Близкие контакты JMM-ной степени

«ничего не понятно, но что-то это значит»

Aleksey Shipilёv
shade@redhat.com, @shipilev
@shipilev

Safe Harbor / Тихая Гавань

Everything on this and any subsequent slides may be a lie. Do not base your decisions on this talk. If you do, ask for professional help.

Всё что угодно на этом слайде, как и на всех следующих, может быть враньём. Не принимайте решений на основании этого доклада. Если всё-таки решите принять, то наймите профессионалов.

дисклеймеры

Этот доклад:

1. ...рассказывает про многопоточность, а не сборку мусора. *(В стопитцотый раз уже, я уже сам всё понял. Уходите.)*
2. ...требует понимания модели на уровне «Прагматики JMM» *(За пять минут и на пальцах не объяснить, уходите)*
3. ...испещрён ядом, фейспалмами и желчью. Без шуток. *(Нет, мы не будем JLS читать под «кумбайя», уходите)*
4. ...содержит всякие стрелочки, порядки, разнарядки. *(Если не умеете в простую математику, то... уходите)*

Введение

Введение: абстрактные машины

«Машины делают то, что я говорю им делать»

```
public int m() {
  int x = 42;
  int y = 34;
  int t = x + y;
  return t;
}
```

$\Rightarrow$

```
m:
  ...prolog...
  mov $76$, %rax
  ...epilog...
  ret
```

Если результат не отличается от поведения абстрактной машины, никого не волнует, как он достигнут!

Введение: JMM – про абстрактные машины!

Если результат не отличается от специфицированного, никого не волнует, как он достигнут!

```
volatile int x;
public int m() {
  x = 1;
  x = 2;
  return x;
}
```

$\Rightarrow$

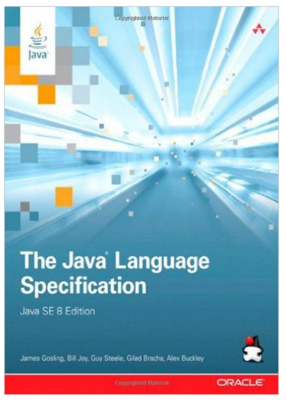
```
m:
  ...prolog...
  mov $2$, (mem)
  mov $2$, %rax
  ...epilog...
  ret
```

(На практике, не все оптимизации тривиальны, и могут потребовать невозможного)

Барьеры

Барьеры: проблема

«Ага, сейчас я быстренько прочитаю про JMM!»



Given a write w, a freeze f, an action a (that is not a read of a `final` field), a read r_1 of the `final` field frozen by f, and a read r_2 such that $hb(w, f)$, $hb(f, a)$, $mc(a, r_1)$, and $dereferences(r_1, r_2)$, then when determining which values can be seen by r_2, we consider $hb(w, r_2)$. (This *happens-before* ordering does not transitively close with other *happens-before* orderings.)

An execution E is described by

- P - a program
- A - a set of actions
- po - program order, whi… performed by t in A
- so - synchronization order…

- Well-formed executions E_1, ..., where E_i = < $P, A_i, po_i, so_i, W_i, V_i, sw_i, hb_i$ >.

Given these sets of actions C_0, ... and executions E_1, ... , every action in C_i must be one of the actions in E_i. All actions in C_i must share the same relative happens-before order and synchronization order in both E_i and E. Formally:

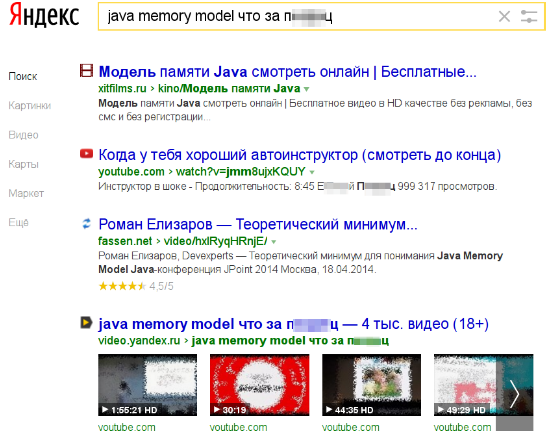
1. C_i is a subset of A_i

…ne in both E_i and E. Only the … Formally:

- There exists a set O of actions such that B consists of a *hang* action plus all the external actions in O and for all $k \geq |O|$, there exists an execution E of P with actions A, and there exists a set of actions O' such that:
  - Both O and O' are subsets of A that fulfill the requirements for sets of observable actions.
  - $O \subseteq O' \subseteq A$
  - $|O'| \geq k$

Барьеры: проблема

«Ё-моё, ну в интернетах-то наверняка пишут что-то...»



Барьеры: JSR 133 Cookbook

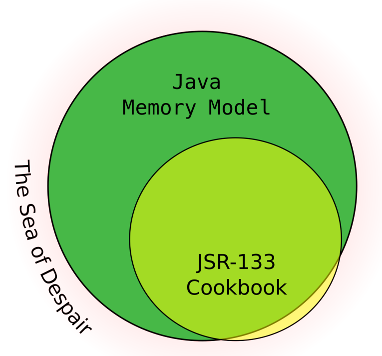
```
...
volatile load
[LoadLoad|LoadStore]
...
```

```
[StoreStore|LoadStore]
volatile store
[StoreLoad]
```

- Вот оно, богатство, и никакой тебе зауми!
- Всё понятно же, вокруг доступов барьеры:
  1. Нельзя переставить доступ **за** `volatile store`
  2. Нельзя переставить доступ **перед** `volatile load`
  3. (1), (2) аналогично для `synchronized enter/exit`
  4. В конце конструктора барьер, запрещающий переставлять

Барьеры: засада с консервативностью

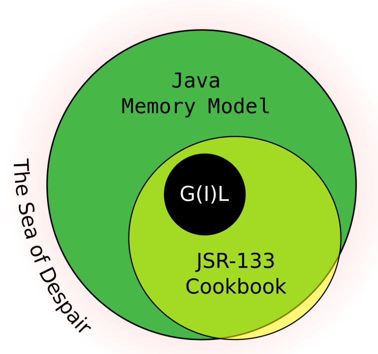
Засада №1: JSR 133 Cookbook описывает *консервативную* реализацию, практический минимум



