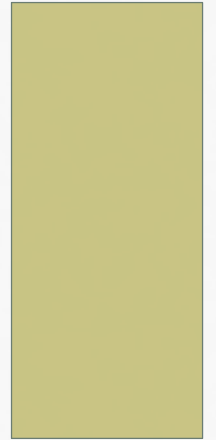


Java Memory Model



JMM

Java Memory Model (JMM) описывает поведение потоков в среде исполнения Java. Модель памяти — часть семантики языка Java, и описывает, на что может и на что не должен рассчитывать программист, разрабатывающий ПО не для конкретной Java-машины, а **для Java в целом**.

Исходная модель памяти Java, разработанная в 1995 году, считается неудачной: многие оптимизации невозможно провести, не потеряв гарантию, что код безопасен.

В J2SE 5.0 (30 сентября 2004) появилась новая модель памяти, разработанная через Java Community Process под названием JSR-133. Она лучше отражала принципы работы современных процессоров и компиляторов, и другие языки брали идеи из модели Java (схожая модель используется в C++11).

Модель памяти

На уровне процессора, модель памяти определяет необходимые и достаточные условия для гарантии того, что записи в память другими процессорами будут видны текущему процессору, и записи текущего процессора будут видимы другими процессорами.

Некоторые процессоры демонстрируют **сильную** модель памяти, где все процессоры всегда видят точно одинаковые значения для любой заданной ячейки памяти.

Другие процессоры демонстрируют более **слабую** модель памяти, где специальные инструкции, называемые барьерами памяти, требуются для «сброса» (flush) или объявления недействительными (invalidate) данных в локальном кэше процессора, с целью сделать записи данного процессора видимыми для других или увидеть записи, сделанные другими процессорами.

Видимость (visibility)

Один поток может в какой-то момент временно сохранить значение некоторых полей не в основную память, а в регистры или локальный кэш процессора, таким образом второй поток, выполняемый на другом процессоре, читая из основной памяти, может не увидеть последних изменений поля. И наоборот, если поток на протяжении какого-то времени работает с регистрами и локальными кэшами, читая данные оттуда, он может сразу не увидеть изменений, сделанных другим потоком в основную память.

Reordering

Для увеличения производительности процессор/компилятор могут переставлять местами некоторые инструкции. С точки зрения потока, наблюдающего за выполнением операций в другом потоке, операции могут быть выполнены не в том порядке, в котором они идут в исходном коде.

Так же эффект может наблюдаться, когда один поток кладет результаты первой операции в локальный кэш, а результат второй операции кладет непосредственно в основную память. Тогда второй поток, обращаясь к основной памяти может сначала увидеть результат второй операции, и только потом первой, когда все кэши синхронизируются.

Reordering

Еще одна причина reordering, может заключаться в том, что процессор (или компилятор!) может решить поменять порядок выполнения операций, если, например, сочтет что такая последовательность выполнится быстрее и не изменит семантику программы.

Happens-before

В Java Memory Model введена такая абстракция как happens-before. Она обозначает, что если операция X связана отношением happens-before с операцией Y, то весь код следуемый за операцией Y, выполняемый в одном потоке, видит все изменения, сделанные другим потоком, до операции X.

Связь happens-before транзитивна, т.е. если X happens-before Y, а Y happens-before Z, то X happens-before Z.

Happens-before

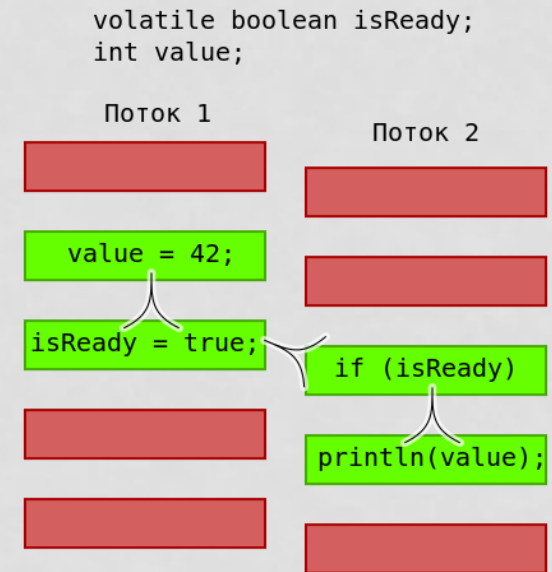
Синхронизация и мониторы:

- Захват монитора (начало `synchronized`, метод `lock`) и всё, что после него в том же потоке.
- Возврат монитора (конец `synchronized`, метод `unlock`) и всё, что перед ним в том же потоке.
 - Таким образом, оптимизатор может заносить строки в синхроблок, но не наружу.
- Возврат монитора и последующий захват другим потоком.

