链滴

JEP 193：Variable Handles

作者：xjlnjut730

原文链接：https://ld246.com/article/1540360885238

来源网站：链滴

学习 JDK 源码过程中，发现了 java.lang.invoke.VarHandle 这个类，查了一下，这个类来源于a href="https://ld246.com/forward?goto=http%3A%2F%2Fopenjdk.java.net%2Fjeps%2F193" arget="_blank" rel="nofollow ugc">JEPS193，因此查了一下规范。以下便是规范的具体内以及对应的翻译，其中上半段翻译是我自己翻译的，后半段翻译时，发现网上已有，翻译比我靠多了，但他只有中文版，以下是我整理的中英文对照着看，方便理解。
<h2 id="Summary-摘要-">Summary（摘要）</h2>
Define a standard means to invoke the equivalents of variousjava.util.concurrent.atomic nd sun.misc.Unsafe operations upon object fields and array elements, a standard set of fence perations for fine-grained control of memory ordering, and a standard reachability-fence ope ation to ensure that a referenced object remains strongly reachable.
定义了一些标准方法来执行 java.util.concurrent.atomic 和 sun.misc.Unsafe 包下对对象属性数组元素的操作，提供了对内存序列细粒度访问对象属性的操作，和一个标准可达性栅栏操作来确保引用的对象是强可达的（strongly reachable）。
<h2 id="Goals-目标-">Goals（目标）</h2>
The following are required goals:

Safety. It must not be possible to place the Java Virtual Machine in a corrupt memory stat . For example, a field of an object can only be updated with instances that are castable to the f eld type, or an array element can only be accessed within an array if the array index is within t e array bounds.
Integrity. Access to a field of an object follows the same access rules as with getfield and utfield byte codes in addition to the constraint that a final field of an object cannot be updat d. (Note: such safety and integrity rules also apply to MethodHandles giving read or write acc ss to a field.)
Performance. The performance characteristics must be the same as or similar to equivalen sun.misc.Unsafe operations (specifically, generated assembler code should be almost identica modulo certain safety checks that cannot be folded away).
Usability. The API must be better than the sun.misc.Unsafe API.

It is desirable, but not required, that the API be as good as thejava.util.concurrent.atomic API.
接下来是所需要达到的目标：

安全性。它必须不能使 JVM 进入不正确的内存状态。比如，对象的一个字段只能被修改成可以化为对应类型的实例，或者数据的索引必须在数组长度范围内才能被访问。
一致性。访问对象的字段遵从 getfield 与 pufield 的访问权限控制，另外，被 final 修饰的字段能被修改。（注：这些安全性和一致性规则同样应用于 MethodHandles 读写字段。）
性能。它提供的性能必须与 sun.misc.Unsafe 操作差不多。（特别地，除了某些无法折叠的安全查，生成的汇编代码应该几乎完全相同。）
可用性。API 实现必须比 sun.misc.Unsafe 更好用。

我们希望该 API 可以与 java.util.concurrent.atomic 一样好，但不是必须的。
<h2 id="Motivation-动机-">Motivation（动机）</h2>
As concurrent and parallel programming in Java continue to expand, programmers are in reasingly frustrated by not being able to use Java constructs to arrange atomic or ordered op rations on the fields of individual classes; for example, atomically incrementing a count field. ntil now the only ways to achieve these effects were to use a stand-alone AtomicInteger (add ng both space overhead and additional concurrency issues to manage indirection) or, in some situations, to use atomic FieldUpdaters (often encountering more overhead than the operatio itself), or to use the unsafe (and unportable and unsupported) sun.misc.UnsafeAPI for JVM in rinsics. Intrinsics are faster, so they have become widely used, to the detriment of safety and ortability.

