方式。

所以，日志在 Java Web 应用中至关重要，但是，很多人却以为日志输出只是一件简单的事情，所以会经常忽略和日志相关的问题。在接下来的几篇文章中，我会来介绍介绍这个容易被大家忽视，但同时也容易导致故障的知识点。

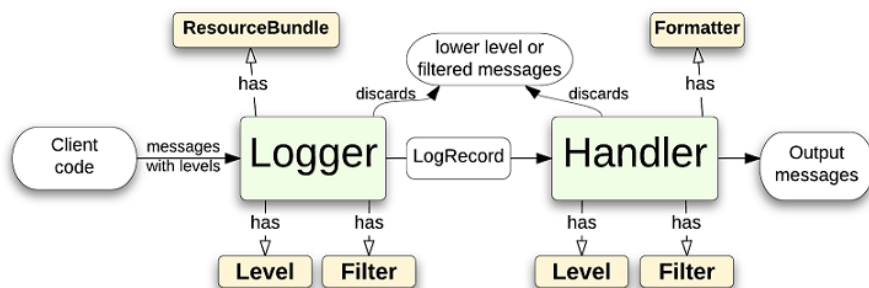
Java 语言之所以强大，就是因为他很成熟的生态体系。包括日志这一功能，就有很多成熟的开源框架可以被直接使用。

首先，我们先来看一下目前有哪些框架被广泛的使用。

常用日志框架

j.u.l

Java Util Logging Framework



j.u.l 是 java.util.logging 包的简称，是 JDK 在 1.4 版本中引入的 Java 原生日志框架。Java Logging API 提供了七个日志级别用来控制输出。这七个级别分别是：SEVERE、WARNING、INFO、CONFIG、FINE、FINER、FINEST。

Log4j



Log4j 是 Apache 的一个开源项目，通过使用 Log4j，我们可以控制日志信息输送的目的地是控制台、文件、GUI 组件，甚至是套接口服务器、NT 的事件记录器、UNIX Syslog 守护进程等；我们也可以控制每一条日志的输出格式；通过定义每一条日志信息的级别，我们能够更加细致地控制日志的生成过程。最令人感兴趣的就是，这些可以通过一个配置文件来灵活地进行配置，而不需要修改应用的代码。

Log4 也有七种日志级别：OFF、FATAL、ERROR、WARN、INFO、DEBUG 和 TRACE。

LogBack



LogBack 也是一个很成熟的日志框架，其实 LogBack 和 Log4j 出自一个人之手，这个人就是 Ceki Gülcü。

logback 当前分成三个模块：logback-core、logback-classic 和 logback-access。logback-core 是其它两个模块的基础模块。logback-classic 是 Log4j 的一个改良版本。此外 logback-classic 完整实现 SLF4J API 使你可以很方便地更换成其它日记系统如 Log4j 或 j.u.l。logback-access 访问模块与 Servlet 容器集成提供通过 Http 来访问日记的功能。

Log4j2

前面介绍过 Log4j，这里要单独介绍一下 Log4j2，之所以要单独拿出来说，而没有和 Log4j 放在一起介绍，是因为作者认为，Log4j2 已经不仅仅是 Log4j 的一个升级版了，而是从头到尾被重写的，这可以认为这其实就是完全不同的两个框架。

关于 Log4j2 解决了 Log4j 的哪些问题，Log4j2 相比较于 Log4j、j.u.l 和 logback 有哪些优势，我们在后续的文章中介绍。

前面介绍了四种日志框架，也就是说，我们想要应用中打印日志的时候，可以使用以上四种类库中的任意一种。比如想要使用 Log4j，那么只要依赖 Log4j 的 jar 包，配置好配置文件并且在代码中使用其 API 打印日志就可以了。

不知道有多少人看过《阿里巴巴 Java 开发手册》，其中有一条规范做了「强制」要求：

1. 【强制】应用中不可直接使用日志系统（Log4j、Logback）中的 API，而应依赖使用日志框架 SLF4J 中的 API，使用门面模式的日志框架，有利于维护和各个类的日志处理方式统一。

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
private static final Logger logger = LoggerFactory.getLogger(ABC.class);
```

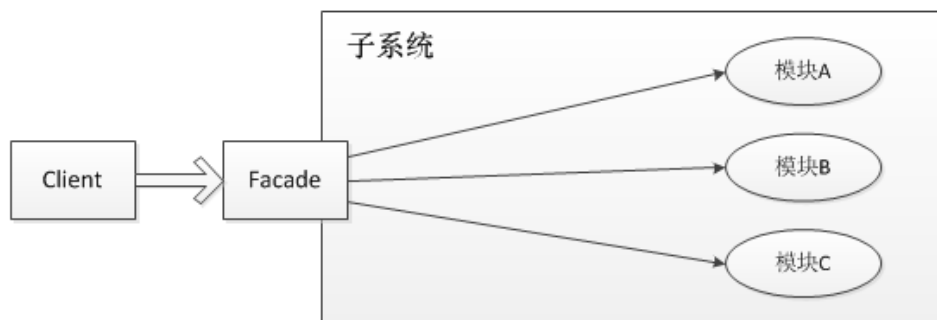
说好了以上四种常用的日志框架是给 Java 应用提供的方便进行记录日志的，那

为什么又不让在应用中直接使用其 API 呢? 这里面推崇使用的 SLF4J 是什么呢? 所谓的门面模式又是什么东西呢?

什么是日志门面

日志门面, 是门面模式的一个典型的应用。

门面模式 (Facade Pattern), 也称之为外观模式, 其核心为: 外部与一个子系统的通信必须通过一个统一的外观对象进行, 使得子系统更易于使用。



就像前面介绍的几种日志框架一样, 每一种日志框架都有自己单独的 API, 要使用对应的框架就要使用其对应的 API, 这就大大的增加应用程序代码对于日志框架的耦合性。

为了解决这个问题, 就是在日志框架和应用程序之间架设一个沟通的桥梁, 对于应用程序来说, 无论底层的日志框架如何变, 都不需要有任何感知。只要门面服务做的足够好, 随意换另外一个日志框架, 应用程序不需要修改任意一行代码, 就可以直接上线。

在软件开发领域有这样一句话: 计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决。而门面模式就是对于这句话的典型实践。

为什么需要日志门面

前面提到过一个重要的原因，就是为了在应用中屏蔽掉底层日志框架的具体实现。这样的话，即使有一天要更换代码的日志框架，只需要修改 jar 包，最多再改改日志输出相关的配置文件就可以了。这就是解除了应用和日志框架之间的耦合。

有人或许会问了，如果我换了日志框架了，应用是不需要改了，那日志门面不还是需要改的吗？

要回答这个问题，我们先来举一个例子，再把门面模式揉碎了重新解释一遍。

日志门面就像饭店的服务员，而日志框架就像是后厨的厨师。对于顾客这个应用来说，我到饭店点菜，我只需要告诉服务员我要一盘番茄炒蛋即可，我不关心后厨的所有事情。因为虽然主厨从把这道菜称之为『番茄炒蛋』A 厨师换成了把这道菜称之为『西红柿炒鸡蛋』的 B 厨师。但是，顾客不需要关心，他只要下达『番茄炒蛋』的命令给到服务员，由服务员再去翻译给厨师就可以了。

所以，对于一个了解了“番茄炒蛋的多种叫法”的服务员来说，无论后厨如何换厨师，他都能准确的帮用户下单。

同理，对于一个设计的全面、完善的日志门面来说，他也应该是天然就兼容了多种日志框架的。所以，底层框架的更换，日志门面几乎不需要改动。

以上，就是日志门面的一个比较重要的好处——解耦。

常用日志门面

介绍过了日志门面的概念和好处之后，我们看看 Java 生态体系中有哪些好的日志门面的实现可供选择。

SLF4J



Java 简易日志门面 (Simple Logging Facade for Java, 缩写 SLF4J), 是一套包装 Logging 框架的界面程式, 以外观模式实现。可以在软件部署的时候决定要使用的 Logging 框架, 目前主要支援的有 Java Logging API、Log4j 及 logback 等框架。以 MIT 授权方式发布。

SLF4J 的作者就是 Log4j 的作者 Ceki Gülcü, 他宣称 SLF4J 比 Log4j 更有效率, 而且比 Apache Commons Logging (JCL) 简单、稳定。

其实, SLF4J 其实只是一个门面服务而已, 他并不是真正的日志框架, 真正的日志的输出相关的实现还是要依赖 Log4j、logback 等日志框架的。

由于 SLF4J 比较常用, 这里多用一些篇幅, 再来简单分析一下 SLF4J, 主要和 Log4J 做一下对比。相比较于 Log4J 的 API, SLF4J 有以下几点优势:

- Log4j 提供 TRACE, DEBUG, INFO, WARN, ERROR 及 FATAL 六种纪录等级, 但是 SLF4J 认为 ERROR 与 FATAL 并没有实质上的差别, 所以拿掉了 FATAL 等级, 只剩下其他五种。
- 大部分人在程序里面会去写 `logger.error(exception)`, 其实这个时候 Log4j 会去把这个 exception 转成字符串。真正的写法应该是 `logger(message, exception)`; 而 SLF4J 就不会使得程序员犯这个错误。

- Log4j 间接的在鼓励程序员使用 string 相加的写法 (这种写法是有性能问题的), 而 SLF4J 就不会有这个问题, 你可以使用 `logger.error( "{ } is+serviceid", serviceid);`
- 使用 SLF4J 可以方便的使用其提供的各种集体的实现的 jar。(类似 `commons-logger`)
- 从 `commons-logger` 和 `Log4j merge` 非常方便, SLF4J 也提供了一个 `swing` 的 `tools` 来帮助大家完成这个 `merge`。
- 提供字串内容替换的功能, 会比较有效率, 说明如下:

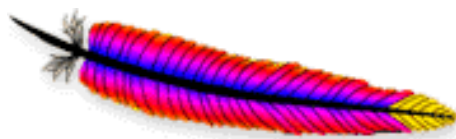
```
// 传统的字符串产生方式, 如果没有要记录 Debug 等级的信息, 就会浪费时间在产生不必要的信息上
logger.debug("There are now " + count + " user accounts: " + userAccountList);

