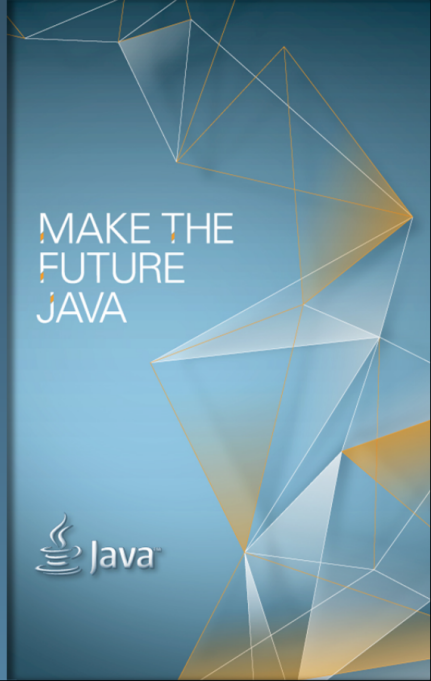
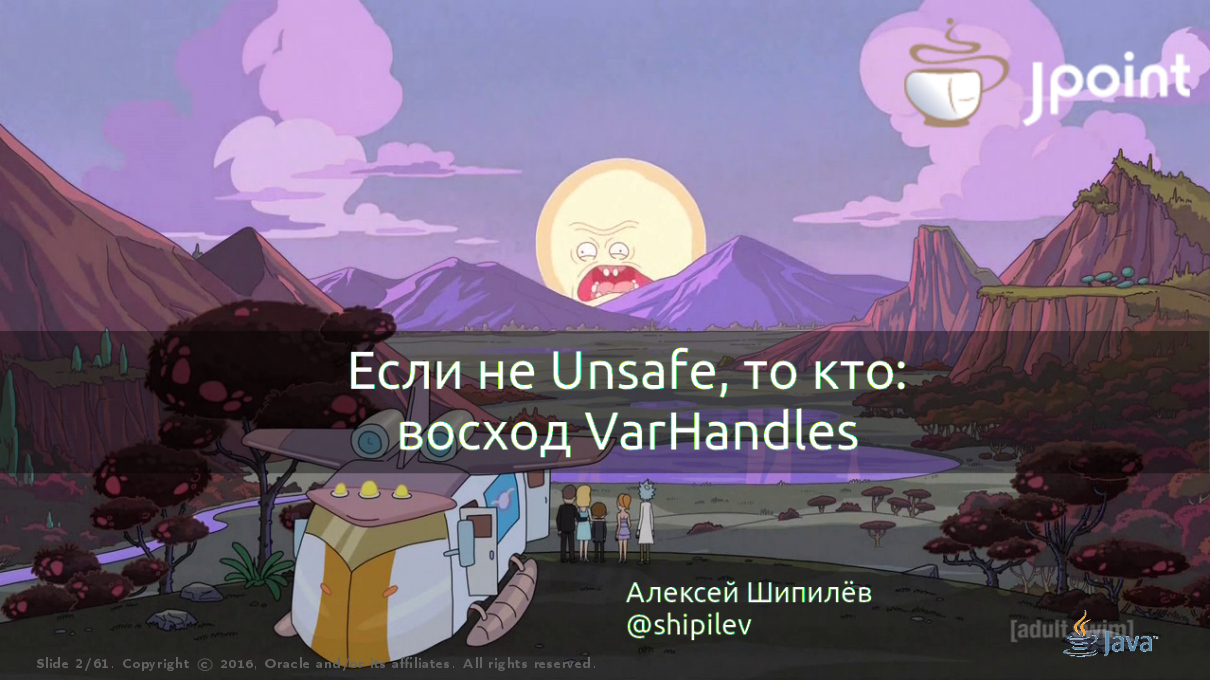
ORACLE

Если не Unsafe, то кто:
восход VarHandles

Алексей Шипилёв
aleksey.shipilev@oracle.com, @shipilev





The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Unsafe

Unsafe: продажная девка мазохизма

`sun.misc.Unsafe` – это секретная дверка в недра JVM

- `sun.misc.*` – не является частью Java стандарта
- Как и прочие `sun.misc.*`, нужна для поддержки стандартной библиотеки: общие части, привилегированные доступы, етц
- Доступ ограничен **стандартной библиотекой** – средствами, доступными в языке: приватными конструкторами, security-менеджерами, проверками стековых привилегий...