- Конформный рантайм не обязан следовать Cookbook, а обязан следовать спецификации
- Отдельные оптимизации уходят от Cookbook, и доказывают свою корректность через JMM

Барьеры: засада с консервативностью

Засада №1: JSR 133 Cookbook описывает *консервативную* реализацию, практический минимум



- Конформный рантайм не обязан следовать Cookbook, а обязан следовать спецификации
- Отдельные оптимизации уходят от Cookbook, и доказывают свою корректность через JMM

Барьеры: lock coarsening

```
void m() {
  synchronized(this) {
    x = 1;
  }
  synchronized(this) {
    y = 1;
  }
}
```

Cookbook →

```
void m() {
  [LoadStore]   // monitorenter
  x = 1;
  [StoreStore]  // monitorexit
  [StoreLoad]

  [LoadStore]   // monitorenter
  y = 1;
  [StoreStore]  // monitorexit
  [StoreLoad]
}
```

Можно переставить `x = 1` и `y = 1`?

JSR 133 Cookbook: нет, нельзя.

Барьеры: lock coarsening, #2

```
void m() {
  synchronized(this) {
    x = 1;
  }
  synchronized(this) {
    y = 1;
  }
}
```

coarsening →

```
void m() {
  synchronized(this) {
    y = 1;
    x = 1;
  }
}
```

JSR 133 Cookbook: не-не-не, так нельзя!
Java Memory Model: ещё как можно!
HotSpot: окей, можно – делаем!

Барьеры: lock coarsening, #3

```
synchronized(this) {
  y = 1;
  x = 1;
}
```

Случай 1:

```
{
  int t1 = y; // 1
  int t2 = x; // 0
}
```

Читает через гонку, видит перестановку и без coarsening'а

Случай 2:

```
synchronized(this) {
  int t1 = x; // 1
  int t2 = y; // 1
}
```

Захватит лок и увидит обе записи в порядке

Барьеры: засада с памятью

Засада №2: Понятие «перестановки» употребимо, когда есть иллюзия общей синхронизированной памяти.

```
volatile int x, y;
```

x = 1;	y = 1;	int r1 = y;	int r3 = x;
		int r2 = x;	int r4 = y;

Если мы барьерами пригвоздим все операции в программном порядке, то всё будет нормально, да?

Барьеры: IRIW

`volatile int x, y;`

`x = 1;`	`y = 1;`	`int r1 = y;`	`int r3 = x;`
		`int r2 = x;`	`int r4 = y;`

$(r1, r2, r3, r4) = (1, 0, 1, 0)$ – запрещено JMM: результат может соответствовать только исполнению, где все synchronization actions образуют линейный порядок, согласованный с программным, и где чтения видят предыдущие записи в этом порядке.

Барьеры: IRIW

```
volatile int x, y;
```

`x = 1;`	`y = 1;`	`int r1 = y;` `int r2 = x;`	`int r3 = x;` `int r4 = y;`

$(r1, r2, r3, r4) = (1, 0, 1, 0)$ – запрещено JMM:

volatile – последовательно согласованы

Барьеры: IRIW с барьерами

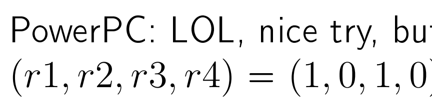
```
volatile int x, y;
```

`<fullFence>`	`<fullFence>`	`<loadFence>`	`<loadFence>`
`x = 1;`	`y = 1;`	`int r1 = y;`	`int r3 = x;`
`<fullFence>`	`<fullFence>`	`<loadFence>`	`<loadFence>`
		`int r2 = x;`	`int r4 = y;`
		`<loadFence>`	`<loadFence>`

Барьеры: IRIW с барьерами

`volatile int x, y;`

<fullFence>	<fullFence>	<loadFence>	<loadFence>
x = 1;	y = 1;	int r1 = y;	int r3 = x;
<fullFence>	<fullFence>	<loadFence>	<loadFence>
		int r2 = x;	int r4 = y;
		<loadFence>	<loadFence>

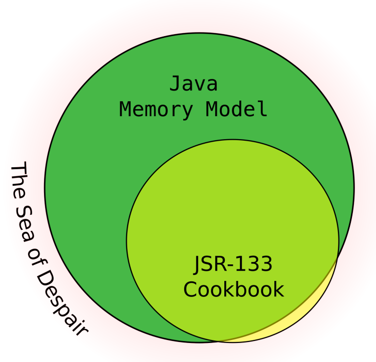


PowerPC: LOL, nice try, but

$$(r1, r2, r3, r4) = (1, 0, 1, 0)$$

(Screams internally)

Барьеры: рецепт провала



Дано:

1. Cookbook говорит «нельзя», а JMM говорит «можно»
2. Cookbook говорит «можно», а JMM говорит «нельзя»

Найти:

В каких случаях можно доверять Cookbook как родному JMM?

Барьеры: ересь, бред и провокация

```
void barrier() {
  synchronized(this) {}; // do barrier!
}
```

Барьеры: ересь, бред и провокация

```
void barrier() {
  synchronized(this) {}; // do barrier!
}
```

```
volatile int v;
void barrier() {
  v = 1; // do barrier!
}
```

Барьеры: ересь, бред и провокация

```
void barrier() {
  synchronized(this) {}; // do barrier!
}
```

```
volatile int v;
void barrier() {
  v = 1; // do barrier!
}
```

```
class MyClass {
  volatile int v;
  MyClass() {
    this.v = 42;
    // do barrier!
  }
}
```

Барьеры: сухой итог

Барьеры – это деталь реализации!
Реализации могут спокойно на них класть болт.

- Вы не умеете ими пользоваться – не пользуйтесь
- Вы не знаете закоулков хардвара – не пользуйтесь
- Хватит цитировать Cookbook и размахивать барьерами
- Cookbook вообще не для вас написан

Лирика

Лирика: «но как оно на самом деле?»

«На самом деле» существует только в воображении.
Всё всегда оперирует моделями:

1. javac делает вид, что сохраняет семантику исходного кода
2. JVM делает вид, что сохраняет семантику байткода
3. Компилятор делает вид, что сохраняет семантику IR
4. Фронтенд CPU делает вид, что сохраняет семантику ISA

...

