Java Memory Model

JSP & Servlets
18 уровень, 2 лекция

ОТКРЫТА

Знакомство с Java Memory Model

Модель памяти Java (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 (к которой, в частности, относится “потоколокальная память”), разработанная в 1995 году, считается неудачной: многие оптимизации невозможно провести, не потеряв гарантию безопасности кода. В частности, есть несколько вариантов написать многопоточного “одиночку”:

- либо каждый акт доступа к одиночке (даже когда объект давно создан, и ничего уже не может измениться) будет вызывать межпоточную блокировку;
- либо при определенном стечении обстоятельств система выдаст недостроенного одиночку;
- либо при определенном стечении обстоятельств система создаст два одиночки;
- либо конструкция будет зависеть от особенностей поведения той или иной машины.

Поэтому механизм работы памяти был переработан. В 2005 году, с выходом Java 5 был презентован новый подход, который был еще улучшен с выходом Java 14.

В основе новой модели лежат три правила:

Правило № 1: однопоточные программы исполняются псевдопоследовательно. Это значит: в реальности процессор может выполнять несколько операций за такт, заодно изменив их порядок, однако все зависимости по данным остаются, так что поведение не отличается от последовательного.

Правило № 2: нет невесть откуда взявшихся значений. Чтение любой переменной (кроме не-volatile long и double, для которых это правило может не выполняться) выдаст либо значение по умолчанию (ноль), либо что-то, записанное туда другой командой.

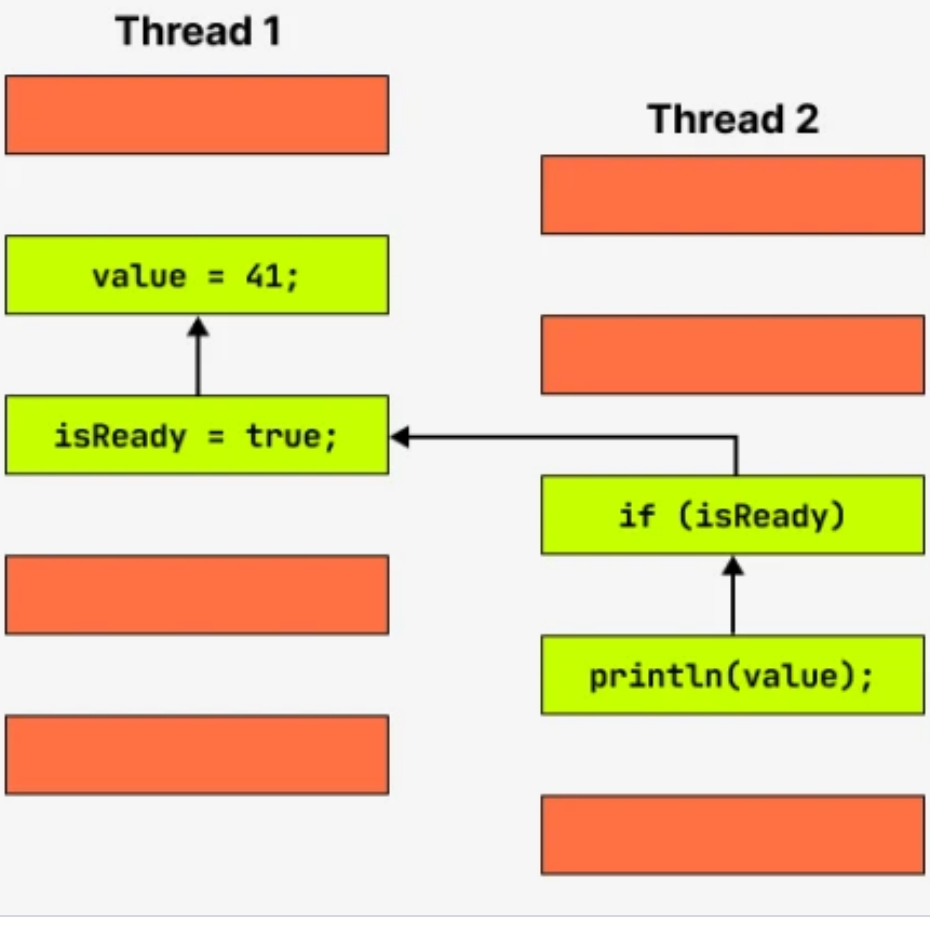
И **правило № 3**: остальные события выполняются по порядку, если связаны отношением строгого частичного порядка “выполняется прежде” (**happens before**).