Unsafe: продажная девка мазохизма

`sun.misc.Unsafe` – это секретная дверка в недра JVM

- `sun.misc.*` – не является частью Java стандарта
- Как и прочие `sun.misc.*`, нужна для поддержки стандартной библиотеки: общие части, привилегированные доступы, етц
- Доступ ограничен **стандартной библиотекой** – средствами, доступными в языке: приватными конструкторами, security-менеджерами, проверками стековых привилегий... модульной инкапсуляцией (Jigsaw)

Unsafe: «Смотрите, какое богатство от нас прячут»[1]

But it's too hard.

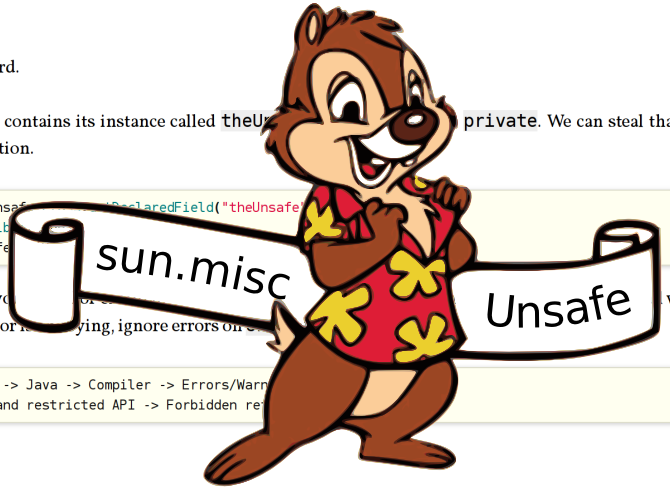
`Unsafe` class contains its instance called `theUnsafe`, which marked as `private`. We can steal that variable via java reflection.

```
Field f = Unsafe.class.getDeclaredField("theUnsafe");
f.setAccessible(true);
Unsafe unsafe = (Unsafe) f.get(null);
```

Note: Ignore your IDE. For example, eclipse show error "Access restriction. . ." but if you run code, all works just fine. If the error is annoying, ignore errors on `Unsafe` usage in:

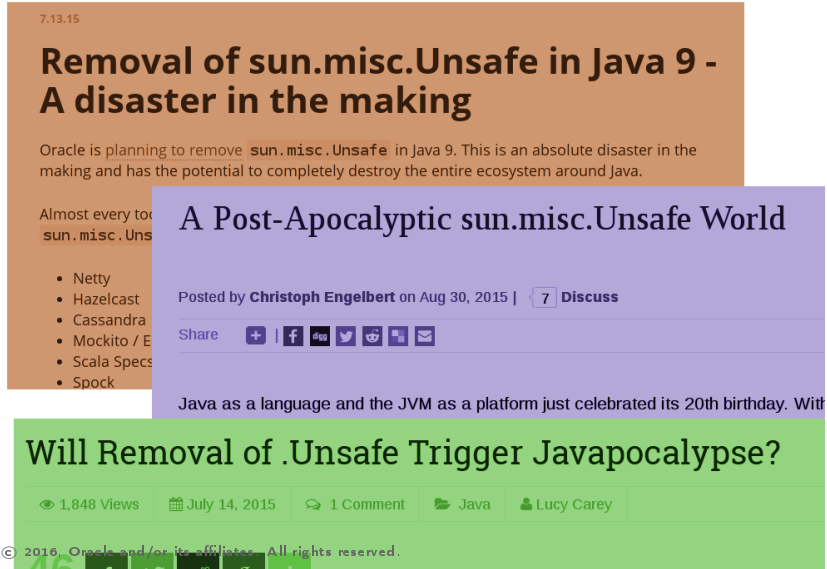
```
Preferences -> Java -> Compiler -> Errors/Warnings ->
Deprecated and restricted API -> Forbidden reference -> Warning
```

Unsafe: «Смотрите, какое богатство от нас прячут»[1]



[1] http://mishadoff.com/blog/java-magic-part-4-sun-dot-misc-dot-unsafe/

Unsafe: «A-a-a, они перепрятали нашу прелесть!»



Unsafe: C-like performance

Девелопер с `Unsafe` – это как самурай с мечом, но без меча

С неконтролируемым доступом к памяти вы получаете:

- отсутствие гарантий на type safety
- отсутствие гарантий на инкапсуляцию
- разрушенную модель безопасности вообще
- сложность оптимизаций (aliasing, например)

Что, каракатицы, позабыли C/C++, и SEGV-ы, memory corruption, и прочую платформенную вакханалию? Забыли, когда в последний раз gdb запускали?