51. Транзисторы делают вид, что переключаются
52. Электрический ток делает вид, что течёт

Лирика: у кого брал?

Модели описываются в спецификации:
JLS, JVMS, SDM, AM

Гипотеза: Большая часть бед в (concurrency-)индустрии от кинестетиков, которые пытаться вывести высокоуровневую модель из *текущего* поведения нижних слоёв, вместо того, чтобы изучить, какие *минимальными* свойствами должны обладать нижние слои.

Теория

Теория: действия и исполнения

Исполнения (executions) содержат в себе действия (actions) и порядки (orders). Исполнения – это поведения **абстрактной машины**, а не физические процессы. Исполнения показывают, какие результаты может давать конкретная программа.

Номенклатура:

- $w(field, V)$ – запись значения V в поле $field$
- $r(field) : V$ – чтение значения V из поля $field$
- $lck(m)$ / $unlck(m)$ – захват/освобождение монитора

Теория: PO – program order

PO связывает действия в одном потоке

Свойства и инварианты:

- PO – линейный порядок в каждом из потоков (но: ничего не говорит о *порядке исполнения*, говорит только о *порядке в исходном коде*)
- **PO consistency**: действия в одном потоке согласованы с порядком в программе
- PO нужен, чтобы протащить в модель инфу об оригинальной программе, поэтому *нарушить* PO нельзя, это исходное данное!

redhat.

Теория: SO – synchronization order

SO связывает специальные действия (synchronized actions)

Свойства и инварианты:

1. **SO** – линейный порядок (но: ничего не говорит о *порядке исполнения*, говорит только о *видимом порядке эффектов*)
2. **SO consistency**: чтения должны видеть последнюю запись, и только её
3. **SO-PO consistency**: порядки в SO и PO согласованы

Теория: SW – synchronizes-with order

SW – подпорядок SO

Свойства и инварианты:

1. **SW** – частичный порядок, установлен только между парами операций, которые «видят» друг друга
2. Важен для протаскивания между потоками

Теория: HB – happens-before order

HB – транзитивное замыкание объединения PO и SW

Свойства и инварианты:

1. **HB** – частичный порядок (не всё связано HB)
2. **HB consistency**: чтения видят

   либо последнюю запись в $\xrightarrow{hb}$,

   либо любую другую запись, не связанную $\xrightarrow{hb}$

Теория: итог

When someone explains something to you multiple times but you still have no idea wtf is going on

Теория: разберём случай с synchronized

```
int x, y;
```

```
void thread1() {
  synchronized(this) {
    x = 1;
  }
  synchronized(this) {
    y = 1;
  }
}
```

```
void thread2() {
  int r1 = y;
  int r2 = x;
}
```

Собираем...

$$w(x,1) \xrightarrow{\text{po}} w(y,1) \xrightarrow{\text{po}} unlck(t) \xrightarrow{\text{race}} r(y):1 \xrightarrow{\text{po}} r(x):0$$

redhat

Теория: разберём случай с synchronized

```
int x, y;

void thread1() {
  synchronized(this) {
    x = 1;
  }
  synchronized(this) {
    y = 1;
  }
}

void thread2() {
  int r1 = y;
  int r2 = x;
}
```

...**корректное** исполнение:

$$w(x,1) \xrightarrow{\text{hb}} w(y,1) \xrightarrow{\text{hb}} unlck(t) \xrightarrow{\text{race}} r(y):1 \xrightarrow{\text{hb}} r(x):0$$

redhat

Теория: корректно синхронизованная

```
int x, y;
```

```
void thread1() {
  synchronized(this) {
    x = 1;
  }
  synchronized(this) {
    y = 1;
  }
}
```

```
void thread2() {
  synchronized(this) {
    int r1 = y;
    int r2 = x;
  }
}
```

$$w(x,1) \xrightarrow{\text{po}} w(y,1) \xrightarrow{\text{po}} unlck(t) \xrightarrow{\text{sw}}$$

$$\xrightarrow{\text{sw}} lck(t) \xrightarrow{\text{po}} r(y):1 \xrightarrow{\text{po}} r(x):0$$

Теория: корректно синхронизованная

```
int x, y;
```

```
void thread1() {
  synchronized(this) {
    x = 1;
  }
  synchronized(this) {
    y = 1;
  }
}
```

```
void thread2() {
  synchronized(this) {
    int r1 = y;
    int r2 = x;
  }
}
```

$$w(x,1) \xrightarrow{hb} w(y,1) \xrightarrow{hb} unlck(t) \xrightarrow{hb}$$

$$\xrightarrow{hb} lck(t) \xrightarrow{hb} r(y):1 \xrightarrow{hb} \boxed{r(x):0}$$ **(некорректное)**

Lazy<T>

Lazy<T>: знакомьтесь

```
public class Lazy<T> {
  final Supplier<T> supp;
  T val;
  public Lazy(Supplier<T> s) {
    supp = s;
  }

  public synchronized T get() {
    if (val == null) {
      val = supp.get();
    }
    return val;
  }
}
```

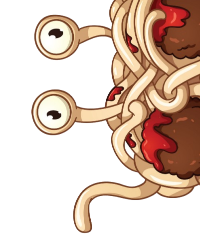
Ленивый такой инстанциатор. Доказывать, что он работает, надо?

Lazy<T>: оптимизируем!

```
final Supplier<T> supp;
volatile T val;

public T get() {
  if (val == null) {
    synchronized (this) {
      if (val == null) {
        val = supp.get();
      }
    }
  }
  return val;
}
```

Lazy<T>: оптимизируем!

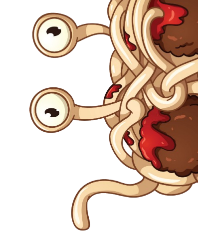


```
final Supplier<T> supp;
volatile T val;

public T get() {
  if (val == null) {
    synchronized (this) {
      if (val == null) {
        val = supp.get();
      }
    }
  }
  return val;
}
```

Святые макароны, но это же double-checked locking (DCL)!

Lazy<T>: оптимизируем!



```
final Supplier<T> supp;
volatile T val;

public T get() {
  if (val == null) {
    synchronized (this) {
      if (val == null) {
        val = supp.get();
      }
    }
  }
  return val;
}
```

Святые макароны, но это же double-checked locking (DCL)!

