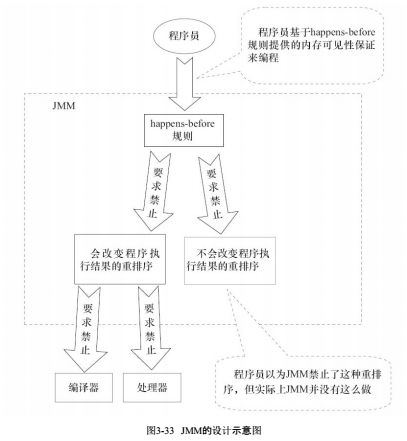
第三章

3.7 happens-before

3.7.1。JMM的设计

程序员希望有一个强内存模型来保证内存的可见性，而编译器和处理器则希望有一个若内存模型，从而可以尽可能多地来做优化，所以需要从中取个平衡点。

JMM把happens-before要求禁止的重排序分为了会改变程序结果的重排序和不会改变结果的重排序。



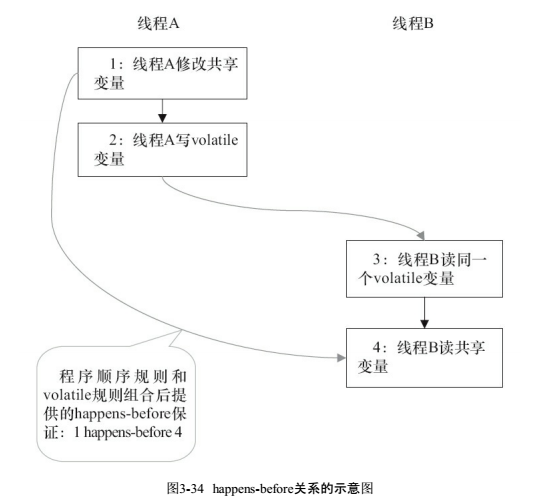
从图上我们可知JMM向程序员提供的happens-before规则能够满足程序员的需求,并且尽可能少地减少对编译器和处理器的束缚。

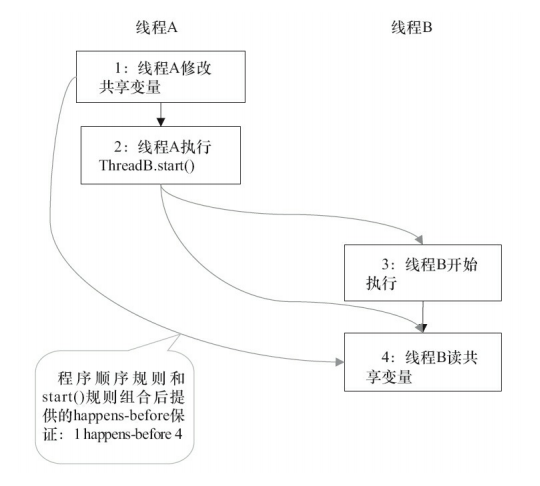
3.7.2 happens-before的定义

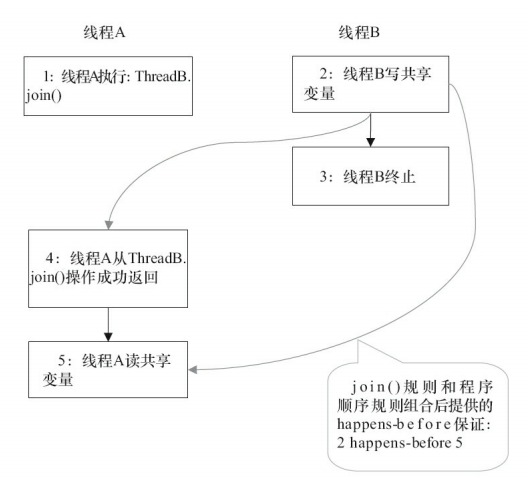
1. 如果一个操作happens-before另一个操作，那么这第一个操作的执行结果将对的第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前，这是JMM对程序员的承诺。
2. 两个操作之间存在happens-before关系，并不意味着Java平台的具体实现必须要按照happens-before关系指定的顺序来执行，如果重排序之后的执行结果与按照happens-before关系执行的结果一至那么允许这种重排序，这是JMM对编译器和处理器重排序的约束原则。
3. as-if-serial保证单线程内程序的执行结果不会被改变。
4. Happens-before保证正确同步的多线程程序的执行结果不会别改变。