Without this JEP, these problems are expected to become worse as atomic APIs expand t cover additional access-consistency policies (aligned with the recent C++11 memory model) s part of Java Memory Model revisions.
Java 中并发与并行编程持续增多，开发人员越来越感到沮丧，因为不能对类的成员执行原子操或对操作进行排序。比如，原子化增加 count 字段，迄今为止，能够实施该目标的仅有使用一个单独 AtomicInteger(增加了空间开销，以及并发问题）或，在某些场景下，使用原子 FieldUpdaters（常比原操作引入了更多的开销），或使用 unsafe（不兼容和不被支持的） sun.misc.UnsafeAPI 来使 JVM 内联。内联函数是快的，因此内联函数使用得越来越广泛，这损害了安全性与兼容性。
没有这份 JEP，这些问题会变得越来越糟糕，未来 JMM 修正模型下，原子操作 API 会扩展以涵其他访问一致性策略（与最近的 c++ 11 内存模型一致)。
<h2 id="Description-描述-">Description（描述）</h2>
A variable handle is a typed reference to a variable, which supports read and write access o the variable under a variety of access modes. Supported variable kinds include instance fiel s, static fields and array elements. Other variable kinds are being considered and may be sup orted such as array views, viewing a byte or char array as a long array, and locations in off-he p regions described by ByteBuffers.
一个变量句柄是一个对象的类型引用，用来支持使用大量的访问模式对变量的读写操作。被支持变量类型包括：对象字段，静态字段和数组元素。其它变量类型也会被考虑，也可能会被支持，比如组视图，将一个 byte 或 char 数组视作 long 数组，以及在 off-heap 被描述成 ByteBuffer 的区域。/p>
Variable handles require library enhancements, JVM enhancements, and compiler support Additionally, it requires minor updates to the Java Language Specification and the Java Virtua Machine Specification. Minor language enhancements, that enhance compile-time type chec ing and complement existing syntax, are also considered.
The resulting specifications are expected to be extensible in natural ways to additional pr mitive-like value types or additional array-like types, if they are ever added to Java. This is not, however, a general-purpose transaction mechanism for controlling accesses and updates to ultiple variables. Alternative forms for expressing and implementing such constructs may be e plored in the course of this JEP, and may be the subject of further JEPs.
变量句柄需要类库增强、JVM 增强和编译器支持。另外，它需要对 Java 语言规范与 JVM 规范行微小更新。语言方面的少许增强，编译期类型检查和补足已有的语法，也在考虑范围内。
一旦它们被加入 Java，最终的规范预期以自然的方式进行扩展，增加类似原始值类型或数组类。
然而，这并不是用于控制对多个变量的访问和更新的通用事务机制。在本 JEP 课程中，可能会探表达和实现这种结构的其他形式，并可能成为未来 JEPs 的主题。
Variable handles are modelled by a single abstract class,java.lang.invoke.VarHandle, wher each variable access mode is represented by a <a href="https://ld246.com/forward?goto=ht p%3A%2F%2Fdocs.oracle.com%2Fjavase%2F8%2Fdocs%2Fapi%2Fjava%2Flang%2Finvoke%2 MethodHandle.html%23sigpoly" target="_blank" rel="nofollow ugc">signature-polymorphic /a> method.
The set of access modes represents a minimal viable set and are designed to be compatib e with C/C++11 atomics without depending on a revised update to the Java Memory Model. dditional access modes will be added if required. Some access modes may not be applicable or certain variable types and, if so, when invoked on an associated VarHandle instance will th ow anUnsupportedOperationException.
变量句柄被实现成一个单独的抽象类，java.lang.invoke.VarHandle，每个对象访问模式被表现签名多态的方法。访问模式集合代表一个最小可行集，设计为兼容 C/C++11 的原子性，而不是依赖个 Java 内存模型的修改更新。额外的访问模式如果有需要，未来可以被继续添加。一些访问模式可无法应用于某些具体变量类型，如果是这样，当执行一个关联的 VarHandle 变量时，会抛出一个 Uns pportedOperationException.
The access modes are grouped into the following categories:
1. read access modes, such as reading a variable with volatile memory ordering effects;< p>
2. write access modes, such as updating a variable with release memory ordering effects;

/p>
3. atomic update access modes, such as a compare-and-set on a variable with volatile m
mory order effects for both read and writing;
4. numeric atomic update access modes, such as get-and-add with plain memory order ef
ects for writing and acquire memory order effects for reading.
5. bitwise atomic update access modes, such as get-and-bitwise-and with release memor
order effects for writing and plain memory order effects for reading.
访问模式被组织成以下分类：
1. 读模式，比如读一个具有 volatile 内存序列效果的变量。
2. 写模式，比如更新一个具有 release 内存序列效果的变量。
3. 原子更新模式，比如对一个对读与写均具有 volatile 内存序列效果的变量进行 CAS 操作。</
>
4. 数值原子更新模式，比如 get-and-add 操作，对写操作具有 plain 内存序列效果，对读具有 a
quire 内存序列效果。
5. 按位原子更新模式，比如 get-and-bitwise-and 操作，对写操作具有 release 内存序列效果
对读具有 plain 内存序列效果。
The later three categories are commonly referred to as read-modify-write modes.
最后三个分类可以归类为读修改写的模式。
The signature-polymorphic characteristic of the access mode methods enables variable h
ndles to support many variable kinds and variable types using just one abstract class. This avo
ds an explosion of variable kind and type-specific classes. Furthermore, even though the acce
s mode method signatures are declared as a variable argument array of Object, such signatur
-polymorphic characteristics ensure there will be no boxing of primitive value arguments and
no packing of arguments into an array. This enables predictable behaviour and performance a
runtime for the HotSpot interpreter and C1/C2 compilers.
访问模式的方法具有签名多态特性，使得变量句柄使用一个抽象类就能够支持许多变量类型。这
免了变量种类和具体类型 class 的膨胀。更进一步，尽管访问模式方法签名被申明为一个 Object 变
参数数组，这种签名多态特性确保数组中的变量没有原始值类型参数的包装操作，也没有参数的拆包
作。对于 HotSpot 解释器和 C1/C2 编译器，这确保了可预期行为和运行时性能。
Methods to create VarHandle instances are located in the same area as that to produce
ethodHandle instances which access equivalent or similar variable kinds.
创建 VarHandle 实例的方法存储在同一个区域，用于生产访问等价或类似的变量类型的 Metho
Handle 实例。
Methods to create VarHandle instances for instance and static field variable kinds are loc
ted in java.lang.invoke.MethodHandles.Lookup and are created by a process of looking up the
field within the associated receiving class. For example, such lookup to obtain a VarHandle for
a field named i of type int on a receiver class Foo might be performed as follows:
创建 VarHandle 实例对于实例与静态字段变量类型存储在 java.lang.invoke.MethodHandles.L
okup，是由查找关联接收类中的字段的过程创建的。举个例子，查询类 Foo 下字段名为 i，类型为 int
的 VarHandle 对象，可以被描述成如下形式：
<pre><code class="highlight-chroma"><span class="highlight
cl">class Foo {
    int i;
    ...
}

