

#### Java关键字(一)——instanceof

- 1、obj 必须为引用类型，不能是基本类型
- 2、obj 为 null
- 3、obj 为 class 类的实例对象
- 4、obj 为 class 接口的实现类
- 5、obj 为 class 类的直接或间接子类
- 6、问题
- 7、深究原理
- 8、instanceof 的实现策略

#### Java关键字(二)——native

- 1、JNI: Java Native Interface
- 3、用C语言编写程序本地方法
- 4、JNI调用C的流程图
- 5、native关键字

#### Java关键字(三)——static

- 1、修饰成员变量
- 2、修饰修饰成员方法
- 3、静态代码块
- 4、静态导包
- 5、静态内部类
- 6、常见问题

#### Java关键字(四)——final

- 1、修饰变量
- 2、修饰方法
- 3、修饰类

#### Java关键字(五)——this

- 1、调用成员变量
- 2、调用构造方法
- 3、调用普通方法
- 4、返回当前对象

#### Java关键字(六)——super

- 1、调用父类的构造方法
- 2、调用父类的成员属性
- 3、调用父类的方法
- 4、this 和 super 出现在同一个构造方法中?

#### Java关键字(七)——synchronized

- 1、求示例代码结果
- 2、修饰代码块
- 3、修饰普通方法
- 4、修饰静态方法
- 5、原子性、可见性、有序性
- 6、锁对象
- 7、可重入
- 8、实现原理
- 9、异常自动unlock

#### Java关键字(八)——volatile

- 1、可见性
- 2、禁止指令重排序
- 3、总结



## Java关键字(一)——instanceof

instanceof 严格来说是Java中的一个双目运算符，用来测试一个对象是否为一个类的实例，用法为：

```
boolean result = obj instanceof Class
```

其中 obj 为一个对象，Class 表示一个类或者一个接口，当 obj 为 Class 的对象，或者是其直接或间接子类，或者是其接口的实现类，结果result 都返回 true，否则返回false。

注意：编译器会检查 obj 是否能转换成右边的class类型，如果不能转换则直接报错，如果不能确定类型，则通过编译，具体看运行时定。

### 1、obj 必须为引用类型，不能是基本类型

```
1  int i = 0;
2  System.out.println(i instanceof Integer); //编译不通过
3  System.out.println(i instanceof Object); //编译不通过
```

instanceof 运算符只能用作对象的判断。

## 2、obj 为 null

```
System.out.println(null instanceof Object);//false
```

关于 null 类型的描述在官方文档：<https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.1> 有一些介绍。一般我们知道Java分为两种数据类型，一种是基本数据类型，有八个分别是 byte short int long float double char boolean,一种是引用类型，包括类，接口，数组等等。而Java中还有一种特殊的 null 类型，该类型没有名字，所以不可能声明为 null 类型的变量或者转换为 null 类型，null 引用是 null 类型表达式唯一可能的值，null 引用也可以转换为任意引用类型。我们不需要对 null 类型有多深刻的了解，我们只需要知道 null 是可以成为任意引用类型的特殊符号。

在 JavaSE 规范 中对 instanceof 运算符的规定就是：如果 obj 为 null，那么将返回 false。

## 3、obj 为 class 类的实例对象

```
1 Integer integer = new Integer(1);
2 System.out.println(integer instanceof Integer);//true
```

这没什么好说的，最普遍的一种用法。

## 4、obj 为 class 接口的实现类

了解Java 集合的，我们知道集合中有个上层接口 List，其有个典型实现类 ArrayList

```
1 public class ArrayList<E> extends AbstractList<E>
2     implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

所以我们可以用 instanceof 运算符判断 某个对象是否是 List 接口的实现类，如果是返回 true，否则返回 false

```
1 ArrayList arrayList = new ArrayList();
2 System.out.println(arrayList instanceof List);//true
```

或者反过来也是返回 true

```
1 List list = new ArrayList();
2 System.out.println(list instanceof ArrayList);//true
```

## 5、obj 为 class 类的直接或间接子类

我们新建一个父类 Person.class，然后在创建它的一个子类 Man.class

```
1 public class Person {
2
3 }
```

Man.class

```
1 public class Man extends Person{
2
3 }
```

测试：

```
1 Person p1 = new Person();
2 Person p2 = new Man();
3 Man m1 = new Man();
4 System.out.println(p1 instanceof Man); //false
5 System.out.println(p2 instanceof Man); //true
6 System.out.println(m1 instanceof Man); //true
```

注意第一种情况，p1 instanceof Man，Man 是 Person 的子类，Person 不是 Man 的子类，所以返回结果为 false。

## 6、问题

前面我们说过编译器会检查 obj 是否能转换成右边的class类型，如果不能转换则直接报错，如果不能确定类型，则通过编译，具体看运行时定。

看如下几个例子：

```
1 Person p1 = new Person();
2
3 System.out.println(p1 instanceof String); //编译报错
4 System.out.println(p1 instanceof List); //false
5 System.out.println(p1 instanceof List<?>); //false
6 System.out.println(p1 instanceof List<Person>); //编译报错
```

按照我们上面的说法，这里就存在问题了，Person 的对象 p1 很明显不能转换为 String 对象，那么自然 Person 的对象 p1 instanceof String 不能通过编译，但为什么 p1 instanceof List 却能通过编译呢？而 instanceof List 又不能通过编译了？

## 7、深究原理

我们可以看Java语言规范Java SE 8 版：<https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.20.2>

### 15.20.2. Type Comparison Operator instanceof

The type of the *RelationalExpression* operand of the `instanceof` operator must be a reference type or the null type; otherwise, a compile-time error occurs.

It is a compile-time error if the *ReferenceType* mentioned after the `instanceof` operator does not denote a reference type that is reliable (§4.7).

If a cast (§15.16) of the *RelationalExpression* to the *ReferenceType* would be rejected as a compile-time error, then the `instanceof` relational expression likewise produces a compile-time error. In such a situation, the result of the `instanceof` expression could never be true.

At run time, the result of the `instanceof` operator is true if the value of the *RelationalExpression* is not null and the reference could be cast to the *ReferenceType* without raising a `ClassCastException`. Otherwise the result is false.

#### Example 15.20.2-1. The instanceof Operator

```
class Point { int x, y; }
class Element { int atomicNumber; }
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        Element e = new Element();
        if (e instanceof Point) { // compile-time error
            System.out.println("I get your point!");
            p = (Point)e; // compile-time error
        }
    }
}
```

This program results in two compile-time errors. The cast `(Point)e` is incorrect because no instance of `Element` or any of its possible subclasses (none are shown here) could possibly be an instance of any subclass of `Point`. The `instanceof` expression is incorrect for exactly the same reason. If, on the other hand, the class `Point` were a subclass of `Element` (an admittedly strange notion in this example),

```
class Point extends Element { int x, y; }
```

then the cast would be possible, though it would require a run-time check, and the `instanceof` expression would then be sensible and valid. The cast `(Point)e` would never raise an exception because it would not be executed if the value of `e` could not correctly be cast to type `Point`.

如果用伪代码描述：

```
1  boolean result;
2  if (obj == null) {
3      result = false;
4  } else {
5      try {
6          T temp = (T) obj; // checkcast
7          result = true;
8      } catch (ClassCastException e) {
9          result = false;
10     }
11 }
```

也就是说有表达式 `obj instanceof T`，`instanceof` 运算符的 `obj` 操作数的类型必须是引用类型或空类型；否则，会发生编译时错误。

如果 `obj` 强制转换为 `T` 时发生编译错误，则关系表达式的 `instanceof` 同样会产生编译时错误。在这种情况下，表达式实例的结果永远为 false。

在运行时，如果 `T` 的值不为 null，并且 `obj` 可以转换为 `T` 而不引发 `ClassCastException`，则 `instanceof` 运算符的结果为 true。否则结果是错误的

简单来说就是：如果 `obj` 不为 null 并且 `(T) obj` 不抛 `ClassCastException` 异常则该表达式值为 true，否则值为 false。

所以对于上面提出的问题就很好理解了，为什么 `p1 instanceof String` 编译报错，因为 `(String)p1` 是不能通过编译的，而 `(List)p1` 可以通过编译。

## 8、instanceof 的实现策略

JavaSE 8 instanceof 的实现算法：<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5 instanceof>

- If S is an ordinary (nonarray) class, then:
  - If T is a class type, then S must be the same class as T, or S must be a subclass of T;
  - If T is an interface type, then S must implement interface T.
- If S is an interface type, then:
  - If T is a class type, then T must be Object.
  - If T is an interface type, then T must be the same interface as S or a superinterface of S.
- If S is a class representing the array type SC[], that is, an array of components of type SC, then:
  - If T is a class type, then T must be Object.
  - If T is an interface type, then T must be one of the interfaces implemented by arrays (JLS §4.10.3).
  - If T is an array type TC[], that is, an array of components of type TC, then one of the following must be true:
    - TC and SC are the same primitive type.
    - TC and SC are reference types, and type SC can be cast to TC by these run-time rules.

1、obj如果为null，则返回false；否则设S为obj的类型对象，剩下的问题就是检查S是否为T的子类型；

2、如果S == T，则返回true；

3、接下来分为3种情况，之所以要分情况是因为instanceof要做的是“子类型检查”，而Java语言的类型系统里数组类型、接口类型与普通类类型三者的子类型规定都不一样，必须分开来讨论。

①、S是数组类型：如果 T 是一个类类型，那么T必须是Object；如果 T 是接口类型，那么 T 必须是由数组实现的接口之一；

②、接口类型：对接口类型的 instanceof 就直接遍历S里记录的它所实现的接口，看有没有跟T一致的；

③、类类型：对类类型的 instanceof 则是遍历S的super链（继承链）一直到Object，看有没有跟T一致的。遍历类的super链意味着这个算法的性能会受类的继承深度的影响。

参考链接：<https://www.zhihu.com/question/21574535>

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别筛选的书籍资料



## Java关键字(二)——native

本篇博客我们将介绍Java中的一个关键字——native。

native 关键字在 JDK 源码中很多类中都有，在 `Object.java` 类中，其 `getClass()` 方法、`hashCode()` 方法、`clone()` 方法等等都是用 native 关键字修饰的。