UTF-8 Scan: посканируем ради UTF-8

Задача: проверить, что все байты меньше 0x7F.

```
byte[] bytes;

@Benchmark
public boolean array_byte() {
  for (byte b : bytes) {
    if (b < 0) return false;
  }
  return true;
}
```


UTF-8 Scan: ByteBuffer

Но есть же прекрасный ByteBuffer, он нам сейчас поможет!

```
@Benchmark
public boolean byteBuffer_byte() {
  ByteBuffer b = ByteBuffer.wrap(bytes)
                           .order(ByteOrder.nativeOrder());
  for (int i = 0; i < bytes.length; i++) {
    if (b.get(i) < 0) return false;
  }
  return true;
}
```


UTF-8 Scan: хм

JDK 8u66, i7-4790K, Linux x86_64:

Benchmark	Size	Score, ns/op	
array_byte	1	2.8	± 0.1
array_byte	1000	167.6	± 0.1
array_byte	1000000	157813.8	± 2232.9
byteBuffer_byte	1	3.9	± 0.1
byteBuffer_byte	1000	188.2	± 0.1
byteBuffer_byte	1000000	176938.7	± 4808.7

UTF-8 Scan: ByteBuffer + getLong

Чёрт с ним, запилим широкое сканирование руками:

```
@Benchmark
public boolean byteBuffer_long() {
  ByteBuffer b = ByteBuffer.wrap(bytes)
                           .order(ByteOrder.nativeOrder());
  int i;
  for (i = 0; i < bytes.length - 7; i += 8) {
    if ((b.getLong(i) & 0x8080808080808080L) != 0) return false;
  }
  for (; i < bytes.length; i++) {
    if (b.get(i) < 0) return false;
  }
  return true;
}
```


UTF-8 Scan: широкие лонги

JDK 8u66, i7-4790K, Linux x86_64:

Benchmark	Size	Score, ns/op	
array_byte	1	2.8	± 0.1
array_byte	1000	167.6	± 0.1
array_byte	1000000	157813.8	± 2232.9
byteBuffer_byte	1	3.9	± 0.1
byteBuffer_byte	1000	188.2	± 0.1
byteBuffer_byte	1000000	176938.7	± 4808.7

UTF-8 Scan: широкие лонги

JDK 8u66, i7-4790K, Linux x86_64:

Benchmark	Size	Score, ns/op	
array_byte	1	2.8	± 0.1
array_byte	1000	167.6	± 0.1
array_byte	1000000	157813.8	± 2232.9
byteBuffer_byte	1	3.9	± 0.1
byteBuffer_byte	1000	188.2	± 0.1
byteBuffer_byte	1000000	176938.7	± 4808.7
byteBuffer_long	1	4.1	± 0.1
byteBuffer_long	1000	287.6	± 0.1
byteBuffer_long	1000000	342378.6	± 1805.2

UTF-8 Scan: Unsafe спешит на помощь

```
@Benchmark
public boolean unsafe_long() {
  int i;
  for (i = (int)ABASE; i < bytes.length - 7; i += 8) {
    if ((U.getLong(bytes, i) & 0x8080808080808080L) != 0)
      return false;
  }
  for (; i < bytes.length; i++) {
    if (bytes[i] < 0) return false;
  }
  return true;
}
```


UTF-8 Scan: так-так-так...

JDK 8u66, i7-4790K, Linux x86_64:

Benchmark	Size	Score, ns/op	
array_byte	1	2.8	± 0.1
array_byte	1000	167.6	± 0.1
array_byte	1000000	157813.8	± 2232.1
byteBuffer_byte	1	3.9	± 0.1
byteBuffer_byte	1000	188.2	± 0.1
byteBuffer_byte	1000000	176938.7	± 4808.1
unsafe_long	1	2.2	± 0.1
unsafe_long	1000	80.2	± 0.1
unsafe_long	1000000	51343.5	± 175.8

VarHandles

VarHandles: пугало

```
package java.lang.invoke;

public abstract class VarHandle {
  public final native Object  get(Object... args);
  public final native void    set(Object... args);
  public final native Object  getVolatile(Object... args);
  public final native void    setVolatile(Object... args);
  public final native boolean compareAndSet(Object... args);
  // ...
}
```

Проблемы, Капитан Очевидность?