...
class Bar {
    static final VarH
ndle VH_FOO_FIELD_I;

    static {
        try {
            VH_FOO_F

```
ELD_I = MethodHandles.lookup().
</span></span><span class="highlight-line"><span class="highlight-cl">            in(Foo.c
ass).
</span></span><span class="highlight-line"><span class="highlight-cl">            findVar
andle(Foo.class, "i", int.class);
</span></span><span class="highlight-line"><span class="highlight-cl">        } catch (Exce
tion e) {
</span></span><span class="highlight-line"><span class="highlight-cl">            throw new
Error(e);
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>The lookup of a VarHandle that accesses a field will, before producing and returning the
arHandle, perform the exact same access control checks (on behalf of the lookup class) as tho
e performed by the lookup up of a MethodHandle that gives read and write access to that sa
e field (see the find{,Static}{Getter,Setter} methods in the MethodHandles.Lookup class).</p>
<p>这个查找过程在生成并返回 VarHandle 前，会检查一系列的访问控制权限。对 MethodHandle 来说也一样，会看提供了读、写的 MethodHandle （参考 MethodHandles.Lookup 的 find{,Static} Getter,Setter} 方法）的针对特定字段是否有对应的权限。</p>
<p>Access mode methods will throw UnsupportedOperationException when invoked under t
e following conditions:</p>
<ul>
<li>Write access mode methods for a VarHandle to a final field.</li>
<li>Numeric-based access mode methods (getAndAdd and addAndGet) for a reference varia
le type or a non-numeric type (such as boolean).</li>
<li>Bitwise-based access mode methods for a reference variable type or the float and double
types (the latter restriction may be removed in a future revision)</li>
</ul>
<p>在下面这些条件下，访问模式方法会抛出 UnsupportedOperationException 异常：</p>
<ul>
<li>对一个 final 变量调用写访问模式方法</li>
<li>对引用类型或非数值类型（如 boolean）调用数值访问模式方法（getAndAdd，addAndGet）
/li>
<li>对引用类型或 float/double 执行按位访问模式方法（后者以后可能会移除）</li>
</ul>
<p>A field need not be marked as volatile for an associated VarHandle to perform volatile ac
ess. In effect, the volatile modifier, if present, is ignored. This is different to the behaviour of j
va.util.concurrent.atomic.Atomic{Int, Long, Reference}FieldUpdater where corresponding field
 have to be marked as volatile. This can be too restrictive in certain cases where it is known ce
tain volatile accesses are not always required.</p>
<p>一个字段不需要声明为 volatile 也可以使用 VarHandle 来进行 volatile access。实际上，如果带了 volatile 修饰符，它会被忽略掉。这个行为跟 java.util.concurrent.atomic.Atomic{Int, Long, Re
erence}FieldUpdater 是不一样的，使用后者时对应的字段需要声明为 volatile。当我们在某些时候要 volatile 语义而其他时候不需要时，FilldUpdater 就显得过于受限了。</p>
<p>Methods to create VarHandle instances for array-based variable types are located in java.
ang.invoke.MethodHandles (see the arrayElement{Getter, Setter}methods in the MethodHand
es class). For example, a VarHandle to an array of intmay be created as follows:</p>
<p>生成用于数组的 VarHandle 位于 java.lang.invoke.MethodHandles （参考 MethodHandles arrayElement{Getter, Setter} 方法）。例如，用于 int 数组的 VarHandle 可以这样生成：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">VarHandle intArrayHandle = MethodHandles.arrayElementVarHandle(int[].class);
</span></span></code></pre>
<p>Access mode methods will throw UnsupportedOperationException when invoked under t
```

e following conditions:

Numeric-based access mode methods (getAndAdd and addAndGet) for an array compon nt reference variable type or a non-numeric type (such as boolean)
Bitwise-based access mode methods for a reference variable type or the float and double types (the latter restriction may be removed in a future revision)

在下列情况下，访问模式方法会抛出 UnsupportedOperationException 异常：

使用数值方法模式方法去修改引用类型或非数值类型（如 boolean）数组的元素
对引用类型或 float/double 执行按位访问模式方法（后者以后可能会移除）