Он же не работает без volatile!

Lazy<T>: не так-то просто

```
Supplier<T> supp;
T val;

public T get() {
  if (supp != EMPTY) {
    synchronized (this) {
      if (val == null) {
        val = supp.get();
        supp = EMPTY;
      }
    }
  }
  return val;
}
```

Здесь тоже нужен volatile, но куда его поставить?

- A. на supp
- B. на val
- C. и на supp, и на val
- D. на звонок другу
- E. на зеро

Lazy<T>: ставим volatile на val

```
Supplier<T> supp;
volatile T val;

public T get() {
  if (supp != EMPTY) {           r(supp) : E
    synchronized (this) {
      if (val == null) {
        val = supp.get();        w(val, V)
        supp = EMPTY;            w(supp, E)
      }
    }
  }
  return val;                    r(val) : null
}
```

po

po

redhat.

Lazy<T>: ставим volatile на val

```
Supplier<T> supp;
volatile T val;

public T get() {
  if (supp != EMPTY) {
    synchronized (this) {
      if (val == null) {
        val = supp.get();
        supp = EMPTY;
      }
    }
  }
  return val;
}
```

$r(supp) : E$

$w(val, V)$

$w(supp, E)$

$r(val) : null$

hb

hb

redhat.

Lazy<T>: ставим volatile на val

```
Supplier<T> supp;
volatile T val;

public T get() {
  if (supp != EMPTY) {
    synchronized (this) {
      if (val == null) {
        val = supp.get();
        supp = EMPTY;
      }
    }
  }
  return val;
}
```

$r(supp) : E$

$w(val, V)$

$w(supp, E)$

$r(val) : null$

race

hb

hb

redhat

Lazy<T>: ставим volatile на val

```
Supplier<T> supp;
volatile T val;

public T get() {
  if (supp != EMPTY) {
    synchronized (this) {
      if (val == null) {
        val = supp.get();
        supp = EMPTY;
      }
    }
  }
  return val;
}
```

$r(supp) : E$

$w(val, V)$

$w(supp, E)$

$r(val) : null$

race

hb

hb

so

redhat

`Lazy<T>`: формальнее

Гипотеза: `Lazy.get()` никогда не вернёт `null`.

`Lazy<T>`: формальнее

Гипотеза: `Lazy.get()` никогда не вернёт `null`.

Гипотеза не верна, т.к. существует контрпример:

$$w(val, V) \xrightarrow{\text{hb}} w(supp, E) \xrightarrow{\text{race}} r(supp) : E \xrightarrow{\text{hb}} r(val) : null$$

`Lazy<T>`: формальнее

Гипотеза: `Lazy.get()` никогда не вернёт `null`.
Гипотеза не верна, т.к. существует контрпример:

$$w(val, V) \xrightarrow{\text{hb}} w(supp, E) \xrightarrow{\text{race}} r(supp) : E \xrightarrow{\text{hb}} r(val) : null$$

Интерпретации:

1. Ничто не ограничивает значение в $r(val)$, даже если val `volatile`: $r(val) : null \xrightarrow{\text{so}} w(val, V)$, упс
2. Прочитанное через гонку значение ничего с собой транзитивно не приносит, такая вот «удача»
3. **Гонки – вселенское зло!**

Lazy<T>: ставим volatile на supp

```
volatile Supplier<T> supp;
T val;

public T get() {
  if (supp != EMPTY) {
    synchronized (this) {
      if (val == null) {
        val = supp.get();
        supp = EMPTY;
      }
    }
  }
  return val;
}
```

$r(supp) : E$

$w(val, V)$

$w(supp, E)$

$r(val) : V$

po

po

redhat

Lazy<T>: **ставим** volatile **на** supp

```
volatile Supplier<T> supp;
T val;

public T get() {
  if (supp != EMPTY) {
    synchronized (this) {
      if (val == null) {
        val = supp.get();
        supp = EMPTY;
      }
    }
  }
  return val;
}
```

$r(supp) : E$

$w(val, V)$

$w(supp, E)$

$r(val) : V$

sw

po

po

redhat.

Lazy<T>: **ставим** volatile **на** supp

```
volatile Supplier<T> supp;
T val;

public T get() {
  if (supp != EMPTY) {
    synchronized (this) {
      if (val == null) {
        val = supp.get();
        supp = EMPTY;
      }
    }
  }
  return val;
}
```

$r(supp) : E$

$w(val, V)$

$w(supp, E)$

$r(val) : V$

hb

hb

hb

redhat.

`Lazy<T>`: ещё формальнее

Гипотеза: `Lazy.get()` никогда не вернёт `null`.

Положим, исполнение с $r(val) : null$ существует.

Lazy<T>: ещё формальнее

Гипотеза: `Lazy.get()` никогда не вернёт `null`.

Положим, исполнение с $r(val) : null$ существует. Этому чтению должны предшествовать:

1. $r(supp) :!E$ – взятая `synchronized`-ветка

`Lazy<T>`: ещё формальнее

Гипотеза: `Lazy.get()` никогда не вернёт `null`.

Положим, исполнение с $r(val) : null$ существует. Этому чтению должны предшествовать:

1. $r(supp) :!E$ – взятая `synchronized`-ветка

   1.1 $r(supp) :!E \xrightarrow{po} lck(t) \xrightarrow{po} r(val) : null \; \ldots \; r(val) : null$

Lazy<T>: ещё формальнее

Гипотеза: `Lazy.get()` никогда не вернёт `null`.

Положим, исполнение с $r(val) : null$ существует. Этому чтению должны предшествовать:

1. $r(supp) :!E$ – взятая `synchronized`-ветка

   1.1 $r(supp) :!E \xrightarrow{po} lck(t) \xrightarrow{po} r(val) : \boxed{null} \ \ldots \ r(val) : null$

   1.2 $r(supp) :!E \xrightarrow{po} lck(t) \xrightarrow{po} r(val) : \boxed{V} \ \ldots \ r(val) : null$

`Lazy<T>`: ещё формальнее

Гипотеза: `Lazy.get()` никогда не вернёт `null`.