3.7.3 happens-before规则

1. 程序顺序规则
2. 监视器锁规则
3. Volatile变量规则
4. 传递性
5. Start()规则
6. Join()规则







3.8双重检查锁定与延迟初始化

3.8.1 有时我们想要对高开销的对象在使用它时再初始化，所以有了延迟初始化。

下面来看一个线程不安全的延迟初始化代码

public class UnsafeLazyInitialization {  
 private static Instance instance;  
 public static Instance getInstance() {  
 if (instance == null) // 1：A线程执行  
 instance = new Instance(); // 2：B线程执行  
 return instance;  
 }  
}

再看一个线程安全的代码

public class SafeLazyInitialization {  
 private static Instance instance;  
 public synchronized static Instance getInstance() {  
 if (instance == null) // 1：A线程执行  
 instance = new Instance(); // 2：B线程执行  
 return instance;  
 }  
}

但是这种方法在频繁调用的时候性能会下降。

于是就又有了双检测锁定方法来降低同步开销。

public class DoubleCheckedLocking { // 1

private static Instance instance; // 2

public static Instance getInstance() { // 3

if (instance == null) { // 4:第一次检查

synchronized (DoubleCheckedLocking.class) { // 5:加锁

if (instance == null) // 6:第二次检查

instance = new Instance(); // 7:问题的根源出在这里

} // 8

} // 9

return instance; // 10

} // 11

}

多个线程试图在同一时间创建对象时，会通过加锁来保证只有一个线程能创建对象。  
 在对象创建好之后，执行getInstance()方法将不需要获取锁，直接返回已创建好的对象。

3.8.2 问题根源

创建一个对象要经过以下几个步骤：

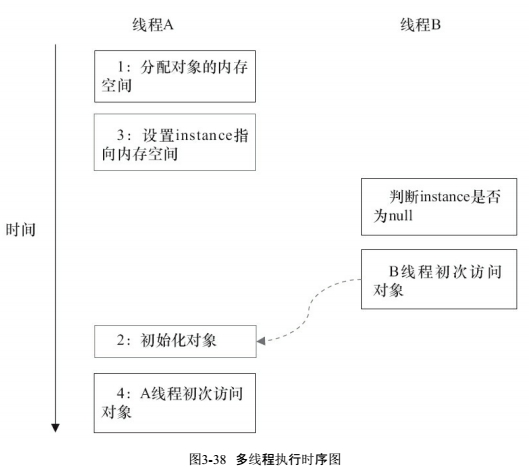
memory = allocate(); // 1：分配对象的内存空间

ctorInstance(memory); // 2：初始化对象

instance = memory; // 3：设置instance指向刚分配的内存地址

在某些编译器上2，3会重排序，遵守了intra-thread规则，保证重排序不会改变单线程内程序的执行结果。

所以会出现下面这种情况：



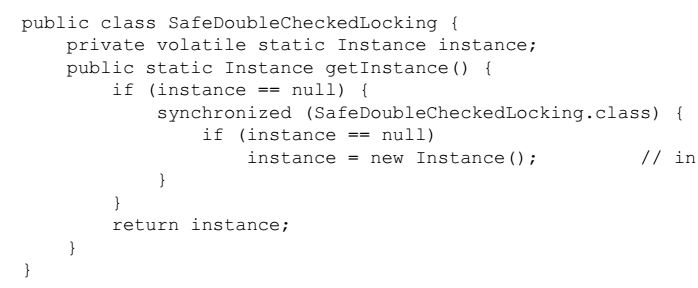
B线程将会看到一个没有被初始化的对象。

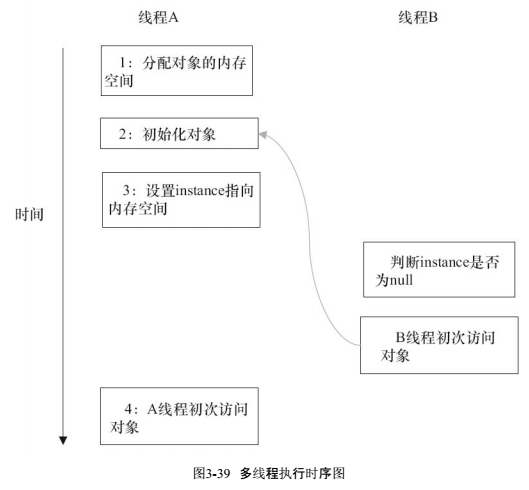
解决方案：

1）不允许2和3重排序。

2）允许2和3重排序，但不允许其他线程“看到”这个重排序。

3.8.3 基于volatile的解决方案





3.8.4 基于类初始化的解决方案：

在执行类的初始化期间，JVM会去获取一个锁。这个锁可以同步多个线程对同一个类的初始化。

public class InstanceFactory {

private static class InstanceHolder {

public static Instance instance = new Instance();