Happens-before

Запись и чтение:

- Любые зависимости по данным (то есть запись в любую переменную и последующее чтение её же) в одном потоке.
- В одном потоке перед записью в `volatile`-переменную, и сама запись.
- `volatile`-чтение и всё, что после него в том же потоке.
- Запись в `volatile`-переменную и последующее считывание её же.
 - Для объектных переменных (например, `volatile List x`;) столь сильные гарантии выполняются для ссылки на объект, но не для его содержимого.



Happens-before

Обслуживание объекта:

- Статическая инициализация и любые действия с любыми экземплярами объектов.
- Запись в `final`-поля в конструкторе и всё, что после конструктора.
 - Как исключение из всеобщей транзитивности, это соотношение `happens-before` не соединяется транзитивно с другими правилами и поэтому может вызвать межпоточную гонку.
- Любая работа с объектом и `finalize()`.

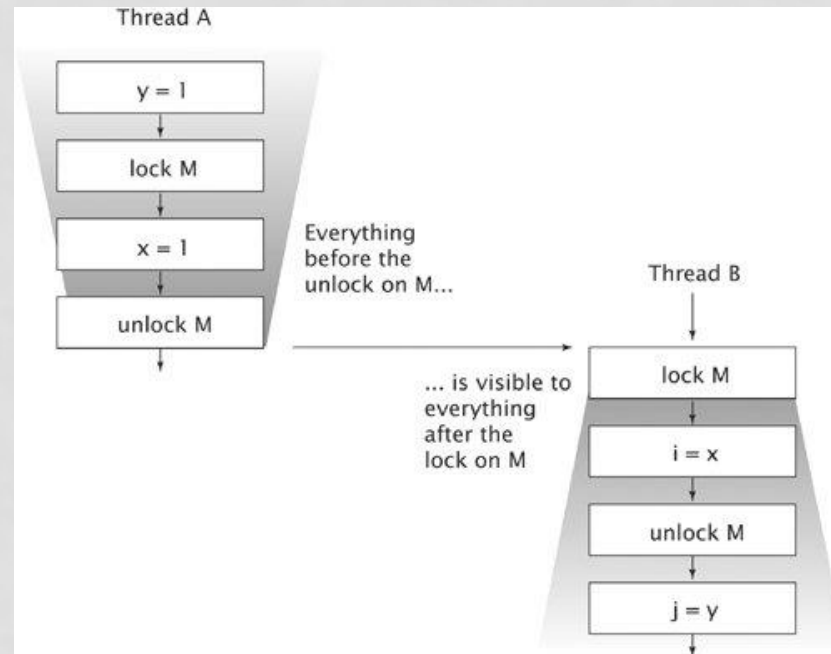
Happens-before

Обслуживание потока:

- Запуск потока и любой код в потоке.
- Зануление переменных, относящихся к потоку, и любой код в потоке.
- Код в потоке и `join()`; код в потоке и `isAlive() == false`.
- `interrupt()` потока и обнаружение факта останова.

Happens-before

В отношении happens-before есть очень большой дополнительный бонус: данное отношение дает не только видимость `volatile` полей или результатов операций защищенных монитором или локом, но и видимость вообще всего, что делалось до события happens-before.



Публикация объекта

Публикацией объектов называется явление, когда один поток создает объект и присваивает на него ссылку какому-нибудь полю, которое может увидеть второй поток.

Если запись в это поле первым потоком, разделена со чтением этого поля вторым потоком отношением `happens-before`, то публикация называется **безопасной**, т.е. второй поток увидит все поля опубликованного объекта, инициализированные первым потоком.

Есть еще один способ добиться безопасной публикации объектов: если ссылка на объект, все поля которого являются `final`, становится видимой любому потоку, то данный поток видит все `final` поля, инициализированные во время создания объекта. Более того он будет видеть все значения достижимые из `final` полей.

Публикация объекта

```
public class AlwaysSafePublished{  
    private final Map map = new HashMap();
```

```
    public AlwaysSafePublished{  
        Collection c = new ArrayList();  
        c.add("a");  
        c.add("A");  
        map.put("1", c)  
    }
```

```
    public int number(){  
        return map.get("1").size();  
    }  
}
```

Публикация объекта

Если кроме как ссылок `final` на ваши объекты никто не ссылается, то **не зависимо от уровня вложенности**, поток, который видит ссылку на опубликованный объект, увидит все значения достижимые через `final` поля, которые были выставлены в конструкторе.

