目录

final关键字在java中使用非常广泛，可以申明成员变量、方法、类、本地变量。一旦将引用声明为final，将无法再改变这个引用。final关键字还能保证内存同步，本博客将会从final关键字的特性到从java内存层面保证同步讲解。这个内容在面试中也有可能会出现。

final使用

final变量

final变量有成员变量或者是本地变量(方法内的局部变量)，在类成员中final经常和static一起使用，作为类常量使用。**其中类常量必须在声明时初始化，final成员常量可以在构造函数初始化。**

```
public class Main {
    public static final int i; //报错，必须初始化 因为常量在常量池中就存在了，调用时不需要类的初始化，所以必须在声明时初始化
    public static final int j;
    Main() {
        i = 2;
        j = 3;
    }
```

```
}
```

就如上所说的，对于类常量，JVM会缓存在常量池中，在读取该变量时不会加载这个类。

```
public class Main {
    public static final int i = 2;
    Main() {
        System.out.println("调用构造函数"); // 该方法不会调用
    }
    public static void main(String[] args) {
        System.out.println(Main.i);
    }
}
```

final修饰基本数据类型变量和引用

```
@Test
public void final修饰基本类型变量和引用() {
    final int a = 1;
    final int[] b = {1};
    final int[] c = {1};
//  b = c;报错
    b[0] = 1;
    final String aa = "a";
    final Fi f = new Fi();
    //aa = "b";报错
    // f = null;//报错
    f.a = 1;
}
```

final方法表示该方法不能被子类的方法重写，将方法声明为final，在编译的时候就已经静态绑定了，不需要在运行时动态绑定。final方法调用时使用的是invokespecial指令。

```
class PersonalLoan{
    public final String getName(){
        return"personal loan”;
    }
}

class CheapPersonalLoan extends PersonalLoan{
    @Override
    public final String getName(){
        return"cheap personal loan";//编译错误，无法被重载
    }
```

```
    public String test() {
        return getName(); //可以调用，因为是public方法
    }
}
```

final类

final类不能被继承，final类中的方法默认也会是final类型的，java中的String类和Integer类都是final类型的。

```
class Si{
    //一般情况下final修饰的变量一定要被初始化。
    //只有下面这种情况例外，要求该变量必须在构造方法中被初始化。
    //并且不能有空参数的构造方法。
    //这样就可以让每个实例都有一个不同的变量，并且这个变量在每个实例中只会被初始化一次
    //于是这个变量在单个实例里就是常量了。
    final int s ;
    Si(int s) {
        this.s = s;
    }
}
class Bi {
    final int a = 1;
    final void go() {
        //final修饰方法无法被继承
    }
}
class Ci extends Bi {
    final int a = 1;
//        void go() {
//            //final修饰方法无法被继承
//        }
}
final char[]a = {'a'};
final int[]b = {1};



final class PersonalLoan{}

class CheapPersonalLoan extends PersonalLoan {  //编译错误，无法被继承
}



@Test
public void final修饰类() {
    //引用没有被final修饰，所以是可变的。
    //final只修饰了Fi类型，即Fi实例化的对象在堆中内存地址是不可变的。
    //虽然内存地址不可变，但是可以对内部的数据做改变。
```

```
        Fi f = new Fi();
        f.a = 1;
        System.out.println(f);
        f.a = 2;
        System.out.println(f);
        //改变实例中的值并不改变内存地址。

        Fi ff = f;
        //让引用指向新的Fi对象，原来的f对象由新的引用ff持有。
        //引用的指向改变也不会改变原来对象的地址
        f = new Fi();
        System.out.println(f);
        System.out.println(ff);
    }
```

final关键字的知识点

1. final成员变量必须在声明的时候初始化或者在构造器中初始化，否则就会报编译错误。final变量一旦被初始化后不能再次赋值。
2. 本地变量必须在声明时赋值。 因为没有初始化的过程
3. 在匿名类中所有变量都必须是final变量。
4. final方法不能被重写, final类不能被继承
5. 接口中声明的所有变量本身是final的。类似于匿名类
6. final和abstract这两个关键字是反相关的，final类就不可能是abstract的。
7. final方法在编译阶段绑定，称为静态绑定(static binding)。
8. 将类、方法、变量声明为final能够提高性能，这样JVM就有机会进行估计，然后优化。

final方法的好处:

1. 提高了性能，JVM在常量池中会缓存final变量
2. final变量在多线程中并发安全，无需额外的同步开销
3. final方法是静态编译的，提高了调用速度
4. **final类创建的对象是只可读的，在多线程可以安全共享**
5.

final关键字的最佳实践

final的用法

1、final 对于常量来说，意味着值不能改变，例如 final int i=100。这个i的值永远都是100。 但是对于变量来说又不一样，只是标识这个引用不可被改变，例如 final File f=new File("c:\test.txt");

那么这个f一定是不能被改变的，如果f本身有方法修改其中的成员变量，例如是否可读，是允许修改的。有个形象的比喻：一个女子定义了一个final的老公，这个老公的职业和收入都是允许改变的，只是这个女人不会换老公而已。

关于空白final

final修饰的变量有三种：静态变量、实例变量和局部变量，分别表示三种类型的常量。
　另外，final变量定义的时候，可以先声明，而不给初值，这中变量也称为final空白，无论什么情况，编译器都确保空白final在使用之前必须被初始化。　　但是，final空白在final关键字final的使用上提供了更大的灵活性，为此，一个类中的final数据成员就可以实现依对象而有所不同，却有保持其恒定不变的特征。

```
public class FinalTest {
final int p;  实例final变量未初始化，必须构造器中初始化，否则编译不通过
final int q=3;
FinalTest(){
p=1;
}
FinalTest(int i){
p=i;//可以赋值，相当于直接定义p q=i;//不能为一个final变量赋值  }
}
```

final内存分配

刚提到了内嵌机制，现在详细展开。 要知道调用一个函数除了函数本身的执行时间之外，还需要额外的时间去寻找这个函数（类内部有一个函数签名和函数地址的映射表）。所以减少函数调用次数就等于降低了性能消耗。

final修饰的函数会被编译器优化，优化的结果是减少了函数调用的次数。如何实现的，举个例子给你看：

```
public class Test{
final void func(){System.out.println("g");};
public void main(String[] args){
for(int j=0;j<1000;j++)
func();
}}
经过编译器优化之后，这个类变成了相当于这样写：
public class Test{
final void func(){System.out.println("g");};
public void main(String[] args){
for(int j=0;j<1000;j++)
{System.out.println("g");}
}}
```

看出来区别了吧？编译器直接将func的函数体内嵌到了调用函数的地方，这样的结果是节省了1000次函数调用，当然编译器处理成字节码，只是我们可以想象成这样，看个明白。

不过，当函数体太长的话，用final可能适得其反，因为经过编译器内嵌之后代码长度大大增加，于是就增加了jvm解释字节码的时间。

在使用final修饰方法的时候，编译器会将被final修饰过的方法插入到调用者代码处，提高运行速度和效率，但被final修饰的方法体不能过大，编译器可能会放弃内联，但究竟多大的方法会放弃，我还没有做测试来计算过。

下面这些内容是通过两个疑问来继续阐述的

使用final修饰方法会提高速度和效率吗

见下面的测试代码，我会执行五次：

```
public class Test
{
    public static void getJava()
    {
        String str1 = "Java ";
        String str2 = "final ";
        for (int i = 0; i < 10000; i++)
        {
            str1 += str2;
        }
    }
    public static final void getJava_Final()
    {
        String str1 = "Java ";
        String str2 = "final ";
        for (int i = 0; i < 10000; i++)
        {
            str1 += str2;
        }
    }
    public static void main(String[] args)
    {
        long start = System.currentTimeMillis();
        getJava();
        System.out.println("调用不带final修饰的方法执行时间为:" +
(System.currentTimeMillis() - start) + "毫秒时间");
        start = System.currentTimeMillis();
        String str1 = "Java ";
        String str2 = "final ";
        for (int i = 0; i < 10000; i++)
        {
            str1 += str2;
        }
        System.out.println("正常的执行时间为:" + (System.currentTimeMillis() -
```

```
start) + "毫秒时间");
        start = System.currentTimeMillis();
        getJava_Final();
        System.out.println("调用final修饰的方法执行时间为:" +
(System.currentTimeMillis() - start) + "毫秒时间");
    }
}
```