Happens before

Лесли Лэмпорт придумал понятие **Happens before**. Это отношение строгого частичного порядка, введенное между атомарными командами (++ и -- не атомарны) и не означающее “физически прежде”.

Оно говорит о том, что вторая команда будет “в курсе” изменений, проведенных первой.

```
volatile boolean isReady;
int value;
```



Например, одно выполняется прежде другого для таких операций:

Синхронизация и мониторы:

- Захват монитора (метод `lock`, начало synchronized) и все, что происходит в том же потоке после него.
- Возврат монитора (метод `unlock`, конец synchronized) и все, что происходит в том же потоке перед ним.
- Возврат монитора и последующий захват другим потоком.

Запись и чтение:

- Запись в любую переменную и последующее чтение ее же в одном потоке.
- Все, что в том же потоке перед записью в volatile-переменную, и сама запись. volatile-чтение и все, что в том же потоке после него.
- Запись в volatile-переменную и последующее считывание ее же. Volatile-запись взаимодействует с памятью так же как и возврат монитора, а чтение как захват. Получается, что если один поток записал в volatile-переменную, а второй обнаружил это, все, что предшествует записи, выполняется раньше всего, что идет после чтения; смотри рисунок.

Обслуживание объекта:

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

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 остановки.

Нюансы работы Happens before

Освобождение (releasing) монитора happens-before происходит прежде, чем получение (acquiring) того же монитора. Стоит обратить внимание, что именно освобождение, а не выход, то есть за безопасность при использовании wait можно не беспокоиться.

Рассмотрим, как это знание поможет нам исправить наш пример. В данном случае все очень просто: достаточно убрать внешнюю проверку и оставить синхронизацию как есть. Теперь второй поток гарантированно увидит все изменения, потому

что он получит монитор только после того, как другой поток его отпустит. А так как он его не отпустит, пока все не проинициализирует, мы увидем все изменения сразу, а не по отдельности:

```
public class Keeper {
    private Data data = null;

    public Data getData() {
        synchronized(this) {
            if(data == null) {
                data = new Data();
            }
        }

        return data;
    }
}
```

Запись в volatile переменную happens-before чтение из той же переменной. То изменение, которое мы внесли, конечно, исправляет некорректность, но возвращает того, кто написал изначальный код, туда, откуда он пришел — к блокировке каждый раз. Спасти может ключевое слово volatile. Фактически, рассматриваемое утверждение значит, что при чтении всего, что объявлено volatile, мы всегда будем получать актуальное значение.

Кроме того, как я говорил раньше, для volatile полей запись всегда (в том числе long и double) является атомарной операцией. Еще один важный момент: если у вас есть volatile сущность, имеющая ссылки на другие сущности (например, массив, List или какой-нибудь еще класс), то всегда “свежей” будет только ссылка на саму сущность, но не на все, в нее входящее.

Итак, обратно к нашим Double-locking баранам. С использованием volatile исправить ситуацию можно так:

```
public class Keeper {
    private volatile Data data = null;

    public Data getData() {
        if(data == null) {
            synchronized(this) {
                if(data == null) {
                    data = new Data();
                }
            }
        }
        return data;
    }
}
```

Тут у нас по-прежнему есть блокировка, но только в случае, если data == null. Остальные случаи мы отсеиваем, используя volatile read. Корректность обеспечивается тем, что volatile store happens-before volatile read, и все операции, которые происходят в конструкторе, видны тому, кто читает значение поля.

‹ Предыдущая лекция

Следующая лекция ›

Комментарии (3) популярные новые старые

JavaCoder

Введите текст комментария

Валера Калиновский Java Developer 31 июля 2022, 18:40

хотелось бы побольше примеров из жизни. По сути в конце написан код очень похожий на trade-safe код для создания синглтона. Вот уж никогда не думал что в этом коде использовался принцип happens before. Почему то никто об этом не упоминает когда разбирает этот код. А что было бы еслиб не было этого принципа? Можно привести код в котором чтото ломается из-за того что нет happens before

Ответить +2

Anonymous #3076791 Уровень 3 14 августа 2022, 23:54

я думал, я один сюда дошел :)

Ответить +2

Oleg Khilko Уровень 51 16 августа 2022, 15:09

1) Многопоточность в Java: основы

2) Многопоточность в Java: средства стандартной библиотеки

Рекомендую посмотреть эти две лекции чтобы лучше понять.

Ответить +5

Русский

СКАЧИВАЙТЕ НАШИ ПРИЛОЖЕНИЯ

ДОСТУПНО В Google Play

Загрузите в App Store