Руководство по sun.misc.Unsafe

Unsafe использовался в основном самими авторами Java, а также частью разработчиков сторонних библиотек для увеличения скорости работы. Сейчас, вместо Unsafe, для разработчиков сторонних библиотек, созданы Дескрипторы методов и переменных.

В первой части описано почему его не нужно применять.

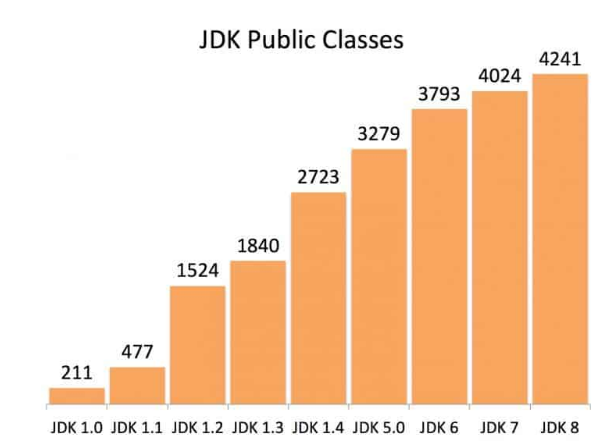
Во второй части частично (т.к. там 113 методов) описана его работа.

Начиная с java 9 пока был оставлен, но не гарантировалась совместимость с ним в последующих версиях.

Короче самому это механизм использовать не нужно. Только если код с ним уже написан. А с Java9 есть механизм “Переменных меток”/”Дескрипторов переменных”, реализующий основные функции, используемые через Unsafe.

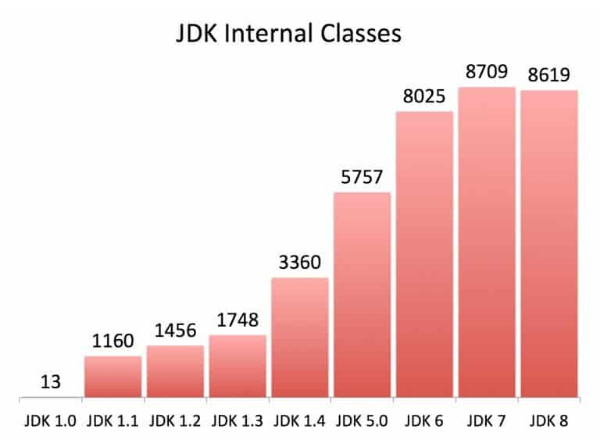
Часть 1

Если мы вернемся к JDK 1.0, то в общедоступном API будет 211 классов. Из интереса я создал график, показывающий рост публичных классов с течением времени. Чтобы извлечь мои данные, я использовал документацию API и скопировал список всех классов.



Я остановился на 8 JDK; JDK 9 и 10 имеют такое же количество открытых классов, что и JDK 8 (в доступных методах есть несколько изменений, но не классов).

Затем я взял файл rt.jar для каждого JDK и извлек список всех содержащихся в нем классов. Поскольку rt.jar был представлен только в JDK 1.2, я использовал файл classes.zip из JDK 1.0 и JDK 1.1. Удалив все внутренние классы, чтобы немного упростить ситуацию, я взял разницу между общим количеством классов и классов в общедоступном API. График ниже показывает эти данные.



Как вы можете видеть, в JDK 1.0 было только тринадцать внутренних классов; это достигло максимума в JDK 8 с колоссальным 8709. Я начал пытаться получить числа для JDK 9 и 10, но это оказалось слишком сложным из-за удаления файлов rt.jar и tools.jar, которые были заменены 97 модулями.

Интересно то, что начиная с JDK 1.1 и далее в JDK было скрыто много функциональных возможностей. Sun с самого начала предупредила разработчиков, а затем Oracle, что внутренние API не предназначены для использования при разработке приложений, не документированы и могут быть удалены из JDK без предварительного уведомления.

Несмотря на эти предупреждения, есть много разработчиков, которые использовали эти классы. Oracle провела анализ большого количества собственного кода, написанного на Java, и обнаружила, что используются три верхних внутренних класса:

1. sun.misc.BASE64Encoder
2. sun.misc.Unsafe
3. sun.misc.BASE64Decoder

Видимо, их код делает больше кодирования, чем декодирования. Использование этих двух классов было обусловлено, прежде всего, отсутствием общедоступного API BASE64 до версии 8 JDK.