结果为：
第一次：
调用不带final修饰的方法执行时间为:1732毫秒时间
正常的执行时间为:1498毫秒时间
调用final修饰的方法执行时间为:1593毫秒时间
第二次：
调用不带final修饰的方法执行时间为:1217毫秒时间
正常的执行时间为:1031毫秒时间
调用final修饰的方法执行时间为:1124毫秒时间
第三次：
调用不带final修饰的方法执行时间为:1154毫秒时间
正常的执行时间为:1140毫秒时间
调用final修饰的方法执行时间为:1202毫秒时间
第四次：
调用不带final修饰的方法执行时间为:1139毫秒时间
正常的执行时间为:999毫秒时间
调用final修饰的方法执行时间为:1092毫秒时间
第五次：
调用不带final修饰的方法执行时间为:1186毫秒时间
正常的执行时间为:1030毫秒时间
调用final修饰的方法执行时间为:1109毫秒时间

由以上运行结果不难看出，执行最快的是“正常的执行”即代码直接编写，而使用final修饰的方法，不像有些书上或者文章上所说的那样，速度与效率与“正常的执行”无异，而是位于第二位，最差的是调用不加final修饰的方法。

观点：加了比不加好一点。

使用final修饰变量会让变量的值不能被改变吗；

见代码：

```
public class Final
{
    public static void main(String[] args)
    {
        Color.color[3] = "white";
        for (String color : Color.color)
            System.out.print(color+" ");
    }
}

class Color
{
```

```
    public static final String[] color = { "red", "blue", "yellow", "black" };
}
```

```
执行结果：
red blue yellow white
看！，黑色变成了白色。
```

在使用findbugs插件时，就会提示public static String[] color = { "red", "blue", "yellow", "black" };这行代码不安全，但加上final修饰，这行代码仍然是不安全的，因为final没有做到保证变量的值不会被修改！

原因是：final关键字只能保证变量本身不能被赋与新值，而不能保证变量的内部结构不被修改。例如在main方法有如下代码Color.color = new String[]{""};就会报错了。

如何保证数组内部不被修改

那可能有的同学就会问了，加上final关键字不能保证数组不会被外部修改，那有什么方法能够保证呢？答案就是降低访问级别，把数组设为private。这样的话，就解决了数组在外部被修改的不安全性，但也产生了另一个问题，那就是这个数组要被外部使用的。

字符串为了保证不可变性，String内部封装的字符数组就是private final char[]

解决这个问题见代码：

```
import java.util.AbstractList;
import java.util.List;

public class Final
{
    public static void main(String[] args)
    {
        for (String color : Color.color)
            System.out.print(color + " ");
        Color.color.set(3, "white");
    }
}

class Color
{
    private static String[] _color = { "red", "blue", "yellow", "black" };
    public static List<String> color = new AbstractList<String>()
    {
        @Override
        public String get(int index)
        {
            return _color[index];
        }
        @Override
        public String set(int index, String value)
```

```
        {
            throw new RuntimeException("为了代码安全,不能修改数组");
        }
        @Override
        public int size()
        {
            return _color.length;
        }
    };


}
```

这样就OK了，既保证了代码安全，又能让数组中的元素被访问了。

final方法的三条规则

规则1：final修饰的方法不可以被重写。

规则2：final修饰的方法仅仅是不能重写，但它完全可以被重载。

规则3：父类中private final方法，子类可以重新定义，这种情况不是重写。

代码示例

```
规则1代码

public class FinalMethodTest
{
        public final void test(){}
}
class Sub extends FinalMethodTest
{
        // 下面方法定义将出现编译错误，不能重写final方法
        public void test(){}
}

规则2代码

public class Finaloverload {
        //final 修饰的方法只是不能重写，完全可以重载
        public final void test(){}
        public final void test(String arg){}
}

规则3代码

public class PrivateFinalMethodTest
{
        private final void test(){}
}
class Sub extends PrivateFinalMethodTest
```

```
{
        // 下面方法定义将不会出现问题
        public void test(){}
}
```

final 和 jvm的关系

与前面介绍的锁和 volatile 相比较，对 final 域的读和写更像是普通的变量访问。对于 final 域，编译器和处理器要遵守两个重排序规则：

1. 在构造函数内对一个 final 域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。
2. 初次读一个包含 final 域的对象的引用，与随后初次读这个 final 域，这两个操作之间不能重排序。