All primitive types and references types are supported for the variable type of variable ki ds that are instance fields, static fields and array elements. Other variable kinds may support a l or a subset of those types.
所有的变量类型的基本类型（primitive types）和引用类型都是被支持的，只要它们的变量种类 variable kinds）是成员变量、静态变量或数组。其他变量种类可能会部分或全部支持。
Methods to create VarHandle instances for array-view-based variable types are also locat d in java.lang.invoke.MethodHandles. For example, a VarHandle to view an array of byte as an unaligned array of long may be created as follows:
生成用于 array-view-based 的 VarHandle 的方法位于 java.lang.invoke.MethodHandles。 个例子，下面生成的 VarHandle 把一个 byte 数组看成一个非对其（unaligned）的 long 数组：</ >
<pre><code class="highlight-chroma">VarHandle longArrayViewHandle = MethodHandles.byteArrayViewVarHandle(
        long[].class, j va.nio.ByteOrder.BIG_ENDIAN);
</code></pre>
Although similar mechanisms can be achieved using java.nio.ByteBuffer, it requires that a ByteBuffer instance be created wrapping a byte array. This does not always guarantee reliable performance due to the fragility of escape analysis and that accesses have to go through the yteBuffer instance. In the case of unaligned access all but the plain access mode methods will hrow IllegalStateException. For aligned access certain volatile operations, depending on the v riable type are possible. Such VarHandle instances may be utilized to vectorize array access.< p>
尽管同样的效果可以通过 java.nio.ByteBuffer 得到，但这种方式需要一个 ByteBuffer 实例用于 裹 byte 数组。由于这导致了脆弱的逃逸分析，它并不总是能够得到可接受的性能并且每次访问都需 通过一个 ByteBuffer 实例。在非对其访问的情况下，除了普通（plain）的方法模式方法，都会抛出 Il egalStateException 异常。对齐访问的情况下，取决于变量的类型，一些 volatile 访问模式是允许 。这些 VarHandle 可以用来向量化（vectorize）数组存取操作。
The number of arguments, the argument types, and return type of access mode methods are governed by variable kind, the variable type and the characteristics of the access mode. V rHandle creation methods (such as those previously described) will document the requiremen s. For example, a compareAndSet on the previously-looked up VH_FOO_FIELD_I handle requir s 3 arguments, an instance of receiver Foo and two ints for the expected and actual values:</ >
访问模式方法的参数的数量、参数的类型、返回值的类型取决于变量种类（viriable kind）、变 类型和访问模式的特性。VarHandle 的生成方法（我们前面提到的那些）会在文档里说明必要条件。 如，对前面我们所生成的 VH_FOO_FIELD_I 调用 compareAndSet 需要 3 个参数，一个 Foo 实例作 接收者，一个 int 作为 expected value 和另一个作为 actual value：
<pre><code class="highlight-chroma">Foo f = ...
boolean r = VH_F O_FIELD_I.compareAndSet(f, 0, 1);

</code></pre>
In contrast, a getAndSet requires 2 arguments, an instance of receiver Foo and one int th t is the value to be set:
相对的，getAndSet 只需要两个参数，一个 Foo 实例作为接收者，一个 int 用于设置值：
<pre><code class="highlight-chroma">int o = (int) VH_FOO_FIELD_I.getAndSet(f, 2);
</code></pre>
Access to array elements will require an additional argument, of type int, between the re eiver and value arguments (if any), that corresponds to the array index of the element to be o erated upon.
访问数组元素的时候需要一个额外的 int 型的参数，它位于接收者和其他参数之间（如果有的话，这个参数对应于需要操作的元素的下标。
For predictable behaviour and performance at runtime VarHandle instances should be he d in static final fields (as required for instances of Atomic{Int, Long, Reference}FieldUpdater). his ensures that constant folding will occur for access mode method invocations, such as fold ng away method signature checks and/or argument cast checks.
为了可预测的行为和运行时性能，VarHandle 实例必须放在一个 static final 的字段里（就跟 At mic{Int, Long, Reference}FieldUpdater 所要求的那样）。这可以保证在调用访问模式方法的时候会生常量折叠，例如去掉方法签名的检查和/或参数的类型转换检查。
Note: Future HotSpot enhancements might support constant folding forVarHandle, or M thodHandle, instances held in non-static final fields, method arguments, or local variables.</ >
注：将来的 HotSpot 增强可能会支持没有使用 static final 持有的 VarHandle 和 MethodHandl 。
A MethodHandle may be produced for a VarHandle access mode method by usingMeth dHandles.Lookup.findVirtual. For example, to produce a MethodHandle to the "compareAndS t" access mode for a particular variable kind and type:
一个 MethodHandle 可以使用 VarHandle 的访问模式方法通过 MethodHandles.Lookup.find irtual 来生成。例如，下面给一个特定的变量类型和变量种类生成一个 compareAndSet 访问模式方对应的 MethodHandle：
<pre><code class="highlight-chroma">Foo f = ...
MethodHandle m ToVhCompareAndSet = MethodHandles.publicLookup().findVirtual(
        VarHandle.cl ss,
        "compareAn Set",
        MethodType methodType(boolean.class, Foo.class, int.class, int.class));
</code></pre>
The MethodHandle can then be invoked with a variable kind and type compatibleVarHan le instance as the first parameter:
MethodHandle 可以用一个变量种类和类型都兼容的 VarHandle 实例作为第一个参数来调用： /p>
<pre><code class="highlight-chroma">boolean r = (boolean) mhToVhCompareAndSet.invokeExact(VH_FOO_FIELD_I, f, 0, 1);
</code></pre>
Or mhToVhCompareAndSet can be bound to the VarHandle instance and then invoked:< p>
或者，mhToVhCompareAndSet 可以绑定到一个 VarHandle 实例然后再调用：
<pre><code class="highlight-chroma">MethodHandle mhToBoundVhCompareAndSet = mhToVhCompareAndSet
        .bindTo(VH_