Положим, исполнение с $r(val) : null$ существует. Этому чтению должны предшествовать:

1. $r(supp) :!E$ – взятая `synchronized`-ветка

   1.1 $r(supp) :!E \xrightarrow{po} lck(t) \xrightarrow{po} r(val) : \boxed{null} \ \ldots \ r(val) : null$

   1.2 $r(supp) :!E \xrightarrow{po} lck(t) \xrightarrow{po} r(val) : \boxed{V} \ \ldots \ r(val) : null$

2. $r(supp) : E$ – не взятая `synchronized`-ветка

   2.1 $r(supp) : E \xrightarrow{po} r(val) : null$

Lazy<T>: **1.1**: $r(supp) :!E \xrightarrow{po} lck(t) \xrightarrow{po} r(val) : \boxed{null}$

Это исполнение берёт ветку $if(val == null)$ внутри `synchronized`:

$$r(supp) :!E \xrightarrow{hb} lck(t) \xrightarrow{hb} r(val) : \boxed{null} \xrightarrow{hb}$$

$$\xrightarrow{hb} \boxed{w(val, V) \xrightarrow{hb} w(supp, E)} \xrightarrow{hb} unlck(t) \xrightarrow{hb} r(val) : null$$

`Lazy<T>`: **1.1:** $r(supp) :!E \xrightarrow{\text{po}} lck(t) \xrightarrow{\text{po}} r(val) : \boxed{null}$

Это исполнение берёт ветку $if(val == null)$ внутри `synchronized`:

$$r(supp) :!E \xrightarrow{\text{hb}} lck(t) \xrightarrow{\text{hb}} r(val) : \boxed{null} \xrightarrow{\text{hb}}$$

$$\xrightarrow{\text{hb}} \boxed{w(val, V) \xrightarrow{\text{hb}} w(supp, E)} \xrightarrow{\text{hb}} unlck(t) \xrightarrow{\text{hb}} r(val) : null$$

Такое исполнение нарушает HB consistency: $r(val)$ должен увидеть $w(val, V)$.

Lazy<T>: 1.2: $r(supp) :!E \xrightarrow{po} lck(t) \xrightarrow{po} r(val) : \boxed{V}$

Это исполнение не берёт ветку внутри synchronized:

$$r(supp) :!E \xrightarrow{po} lck(t) \xrightarrow{po} r(val) : \boxed{V} \xrightarrow{po} unlck(t) \xrightarrow{po}$$
$$\xrightarrow{po} r(val) : null$$

`Lazy<T>:` **1.2:** $r(supp) :!E \xrightarrow{po} lck(t) \xrightarrow{po} r(val) : \boxed{V}$

Это исполнение не берёт ветку внутри `synchronized`:

$$r(supp) :!E \xrightarrow{po} lck(t) \xrightarrow{po} r(val) : \boxed{V} \xrightarrow{po} unlck(t) \xrightarrow{po}$$
$$\xrightarrow{po} r(val) : null$$

Но $unlck_{n-1}(t) \xrightarrow{sw} lck_n(t)$,

Lazy<T>: 1.2: $r(supp) :!E \xrightarrow{po} lck(t) \xrightarrow{po} r(val) : \boxed{V}$

Это исполнение не берёт ветку внутри synchronized:

$$r(supp) :!E \xrightarrow{po} lck(t) \xrightarrow{po} r(val) : \boxed{V} \xrightarrow{po} unlck(t) \xrightarrow{po}$$
$$\xrightarrow{po} r(val) : null$$

Но $unlck_{n-1}(t) \xrightarrow{sw} lck_n(t)$, а значит:

$$w(val, V) \;\ldots\; \xrightarrow{hb} \boxed{unlck(t)} \xrightarrow{hb}$$
$$\xrightarrow{hb} \boxed{lck(t)} \xrightarrow{hb} r(val) : V \xrightarrow{hb} \;\ldots\; \xrightarrow{hb} r(val) : null$$

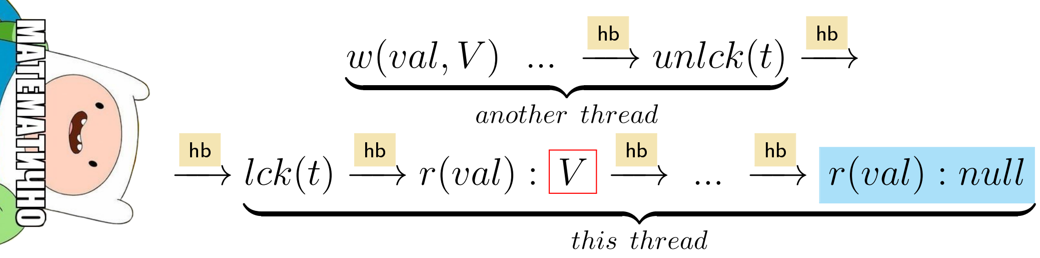
Lazy<T>: 1.2: $r(supp) :!E \xrightarrow{\text{po}} lck(t) \xrightarrow{\text{po}} r(val) : \boxed{V}$

$$\underbrace{w(val, V) \;\; \ldots \;\xrightarrow{\text{hb}} unlck(t)}_{another\ thread} \xrightarrow{\text{hb}}$$

$$\xrightarrow{\text{hb}} \underbrace{lck(t) \xrightarrow{\text{hb}} r(val) : \boxed{V} \xrightarrow{\text{hb}} \;\ldots\; \xrightarrow{\text{hb}} r(val) : null}_{this\ thread}$$

redhat.

Lazy<T>: 1.2: $r(supp) : !E \xrightarrow{\text{po}} lck(t) \xrightarrow{\text{po}} r(val) : \boxed{V}$



Такое исполнение нарушает HB consistency: если $r(val) : V$ видит $w(val, V)$ как последний, то и $r(val) : null$ должен!

