

目录	
第1章 基础	
01 开篇词：为什么学习本专栏	
02 String、Long 源码解析和面试题	
03 Java 常用关键字理解	最近阅读
04 Arrays、Collections、Objects 常用方法源码解析	
第2章 集合	
05 ArrayList 源码解析和设计思路	
06 LinkedList 源码解析	
07 List 源码会问哪些面试题	
08 HashMap 源码解析	
09 TreeMap 和 LinkedHashMap 核心源码解析	
10 Map源码会问哪些面试题	
11 HashSet、TreeSet 源码解析	
12 彰显细节：看集合源码对我们实际工作的帮助和应用	
13 差异对比：集合在 Java 7 和 8 有何不同和改进	
14 简化工作：Guava Lists Maps 实际工作运用和源码	
第3章 并发集合类	
15 CopyOnWriteArrayList 源码解析和设计思路	
16 ConcurrentHashMap 源码解析和设计思路	
17 并发 List、Map源码面试题	
18 场景集合：并发 List、Map的应用	

## 03 Java 常用关键字理解

更新时间：2019-08-27 14:25:15



世上无难事,只要肯登攀。  
——毛泽东

果断更，请联系QQ/微信6426006

Java 中的关键字很多，大约有 50+，在命名上我们不能和这些关键字冲突的，编译会报错，每个关键字都代表着不同场景下的不同含义，接下来我们挑选 6 个比较重要的关键字，深入学习一下。

### 1 static

意思是静态的、全局的，一旦被修饰，说明被修饰的东西在一定范围内是共享的，谁都可以访问，这时候需要注意并发读写的问题。

#### 1.1 修饰的对象

static 只能修饰类变量、方法和方法块。

当 **static** 修饰类变量时，如果该变量是 public 的话，表示该变量任何类都可以直接访问，而且无需初始化类，直接使用 类名.static 变量 这种形式访问即可。

这时候我们非常需要注意的一点就是线程安全的问题了，因为当多个线程同时对共享变量进行读写时，很有可能会出现并发问题，如我们定义了：`public static List<String> list = new ArrayList<>();` 这样的共享变量。这个 list 如果同时被多个线程访问的话，就有线程安全的问题，这时候一般有两个解决办法：

1. 把线程不安全的 ArrayList 换成 线程安全的 CopyOnWriteArrayList；
2. 每次访问时，手动加锁。

⚡ **STATIC 修饰方法时**，代表该方法和⚡则是无大的，任意尖都可以直接访问（如果仅限于 public 的话）。

有一点需要注意的是，该方法内部只能调用同样被 static 修饰的方法，不能调用普通方法，我们常用的 util 类里面的各种方法，我们比较喜欢用 static 修饰方法，好处就是调用特别方便。

static 方法内部的变量在执行时是没有线程安全问题的。方法执行时，数据运行在栈里面，栈的数据每个线程都是隔离开的，所以不会有线程安全的问题，所以 util 类的各个 static 方法，我们是可以放心使用的。

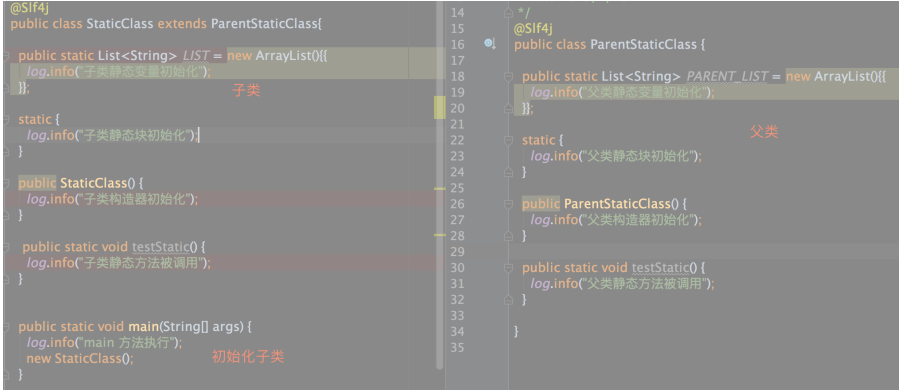
当 **static 修饰方法块时**，我们叫做静态块，静态块常常用于在类启动之前，初始化一些值，比如：

```
public static List<String> list = new ArrayList();
// 进行一些初始化的工作
static {
    list.add("1");
}
```

这段代码演示了静态块做一些初始化的工作，但需要注意的是，静态块只能调用同样被 static 修饰的变量，并且 static 的变量需要写在静态块的前面，不然编译也会报错。