```
1 public final native Class<?> getClass();
2 public native int hashCode();
3 protected native Object clone() throws CloneNotSupportedException;
```

那么为什么要用 native 来修饰方法，这样做有什么用？

### 1、JNI：Java Native Interface

在介绍 native 之前，我们先了解什么是 JNI。

一般情况下，我们完全可以使用 Java 语言编写程序，但某些情况下，Java 可能会不满足应用程序的需求，或者是不能更好的满足需求，比如：

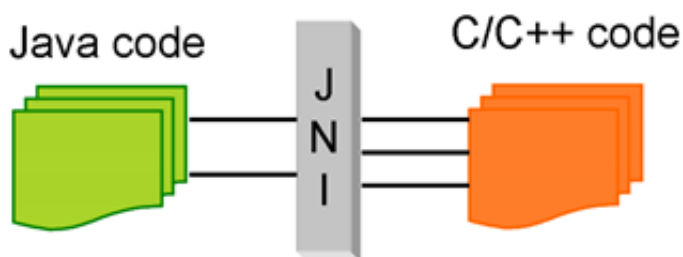
- ①、标准的 Java 类库不支持应用程序平台所需的平台相关功能。
- ②、我们已经用另一种语言编写了一个类库，如何用Java代码调用？



③、某些运行次数特别多的方法代码，为了加快性能，我们需要用更接近硬件的语言（比如汇编）编写。

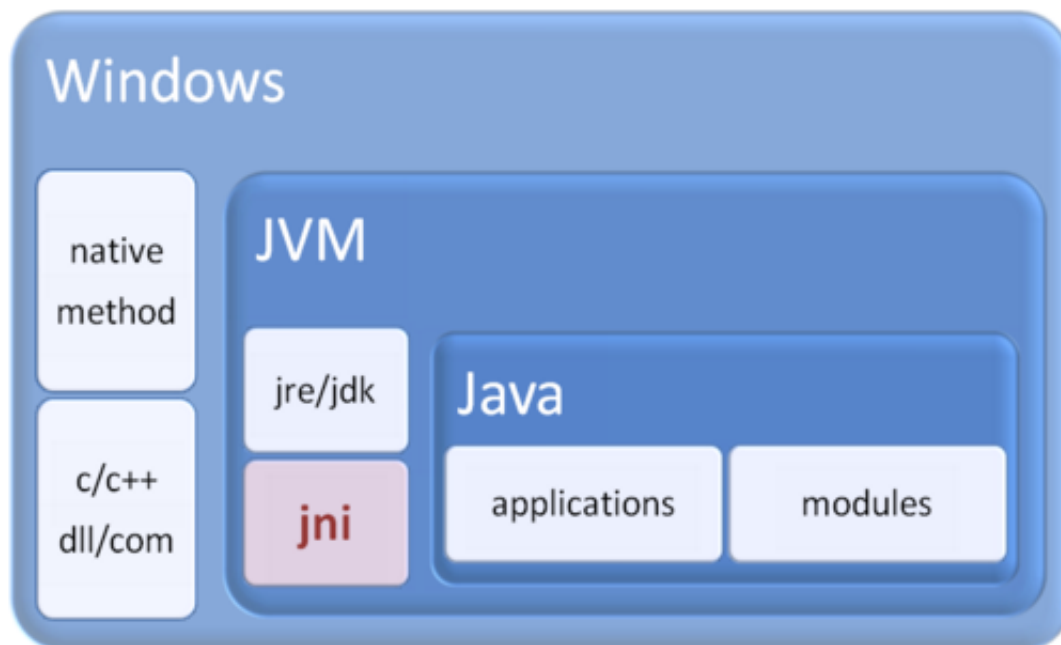
上面这三种需求，其实说到底就是如何用 Java 代码调用不同语言编写的代码。那么 JNI 应运而生了。

从Java 1.1开始，Java Native Interface (JNI)标准就成为java平台的一部分，它允许Java代码和其他语言写的代码进行交互。JNI一开始是为了本地已编译语言，尤其是C和C++而设计的，但是它并不妨碍你使用其他语言，只要调用约定受支持就可以了。使用java与本地已编译的代码交互，通常会丧失平台可移植性。但是，有些情况下这样做是可以接受的，甚至是必须的，比如，使用一些旧的库，与硬件、操作系统进行交互，或者为了提高程序的性能。JNI标准至少保证本地代码能工作在任何Java 虚拟机实现下。



通过 JNI，我们就可以通过 Java 程序（代码）调用到操作系统相关的技术实现的库函数，从而与其他技术和系统交互，使用其他技术实现的系统的功能；同时其他技术和系统也可以通过 JNI 提供的相应原生接口开调用 Java 应用系统内部实现的功能。

在windows系统上，一般可执行的应用程序都是基于 native 的PE结构，windows上的 JVM 也是基于native结构实现的。Java应用体系都是构建于 JVM 之上。



可能有人会问，Java不是跨平台的吗？如果用 JNI，那么程序不就将失去跨平台的优点？确实是这样的。

JNI 的缺点：

①、程序不再跨平台。要想跨平台，必须不同的系统环境下重新编译本地语言部分。



②、程序不再是绝对安全的，本地代码的不当使用可能导致整个程序崩溃。一个通用规则是，你应该让本地方法集中在少数几个类当中。这样就降低了JAVA和C之间的耦合性。

目前来讲使用 JNI 的缺点相对于优点还是可以接受的，可能后面随着 Java 的技术发展，我们不再需要 JNI，但是目前 JDK 还是一直提供对 JNI 标准的支持。

### 3、用C语言编写程序本地方法

上面讲解了什么是 JNI，那么我们接下来就写个例子，如何用 Java 代码调用本地的 C 程序。

官方文档如下：<https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html>

步骤如下：

①、编写带有 native 声明的方法的java类，生成.java文件；(注意这里出现了 native 声明的方法关键字)

②、使用 javac 命令编译所编写的java类，生成.class文件；

③、使用 javah -jni java类名 生成扩展名为 h 的头文件，也即生成.h文件；

④、使用C/C++（或者其他编程想语言）实现本地方法，创建.h文件的实现，也就是创建.cpp文件实现.h文件中的方法；

⑤、将C/C++编写的文件生成动态连接库，生成dll文件；

下面我们通过一个 HelloWorld 程序的调用来完成这几个步骤。

注意：下面所有操作都是在所有操作都是在目录：D:\JNI 下进行的。

一、编写带有 native 声明的方法的java类

```
1 public class HelloJNI {
2     //native 关键字告诉 JVM 调用的是该方法在外部定义
3     private native void helloJNI();
4
5     static{
6         System.loadLibrary("helloJNI");//载入本地库
7     }
8     public static void main(String[] args) {
9         HelloJNI jni = new HelloJNI();
10        jni.helloJNI();
11    }
12
13 }
```

用 native 声明的方法表示告知 JVM 调用，该方法在外部定义，也就是我们会用 C 语言去实现。

System.loadLibrary("helloJNI");加载动态库，参数 helloJNI 是动态库的名字。我们可以这样理解：程序中的方法 helloJNI() 在程序中没有实现，但是我们下面要调用这个方法，怎么办呢？我们就需要对这个方法进行初始化，所以用 static 代码块进行初始化。



这时候如果我们直接运行该程序，会报“A Java Exception has occurred”错误：

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: no helloJNI in java.library.path
    at java.lang.ClassLoader.loadLibrary(Unknown Source)
    at java.lang.Runtime.loadLibrary0(Unknown Source)
    at java.lang.System.loadLibrary(Unknown Source)
    at com.ys.jni.HelloJNI.<clinit> (HelloJNI.java:8)
```

二、使用 javac 命令编译所编写的java类，生成.class文件

```
D:\JNI>javac HelloJNI.java
```

执行上述命令后，生成 HelloJNI.class 文件：

名称	修改日期	类型	大小
 HelloJNI.class	2018/3/8 0:30	CLASS 文件	1 KB
 HelloJNI.java	2018/3/8 0:30	JAVA 文件	1 KB

三、使用 javah -jni java类名 生成扩展名为 h 的头文件

```
D:\JNI>javah -jni HelloJNI
```

执行上述命令后，在 D:\JNI 目录下多出了个 HelloJNI.h 文件：

名称	修改日期	类型	大小
 HelloJNI.class	2018/3/8 0:30	CLASS 文件	1 KB
 HelloJNI.h	2018/3/8 0:32	C Header File	1 KB
 HelloJNI.java	2018/3/8 0:30	JAVA 文件	1 KB

四、使用C语言实现本地方法

如果不想安装visual studio 的，我们需要在 windows平台安装 gcc。

安装教程如下：<http://blog.csdn.net/altland/article/details/63252757>

注意安装版本的选择，根据系统是32位还是64位来选择。64位点击下载。

安装完成之后注意配置环境变量，在 cmd 中输入 g++ -v，如果出现如下信息，则安装配置完成：

```
D:\JNI>g++ -v
Using built-in specs.
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=c:/mingw/bin/./libexec/gcc/mingw32/6.3.0/lto-wrapper.exe
Target: mingw32
Configured with: ../src/gcc-6.3.0/configure --build=x86_64-pc-linux-gnu --host=mingw32 --target=mingw32 --with-mpc=/mingw --with-isl=/mingw --prefix=/mingw --disable-win32-registry --enable-languages=c,c++,objc,obj-c++,fortran,ada --with-pkgversion='MinGW.org GCC-6.3.0-1' --enable-threads --with-dwarf2 --disable-sjlj-exceptions --enable-version-specific-runtime-libs --with-libintl-prefix=/mingw --enable-libstdcxx-debug --with-tune=generic --enable-nls
Thread model: win32
gcc version 6.3.0 (MinGW.org GCC-6.3.0-1)
```

接着输入如下命令：

```
1 gcc -m64 -Wl,--add-stdcall-alias -I"C:\Program
Files\Java\jdk1.8.0_152\include" -I"C:\Program
Files\Java\jdk1.8.0_152\include\include\win32" -shared -o helloJNI.dll
helloJNI.c
```

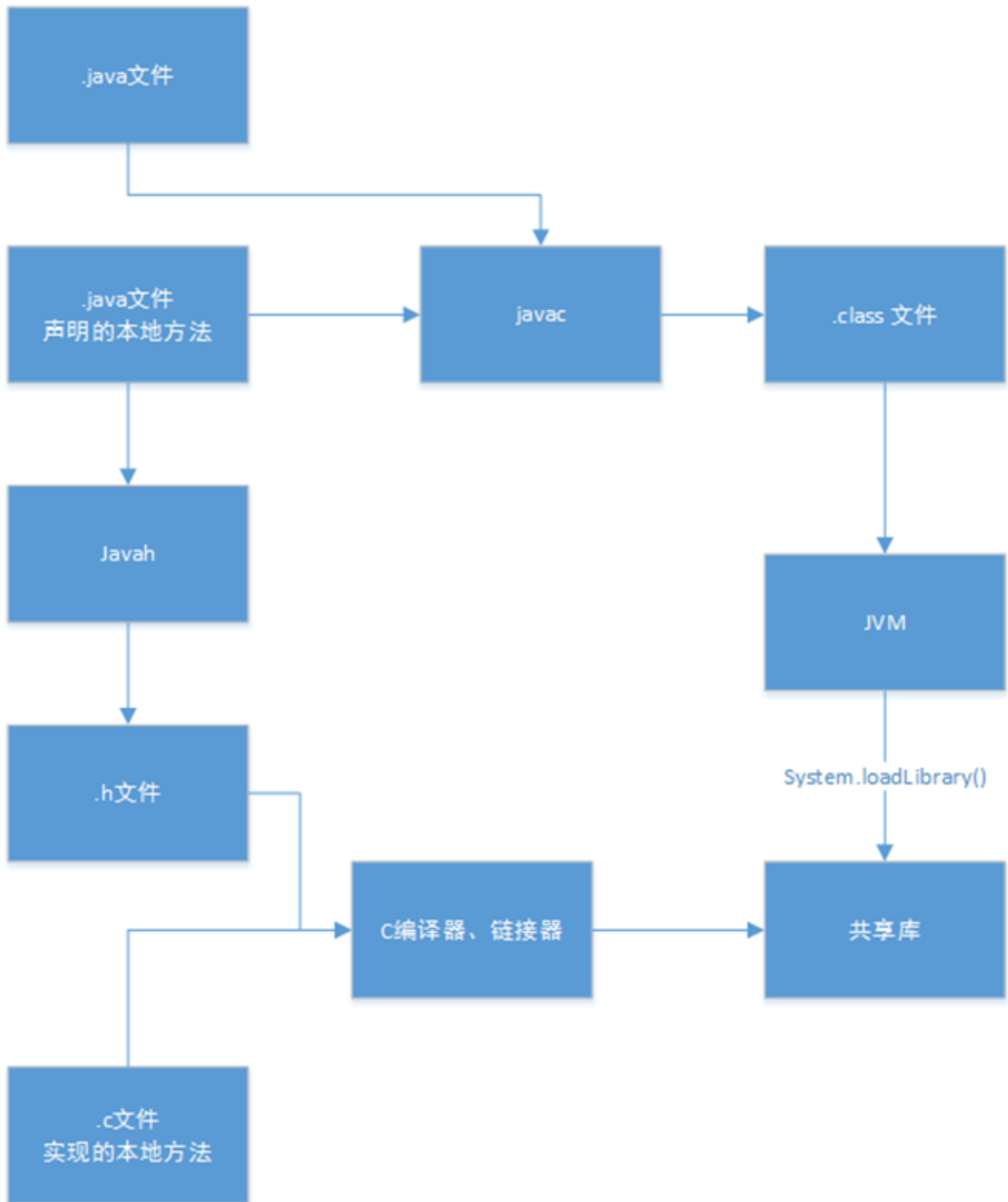
-m64表示生成dll库是64位的。后面的路径表示本机安装的JDK路径。生成之后多了一个helloJNI.dll文件