`Lazy<T>`: **2.1:** $r(supp) : E \xrightarrow{\textsf{po}} r(val) : null$

Поскольку $w(supp, E) \xrightarrow{\textsf{sw}} r(supp) : E$, то:

$$\underbrace{w(val, V) \xrightarrow{\textsf{hb}} w(supp, E)}_{another\ thread} \xrightarrow{\textsf{hb}} \underbrace{r(supp) : E \xrightarrow{\textsf{hb}} r(val) : null}_{this\ thread}$$

Lazy<T>: 2.1: $r(supp) : E \xrightarrow{\text{po}} r(val) : null$

Поскольку $w(supp, E) \xrightarrow{\text{sw}} r(supp) : E$, то:

$$\underbrace{w(val, V) \xrightarrow{\text{hb}} w(supp, E)}_{another\ thread} \xrightarrow{\text{hb}} \underbrace{r(supp) : E \xrightarrow{\text{hb}} r(val) : null}_{this\ thread}$$

Такое исполнение нарушает HB consistency:
$r(val)$ обязан увидеть $w(val, V)$.

Lazy<T>: полезное свойство №1

Гипотеза: `Lazy.get()` никогда не вернёт `null`.

Попытка построить валидные исполнения с $r(val) : null$ перебрала все исполнения-кандидаты, и все кандидаты пофейлили HB consistency. Таким образом, валидных исполнений с $r(val) : null$ не существует. ЧТД.

Мы доказали интересное *высокоуровневое* свойство `Lazy<T>`: он всегда возвращает значение из `Supplier`-а.

Публикация

Публикация: полезное свойство №2

Уже доказано, что Lazy<T> возвращает только значение V.
Внимательно рассматривая те же исполнения, можно убедиться, что во всех чтениях $r(val) : V$ видит **только** запись через HB:

$$w(val, V) \xrightarrow{hb} \ldots \xrightarrow{hb} r(val) : V$$

1. $\xrightarrow{hb}$ через запись/чтение $supp$, приносящее $\xrightarrow{sw}$
2. $\xrightarrow{hb}$ через запись/чтение под локом, приносящее $\xrightarrow{sw}$
3. $\xrightarrow{hb}$ непосредственно к записи val в том же треде

Публикация: полезное свойство №2

Из транзитивности HB нам можно присобачить всё происходящее в `Supplier`:

$$\underbrace{\ldots}_{Supplier.get} \xrightarrow{hb} \underbrace{w(val, V) \xrightarrow{hb} \ldots}_{Lazy.get()_{init}} \xrightarrow{hb} \underbrace{\ldots \xrightarrow{hb} r(val) : V}_{Lazy.get()_{read}} \xrightarrow{hb} \underbrace{\ldots}_{return}$$

Публикация: полезное свойство №2

Из транзитивности HB нам можно присобачить всё происходящее в `Supplier`:

$$\underbrace{\ldots}_{Supplier.get} \xrightarrow{hb} \underbrace{w(val, V) \xrightarrow{hb} \ldots}_{Lazy.get()_{init}} \xrightarrow{hb} \underbrace{\ldots \xrightarrow{hb} r(val) : V}_{Lazy.get()_{read}} \xrightarrow{hb} \underbrace{\ldots}_{return}$$

Полезное свойство №2: «All actions in `Supplier.get` *happen-before* actions after returning the value from `Lazy.get`»

Публикация: полезное свойство №2

Теперь этим свойством можно свободно пользоваться:

```
final Lazy<Box> lazy = new Lazy(() -> new MyObject(42));

class MyObject {
  int x;
  public MyObject(int x) {
    this.x = x;
  }
}

public void m1() {
  Box b = lazy.get();
  assert (b.x == 42); // passes
}
```

Поскольку $supp.get() \xrightarrow{\text{hb}} lazy.get()$, то

$$w(b.x, 42) \xrightarrow{\text{hb}} r(b.x) : 42.$$

Публикация: безопасная публикация (безопасная)

```
volatile int v;

x = ...;                    if (v == 1) {   acquire
y = ...;                      int lx = x;
q = ...;               hb     int ly = y;
v = 1;   release              int lq = q;
                            }
```

- как будто «коммит в память», но только для пар операций
- release «коммитит», acquire забирает следы коммита
- acquire должен увидеть release!

Публикация: безопасная публикация (безопасная)

Гарантируются языком:

1. volatile write $\xrightarrow{hb}$ volatile read [если read видит write]
2. monitor exit $\xrightarrow{hb}$ monitor enter [если в таком порядке]
3. `Thread.start` $\xrightarrow{hb}$ (первое действие в потоке)
4. (последнее действие в потоке) $\xrightarrow{hb}$ `Thread.join`
5. (записи по умолчанию) $\xrightarrow{hb}$ (первое действие в потоке)

Публикация: безопасная публикация (безопасная)

Подобно тому, как `Lazy<T>` расширяет гарантии на свои свойства, другие библиотеки тоже не промах:

- `Executor.submit()` $\xrightarrow{hb}$ `Callable.call()`
- `ConcurrentHashMap.put()` $\xrightarrow{hb}$ `ConcurrentHashMap.get()`
- `Semaphore.release()` $\xrightarrow{hb}$ `Semaphore.acquire()`
- `Future.set()` $\xrightarrow{hb}$ `Future.get()`
- ...

Публикация: 25 кадр

Безопасная публикация: самый главный вывод из JMM

- От вас всего-то требуется в правильном порядке сделать `release` и `acquire`, когда публикуете между потоками!
- Не надо никаких размышлений о барьерах!
- Не надо никаких размышлений о когерентности кешей!
- Не надо никаких размышлений об оптимизациях!

Загоны

Двойнуха: оптимизируем...

```
volatile Supplier<T> supp;
T val;

public T get() {
  // avoid volatile read, for performance!!!
  if ((val == null) && (supp != EMPTY)) {
    ...
    val = supp.get();
    ...
  }
  return val;
}
```

Двойнуха: оптимизируем...

```
volatile Supplier<T> supp;
T val;

public T get() {
  // avoid volatile read, for performance!!!
  if (val == null) {
    ...
    val = supp.get();
    ...
  }
  return val;
}
```

$r(val) : V$

$w(val, V)$

$r(val) :?$

po

Двойнуха: оптимизируем...

```
volatile Supplier<T> supp;
T val;

public T get() {
  // avoid volatile read, for performance!!!
  if (val == null) {
    ...
    val = supp.get();
    ...
  }
  return val;
}
```

$r(val) : V$

$w(val, V)$

hb

$r(val) :?$

Двойнуха: оптимизируем...

```
volatile Supplier<T> supp;
T val;

public T get() {
  // avoid volatile read, for performance!!!
  if (val == null) {
    ...
    val = supp.get();
    ...
  }
  return val;
}
```

Двойнуха: формальнее

Оба случая валидны:

$$w(val, V) \xrightarrow{\text{race}} r(val) : V \xrightarrow{\text{hb}} r(val) : \boxed{null}$$

$$w(val, V) \xrightarrow{\text{race}} r(val) : V \xrightarrow{\text{hb}} r(val) : V$$

redhat.