// 为了避免上述问题, 我们可以先检查是不是开启了 Debug 信息记录功能, 只是程序的编码会比较复杂
if (logger.isDebugEnabled()) {
    logger.debug("There are now " + count + " user accounts: " + userAccountList);
}

// 如果 Debug 等级没有开启, 则不会产生不必要的字符串, 同时也能保持程序编码的简洁
logger.debug("There are now { } user accounts: { }", count, userAccountList);
```

- SLF4J 只支持 MDC, 不支持 NDC。

commons-logging



Apache CommonsTM
<http://commons.apache.org/>

Apache Commons Logging 是一个基于 Java 的日志记录实用程序, 是用于日志记录和其他工具包的编程模型。它通过其他一些工具提供 API, 日志实现和包装器实现。

commons-logging 和 SLF4J 的功能是类似的，主要是用来做日志 门面的。提供更加友好的 API 工具。

总结

在 Java 生态体系中，围绕着日志，有很多成熟的解决方案。关于日志输出，主要有两类工具。

一类是日志框架，主要用来进行日志的输出，比如输出到哪个文件，日志格式如何等。另外一类是日志门面，主要一套通用的 API，用来屏蔽各个日志框架之间的差异的。

所以，对于 Java 工程师来说，关于日志工具的使用，最佳实践就是在应用中使用如 Log4j + SLF4J 这样的组合来进行日志输出。

这样做的最大好处，就是业务层的开发不需要关心底层日志框架的实现及细节，在编码的时候也不需要考虑日后更换框架所带来的成本。这也是门面模式所带来的好处。

综上，请不要在你的 Java 代码中出现任何 Log4j 等日志框架的 API 的使用，而是应该直接使用 SLF4J 这种日志门面。

为什么阿里巴巴禁止把 SimpleDateFormat 定义成 static 变量？

在日常开发中，我们经常会用到时间，我们有很多办法在 Java 代码中获取时间。但是不同的方法获取到的时间的格式都不尽相同，这时候就需要一种格式化工具，把时间显示成我们需要的格式。

最常用的方法就是使用 SimpleDateFormat 类。这是一个看上去功能比较简单的类，但是，一旦使用不当也有可能导致很大的问题。

在阿里巴巴 Java 开发手册中，有如下明确规定：

5. 【强制】SimpleDateFormat 是线程不安全的类，一般不要定义为 static 变量，如果定义为 static，必须加锁，或者使用 DateUtils 工具类。

那么，本文就围绕 SimpleDateFormat 的用法、原理等来深入分析下如何以正确的姿势使用它。

SimpleDateFormat 用法

SimpleDateFormat 是 Java 提供的一个格式化和解析日期的工具类。它允许进行格式化（日期 -> 文本）、解析（文本 -> 日期）和规范化。SimpleDateFormat 使得可以选择任何用户定义的日期 - 时间格式的模式。

在 Java 中，可以使用 SimpleDateFormat 的 format 方法，将一个 Date 类型转化成 String 类型，并且可以指定输出格式。

```
// Date 转 String
```

```
Date data = new Date();
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
String dataStr = sdf.format(data);
System.out.println(dataStr);
```

以上代码，转换的结果是：2018-11-25 13:00:00，日期和时间格式由”日期和时间模式”字符串指定。如果你想要转换成其他格式，只要指定不同的时间模式就行了。

在 Java 中，可以使用 SimpleDateFormat 的 parse 方法，将一个 String 类型转化成 Date 类型。

```
// String 转 Data
System.out.println(sdf.parse(dataStr));
```

日期和时间模式表达方法

在使用 SimpleDateFormat 的时候，需要通过字母来描述时间元素，并组装成想要的日期和时间模式。常用的时间元素和字母的对应表如下：

字母	日期或时间元素	表示	示例
G	Era 标志符	Text	AD
y	年	Year	1996; 96
M	年中的月份	Month	July; Jul; 07
w	年中的周数	Number	27
W	月份中的周数	Number	2
D	年中的天数	Number	189
d	月份中的天数	Number	10
F	月份中的星期	Number	2
E	星期中的天数	Text	Tuesday; Tue
a	Am/pm 标记	Text	PM
H	一天中的小时数 (0-23)	Number	0
k	一天中的小时数 (1-24)	Number	24
K	am/pm 中的小时数 (0-11)	Number	0
h	am/pm 中的小时数 (1-12)	Number	12
m	小时中的分钟数	Number	30
s	分钟中的秒数	Number	55
S	毫秒数	Number	978
z	时区	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	时区	RFC 822 time zone	-0800

模式字母通常是重复的，其数量确定其精确表示。如下表是常用的输出格式的表示方法。

日期和时间模式	结果
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, 'yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o'clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700

输出不同时区的时间

时区是地球上的区域使用同一个时间定义。以前，人们通过观察太阳的位置（太阳）决定时间，这就使得不同经度的地方的时间有所不同（地方时）。1863 年，首次使用时区的概念。时区通过设立一个区域的标准时间部分地解决了这个问题。

世界各个国家位于地球不同位置上，因此不同国家，特别是东西跨度大的国家日出、日落时间必定有所偏差。这些偏差就是所谓的时差。

现今全球共分为 24 个时区。由于实用上常常 1 个国家，或 1 个省份同时跨着 2 个或更多时区，为了照顾到行政上的方便，常将 1 个国家或 1 个省份划在一起。所以时区并不严格按南北直线来划分，而是按自然条件来划分。例如，中国幅员宽广，差不多跨 5 个时区，但为了使用方便简单，实际上在只用东八时区的标准时即北京时间为准。

由于不同的时区的时间是不一样的，甚至同一个国家的不同城市时间都可能不一样，所以，在 Java 中想要获取时间的时候，要重点关注一下时区问题。

默认情况下，如果不指明，在创建日期的时候，会使用当前计算机所在的时区作

为默认时区，这也是为什么我们通过只要使用 `new Date()` 就可以获取中国的当前时间的原因。

那么，如何在 Java 代码中获取不同时区的时间呢？SimpleDateFormat 可以实现这个功能。

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
sdf.setTimeZone(TimeZone.getTimeZone("America/Los_Angeles"));
System.out.println(sdf.format(Calendar.getInstance().getTime()));
```

以上代码，转换的结果是：2018-11-24 21:00:00。既中国的时间是 11 月 25 日的 13 点，而美国洛杉矶时间比中国北京时间慢了 16 个小时（这还和冬夏令时有关系，就不详细展开了）。

如果你感兴趣，你还可以尝试打印一下美国纽约时间（America/New_York）。纽约时间是 2018-11-25 00:00:00。纽约时间比中国北京时间早了 13 个小时。

当然，这不是显示其他时区的唯一方法，不过本文主要为了介绍 SimpleDateFormat，其他方法暂不介绍了。

SimpleDateFormat 线程安全性

由于 SimpleDateFormat 比较常用，而且在一般情况下，一个应用中的时间显示模式都是一样的，所以很多人愿意使用如下方式定义 SimpleDateFormat：

```
public class Main {

    private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat(
        "yyyy-MM-dd HH:mm:ss");

    public static void main(String[] args) {
        simpleDateFormat.setTimeZone(TimeZone.getTimeZone("America/New_York"));
        System.out.println(simpleDateFormat.format(Calendar.getInstance().
            getTime()));
    }
}
```