下面，我们通过一些示例性的代码来分别说明这两个规则：

```
public class FinalExample {
    int i;                            // 普通变量
    final int j;                      //final 变量
    static FinalExample obj;

    public void FinalExample () {     // 构造函数
        i = 1;                        // 写普通域
        j = 2;                        // 写 final 域
    }

    public static void writer () {    // 写线程 A 执行
        obj = new FinalExample ();
    }

    public static void reader () {        // 读线程 B 执行
        FinalExample object = obj;        // 读对象引用
        int a = object.i;                 // 读普通域
        int b = object.j;                 // 读 final 域
    }
}
```

这里假设一个线程 A 执行 writer () 方法，随后另一个线程 B 执行 reader () 方法。下面我们通过这两个线程的交互来说明这两个规则。

写 final 域的重排序规则

写 final 域的重排序规则禁止把 final 域的写重排序到构造函数之外。这个规则的实现包含下面 2 个方面：

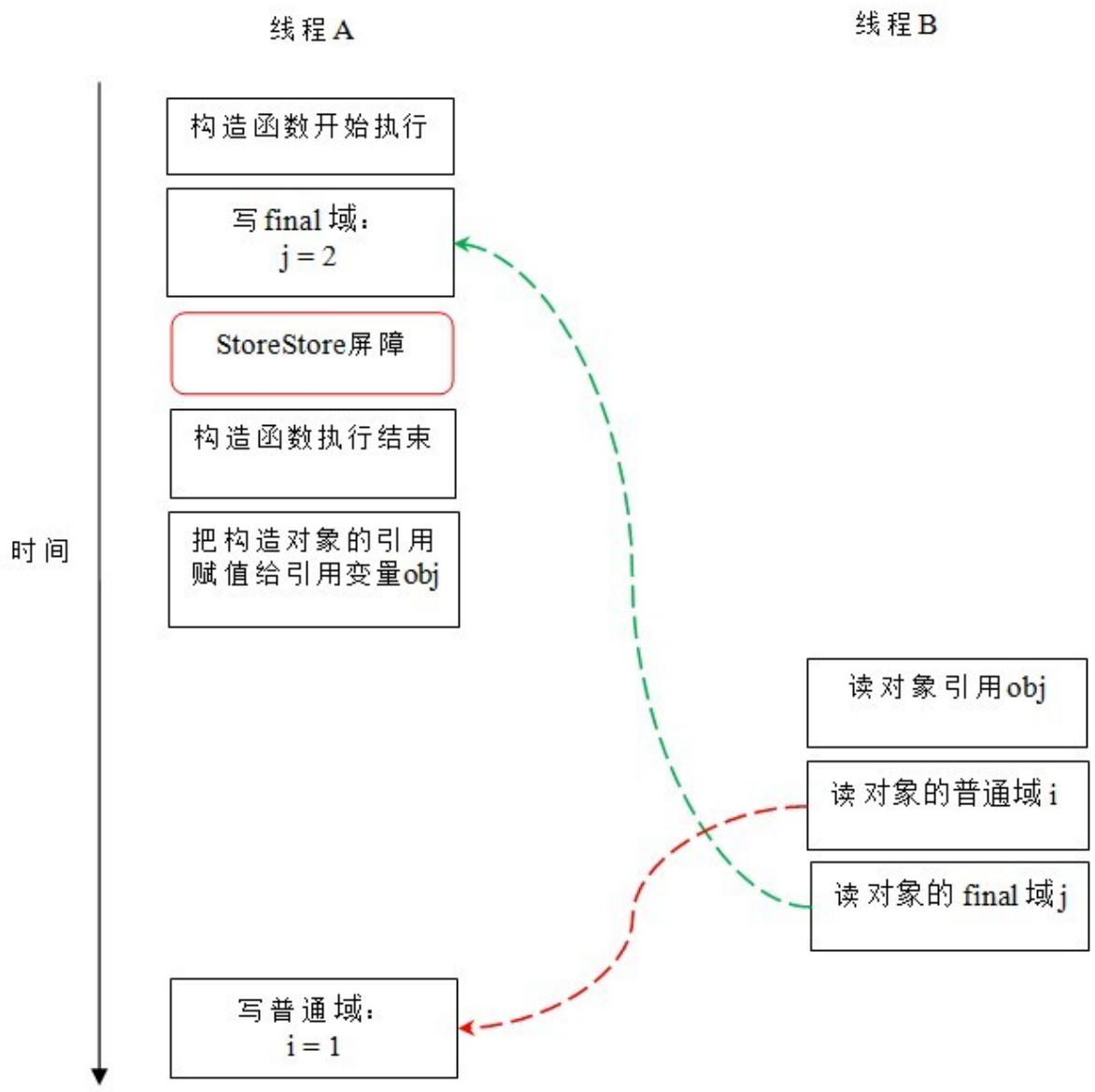
- JMM 禁止编译器把 final 域的写重排序到构造函数之外。

- 编译器会在 final 域的写之后，构造函数 return 之前，插入一个 StoreStore 屏障。这个屏障禁止处理器把 final 域的写重排序到构造函数之外。