名称	修改日期	类型	大小
 helloJNI.c	2018/3/8 0:46	C Source File	1 KB
 HelloJNI.class	2018/3/8 0:30	CLASS 文件	1 KB
 helloJNI.dll	2018/3/8 1:45	应用程序扩展	41 KB
 HelloJNI.h	2018/3/8 0:32	C Header File	1 KB
 HelloJNI.java	2018/3/8 0:30	JAVA 文件	1 KB

最后运行 HelloJNI：输出 Hello JNI! 大功告成。

```
D:\JNI>gcc -m64 -Wl,--add-stdcall-alias -I"C:\Program Files\Java\jdk1.8.0_152\include" -I"C:\Program Files\Java\jdk1.8.0_152\include\include\win32" -shared -o helloJNI.dll helloJNI.c
D:\JNI>java HelloJNI
Hello JNI!
D:\JNI>
```

## 4、JNI调用C的流程图



图片引用自: <https://www.cnblogs.com/Qian123/p/5702574.html>

## 5、native关键字

通过上面介绍了那么多JNI的知识，终于到介绍本篇文章的主角——native 关键字了。相信大家看完上面的介绍，应该也是知道什么是 native 了吧。

native 用来修饰方法，用 native 声明的方法表示告知 JVM 调用，该方法在外部定义，我们可以用任何语言去实现它。简单地讲，一个native Method就是一个 Java 调用非 Java 代码的接口。

native 语法：

- ①、修饰方法的位置必须在返回类型之前，和其余的方法控制符前后关系不受限制。
- ②、不能用 abstract 修饰，也没有方法体，也没有左右大括号。
- ③、返回值可以是任意类型

我们在日常编程中看到native修饰的方法，只需要知道这个方法的作用是什么，至于别的就不用管了，操作系统会给我们实现。

参考文档：<https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html>  
<https://www.cnblogs.com/Qian123/p/5702574.html>

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别筛选的书籍资料



## Java关键字(三)——static

我们说Java是一种面向对象编程的语言，而对象是把数据及对数据的操作方法放在一起，作为一个相互依存的整体，对同类对象抽象出其共性，便是Java中的类，我们可以用类描述世间万物，也可以说万物皆对象。但是这里有个特殊的東西——static，它不属于对象，那么为什么呢？

static 是Java的一个关键字，可以用来修饰成员变量、修饰成员方法、构造静态代码块、实现静态导包以及实现静态内部类，下面我们来分别介绍。

# 1、修饰成员变量

用 static 修饰成员变量可以说是该关键字最常用的一个功能，通常将用 static 修饰的成员变量称为类成员或者静态成员，那么静态成员和不用 static 修饰的非静态成员有什么区别呢？

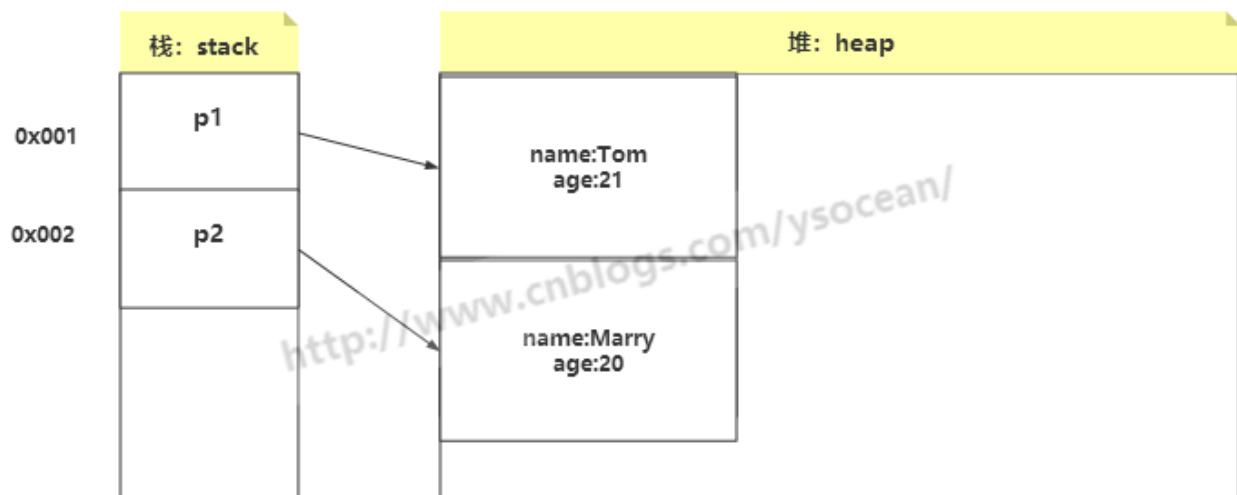
我们先看看不用 static 修饰的成员变量在内存中的构造。

```
1 package com.js.bean;
2
3 /**
4  * Create by YSOcean
5  */
6 public class Person {
7     private String name;
8     private Integer age;
9
10    public Person(String name, Integer age) {
11        this.name = name;
12        this.age = age;
13    }
14
15    @Override
16    public String toString() {
17        return "Person{" +
18            "name='" + name + '\'' +
19            ", age=" + age +
20            '}';
21    }
22    //get和set方法省略
23 }
```

首先，我们创建一个实体类 Person，有两个属性 name 和 age，都是普通成员变量（没有用 static 关键字修饰），接着我们通过其构造方法创建两个对象：

```
1 Person p1 = new Person("Tom",21);
2 Person p2 = new Person("Marry",20);
3 System.out.println(p1.toString()); //Person{name='Tom', age=21}
4 System.out.println(p2.toString()); //Person{name='Marry', age=20}
```

这两个对象在内存中的存储结构如下：



由上图可知，我们创建的两个对象 p1 和 p2 存储在堆中，但是其引用地址是存放在栈中的，而且这两个对象的两个变量互相独立，我们修改任何一个对象的属性值，是不改变另外一个对象的属性值的。

下面我们将 Person 类中的 age 属性改为由 static 关键字修饰：

```
1 package com.js.bean;
2
3 /**
4  * Create by YSOcean
5  */
6 public class Person {
7     private String name;
8     private static Integer age;
9
10    public Person(String name, Integer age) {
11        this.name = name;
12        this.age = age;
13    }
14
15    @Override
16    public String toString() {
17        return "Person{" +
18            "name='" + name + '\'' +
19            ", age=" + age +
20            '}';
21    }
22    //get和set方法省略
23
24 }
```

同样我们还是向上面一样，创建 p1 和 p2 两个对象，并打印这两个对象，看看和上面打印的有啥区别呢？

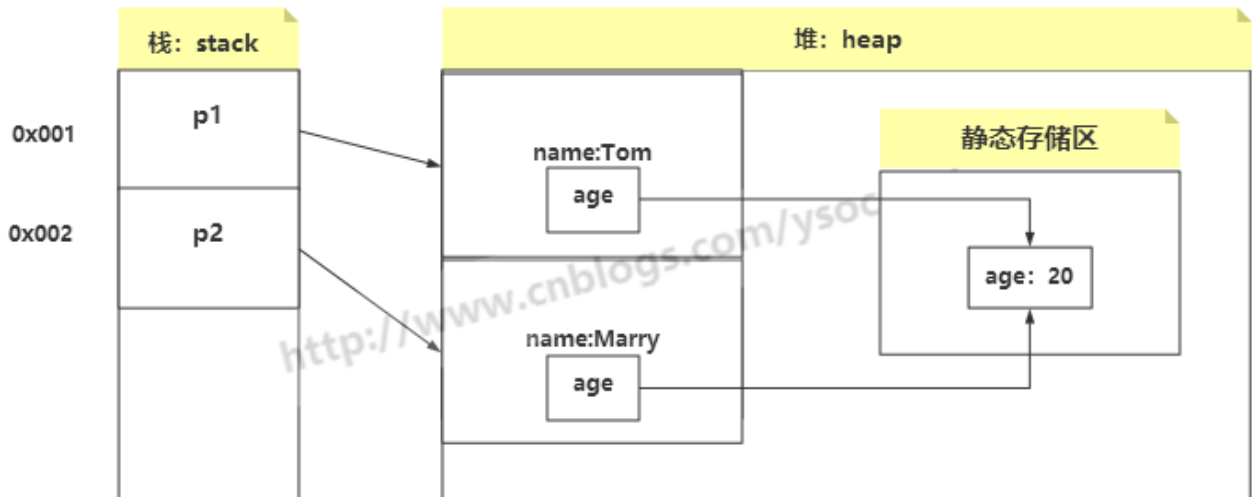


```

1 Person p1 = new Person("Tom",21);
2 Person p2 = new Person("Marry",20);
3 System.out.println(p1.toString()); //Person{name='Tom', age=20}
4 System.out.println(p2.toString()); //Person{name='Marry', age=20}

```

我们发现第三行代码打印的 p1 对象 age 属性变为 20了，这是为什么呢？



这是因为用在 jvm 的内存构造中，会在堆中开辟一块内存空间，专门用来存储用 static 修饰的成员变量，称为静态存储区，无论我们创建多少个对象，用 static 修饰的成员变量有且只有一份存储在静态存储区中，所以该静态变量的值是以最后创建对象时设置该静态变量的值为准，也就是由于 p1 先设置 age = 21，后来创建了 p2 对象，p2 将 age 改为了 20，那么该静态存储区的 age 属性值也被修改成了 20。

PS：在 JDK1.8 以前，静态存储区是存放在方法区的，而方法区不属于堆，在 JDK1.8 之后，才将方法区干掉了，方法区中的静态存储区改为到堆中存储。

总结：static 修饰的变量被所有对象所共享，在内存中只有一个副本。由于与对象无关，所以我们可以直接通过 类名.静态变量 的方式来直接调用静态变量。对应的非静态变量是对象所拥有的，多少个对象就有多少个非静态变量，各个对象所拥有的副本不受影响。

## 2、修饰修饰成员方法

用 static 关键字修饰成员方法也是一样的道理，我们可以直接通过 类名.静态方法名() 的方式来调用，而不用创建对象。

```

1 public class Person {
2     private String name;
3     private static Integer age;
4
5     public static void printClassName(){
6         System.out.println("com.ys.bean.Person");
7     }
8     public Person(String name, Integer age) {
9         this.name = name;
10        this.age = age;