Что приводит нас к самому печально известному внутреннему API в JDK: sun.misc.Unsafe. Подсказка, несомненно, заключается в названии, означающем этот класс, который позволит вам делать вещи, которые находятся далеко за пределами определенных границ обычного кода Java.

Я был удивлен, обнаружив, что класс Unsafe был представлен только в JDK

 Согласно [презентации, сделанной Марком Рейнхольдом на Языковом Саммите JVM](https://www.youtube.com/watch?v=4HG0YQVy8UM) , небезопасное введение было частью основной переписки Отражения и Сериализации, а также требовалось для поддержки прямых буферов и NIO.

Это безопасно?

Несмотря на отсутствие документов, [исходный код Unsafe](http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/sun/misc/Unsafe.java) легко доступен благодаря проекту [OpenJDK](http://openjdk.java.net/" \t "_blank) . Вы можете запустить javadoc в Unsafe.java и получить хотя бы минимальный набор документации для всех 113 (в JDK 8) доступных методов.

Тема этого поста не для обсуждения деталей небезопасной функциональности. Достаточно сказать, что создание экземпляра Unsafe требует больше работы, чем обычный класс. Конструктор является приватным, и, хотя есть статический метод getUnsafe (), вам нужно получить к нему доступ через bootclasspath. Вероятно, наиболее распространенный способ доступа к небезопасным - это отражение, чтобы получить доступ к внутреннему экземпляру.

Как только у вас есть доступ к Unsafe, перчатки будут сняты, так что вам лучше знать, что вы делаете, и будьте осторожны. Как следует из названия класса, многие вещи, которые вы можете сделать, небезопасны. Например, вы можете выделить и освободить собственную память (аналогично malloc и свободную в C). Вы также можете манипулировать памятью с помощью адресов, как если бы вы использовали указатели в C. Вы получаете доступ не только к памяти: вы можете делать такие вещи, как выделение нового экземпляра объекта, но не запускать конструктор. Подумайте, что может произойти, если вы используете этот объект. Одна фраза часто встречается в документации: «результаты не определены». Есть много способов использовать эти методы, которые будут возвращать бессмысленный результат (если вам повезет) или вызвать внезапную остановку JVM (если это не так).

Язык Java был разработан, чтобы быть безопасным. В этом вся суть исключения использования явных указателей и ручного управления памятью, а также многих других основных функций. Java никогда не предназначалась для использования в качестве языка системного программирования, где C и начал жизнь (старый добрый [UNIX)](https://en.wikipedia.org/wiki/Unix)). Для написания системного кода вам нужны эти типы низкоуровневых, потенциально опасных интерфейсов, которые позволят вам реализовать то, что требуется. Java всегда предназначалась для разработки кода приложения, который не нуждался бы в таком доступе напрямую. Концепция заключается в том, чтобы разработчики доверяли разработчикам JVM и базовых библиотек, чтобы гарантировать безопасность кода приложения. Класс Unsafe был введен, чтобы позволить разработчикам общедоступных классов использовать его для повышения производительности или использования низкоуровневых функций (таких как ограждение памяти), что было бы невозможно в стандартном коде Java.

Проблема, которая быстро стала очевидной, когда было предложено инкапсулировать все внутренние API в JDK 9, заключалась в том, что многие люди использовали эти классы. Было проведено [увлекательное исследование,](http://www.inf.usi.ch/faculty/lanza/Downloads/Mast2015a.pdf) которое проанализировало 74 Гб скомпилированного Java-кода из [Maven Central,](https://mvnrepository.com/repos/central" \t "_blank) рассматривая только использование sun.mis.Unsafe. Результаты показали, что 25% кода каким-то образом полагались на этот внутренний класс.