OO_FIELD_I);
boolean r = (bool
an) mhToBoundVhCompareAndSet.invokeExact(f, 0, 1);
</code></pre>
Such a MethodHandle lookup using findVirtual will perform an asTypetransformation to
djust arguments and return values. The behaviour is equivalent to a MethodHandle produced
using MethodHandles.varHandleInvoker, the analog of MethodHandles.invoker`:
像这样的使用 findVirtual 进行的 MethodHandle 查找会使用一个 asType 转换来调整参数然
再返回结果。这个行为跟使用 MethodHandles.invoker 的类比物 MethodHandles.varHandleInvok
r 来生成 MethodHandle 是一样的：
<pre><code class="highlight-chroma"><span class="highlight
cl">MethodHandle mhToVhCompareAndSet = MethodHandles.varHandleExactInvoker(
        VarHandle.A
cessMode.COMPARE_AND_SET,
        MethodType
methodType(boolean.class, Foo.class, int.class, int.class));
boolean r = (bool
an) mhToVhCompareAndSet.invokeExact(VH_FOO_FIELD_I, f, 0, 1);
</code></pre>
Thus a VarHandle may be used in erased or reflective scenarios by a wrapping class, for e
ample replacing the Unsafe usages within thejava.util.concurrent.AtomicFieldUpdater/A
omicArray classes. (Although further work is required such that the updaters are grant
d access to the look up fields in the declaring class.)
所以通过包装在一个类中， VarHandle 可以在（类型被擦除）或反射的情景下使用。比方说，
来替代 java.util.concurrent.AtomicFieldUpdater/AtomicArray 中对 Unsafe 的使用
尽管需要更进一步的工作，以保证这些 updater 对相应的字段用于足够的访问权限）。
The source compilation of an access mode method invocation will follow the same rules
s for signature-polymorphic method invocation toMethodHandle.invokeExact and MethodHa
dle.invoke. The following additions will be required to the Java Language Specification:

Make reference to the signature-polymorphic access mode methods in the VarHandle cla
s.
Allow signature-polymorphic methods to return types other than Object, indicating that t
e return type is not polymorphic (and would otherwise be declared via a cast at the call site).
his makes it easier invoke write-based access methods that return void and invoke compareA
dSet that returns a boolean value.

对访问模式方法的调用的编译跟具有签名多态的 MethodHandle.invokeExact 和 MethodHandl
.invoke 所遵守的规则是一样的。下面这些是对 Java 语言规范所附加的内容：
1. 生成对 VarHandle 的签名多态的访问模式方法的引用
2. 允许签名多态方法返回不是 Object 类型的值，这意味着返回值类型不再是多态的（并且也因
可以在调用的地方声明一个强制类型转换）。这可以让写访问模式方法放回 void，compareAndSet
回 boolean 变得更容易。
It would be desirable, but not a requirement, that source compilation of a signature-pol
morphic method invocation be enhanced to perform target typing of the polymorphic return
ype such that an explicit cast is not required.
如果对签名多态的方法的调用行为可以增强为自动识别返回值的类型会很好，但这不是必须的。<
p>
Note: a syntax and runtime support for looking up a MethodHandle or aVarHandle lever
ging the syntax of method references, such as VarHandle VH_FOO_FIELD_I = Foo::i is desirable
but not in scope for this JEP.
注：使用像方法引用那样的语法来生成 VarHandle 和 MethodHandle，比方说 VarHandle VH_
OO_FIELD_I = Foo::i，它所需要的语法和运行时支持是可取的，但不会在这篇 JEP 里讨论。
The runtime invocation of an access mode method invocation will follow similar rules as

or signature-polymorphic method invocation toMethodHandle.invokeExact and MethodHande.invoke. The following additions will be required to the Java Virtual Machine Specification:</>

Make reference to the signature-polymorphic access mode methods in the VarHandle class.
Specify invokevirtual byte code behaviour of invocation to access mode signature-polymrphic methods. It is anticipated that such behaviour can be specified by defining a transformaton from the access mode method invocation to a MethodHandle which is then invoked usingnvokeExact with the same parameters (see previous use ofMethodHandles.Lookup.findVirtual)

运行时对访问模式方法的调用跟使用 MethodHandle.invokeExact 和 MethodHandle.invoke 行签名多态方法调用所遵循的规则是类似的。下面是对 Java 虚拟机规范所附加的要求：
1. 在 VarHandle 内引用签名多态的访问模式方法
2. 定义对签名多态的访问模式方法进行 invokevirtual 时的行为。预期这种行为会通过一个从访模式方法的调用到对应的 MethodHandle 之间的使用相同参数的转换来定义（参考前面对 MethodHndles.Lookup.findVirtual 的使用）。
It is important that the VarHandle implementations for the supported variable kinds, type and access modes are reliably efficient and meet the performance goals. Leveraging signatur-polymorphic methods helps in terms of avoiding boxing and array packing. Implementations will:

Reside in the java.lang.invoke package where HotSpot treats final fields of classes in that ackage as really final, which enables constant folding when the VarHandle itself is referenced n a static final field;
Leverage the JDK internal annotations @Stable for constant folding of values that change only once, and @ForceInline to ensure methods get inlined even if normal inlining thresholds are reached; and
Use sun.misc.Unsafe for underlying enhanced volatile access.