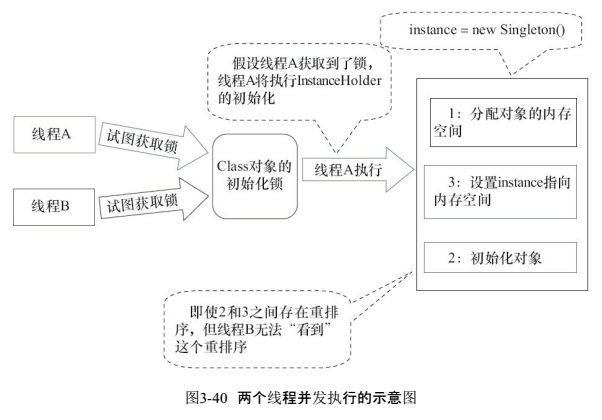
}p

ublic static Instance getInstance() {

return InstanceHolder.instance ; // 这里将导致InstanceHolder类被初始化

}

}



初始化一个类，包括执行这个类的静态初始化和初始化在这个类中声明的静态字段。根

据Java语言规范，在首次发生下列任意一种情况时，一个类或接口类型T将被立即初始化。

1）T是一个类，而且一个T类型的实例被创建。

2）T是一个类，且T中声明的一个静态方法被调用。

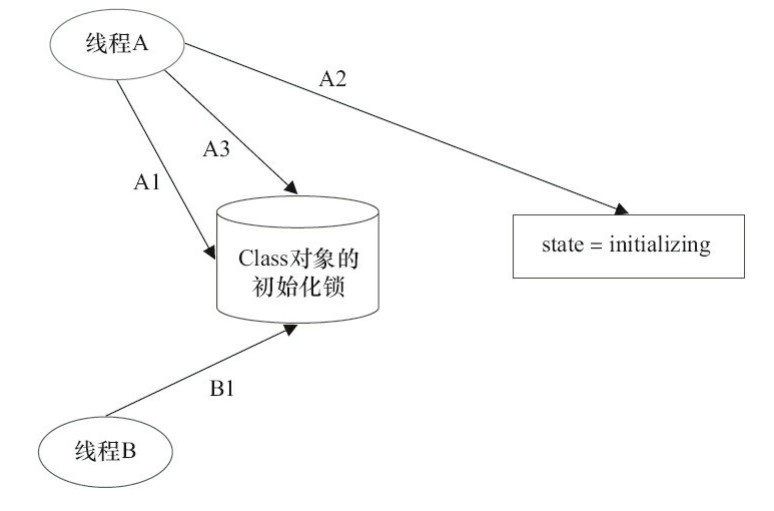
3）T中声明的一个静态字段被赋值。

4）T中声明的一个静态字段被使用，而且这个字段不是一个常量字段。

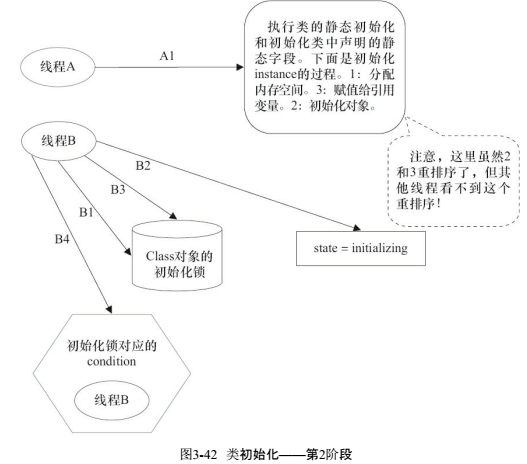
5）T是一个顶级类（Top Level Class，见Java语言规范的§7.6），而且一个断言语句嵌套在T内部被执行。

类初始化的过程：

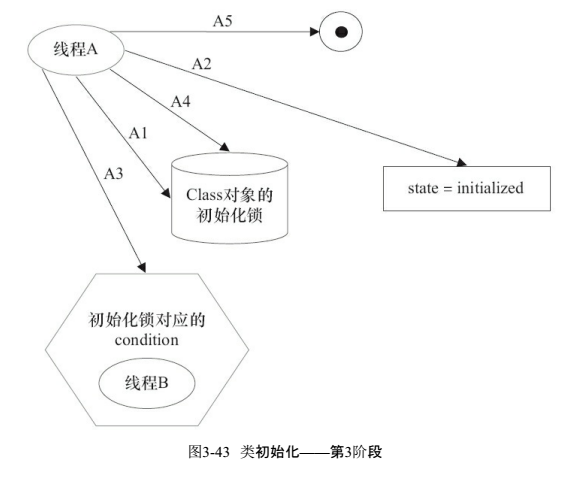
第1阶段：通过在Class对象上同步（即获取Class对象的初始化锁），来控制类或接口的初始化。这个获取锁的线程会一直等待，直到当前线程能够获取到这个初始化锁。