现在让我们分析 writer () 方法。writer () 方法只包含一行代码：finalExample = new FinalExample ()。这行代码包含两个步骤：

1. 构造一个 FinalExample 类型的对象；
2. 把这个对象的引用赋值给引用变量 obj。

假设线程 B 读对象引用与读对象的成员域之间没有重排序（马上会说明为什么需要这个假设），下图是一种可能的执行时序：



在上图中，写普通域的操作被编译器重排序到了构造函数之外，读线程 B 错误的读取了普通变量 i 初始化之前的值,int默认值0。而写 final 域的操作，被写 final 域的重排序规则“限定”在了构造函数之内，读线程 B 正确的读取了 final 变量初始化之后的值。

写 final 域的重排序规则可以确保：在对象引用为任意线程可见之前，对象的 final 域已经被正确初始化过了，而普通域不具有这个保障。以上图为例，在读线程 B“看到”对象引用 obj 时，很可能 obj 对象还没有构造完成（对普通域 i 的写操作被重排序到构造函数外，此时初始值 1 还没有写入普通域 i）。

读 final 域的重排序规则

读 final 域的重排序规则如下：

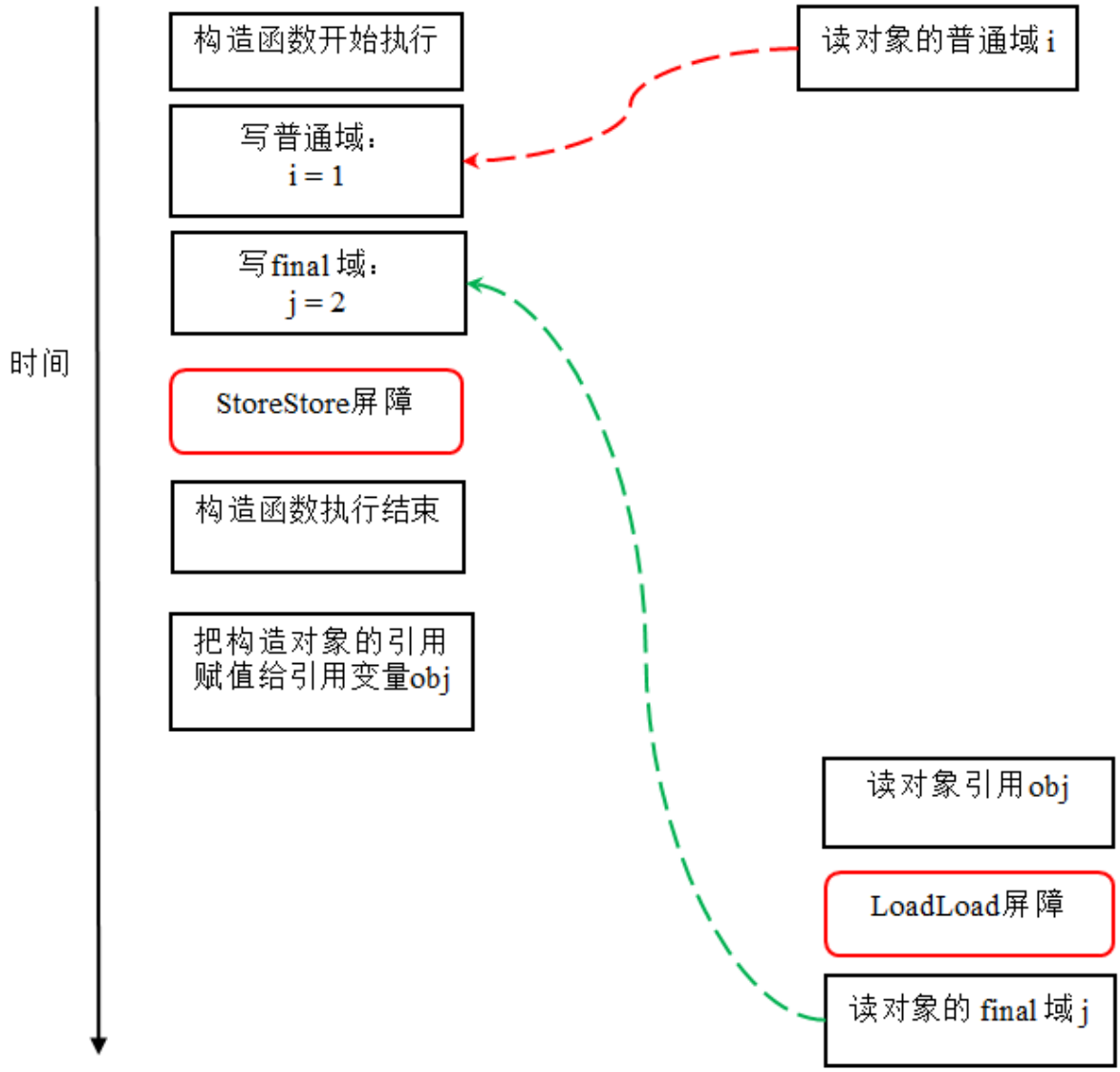
- 在一个线程中，初次读对象引用与初次读该对象包含的 final 域，JMM 禁止处理器重排序这两个操作（注意，这个规则仅仅针对处理器）。编译器会在读 final 域操作的前面插入一个 LoadLoad 屏障。

