> 主要总结《Java并发编程实战》中“第3章:对象共享”的内容。

前言

在没有Java相关并发知识的前提下，第一次看这本书《Java并发编程实战》，其实有些看不太懂，因为里面的很多知识讲的比较抽象，比如可见性、volatile、final等讲的其实都不深入，所以导致自己理解的也很片面。后来就先专门看了“Java内存模型”相关的知识，再对相关知识理解起来，就要深入一些，所以才有了前面写的4篇关于“Java内存模型”相关的文章。

“第二章:基础知识”主要讲解线程安全性、原子性、加锁机制（主要讲解内置锁、synchronized重入）、用锁保护状态，这些知识在我相关系列的前面4篇中，已经讲的比较清楚，就直接跳过。

今天的这篇文章，主要是对《Java并发编程实战》中“第3章:对象共享”的内容进行总结，这章内容看了2遍，因为相关知识的匮乏，有些知识点还是理解的不全，所以也只能基于自己的理解，对所学内容总结一下，要不然很快就忘了，后续也会再重拾相关知识，再二次理解消化。

可见性

> 指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

在多线程环境下，一个线程对共享变量的操作对其他线程是不可见的。Java提供了volatile来保证可见性，当一个变量被volatile修饰后，表示着线程本地内存无效，当一个线程修改共享变量后他会立即被更新到主内存中，其他线程读取共享变量时，会直接从主内存中读取。当然，synchronize和Lock都可以保证可见性。synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。

> 评价：该小节内容主要是讲解的volatile的可见性，提到volatile不具备原子性特性，也引出了synchronize，但是对于可见性，final其实可见性是最强的。这部分内容中规中矩，建议大家将volatile、synchronize和final的可见性、原子性对比起来看，最好能理解他们可见性的原理，以及重排序的机制，这样就能对“可见性”这一概念理解的更加深入。

发布和溢出

发布：发布一个对象，是对象能够在当前作用域之外的代码中使用。比如将对象的引用保存到其它代码可以访问的地方，或者在一个非私有方法中返回对象的引用，简单来说，就是外部可以访问到这个对象和里面的成员或者方法，为了保证多线程下没有问题，需要保证对象内部状态的封装性不被破坏。

> 书中写的有点像八股文，讲的也有点抽象，我理解其实就拿到一个对象时，需要保证对象内部数据已经完全初始化，然后对象内部的成员和方法，要么就完全封装，不能将修改的方式对外暴露，如果不能做到这一点，就需要保证修改和访问是线程安全的。

溢出：当一个不应该发布的对象被发布时，这种情况被称为溢出（Escaspe）。溢出的情况我总结有2种：

- 将成员变量的引用返回，外部就可以修改数据，多线程下可能会存在问题；
- 在构造函数过程中使this引用溢出。

成员变量溢出

```
class UnsafeStates {
  private String[] states = new String[] {"AK","AL" ...
  public String[] getStates {return states;}
}
```

