Дескрипторы переменных Java 9

Очень важно помнить, что **режимы доступа отменят предыдущие эффекты упорядочения памяти**. Это означает, что, например, если мы используем *VarHandle.get ()*, это будет обычная операция чтения, даже если мы объявили нашу переменную как *volatile.*

По этой причине разработчики должны быть предельно осторожны при использовании операций *VarHandle.*

**1. Вступление**

Java 9 принесла много новых полезных функций для разработчиков.

Одним из них является *[java.lang.invoke.VarHandle](https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/VarHandle.html)*API, представляющий дескрипторы переменных.

**2. Что такое переменные дескрипторы?**

Как правило, **дескриптор переменной - это просто типизированная ссылка на переменную**.

Переменная может быть элементом массива, экземпляром или статическим полем класса.

Класс *VarHandle*обеспечивает запись и чтение доступа к переменным при определенных условиях.

*VarHandles*являются неизменяемыми и не имеют видимого состояния. Более того, их нельзя подразделить на подклассы.

Каждый *VarHandle*имеет:

- общий тип T, который является типом каждой переменной, представленной этим VarHandle

- список типов координат CT, которые являются типами выражений координат, которые позволяют найти переменную, на которую ссылается этот VarHandle Список типов координат может быть пустым.

Цель *VarHandle*- определить стандарт для вызова **аналогов *java.util.concurrent.atomic*** и ***sun.misc.Unsafe*** для полей и элементов массива.

Эти операции в большинстве случаев являются атомарными или упорядоченными операциями. Например, приращение атомного поля.

**3. Создание дескрипторов переменных**

Чтобы использовать *VarHandle*, мы должны сначала иметь переменные.

Давайте объявим простой класс с различными переменными типа *int*, который мы будем использовать в наших примерах:

public class VariableHandlesTest {

public int publicTestVariable = 1;

private int privateTestVariable = 1;

public int variableToSet = 1;

public int variableToCompareAndSet = 1;

public int variableToGetAndAdd = 0;

public byte variableToBitwiseOr = 0;

}

**3.1. Дескрипторы переменных для открытых переменных**

* Теперь мы можем получить *VarHandle*для нашего *publicTestVariable*с помощью метода *findVarHandle ()*\*\* :

VarHandle publicIntHandle = MethodHandles.lookup()

.in(VariableHandlesTest.class)

.findVarHandle(VariableHandlesTest.class, "publicTestVariable", int.class);

assertThat(

publicIntHandle.coordinateTypes().size() == 1);

assertThat(

publicIntHandle.coordinateTypes().get(0) == VariableHandles.class);

Мы можем видеть, что *coordinateTypes*этого *VarHandle*не является пустым и имеет один элемент, который является нашим классом *VariableHandlesTest*.

**3.2. Дескрипторы переменных для частных переменных**

Если у нас есть закрытый член и нам нужен дескриптор переменной для такой переменной **, мы можем получить это с помощью метода *privateLookupIn ()***:

VarHandle privateIntHandle = MethodHandles

.privateLookupIn(VariableHandlesTest.class, MethodHandles.lookup())

.findVarHandle(VariableHandlesTest.class, "privateTestVariable", int.class);

assertThat(privateIntHandle.coordinateTypes().size() == 1);

assertThat(privateIntHandle.coordinateTypes().get(0) == VariableHandlesTest.class);

Здесь мы выбрали метод *privateLookupIn ()*, который имеет более широкий доступ, чем обычный *lookup () .*Это позволяет нам получить доступ к переменным *private, public*или *protected*.

До Java 9 эквивалентным API для этой операции был класс *Unsafe \_ и метод setAccessible () из API Reflection\_*.

Однако у этого подхода есть свои недостатки. Например, он будет работать только для конкретного экземпляра переменной.

*VarHandle*- лучшее и более быстрое решение в таких случаях.

**3.3. Дескрипторы переменных для массивов**

Мы могли бы использовать предыдущий синтаксис для получения полей массива.

Однако мы также можем получить *VarHandle*для массива определенного типа:

VarHandle arrayVarHandle = MethodHandles.arrayElementVarHandle(int[].class);

assertThat(arrayVarHandle.coordinateTypes().size() == 2);

assertThat(arrayVarHandle.coordinateTypes().get(0) == int[].class);

Теперь мы можем видеть, что у такого *VarHandle*есть два типа координат *int*и *[]*, которые представляют массив примитивов *int*.

**4. Вызов *VarHandle*методов**

* Большинство методов *VarHandle*ожидают переменное число аргументов типа Object. \*\* Использование *Object …​*в качестве аргумента отключает статическую проверку аргументов.

Вся проверка аргументов выполняется во время выполнения. Кроме того, разные методы ожидают разного количества аргументов разных типов.