```

```
11     }
12
13     @Override
14     public String toString() {
15         return "Person{" +
16             "name='" + name + '\'' +
17             ", age=" + age +
18             "'}";
19     }
20     //get和set方法省略
21
22 }
```

调用静态方法：

```
1 Person.printClassName(); //com.js.bean.Person
```

### 3、静态代码块

用 static 修饰的代码块称为静态代码块，静态代码块可以置于类的任意一个地方（和成员变量成员方法同等地位，不可放入方法中），并且一个类可以有多个静态代码块，在类初次载入内存时加载静态代码块，并且按照声明静态代码块的顺序来加载，且仅加载一次，优先于各种代码块以及构造函数。

```
1 public class CodeBlock {
2     static{
3         System.out.println("静态代码块");
4     }
5 }
```

由于静态代码块只在类载入内存时加载一次的特性，我们可以利用静态代码块来优化程序性能，比如某个比较大配置文件需要在创建对象时加载，这时候为了节省内存，我们可以将该配置文件的加载时机放入到静态代码块中，那么我们无论创建多少个对象时，该配置文件也只加载了一次。

### 4、静态导包

用 static 来修饰成员变量，成员方法，以及静态代码块是最常用的三个功能，静态导包是 JDK1.5 以后的新特性，用 import static 包名 来代替传统的 import 包名 方式。那么有什么用呢？

比如我们创建一个数组，然后用 JDK 自带的 Arrays 工具类的 sort 方法来对数组进行排序：

```

1 package com.js.test;
2
3 import java.util.Arrays;
4 /**
5  * Create by YSOcean
6  */
7 public class StaticTest {
8
9     public static void main(String[] args) {
10         int[] arrays = {3,4,2,8,1,9};
11         Arrays.sort(arrays);
12     }
13 }

```

我们可以看到，调用 sort 方法时，需要进行 import java.util.Arrays 的导包操作，那么变为静态导包呢？

```

1 package com.js.test;
2
3 import static java.util.Arrays.*;
4 /**
5  * Create by YSOcean
6  */
7 public class StaticTest {
8
9     public static void main(String[] args) {
10         int[] arrays = {3,4,2,8,1,9};
11         sort(arrays);
12     }
13 }

```

我们可以看到第三行代码的 import java.util.Arrays 变为了 import static java.util.Arrays.\*，意思是导入 Arrays 类中的所有静态方法，当然你也可以将 \* 变为某个方法名，也就是只导入该方法，那么我们在调用该方法时，就可以不带上类名，直接通过方法名来调用（第 11 行代码）。

静态导包只会减少程序员的代码编写量，对于性能是没有任何提升的（也不会降低性能，Java核心技术第10版卷1第148页4.7.1章节类的导入有介绍），反而会降低代码的可读性，所以实际如何使用需要权衡。

## 5、静态内部类

首先我们要知道什么是内部类，定义在一个类的内部的类叫内部类，包含内部类的类叫外部类，内部类用 static 修饰便是我们所说的静态内部类。

定义内部类的好处是外部类可以访问内部类的所有方法和属性，包括私有方法和私有属性。

访问普通内部类，我们需要先创建外部类的对象，然后通过外部类名.new 创建内部类的实例。

```

1 package com.js.bean;
2
3 /**
4  * Create by hadoop
5  */
6 public class OutClass {
7
8     public class InnerClass{
9
10    }
11 }

```

```

1 * OuterClass oc = new OuterClass();
2 * OuterClass.InnerClass in = oc.new InnerClass();

```

访问静态内部类，我们不需要创建外部类的对象，可以直接通过 外部类名.内部类名 来创建实例。

```

1 package com.js.bean;
2
3 /**
4  * Create by hadoop
5  */
6 public class OutClass {
7
8     public static class InnerClass{
9
10    }
11 }

```

```

1 OuterClass.StaticInnerClass sic = new OuterClass.StaticInnerClass();

```

## 6、常见问题

### ①、静态变量能存在于普通方法中吗？

能。很明显，普通方法必须通过对象来调用，静态变量都可以直接通过类名来调用了，更不用说通过对象来调用，所以是可以存在于普通方法中的。

### ②、静态方法能存在普通变量吗？

不能。因为静态方法可以直接通过类名来直接调用，不用创建对象，而普通变量是必须通过对象来调用的。那么将普通变量放在静态方法中，在直接通过类来调用静态方法时就会报错。所以不能。

### ③、静态代码块能放在方法体中吗？

不能。首先我们要明确静态代码块是在类加载的时候自动运行的。

普通方法需要我们创建对象，然后手工去调用方法，所以静态代码块不能声明在普通方法中。

那么对于用 static 修饰的静态方法呢？同样也是不能的。因为静态方法同样也需要我们手工通过类名来调用，而不是直接在类加载的时候就运行了。

也就是说静态代码块能够自动执行，而不管是普通方法还是静态方法都是需要手工执行的。

④、静态导包会比普通导包消耗更多的性能？

不会。静态导包实际上在编译期间都会被编译器进行处理，将其转换成普通按需导包的形式，所以在程序运行期间是不影响性能的。

⑤、static 可以用来修饰局部变量吗？

不能。不管是在普通方法还是在静态方法中，static 关键字都不能用来修饰局部变量，这是Java的规定。稍微想想也能明白，局部变量的声明周期是随着方法的结束而结束的，因为static 修饰的变量是全局的，不与对象有关的，如果用 static 修饰局部变量容易造成理解上的冲突，所以Java规定 static 关键字不能用来修饰局部变量。

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别筛选的书籍资料



## Java关键字(四)——final

对于Java中的 final 关键字，我们首先可以从字面意思上去理解，百度翻译显示如下：

## final

英 ['faɪnəl]   美 ['faɪnəl]  

adj. 最后的，最终的；决定性的；不可更改的

n. 决赛；结局；期末考试；〈口〉（报纸的）末版

复数： [finals](#)

记忆技巧：fin 结束 + al ...的 → 最后的；决定性的

高考

CET4

考研

也就是说 final 英文意思表示是最后的，不可更改的。那么对应 in Java 中也是表达这样的意思，可以用 final 关键字修饰变量、方法和类。不管是用来修饰什么，其本意都是指“它是无法更改的”，这是我们需要牢记的，为什么要无法更改？无非就是设计所需或者能提高效率，与前面介绍 static 关键字需要记住其与对象无关的理念一样，牢记 final 的不可变的设计理念后再来了解 final 关键字的用法，便会顺其自然了。

## 1、修饰变量

稍微有点Java基础的都知道用final关键字修饰的变量称为常量，常量的意思是不可更改。变量为基本数据类型，不可更改很容易理解，那么对于引用类型呢？不可能改的是其引用地址，还是对象的内容？

我们首先构造一个实体类：Person

```
1 package com.js.bean;
2
3 /**
4  * Create by YSOcean
5  */
6 public class Person {
7     private String name;
8
9     public Person(String name) {
10         this.name = name;
11     }
12
13     public String getName() {
14         return name;
15     }
16
17     public void setName(String name) {
18         this.name = name;
19     }
20 }
```

接着根据创建一个 Person 对象：

```

8  ▶ public class FinalTest {
9
10 ▶ public static void main(String[] args) {
11     final Person p = new Person( name: "Tom" );
12     p = new Person( name: "Marry" );
13
14 }
15

```

Cannot assign a value to final variable 'p'

可以看到，首先通过 final 关键字修饰一个对象 p，然后接着将 p 对象指向另一个新的对象，发现报错，也就是说final修饰的引用类型是不能改变其引用地址的。

接着我们改动 p 对象的 name 属性：

```

8  ▶ public class FinalTest {
9
10 ▶ public static void main(String[] args) {
11     final Person p = new Person( name: "Tom" );
12     p.setName("Marry");
13 }
14 }

```

发现程序没有报错。

结论：被 final 修饰的变量不可更改其引用地址，但是可以更改其内部属性。

## 2、修饰方法

final 关键字修饰的方法不可被覆盖。

在《Java编程思想》第4版 7.8.2 章节 final 方法p176 页这样描述：使用 final 方法原因有两个：

①、第一个原因是把方法锁定，以防止任何继承类修改它的含义，这是出于设计的考虑：想要确保在继承中使方法的行为保持不变，并且不会被覆盖。

②、第二个原因是效率，在 Java 的早期实现中，如果将一个方法声明为 final，就是同意编译器将针对该方法的所有调用都转为内嵌调用，内嵌调用能够提高方法调用效率，但是如果方法很大，内嵌调用不会提高性能。而在目前的Java版本中（JDK1.5以后），虚拟机可以自动进行优化了，而不需要使用 final 方法。

所以final 关键字只有明确禁止覆盖方法时，才使用其修饰方法。



PS: 《Java编程思想》中指出类中所有的 private 方法都隐式指定为 final 的, 所以对于 private 方法, 我们显式的声明 final 并没有什么效果。但是我们创建一个父类, 并在父类中声明一个 private 方法, 其子类中是能够重写其父类的private 方法的, 这是为什么呢?

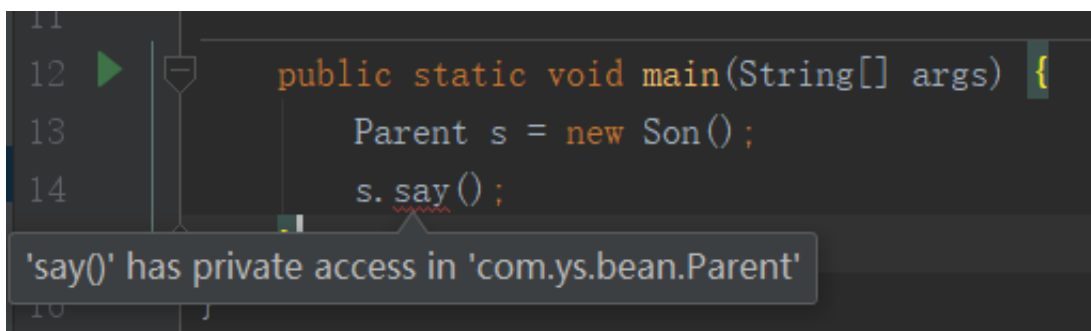
父类: Parent.class

```
1 package com.ys.bean;
2 /**
3  * Create by YSOcean
4  */
5 public class Parent {
6     private void say(){
7         System.out.println("parent");
8     }
9 }
```

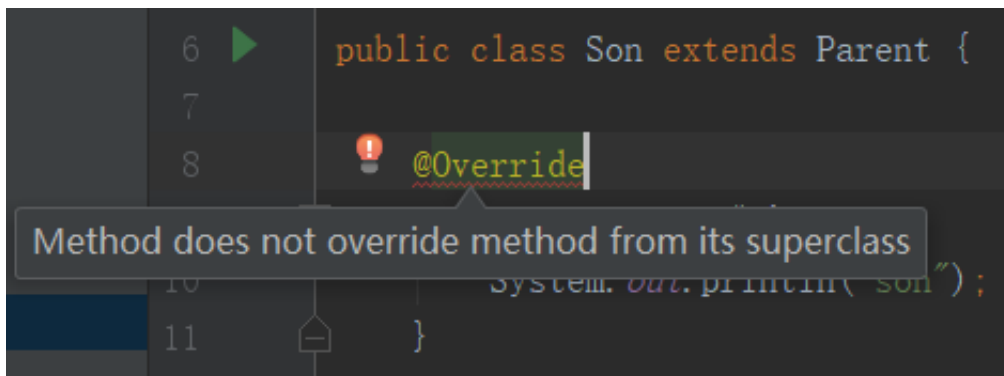
子类: Son.class

```
1 package com.ys.bean;
2 /**
3  * Create by YSOcean
4  */
5 public class Son extends Parent {
6
7     private void say(){
8         System.out.println("son");
9     }
10
11 }
```