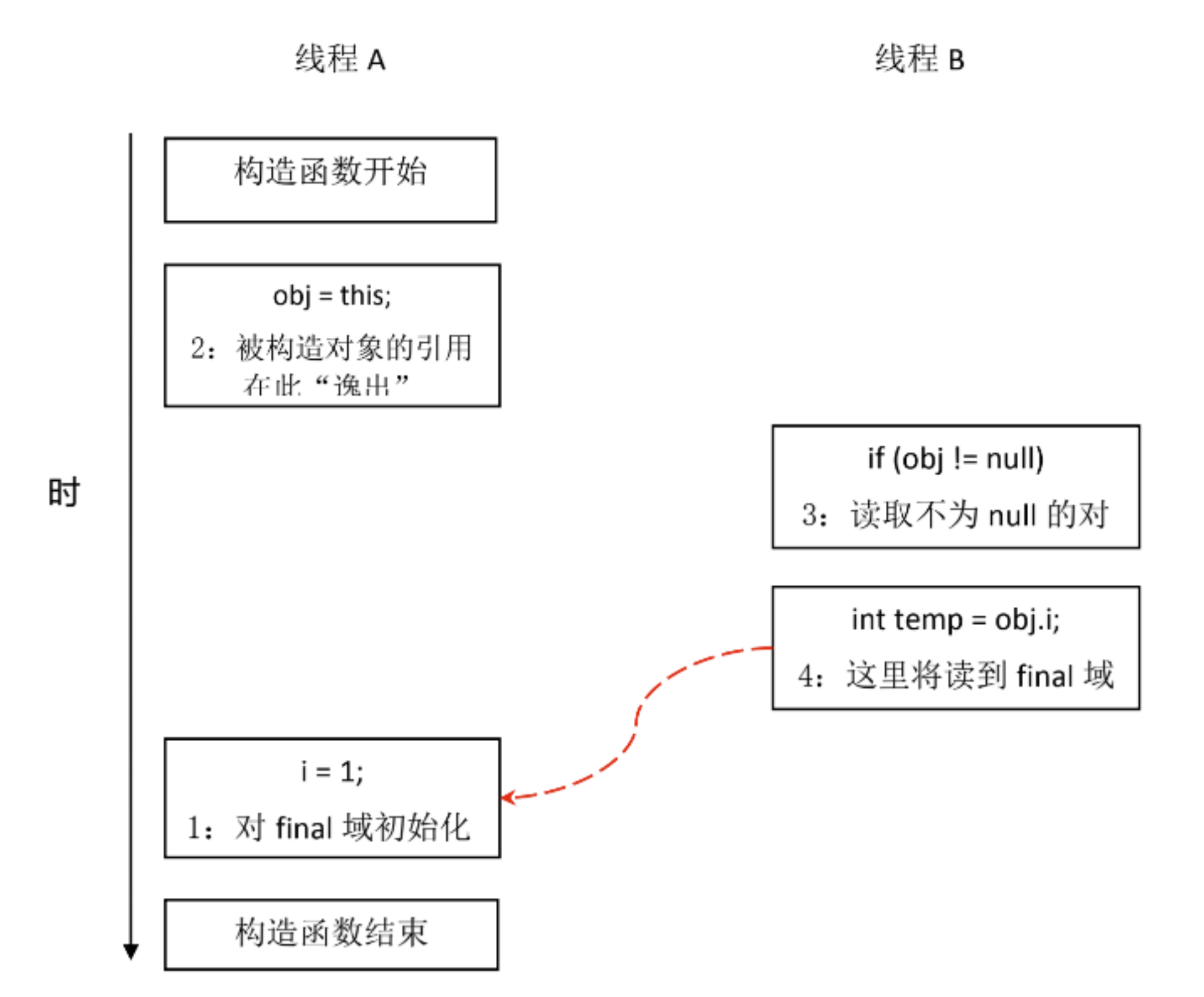
这个是书中的示例，当你发布这个对象时，任何调用者都可以修改数组中的内容，有人可能会说，我在states前面加一个final，这个其实解决不了问题，因为final可以保证states的地址不被改变，但是不能保证内部的数据不被改变。

this引用溢出

> 书中的注册示例没有看懂（还是自己太弱了。。。），我给一个前面一篇文章的示例，里面可以从原理讲解this引用是如何引出，这个应该比书中给的示例更能深入理解。

```
public class FinalReferenceEscapeExample {
  final int i;
  static FinalReferenceEscapeExample obj;
  public FinalReferenceEscapeExample () {
      i = 1;                              //1写final域
      obj = this;                          //2 this引用在此“逸出”
  }
  public static void writer() {
      new FinalReferenceEscapeExample ();
  }
  public static void reader {
      if (obj != null) {                     //3
          int temp = obj.i;                 //4
      }
  }
}
```

假设一个线程A执行writer()方法，另一个线程B执行reader()方法。这里的操作2使得对象还未完成构造前就为线程B可见。即使这里的操作2是构造函数的最后一步，且即使在程序中操作2排在操作1后面，执行read()方法的线程仍然可能无法看到final域被初始化后的值，因为这里的操作1和操作2之间可能被重排序。实际的执行时序可能如下图所示：



