VitaliSergey 16 октября 2019 в 16:48

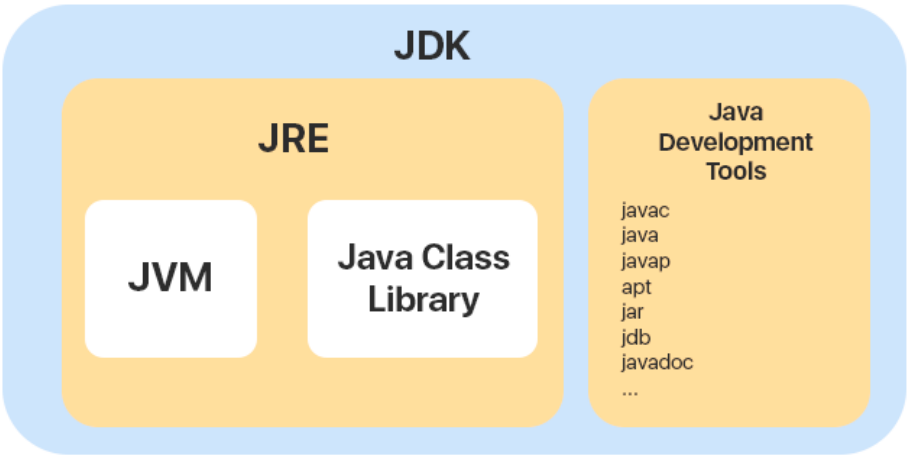
Инструменты для запуска и разработки Java приложений, компиляция, выполнение на JVM

Программирование, Java

Ни для кого не секрет, что на данный момент Java — один из самых популярных языков программирования в мире. Дата официального выпуска Java — 23 мая 1995 года.

Эта статья посвящена основам основ: в ней изложены базовые особенности языка, которые придутся кстати начинающим “джавистам”, а опытные Java-разработчики смогут освежить свои знания.

** Статья подготовлена на основе доклада Евгения Фраймана — Java разработчика компании IntexSoft.*
В статье присутствуют ссылки на внешние материалы.



1. JDK, JRE, JVM

Java Development Kit — комплект разработчика приложений на языке Java. Он включает в себя Java Development Tools и среду выполнения Java — JRE (Java Runtime Environment).

Java development tools включают в себя около 40 различных тулов: javac (компилятор), java (лаунчер для приложений), javap (java class file disassembler), jdb (java debugger) и др.

Среда выполнения JRE — это пакет всего необходимого для запуска скомпилированной Java-программы. Включает в себя виртуальную машину JVM и библиотеку классов Java — **Java Class Library**.

JVM — это программа, предназначенная для выполнения байт-кода. Первое преимущество JVM — это принцип *“Write once, run anywhere”*. Он означает, что приложение, написанное на Java, будет работать одинаково на всех платформах. Это является большим преимуществом JVM и самой Java.

До появления Java, многие компьютерные программы были написаны под определенные компьютерные системы, а предпочтение отдавалось ручному управлению памятью, как более эффективному и предсказуемому. Со второй половины 1990-х годов, после появления Java, автоматическое управление памятью стало общей практикой.

Существует множество реализаций JVM, как коммерческих, так и с открытым кодом. Одна из целей создания новых JVM — увеличение производительности для конкретной платформы. Каждая JVM пишется под платформу отдельно, при этом есть возможность написать ее так, чтобы она работала быстрее на конкретной платформе. Самая распространённая реализация JVM — это JVM Hotspot от OpenJDK. Также есть реализации IBM J9, Excelsior JET.

2. Выполнение кода на JVM

Согласно спецификации Java SE, для того, чтобы получить код, работающий в JVM, необходимо выполнить 3 этапа:

- Загрузка байт-кода и создание экземпляра класса Class
  Грубо говоря, чтобы попасть на JVM, класс должен быть загружен. Для этого существуют отдельные класс-загрузчики, к ним мы вернемся чуть позже.