初次读对象引用与初次读该对象包含的 final 域，这两个操作之间存在间接依赖关系。由于编译器遵守间接依赖关系，因此编译器不会重排序这两个操作。大多数处理器也会遵守间接依赖，大多数处理器也不会重排序这两个操作。但有少数处理器允许对存在间接依赖关系的操作做重排序（比如 alpha 处理器），这个规则就是专门用来针对这种处理器。

reader() 方法包含三个操作：

1. 初次读引用变量 obj;
2. 初次读引用变量 obj 指向对象的普通域 i。
3. 初次读引用变量 obj 指向对象的 final 域 j。

现在我们假设写线程 A 没有发生任何重排序，同时程序在不遵守间接依赖的处理器上执行，下面是一种可能的执行时序：



在上图中，读对象的普通域的操作被处理器重排序到读对象引用之前。读普通域时，该域还没有被写线程 A 写入，这是一个错误的读取操作。而读 final 域的重排序规则会把读对象 final 域的操作"限定"在读对象引用之后，此时该 final 域已经被 A 线程初始化过了，这是一个正确的读取操作。

读 final 域的重排序规则可以确保：在读一个对象的 final 域之前，一定会先读包含这个 final 域的对象的引用。在这个示例程序中，如果该引用不为 null，那么引用对象的 final 域一定已经被 A 线程初始化过了。

如果 final 域是引用类型

上面我们看到的 final 域是基础数据类型，下面让我们看看如果 final 域是引用类型，将会有什么效果?