这种定义方式，存在很大的安全隐患。

问题重现

我们来看一段代码，以下代码使用线程池来执行时间输出。

```
/** * @author Hollis */
public class Main {

    /**
     * 定义一个全局的 SimpleDateFormat
     */
    private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat
("yyyy-MM-dd HH:mm:ss");

    /**
     * 使用 ThreadFactoryBuilder 定义一个线程池
     */
    private static ThreadFactory namedThreadFactory = new ThreadFactoryBuilder()
        .setNameFormat("demo-pool-%d").build();

    private static ExecutorService pool = new ThreadPoolExecutor(5, 200,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>(1024), namedThreadFactory, new
ThreadPoolExecutor.
AbortPolicy());

    /**
     * 定义一个 CountDownLatch，保证所有子线程执行完之后主线程再执行
     */
    private static CountDownLatch countDownLatch = new CountDownLatch(100);

    public static void main(String[] args) {
        // 定义一个线程安全的 HashSet
        Set<String> dates = Collections.synchronizedSet(new HashSet<String>());
        for (int i = 0; i < 100; i++) {
            // 获取当前时间
            Calendar calendar = Calendar.getInstance();
            int finalI = i;
            pool.execute(() -> {
                // 时间增加
                calendar.add(Calendar.DATE, finalI);
                // 通过 simpleDateFormat 把时间转换成字符串
                String dateString = simpleDateFormat.format(calendar.
getTime());
                // 把字符串放入 Set 中
```

```

        dates.add(dateString);
        //countDown
        countDownLatch.countDown();
    });
}
// 阻塞, 直到 countDown 数量为 0
countDownLatch.await();
// 输出去重后的时间个数
System.out.println(dates.size());
}
}

```

以上代码, 其实比较简单, 很容易理解。就是循环一百次, 每次循环的时候都在当前时间基础上增加一个天数 (这个天数随着循环次数而变化), 然后把所有日期放入一个线程安全的、带有去重功能的 Set 中, 然后输出 Set 中元素个数。

上面的例子我特意写的稍微复杂了一些, 不过我几乎都加了注释。这里面涉及到了[线程池的创建](#)、[CountDownLatch](#)、lambda 表达式、线程安全的 HashSet 等知识。感兴趣的朋友可以逐一了解一下。

正常情况下, 以上代码输出结果应该是 100。但是实际执行结果是一个小于 100 的数字。

原因就是 SimpleDateFormat 作为一个非线程安全的类, 被当做了共享变量在多个线程中进行使用, 这就出现了线程安全问题。

在阿里巴巴 Java 开发手册的第一章第六节——并发处理中关于这一点也有明确说明:

5. 【强制】SimpleDateFormat 是线程不安全的类, 一般不要定义为 static 变量, 如果定义为 static, 必须加锁, 或者使用 DateUtils 工具类。

正例: 注意线程安全, 使用 DateUtils。亦推荐如下处理:

```

private static final ThreadLocal<DateFormat> df = new ThreadLocal<DateFormat>() {
    @Override
    protected DateFormat initialValue() {
        return new SimpleDateFormat("yyyy-MM-dd");
    }
};

```

那么，接下来我们就来看下到底是为什么，以及该如何解决。

线程不安全原因

通过以上代码，我们发现了在并发场景中使用 SimpleDateFormat 会有线程安全问题。其实，JDK 文档中已经明确表明了 SimpleDateFormat 不应该用在多线程场景中：

Date formats are not synchronized. It is recommended to create separate format instances for each thread. If multiple threads access a format concurrently, it must be synchronized externally.

那么接下来分析下为什么会出现这种问题，SimpleDateFormat 底层到底是怎么实现的？

我们跟一下 SimpleDateFormat 类中 format 方法的实现其实就能发现端倪。

```
@Override
public StringBuffer format(@NotNull Date date, @NotNull StringBuffer toAppendTo,
                           @NotNull FieldPosition pos)
{
    pos.beginIndex = pos.endIndex = 0;
    return format(date, toAppendTo, pos.getFieldDelegate());
}

// Called from Format after creating a FieldDelegate
private StringBuffer format(Date date, StringBuffer toAppendTo,
                             FieldDelegate delegate) {
    // Convert input date to time field list
    calendar.setTime(date);

    boolean useDateFormatSymbols = useDateFormatSymbols();

    for (int i = 0; i < compiledPattern.length; ) {
        int tag = compiledPattern[i] >> 8;
        int count = compiledPattern[i++] & 0xff;
        if (count == 255) {
            count = compiledPattern[i++] << 16;
            count |= compiledPattern[i++];
        }

        switch (tag) {
            case TAG_QUOTE_ASCII_CHAR:
                toAppendTo.append((char)count);
                break;

            case TAG_QUOTE_CHARS:
                toAppendTo.append(compiledPattern, i, count);
                i += count;
                break;

            default:
                subFormat(tag, count, delegate, toAppendTo, useDateFormatSymbols);
                break;
        }
    }
    return toAppendTo;
}
```

SimpleDateFormat 中的 format 方法在执行过程中, 会使用一个成员变量 calendar 来保存时间。这其实就是问题的关键。

由于我们在声明 SimpleDateFormat 的时候, 使用的是 static 定义的。那么这个 SimpleDateFormat 就是一个共享变量, 随之, SimpleDateFormat 中的 calendar 也就可以被多个线程访问到。

假设线程 1 刚刚执行完 `calendar.setTime` 把时间设置成 2018-11-11, 还没等执行完, 线程 2 又执行了 `calendar.setTime` 把时间改成了 2018-12-12。这时候线程 1 继续往下执行, 拿到的 `calendar.getTime` 得到的时间就是线程 2 改过之后的。

除了 format 方法以外, SimpleDateFormat 的 parse 方法也有同样的问题。

所以, 不要把 SimpleDateFormat 作为一个共享变量使用。

如何解决

前面介绍过了 SimpleDateFormat 存在的问题以及问题存在的原因, 那么有什么办法解决这种问题呢?

解决方法有很多, 这里介绍三个比较常用的方法。

使用局部变量

```
for (int i = 0; i < 100; i++) {  
    // 获取当前时间  
    Calendar calendar = Calendar.getInstance();  
    int finalI = i;  
    pool.execute(() -> {  
        // SimpleDateFormat 声明成局部变量  
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd  
HH:mm:ss");  
        // 时间增加  
        calendar.add(Calendar.DATE, finalI);  
        // 通过 simpleDateFormat 把时间转换成字符串
```

```
String dateString = simpleDateFormat.format(calendar.getTime());
// 把字符串放入 Set 中
dates.add(dateString);
//countDown
countDownLatch.countDown();
});
}
```

SimpleDateFormat 变成了局部变量，就不会被多个线程同时访问到了，就避免了线程安全问题。

加同步锁

除了改成局部变量以外，还有一种方法大家可能比较熟悉的，就是对于共享变量进行加锁。

```
for (int i = 0; i < 100; i++) {
    // 获取当前时间
    Calendar calendar = Calendar.getInstance();
    int finalI = i;
    pool.execute(() -> {
        // 加锁
        synchronized (simpleDateFormat) {
            // 时间增加
            calendar.add(Calendar.DATE, finalI);
            // 通过 simpleDateFormat 把时间转换成字符串
            String dateString = simpleDateFormat.format(calendar.getTime());
            // 把字符串放入 Set 中
            dates.add(dateString);
            //countDown
            countDownLatch.countDown();
        }
    });
}
```

通过加锁，使多个线程排队顺序执行。避免了并发导致的线程安全问题。

其实以上代码还有可以改进的地方，就是可以把锁的粒度再设置的小一点，可以对 `simpleDateFormat.format` 这一行加锁，这样效率更高一些。

使用 ThreadLocal

第三种方式, 就是使用 ThreadLocal。ThreadLocal 可以确保每个线程都可以得到单独的一个 SimpleDateFormat 的对象, 那么自然也就不存在竞争问题了。

```
/**
 * 使用 ThreadLocal 定义一个全局的 SimpleDateFormat
 */
private static ThreadLocal<SimpleDateFormat> simpleDateFormatThreadLocal = new
ThreadLocal<SimpleDateFormat>() {
    @Override
    protected SimpleDateFormat initialValue() {
        return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    }
};

// 用法
String dateString = simpleDateFormatThreadLocal.get().format(calendar.getTime());
```

用 ThreadLocal 来实现其实是有点类似于缓存的思路, 每个线程都有一个独有的对象, 避免了频繁创建对象, 也避免了多线程的竞争。

当然, 以上代码也有改进空间, 就是, 其实 SimpleDateFormat 的创建过程可以改为延迟加载。这里就不详细介绍了。

使用 DateTimeFormatter

如果是 Java8 应用, 可以使用 DateTimeFormatter 代替 SimpleDateFormat, 这是一个线程安全的格式化工具类。就像官方文档中说的, 这个类 simple beautiful strong immutable thread-safe。

```
// 解析日期
String dateStr= "2016 年 10 月 25 日 ";
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy 年 MM 月 dd 日 ");
LocalDate date= LocalDate.parse(dateStr, formatter);

// 日期转换为字符串
LocalDateTime now = LocalDateTime.now();
DateTimeFormatter format = DateTimeFormatter.ofPattern("yyyy 年 MM 月 dd 日 hh:mm a");
String nowStr = now .format(format);
```



```
System.out.println(nowStr);
```

总结

本文介绍了 SimpleDateFormat 的用法，SimpleDateFormat 主要可以在 String 和 Date 之间做转换，还可以将时间转换成不同时区输出。同时提到在并发场景中 SimpleDateFormat 是不能保证线程安全的，需要开发者自己来保证其安全性。