- Связывание или линковка
  После загрузки класса начинается процесс линковки, на котором байт-код разбирается и проверяется. Процесс линковки в свою очередь происходит в 3 шага:

  — verification или проверка байт-кода: проверяется корректность инструкций, возможность переполнения стека на данном участке кода, совместимость типов переменных; проверка происходит один раз для каждого класса;
  — preparation или подготовка: на данном этапе в соответствии со спецификацией выделяется память под статические поля и происходит их инициализация;
  — resolution или разрешение: разрешение символьных ссылок (когда в байт-коде мы открываем файлы с расширением .class, мы видим числовые значения вместо символьных ссылок).

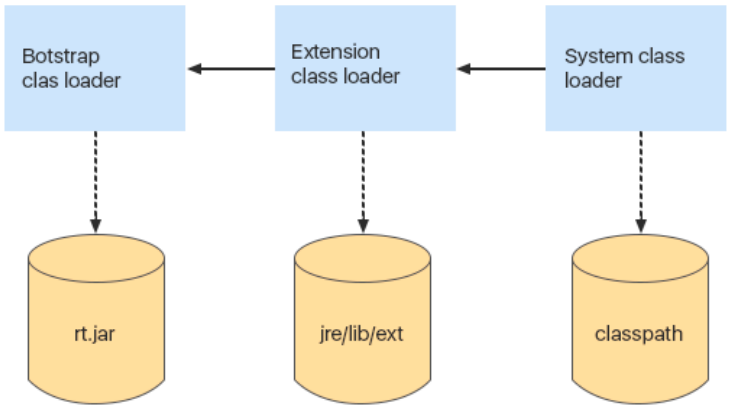
- Инициализация полученного объекта Class
  На последнем этапе класс, который мы создали, инициализируется, и JVM может начинать его исполнение.

3. Загрузчики классов и их иерархия

Вернемся к загрузчикам классов — это специальные классы, которые являются частью JVM. Они загружают классы в память и делают их доступными для выполнения. Загрузчики работают со всеми классами: и с нашими, и с теми, которые непосредственно нужны для Java.

Представьте ситуацию: мы написали свое приложение, и помимо стандартных классов там есть наши классы, и их очень много. Как с этим будет работать JVM? В Java реализована отложенная загрузка классов, иными словами lazy loading. Это значит, что загрузка классов не будет выполняться до тех пор, пока в приложении не встретится обращение к классу.

Иерархия загрузчиков классов



Первый загрузчик классов — это **Bootstrap classloader**. Он написан на C++. Это базовый загрузчик, который загружает все системные классы из архива *rt.jar*. При этом, есть небольшое отличие между загрузкой классов из *rt.jar* и наших классов: когда JVM загружает классы из *rt.jar*, она не выполняет все этапы проверки, которые выполняются при загрузке любого другого класс-файла т.к. JVM изначально известно, что все эти классы уже проверены. Поэтому, включать в этот архив какие-либо свои файлы не стоит.

Следующий загрузчик — это **Extension classloader.** Он загружает классы расширений из папки *jre/lib/ext*. Допустим, вы хотите, чтобы какой-то класс загружался каждый раз при старте Java машины. Для этого вы можете скопировать исходный файл класса в эту папку, и он будет автоматически загружаться.

Еще один загрузчик — **System classloader**. Он загружает классы из classpath'а, который мы указали при запуске приложения.

Процесс загрузки классов происходит по иерархии:

- В первую очередь мы запрашиваем поиск в кэше System Class Loader (кэш системного загрузчика содержит классы, которые уже были им загружены);
- Если класс не был найден в кэше системного загрузчика, мы смотрим кэш Extension class loader;
- Если класс не найден в кэше загрузчика расширений, класс запрашивается у загрузчика Bootstrap.

Если класс не найден в кэше Bootstrap, он пытается загрузить этот класс. Если Bootstrap не смог загрузить класс, он делегирует загрузку класса загрузчику расширений. Если на этот момент класс будет загружен, он остается в кэше у Extension classloader, а загрузка класса является завершенной.