Так в примере, любой поток, успешно зашедший в метод `number`, всегда вернет значение 2.

Конечно при условии, что после конструктора содержимое всех объектов больше не модифицируется. 😊

Это делает singleton на double-check работоспособным.

Все это верно только для объектов, во время конструирования которых, **ссылка на объект не покидет конструктор, прежде чем он завершен.**

Атомарность записи-чтения полей

JMM гарантирует атомарность записи-чтения всех не long/double полей. А volatile - абсолютно всех полей.

Поля, представляющие ссылки на объекты, тоже всегда пишутся-читаются атомарно.

Спецификация не запрещает иметь атомарность записи чтения long\double полей для 64-битной виртуальной машины.

Данная атомарность гарантирует, что любой поток в любой момент времени зачитает из поля либо значение по умолчанию, либо полное значение, записанное туда в некий момент времени, и никогда не найдет там какого-то мусора.

JCStress

```
public class VolatileTest {
    @JCStressTest
    @State
    @Outcome(id = "[1, 2]", expect = Expect.ACCEPTABLE, desc = "T1 updated, then T2 updated.")
    @Outcome(id = "[2, 1]", expect = Expect.ACCEPTABLE, desc = "T2 updated, then T1 updated.")
    @Outcome(id = "[1, 1]", expect = Expect.ACCEPTABLE, desc = "Both T1 and T2 updated
    concurrently.")
        public static class VolatileIncrementAtomicityTest {
            volatile int v;

            @Actor
            public void actor1(IntResult2 r) {
                r.r1 = ++v;
            }

            @Actor
            public void actor2(IntResult2 r) {
                r.r2 = ++v;
            }
        }
}
```

(fork: #1, iteration #1, JVM args: [-server])

Observed state Occurrences Expectation Interpretation

1, 1	1,543,069	ACCEPTABLE	Both T1 and T2 updated concurrently.
1, 2	29,034,989	ACCEPTABLE	T1 updated, then T2 updated.
2, 1	26,223,172	ACCEPTABLE	T2 updated, then T1 updated.

Мифы

Миф-1

- Компьютер делает ровно то, что мы его просим

```
int m() {  
    int a = 42;  
    int b = 13;  
    int r = a + b;  
    return r;  
}
```

Миф-1

- Компьютер делает ровно то, что мы его просим

```
int m() {  
    int a = 42;  
    int b = 13;  
    int r = a + b;  
    return r;  
}
```

```
mov %eax, 55;  
ret
```

Миф-1

- Оптимизация есть во всех языках.
- Наблюдаемый результат выполнения – один из результатов выполнения абстрактной машины.
- Внутри может происходить все что угодно.
- Когда происходит `debug`, дебаггер пытается реконструировать пошаговое выполнение. В случае с Java дебаггер обычно эмулирует состояние абстрактной Java-машины, нежели лезет во внутреннюю работу JVM

Миф-1

Таким образом, если JMM говорит о каком-то порядке выполнения это не значит, что физическая имплементация выполнения кода не может опускать какие-то инструкции...

```
volatile int x;  
void m() {  
    x = 1;  
    x = 2;  
    System.out.println(x);  
}
```

Миф-1

Таким образом, если JMM говорит о каком-то порядке выполнения это не значит, что физическая имплементация выполнения кода не может опускать какие-то инструкции...

```
volatile int x;  
void m() {  
    x = 1;  
    x = 2;  
    System.out.println(x);  
}
```

```
mov %eax, 2 # first argument  
call System_out_println
```


Миф-2

Барьеры – действительно барьеры

```
@JCStressTest
@State
public class SynchronizedBarriers {
    int x, y;

    @Actor
    void actor() {
        synchronized (this) {
            x = 1;
        }
        synchronized (this) {
            y = 1;
        }
    }

    @Actor
    void observer(IntResult2 r) {
        // Caveat: get_this_in_order()-s happen in program order
        r.r1 = get_this_in_order(y);
        r.r2 = get_this_in_order(x);
    }
}
```

Миф-2

(fork: #1, iteration #1, JVM args: [-server, -XX:+UnlockDiagnosticVMOptions, -XX:+StressLCM, -XX:+StressGCM])

Observed state	Occurrences	Expectation	Interpretation
0, 0	43,558,372	ACCEPTABLE	All other cases are acceptable.
0, 1	22,512	ACCEPTABLE	All other cases are acceptable.
1, 0	1,565	INTERESTING	X and Y are visible in different order
1, 1	1,372,341	ACCEPTABLE	All other cases are acceptable.