其实仔细看看, 这种写法是方法的覆盖吗? 我们通过多态的形式并不能调用到父类的 say() 方法:



并且, 如果我们在子类的 say() 方法中, 添加 @Override 注解也是会报错的。



所以这种形式并不算方法的覆盖。

### 3、修饰类

`final` 修饰类表示该类不可被继承。

也就是说不希望某个类有子类的时候，用 `final` 关键字来修饰。并且由于是用 `final` 修饰的类，其类中所有的方法也被隐式的指为 `final` 方法。

在 JDK 中有个最明显的类 `String`，就是用 `final` 修饰的，将 `String` 类用 `final` 修饰很重要的一个原因是常量池。关于 `String` 类的描述，可以参考我前面文章。

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

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别筛选的书籍资料



## Java关键字(五)——this

this 也是Java中的一个关键字，在《Java编程思想》第四版第五章5.4小节对 this 关键字是这样介绍的：

this 关键字只能在方法内部使用，表示对“调用方法的那个对象”的引用。

其实简单来说 this 关键字就是表示当前对象，下面我们来具体介绍 this 关键字在Java中的用法。

### 1、调用成员变量

在一个类的方法内部，如果我们想调用其成员变量，不用 this，我们会怎么做？

```
1 package com.ys.test;
2
3 /**
4  * Create by YSOcean
5  */
6 public class ThisTest {
7
8     private String name = "Tom";
9
10    public void setName(String name){
11        name = name;
12    }
13
14    public String getName() {
15        return name;
16    }
17
18 }
```

看上面的代码，我们在 ThisTest 类中创建了一个 name 属性，然后创建了一个 setName 方法，注意这个方法的形参也是 String name，那么我们通过 name = name 这样赋值，会改变成员变量 name 的属性吗？

```
1 public static void main(String[] args) {
2     ThisTest tt = new ThisTest();
3     tt.setName("Marry");
4     System.out.println(tt.getName()); //Tom
5 }
```

打印结果是 Tom，而不是我们重新设置的 Marry，显然这种方式是不能在方法内部调用到成员变量的。因为形参的名字和成员变量的名字相同，setName 方法内部的 name = name，根据最近原则，编译器默认是将这两个 name 属性都解析为形参 name，从而导致我们设值操作和成员变量 name 完全没有关系，当然设置不了。

解决办法就是使用 this 关键字。我们将 setName 方法修改如下：

```
1 public void setName(String name){
2     this.name = name;
3 }
```

在调用上面的 main 方法进行赋值，打印的结果就是 Marry 了。

this 表示当前对象，也就是调用该方法的对象，对象.name 肯定就是调用的成员变量。

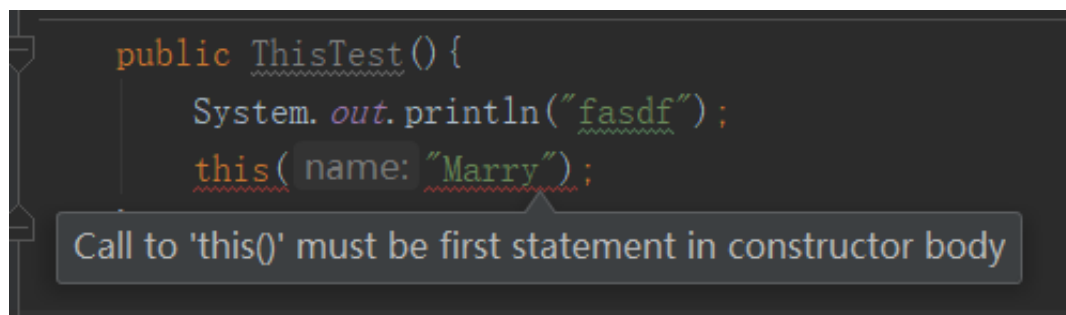
## 2、调用构造方法

构造方法是与类同名的一个方法，构造方法没有返回值，但是也不能用 void 来修饰。在一个类中，必须存在一个构造方法，如果没有，编译器会在编译的时候自动为这个类添加一个无参构造方法。一个类能够存在多个构造方法，调用的时候根据参数来区分。

```
1 package com.js.test;
2
3 /**
4  * Create by YSOcean
5  */
6 public class ThisTest {
7     private String name;
8
9     public ThisTest(){
10         this("Marry");
11     }
12     public ThisTest(String name) {
13         this.name = name;
14     }
15 }
```

通过 this("Marry") 来调用另外一个构造方法 ThisTest(String name) 来给成员变量初始化赋值。

注意：通过 this 来调用构造方法，只能将这条代码放在构造函数的第一行，这是编译器的规定，如下所示：放在第二行会报错。



### 3、调用普通方法

this 表示当前对象，那么肯定能够调用当前类的普通方法。

```
1 public void printName(){
2     this.say();
3 }
4
5 public void say(){
6     System.out.println("say method...");
7 }
```

第 2 行代码，在 printName() 方法内部调用了 say() 方法。

### 4、返回当前对象

```
1 /**
2  * Create by YSOcean
3  */
4 public class ThisTest {
5
6     public Object newObject(){
7         return this;
8     }
9 }
```

这表示的意思是谁调用 newObject() 方法，那么就返回谁的引用。

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别筛选的书籍资料



## Java关键字(六)——super

在Java关键字(五)——this 中我们说 this 关键字是表示当前对象的引用。而Java中的super关键字则是表示父类对象的引用。

我们分析这句话“父类对象的引用”，那说明我们使用的时候只能在子类中使用，既然是对象的引用，那么我们也可以用来调用成员属性以及成员方法，当然了，这里的super关键字还能够调用父类的构造方法。具体有如下几种用法：

### 1、调用父类的构造方法

Java中的继承大家都应该了解，子类继承父类，我们是能够用子类的对象调用父类的属性和方法的，我们知道属性和方法只能够通过对象调用，那么我们可以大胆假设一下：

在创建子类对象的同时，也创建了父类的对象，而创建对象是通过调用构造函数实现的，那么我们在创建子类对象的时候，应该会调用父类的构造方法。

下面我们看这段代码：

```
1 public class Parent {
2
3     public Parent(){
4         System.out.println("父类默认无参构造方法");
5     }
6 }
7
8
9 public class Son extends Parent {
10
11     public Son(){
12         System.out.println("子类默认无参构造方法");
13     }
14 }
```

下面我们创建子类的对象：

```
1     public static void main(String[] args) {
2         Son son = new Son();
3     }
```

打印结果：

```
Run Son
"C:\Program Files\Java\jdk1.8.0_152\bin\java" ...
父类默认无参构造方法
子类默认无参构造方法
Process finished with exit code 0
```

通过打印结果看到我们在创建子类对象的时候，首先调用了父类的构造方法，接着调用子类的构造方法，也就是说在创建子类对象的时候，首先创建了父类对象，与前面我们猜想的一致。

那么问题又来了：是在什么时候调用的父类构造方法呢？

可以参考Java官方文档：<https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#d5e1427>

8

#### 8.8.9. Default Constructor

If a class contains no constructor declarations, then a default constructor is implicitly declared. The form of the default constructor for a top level class, member class, or local class is as follows:

- The default constructor has the same accessibility as the class (§8.6).
- The default constructor has no formal parameters, except in a non-private inner member class, where the default constructor implicitly declares one formal parameter representing the immediately enclosing instance of the class (§8.8.1 §15.9.2 §15.9.3).
- The default constructor has no throws clauses.

• If the class being declared is the primordial class `Object`, then the default constructor has an empty body. Otherwise, the default constructor simply invokes the superclass constructor with no arguments.

The form of the default constructor for an anonymous class is specified in §15.9.5.1.

It is a compile-time error if a default constructor is implicitly declared but the superclass does not have an accessible constructor that takes no arguments and has no throws clause.

红色框内的英文翻译为：如果声明的类是原始类Object，那么默认的构造函数有一个空的主体。否则，默认构造函数只是简单地调用没有参数的超类构造函数。

也就是说除了顶级类 `Object.class` 构造函数没有调用父类的构造方法，其余的所有类都默认在构造函数中调用了父类的构造函数（没有显式声明父类的子类其父类是 `Object`）。

那么是通过什么来调用的呢？我们接着看官方文档：

#### 8.8.7.1. Explicit Constructor Invocations

```
ExplicitConstructorInvocation:
    [TypeArguments] this ( [ArgumentList] ) ;
    [TypeArguments] super ( [ArgumentList] ) ;
    ExpressionName . [TypeArguments] super ( [ArgumentList] ) ;
    Primitive . [TypeArguments] super ( [ArgumentList] ) ;
```

The following productions from §4.5.1 and §15.12 are shown here for convenience.

```
TypeArguments:
    < [TypeArgumentList] >

ArgumentList:
    Expression [ , Expression ]
```

Explicit constructor invocation statements are divided into two kinds:

- Alternate constructor invocations begin with the keyword `this` (possibly prefaced with explicit type arguments). They are used to invoke an alternate constructor of the same class.
- Superclass constructor invocations begin with either the keyword `super` (possibly prefaced with explicit type arguments) or a *Primary* expression or an *ExpressionName*. They are used to invoke a constructor of the direct superclass. They are further divided:
  - Unqualified superclass constructor invocations begin with the keyword `super` (possibly prefaced with explicit type arguments).
  - Qualified superclass constructor invocations begin with a *Primary* expression or an *ExpressionName*. They allow a subclass constructor to explicitly specify the newly created object's immediately enclosing instance with respect to the direct superclass (§8.1.3). This may be necessary when the superclass is an inner class.

An explicit constructor invocation statement in a constructor body may not refer to any instance variables or instance methods or inner classes declared in this class or any superclass, or use `this` or `super` in any expression; otherwise, a compile-time error occurs.

上面的意思大概就是超类构造函数通过 `super` 关键字调用，并且是以 `super` 关键字开头。

所以上面的 `Son` 类的构造方法实际上应该是这样的：



```
public class Son extends Parent {  
    public Son() {  
        super();  
        System.out.println("子类默认无参构造方法");  
    }  
}  
  
public static void main(String[] args) {  
    Son son = new Son();  
}
```

子类通过super调用父类构造方法

①、子类默认是通过 super() 调用父类的无参构造方法，如果父类显示声明了一个有参构造方法，而没有声明无参构造方法，实例化子类是会报错的。

```
1 public class Parent {  
2  
3     public Parent(String name){  
4         System.out.println("父类有参构造方法");  
5     }  
6 }  
7  
8 public class Son extends Parent {  
9  
10    public Son(){  
11        System.out.println("子类默认无参构造方法");  
12    }  
13  
14    public static void main(String[] args) {  
15        Son son = new Son();  
16    }  
17  
18 }
```

上面代码是会报错的：

```
7  
8 public Son() {  
    System.out.println("子类默认无参构造方法");  
}
```

There is no default constructor available in 'com.js.bean.Parent'

解决办法就是通过 super 关键字调用父类的有参构造方法：

```

1 public class Son extends Parent {
2
3     public Son(){
4         super("Tom");
5         System.out.println("子类默认无参构造方法");
6     }
7
8     public static void main(String[] args) {
9         Son son = new Son();
10    }
11
12 }

```

注意看第 4 行代码，同理，多个参数也是这种调法。

## 2、调用父类的成员属性

```

1 public class Parent {
2     public String name;
3
4     public Parent(){
5         System.out.println("父类默认无参构造方法");
6     }
7 }
8
9 public class Son extends Parent {
10
11     public Son(){
12         System.out.println("子类默认无参构造方法");
13     }
14
15     public void printName(){
16         System.out.println(super.name);
17     }
18
19 }

```

第 16 行代码 `super.父类属性` 通过这种形式来调用父类的属性。

## 3、调用父类的方法

```

1 public class Parent {
2     public String name;
3
4     public Parent(){
5         System.out.println("父类默认无参构造方法");
6     }

```

```

7
8     public void setName(String name){
9         this.name = name;
10    }
11 }
12
13 public class Son extends Parent {
14
15     public Son(){
16         super();//1、调用父类构造函数
17         System.out.println("子类默认无参构造方法");
18     }
19
20     public void printName(){
21         super.setName("Tom");//2、调用父类方法
22         System.out.println(super.name);//3、调用父类属性
23     }
24
25     public static void main(String[] args) {
26         Son son = new Son();
27         son.printName();//Tom
28     }
29
30 }

```

这个例子我们在子类中分别调用了父类的构造方法、普通方法以及成员属性。

## 4、this 和 super 出现在同一个构造方法中？

不能!!!