Эта значительная зависимость, особенно от популярных библиотек и сред с открытым исходным кодом, является большой причиной того, что [JEP 260](http://openjdk.java.net/jeps/260)  был включен в JDK 9. Он предоставляет модуль jdk.unsupported, который экспортирует внутренние пакеты, которые считаются критическими для JDK. (Интересно, что [документация API модуля по](https://docs.oracle.com/javase/9/docs/api/overview-summary.html)прежнему не предоставляет никакой информации по этому вопросу). Этот модуль экспортирует пакет com.sun.nio.file, экспортирует и открывает (для отражающего доступа) пакеты sun.misc и sun.reflect.

К счастью, Oracle провела [очистку класса Unsafe](http://mail.openjdk.java.net/pipermail/jdk9-dev/2016-February/003695.html) в JDK 9. Мне особенно понравился комментарий о связи между чрезмерным использованием extern и повышенным уровнем смертности котят.

Причина, по которой Unsafe интенсивно используется, заключается в том, что разработчикам библиотек необходим такой же повышенный уровень производительности и расширенные возможности, которые используются основными разработчиками Java API. Без этого многие корпоративные приложения будут работать намного медленнее, чем сейчас.

Это подводит меня к сути моего поста.

Java - чрезвычайно успешная платформа для разработки приложений, и причин для этого много. Сама безопасность, которую обеспечивает язык, является основной, почему разработчики считают его таким привлекательным. Не нужно беспокоиться о неправильных манипуляциях с указателями, забыть об освобождении памяти и утечках памяти (хотя они все еще могут быть в Java) делает надежный и быстрый код намного проще для написания. Тем не менее, наличие секретного соуса также было жизненно важно для успеха Java, поскольку оно позволило разработать мощные, высокопроизводительные библиотеки и фреймворки, предоставляя разработчикам множество функциональных возможностей для наращивания.

Мы уже видели, как части Unsafe были реализованы таким образом, чтобы они были доступны через публичный API. [Переменные метки](http://openjdk.java.net/jeps/193) , представленные в JDK 9, являются отличным примером этого. Разработчики могут использовать их для ограничения операций доступа к памяти и выполнения атомарных операций непосредственно над переменными без необходимости создавать экземпляры классов в  пакете [java.util.concurrent.atomic](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html" \t "_blank) .

Часть 2

**1. Обзор**

В этой статье мы рассмотрим захватывающий класс, предоставляемый JDK - *Unsafe*из пакета *sun.misc*. Этот класс предоставляет нам низкоуровневые механизмы, которые были разработаны для использования только базовой библиотекой Java, а не стандартными пользователями.

Это предоставляет нам низкоуровневые механизмы, в первую очередь предназначенные для внутреннего использования в основных библиотеках.

**2. Получение экземпляра *Unsafe***

Во-первых, чтобы иметь возможность использовать класс *Unsafe*, нам нужно получить экземпляр - что непросто, учитывая, что класс был разработан только для внутреннего использования.

* Способ получения экземпляра - через статический метод *getUnsafe () .*\*\* По умолчанию это то, что это вызовет исключение *SecurityException.*

К счастью, **мы можем получить экземпляр, используя отражение:**

Field f = Unsafe.class.getDeclaredField("theUnsafe");

f.setAccessible(true);

unsafe = (Unsafe) f.get(null);

**3. Создание класса с помощью *Unsafe***

Допустим, у нас есть простой класс с конструктором, который устанавливает значение переменной при создании объекта:

class InitializationOrdering {

private long a;

public InitializationOrdering() {

this.a = 1;

}

public long getA() {

return this.a;

}

}

Когда мы инициализируем этот объект с помощью конструктора, метод *getA ()*вернет значение 1:

InitializationOrdering o1 = new InitializationOrdering();

assertEquals(o1.getA(), 1);

Но мы можем использовать метод *allocateInstance ()*, используя *Unsafe.*Он только выделит память для нашего класса и не вызовет конструктор

InitializationOrdering o3

= (InitializationOrdering) unsafe.allocateInstance(InitializationOrdering.class);

assertEquals(o3.getA(), 0);

Обратите внимание, что конструктор не был вызван, и из-за этого метод *getA ()*вернул значение по умолчанию для типа *long*- 0.