1.2 初始化时机

对于被 static 修饰的类变量、方法块和静态方法的初始化时机，我们写了一个测试 demo，如下图所示：



打印出来的结果是：

- 父类静态变量初始化
- 父类静态块初始化
- 子类静态变量初始化
- 子类静态块初始化
- main 方法执行
- 父类构造器初始化
- 子类构造器初始化

从结果中，我们可以看出两点：

1. 父类的静态变量和静态块比子类优先初始化；
2. 静态变量和静态块比类构造器优先初始化。

## 2 final

final 的意思是不变的，一般来说用于以下三种场景：

1. 被 final 修饰的类，表明该类是无法继承的；
2. 被 final 修饰的方法，表明该方法是无法覆写的；
3. 被 final 修饰的变量，说明该变量在声明的时候，就必须初始化完成，而且以后也不能修改其内存地址。

第三点注意下，我们说的是无法修改其内存地址，并没有说无法修改其值。因为对于 List、Map 这些集合类来说，被 final 修饰后，是可以修改其内部值的，但却无法修改其初始化时的内存地址。

例子我们就不举了，1-1 小节 String 的不变性就是一个很好的例子。

## 3 try、catch、finally

这三个关键字常用于我们捕捉异常的一整套流程，try 用来确定代码执行的范围，catch 捕捉可能会发生的异常，finally 用来执行一定要执行的代码块，除了这些，我们还需要清楚，每个地方如果发生异常会怎么办，我们举一个例子来演示一下：

```
public void testCatchFinally() {
    try {
        log.info("try is run");
        if (true) {
            throw new RuntimeException("try exception");
        }
    } catch (Exception e) {
        log.info("catch is run");
        if (true) {
            throw new RuntimeException("catch exception");
        }
    } finally {
        log.info("finally is run");
    }
}
```

这个代码演示了在 try、catch 中都遇到了异常，代码的执行顺序为：try -> catch -> finally，输出的结果如下：

```
[main] INFO demo.one.FinallyDemo - try is run
[main] INFO demo.one.FinallyDemo - catch is run
[main] INFO demo.one.FinallyDemo - finally is run

java.lang.RuntimeException: catch exception
    at demo.one.FinallyDemo.testCatchFinally(FinallyDemo.java:25) <22 internal calls>
```

可以看到两点：

1. finally 先执行后，再抛出 catch 的异常；
2. 最终捕获的异常是 catch 的异常，try 抛出来的异常已经被 catch 吃掉了，所以当我们遇见 catch 也有可能会抛出异常时，我们可以先打印出 try 的异常，这样 try 的异常在日志中就会有所体现。

## 4 volatile

目录

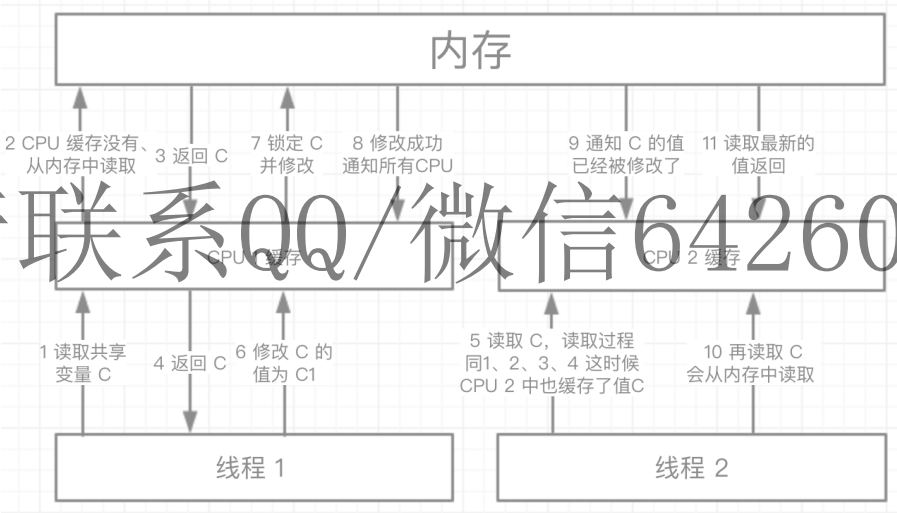
我们再说原理之前，先说下基础知识。就是在多核 CPU 下，为了提高效率，线程在拿值时，是直接和 CPU 缓存打交道的，而不是内存。主要是因为 CPU 缓存执行速度更快，比如线程要拿值 C，会直接从 CPU 缓存中拿，CPU 缓存中没有，就会从内存中拿，所以线程读的操作永远都是拿 CPU 缓存的值。