在上一篇博客对 this 关键字的介绍中，我们知道能够通过 this 关键字调用自己的构造方法。而本篇博客介绍 super 关键字，我们知道了能够通过 super 调用父类的构造方法，那么这两个关键字能同时出现在子类的构造方法中吗？

①、假设 super() 在 this() 关键字的前面

首先通过 super() 调用父类构造方法，对父类进行一次实例化。接着调用 this()，this() 方法会调用子类的构造方法，在子类的构造方法中又会对父类进行一次实例化。也就是说我们对子类进行一次实例化，对造成对父类进行两次实例化，所以显然编译器是不允许的。

```

1 public class Parent {
2     public String name;
3
4     public Parent(){
5         System.out.println("父类默认无参构造方法");
6     }
7
8     public Parent(String name){
9         System.out.println("父类有参构造方法");

```

```
10     }
11
12 }
13
14 public class Son extends Parent {
15
16     public Son(){
17         super(); //1、调用父类构造函数
18         this("Tom"); //2、调用子类构造方法
19         System.out.println("子类默认无参构造方法");
20     }
21
22     public Son(String name){
23         System.out.println("子类有参构造方法");
24     }
25
26 }
```

反过来 this() 在 super() 之前也是一样。

而且编译器有限定 this() 和 super() 这两个关键字都只能出现在构造方法的第一行，将这两个关键字放在一起，总有一个关键字在第二行，编译是不能通过的。

本系列教程持续更新，可以微信搜索「IT可乐」第一时间阅读。回复《电子书》有我为大家特别筛选的书籍资料



# Java关键字(七)——synchronized

synchronized 这个关键字，我相信对于并发编程有一定了解的人，一定会特别熟悉，对于一些可能在多线程环境下可能会有并发问题的代码，或者方法，直接加上synchronized，问题就搞定了。

但是用归用，你明白它为什么要这么用？为什么就能解决我们所说的线程安全问题？

下面，可乐将和大家一起深入的探讨这个关键字用法。

## 1、求示例代码结果

首先大家看一段代码，大家想想最后的打印count结果是多少？

```
1 package
2 com.js.algorithmproject.leetcode.demo.concurrency.synchronizedtest;
3
4 /**
5  * Create by ItCoke
6  */
7 public class SynchronizedTest implements Runnable{
8
9     public static int count = 0;
10
11     @Override
12     public void run() {
13         addCount();
14
15     }
16
17     public void addCount(){
18         int i = 0;
19         while (i++ < 100000) {
20             count++;
21         }
22     }
23
24     public static void main(String[] args) throws Exception{
25         SynchronizedTest obj = new SynchronizedTest();
26         Thread t1 = new Thread(obj);
27         Thread t2 = new Thread(obj);
28         t1.start();
29         t2.start();
30         t1.join();
31         t2.join();
32         System.out.println(count);
33
34     }
35 }
```

```
36
37 }
```

代码很简单，主线程中启动两个线程t1和t2，分别调用 addCount() 方法，将count的值都加100000，然后调用 join() 方法，表示主线程等待这两个线程执行完毕。最后打印 count 的值。

应该没有答案一定是 200000 的同学吧，很好，大家都具备一定的并发知识。

这题的答案是一定小于等于 200000，至于原因也很好分析，比如 t1线程获取count的值为0，然后执行了加1操作，但是还没来得及同步到主内存，这时候t2线程去获取主内存的count值，发现还是0，然后继续自己的加1操作。也就是t1和t2都执行了加1操作，但是最后count的值依然是1。

那么我们应该如何保证结果一定是 200000呢？答案就是用 synchronized。

## 2、修饰代码块

直接上代码：

```
1 package
2 com.js.algorithmproject.leetcode.demo.concurrency.synchronizedtest;
3
4 /**
5  * Create by ItCoke
6  */
7 public class SynchronizedTest implements Runnable{
8
9     public static int count = 0;
10
11     private Object objMonitor = new Object();
12
13     @Override
14     public void run() {
15         addCount();
16
17     }
18
19     public void addCount(){
20         synchronized (objMonitor){
21             int i = 0;
22             while (i++ < 100000) {
23                 count++;
24             }
25         }
26
27     }
28
29     public static void main(String[] args) throws Exception{
30         SynchronizedTest obj = new SynchronizedTest();
31         Thread t1 = new Thread(obj);
```

```

32         Thread t2 = new Thread(obj);
33         t1.start();
34         t2.start();
35         t1.join();
36         t2.join();
37         System.out.println(count);
38
39     }
40
41
42 }

```

我们在 addCount 方法体中增加了一个 synchronized 代码块，将里面的 while 循环包括在其中，保证同一时刻只能有一个线程进入这个循环去改变count的值。

### 3、修饰普通方法

```

1  package
2  com.ys.algorithmproject.leetcode.demo.concurrency.synchronizedtest;
3
4  /**
5   * Create by ItCoke
6   */
7  public class SynchronizedTest implements Runnable{
8
9      public static int count = 0;
10
11     private Object objMonitor = new Object();
12
13     @Override
14     public void run() {
15         addCount();
16
17     }
18
19     public synchronized void addCount(){
20         int i = 0;
21         while (i++ < 100000) {
22             count++;
23         }
24
25     }
26
27     public static void main(String[] args) throws Exception{
28         SynchronizedTest obj = new SynchronizedTest();
29         Thread t1 = new Thread(obj);
30         Thread t2 = new Thread(obj);

```



```

31         t1.start();
32         t2.start();
33         t1.join();
34         t2.join();
35         System.out.println(count);
36
37     }
38
39
40 }

```

对比上面修饰代码块，直接将 `synchronized` 加到 `addCount` 方法中，也能解决线程安全问题。

## 4、修饰静态方法

这个我们就不贴代码演示了，将 `addCount()` 声明为一个 `static` 修饰的方法，然后在加上 `synchronized`，也能解决线程安全问题。

## 5、原子性、可见性、有序性

通过 `synchronized` 修饰的方法或代码块，能够同时保证这段代码的原子性、可见性和有序性，进而能够保证这段代码的线程安全。

比如通过 `synchronized` 修饰的代码块：

```

public void addCount(){
    synchronized(objMonitor){
        int i = 0;
        while (i++ < 100000) {
            count++;
        }
    }
}

```

其中 `objMonitor` 表示锁对象（下文会介绍这个锁对象），只有获取到这个锁对象之后，才能执行里面的代码，执行完毕之后，在释放这个锁对象。那么同一时刻就会只有一个线程去执行这段代码，把多线程变成了单线程，当然不会存在并发问题了。

这个过程，大家可以想象在公司排队上厕所的情景。

对于原子性，由于同一时刻单线程操作，肯定能够保证原子性。

对于有序性，在JMM内存模型中的Happens-Before规定如下，所以也是能够保证有序性的。

程序的顺序性规则（Program Order Rule）：在一个线程内，按照控制流顺序，书写在前面的操作先行发生于书写在后面的操作。

最后对于可见性，JMM内存模型也规定了：

对一个变量执行unlock操作之前，必须先把此变量同步回主内存中（执行store、write操作）。

大家可能会奇怪，synchronized 并没有lock和unlock操作啊，怎么也能够保证可见性，大家不要急，其实JVM对于这个关键字已经隐式的实现了，下文看字节码会明白的。

## 6、锁对象

大家要注意，我在通过synchronized修饰同步代码块时，使用了一个 Object 对象，名字叫objMonitor。而对于修饰普通方法和静态方法时，只是在方法声明时说明了，并没有锁住什么对象，其实这三者都有各自的锁对象，只有获取了锁对象，线程才能进入执行里面的代码。

- 1 1、修饰代码块：锁定锁的是synchronized括号里配置的对象
- 2 2、修饰普通方法：锁定调用当前方法的this对象
- 3 3、修饰静态方法：锁定当前类的Class对象

多个线程之间，如果要通过 synchronized 保证线程安全，获取的要是同一把锁。如果多个线程多把锁，那么就会有线程安全问题。如下：

```
1 package
  com.ys.algorithmproject.leetcode.demo.concurrency.synchronizedtest;
2
3
4 /**
5  * Create by ItCoke
6  */
7 public class SynchronizedTest implements Runnable{
8
9     public static int count = 0;
10
11
12
13     @Override
14     public void run() {
15         addCount();
16
17     }
18
19     public void addCount(){
20         Object objMonitor = new Object();
21         synchronized(objMonitor){
22             int i = 0;
23             while (i++ < 100000) {
24                 count++;
25             }
26         }
27     }
28
29     public static void main(String[] args) throws Exception{
30         SynchronizedTest obj = new SynchronizedTest();
31         Thread t1 = new Thread(obj);
```

```

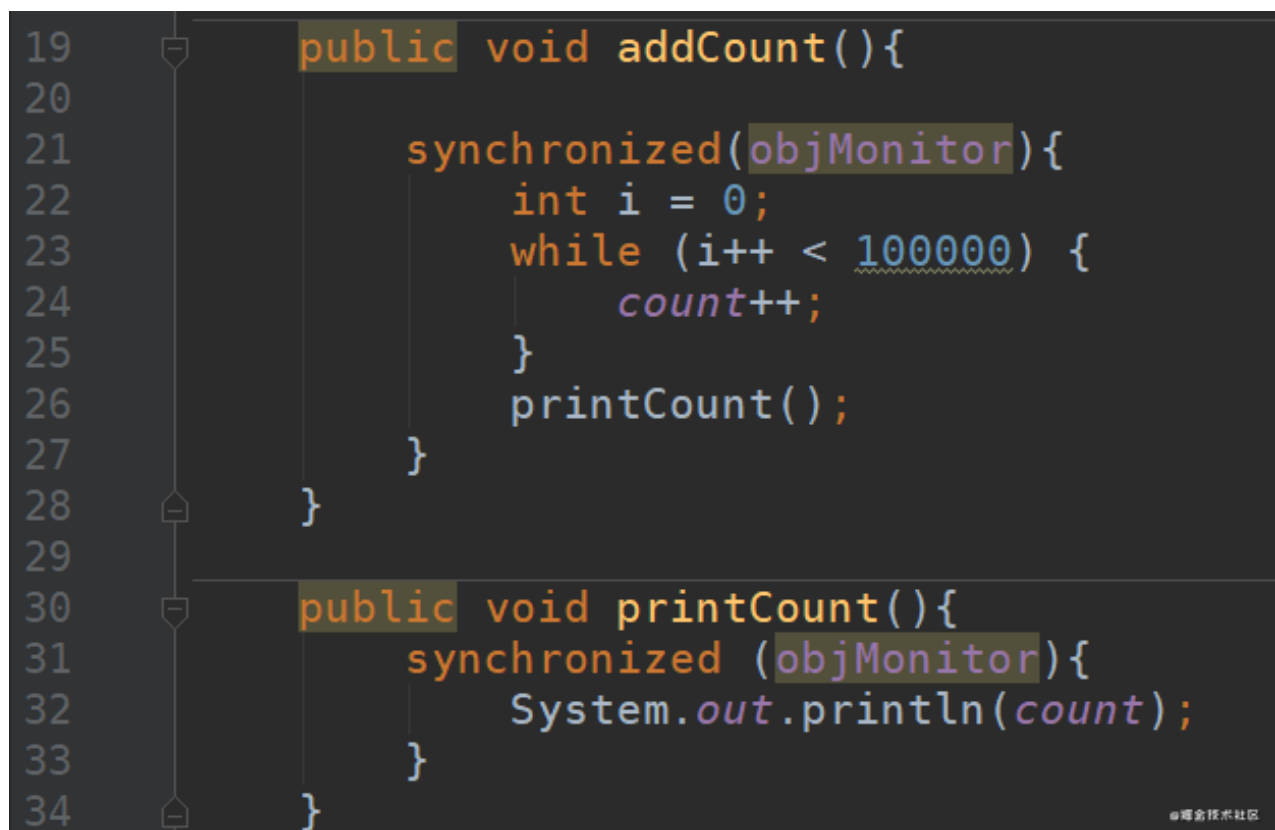
32         Thread t2 = new Thread(obj);
33         t1.start();
34         t2.start();
35         t1.join();
36         t2.join();
37         System.out.println(count);
38
39     }
40
41
42 }