主要的几个手段有改为局部变量、使用 synchronized 加锁、使用 Threadlocal 为每一个线程单独创建一个等。

希望通过此文，你可以在使用 SimpleDateFormat 的时候更加得心应手。

为什么阿里巴巴禁止开发人员使用 isSuccess 作为变量名？

在日常开发中，我们会经常要在类中定义布尔类型的变量，比如在给外部系统提供一个 RPC 接口的时候，我们一般会定义一个字段表示本次请求是否成功的。

关于这个”本次请求是否成功”的字段定义，其实是有很多种讲究和坑的，稍有不慎就会掉入坑里，作者在很久之前就遇到过类似的问题，本文就来围绕这个简单分析一下。到底该如何定一个布尔类型的成员变量。

一般情况下，我们可以有以下四种方式来定义一个布尔类型的成员变量：

```
boolean success  
boolean isSuccess  
Boolean success  
Boolean isSuccess
```

以上四种定义形式，你日常开发中最常用的是哪种呢？到底哪一种才是正确的使用姿势呢？

通过观察我们可以发现，前两种和后两种的主要区别是变量的类型不同，前者使用的是 boolean，后者使用的是 Boolean。

另外，第一种和第三种在定义变量的时候，变量命名是 success，而另外两种使用 isSuccess 来命名的。

首先，我们来分析一下，到底应该用 success 来命名，还是使用 isSuccess 更好一点。

success 还是 isSuccess

到底应该用 success 还是 isSuccess 来给变量命名呢？从语义上面来讲，两种命名方式都可以讲的通，并且也都没有歧义。那么还有什么原则可以参考来让我们做选择呢。

在阿里巴巴 Java 开发手册中关于这一点，有过一个『强制性』规定：

8. 【强制】POJO 类中布尔类型的变量，都不要加 is，否则部分框架解析会引起序列化错误。
反例：定义为基本数据类型 boolean isSuccess；的属性，它的方法也是 isSuccess()，RPC 框架在反向解析的时候，“以为”对应的属性名称是 success，导致属性获取不到，进而抛出异常。

那么，为什么会有这样的规定呢？我们看一下 POJO 中布尔类型变量不同的命名有什么区别吧。

```
class Model1 {
    private Boolean isSuccess;
    public void setSuccess(Boolean success) {
        isSuccess = success;
    }
    public Boolean getSuccess() {
        return isSuccess;
    }
}

class Model2 {
    private Boolean success;
    public Boolean getSuccess() {
        return success;
    }
    public void setSuccess(Boolean success) {
        this.success = success;
    }
}

class Model3 {
    private boolean isSuccess;
    public boolean isSuccess() {
        return isSuccess;
    }
    public void setSuccess(boolean success) {
```

```
        isSuccess = success;
    }
}

class Model4 {
    private boolean success;
    public boolean isSuccess() {
        return success;
    }
    public void setSuccess(boolean success) {
        this.success = success;
    }
}
```

以上代码的 setter/getter 是使用 IntelliJ IDEA 自动生成的，仔细观察以上代码，你会发现以下规律：

- 基本类型自动生成的 getter 和 setter 方法，名称都是 `isXXX()` 和 `setXXX()` 形式的。
- 包装类型自动生成的 getter 和 setter 方法，名称都是 `getXXX()` 和 `setXXX()` 形式的。

既然，我们已经达成一致共识使用基本类型 `boolean` 来定义成员变量了，那么我们来具体看下 `Model3` 和 `Model4` 中的 setter/getter 有何区别。

我们可以发现，虽然 `Model3` 和 `Model4` 中的成员变量的名称不同，一个是 `success`，另外一个为 `isSuccess`，但是他们自动生成的 getter 和 setter 方法名称都是 `isSuccess` 和 `setSuccess`。

Java Bean 中关于 setter/getter 的规范

关于 Java Bean 中的 getter/setter 方法的定义其实是有明确的规定的，根据 [JavaBeans\(TM\) Specification](#) 规定，如果是普通的参数 `propertyName`，要以以下方式定义其 setter/getter：

```
public <PropertyType> get<PropertyName>();
public void set<PropertyName>(<PropertyType> a);
```

但是，布尔类型的变量 `propertyName` 则是单独定义的：

```
public boolean is<PropertyName>();
public void set<PropertyName>(boolean m);
```

8.3.2 Boolean properties

In addition, for boolean properties, we allow a getter method to match the pattern:

```
public boolean is<PropertyName>();
```

This “is<PropertyName>” method may be provided instead of a “get<PropertyName>” method, or it may be provided in addition to a “get<PropertyName>” method.

In either case, if the “is<PropertyName>” method is present for a boolean property then we will use the “is<PropertyName>” method to read the property value.

An example boolean property might be:

```
public boolean isMarsupial();
public void setMarsupial(boolean m);
```

通过对照这份 JavaBeans 规范，我们发现，在 Model4 中，变量名为 `isSuccess`，如果严格按照规范定义的话，他的 getter 方法应该叫 `isIsSuccess`。但是很多 IDE 都会默认生成成为 `isSuccess`。

那这样做会带来什么问题呢。

在一般情况下，其实是没有影响的。但是有一种特殊情况就会有问题，那就是发生序列化的时候。

序列化带来的影响

关于序列化和反序列化请参考 [Java 对象的序列化与反序列化](#)。我们这里拿比较常用的 JSON 序列化来举例，看看常用的 fastJson、jackson 和 Gson 之间有何区别：

```
public class BooleanMainTest {

    public static void main(String[] args) throws IOException {
        // 定一个 Model3 类型
        Model3 model3 = new Model3();
        model3.setSuccess(true);
```

```

        // 使用 fastjson(1.2.16) 序列化 model3 成字符串并输出
        System.out.println("Serializable Result With fastjson : " + JSON.
toJSONString(model3));

        // 使用 Gson(2.8.5) 序列化 model3 成字符串并输出
        Gson gson =new Gson();
        System.out.println("Serializable Result With Gson : " +gson.
toJson(model3));

        // 使用 jackson(2.9.7) 序列化 model3 成字符串并输出
        ObjectMapper om = new ObjectMapper();
        System.out.println("Serializable Result With jackson : " +om.
writeValueAsString(model3));
    }

}

class Model3 implements Serializable {

    private static final long serialVersionUID = 1836697963736227954L;
    private boolean isSuccess;
    public boolean isSuccess() {
        return isSuccess;
    }
    public void setSuccess(boolean success) {
        isSuccess = success;
    }
    public String getHollis(){
        return "hollischuang";
    }
}

```

以上代码的 Model3 中，只有一个成员变量即 isSuccess，三个方法，分别是 IDE 帮我们自动生成的 isSuccess 和 setSuccess，另外一个作者自己增加的一个符合 getter 命名规范的方法。

以上代码输出结果：

```

Serializable Result With fastjson : {"hollis":"hollischuang","success":true}
Serializable Result With Gson : {"isSuccess":true}
Serializable Result With jackson : {"success":true,"hollis":"hollischuang"}

```

在 fastjson 和 jackson 的结果中，原来类中的 isSuccess 字段被序列化成功 success，并且其中还包含 hollis 值。而 Gson 中只有 isSuccess 字段。

我们可以得出结论: fastjson 和 jackson 在把对象序列化成 json 字符串的时候, 是通过反射遍历出该类中的所有 getter 方法, 得到 getHollis 和 isSuccess, 然后根据 JavaBeans 规则, 他会认为这是两个属性 hollis 和 success 的值。直接序列化成 json:{"hollis": "hollischuang", "success": true}

但是 Gson 并不是这么做的, 他是通过反射遍历该类中的所有属性, 并把其值序列化成 json:{"isSuccess": true}

可以看到, 由于不同的序列化工具, 在进行序列化的时候使用到的策略是不一样的, 所以, 对于同一个类的同一个对象的序列化结果可能是不同的。

前面提到的关于对 getHollis 的序列化只是为了说明 fastjson、jackson 和 Gson 之间的序列化策略的不同, 我们暂且把他放到一边, 我们把他从 Model3 中删除后, 重新执行下以上代码, 得到结果:

```
Serializable Result With fastjson :{"success":true}
Serializable Result With Gson :{"isSuccess":true}
Serializable Result With jackson :{"success":true}
```

现在, 不同的序列化框架得到的 json 内容并不相同, 如果对于同一个对象, 我使用 fastjson 进行序列化, 再使用 Gson 反序列化会发生什么?

```
public class BooleanMainTest {
    public static void main(String[] args) throws IOException {
        Model3 model3 = new Model3();
        model3.setSuccess(true);
        Gson gson = new Gson();
        System.out.println(gson.fromJson(JSON.toJSONString(model3), Model3.class));
    }
}

class Model3 implements Serializable {
    private static final long serialVersionUID = 1836697963736227954L;
    private boolean isSuccess;
    public boolean isSuccess() {
        return isSuccess;
    }
    public void setSuccess(boolean success) {
```

```
        isSuccess = success;
    }
    @Override
    public String toString() {
        return new StringJoiner(", ", Model3.class.getSimpleName() + "[", "]")
            .add("isSuccess=" + isSuccess)
            .toString();
    }
}
```

以上代码，输出结果：

```
Model3[isSuccess=false]
```

这和我们预期的结果完全相反，原因是因为 JSON 框架通过扫描所有的 getter 后发现有一个 isSuccess 方法，然后根据 JavaBeans 的规范，解析出变量名为 success，把 model 对象序列化城字符串后内容为 `{"success":true}`。

