关键字: final详解

面试问题去理解final

- 所有的final修饰的字段都是编译期常量吗?
- 如何理解private所修饰的方法是隐式的final?
- 说说final类型的类如何拓展? 比如String是final类型，我们想写个MyString复用所有String中方法，同时增加一个新的toMyString()的方法，应该如何做?
- final方法可以被重载吗? 可以
- 父类的final方法能不能够被子类重写? 不可以
- 说说final域重排序规则?
- 说说final的原理?
- 使用 final 的限制条件和局限性?

final基础使用

修饰类

当某个类的整体定义为final时，就表明了你不能打算继承该类，而且也不允许别人这么做。即这个类是不能有子类的。

注意：final类中的所有方法都隐式为final，因为无法覆盖他们，所以在final类中给任何方法添加final关键字是没有任何意义的。

设计模式中最重要的两种关系，一种是继承/实现；另外一种是组合关系。所以当遇到不能用继承的(final修饰的类), 应该考虑用组合, 如下代码大概写个组合实现的意思：

```
/**
* @pdai
*/
class MyString{

    private String innerString;

    // ...init & other methods

    // 支持老的方法
    public int length(){
        return innerString.length(); // 通过innerString调用老的方法
    }

    // 添加新方法
    public String toMyString(){
        //...
    }
}
```

修饰方法

> 常规的使用就不说了，这里说下:

- private 方法是隐式的final
- final方法是可以被重载的

private final

类中所有private方法都隐式地指定为final的，由于无法取用private方法，所以也就不能覆盖它。可以对private方法增添final关键字，但这样做并没有什么好处。看下下面的例子:

```
public class Base {
    private void test() {
    }
}

public class Son extends Base{
    public void test() {
    }
    public static void main(String[] args) {
        Son son = new Son();
        Base father = son;
        //father.test();
    }
}
```

Base和Son都有方法test(),但是这并不是一种覆盖，因为private所修饰的方法是隐式的final，也就是无法被继承，所以更不用说是覆盖了，在Son中的test()方法不过是属于Son的新成员罢了，Son进行向上转型得到father，但是father.test()是不可执行的，因为Base中的test方法是private的，无法被访问到。

final方法是可以被重载的

我们知道父类的final方法是不能够被子类重写的，那么final方法可以被重载吗？答案是可以的，下面代码是正确的。

```
public class FinalExampleParent {
    public final void test() {
    }

    public final void test(String str) {
    }
}
```

修饰参数

Java允许在参数列表中以声明的方式将参数指明为final，这意味这你无法在方法中更改参数引用所指向的对象。这个特性主要用来向匿名内部类传递数据。

修饰变量

> 常规的用法比较简单，这里通过下面三个问题进一步说明。

所有的final修饰的字段都是编译期常量吗?

现在来看编译期常量和非编译期常量, 如:

```
public class Test {
    //编译期常量
    final int i = 1;
    final static int J = 1;
    final int[] a = {1,2,3,4};
    //非编译期常量
    Random r = new Random();
    final int k = r.nextInt();

    public static void main(String[] args) {

    }
}
```

k的值由随机数对象决定，所以不是所有的final修饰的字段都是编译期常量，只是k的值在被初始化后无法被更改。

static final

一个既是static又是final 的字段只占据一段不能改变的存储空间，它必须在定义的时候进行赋值，否则编译器将不予通过。

```
import java.util.Random;
public class Test {
    static Random r = new Random();
    final int k = r.nextInt(10);
    static final int k2 = r.nextInt(10);
    public static void main(String[] args) {
        Test t1 = new Test();
        System.out.println("k="+t1.k+" k2="+t1.k2);
        Test t2 = new Test();
        System.out.println("k="+t2.k+" k2="+t2.k2);
    }
}
```

上面代码某次输出结果:

```
k=2 k2=7
k=8 k2=7
```

我们可以发现对于不同的对象k的值是不同的，但是k2的值却是相同的，这是为什么呢? 因为static关键字所修饰的字段并不属于一个对象，而是属于这个类的。也可简单的理解为static final所修饰的字段仅占据内存的一个一份空间，一旦被初始化之后便不会被更改。

blank final

Java允许生成空白final，也就是说被声明为final但又没有给出定值的字段,但是必须在该字段被使用之前被赋值，这给予我们两种选择：

- 在定义处进行赋值(这不叫空白final)
- 在构造器中进行赋值，保证了该值在被使用前赋值。

这增强了final的灵活性。

看下面代码:

```
public class Test {
    final int i1 = 1;
    final int i2;//空白final
    public Test() {
        i2 = 1;
    }
    public Test(int x) {
        this.i2 = x;
    }
}
```