```

我们把原来的锁 `objMonitor` 对象从全局变量移到 `addCount()` 方法中，那么每个线程进入每次进入 `addCount()` 方法都会新建一个 `objMonitor` 对象，也就是多个线程用多把锁，肯定会有线程安全问题。

## 7、可重入

可重入什么意思？字面意思就是一个线程获取到这个锁了，在未释放这把锁之前，还能进入获取锁，如下：



```

19     public void addCount(){
20
21         synchronized(objMonitor){
22             int i = 0;
23             while (i++ < 100000) {
24                 count++;
25             }
26             printCount();
27         }
28     }
29
30     public void printCount(){
31         synchronized (objMonitor){
32             System.out.println(count);
33         }
34     }

```

在 `addCount()` 方法的 `synchronized` 代码块中继续调用 `printCount()` 方法，里面也有一个 `synchronized`，而且都是获取的同一把锁——`objMonitor`。

`synchronized` 是能够保证这段代码正确运行的。至于为什么具有这个特性，可以看下文的实现原理。

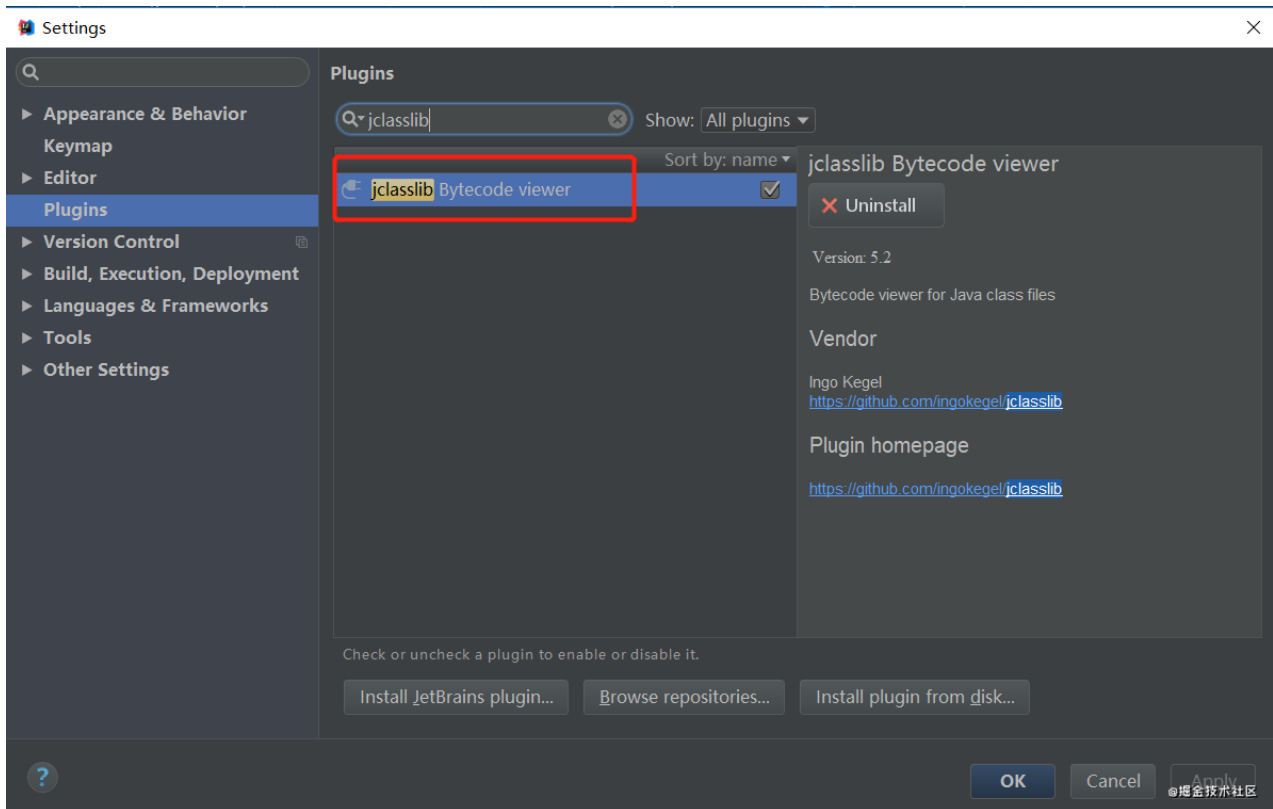
## 8、实现原理

对于如下这段代码：

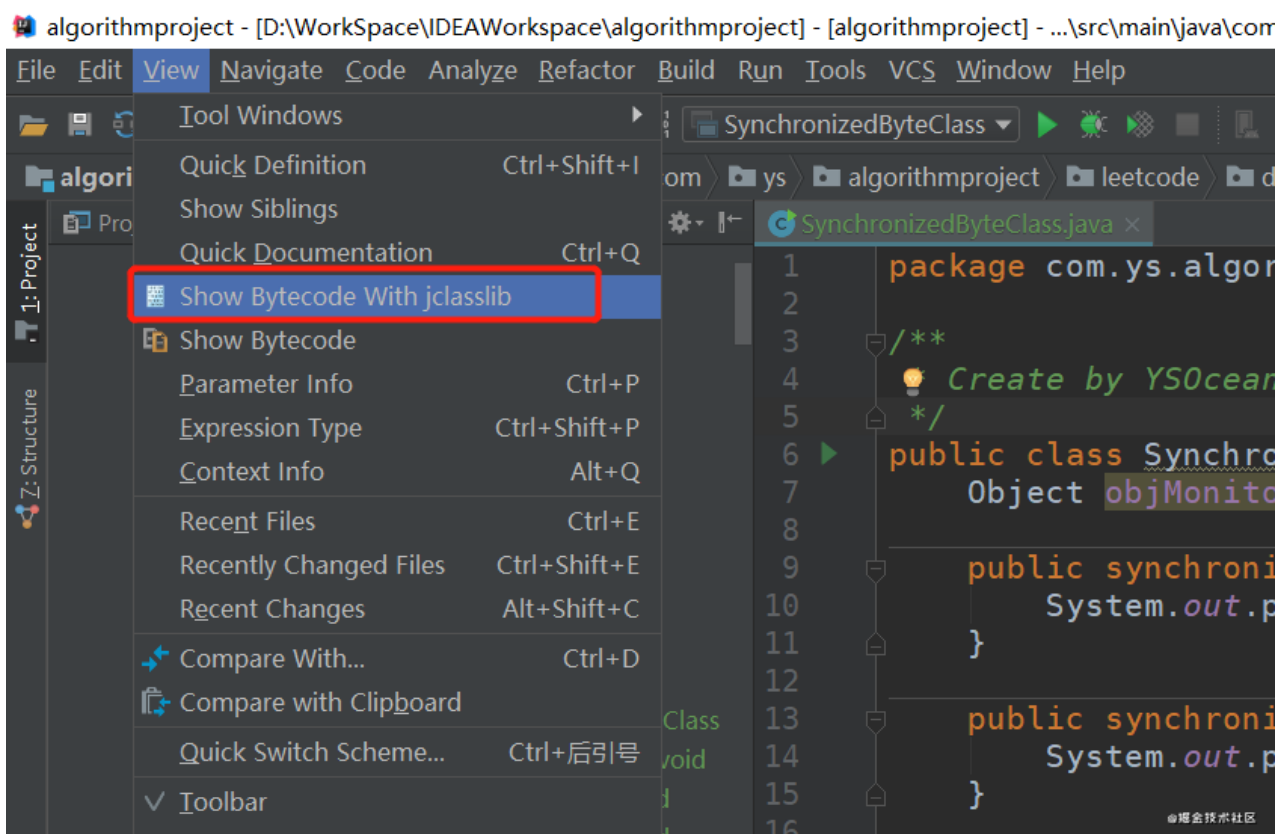
```
1 package
2 com.ys.algorithmproject.leetcode.demo.concurrency.synchronizedtest;
3
4 /**
5  * Create by YSOcean
6  */
7 public class SynchronizedByteClass {
8     Object objMonitor = new Object();
9
10    public synchronized void method1(){
11        System.out.println("Hello synchronized 1");
12    }
13
14    public synchronized static void method2(){
15        System.out.println("Hello synchronized 2");
16    }
17
18    public void method3(){
19        synchronized(objMonitor){
20            System.out.println("Hello synchronized 2");
21        }
22    }
23
24    public static void main(String[] args) {
25
26    }
27 }
```

我们可以通过两种方法查看其class文件的汇编代码。

①、IDEA下载 jclasslib 插件



然后点击 View——Show Bytecode With jclasslib



②、通过 javap 命令

javap -v 文件名（不要后缀）

注意：这里生成汇编的命令是根据编译之后的字节码文件（class文件），所以要先编译。

③、修饰代码块汇编代码

我们直接看method3() 的汇编代码：

```
public void method3();
descriptor: ()V
flags: ACC_PUBLIC
Code:
  stack=2, locals=3, args_size=1
    0: aload_0
    1: getfield      #3                // Field objMonitor:Ljava/lang/Object;
    4: dup
    5: astore_1
    6: monitorenter
    7: getstatic     #4                // Field java/lang/System.out:Ljava/io/PrintStream;
   10: ldc           #7                // String Hello synchronized 2
   12: invokevirtual #6                // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   15: aload_1
   16: monitorexit
   17: goto          25
   20: astore_2
   21: aload_1
   22: monitorexit
   23: aload_2
   24: athrow
   25: return
Exception table:
   from    to  target type
     7     17     20   any
    20     23     20   any
LineNumberTable:
```

©掘金技术社区

对于上图出现的 monitorenter 和 monitorexit 指令，我们查看 JVM虚拟机规范：<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html>，可以看到对这两个指令的介绍。

下面我们说明一下这两个指令：

## 一、monitorenter

monitorenter

Forms

monitorenter = 194 (0xc2)

Operand Stack

..., objectref →  
...