请看下列示例代码：

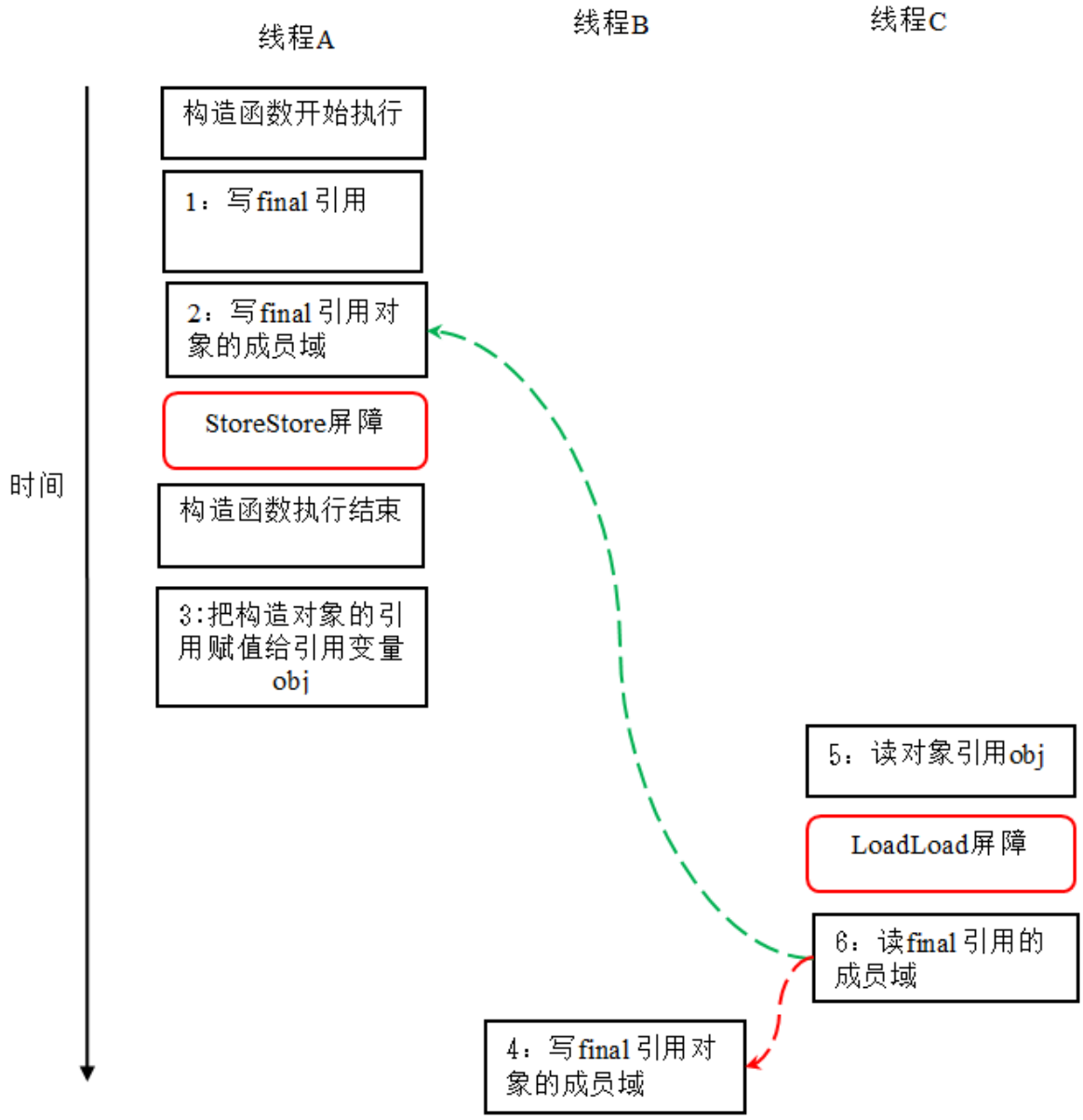
```
public class FinalReferenceExample {
final int[] intArray;                     //final 是引用类型
static FinalReferenceExample obj;
```

```
public FinalReferenceExample () {         // 构造函数
    intArray = new int[1];               //1
    intArray[0] = 1;                    //2
}

public static void writerOne () {           // 写线程 A 执行
    obj = new FinalReferenceExample ();  //3
}

public static void writerTwo () {           // 写线程 B 执行
    obj.intArray[0] = 2;                 //4
}

public static void reader () {               // 读线程 C 执行
    if (obj != null) {                     //5
        int temp1 = obj.intArray[0];        //6
    }
}
}
```

这里 final 域为一个引用类型，它引用一个 int 型的数组对象。对于引用类型，写 final 域的重排序规则对编译器和处理器增加了如下约束：

1. 在构造函数内对一个 final 引用的对象的成员域的写入，与随后在构造函数外把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

对上面的示例程序，我们假设首先线程 A 执行 writerOne() 方法，执行完后线程 B 执行 writerTwo() 方法，执行完后线程 C 执行 reader () 方法。下面是一种可能的线程执行时序：



在上图中，1 是对 final 域的写入，2 是对这个 final 域引用的对象的成员域的写入，3 是把被构造的对象的引用赋值给某个引用变量。这里除了前面提到的 1 不能和 3 重排序外，2 和 3 也不能重排序。

JMM 可以确保读线程 C 至少能看到写线程 A 在构造函数中对 final 引用对象的成员域的写入。即 C 至少能看到数组下标 0 的值为 1。而写线程 B 对数组元素的写入，读线程 C 可能看的到，也可能看不到。JMM 不保证线程 B 的写入对读线程 C 可见，因为写线程 B 和读线程 C 之间存在数据竞争，此时的执行结果不可预知。