Двойнуха: формальнее

Оба случая валидны:

$$w(val, V) \xrightarrow{\text{race}} r(val) : V \xrightarrow{\text{hb}} r(val) : \boxed{null}$$

$$w(val, V) \xrightarrow{\text{race}} r(val) : V \xrightarrow{\text{hb}} r(val) : V$$

Интерпретации:

1. HB consistency проверяется для каждого чтения *изолировано*

Двойнуха: формальнее

Оба случая валидны:

$$w(val, V) \xrightarrow{\text{race}} r(val) : V \xrightarrow{\text{hb}} r(val) : \boxed{null}$$

$$w(val, V) \xrightarrow{\text{race}} r(val) : V \xrightarrow{\text{hb}} r(val) : V$$

Интерпретации:

1. HB consistency проверяется для каждого чтения *изолировано*
2. HB ничего не говорит о порядке исполнения: операции, связанные HB, не обязаны выполняться в таком порядке

Двойнуха: формальнее

Оба случая валидны:

$$w(val, V) \xrightarrow{\text{race}} r(val) : V \xrightarrow{\text{hb}} r(val) : \boxed{null}$$

$$w(val, V) \xrightarrow{\text{race}} r(val) : V \xrightarrow{\text{hb}} r(val) : V$$

Интерпретации:

1. HB consistency проверяется для каждого чтения *изолировано*
2. HB ничего не говорит о порядке исполнения: операции, связанные HB, не обязаны выполняться в таком порядке
3. Голые чтения и записи **не когерентны**: нет общего порядка чтений и записей в *одну переменную*

Двойнуха: когерентность

Когерентность есть:

```
mov (%r1, C), %r2
mov (%r1, C), %r3
```

Двойнуха: когерентность

Когерентность есть:

```
mov (%r1, C), %r2
mov (%r1, C), %r3
```

Когерентности нет:

```
T r1 = field;
int r2 = r1.x;
int r3 = r1.x;
```

Двойнуха: важное отличие!

Эти случаи валидны:

$$w(val, V) \xrightarrow{\text{race}} r(val) : V \xrightarrow{\text{hb}} r(val) : \boxed{null}$$

$$w(val, V) \xrightarrow{\text{race}} r(val) : V \xrightarrow{\text{hb}} r(val) : V$$

$$w(val, V) \xrightarrow{\text{hb}} r(val) : V \xrightarrow{\text{hb}} r(val) : V$$

Нарушает HB consistency:

$$w(val, V) \xrightarrow{\text{hb}} r(val) : V \xrightarrow{\text{hb}} r(val) : null$$

Двойнуха: разрушаем гонку

```
volatile Supplier<T> supp;
volatile T val;

public T get() {
  // still a volatile read :(
  if (val == null) {
    ...
    val = supp.get();
    ...
  }
  return val;
}
```

$r(val) : V$

$w(val, V)$

po

$r(val) : V$

Двойнуха: разрушаем гонку

```
volatile Supplier<T> supp;
volatile T val;

public T get() {
  // still a volatile read :(
  if (val == null) {
    ...
    val = supp.get();
    ...
  }
  return val;
}
```

$r(val) : V$

$w(val, V)$

so

$r(val) : V$

Двойнуха: разрушаем гонку

```
volatile Supplier<T> supp;
volatile T val;

public T get() {
  // still a volatile read :(
  if (val == null) {
    ...
    val = supp.get();
    ...
  }
  return val;
}
```

$r(val) : V$

so

$w(val, V)$

so

$r(val) : V$

Двойнуха: формальнее

Нарушает SO consistency:

$$w(val, V) \xrightarrow{\text{so}} r(val) : V \xrightarrow{\text{po}} r(val) : null$$

$$w(val, V) \xrightarrow{\text{so}} r(val) : V \xrightarrow{\text{so}} r(val) : null$$

Интерпретации:

1. Линейный порядок и когерентность доступны только *synchronized actions*, для остальных – есть варианты
2. Только *synchronized actions* явно привязаны к программному порядку, для остальных – есть варианты

Двойнуха: 25 кадр

Гонки – полная жесть,
избегайте любой ценой

- Даже если в конкретном месте «всё получилось»
- Не уверен? Не убирай `synchronized` / `volatile`!

Однуха: одно чтение

```
volatile Supplier<T> supp;
T val;

public T get() {
  // avoid volatile read, for performance! READ ONCE!
  T v = val;
  if ((v == null) && (supp != EMPTY)) {
    ...
    v = supp.get();
    val = v;
    ...
  }
  return v;
}
```

Однуха: а что такого?

Валидное исполнение!

$$w(val.x, X) \xrightarrow{\text{hb}} w(val, V) \xrightarrow{\text{race}} r(val) : V \xrightarrow{\text{hb}} r(val.x) :!X$$

Интерпретации:

1. Прочитали в **одном** месте через гонку – прощай транизитивный HB
2. Прочитали внутри `Lazy` через гонку – прощай свойство

   $\texttt{Supplier.get} \xrightarrow{\text{hb}} \texttt{Lazy.get}$
3. Для безопасной публикации нужно **на всех путях** иметь соответствующие release/acquire

redhat.

Однуха: 25 кадр

Гонки – полная жесть,
избегайте любой ценой

- Транзитивные свойства *очень* легко разрушить
- Не уверен? Не убирай `synchronized` / `volatile`!
- Нельзя сначала убедиться, что всё работает, а потом вносить гонки

Инициализация

Инициализация: специальный случай

```java
final Supplier<T> supp;
T val;

public T get() {
  T v = val;
  if (v == null) {
    ...
    v = supp.get();
    val = v;
    ...
  }
  return v;
}
```

```java
class My {
  final String msg;
  My(String msg) {
    this.msg = msg;
  }
}

Lazy<My> L = Lazy.with(
  () -> new My("peekaboo"));

String m() {
  return L.get().msg;
}
```


redhat.