根据 `{"success":true}` 这个 json 串，Gson 框架在通过解析后，通过反射寻找 Model 类中的 success 属性，但是 Model 类中只有 isSuccess 属性，所以，最终反序列化后的 Model 类的对象中，isSuccess 则会使用默认值 false。

但是，一旦以上代码发生在生产环境，这绝对是一个致命的问题。

所以，作为开发者，我们应该想办法尽量避免这种问题的发生，对于 POJO 的设计者来说，只需要做简单的一件事就可以解决这个问题了，那就是把 isSuccess 改为 success。这样，该类里面的成员变量是 success，getter 方法是 isSuccess，这是完全符合 JavaBeans 规范的。无论哪种序列化框架，执行结果都一样。就从源头避免了这个问题。

引用以下 R 大关于阿里巴巴 Java 开发手册这条规定的评价 (<https://www.zhihu.com/question/55642203>):

8. 【强制】POJO 类中布尔类型的变量，都不要加 is，否则部分框架解析会引起序列化错误。
反例：定义为基本数据类型 boolean isSuccess；的属性，它的方法也是 isSuccess()，RPC 框架在反向解析的时候，“以为”对应的属性名称是 success，导致属性获取不到，进而抛出异常。

虽然也是主观规定，但这是阿里系的 Java 代码的惨痛的经验教训。对阿里系的代码来说，总之遵循的话可以减少灵异状况的发生，所以放在这个上下文里也是很合理的。

这个可以说是在用规范来规避团队里常用的库的坑。这坑的解决办法可以是修改常用库里的逻辑来尽可能“聪明”地识别情况，也可以靠这样的规范。某种意义上说规范是从上游卡住了问题的发生，比起把下游处理弄复杂要更干净一些吧。

所以，在定义 POJO 中的布尔类型的变量时，不要使用 isSuccess 这种形式，而要直接使用 success！

Boolean 还是 boolean？

前面我们介绍完了在 success 和 isSuccess 之间如何选择，那么排除错误答案后，备选项还剩下：

```
boolean success
Boolean success
```

那么，到底应该用 Boolean 还是 boolean 来给定一个布尔类型的变量呢？

我们知道，boolean 是基本数据类型，而 Boolean 是包装类型。关于基本数据类型和包装类之间的关系和区别请参考[一文读懂什么是 Java 中的自动拆装箱](#)

那么，在定义一个成员变量的时候到底是使用包装类型更好还是使用基本数据类型呢？

我们来看一段简单的代码

```
/**
 * @author Hollis
 */
public class BooleanMainTest {
    public static void main(String[] args) {
```

```
        Model model1 = new Model();
        System.out.println("default model : " + model1);
    }
}

class Model {
    /**
     * 定一个 Boolean 类型的 success 成员变量
     */
    private Boolean success;
    /**
     * 定一个 boolean 类型的 failure 成员变量
     */
    private boolean failure;

    /**
     * 覆盖 toString 方法, 使用 Java 8 的 StringJoiner
     */
    @Override
    public String toString() {
        return new StringJoiner(", ", Model.class.getSimpleName() + "[", "]")
            .add("success=" + success)
            .add("failure=" + failure)
            .toString();
    }
}
```

以上代码输出结果为:

```
default model : Model[success=null, failure=false]
```

可以看到, 当我们没有设置 Model 对象的字段的值的时候, Boolean 类型的变量会设置默认值为 `null`, 而 boolean 类型的变量会设置默认值为 `false`。

即对象的默认值是 `null`, boolean 基本数据类型的默认值是 `false`。

在阿里巴巴 Java 开发手册中, 对于 POJO 中如何选择变量的类型也有着一些规定:

8. 关于基本数据类型与包装数据类型的使用标准如下:

- 1) **【强制】**所有的 POJO 类属性必须使用包装数据类型。
- 2) **【强制】**RPC 方法的返回值和参数必须使用包装数据类型。
- 3) **【推荐】**所有的局部变量使用基本数据类型。

说明: POJO 类属性没有初值是提醒使用者在需要使用时, 必须自己显式地进行赋值, 任何 NPE 问题, 或者入库检查, 都由使用者来保证。

正例: 数据库的查询结果可能是 null, 因为自动拆箱, 用基本数据类型接收有 NPE 风险。

反例: 比如显示成交总额涨跌情况, 即正负 x%, x 为基本数据类型, 调用的 RPC 服务, 调用不成功时, 返回的是默认值, 页面显示为 0%, 这是不合理的, 应该显示成中划线。所以包装数据类型的 null 值, 能够表示额外的信息, 如: 远程调用失败, 异常退出。

这里建议我们使用包装类型, 原因是什么呢?

举一个扣费的例子, 我们做一个扣费系统, 扣费时需要从外部的定价系统中读取一个费率的价值, 我们预期该接口的返回值中会包含一个浮点型的费率字段。当我们取到这个值得时候就使用公式: 金额 * 费率 = 费用 进行计算, 计算结果进行划扣。

如果由于计费系统异常, 他可能会返回个默认值, 如果这个字段是 Double 类型的话, 该默认值为 null, 如果该字段是 double 类型的话, 该默认值为 0.0。

如果扣费系统对于该费率返回值没做特殊处理的话, 拿到 null 值进行计算会直接报错, 阻断程序。拿到 0.0 可能就直接进行计算, 得出接口为 0 后进行扣费了。这种异常情况就无法被感知。

这种使用包装类型定义变量的方式, 通过异常来阻断程序, 进而可以被识别到这种线上问题。如果使用基本数据类型的话, 系统可能不会报错, 进而认为无异常。

以上, 就是建议在 POJO 和 RPC 的返回值中使用包装类型的原因。

但是关于这一点, 作者之前也有过不同的看法: 对于布尔类型的变量, 我认为可以和其他类型区分开来, 作者并不认为使用 null 进而导致 NPE 是一种最好的实践。因为布尔类型只有 true/false 两种值, 我们完全可以和外部调用方约定好当返回值为 false 时的明确语义。

后来，作者单独和《阿里巴巴 Java 开发手册》、《码出高效》的作者——孤尽 单独 1V1(qing) Battle(jiao) 了一下。最终达成共识，还是**尽量使用包装类型**。

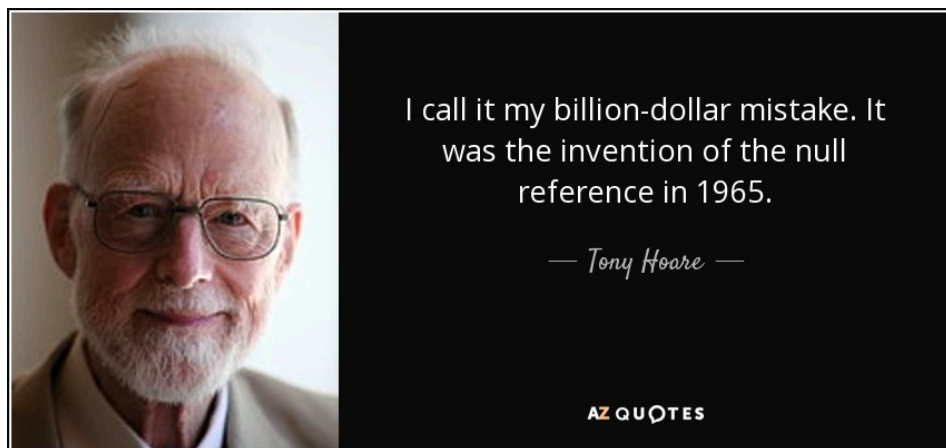
但是，作者还是想强调一个我的观点，**尽量避免在你的代码中出现不确定的 null 值**。

null 何罪之有?