从上图我们可以看出：在构造函数返回前，被构造对象的引用不能为其他线程可见，因为此时的final域可能还没有被初始化。在构造函数返回后，任意线程都将保证能看到final域正确初始化之后的值。

线程封闭

文章讲解了“栈封闭”和“ThreadLocal类”这两部分内容：

- 栈封闭：“栈封闭”很好理解，类似于函数中的变量，函数结束后，变量内容自动释放，不会暴露对外，“栈封闭”其实讲的和这个原理类似，也就“栈封闭”的内容，肯定不会“溢出”。“栈封闭”可以很好的保证线程安全性。
- ThreadLocal类，这个其实和C++多线程的线程特定数据是一个道理，也就是每个线程可以有自己的ThreadLocal类，里面只保存本线程的数据，对其它线程不可见，一般该线程的全局变量，都可以保存到里面，其它线程不会干扰。

不变性

主要讲解final知识，强调了“不可变对象一定是线程安全的”，这个可以直接参考《Java并发编程系列4–final》这篇文章，里面的示例“使用volatile类型来发布不可变的对象”总结的不错，里面有一句话印象很深刻：

> 通过使用包含多个状态的容器对象来维持不变形条件，并使用一个volatile类型的引用来保证可见性，使得对象的在没有显示地使用锁的情况下，仍然是安全的。

这个怎么理解呢，书中的示例其实就是整了一个类OneValueCache和一个工厂Factory，工厂中有一个OneValueCache的成员变量，可以把Factory看成是获取OneValueCache对象的单例模式，如果需要这个单例模式对所有线程可见，就需要将该成员变量定义成volatile类型，保证所有线程可见，然后OneValueCache对象内部成员都是final，所以可以保证线程安全。上述方法总结一句话就是“通过final保证对象线程安全，通过volatile保证内存可见，实现多线性安全性”

安全发布

不安全发布

不正确的安全发布，会导致多线程运行时出现异常：

```
public class test {
  private int n;
  public void(int n) {
    this.n = n;
  }
  public void forTest() {
    if (n != n) {
      // 这里可能会进入，多线程下，因为重排序影响，this.n=n可能会排在构造函数完成外面，由于n!=n不能保证原子性，会出现问题。（如果还是不懂，建议先看一下重排序规则，里面很多类似的示例）
    }
  }
}
```

安全发布常用模式

书中总结了以下方法：

1. 在静态初始化函数中初始化一个对象的引用；
2. 将对象的引用保存在volatile类型的域或者AtomicReferance对象中；
3. 将对象的引用保存在某个正确构造对象的final类型域中；
4. 将对象的引用保存在一个由锁保护的域中。
   上面的4个方法，后面3个很好理解，对于第1个，因为静态初始器由JVM在类的初始化阶段执行，由于JVM内部存在这同步机制，因此通过这种方式初始化的任何对象都可以安全地发布：

```
public static Holder holder = new Holder(18);
```

其它概念

文章提到了“事实不可变对象”，这个概念有点绕，文中解释为”如果对象从技术上看是可变的，但是状态在发布后不会再改变“。然后给了个例子：

```
public Map<String, Data> test = Collections.synchronizedMap(new HashMap<String, Data>())
```

虽然Data对象可变，但是test不可变，也就是通过不可变容器或者其它方式，来装载可变对象，让其处理成不可变的方式。

> 有点绕，还是个八股文，就稍微了解即可。

对于可变对象，处理时就需要加锁。

总结

学习Java并发编程，前面的基础知识学了快2周，主要包括JMM、重排序规则、原子性、可见性和安全发布，很多知识都是围绕volatile、synchronize、final三者展开，之所以学了这么久，主要还是想把基础打牢，后面的应用总结和讲解，可能就没有这么细致。

学习交流

> 尽信书则不如无书，因个人能力有限，难免有疏漏和错误之处，如发现 bug 或者有更好的建议，欢迎批评指正，不吝感激。