Если мы не сможем дать правильное количество аргументов с правильными типами, вызов метода вызовет *WrongMethodTypeException.*

Например, *get ()*ожидает по крайней мере один аргумент, который помогает найти переменную, но *set ()*ожидает два аргумента, вторым будет являться значение, которое будет присвоено переменной.

**5. Режимы доступа с переменными**

Как правило, все методы класса *VarHandle \_\_*попадают в пять различных режимов доступа.

Давайте рассмотрим каждый из них в следующих подразделах.

**5.1. Доступ для чтения**

Методы с уровнем доступа для чтения позволяют получить значение переменной при заданных эффектах упорядочения памяти. Существует несколько методов с этим режимом доступа, таких как: *get (), getAcquire (), getVolatile ()*и *getOpaque () .*

Мы можем легко использовать метод *get ()*для нашего *VarHandle*:

assertThat((int) publicIntHandle.get(this) == 1);

Метод *get ()*принимает в качестве параметров только *CoordinateTypes*, поэтому мы можем просто использовать *this*в нашем случае.

**5.2. Доступ для записи**

Методы с уровнем доступа записи позволяют нам устанавливать значение переменной под конкретные эффекты упорядочения памяти.

Подобно методам с доступом для чтения, у нас есть несколько методов с доступом для записи: *set (), setOpaque (), setVolatile ()*и *setRelease () .*

Мы можем использовать метод *set ()*в нашем *VarHandle*:

publicIntHandle.set(this, 15);

assertThat((int) publicIntHandle.get(this) == 15);

Метод *set ()*ожидает как минимум два аргумента. Первый поможет найти переменную, а второй - значение, которое будет установлено для переменной.

**5.3. Доступ к обновлению Atomic**

Методы с этим уровнем доступа могут использоваться для атомарного обновления значения переменной.

Давайте используем метод *compareAndSet ()*, чтобы увидеть эффекты:

publicIntHandle.compareAndSet(this, 1, 100);

assertThat((int) publicIntHandle.get(this) == 100);

Помимо *\_CoordinateTypes,* метод*compareAndSet ()*принимает два дополнительных значения: *oldValue и newValue.*Метод устанавливает значение переменной, если оно равно*oldVariable\_*, или оставляет его неизменным в противном случае.

**5.4. Доступ к добавлению атомарных чисел**

Эти методы позволяют выполнять числовые операции, такие как *getAndAdd*() под определенными эффектами упорядочения памяти.

Давайте посмотрим, как мы можем выполнять атомарные операции, используя *VarHandle*:

int before = (int) publicIntHandle.getAndAdd(this, 200);

assertThat(before == 0);

assertThat((int) publicIntHandle.get(this) == 200);

Здесь метод *getAndAdd ()*сначала возвращает значение переменной, затем добавляет предоставленное значение.

**5.5. Побитовый атомарный доступ к обновлению**

Методы с таким доступом позволяют атомарно выполнять побитовые операции под определенными эффектами упорядочения памяти.

Давайте рассмотрим пример использования метода *getAndBitwiseOr ()*:

byte before = (byte) publicIntHandle.getAndBitwiseOr(this, (byte) 127);

assertThat(before == 0);

assertThat(variableToBitwiseOr == 127);

Этот метод получит значение нашей переменной и выполнит над ней побитовую операцию ИЛИ.

* Вызов метода вызовет *IllegalAccessException*, если он не сможет сопоставить режим доступа, требуемый методом, с режимом, разрешенным переменной. \*\*

Например, это произойдет, если мы попытаемся использовать метод *set ()*для переменной *final*.

**6. Эффекты упорядочения памяти**

Ранее мы упоминали, что методы ***VarHandle*разрешают доступ к переменным при определенных эффектах упорядочения памяти.**

Для большинства методов есть 4 эффекта упорядочения памяти:

* Обычные операции чтения и записи гарантируют побитовую атомарность для ссылок и примитивов до 32 бит. Кроме того, они не налагают ограничений на порядок по отношению к другим признакам.
* Opaque операции являются побитовыми атомарными и когерентно упорядочены в отношении доступа к одной и той же переменной.
* Операции Acquire и Release подчиняются свойствам Opaque. Кроме того, чтение с получением будет упорядочено только после того, как будет совпадать с записью в режиме освобождения.
* Volatile операции полностью упорядочены по отношению друг к другу.

Очень важно помнить, что **режимы доступа отменят предыдущие эффекты упорядочения памяти**. Это означает, что, например, если мы используем *get ()*, это будет обычная операция чтения, даже если мы объявили нашу переменную как *volatile.*

По этой причине разработчики должны быть предельно осторожны при использовании операций *VarHandle*

**7. Заключение**

В этом уроке мы представили переменные дескрипторы и как их использовать.

Эта тема довольно сложна, поскольку переменные маркеры предназначены для низкоуровневых манипуляций, и их не следует использовать без необходимости