关于 null 值的使用，我在[使用 Optional 避免 NullPointerException](#)、[9 Things about Null in Java](#) 等文中就介绍过。

`null` 是很模棱两可的，很多时候会导致令人疑惑的错误，很难去判断返回一个 `null` 代表着什么意思。

图灵奖得主 Tony Hoare 曾经公开表达过 `null` 是一个糟糕的设计。



我把 null 引用称为自己的十亿美元错误。它的发明是在 1965 年，那时我用一个面向对象语言 (ALGOL W) 设计了第一个全面的引用类型系统。我的目的是确保所有引用的使用都是绝对安全的，编译器会自动进行检查。但是我未能抵御住诱惑，加入了 Null 引用，仅仅是因为实现起来非常容易。它导致了数不清的错误、漏洞和系

统崩溃，可能在之后 40 年中造成了十亿美元的损失。

总结

本文围绕布尔类型的变量定义的类型和命名展开了介绍，最终我们可以得出结论，在定义一个布尔类型的变量，尤其是一个给外部提供的接口返回值时，要使用 success 来命名，阿里巴巴 Java 开发手册建议使用封装类来定义 POJO 和 RPC 返回值中的变量。但是这并不意味着可以随意的使用 null，我们还是要尽量避免出现对 null 的处理的。

为什么阿里巴巴禁止开发人员修改 serialVersionUID 字段的值？

序列化是一种对象持久化的手段。普遍应用在网络传输、RMI 等场景中。类通过实现 `java.io.Serializable` 接口以启用其序列化功能。

在我的博客中，其实已经有多篇文章介绍过序列化了，对序列化的基础知识不够了解的朋友可以参考以下几篇文章：

[Java 对象的序列化与反序列化](#)、[深入分析 Java 的序列化与反序列化](#)、[单例与序列化的那些事儿](#)

在这几篇文章中，我分别介绍过了序列化涉及到的类和接口、如何自定义序列化策略、`transient` 关键字和序列化的关系等，还通过学习 `ArrayList` 对序列化的实现源码深入学习了序列化。并且还拓展分析了一下序列化对单例的影响等。

但是，还有一个知识点并未展开介绍，那就是关于 `serialVersionUID`。这个字段到底有什么用？如果不设置会怎么样？为什么《阿里巴巴 Java 开发手册》中有以下规定：

10. 【强制】序列化类新增属性时，请不要修改 `serialVersionUID` 字段，避免反序列化失败；如果完全不兼容升级，避免反序列化混乱，那么请修改 `serialVersionUID` 值。

说明：注意 `serialVersionUID` 不一致会抛出序列化运行时异常。

背景知识

Serializable 和 Externalizable

类通过实现 `java.io.Serializable` 接口以启用其序列化功能。**未实现此接口的类将无法进行序列化或反序列化**。可序列化类的所有子类型本身都是可序列化的。

如果读者看过 `Serializable` 的源码，就会发现，他只是一个空的接口，里面什么东西都没有。**Serializable 接口没有方法或字段，仅用于标识可序列化的语义**。但是，如果一个类没有实现这个接口，想要被序列化的话，就会抛出 `java.io.NotSerializableException` 异常。

它是怎么保证只有实现了该接口的方法才能进行序列化与反序列化的呢？

原因是在执行序列化的过程中，会执行到以下代码：

```
if (obj instanceof String) {
    writeString((String) obj, unshared);
} else if (cl.isArray()) {
    writeArray(obj, desc, unshared);
} else if (obj instanceof Enum) {
    writeEnum((Enum<?>) obj, desc, unshared);
} else if (obj instanceof Serializable) {
    writeOrdinaryObject(obj, desc, unshared);
} else {
    if (extendedDebugInfo) {
        throw new NotSerializableException(
            cl.getName() + "\n" + debugInfoStack.toString());
    } else {
        throw new NotSerializableException(cl.getName());
    }
}
```

在进行序列化操作时，会判断要被序列化的类是否是 `Enum`、`Array` 和 `Serializable` 类型，如果都不是则直接抛出 `NotSerializableException`。

Java 中还提供了 `Externalizable` 接口，也可以实现它来提供序列化能力。

`Externalizable` 继承自 `Serializable`, 该接口中定义了两个抽象方法: `writeExternal()` 与 `readExternal()`。

当使用 `Externalizable` 接口来进行序列化与反序列化的时候需要开发人员重写 `writeExternal()` 与 `readExternal()` 方法。否则所有变量的值都会变成默认值。

transient

`transient` 关键字的作用是控制变量的序列化, 在变量声明前加上该关键字, 可以阻止该变量被序列化到文件中, 在被反序列化后, `transient` 变量的值被设为初始值, 如 `int` 型的是 0, 对象型的是 `null`。

自定义序列化策略

在序列化过程中, 如果被序列化的类中定义了 `writeObject` 和 `readObject` 方法, 虚拟机会试图调用对象类里的 `writeObject` 和 `readObject` 方法, 进行用户自定义的序列化和反序列化。

如果没有这样的方法, 则默认调用是 `ObjectOutputStream` 的 `defaultWriteObject` 方法以及 `ObjectInputStream` 的 `defaultReadObject` 方法。

用户自定义的 `writeObject` 和 `readObject` 方法可以允许用户控制序列化的过程, 比如可以在序列化的过程中动态改变序列化的数值。

所以, 对于一些特殊字段需要定义序列化的策略的时候, 可以考虑使用 `transient` 修饰, 并自己重写 `writeObject` 和 `readObject` 方法, 如 `java.util.ArrayList` 中就有这样的实现。

我们随便找几个 Java 中实现了序列化接口的类, 如 `String`、`Integer` 等, 我们可以发现一个细节, 那就是这些类除了实现了 `Serializable` 外, 还定义了一个 `serialVersionUID`


```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];

    /** Cache the hash code for the string */
    private int hash; // Default to 0

    /** use serialVersionUID from JDK 1.0.2 for interoperability */
    private static final long serialVersionUID = -6849794470754667710L;
```

那么，到底什么是 serialVersionUID 呢？为什么要设置这样一个字段呢？

什么是 serialVersionUID

序列化是将对象的状态信息转换为可存储或传输的形式过程。我们都知道，Java 对象是保存在 JVM 的堆内存中的，也就是说，如果 JVM 堆不存在了，那么对象也就跟着消失了。

而序列化提供了一种方案，可以让你在即使 JVM 停机的情况下也能把对象保存下来的方案。就像我们平时用的 U 盘一样。把 Java 对象序列化可存储或传输的形式（如二进制流），比如保存在文件中。这样，当再次需要这个对象的时候，从文件中读取出二进制流，再从二进制流中反序列化出对象。

虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个类的序列化 ID 是否一致，这个所谓的序列化 ID，就是我们在代码中定义的 serialVersionUID。

如果 serialVersionUID 变了会怎样

我们举个例子吧，看看如果 serialVersionUID 被修改了会发生什么？

```
public class SerializableDemo1 {
    public static void main(String[] args) {
        //Initializes The Object
        User1 user = new User1();
        user.setName("hollis");
```

```
//Write Obj to File
ObjectOutputStream oos = null;
try {
    oos = new ObjectOutputStream(new FileOutputStream("tempFile"));
    oos.writeObject(user);
} catch (IOException e) {
    e.printStackTrace();
} finally {
    IOUtils.closeQuietly(oos);
}
}
}

class User1 implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

我们先执行以上代码，把一个 User1 对象写入到文件中。然后我们修改一下 User1 类，把 `serialVersionUID` 的值改为 `2L`。

```
class User1 implements Serializable {
    private static final long serialVersionUID = 2L;
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

然后执行以下代码，把文件中的对象反序列化出来：

```
public class SerializableDemo2 {
    public static void main(String[] args) {
        //Read Obj from File
        File file = new File("tempFile");
        ObjectInputStream ois = null;
```

```
try {
    ois = new ObjectInputStream(new FileInputStream(file));
    User1 newUser = (User1) ois.readObject();
    System.out.println(newUser);
} catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} finally {
    IOUtils.closeQuietly(ois);
    try {
        FileUtils.forceDelete(file);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
```

执行结果如下:

```
java.io.InvalidClassException: com.hollis.User1; local class incompatible:
stream classdesc
serialVersionUID = 1, local class serialVersionUID = 2
```

可以发现, 以上代码抛出了一个 `java.io.InvalidClassException`, 并且指出 `serialVersionUID` 不一致。

这是因为, 在进行反序列化时, JVM 会把传来的字节流中的 `serialVersionUID` 与本地相应实体类的 `serialVersionUID` 进行比较, 如果相同就认为是一致的, 可以进行反序列化, 否则就会出现序列化版本不一致的异常, 即是 `InvalidCastException`。

这也是《阿里巴巴 Java 开发手册》中规定, 在兼容性升级中, 在修改类的时候, 不要修改 `serialVersionUID` 的原因。除非是完全不兼容的两个版本。所以, `serialVersionUID` 其实是验证版本一致性的。

如果读者感兴趣, 可以把各个版本的 JDK 代码都拿出来看一下, 那些向下兼容的类的 `serialVersionUID` 是没有变化过的。比如 String 类的 `serialVersionUID`

nUID 一直都是 -6849794470754667710L。

但是，作者认为，这个规范其实还可以再严格一些，那就是规定：

如果一个类实现了 `Serializable` 接口，就必须手动添加一个 `private static final long serialVersionUID` 变量，并且设置初始值。

为什么要明确一个 serialVersionUID

如果我们没有在类中明确的定义一个 `serialVersionUID` 的话，看看会发生什么。

尝试修改上面的 demo 代码，先使用以下类定义一个对象，该类中不定义 `serialVersionUID`，将其写入文件。

```
class User1 implements Serializable {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

然后我们修改 `User1` 类，向其中增加一个属性。在尝试将其从文件中读取出来，并进行反序列化。

```
class User1 implements Serializable {
    private String name;
    private int age;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
}
```

```
public void setAge(int age) {  
    this.age = age;  
}  
}
```

执行结果: `java.io.InvalidClassException: com.hollis.User1;
local class incompatible: stream classdesc serialVersionUID
= -2986778152837257883, local class serialVersionUID =
7961728318907695402`

同样, 抛出了 `InvalidClassException`, 并且指出两个 `serialVersionUID` 不同, 分别是 `-2986778152837257883` 和 `7961728318907695402`。

从这里可以看出, 系统自己添加了一个 `serialVersionUID`。

所以, 一旦类实现了 `Serializable`, 就建议明确的定义一个 `serialVersionUID`。不然在修改类的时候, 就会发生异常。

`serialVersionUID` 有两种显示的生成方式:

一是默认的 1L, 比如: `private static final long serialVersionUID
= 1L;`

二是根据类名、接口名、成员方法及属性等来生成一个 64 位的哈希字段, 比如:

```
private static final long serialVersionUID = xxxxL;
```

后面这种方式, 可以借助 IDE 生成, 后面会介绍。

背后原理

知其然, 要知其所以然, 我们再来看看源码, 分析一下为什么 `serialVersionUID` 改变的时候会抛异常? 在没有明确定义的情况下, 默认的 `serialVersionUID` 是怎么来的?

为了简化代码量，反序列化的调用链如下：

```
ObjectInputStream.readObject -> readObject0 -> readOrdinaryObject -> readClassDesc -> readNonProxyDesc -> ObjectStreamClass.initNonProxy
```

在 `initNonProxy` 中，关键代码如下：

```
/**
 * Initializes class descriptor representing a non-proxy class.
 */
void initNonProxy(ObjectStreamClass model,
                  Class<?> cl,
                  ClassNotFoundException resolveEx,
                  ObjectStreamClass superDesc)
    throws InvalidClassException
{
    long suid = Long.valueOf(model.getSerialVersionUID());
    ObjectStreamClass osc = null;
    if (cl != null) {
        osc = lookup(cl, all: true);
        if (osc.isProxy()) {
            throw new InvalidClassException(
                "cannot bind non-proxy descriptor to a proxy class");
        }
        if (model.isEnum != osc.isEnum) {
            throw new InvalidClassException(model.isEnum ?
                "cannot bind enum descriptor to a non-enum class" :
                "cannot bind non-enum descriptor to an enum class");
        }

        if (model.serializable == osc.serializable &&
            !cl.isArray() &&
            suid != osc.getSerialVersionUID()) {
            throw new InvalidClassException(osc.name,
                "local class incompatible: " +
                "stream classdesc serialVersionUID = " + suid +
                ", local class serialVersionUID = " +
                osc.getSerialVersionUID());
        }

        if (!classNameEqual(model.name, osc.name)) {
            throw new InvalidClassException(osc.name,
                "local class name incompatible with stream class " +
                "name \"" + model.name + "\"");
        }

        if (!model.isEnum) {
            if ((model.serializable == osc.serializable) &&
                (model.externalizable != osc.externalizable)) {
                throw new InvalidClassException(osc.name,
                    "Serializable incompatible with Externalizable");
            }

            if ((model.serializable != osc.serializable) ||
                (model.externalizable != osc.externalizable) ||
                !(model.serializable || model.externalizable)) {
                deserializeEx = new ExceptionInfo(
                    osc.name, msg: "class invalid for deserialization");
            }
        }
    }
}
```

在反序列化过程中，对 `serialVersionUID` 做了比较，如果发现不相等，则直接抛出异常。

深入看一下 `getSerialVersionUID` 方法：

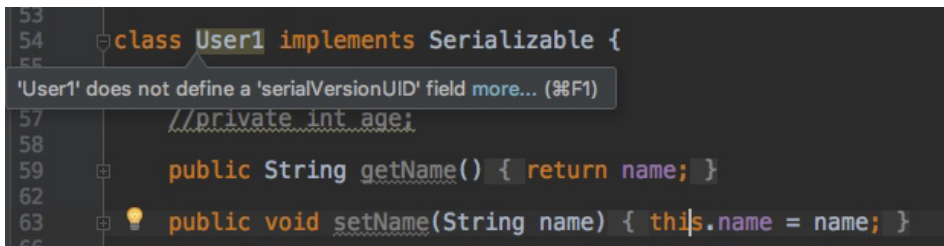
```
public long getSerialVersionUID() {
    // REMIND: synchronize instead of relying on volatile?
    if (suid == null) {
        suid = AccessController.doPrivileged(
            new PrivilegedAction<Long>() {
                public Long run() {
                    return computeDefaultSUID(c1);
                }
            }
        );
    }
    return suid.longValue();
}
```

在没有定义 `serialVersionUID` 的时候，会调用 `computeDefaultSUID` 方法，生成一个默认的 `serialVersionUID`。

这也就找到了以上两个问题的根源，其实是代码中做了严格的校验。

IDEA 提示

为了确保我们不会忘记定义 `serialVersionUID`，可以调节一下 IntelliJ IDEA 的配置，在实现 `Serializable` 接口后，如果没定义 `serialVersionUID` 的话，IDEA (eclipse 一样) 会进行提示：



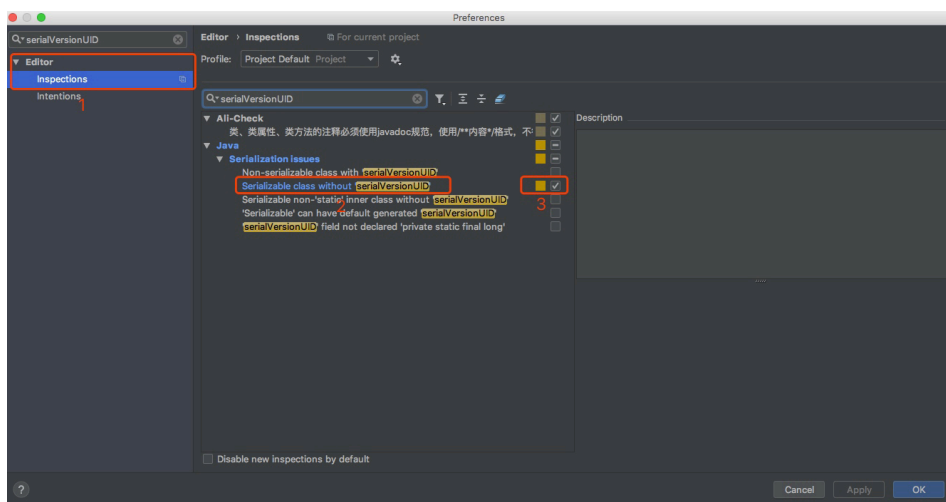
```
53
54 class User1 implements Serializable {
55
56     //private int age;
57
58     public String getName() { return name; }
59
60     public void setName(String name) { this.name = name; }
61
62
63
64 }
```

'User1' does not define a 'serialVersionUID' field more... (⌘F1)

并且可以一键生成一个：



当然，这个配置并不是默认生效的，需要手动到 IDEA 中设置一下：



在图中标号 3 的地方 (Serializable class without serialVersionUID 的配置)，打上勾，保存即可。

总结

serialVersionUID 是用来验证版本一致性的。所以在做兼容性升级的时候，不要改变类中 serialVersionUID 的值。

如果一个类实现了 Serializable 接口，一定要记得定义 serialVersionUID，

否则会发生异常。可以在 IDE 中通过设置，让他帮忙提示，并且可以一键快速生成一个 `serialVersionUID`。

之所以会发生异常，是因为反序列化过程中做了校验，并且如果没有明确定义的话，会根据类的属性自动生成一个。

为什么阿里巴巴建议开发者谨慎使用继承？

从学习 Java 的第一天起，我们就知道 Java 是一种面向对象语言，而学习 Java 的第二天，我们就知道了面向对象的三大基本特性是：封装、继承、多态。

所以，对于很多开发者来说，继承肯定都是不陌生的。但是，继承一定适合所有的场景吗？毫无忌讳的使用继承来做代码扩展真的好吗？

为什么《阿里巴巴 Java 开发手册》中有一条规定：谨慎使用继承的方式进行扩展，优先使用组合的方式实现。

9. 【推荐】谨慎使用继承的方式进行扩展，优先使用聚合/组合的方式来实现。

说明：不得已使用继承的话，必须符合里氏代换原则，此原则说父类能够出现的地方子类一定能够出现，比如，“把钱交出来”，钱的子类美元、欧元、人民币等都可以出现。

本文就来针对这些问题，简单分析一下。

面向对象的复用技术

每个人在刚刚学习继承的时候都会或多或少的有这样一个印象：继承可以帮助我实现类的复用。所以，很多开发人员在需要复用一些代码的时候会很自然的使用类的继承的方式，因为书上就是这么写的（老师就是这么教的）。但是，其实这样做是不对的。长期大量的使用继承会给代码带来很高的维护成本。

前面提到复用，这里就简单介绍一下面向对象的复用技术。

复用性是面向对象技术带来的很棒的潜在好处之一。如果运用的好的话可以帮助我们节省很多开发时间，提升开发效率。但是，如果被滥用那么就可能产生很多难以

维护的代码。

作为一门面向对象开发的语言，代码复用是 Java 引人注意的功能之一。**Java** 代码的复用有继承，组合以及代理三种具体的表现形式。

为什么阿里巴巴禁止使用 count(列名) 或 count(常量) 来替代 count(*) ?

数据库查询相信很多人都不陌生，所有经常有人调侃程序员就是 CRUD 专员，这所谓的 CRUD 指的就是数据库的增删改查。

在数据库的增删改查操作中，使用最频繁的就是查询操作。而在所有查询操作中，统计数量操作更是经常被用到。

关于数据库中行数统计，无论是 MySQL 还是 Oracle，都有一个函数可以使用，那就是 COUNT。

认识 COUNT

关于 COUNT 函数，在 MySQL 官网中有详细介绍：

• COUNT(expr) [over_clause]

Returns a count of the number of non-NULL values of **expr** in the rows retrieved by a SELECT statement. The result is a BIGINT value.

If there are no matching rows, COUNT() returns 0.

This function executes as a window function if **over_clause** is present. **over_clause** is as described in Section 12.21.2, "Window Function Concepts and Syntax".