Инициализация: безопасная инициализация

Специальное правило для `final`:

$$w(val.x, X) \xrightarrow{\text{hb}} F \xrightarrow{\text{hb}} w(val, V) \xrightarrow{\text{mc}} r(val) : V \xrightarrow{\text{dr}} r(val.x) :?$$

означает, что:

$$w(val.x, X) \xrightarrow{\text{hb}} r(val.x) :?$$

Инициализация: безопасная инициализация

Специальное правило для `final`:

$$w(val.x, X) \xrightarrow{\text{hb}} F \xrightarrow{\text{hb}} w(val, V) \xrightarrow{\text{mc}} r(val) : V \xrightarrow{\text{dr}} r(val.x) :?$$

означает, что:

$$w(val.x, X) \xrightarrow{\text{hb}} r(val.x) :?$$

Интерпретации:

1. В сконструированном и прочитанном через гонку объекте видны `final`-поля, плюс видно всё за ними

Инициализация: обратите внимание

$$w(X.field, m) \xrightarrow{hb} w(R.x, X) \xrightarrow{race} r(R.x) : X \xrightarrow{hb}$$

$$w(val, X) \xrightarrow{hb} r(val) : X \xrightarrow{hb} r(X.field) :!m$$

Инициализация: обратите внимание

$$w(X.field, m) \xrightarrow{hb} w(R.x, X) \xrightarrow{race} r(R.x) : X \xrightarrow{hb}$$

$$w(val, X) \xrightarrow{hb} r(val) : X \xrightarrow{hb} r(X.field) :!m$$

Интерпретации:

1. `final` позволяет увернуться от гонки *только на новом объекте*, поля которого записаны публикующим потоком!
2. Невозможно «отмыться» от уже разворачивающейся гонки!

Инициализация: 25 кадр

Безопасная инициализация: второй главный вывод из JMM

- От вас всего-то требуется объявить поля `final` и не показывать недоконструированный объект
- Не надо никаких размышлений о барьерах!
- Не надо никаких размышлений о перестановках!
- Не надо!

Безгонки

Безгонки: «безопасная гонка»

```
public class String {
  int hash;
  public int hashCode() {
    if (hash == 0 && value.length > 0) {
      hash = ...
    }
    return hash;
  }
}
```

Проблемы?

Безгонки: «безопасная гонка»

```
public class String {
  int hash;
  public int hashCode() {
    if (hash == 0 && value.length > 0) {
      hash = ...
    }
    return hash;
  }
}
```

Проблемы?
Двойное чтение же, ну!

Безгонки: «безопасная гонка»

```
public class AbstractMap<K, V> {
  transient Set<K> keySet; // non-volatile

  public Set<K> keySet() {
    Set<K> ks = keySet; // RULE 1: SINGLE READ
    if (ks == null) {
      ks = new KeySet(); // RULE 2: SAFELY INITIALIZED
      keySet = ks;
    }
    return ks;
  }
}
```

Одно из правил не выполнено, и это – **полная гонка**.

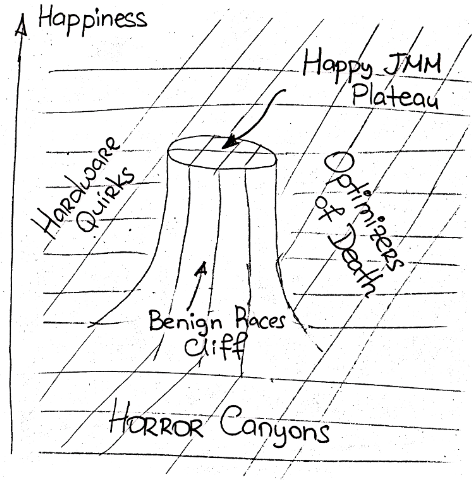
Безгонки: 25 кадр

Безопасные гонки: третий (опасный) вывод из JMM

- От вас всего-то требуется не делать лишних чтений, и оперировать **безопасно инициализированными** объектами!
- Не надо никаких размышлений о хардваре!
- Не надо никаких размышлений о возможных оптимизациях!
- Не надо размахивать руками!

Выводы

Выводы: в одной картинке



Выводы: в трёх параграфах

1. Запомните два правила: **безопасная публикация** и **безопасная инициализация**. Этих правил вам хватит в 99.99% случаев!
2. Ещё в 0.00999% случаев вам понадобятся **безопасные гонки**, и нужно будет уметь доказывать, что они безопасные
3. Все остальные фантазии и ценные мнения по поводу «что делают и не делают оптимизаторы», «что делает и не делает хардвар», ~~засуньте~~ оставьте при себе
4. Хотите большего? **Учите формализм!**

Выводы: в 140 символах

Aleksey Shipilëv
@shipilev

You don't have to be smart to write correct concurrent code; but you have to be super-smart if you try to outsmart the rules even a tiny bit

2:46 PM - 23 Sep 2016

Выводы: больше примеров

Больше примеров, тестов, дискуссий:
https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/

Backup

Backup: Засада: final против volatile

```
class Lazy<T> {
  volatile Supplier<T> supp;
  Lazy(Supplier<T> s) { this.supp = s; }
  Supplier<T> getSupp() { return supp; }
}
```

Backup: Засада: final против volatile

```java
class Lazy<T> {
  volatile Supplier<T> supp;
  Lazy(Supplier<T> s) { this.supp = s; }
  Supplier<T> getSupp() { return supp; }
}
```

Формально, гонка на инстансе Lazy разрешает $r(supp) : null$!

$$w(L.supp, S) \xrightarrow{\text{hb}} w(lazy, L) \xrightarrow{\text{race}}$$
$$\xrightarrow{\text{race}} r(lazy) : L \xrightarrow{\text{hb}} r(L.supp) : null$$

Backup: EMPTY можно, null нельзя!

```
volatile Supplier<T> supp;
T val;

public T get() {
  if (supp != EMPTY) {
    // (... get val ...)
  }
  return val;
}
```

Залётный $supp == null$ обрабатывается

```
volatile Supplier<T> supp;
T val;

public T get() {
  if (supp != null) {
    // (... get val ...)
  }
  return val;
}
```

Залётный $supp == null$ приводит к $val == null$