VarHandle 对于所支持的变量类型、种类能够具有可靠的效率以达到目标性能要求是非常重要。利用签名多态的方法可以避免自动装箱和数组的打包。（Java）实现必须：

在包 java.lang.invoke 的内部，HotSpot 将类中的 final 字段认为是真正的 final，这使得 VarHandle 被 static final 域引用的时候可以进行常量折叠。
利用 JDK 内部的 @Stable 为那些仅改变一次的值进行常量折叠，利用 @ForceInline 来保证方即使已经达到普通方法的 inline 上限也会被 inline
使用 sun.misc.Unsafe 实现底层增强的 volatile 访问

A couple of HotSpot intrinsics are necessary, some of which are enumerated as follows:

An intrinsic for Class.cast, which has already been added (see JDK-8054492). Before this intrinsic was added a constant fold Class.castwould leave behind redundant checks that may cause unnecessary de-optimizations.

An intrinsic for an acquire-get access mode that can synchronize with an intrinsic for a se-release access mode (seesun.misc.Unsafe.putOrdered{Int, Long, Object}) when concurrently accessing variables.

Intrinsics for array bounds checks <a href="https://ld246.com/forward?goto=https%3A%F%2Fbugs.openjdk.java.net%2Fbrowse%2FJDK-8042997" target="_blank" rel="nofollow ugc"JDK-8042997. Static methods can be added java.util.Arrays that perform such checks and accept a function that is invoked to return an exception to be thrown or string message, to be included in an exception to be thrown, if the check fails. Such intrinsics enable better comparions using unsigned values (since an array length is always positive) and better hoisting of rane checks outside of unrolled loops over the array elements.

一些 HotSpot 固有的支持（intrinsics）是必须的，部分罗列如下：

对 Class.cast 的支持，它已经被添加了（参考 JDK-8054492）。在虚拟机添加这个支持前，一常量折叠的 Class.cast 还会遗留冗余的检查，这会导致不必要的性能损失。
当并发访问时，acquire-get 访问模式能够与 set-release 访问模式进行同步（参考 sun.misc.Unafe.putOrdered{Int, Long, Object}）。
对数组范围检查 JDK-8042997 的原生支持。静态方法可以被添加到 java.util.Arrays 来做这个查，它接受一个待调用的函数，然后在检查出错的情况下，返回一个异常或者一个出错消息，这个错消息可以被用于包含在一个待抛出的异常中。像这样的原生支持可以使用无符号数进行更好地比较（竟，数组长度总是正的）并且更好地把范围检查提升到一个被展开（unrolled）了的循环的外面进行查。

In addition further improvements to range checks by HotSpot have been implemented (a href="https://ld246.com/forward?goto=https%3A%2F%2Fbugs.openjdk.java.net%2Fbrows%2FJDK-8073480" target="_blank" rel="nofollow ugc">JDK-8073480) or are needed (<ahref="https://ld246.com/forward?goto=https%3A%2F%2Fbugs.openjdk.java.net%2Fbrowse%FJDK-8003585" target="_blank" rel="nofollow ugc">JDK-8003585 to strength reduce rage checks in say the fork/join framework or in say HashMap or ConcurrentHashMap).
此外，HotSpot 里更近一步的范围检查已经在 JDK-8073480 实现了（JDK-8003585 则用于强去除 fork/join 框架、HashMap 和 ConcurrentHashMap 里的范围检查）。
The VarHandle implementations should have minimal dependencies on other classes with the java.lang.invoke package to avoid increasing startup time and to avoid cyclic dependences occurring during static initialization. For example, ConcurrentHashMap is used by such clases and if ConcurrentHashMap is modified to use VarHandles it needs to be ensured no cyclic ependencies are introduced. Other more subtle cycles are possible with the use of ThreadLocaRandom and its use of AtomicInteger. It is also is desirable that the C2 HotSpot compilation tme is not unduly increased for methods containing VarHandle method invocations.
VarHandle 的实现必须保持对 java.lang.invoke 包里的其他类的最小依赖，以避免启动时间的加和在静态初始化时产生循环依赖。比方说，如果 VarHandle 的某些实现使用 ConcurrentHashMa，而 ConcurrentHashMap 也被修改成使用了 VarHandle，此时必须保证没有引入循环依赖。另一更微妙的循环是 ThreadLocalRandom 和他对 AtomicInteger 的使用。保证 HotSpot 的 C2 编译器译时间不会因为对 VarHandle 的使用而过度增加也是很值得要的。
<h2 id="Memory-fences-内存栅栏-">Memory fences（内存栅栏）</h2>
Fenced operations are defined as static methods on the VarHandle class and represents a minimal viable set for fine grained control of memory ordering.
屏障操作（fenced operations）作为 VarHandle 的静态方法来定义，是一个最小可用的精细制内存顺序工具集。
<pre><code class="highlight-chroma">/**
* Ensures that loas and stores before the fence will not be
* reordered with loads and stores after the fence.

```
</span></span><span class="highlight-line"><span class="highlight-cl">*
</span></span><span class="highlight-line"><span class="highlight-cl">* @apiNote Ignor
ng the many semantic differences from C and
</span></span><span class="highlight-line"><span class="highlight-cl">* C++, this metho
 has memory ordering effects compatible with
</span></span><span class="highlight-line"><span class="highlight-cl">* atomic_thread_f
nce(memory_order_seq_cst)
</span></span><span class="highlight-line"><span class="highlight-cl">*/
</span></span><span class="highlight-line"><span class="highlight-cl">public static void
ullFence() {}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">/**
</span></span><span class="highlight-line"><span class="highlight-cl">* Ensures that loa
s before the fence will not be reordered with
</span></span><span class="highlight-line"><span class="highlight-cl">* loads and stores
after the fence.
</span></span><span class="highlight-line"><span class="highlight-cl">*
</span></span><span class="highlight-line"><span class="highlight-cl">* @apiNote Ignor
ng the many semantic differences from C and
</span></span><span class="highlight-line"><span class="highlight-cl">* C++, this metho
 has memory ordering effects compatible with
</span></span><span class="highlight-line"><span class="highlight-cl">* atomic_thread_f
nce(memory_order_acquire)
</span></span><span class="highlight-line"><span class="highlight-cl">*/
</span></span><span class="highlight-line"><span class="highlight-cl">public static void
cquireFence() {}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">/**
</span></span><span class="highlight-line"><span class="highlight-cl">* Ensures that loa
s and stores before the fence will not be
</span></span><span class="highlight-line"><span class="highlight-cl">* reordered with s
ores after the fence.
</span></span><span class="highlight-line"><span class="highlight-cl">*
</span></span><span class="highlight-line"><span class="highlight-cl">* @apiNote Ignor
ng the many semantic differences from C and
</span></span><span class="highlight-line"><span class="highlight-cl">* C++, this metho
 has memory ordering effects compatible with
</span></span><span class="highlight-line"><span class="highlight-cl">* atomic_thread_f
nce(memory_order_release)
</span></span><span class="highlight-line"><span class="highlight-cl">*/
</span></span><span class="highlight-line"><span class="highlight-cl">public static void
eleaseFence() {}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">/**
</span></span><span class="highlight-line"><span class="highlight-cl">* Ensures that loa
s before the fence will not be reordered with
</span></span><span class="highlight-line"><span class="highlight-cl">* loads after the f
nce.
</span></span><span class="highlight-line"><span class="highlight-cl">*/
</span></span><span class="highlight-line"><span class="highlight-cl">public static void
oadLoadFence() {}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">/**
</span></span><span class="highlight-line"><span class="highlight-cl">* Ensures that sto
```