可以看到i2的赋值更为灵活。但是请注意，如果字段由static和final修饰，仅能在定义处赋值，因为该字段不属于对象，属于这个类。

final域重排序规则

上面我们聊的final使用，应该属于Java基础层面的，当理解这些后我们就真的算是掌握了final吗? 有考虑过final在多线程并发的情况吗? 在java内存模型中我们知道java内存模型为了能让处理器和编译器底层发挥他们的最大优势，对底层的约束就很少，也就是说针对底层来说java内存模型就是一弱内存数据模型。同时，处理器和编译为了性能优化会对指令序列有编译器和处理器重排序。那么，在多线程情况下,final会进行怎样的重排序? 会导致线程安全的问题吗? 下面，就来看看final的重排序。

*final*域为基本类型

先看一段示例性的代码：

```
public class FinalDemo {
    private int a;  //普通域
    private final int b; //final域
    private static FinalDemo finalDemo;

    public FinalDemo() {
        a = 1; // 1. 写普通域
        b = 2; // 2. 写final域
    }

    public static void writer() {
        finalDemo = new FinalDemo();
    }

    public static void reader() {
```

```
        FinalDemo demo = finalDemo; // 3.读对象引用
        int a = demo.a;    //4.读普通域
        int b = demo.b;    //5.读final域
    }
}
```

假设线程A在执行writer()方法，线程B执行reader()方法。

写final域重排序规则

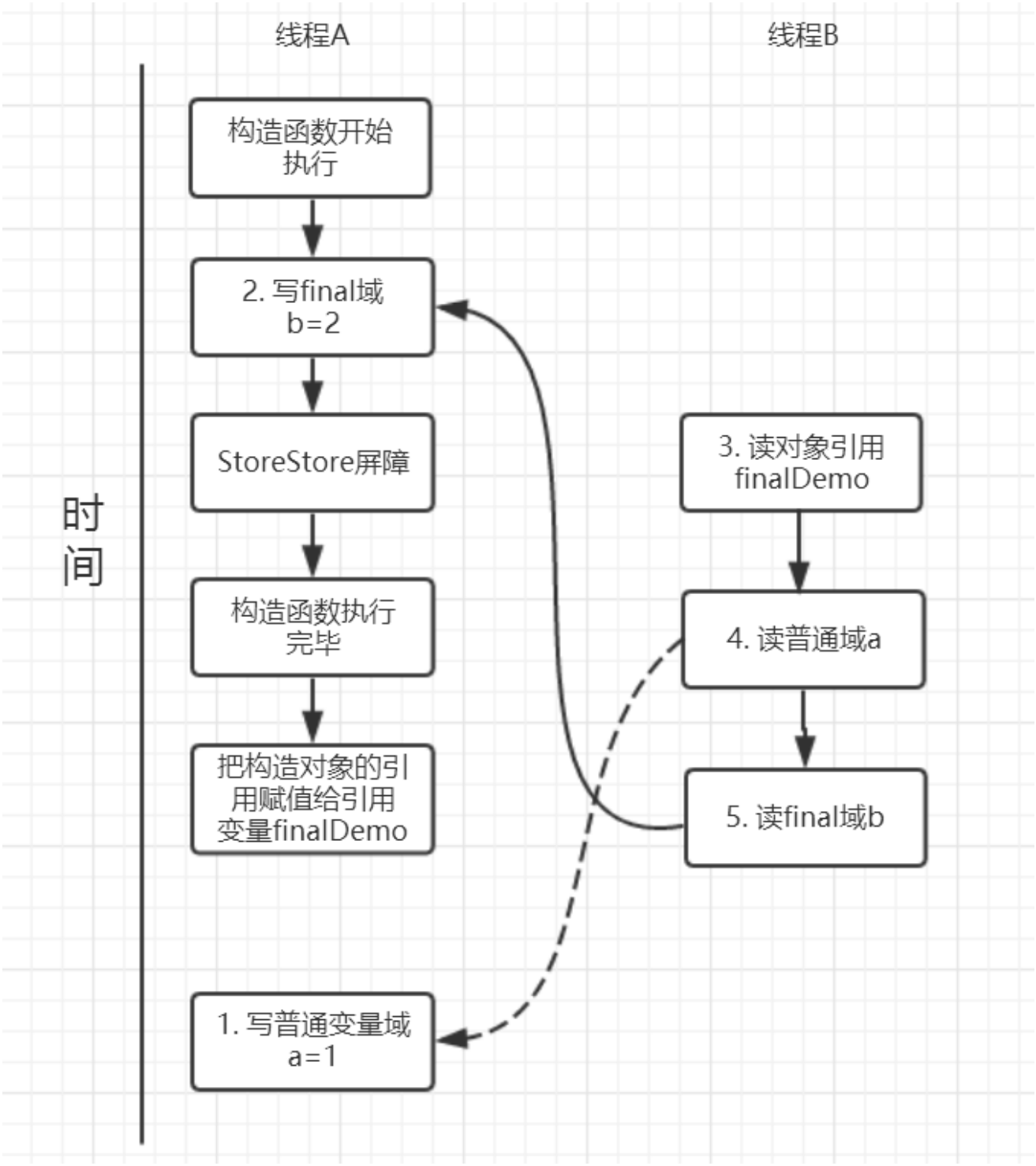
写final域的重排序规则禁止对final域的写重排序到构造函数之外，这个规则的实现主要包含了两个方面：