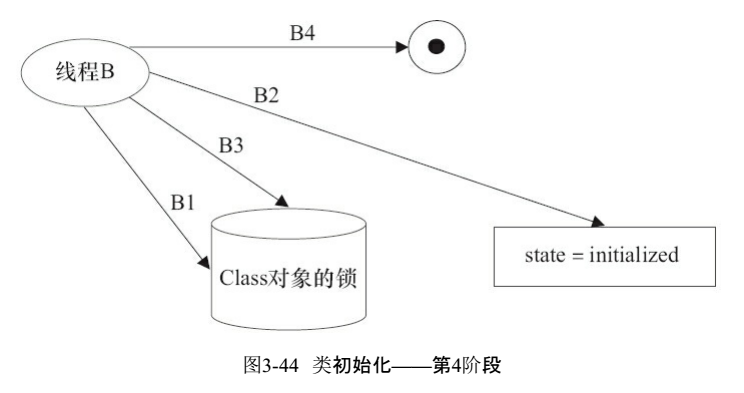
第2阶段：线程A执行类的初始化，同时线程B在初始化锁对应的condition上等待。

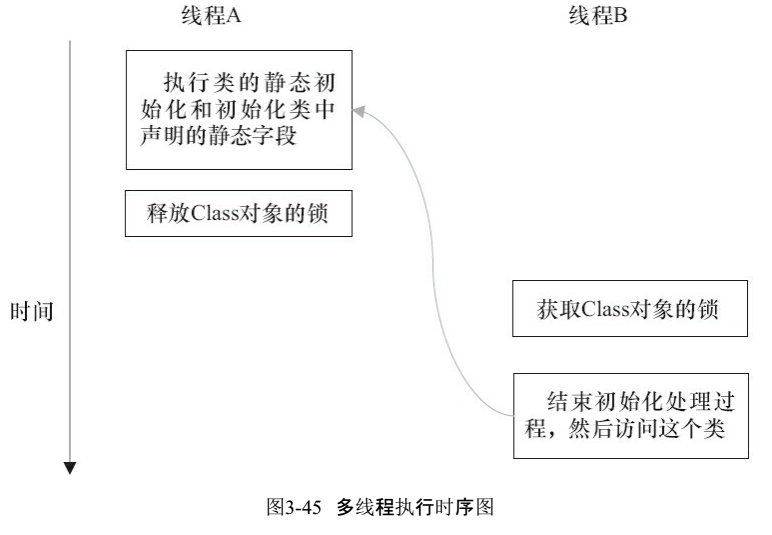


第3阶段：线程A设置state=initialized，然后唤醒在condition中等待的所有线程。

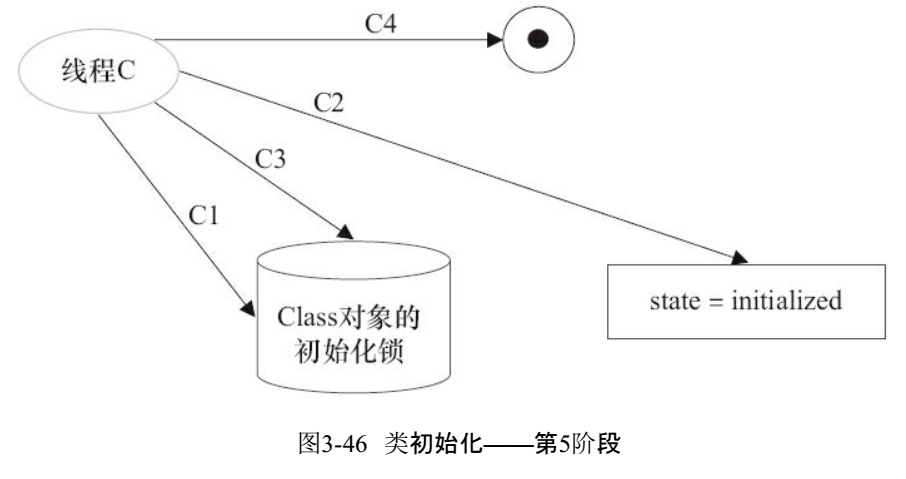


第4阶段：线程B结束类的初始化处理。





第5阶段：线程C执行类的初始化的处理。



这个happens-before关系将保证：线程A执行类的初始化时的写入操作，线程C一定能看。

如果确实需要对实例字段使用线程安全的延迟初始化，请使用上面介绍的基于volatile的延迟初始化的方案；如果确实需要对静态字段使用线程安全的延迟初始化，请使用上面介绍的基于类初始化的方案。