MethodHandles: JDK 7+

MethodHandle: «Handle» = «ручка», «ссылка».

```
private MethodHandle mh;
private static int doMe(int v) { return v; }

@Setup
public void setup() throws Exception {
  mh = ...lookup(..., "doMe", ...);
}

@Benchmark
int methodHandle() { return ... mh.invoke(...); }
```


MethodHandles: отличия от Reflection, #1

```
private static int doMe(int v) { return v; }

MethodHandles.Lookup lookup = MethodHandles.lookup();
MethodHandle mh = lookup.findStatic(MethodHandleSample.class,
        "doMe", MethodType.methodType(int.class, int.class));
```

Доступ проверяется на lookup'е, не на вызове!

MethodHandles: отличия от Reflection, #2

«Signature Polymorphism»

```
public class MethodHandle {
  public final Object invoke(Object... args) throws Throwable;
}
```

```
int methodHandle() { return (int) mh.invoke(v); }
```

```
public int methodHandle() throws java.lang.Throwable;
  0: getstatic     #4   // Field mh:Lj/l/i/MethodHandle;
  3: aload_0
  4: getfield      #2   // Field v:I
  7: invokevirtual #5   // Method j/l/i/MethodHandle.invoke:(I)I
 10: ireturn
```

MethodHandles: (не) отличия от Reflection, #3

На вызове нужно-таки проверить сигнатуры, и вызвать указываемый метод:

```
static       MethodHandle mh;
static final MethodHandle mh_constant;
static       Method meh;
static final Method meh_constant;

// invokevirtual #5   // Method j/l/i/MethodHandle.invoke:(I)I

int mh_invoke()          { return (int) mh.invoke(v); }
int mh_constant_invoke() { return (int) mh_constant.invoke(v); }
int reflect()            { return (int) meh.invoke(null, v); }
int reflect_constant()   { return (int) meh_constant.invoke(null, v);
```


MethodHandles: Performance landscape

Benchmark	Score, ns/op	
plain	1.922	± 0.002
mh_constant_invoke	1.923	± 0.002
mh_constant_invokeExact	1.924	± 0.004
mh_invoke	4.185	± 0.056
mh_invokeExact	4.152	± 0.006
reflect	7.695	± 0.104
reflect_constant	7.737	± 0.161

- вызов от константного MethodHandle дешев
- вызов неконстантного MethodHandle проверяет сигнатуры и ищет метод
- Method ещё и возится с varargs и прочей вакханалией

VarHandles: VarHandle.java

```
package java.lang.invoke;

public abstract class VarHandle {
    public final native
    @MethodHandle.PolymorphicSignature
    @HotSpotIntrinsicCandidate
    boolean compareAndSet(Object... args);

    ...
}
```

Отбрасывая мелкие детали, механика похожа на MethodHandle

VarHandles: фабрики

```
// Поля объектов
MethodHandles.Lookup.findVarHandle(Class<?>, String, Class<?>)
MethodHandles.Lookup.findStaticVarHandle(Class<?>, String, Class<?>)
MethodHandles.Lookup.unreflectVarHandle(Field)

// Массивы
MethodHandles.arrayElementVarHandle(Class<?>)

// View поверх byte[]
MethodHandles.byteArrayViewVarHandle(Class<?>, ByteOrder)

// View поверх ByteBuffer
MethodHandles.byteBufferViewVarHandle(Class<?>, ByteOrder)
```


VarHandles: пример

```
static final VarHandle VH_LONG =
   MethodHandles.byteArrayViewVarHandle(long[].class,
                                        ByteOrder.nativeOrder());

long useIt(byte[] arr, int idx) {
  return (long) VH_LONG.get(arr, idx);
}
```


VarHandles: куда же мы без ассемблера

```
long useIt(byte[] arr, int idx) {
  return (long) VH_LONG.get(arr, idx);
}
```

```
useIt(arr: %r10d, idx: %ebp):
  mov    0xc(%r12,%r10,8),%r8d   # arr.length
  add    $0xfffffff9,%r8d
  test   %r8d,%r8d               # range check 1
  jl     RANGE_CHECK_FAILED
  cmp    %r8d,%ebp               # range check 2
  jae    RANGE_CHECK_FAILED
  shl    $0x3,%r10               # idx /= 8
  mov    0x10(%r10,%rbp,1),%rax  # Unsafe.getLongUnaligned
  ret
```