- 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域写之后，构造函数return之前，插入一个storestore屏障。这个屏障可以禁止处理器把final域的写重排序到构造函数之外。

我们再来分析writer方法，虽然只有一行代码，但实际上做了两件事情：

- 构造了一个FinalDemo对象；
- 把这个对象赋值给成员变量finalDemo。

我们来画下存在的一种可能执行时序图，如下：



由于a,b之间没有数据依赖性，普通域(普通变量)a可能会被重排序到构造函数之外，线程B就有可能读到的是普通变量a初始化之前的值(零值)，这样就可能出现错误。而final域变量b，根据重排序规则，会禁止final修饰的变量b重排序到构造函数之外，从而b能够正确赋值，线程B就能够读到final变量初始化后的值。

因此，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，而普通域就不具有这个保障。比如在上例，线程B有可能就是一个未正确初始化的对象finalDemo。

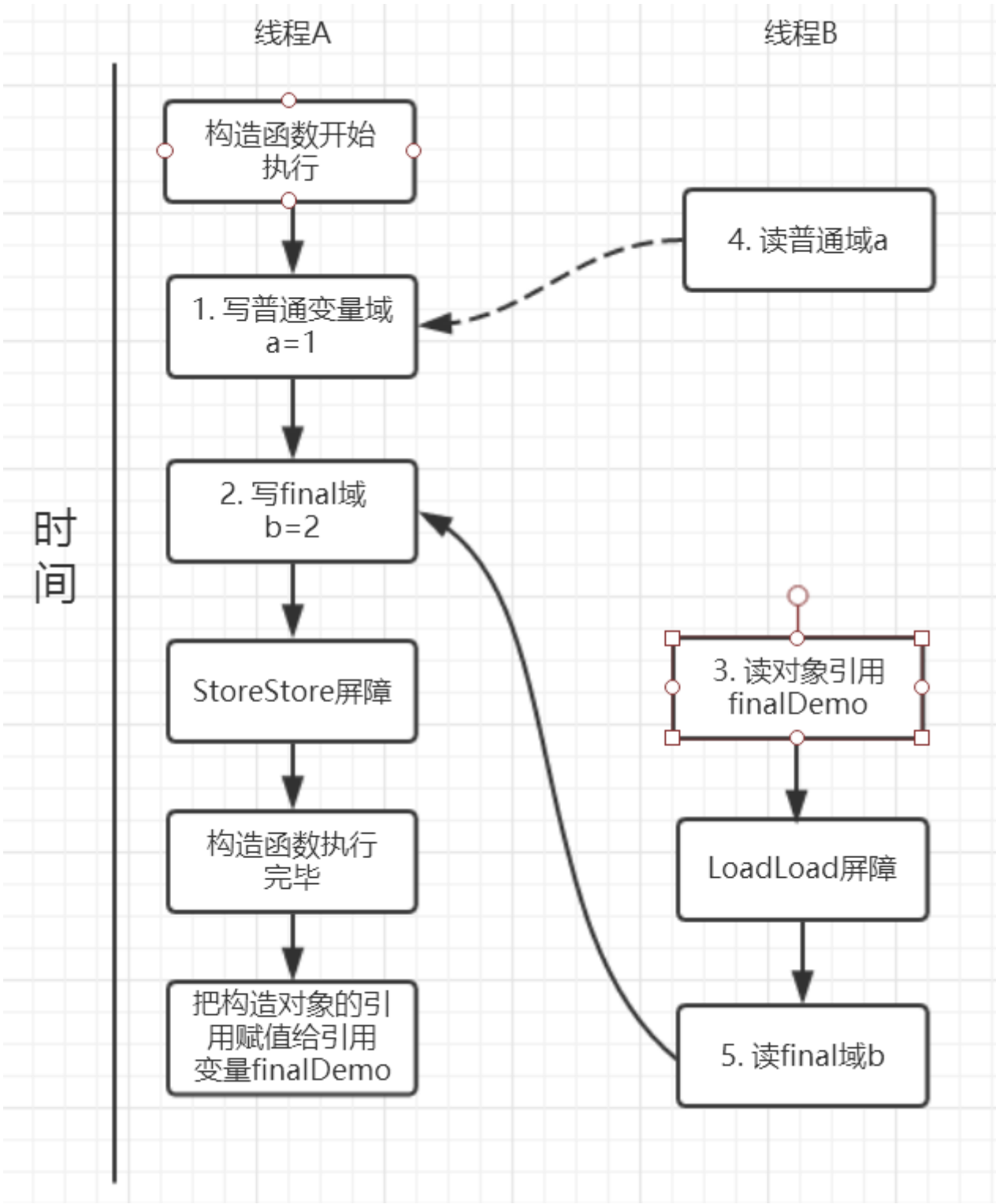
读final域重排序规则

读final域重排序规则为：在一个线程中，初次读对象引用和初次读该对象包含的final域，JMM会禁止这两个操作的重排序。(注意，这个规则仅仅是针对处理器)，处理器会在读final域操作的前面插入一个LoadLoad屏障。实际上，读对象的引用和读该对象的final域存在间接依赖性，一般处理器不会重排序这两个操作。但是有一些处理器会重排序，因此，这条禁止重排序规则就是针对这些处理器而设定的。

read()方法主要包含了三个操作：

- 初次读引用变量finalDemo;
- 初次读引用变量finalDemo的普通域a;
- 初次读引用变量finalDemo的final与b;

假设线程A写过程没有重排序，那么线程A和线程B有一种的可能执行时序为下图：



读对象的普通域被重排序到了读对象引用的前面就会出现线程B还未读到对象引用就在读取该对象的普通域变量，这显然是错误的操作。而final域的读操作就“限定”了在读final域变量前已经读到了该对象的引用，从而就可以避免这种情况。