Миф-3

Нельзя мыслить категориями «локальный кеш потока» и «главная память»

Миф-3

```
@JCStressTest
@State
class IRIW {
    int x;
    int y;

    @Actor
    void writer1() {
        x = 1;
    }

    @Actor
    void writer2() {
        y = 1;
    }

    @Actor
    void reader1(IntResult4 r) {
        r.r1 = x;
        r.r2 = y;
    }

    @Actor
    void reader2(IntResult4 r) {
        r.r3 = y;
        r.r4 = x;
    }
}
```

Миф-3

```
@JCStressTest
```

```
@State
```

```
class IRIW {
```

```
    int x;
```

```
    int y;
```

```
    @Actor
```

```
    void writer1() {
```

```
        x = 1;
```

```
    }
```

```
    @Actor
```

```
    void writer2() {
```

```
        y = 1;
```

```
    }
```

```
    @Actor
```

```
    void reader1(IntResult4 r) {
```

```
        r.r1 = x;
```

```
        r.r2 = y;
```

```
    }
```

```
    @Actor
```

```
    void reader2(IntResult4 r) {
```

```
        r.r3 = y;
```

```
        r.r4 = x;
```

```
    }
```

```
}
```

1, 0, 1, 0 152 ACCEPTABLE Threads see the updates in the inconsistent order

```
@JCStressTest
@State
public class FencedIRIW {

    int x;
    int y;

    @Actor
    public void actor1() {
        UNSAFE.fullFence();
        x = 1;
        UNSAFE.fullFence();
    }

    @Actor
    public void actor2() {
        UNSAFE.fullFence();
        y = 1;
        UNSAFE.fullFence();
    }

    @Actor
    public void actor3(IntResult4 r) {
        UNSAFE.loadFence();
        r.r1 = x;
        UNSAFE.loadFence();
        r.r2 = y;
        UNSAFE.loadFence();
    }

    @Actor
    public void actor4(IntResult4 r) {
        UNSAFE.loadFence();
        r.r3 = y;
        UNSAFE.loadFence();
        r.r4 = x;
        UNSAFE.loadFence();
    }
}
```

```

@JCStressTest
@State
public class FencedIRIW {

    int x;
    int y;

    @Actor
    public void actor1() {
        UNSAFE.fullFence();
        x = 1;
        UNSAFE.fullFence();
    }

    @Actor
    public void actor2() {
        UNSAFE.fullFence();
        y = 1;
        UNSAFE.fullFence();
    }

    @Actor
    public void actor3(IntResult4 r) {
        UNSAFE.loadFence();
        r.r1 = x;
        UNSAFE.loadFence();
        r.r2 = y;
        UNSAFE.loadFence();
    }

    @Actor
    public void actor4(IntResult4 r) {
        UNSAFE.loadFence();
        r.r3 = y;
        UNSAFE.loadFence();
        r.r4 = x;
        UNSAFE.loadFence();
    }
}

```

1, 0, 1, 0

47

ACCEPTABLE Threads see the updates in the inconsistent order

```
@JCStressTest
@State
public class VolatileIRIW {
    volatile int x, y;

    @Actor
    public void actor1() {
        x = 1;
    }

    @Actor
    public void actor2() {
        y = 1;
    }

    @Actor
    public void actor3(IntResult4 r) {
        r.r1 = x;
        r.r2 = y;
    }

    @Actor
    public void actor4(IntResult4 r) {
        r.r3 = y;
        r.r4 = x;
    }
}
```



```
@JCStressTest
```

```
@State
```

```
public class VolatileIRIW {  
    volatile int x, y;
```

```
    @Actor
```

```
    public void actor1() {  
        x = 1;  
    }
```

1, 0, 1, 0 0 ACCEPTABLE
Threads see the updates in the inconsistent order

```
    @Actor
```

```
    public void actor2() {  
        y = 1;  
    }
```

```
    @Actor
```

```
    public void actor3(IntResult4 r) {  
        r.r1 = x;  
        r.r2 = y;  
    }
```

```
    @Actor
```

```
    public void actor4(IntResult4 r) {  
        r.r3 = y;  
        r.r4 = x;  
    }
```

```
}
```