VarHandles: куда же мы без ассемблера, #2

```
static final VarHandle VH =
  MethodHandles.lookup()
               .findVarHandle(Test.class, "v", int.class);

@Benchmark long test() {
  return (int) VH.get(this);
}
```

Компилируется в:

```
test(this: %rsi):
  movslq 0xc(%rsi),%rax          # get $v
  retq
```


Java

VarHandles: обратно в UTF-8 Scan

```
static final VarHandle VH = MethodHandles.byteArrayViewVarHandle(
  long[].class, ByteOrder.nativeOrder());

@Benchmark public boolean varHandles() {
  int li;
  for (li = 0; li < bytes.length / 8; li++) {
    if (((long) VH.get(bytes, li) & 0x8080808080808080L) != 0)
      return false;
  }
  for (int bi = li * 8; bi < bytes.length; bi++) {
    if (bytes[bi] < 0) return false;
  }
  return true;
}
```


VarHandles: опачки

JDK 9b114, i7-4790K, Linux x86_64:

Benchmark	Size	Score, ns/op	
array_byte	1	2.8	± 0.1
array_byte	1000	185.5	± 0.2
array_byte	1000000	172356.2	± 436.6
unsafe_long	1	2.2	± 0.1
unsafe_long	1000	46.0	± 0.1
unsafe_long	1000000	40782.5	± 1561.5

VarHandles: опачки

JDK 9b114, i7-4790K, Linux x86_64:

Benchmark	Size	Score, ns/op	
array_byte	1	2.8	± 0.1
array_byte	1000	185.5	± 0.2
array_byte	1000000	172356.2	± 436.6
unsafe_long	1	2.2	± 0.1
unsafe_long	1000	46.0	± 0.1
unsafe_long	1000000	40782.5	± 1561.5
varHandles	1	2.4	± 0.1
varHandles	1000	43.2	± 0.1
varHandles	1000000	35179.8	± 7.6

Лирическое отступление: другие API

ByteBuffers: Помните `Unsafe.getLongUnaligned`?

```
mov     0x10(%r10,%rbp,1),%rax   # Unsafe.getLongUnaligned
```

- На платформе можно делать невыровненные чтения? Тогда читаем!
- Нельзя? Ну тогда:

```
public final long getLongUnaligned(Object o, long offset) {
  if ((offset & 7) == 0) {
    return getLong(o, offset);
  } else if ((offset & 3) == 0) {
    return makeLong(getInt(o, offset),
                    getInt(o, offset + 4));
  } else ...
}
```


ByteBuffers: «та-да!»

JDK 9b114, i7-4790K, Linux x86_64:

Benchmark	Size	Score, ns/op	
byteBuffer_long	1	3.0	± 0.1
byteBuffer_long	1000	49.5	± 0.1
byteBuffer_long	1000000	40436.1	± 471.2
unsafe_long	1	2.2	± 0.1
unsafe_long	1000	46.0	± 0.1
unsafe_long	1000000	40782.5	± 1561.5
varHandles	1	2.4	± 0.1
varHandles	1000	43.2	± 0.1
varHandles	1000000	35179.8	± 7.6

A*FU: какие плохие апдейтеры

```
static final long OFF;
static final AtomicIntegerFieldUpdater<AFU> UPDATER;
static final VarHandle VH;

AtomicInteger ai = new AtomicInteger();
volatile int v;

@Benchmark int atomic()        { return ai.getAndAdd(1); }
@Benchmark int atomicUpdater() { return UPDATER.getAndAdd(this, 1); }
@Benchmark int unsafe()        { return U.getAndAddInt(this, OFF, 1);
@Benchmark int varHandle()     { return (int) VH.getAndAdd(this, 1);
```


A*FU: «та-да!»

Benchmark	Score, ns/op JDK 8u66		JDK 9b114		JDK 8u92	
atomic	7.005	± 0.003	7.014	± 0.008	7.015	± 0.006
unsafe	6.759	± 0.016	6.765	± 0.006	6.764	± 0.009
atomicUpdater						

[2] http://shipilev.net/blog/2015/faster-atomic-fu/

A*FU: «та-да!»

Benchmark	Score, ns/op JDK 8u66		JDK 9b114		JDK 8u92	
atomic	7.005	± 0.003	7.014	± 0.008	7.015	± 0.006
unsafe	6.759	± 0.016	6.765	± 0.006	6.764	± 0.009
atomicUpdater	8.757	± 0.003	6.777	± 0.021	6.764	± 0.002
varHandle			6.776	± 0.041		

С минимальным докручиванием[2] Atomic*FieldUpdater-ы получают то же сворачивание проверок, что и в VarHandles.