es before the fence will not be reordered with
* stores after the f
nce.
*/
public static void s
oreStoreFence() {}
</code></pre>
A full fence is stronger (in terms of ordering guarantees) than an acquire fence which is st
onger than a load load fence. Likewise a full fence is stronger than a release fence which is str
nger than a store store fence.
一个 full fense 比 acquire fence 要更强一些（在对排序的保证这一意义上），后者又比 load lo
d fence 更强。类似的，full fence 比 release fence 更强，后者比 store store fence 又更强。
<h2 id="Reachability-fence-可访问性栅栏-">Reachability fence（可访问性栅栏）</h2>
The reachability fence is defined as a static method on java.lang.ref.Reference:
可访问行栅栏作为静态方法定义在 java.lang.ref.Reference 中：
<pre><code class="language-go highlight-chroma"><span cla
s="highlight-cl">class java
/span>.lang<span c
ass="highlight-p">.ref<span class="highlight-p
>.Reference {</spa
>
  <span class="hi
hlight-c1">// add:
<span cla
s="highlight-c1">
  <span class="hi
hlight-cm">/**
<span cla
s="highlight-cm">   * Ensures that the object referenced by the given reference
<span cla
s="highlight-cm">   * remains strongly reachable (as defined in the {
link
<span cla
s="highlight-cm">   * java.lang.ref} package documentation), regardless of any prior
<span cla
s="highlight-cm">   * actions of the program that might otherwise cause the object to
<span cla
s="highlight-cm">   * become unreachable; thus, the referenced object is not
<span cla
s="highlight-cm">   * reclaimable by garbage collection at least until after the
<span cla
s="highlight-cm">   * invocation of this method. Invocation of this method does not
<span cla
s="highlight-cm">   * itself initiate garbage collection or finalization.
<span cla
s="highlight-cm">   *
<span cla
s="highlight-cm">   * @param ref the reference. If null, this method has no effect.
<span cla
s="highlight-cm">   */
  <span class="hi
hlight-nx">public static <span class="highlight
nx">void reachabilityFence<span class="highli
ht-p">(Object ref<