Description

The objectref must be of type reference.  
Each object is associated with a monitor. A monitor is locked if and only if it has an owner. The thread that executes monitorenter attempts to gain ownership of the monitor associated with objectref, as follows:

- If the entry count of the monitor associated with objectref is zero, the thread enters the monitor and sets its entry count to one. The thread is then the owner of the monitor.
- If the thread already owns the monitor associated with objectref, it reenters the monitor, incrementing its entry count.
- If another thread already owns the monitor associated with objectref, the thread blocks until the monitor's entry count is zero, then tries again to gain ownership.

Run-time Exception

If objectref is null, monitorenter throws a NullPointerException.

©掘金技术社区

每个对象与一个监视器锁（monitor）关联。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

- 1、如果 monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
- 2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1。
- 3.如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

## 二、monitorexit

monitorexit

Operation

Exit monitor for object

Format

monitorexit

Forms

monitorexit = 195 (0xc3)

Operand Stack

..., objectref →  
—

Description

The objectref must be of type `reference`.  
The thread that executes `monitorexit` must be the owner of the monitor associated with the instance referenced by `objectref`.  
The thread decrements the entry count of the monitor associated with `objectref`. If as a result the value of the entry count is zero, the thread exits the monitor and is no longer its owner. Other threads that are blocking to enter the monitor are allowed to attempt to do so.

Run-time Exceptions

If `objectref` is `null`, `monitorexit` throws a `NullPointerException`.

掘金技术社区

执行monitorexit的线程必须是object ref所对应的monitor的所有者。

指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

通过上面介绍，我们可以知道 synchronized 底层就是通过这两个命令来执行的同步机制，由此我们也可以看出synchronized 具有可重入性。

### ③、修饰普通方法和静态方法汇编代码

```
public synchronized void method1();
descriptor: ()V
flags: ACC_PUBLIC, ACC_SYNCHRONIZED
Code:
    stack=2, locals=1, args_size=1
        0: getstatic    #4             // Field java/lang/System.out:Ljava/io/PrintStream;
        3: ldc         #5             // String Hello synchronized 1
        5: invokevirtual #6             // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        8: return
LineNumberTable:
    line 10: 0
    line 11: 8
LocalVariableTable:
    Start  Length  Slot  Name  Signature
        0         9      0   this  Lcom/ys/algorithmproject/leetcode/demo/concurrency/synchronizedtest/SynchronizedByJVM$1;
```

```
public static synchronized void method2();
descriptor: ()V
flags: ACC_PUBLIC, ACC_STATIC, ACC_SYNCHRONIZED
Code:
    stack=2, locals=0, args_size=0
        0: getstatic    #4             // Field java/lang/System.out:Ljava/io/PrintStream;
        3: ldc         #7             // String Hello synchronized 2
        5: invokevirtual #6             // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        8: return
LineNumberTable:
    line 14: 0
    line 15: 8
```

可以看到都是通过指令 ACC\_SYNCHRONIZED 来控制的，虽然没有看到方法的同步并没有通过指令 monitorenter和monitorexit来完成，但其本质也是通过这两条指令来实现。

当方法调用时，调用指令将会检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。在方法执行期间，其他任何线程都无法再获得同一个monitor对象。其实和修饰代码块本质上没有区别，只是方法的同步是一种隐式的方式来实现。

## 9、异常自动unlock

可能会有细心的朋友发现，我在介绍 synchronized 修饰代码块时，给出的汇编代码，用红框圈住了两个 monitorexit，根据我们前面介绍，获取monitor加1，退出monitor减1，等于0时，就没有锁了。那为啥会有两个 monitorexit，而只有一个 monitorenter 呢？



```

public void method3();
descriptor: ()V
flags: ACC_PUBLIC
Code:
    stack=2, locals=3, args_size=1
    0: aload_0
    1: getfield      #3          // Field objMonitor:Ljava/lang/Object;
    4: dup
    5: astore_1
    6: monitorenter
    7: getstatic    #4          // Field java/lang/System.out:Ljava/io/PrintStream;
    10: ldc          #7           // String Hello synchronized 2
    12: invokevirtual #6         // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    15: aload_1
    16: monitorexit
    17: goto         25
    20: astore_2
    21: aload_1
    22: monitorexit
    23: aload_2
    24: athrow
    25: return

```

第 6 行执行 monitorenter，然后第16行执行monitorexit，然后执行第17行指令 goto 25，表示跳到第25行代码，第25行是 return，也就是直接结束了。

那第20-24行代码中是什么意思呢？其中第 24 行指令 athrow 表示Java虚拟机隐式处理方法完成异常结束时的监视器退出，也就是执行发生异常了，然后去执行 monitorexit。

进而可以得到结论：

synchronized 修饰的方法或代码块，在执行过程中抛出异常了，也能释放锁（unlock）

我们可以看如下方法，手动抛出异常：

```

public void method4(){
    synchronized(objMonitor){
        System.out.println("Hello synchronized 2");
        throw new RuntimeException();
    }
}

```

然后获取其汇编代码，就只有一个 monitorexit 指令了。

```

public void method4();
descriptor: ()V
flags: ACC_PUBLIC
Code:
    stack=2, locals=3, args_size=1
    0: aload_0
    1: getfield      #3          // Field objMonitor:Ljava/lang/Object;
    4: dup
    5: astore_1
    6: monitorenter
    7: getstatic    #4          // Field java/lang/System.out:Ljava/io/PrintStream;
    10: ldc          #7           // String Hello synchronized 2
    12: invokevirtual #6         // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    15: new          #8           // class java/lang/RuntimeException
    18: dup
    19: invokespecial #9         // Method java/lang/RuntimeException.<init>:()V
    22: athrow
    23: astore_2
    24: aload_1
    25: monitorexit
    26: aload_2
    27: athrow
Exception table:
    from      to      target_type

```



## Java关键字(八)——volatile

在上一篇文章中，我们介绍了 `synchronized` 关键字，`synchronized` 是jdk1.5提供的线程同步机制，可以用来修饰代码块，修饰普通方法，修饰`static`声明的静态方法，能够保证原子性、可见性、有序性，在jdk1.5，`synchronized` 是一个重量级的同步机制，线程挂起和阻塞都要从用户态转入内核态，比较耗性能，但是在jdk1.6时，进行了一系列的优化，比如自旋锁，锁粗化，锁消除等，使得其性能有了很大的提升，关于锁的这些优化，我在锁详解 这篇文章也有详细的介绍。

通常，对于新人来讲，不管三七二十一，对于需要同步的处理，直接上 `synchronized` 准没错，但是JVM还为我们提供了另外一个轻量级的线程同步机制——`volatile`。可能很多人没有在代码中实际使用过这个关键字，但其实，如果你研究过jdk并发包，这个关键字还是使用挺多的，下面我们就来详细介绍这个关键字。

### 1、可见性

通过 `volatile` 关键字修饰的变量，能够保证变量的可见性。

也就是说，当一个线程修改了被`volatile`修饰的变量，修改后的值对于其他线程来说是立即可以知道的。

注意：这里有个误区，可见性并不是原子性。

对于如下代码：

```
1 package com.js.algorithmproject.leetcode.demo.concurrency.volatilettest;  
2  
3 import java.io.InputStream;
```

```

4  import java.util.stream.IntStream;
5
6  /**
7   * Create by YSOcean
8   */
9  public class VolatileTest implements Runnable{
10     public static volatile int count = 0;
11
12
13     @Override
14     public void run() {
15         for (int i = 0; i < 100000; i++) {
16             count++;
17         }
18     }
19
20     public static void main(String[] args) throws Exception{
21         VolatileTest vt = new VolatileTest();
22         Thread t1 = new Thread(vt);
23         Thread t2 = new Thread(vt);
24         t1.start();
25         t2.start();
26         t1.join();
27         t2.join();
28         System.out.println(count);
29     }
30 }

```

代码逻辑很简单，启动两个线程，分别运行run方法，run方法里面循环100000次count++，通过join()方法等待这两个线程执行完毕后，打印 count 的最终值。

注意这个 count 是用 volatile修饰的。

我们说volatile 能够保证可见性，一旦 volatile 被修改了，别的线程也能马上知道，所有很明显这段代码运行结果是 200000。

没错，这个答案是错的，是不是很意外，结果应该小于等于200000。问题就出现在 count++ 这个操作不是原子性。

我们可以通过 javap 命令查看这个run方法的汇编指令：

```

public void run();
descriptor: ()V
flags: ACC_PUBLIC
Code:
    stack=2, locals=2, args_size=1
    0: iconst_0
    1: istore_1
    2: iload_1
    3: ldc          #2          // int 100000
    5: if_icmpge     22
    8: getstatic     #3          // Field count:I
   11: iconst_1
   12: iadd
   13: putstatic     #3          // Field count:I
   16: iinc          1, 1
   19: goto          2
   22: return

```

© 掘金技术社区

虽然 `count++` 在我们程序员看来，就一条代码，但实际上在字节码层面发生了4条指令。首先 `getstatic` 指令，表示从寄存器中取出 `count` 的值并压到栈顶，`iconst_1` 指令表示把int类型1压到操作栈顶；`iadd` 表示执行加法操作；最后`putstatic`指令表示将修改后的`count`写回到寄存器（由于被`volatile`修饰，这时候直接写回内存）。

这就存在一个问题，线程1执行第一条指令把 `count` 值取到操作栈顶时，虽然 `volatile` 指令能够保证此时的 `count` 是正确的，但是在执行 `iconst_1`和`iadd`指令时，有可能线程2已经执行了`count++`所有指令，这时候线程1取到的`count`值就变成过期的数据了，这就会造成问题。

## 2、禁止指令重排序

禁止指令重排序是 `volatile` 关键词的第二个作用。

为了使得处理器内部的运算单元尽量被充分利用，提高运算效率，处理器可能会对输入的代码进行排序。比如 A,B,C三条按顺序运行的代码，经过指令重排后，可能是 B,A,C的顺序来执行。

通过 `volatile` 修饰的变量，在进行指令优化时，要求不能将在对 `volatile` 变量访问的语句放在其后面执行，也不能把 `volatile` 变量后面的语句放到其前面执行。

## 3、总结

这里我们对比一下`volatile` 关键字和 `synchronized` 关键字。

1、关键字`volatile`是线程同步的轻量级实现，`synchronized` 是重量级实现，但是我们并不能说 `volatile` 一定比 `synchronized` 性能好，因为Java虚拟机对锁进行了偏向锁，自旋锁，锁消除，锁粗化等一系列优化措施。所以我们很难说 `volatile`性能肯定比`synchronized`要好；

2、`volatile`只能修饰变量，而`synchronized`可以修饰方法、代码块等。

3、多线程访问`volatile`不会发生阻塞，而`synchronized`会出现阻塞。

4、`volatile`能保证数据的可见性，但不能保证数据的原子性；而`synchronized`可以保证原子性，也可以保证可见性（通过JMM）。关键字`volatile`解决的是变量在多个线程之间的可见性；而`synchronized`关键字解决的是多个线程之间访问资源的同步性。