[2] http://shipilev.net/blog/2015/faster-atomic-fu/

VarHandles: промежуточный итог

- `VarHandle` – это *низкоуровневый* API, побратим `MethodHandle`
- `VarHandles` – это механизм, чтобы добиться нужной функциональности в высокоуровневом API (см. `java.util.concurrent.atomic.*`, например), вряд ли хорошая идея VarHandle-ы публиковать в пользовательском коде
- В большинстве случаев улучшения должны достигаться подкручиванием *публичного* API, а не латанием заплаток в обход его родимого
- Чем больше констант, тем больше можно свернуть, тем лучше производительность: относится и к `Unsafe`, и к `VarHandles`, и к `A*FU`.

Экзотические доступы

Экзотические доступы: C/C++11

Заявленная цель JEP 188: Java Memory Model Update[3] – подойти ближе к прорыву в C/C++11 (`std::atomic`, релаксации sequential consistency, и т.п.)

- Требует существенного перепиливания самой модели (в прогрессе)
- Разумным компромиссом является введение «экзотических» режимов доступа в API
- ...хотя они и не будут пока существовать в модели – но вон, `lazySet`-ом пользуются, и никто ещё не умер

[3] `http://openjdk.java.net/jeps/188`

weakCompareAndSet: рациональность

Гарантии прогресса:
compareAndSet хотя бы один из тредов выполнит успешно

```
void incr() {
  int v;
  do {
    v = ai.get();
  } while (!ai.compareAndSet(v, v + 1));
}
```


weakCompareAndSet: LL/SC

Если есть только LL/SC, можно сконструировать CAS вот так:

```
boolean CAS(expected, update):
  int cur = load-linked(&loc);
  if (cur == expected) {
     return store-conditional(&loc, update);
  } else {
     return false;
  }
```


weakCompareAndSet: LL/SC

Если есть только LL/SC, можно сконструировать CAS вот так:

```
boolean CAS(expected, update):
  int cur = load-linked(&loc);
  if (cur == expected) {
     return store-conditional(&loc, update);
  } else {
     return false;
  }
```

Но на *реальном хардваре* LL/SC может неожиданно пофейлиться («spurious failure»)...

weakCompareAndSet: LL/SC

Поэтому придётся оборачивать в цикл и бить до победного:

```
boolean CAS(expected, update):
  while (true) {
    int cur = load-linked(&loc);
    if (cur != expected)
      return false;
    if (store-conditional(&loc, update))
      return true;
  }
```


weakCompareAndSet: рациональность

Цикл циклом погоняет:

```
void incr() {
  int v;
  do {
    v = ai.get();
  } while (!compareAndSet(v, v + 1)); // ещё один цикл внутри
}
```


weakCompareAndSet: рациональность

```
void incr() {
  int v;
  do {
    v = ai.get();
  } while (!weakCompareAndSet(v, v + 1)); // нипалучилось
}

boolean weakCAS(expected, update):
  int cur = load-linked(&loc);
  if (cur == expected) {
     return store-conditional(&loc, update);
  } else {
     return false;
  }
}
```


weakCompareAndSet: результаты? Какие результаты?

Но ведь в Java 5 уже давно есть `weakCompareAndSet`?

```
public class AtomicInteger {
  public final boolean compareAndSet(int expect, int update) {
    return U.compareAndSwapInt(this, VALUE, expect, update);
  }
  public final boolean weakCompareAndSet(int expect, int update) {
    return U.compareAndSwapInt(this, VALUE, expect, update);
  }
}
```

Work in progress! Скоро всё будет: дырки уже просверлены, мейнтейнеры non-x86 бекендов завербованы, etc.

compareAndExchange: рациональность

В Java 8 есть только один способ сделать CAS:

```
// true -- получилось
// false -- не получилось
boolean compareAndSet(T expected, T update);
```


compareAndExchange: рациональность

В Java 8 есть только один способ сделать CAS:

```
// true -- получилось
// false -- не получилось
boolean compareAndSet(T expected, T update);
```

В некоторых случаях хочется видеть «failure witness»:

```
// expected -- получилось
// текущее -- не получилось
T compareAndExchange(T expected, T update);
```


compareAndExchange: боль[4]

```
public class NonBlockingHashMap {
  // CAS to claim the key-slot failed.

  // This re-read of the Key points out an annoying short-coming
  // of Java CAS.  Most hardware CAS's report back the existing
  // value - so that if you fail you have a *witness* - the value
  // which caused the CAS to fail.  The Java API turns this into
  // a boolean destroying the witness.  Re-reading does not recover
  // the witness because another thread can write over the memory
  // after the CAS.
}
```