span>) {}

<span class="high
ight-p">}
</code></pre>
See <a href="https://ld246.com/forward?goto=https%3A%2F%2Fbugs.openjdk.java.net
2Fbrowse%2FJDK-8133348" target="_blank" rel="nofollow ugc">JDK-8133348.
It is currently out of scope to provide an annotation, @Finalized say, to be declared on a
method, which at either compile or runtime results in as if the method body was wrapped as
ollows:
现在已经太迟了，无法添加一个类似于 @Finalized 的东西，用于修饰一个方法，使得在编译时
运行时对应的方法体看起来像下面这样：
<pre><code class="highlight-chroma"><span class="highlight
cl">try {
    <method bo
y>
} finally {
    Reference.reac
abilityFence(this);
}
</code></pre>
It is anticipated that such functionality could be supported by a compile-time annotation
processor.
可以预感，类似的机制将会在某些编译期处理器得到支持。
<h2 id="Alternatives-其它选择-">Alternatives（其它选择）</h2>
Introducing new forms of "value type" were considered that support volatile operations.
owever, this would be inconsistent with properties of other types, and would also require mor
effort for programmers to use. Reliance uponjava.util.concurrent.atomic FieldUpdaters was al
o considered, but their dynamic overhead and usage limitations make them unsuitable.
引入一种新形式的值类型（value type）用于支持 volatile 操作。然而，这会导致跟其他类型的
质不一致，程序员也需要付出更多努力来学习使用它。也考虑过依靠 java.util.concurrent.atomic Fie
dUpdaters 来完成这一目标，但它们的动态损耗（dynamic overhead）和使用限制使得这一选项并
适用的。
Several other alternatives, including those based on field references, have been raised an
dismissed as unworkable on syntactic, efficiency, and/or usability grounds over the many yea
s that these issues have been discussed.
一些其他的选择，包括那些基于字段引用（field references）的方法在这些年都有人提出并讨
过，但最终因为语法上不可行、效率或者可用性问题消失了。
Syntax enhancements were considered in a previous version of this JEP but were deemed
too "magical", with the overloaded use of the volatile keyword scoping to floating interfaces,
ne for references and one for each supported primitive type.
语法增强在这个 JEP 之前的版本考虑过，但被认为太过于奇异（magical）了。它重载了 volatile
关键字的语义并扩展到飘浮接口（译者注：with the overloaded use of the volatile keyword scopi
g to floating interfaces），一个用于引用类型而另一个用于所有支持的基本类型（primitive type
。
Generic types extending from VarHandle were considered in a previous version of this JE
but such an addition, with enhanced polymorphic signatures for generic types and special tr
atment of boxed type variables, was considered immature given a future Java release with val
e types and generics over primitives with <a href="https://ld246.com/forward?goto=http%3
%2F%2Fopenjdk.java.net%2Fjeps%2F218" target="_blank" rel="nofollow ugc">JEP 218,
nd improved arrays with <a href="https://ld246.com/forward?goto=http%3A%2F%2Fcr.open
dk.java.net%2F%7Ejrose%2Fpres%2F201207-Arrays-2.pdf" target="_blank" rel="nofollow ugc
>Arrays 2.0.
上一个版本的 JEP 也考虑过从 VarHandle 扩展出泛型类型（generic type），但这个带有多态

名的泛型加上对自动装箱类型的特殊对待，被认为是不成熟的。因为将来的 Java 版本会带有值类型（alue type）、允许基于基本数据类型的泛型（参考 JEP-218）和一个增强的数组 Arrays 2.0。
An implementation-specific invokedynamic approach was also considered in a previous vrsion of this JEP. This required that compiled method calls with and without invokedynamic wre carefully aligned to be the same in terms of semantics. In addition the use of invokedynami in core classes such as sayConcurrentHashMap will result in cyclic dependencies.
基于特定实现的 invokedynamic 这一方法在这个 JEP 的之前版本也考虑过。这需要仔细地让带不带 invokedynamic 的编译后的方法调用在语义上保持一致。此外，一些使用了 invokedynamic 核心类，如 ConcurrentHashMap 将会导致循环依赖。
<h2 id="Testing-测试-">Testing（测试）</h2>
Stress tests will be developed using the jcstress harness.
压力测试将会使用 jcstress 工具来开发。
<h2 id="Risks-and-Assumptions-风险和假设-">Risks and Assumptions（风险和假设）</h2>
A prototype implementation of VarHandle has been performance-tested with nano-benchmarks and fork/join benchmarks, where the fork/join library's use ofsun.misc.Unsafe was replced with VarHandle. No major performance issues have been observed so far, and the HotSpo compiler issues identified do not seem onerous (folding cast checks and improving array bonds checks). We are therefore confident of the feasibility of this approach. However, we expec that it will require more experimentation to ensure the compilation techniques are reliable in he performance-critical contexts where these constructs are most often needed.
有个 VarHandle 的原型实现已经使用 nano-benchmarks 和 fork/join benchmarks 进行了性测试，其中 fork/join 使用了 sun.misc.Unsafe 的地方都替换成了 VarHandle。目前为止还没有发现显的性能损失，HotSpot 上的问题也都不太麻烦（折叠掉强制类型转换检查和改进数组范围检查）。们对这个方法的可行性是有信心的。尽管如此，我们也启动能够进行更多的实验，来保证在性能要求常严格的环境下有可靠的编译技术，因为这种情况会更需要 VarHandle。
<h2 id="Dependences-依赖-">Dependences(依赖)</h2>
The classes in java.util.concurrent (and other areas identified in the JDK) will be migrated rom sun.misc.Unsafe to VarHandle.
This JEP does not depend on JEP 188: Java Memory Model Update.
那些在包 java.util.concurrent 里的类（包括 JDK 中其他一下地方）会从 sun.misc.Unsafe 迁到 VarHandle。
这篇 JEP 不依赖于 JEP 188: Java Memory Model Update。
<h2 id="参考文献">参考文献</h2>
JEP 193: Variable Handles
翻译 - JEP 193：Variable Handles | Jekton