读final域的重排序规则可以确保：在读一个对象的final域之前，一定会先读这个包含这个final域的对象的引用。

*final*域为引用类型

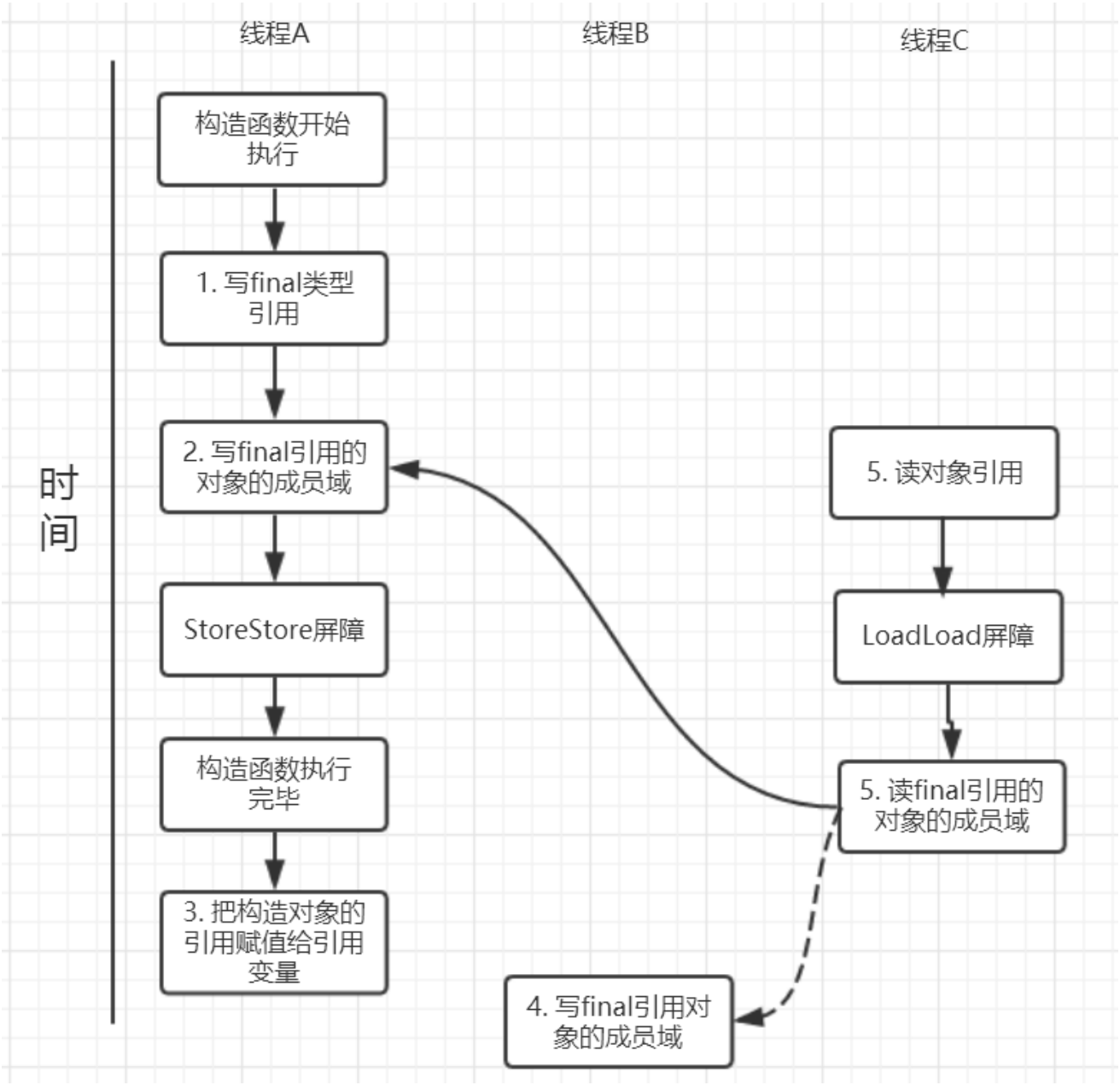
我们已经知道了final域是基本数据类型的时候重排序规则是怎么的了？如果是引用数据类型了？我们接着继续来探讨。

对final修饰的对象的成员域写操作

针对引用数据类型，final域写针对编译器和处理器重排序增加了这样的约束：在构造函数内对一个final修饰的对象的成员域的写入，与随后在构造函数之外把这个被构造的对象的引用赋给一个引用变量，这两个操作是不能被重排序的。注意这里的是“增加”也就说前面对final基本数据类型的重排序规则在这里还是使用。这句话是比较拗口的，下面结合实例来看。

```
public class FinalReferenceDemo {
    final int[] arrays;
    private FinalReferenceDemo finalReferenceDemo;

    public FinalReferenceDemo() {
        arrays = new int[1];  //1
        arrays[0] = 1;        //2
    }

    public void writerOne() {
        finalReferenceDemo = new FinalReferenceDemo(); //3
    }

    public void writerTwo() {
        arrays[0] = 2;  //4
    }

    public void reader() {
        if (finalReferenceDemo != null) {  //5
            int temp = finalReferenceDemo.arrays[0];  //6
        }
    }
}
```

针对上面的实例程序，线程线程A执行wirterOne方法，执行完后线程B执行writerTwo方法，然后线程C执行reader方法。下图就以这种执行时序出现的一种情况来讨论(耐心看完才有收获)。