**4. Изменение частных полей**

Допустим, у нас есть класс с закрытым значением *secret*:

class SecretHolder {

private int SECRET\_\_VALUE = 0;

public boolean secretIsDisclosed() {

return SECRET\_\_VALUE == 1;

}

}

Используя метод *putInt ()*из *Unsafe,*мы можем изменить значение частного поля *SECRET*VALUE\_\_, изменив/повредив состояние этого экземпляра:

SecretHolder secretHolder = new SecretHolder();

Field f = secretHolder.getClass().getDeclaredField("SECRET\_\_VALUE");

unsafe.putInt(secretHolder, unsafe.objectFieldOffset(f), 1);

assertTrue(secretHolder.secretIsDisclosed());

Как только мы получим поле с помощью вызова отражения, мы можем изменить его значение на любое другое значение *int*, используя *Unsafe*.

**5. Бросать исключение**

Код, который вызывается через *Unsafe*, не проверяется компилятором так же, как обычный код Java. Мы можем использовать метод *throwException ()*, чтобы генерировать любое исключение, не ограничивая вызывающего его обработчиком, даже если это проверенное исключение:

@Test(expected = IOException.class)

public void givenUnsafeThrowException\_\_whenThrowCheckedException\_\_thenNotNeedToCatchIt() {

unsafe.throwException(new IOException());

}

После создания *IOException,*, который отмечен, нам не нужно ни перехватывать его, ни указывать его в объявлении метода.

**6. Память вне кучи**

Если приложению не хватает доступной памяти на JVM, мы можем в конечном итоге заставить процесс GC запускаться слишком часто. В идеале мы хотели бы иметь специальную область памяти, вне кучи и не контролируемую процессом GC.

Метод *allocateMemory ()*из класса *Unsafe*дает нам возможность выделять огромные объекты из кучи, что означает, что **эта память не будет видна и учтена GC и JVM**.

Это может быть очень полезно, но мы должны помнить, что эта память должна управляться вручную и правильно восстанавливаться с помощью *freeMemory ()*, когда она больше не нужна.

Допустим, мы хотим создать большой массив байтов вне памяти. Мы можем использовать метод *allocateMemory ()*, чтобы добиться этого:

class OffHeapArray {

private final static int BYTE = 1;

private long size;

private long address;

public OffHeapArray(long size) throws NoSuchFieldException, IllegalAccessException {

this.size = size;

address = getUnsafe().allocateMemory(size \*\* BYTE); //создаем массив байтов вне кучи и

одновременно сохраняем в “address” начальный адрес массива

}

private Unsafe getUnsafe() throws IllegalAccessException, NoSuchFieldException { //получаем объект класса Unsafe

Field f = Unsafe.class.getDeclaredField("theUnsafe");

f.setAccessible(true);

return (Unsafe) f.get(null);

}

public void set(long i, byte value) throws NoSuchFieldException, IllegalAccessException {

getUnsafe().putByte(address + i \*\* BYTE, value);//принимает индекс и заданное значение

value , которые будут сохранены в массиве

}

public int get(long idx) throws NoSuchFieldException, IllegalAccessException {

return getUnsafe().getByte(address + idx \*\* BYTE);//извлекает значение байта,используя

его индекс, который является смещением от начального адреса массива

}

public long size() {

return size;

}

public void freeMemory() throws NoSuchFieldException, IllegalAccessException {

getUnsafe().freeMemory(address);

}

}

В конструкторе *OffHeapArray,*мы инициализируем массив с заданным размером *. Мы храним начальный адрес массива в поле*address *. Метод*set () *принимает индекс и заданное значение*value *, которые будут сохранены в массиве. Метод*get () \_\_ извлекает значение байта, используя его индекс, который является смещением от начального адреса массива.

Далее мы можем выделить этот массив вне кучи, используя его конструктор:

long SUPER\_\_SIZE = (long) Integer.MAX\_\_VALUE \*\* 2;

OffHeapArray array = new OffHeapArray(SUPER\_\_SIZE);

Мы можем поместить N чисел байтов в этот массив и затем извлечь эти значения, суммируя их, чтобы проверить, правильно ли работает наша адресация:

int sum = 0;

for (int i = 0; i < 100; i++) {

array.set((long) Integer.MAX\_\_VALUE + i, (byte) 3);

sum += array.get((long) Integer.MAX\_\_VALUE + i);

}

assertEquals(array.size(), SUPER\_\_SIZE);

assertEquals(sum, 300);

В конце нам нужно освободить память обратно в ОС, вызвав *freeMemory () .*

**7. *CompareAndSwap*Operation**

Очень эффективные конструкции из пакета *java.concurrent*, такие как *AtomicInteger,*используют методы *compareAndSwap ()*из *Unsafe*, чтобы обеспечить максимально возможную производительность. Эта конструкция широко используется в алгоритмах без блокировки, которые могут использовать инструкцию процессора CAS, чтобы обеспечить большую скорость по сравнению со стандартным механизмом пессимистической синхронизации в Java.