[4] https://sourceforge.net/projects/high-scale-lib/

compareAndExchange: кто на свете всех быстрее?

```
@Benchmark
public int cas() {
  int ex, nx;
  do {
    ex = (int) VH.getVol();
    nx = mutate(ex);
  } while(!VH.cas(ex, nx));
  return nx;
}
```

```
@Benchmark
public int cae() {
  int x = (int) VH.getVol();
  int ex, nx;
  do {
    ex = x;
    nx = mutate(ex);
    x = (int) VH.cae(ex, nx);
  } while (x != ex);
  return nx;
}
```


compareAndExchange: кто на свете всех быстрее?

```java
@Benchmark
public int cas() {
  int ex, nx;
  do {
    ex = (int) VH.getVol();
    nx = mutate(ex);
  } while(!VH.cas(ex, nx));
  return nx;
}
```

866.1 ± 11.3 ns/op

```java
@Benchmark
public int cae() {
  int x = (int) VH.getVol();
  int ex, nx;
  do {
    ex = x;
    nx = mutate(ex);
    x = (int) VH.cae(ex, nx);
  } while (x != ex);
  return nx;
}
```

1006.3 ± 7.2 ns/op

compareAndExchange: кто на свете всех быстрее?[5]

```
@Benchmark
public int cae_reread() {
  int x;
  int ex, nx;
  do {
    ex = (int) VH.getVol();
    nx = mutate(ex);
    x = (int) VH.cae(ex, nx);
  } while (x != ex);
  return nx;
}
```

875.8 ± 7.7 ns/op

```
@Benchmark
public int cae() {
  int x = (int) VH.getVol();
  int ex, nx;
  do {
    ex = x;
    nx = mutate(ex);
    x = (int) VH.cae(ex, nx);
  } while (x != ex);
  return nx;
}
```

1006.3 ± 7.2 ns/op

[5] https://bugs.openjdk.java.net/browse/JDK-8141640

compareAndExchange: сюрприз!

`compareAndExchange` – это экзотический инструмент

- Помогает в случаях, когда действительно нужен witness
- А когда всё равно нужен `boolean` флаг и актуальное значение, `compareAndSet` спокойно работает и так
- Не бросайтесь переписывать код на новый блестящий-свистящий API, если на то нет существенных причин[6]

[6] см. также: «стримоз» головного мозга

acquire/release: рациональность

plain

- обычные доступы, никаких гарантий на ordering, кроме зацепленных в happens-before
- легко оптимизируются
- самое легкое на HW

volatile

- синхронизованные доступы, полные гарантии на ordering, включая sequential consistency
- практически не оптимизируются
- самое тяжёлое на HW

acquire/release: рациональность

plain	acquire/release	volatile
▪ обычные доступы, никаких гарантий на ordering, кроме зацепленных в happens-before ▪ легко оптимизируются ▪ самое легкое на HW	▪ синхронизованные доступы, частичные гарантии на ordering, не включая sequential cosnsitency ▪ практически не оптимизируются ▪ средне тяжёлые на HW	▪ синхронизованные доступы, полные гарантии на ordering, включая sequential consistency ▪ практически не оптимизируются ▪ самое тяжёлое на HW

acquire/release: они уже протекли в джаву

```
class AtomicInteger {
    /**
     * Eventually sets to the given value.
     */
    public final void lazySet(int newValue) {
        unsafe.putOrderedInt(this, valueOffset, newValue);
    }
}
```

«О боже мой, ну это же release!»

Тьма тьмущая гайдов и свидетельств, что помогает в некоторых случаях, когда не нужна sequential consistency, а просто тупой producer-consumer.[7]

[7] http://psy-lob-saw.blogspot.ru/2012/12/atomiclazyset-is-performance-win-for.html

acquire/release: VarHandles

```
class VarHandle {
  Object getAcquire(Object... args);
  void setRelease(Object... args);
}
```