由于对final域的写禁止重排序到构造方法外，因此1和3不能被重排序。由于一个final域的引用对象的成员域写入不能与随后将这个被构造出来的对象赋给引用变量重排序，因此2和3不能重排序。

对final修饰的对象的成员域读操作

JMM可以确保线程C至少能看到写线程A对final引用的对象的成员域的写入，即能看下arrays[0] = 1，而写线程B对数组元素的写入可能看到可能看不到。JMM不保证线程B的写入对线程C可见，线程B和线程C之间存在数据竞争，此时的结果是不可预知的。如果可见的，可使用锁或者volatile。

关于*final*重排序的总结

按照final修饰的数据类型分类：

- 基本数据类型：
  - `final域写`：禁止final域写与构造方法重排序，即禁止final域写重排序到构造方法之外，从而保证该对象对所有线程可见时，该对象的final域全部已经初始化过。
  - `final域读`：禁止初次读对象的引用与读该对象包含的final域的重排序。
- 引用数据类型：

- 额外增加约束：禁止在构造函数对一个final修饰的对象的成员域的写入与随后将这个被构造的对象的引用赋值给引用变量 重排序

final再深入理解

*final*的实现原理

上面我们提到过，写final域会要求编译器在final域写之后，构造函数返回前插入一个StoreStore屏障。读final域的重排序规则会要求编译器在读final域的操作前插入一个LoadLoad屏障。

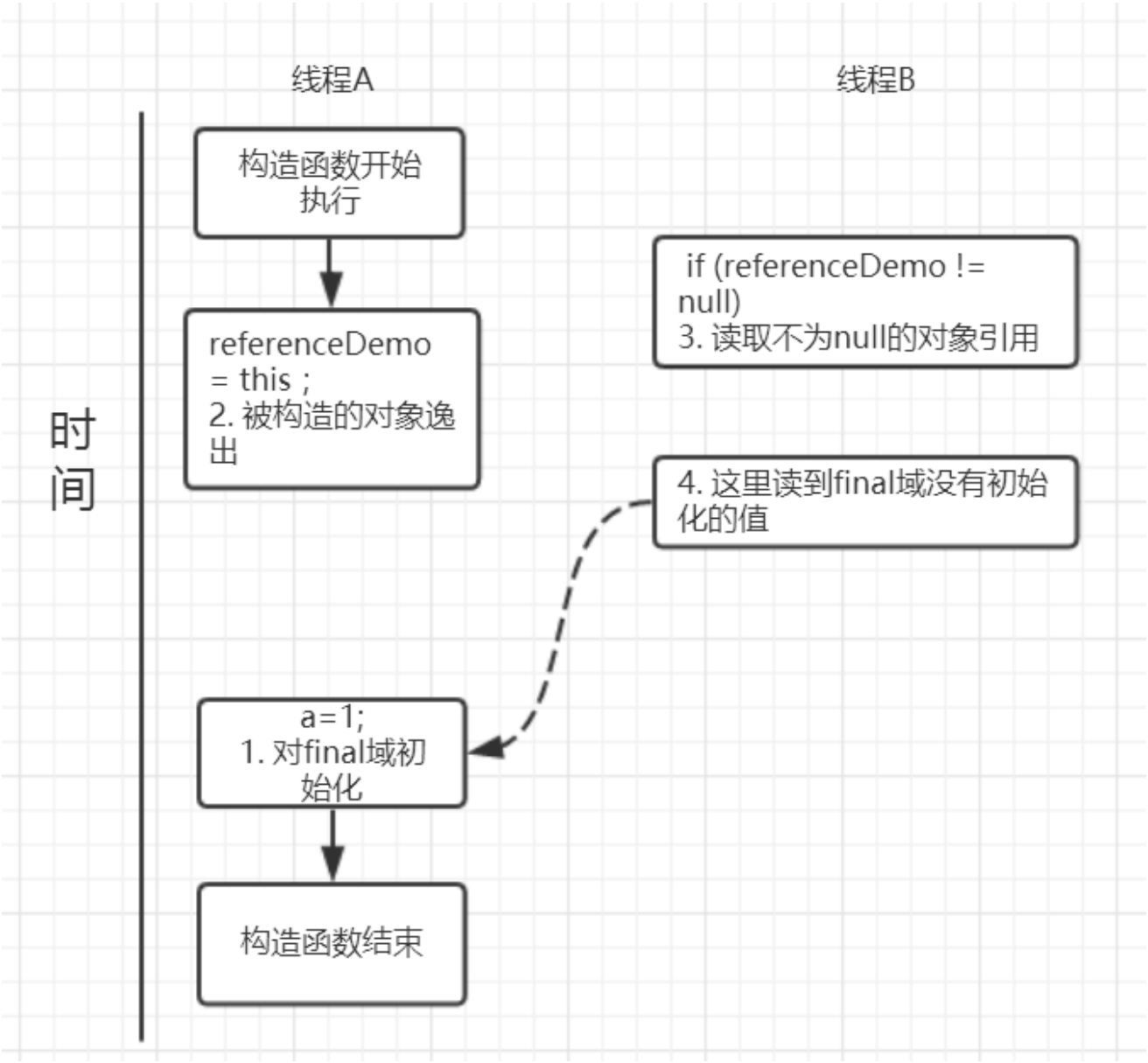
很有意思的是，如果以X86处理为例，X86不会对写-写重排序，所以StoreStore屏障可以省略。由于不会对有间接依赖性的操作重排序，所以在X86处理器中，读final域需要的LoadLoad屏障也会被省略掉。也就是说，以X86为例的话，对final域的读/写的内存屏障都会被省略！具体是否插入还是得看是什么处理器