如果想要确保读线程 C 看到写线程 B 对数组元素的写入，写线程 B 和读线程 C 之间需要使用同步原语（lock 或 volatile）来确保内存可见性。

参考文章

static（可以参考JVM对象的创建过程和类加载机制进行理解）

1. 静态变量

- 静态变量：又称为类变量，也就是说这个变量属于类的，类所有的实例都共享静态变量，可以直接通过类名来访问它。静态变量在内存中只存在一份。
- 实例变量：每创建一个实例就会产生一个实例变量，它与该实例同生共死。

```
public class A {

     private int x;// 实例变量，是在类被加载后，new对象时候进行默认或者显示赋值初始化
    private  static  int  y;//静态变量在类加载的linking的准备阶段静态变量分配内存，方法区，并且默认赋值，在类初始化阶段进行执行clinit方法，对类的静态变量进行显示赋值。
    public static void main(String[] args) {
        // int x = A.x;  // Non-static field 'x' cannot be referenced from a stat
        A a = new A();
        int x = a.x;
        int y = A.y;
    }
}
```

2. 静态方法

静态方法在类加载的时候就存在了，它不依赖于任何实例。所以静态方法必须有实现，也就是说它不能是抽象方法。

```
public abstract class A {
    public static void func1(){
    }
    // public abstract static void func2();  // Illegal combination of modifiers:
}
```

只能访问所属类的静态字段和静态方法，方法中不能有 this 和 super 关键字，因此这两个关键字与具体对象关联。

静态方法为什么不可以调用非静态属性或者方法？
静态方法和静态变量是在类加载阶段进行默认或显示初始化的，而非静态属性和方法属于new后的对象，类加载先于对象初始化，静态方法先于对象存在，不可能类没加载完就调用一个具体对象的非静态方法。
https://dingqidong.com/?id=137

```
public class A {

    private static int x;
    private int y;

    public static void func1(){
       int a = x;
      静态方法内调用一个非静态成员为什么是非法的？
       // int b = y;  // Non-static field 'y' cannot be referenced from a static  //
    } int b = this.y;     // 'A.this' cannot be referenced from a static con
}
```

3. 静态语句块

静态语句块在类初始化时运行一次。

静态代码块是在类加载的初始化阶段执行的，执行clinit方法，静态变量的赋值和静态代码块梵高clinit方法中进行初始化。

```
public class A {
    static {
        System.out.println("123");
    }

    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new A();
    }
}
```

```
123
```

4. 静态内部类

非静态内部类依赖于外部类的实例，也就是说需要先创建外部类实例，才能用这个实例去创建非静态内部类。而静态内部类不需要，像一个独立的类

```
public class OuterClass {

    class InnerClass {
    }

    static class StaticInnerClass {
    }

    public static void main(String[] args) {
        // InnerClass innerClass = new InnerClass(); // 'OuterClass.this' cannot
        OuterClass outerClass = new OuterClass();
        InnerClass innerClass = outerClass.new InnerClass();
        StaticInnerClass staticInnerClass = new StaticInnerClass();
```

```
    }
}
```

静态内部类不能访问外部类的非静态的变量和方法。

5. 静态导包

在使用静态变量和方法时不用再指明 ClassName，从而简化代码，但可读性大大降低。

```
import static com.xxx.ClassName.*
```

6. 初始化顺序

静态变量和静态语句块优先于实例变量和普通语句块，静态变量和静态语句块的初始化顺序取决于它们在代码中的顺序。

```
public static String staticField = "静态变量"; 1
```

```
static {
    System.out.println("静态语句块"); 2
}
```

```
public String field = "实例变量";  3
```

```
{
    System.out.println("普通语句块"); 4
}
```

1 2 是在类加载的准备和初始化阶段进行赋值和执行的，执行类的clinit方法

4 5 6 是在执行new 对象时候执行构造器init方法

4 5是对对象属性和普通代码块进行默认或者显示初始化，最后才是我们程序员写的构造器5的初始化

最后才是构造函数的初始化。

```
public InitialOrderTest() {   5
    System.out.println("构造函数");
}
```

存在继承的情况下，初始化顺序为：

- 父类（静态变量、静态语句块）
- 子类（静态变量、静态语句块）
- 父类（实例变量、普通语句块）
- 父类（构造函数）