这时候会产生一个问题，CPU 缓存中的值和内存中的值可能并不是时刻都同步，导致线程计算的值可能不是最新的，共享变量的值有可能已经被其它线程所修改了，但此时修改是机器内存的值，CPU 缓存的值还是老的，导致计算会出现问题。

这时候有个机制，就是内存会主动通知 CPU 缓存。当前共享变量的值已经失效了，你需要重新来拉取一份，CPU 缓存就会重新从内存中拿取一份最新的值。

volatile 关键字就会触发这种机制，加了 volatile 关键字的变量，就会被识别成共享变量，内存中值被修改后，会通知到各个 CPU 缓存，使 CPU 缓存中的值也对应被修改，从而保证线程从 CPU 缓存中拿取出来的值是最新的。

我们画了一个图来说明一下：



从图中我们可以看到，线程 1 和线程 2 一开始都读取了 C 值，CPU 1 和 CPU 2 缓存中也都有了 C 值，然后线程 1 把 C 值修改了，这时候内存的值和 CPU 2 缓存中的 C 值就不等了，内存这时发现 C 值被 volatile 关键字修饰，发现其是共享变量，就会使 CPU 2 缓存中的 C 值状态置为无效，CPU 2 会从内存中重新拉取最新的值，这时候线程 2 再来读取 C 值时，读取的已经是内存中最新的值了。

5 transient

transient 关键字我们常用来修饰类变量，意思是当前变量是无需进行序列化的。在序列化时，就会忽略该变量，这些在序列化工具底层，就已经对 transient 进行了支持。

6 default

default 关键字一般会用在接口的方法上，意思是对于该接口，子类是无需强制实现的，但自己必须有默认实现，我们举个例子如下：

目录

```
4  /**
5   * ParentInterface
6   * author wenhe
7   * date 2019/7/29
8   */
9  public interface ParentInterface {
10
11      default void test(){
12          // doSomething
13      }
14
15  }
```

接口中的 test 方法  
被 default 修饰

```
4  * SubClass
5  * author wenhe
6  * date 2019/7/29
7  */
8  public class SubClass implements ParentInterface {
9
10
11  }
```

子类并不需要强制性的实现接口中的  
方法

default 关键字被很多源码使用，我们后面会说。

7 面试题

7.1 如何证明 static 静态变量和类无关？

答：从三个方面就可以看出静态变量和类无关。

- 1. 我们不需要初始化类就可直接使用静态变量；
- 2. 我们在类中写个 main 方法运行，即便不写初始化类的代码，静态变量都会自动初始化；
- 3. 静态变量只会初始化一次，初始化完成之后，不管我再 new 多少个类出来，静态变量都不会再初始化了。

不仅仅是静态变量，静态方法块也和类无关。

7.2 常常见变量和方法被 static 和 final 两个关键字修饰，为什么这么做？

答：这么做有两个目的：

- 1. 变量和方法于类无关，可以直接使用，使用比较方便；
- 2. 强调变量内存地址不可变，方法不可继承覆写，强调了方法内部的稳定性。

7.3 catch 中发生了未知异常，finally 还会执行么？

答：会的，catch 发生了异常，finally 还会执行的，并且是 finally 执行完成之后，才会抛出 catch 中的异常。

不过 catch 会吃掉 try 中抛出的异常，为了避免这种情况，在一些可以预见 catch 中会发生异常的地方，先把 try 抛出的异常打印出来，这样从日志中就可以看到完整的异常了。

7.4 volatile 关键字的作用和原理

答：这个上文说的比较清楚，可以参考上文。

总结

Java 的关键字属于比较基础的内容，我们需要清晰明确其含义，才能在后续源码阅读和工作中碰到这些关键字时了然于心，才能明白为什么会在哪里使用这样的关键字。比如 String 源码是如何使用 final 关键字达到起不变性的，比如 Java 8 集合中 Map 是如何利用 default 关键字新增各种方法的，这些我们在后续内容都会提到。