为什么*final*引用不能从构造函数中“溢出”

这里还有一个比较有意思的问题：上面对final域写重排序规则可以确保我们在使用一个对象引用的时候该对象的final域已经在构造函数被初始化过了。但是这里其实是有一个前提条件的，也就是：在构造函数，不能让这个被构造的对象被其他线程可见，也就是说该对象引用不能在构造函数中“溢出”。以下面的例子来说：

```
public class FinalReferenceEscapeDemo {
    private final int a;
    private FinalReferenceEscapeDemo referenceDemo;

    public FinalReferenceEscapeDemo() {
        a = 1;  //1
        referenceDemo = this; //2
    }

    public void writer() {
        new FinalReferenceEscapeDemo();
    }

    public void reader() {
        if (referenceDemo != null) {  //3
            int temp = referenceDemo.a; //4
        }
    }
}
```

可能的执行时序如图所示：



假设一个线程A执行writer方法另一个线程执行reader方法。因为构造函数中操作1和2之间没有数据依赖性，1和2可以重排序，先执行了2，这个时候引用对象referenceDemo是个没有完全初始化的对象，而当线程B去读取该对象时就会出错。尽管依然满足了final域写重排序规则：在引用对象对所有线程可见时，其final域已经完全初始化成功。但是，引用对象“this”逸出，该代码依然存在线程安全的问题。

使用 *final* 的限制条件和局限性

当声明一个 final 成员时，必须在构造函数退出前设置它的值。

```
public class MyClass {
  private final int myField = 1;
  public MyClass() {
    ...
  }
}
```

或者

```
public class MyClass {
  private final int myField;
  public MyClass() {
    ...
    myField = 1;
    ...
  }
}
```

将指向对象的成员声明为 final 只能将该引用设为不可变的，而非所指的对象。

下面的方法仍然可以修改该 list。

```
private final List myList = new ArrayList();
myList.add("Hello");
```

声明为 final 可以保证如下操作不合法

```
myList = new ArrayList();
myList = someOtherList;
```

如果一个对象将会在多个线程中访问并且你并没有将其成员声明为 final，则必须提供其他方式保证线程安全。

" 其他方式 " 可以包括声明成员为 volatile，使用 synchronized 或者显式 Lock 控制所有该成员的访问。

思考一个有趣的现象：

```
byte b1=1;
byte b2=3;
byte b3=b1+b2;//当程序执行到这一行的时候会出错，因为b1、b2可以自动转换成int类型的变量，运算时java虚拟机对它进行了转换，结果导致把一个int赋值给byte-----出错
```

如果对b1 b2加上final就不会出错

```
final byte b1=1;
final byte b2=3;
byte b3=b1+b2;//不会出错，相信你看了上面的解释就知道原因了。
```