Мы можем построить счетчик на основе CAS, используя метод *compareAndSwapLong ()*из *Unsafe*:

class CASCounter {

private Unsafe unsafe;

private volatile long counter = 0;

private long offset;

private Unsafe getUnsafe() throws IllegalAccessException, NoSuchFieldException {

Field f = Unsafe.class.getDeclaredField("theUnsafe");

f.setAccessible(true);

return (Unsafe) f.get(null);

}

public CASCounter() throws Exception {

unsafe = getUnsafe();

offset = unsafe.objectFieldOffset(CASCounter.class.getDeclaredField("counter"));

}

public void increment() {

long before = counter;

while (!unsafe.compareAndSwapLong(this, offset, before, before + 1)) {

before = counter;

}

}

public long getCounter() {

return counter;

}

}

В конструкторе *CASCounter*мы получаем адрес поля счетчика, чтобы иметь возможность использовать его позже в методе *increment ()*.

Это поле должно быть объявлено как volatile, чтобы быть видимым для всех потоков, которые пишут и читают это значение. Мы используем метод *objectFieldOffset ()*, чтобы получить адрес памяти поля *offset*.

Наиболее важной частью этого класса является метод *increment ()*. Мы используем *compareAndSwapLong ()*в цикле *while*для увеличения ранее извлеченного значения, проверяя, изменилось ли это предыдущее значение с момента его получения.

Если это так, то мы повторяем эту операцию, пока не добьемся успеха. Здесь нет блокировки, поэтому это называется алгоритмом без блокировки.

Мы можем проверить наш код, увеличив общий счетчик из нескольких потоков:

int NUM\_\_OF\_\_THREADS = 1\_\_000;

int NUM\_\_OF\_\_INCREMENTS = 10\_\_000;

ExecutorService service = Executors.newFixedThreadPool(NUM\_\_OF\_\_THREADS);

CASCounter casCounter = new CASCounter();

IntStream.rangeClosed(0, NUM\_\_OF\_\_THREADS - 1)

.forEach(i -> service.submit(() -> IntStream

.rangeClosed(0, NUM\_\_OF\_\_INCREMENTS - 1)

.forEach(j -> casCounter.increment())));

Далее, чтобы утверждать, что состояние счетчика правильное, мы можем получить значение счетчика из него:

assertEquals(NUM\_\_OF\_\_INCREMENTS \*\* NUM\_\_OF\_\_THREADS, casCounter.getCounter());

**8. Парк/Распаркуйте**

В API *Unsafe*есть два замечательных метода, которые используются JVM для переключения потоков контекста. Когда поток ожидает какого-либо действия, JVM может заблокировать этот поток с помощью метода *park ()*из класса *Unsafe*.

Он очень похож на метод *Object.wait ()*, но он вызывает собственный код ОС, таким образом, используя некоторые особенности архитектуры для достижения максимальной производительности.

Когда поток заблокирован и его необходимо снова запустить, JVM использует метод *unpark ()*. \*\* Мы часто видим вызовы этого метода в дампах потоков, особенно в приложениях, использующих пулы потоков.

**9. Заключение**

В этой статье мы рассмотрели класс *Unsafe*и его наиболее полезную конструкцию.

Мы увидели, как получить доступ к закрытым полям, как выделить память вне кучи и как использовать конструкцию сравнения и обмена для реализации алгоритмов без блокировки.

Реализация всех этих примеров и фрагментов кода может быть найдена [over на GitHub](https://github.com/eugenp/tutorials/tree/master/core-java)- это проект Maven, поэтому его легко импортировать и беги как есть.