果断更，请联系QQ/微信64260066

目录

欢迎在这里发表留言，作者筛选后可公开显示

所相虚妄

这个读取应该是在通知之后才能用有作用吧，读取值之后，然后完成了计算，那么就有再写会缓存中的话，那么这个通知机制就没作用了吧。

👍 1    回复

2019-12-06

所相虚妄

try 那个代码是不是少了吧System.out.println(e.getMessage());这种代码了

👍 0    回复

2019-12-06

所相虚妄

我感觉文章中的表达应该是，静态方法和变量和对象没啥关系，就算没有初始化成对象，依然可以使用吧，

👍 0    回复

2019-12-06

wt4446

volatile的底层原理是cas吗？

👍 0    回复

2019-11-29

文贺    回复    wt4446

和 CAS 有关，底层实现是计算机底层实现的。

回复

2019-11-30 13:04:22

风云独舞九天

补充一波，有非静态代码块的情况： 父类静态变量 父类静态代码块 子类静态变量 子类静态代码块 父类非静态代码块 父类构造方法 子类非静态代码块 子类构造方法

👍 0    回复

2019-11-18

java\_simon

您好，问下在修改volatile变量时，先修改CPU缓存中的值再修改内存中的值吗？顺序是怎么样的呢？

👍 0    回复

2019-10-30

文贺    回复    java\_simon

同学你好，先修改当前机器(修改变量命令所有机器)的 CPU 缓存，再修改主存，在修改其他CPU 缓存。

回复

2019-10-31 10:45:55

<div>← 慕课专栏</div>	<div>面试官系统精讲Java源码及大厂真题 / 03 Java 常用关键字理解</div>
<div>目录</div>	<div>不是说String中内存地址不可变吗， String str = "abc"; str = "hello"; 那我上面的代码是只改变了str的值吗， 但是我在idea编辑器查看地址的时候发现上面的str的地址和下面改变后的str地址也变了 如果说确实地址是没变的，那也就是说idea编辑器看到的不是内存地址，那上面显示的又是 什么呢? 麻烦老师解答一下</div>
	<div><div>👍 1    回复</div><div>2019-09-30</div></div>
	<div><div>文贺 回复 qq_orientation_0</div><div>String 比较特殊，你重新赋值时，会产生新的内存引用，新的hello 和原来的 abc 是两个字符串，在 idea 上能看到两个内存地址。</div><div>回复</div><div>2019-10-03 07:54:21</div></div>
	<div><div>Blue_Fish0323 回复 qq_orientation_0</div><div>可以这么理解，str指向的是字符串"abc"的地址，你重新赋值的时候指向的是"hello"的地址，但是"abc"这个字符串的地址时没有变化的</div><div>回复</div><div>2019-10-17 20:54:35</div></div>
	<div><div>慕虎3144156</div><div>testCatchFinally这个再次运行时和楼主发的是一样的顺序了。不知道为什么</div><div>👍 0    回复</div><div>2019-09-29</div></div>
	<div><div>文贺 回复 慕虎3144156</div><div>同事自己在检查下运行环境哈，谢谢</div><div>回复</div><div>2019-09-30 14:06:17</div></div>
	<div>果断更，请联系QQ/微信6426006</div>
	<div><div>慕虎3144156</div><div>testCatchFinally，这个例子为什么在我电脑上和作者的不一样？ 我的结果是下面的 try is run Exception in thread "main" java.lang.RuntimeException: catch exception at testCatchFinally.testCatchFinally(testCatchFinally.java:21) catch is run at testCatchFinally.main (testCatchFinally.java:9) finally is run</div><div>👍 0    回复</div><div>2019-09-29</div></div>
	<div><div>风舞炫动</div><div>线程2在第10部如果是修改C的值为某一个值 这种情况是如何处理的呢？</div><div>👍 0    回复</div><div>2019-09-26</div></div>
	<div><div>文贺 回复 风舞炫动</div><div>会拿到最新的值进行修改。</div><div>回复</div><div>2019-09-26 19:01:43</div></div>
	<div><div>风舞炫动 回复    文贺</div><div>明白啦， 谢谢老师！</div><div>回复</div><div>2019-10-08 11:39:22</div></div>

目录	👍 0    回复	2019-09-23
----	-----------	------------

文贺 回复 tongguangyu

final 修饰的是不能修改哈，你应该问的是如何证明不可修改是吧，这个在编译期间是强制的，final 变量如果初始化时不赋值，编译器就会报错：Variable might not have been initialized，如果再次给 final 变量赋值，编译器会报错：Cannot assign a value to final variable 。

回复 2019-09-23 10:41:26

狗肉是只猫 回复 tongguangyu

final Student stu = new Student("五五开"); Student stu2 = new Student("white"); stu = stu2 // 错误，final修饰的对象，不可重新指向另一个对象。 stu.setName("卢本伟"); // 正确，可以修改值。

回复 2019-09-23 20:27:23

qq\_梦里有只小狐狸\_0 回复 狗肉是只猫

卢老爷的例子可以说很生动了

[点击展开剩余评论](#)

千学不如一看，千看不如一练

果断更， 请联系QQ/微信6426006