Для всех типов данных и на всех платформах!

```
class AtomicInteger {
    public final void lazySet(int newValue) {
        VH.setRelease(this, newValue);
    }
}
```


acquire/release: IRIW c volatile

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

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

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

acquire/release: IRIW с volatile

```
                    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: ломает sequential consistency

acquire/release: IRIW c acq/rel?

int x, y;

`rel(&x, 1);`	`rel(&y, 1);`	`int r1 = acq(&y);` `int r2 = acq(&x);`	`int r3 = acq(&x);` `int r4 = acq(&y);`

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

acquire/release: IRIW с acq/rel?

int x, y;

`rel(&x, 1);`	`rel(&y, 1);`	`int r1 = acq(&y);` `int r2 = acq(&x);`	`int r3 = acq(&x);` `int r4 = acq(&y);`

$(r1, r2, r3, r4) = (1, 0, 1, 0)$ **разрешено!**

Opaque: рациональность

plain

- обычные доступы, никаких гарантий на ordering, кроме зацепленных в happens-before
- легко оптимизируются
- самое легкое на HW

acquire/release

- синхронизованные доступы, частичные гарантии на ordering, не включая sequential cosnsitency
- практически не оптимизируются
- средне тяжёлые на HW

Opaque: рациональность

plain

- обычные доступы, никаких гарантий на ordering, кроме зацепленных в happens-before
- легко оптимизируются
- самое легкое на HW

opaque

- обычные доступы, никаких гарантий на ordering, кроме зацепленных в happens-before
- практически не оптимизируются
- сравнительно тяжёлые на HW

acquire/release

- синхронизованные доступы, частичные гарантии на ordering, не включая sequential cosnsitency
- практически не оптимизируются
- средне тяжёлые на HW

Opaque: потестим

```
@Group @Benchmark void read(Control control) {
  int lastVal = (int) VH.getXXX(this);
  do {
    if (lastVal != (int) VH.getXXX(this)) return;
  } while (!control.stopMeasurement);
}

@Group @Benchmark long write() {
  long acc = 0;
  for (int c = 0; c < count; c++) {
    acc += c;
    VH.setXXX(this, c);
  }
  return acc;
}
```


Opaque: оппа

Benchmark	Score, ns/op	
	read	write
plain		

Opaque: оппа

Benchmark	Score, ns/op read		write	
plain	2520077320	± 840865394	2659	± 1
volatile				

Opaque: оппа

Benchmark	Score, ns/op			
	read		write	
plain	2520077320	± 840865394	2659	± 1
volatile	33	± 2	74600	± 3265
opaque				

Opaque: оппа

Benchmark	Score, ns/op read		write	
plain	2520077320	± 840865394	2659	± 1
volatile	33	± 2	74600	± 3265
opaque	33	± 2	3496	± 131

- opaque опубликовалось через гонку, не заплатив за volatile
- opaque прочиталось через гонку, не оптимизировавшись из цикла
- (очень похоже на std::atomic.load(..., memory_order_relaxed))

Будущее

Будущее: судьба Unsafe

Выпилить `sun.misc.Unsafe` из JDK 9 прямо сейчас!

Аргументы «за»:

- Нужно просверлить кучу дырок под новые фичи, но новым методам утекать нельзя
- Давно в этом гадюшнике надо прибраться (single address vs. double address, проверки на java-овой стороне, например)

Аргументы «против»:

- Слишком много зависимостей, и слишком мало времени для миграции
- Даже перенести в другой package нельзя, т.к. именно `sun.misc.Unsafe` ожидается при компиляции

Что делать?

Будущее: зачем иметь один `Unsafe`, когда можно два?

`sun.misc.Unsafe`

- откачена на состояние JDK 8 GA
- доступен в JDK 9, см. JEP 260[8]
- мы его *точно* выпилим позднее

`jdk.internal.misc.Unsafe`

- новая песочница: отутюжена, почищена, новыми методами завалена
- модульной инкапсуляцией защищена
- для тестирования доступна специальным ключиком: `-XaddExports:java.base/jdk.internal.misc=ALL-UNNAMED`

[8] `http://openjdk.java.net/jeps/260`

Будущее: famous last words

Public API > Public API Hacks >> Non-public API > Unsafe

- Использовать приватные API – к техническому долгу, долгим слезам и заламываниям рук
- (F)OSS – не бесплатный сыр; хотите уверенности в завтрашнем дне – инвестируйте в сегодняшнее дно
- **Ваша** задача в JDK 9 – смигрировать с `Unsafe` на публичные API. Требуйте у вендоров библиотек совместимости, выбирайте с умом – шлите лесом рокстаров и ниндзь, с костылями которых вам потом жить

DIXI. Q/A?