```
1  mysql> SELECT student.student_name,COUNT(*)
2          FROM student,course
3          WHERE student.student_id=course.student_id
4          GROUP BY student_name;
```

COUNT(*) is somewhat different in that it returns a count of the number of rows retrieved, whether or not they contain NULL values.

简单翻译一下：

1. COUNT(expr)，返回 SELECT 语句检索的行中 expr 的值不为 NULL 的数量。结果是一个 BIGINT 值。
2. 如果查询结果没有命中任何记录，则返回 0。

3. 但是，值得注意的是，`COUNT(*)` 的统计结果中，会包含值为 NULL 的行数。

即以下表记录

```
create table #bla(id int,id2 int)
insert #bla values(null,null)
insert #bla values(1,null)
insert #bla values(null,1)
insert #bla values(1,null)
insert #bla values(null,1)
insert #bla values(1,null)
insert #bla values(null,null)
```

使用语句 `count(*)`,`count(id)`,`count(id2)` 查询结果如下：

```
select count(*),count(id),count(id2)
from #bla
results 7 3 2
```

除了 `COUNT(id)` 和 `COUNT(*)` 以外，还可以使用 `COUNT(常量)` (如 `COUNT(1)`) 来统计行数，那么这三条 SQL 语句有什么区别呢？到底哪种效率更高呢？为什么《阿里巴巴 Java 开发手册》中强制要求不让使用 `COUNT(列名)` 或 `COUNT(常量)` 来替代 `COUNT(*)` 呢？

1. 【强制】不要使用 `count(列名)` 或 `count(常量)` 来替代 `count(*)`，`count(*)` 是 SQL92 定义的标准统计行数的语法，跟数据库无关，跟 NULL 和非 NULL 无关。

说明：`count(*)` 会统计值为 NULL 的行，而 `count(列名)` 不会统计此列为 NULL 值的行。

COUNT(列名)、COUNT(常量) 和 COUNT(*) 之间的区别

前面我们提到过 `COUNT(expr)` 用于做行数统计，统计的是 `expr` 不为 NULL 的行数，那么 `COUNT(列名)`、`COUNT(常量)` 和 `COUNT(*)` 这三种语法中，`expr` 分别是列名、常量和 *。

那么列名、常量和 * 这三个条件中，常量是一个固定值，肯定不为 NULL。*

可以理解为查询整行，所以肯定也不为 NULL，那么就只有列名的查询结果有可能是 NULL 了。

所以，`COUNT(常量)` 和 `COUNT(*)` 表示的是直接查询符合条件的数据库表的行数。而 `COUNT(列名)` 表示的是查询符合条件的列的值不为 NULL 的行数。

除了查询得到结果集有区别之外，`COUNT(*)` 相比 `COUNT(常量)` 和 `COUNT(列名)` 来讲，`COUNT(*)` 是 SQL92 定义的标准统计行数的语法，因为他是标准语法，所以 MySQL 数据库对他进行过很多优化。

SQL92，是数据库的一个 ANSI/ISO 标准。它定义了一种语言（SQL）以及数据库的行为（事务、隔离级别等）。

COUNT(*) 的优化

前面提到了 `COUNT(*)` 是 SQL92 定义的标准统计行数的语法，所以 MySQL 数据库对他进行过很多优化。那么，具体都做过哪些事情呢？

这里的介绍要区分不同的执行引擎。MySQL 中比较常用的执行引擎就是 InnoDB 和 MyISAM。

MyISAM 和 InnoDB 有很多区别，其中有一个关键的区别和我们接下来要介绍的 `COUNT(*)` 有关，那就是 MyISAM 不支持事务，MyISAM 中的锁是表级锁；而 InnoDB 支持事务，并且支持行级锁。

因为 MyISAM 的锁是表级锁，所以同一张表上面的操作需要串行进行，所以，MyISAM 做了一个简单的优化，那就是它可以把表的总行数单独记录下来，如果从一张表中使用 `COUNT(*)` 进行查询的时候，可以直接返回这个记录下来的数值就可以了，当然，前提是不能有 where 条件。

MyISAM 之所以可以把表中的总行数记录下来供 `COUNT(*)` 查询使用，那是因为 MyISAM 数据库是表级锁，不会有并发的数据库行数修改，所以查询得到的行数

是准确的。

但是，对于 InnoDB 来说，就不能做这种缓存操作了，因为 InnoDB 支持事务，其中大部分操作都是行级锁，所以可能表的行数可能会被并发修改，那么缓存记录下来的总行数就不准确了。

但是，InnoDB 还是针对 COUNT(*) 语句做了些优化的。

在 InnoDB 中，使用 COUNT(*) 查询行数的时候，不可避免的要进行扫表了，那么，就可以在扫表过程中下功夫来优化效率了。

从 MySQL 8.0.13 开始，针对 InnoDB 的 `SELECT COUNT(*) FROM tbl_name` 语句，确实在扫表的过程中做了一些优化。前提是查询语句中不包含 WHERE 或 GROUP BY 等条件。

我们知道，COUNT(*) 的目的只是为了统计总行数，所以，他根本不关心自己查到的具体值，所以，他如果能够在扫表的过程中，选择一个成本较低的索引进行的话，那就可以大大节省时间。

我们知道，InnoDB 中索引分为聚簇索引（主键索引）和非聚簇索引（非主键索引），聚簇索引的叶子节点中保存的是整行记录，而非聚簇索引的叶子节点中保存的是该行记录的主键的值。

所以，相比之下，非聚簇索引要比聚簇索引小很多，所以 MySQL 会优先选择最小的非聚簇索引来扫表。所以，当我们建表的时候，除了主键索引以外，创建一个非主键索引还是有必要的。

至此，我们介绍完了 MySQL 数据库对于 COUNT(*) 的优化，这些优化的前提都是查询语句中不包含 WHERE 以及 GROUP BY 条件。

COUNT(*) 和 COUNT(1)

介绍完了 `COUNT(*)`，接下来看看 `COUNT(1)`，对于，这二者到底有没有区别，网上的说法众说纷纭。

有的说 `COUNT(*)` 执行时会转换成 `COUNT(1)`，所以 `COUNT(1)` 少了转换步骤，所以更快。

还有的说，因为 MySQL 针对 `COUNT(*)` 做了特殊优化，所以 `COUNT(*)` 更快。

那么，到底哪种说法是对的呢？看下 MySQL 官方文档是怎么说的：

InnoDB handles SELECT COUNT(*) and SELECT COUNT(1) operations in the same way. There is no performance difference.

画重点：`same way`，`no performance difference`。所以，对于 `COUNT(1)` 和 `COUNT(*)`，MySQL 的优化是完全一样的，根本不存在谁比谁快！

那既然 `COUNT(*)` 和 `COUNT(1)` 一样，建议用哪个好呢？

建议使用 `COUNT(*)`！因为这个是 SQL92 定义的标准统计行数的语法，而且本文只是基于 MySQL 做了分析，关于 Oracle 中的这个问题，也是众说纷纭的呢。

COUNT(字段)

最后，就是我们一直还没提到的 `COUNT( 字段 )`，他的查询就比较简单粗暴了，就是进行全表扫描，然后判断指定字段的值是不是为 NULL，不为 NULL 则累加。

相比 `COUNT(*)`，`COUNT( 字段 )` 多了一个步骤就是判断所查询的字段是否为 NULL，所以他的性能要比 `COUNT(*)` 慢。

总结

本文介绍了 COUNT 函数的用法，主要用于统计表行数。主要用法有 `COUNT(*)`、

COUNT(字段) 和 COUNT(1)。

因为 COUNT(*) 是 SQL92 定义的标准统计行数的语法，所以 MySQL 对他进行了很多优化，MyISAM 中会直接把表的总行数单独记录下来供 COUNT(*) 查询，而 InnoDB 则会在扫表的时候选择最小的索引来降低成本。当然，这些优化的前提都是没有进行 where 和 group 的条件查询。

在 InnoDB 中 COUNT(*) 和 COUNT(1) 实现上没有区别，而且效率一样，但是 COUNT(字段) 需要进行字段的非 NULL 判断，所以效率会低一些。

因为 COUNT(*) 是 SQL92 定义的标准统计行数的语法，并且效率高，所以请直接使用 COUNT(*) 查询表的行数！

参考资料: [《极客时间——MySQL 实战 45 讲》](#)



扫码下载
最新版《Java 开发手册》



阿里云开发者“藏经阁”
海量免费电子书下载



扫码关注 Hollis
一个对 Coding 有着独特追求的人