4. Структура Class-файлов и процесс загрузки

Перейдем непосредственно к структуре Class-файлов.

Один класс, написанный на Java, компилируется в один файл с расширением .class. Если в нашем Java файле лежит несколько классов, один файл Java может быть скомпилирован в несколько файлов с расширением .class — файлов байт-кода данных классов.

Все числа, строки, указатели на классы, поля и методы хранятся в *Constant pool* — области памяти *Meta space*. Описание класса хранится там же и содержит имя, модификаторы, супер-класс, супер-интерфейсы, поля, методы и атрибуты. Атрибуты, в свою очередь, могут содержать любую дополнительную информацию.

Таким образом, при загрузке классов:

- происходит чтение класс-файла, т.е проверка корректности формата
- создается представление класса в Constant pool (Meta space)
- грузятся супер-классы и супер-интерфейсы; если они не будут загружены, то и сам класс не будет загружен

5. Исполнение байт-кода на JVM

В первую очередь, для исполнения байт-кода, JVM может его *интерпретировать*. Интерпретация — довольно медленный процесс. В процессе интерпретации, интерпретатор "бежит" построчно по класс-файлу и переводит его в команды, которые понятны JVM.

Также JVM может его *транслировать*, т.е. скомпилировать в машинный код, который будет исполняться непосредственно на CPU.

Команды, которые исполняются часто, не будут интерпретироваться, а сразу будут транслироваться.

6. Компиляция

Компилятор — это программа, которая преобразует исходные части программ, написанные на языке программирования высокого уровня, в программу на машинном языке, "понятную" компьютеру.

Компиляторы делятся на:

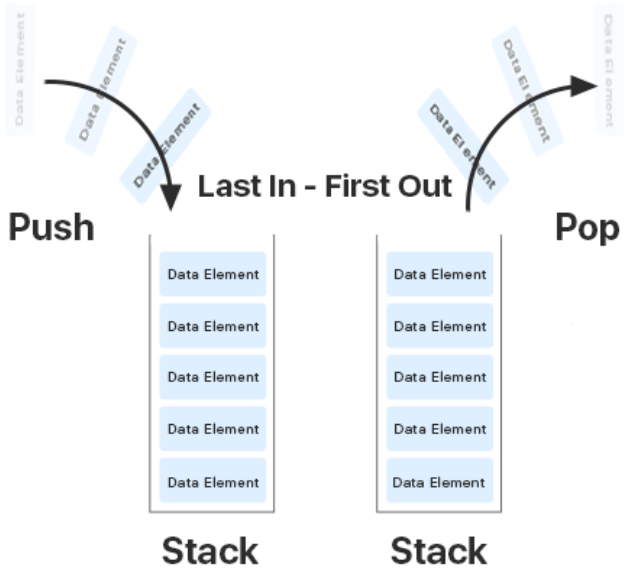
- *Не оптимизирующие*
- *Простые оптимизирующие* (Hotspot Client): работают быстро, но порождают неоптимальный код
- *Сложные оптимизирующие* (Hotspot Server): производят сложные оптимизирующие преобразования прежде чем сформировать байт-код

Также компиляторы могут классифицироваться по моменту компиляции:

- *Динамические компиляторы*
  Работают одновременно с программой, что сказывается на производительности. Важно, чтобы эти компиляторы работали на коде, который часто исполняется. Во время исполнения программы JVM знает, какой код выполняется чаще всего, и, чтобы постоянно не интерпретировать его, виртуальная машина сразу переводит его в команды, которые уже будут исполняться непосредственно на процессоре.
- *Статические компиляторы*
  Дольше компилируют, но порождают оптимальный код для исполнения. Из плюсов: не требуют ресурсов во время исполнения программы, каждый метод компилируется с применением оптимизаций.

7. Организация памяти в Java

Стек — это область памяти в Java, которая работает по схеме LIFO — "*Last in — Fisrt Out*" или "*Последним вошел, первым вышел*".



Он нужен для того, чтобы хранить методы. Переменные в стеке существуют до тех пор, пока выполняется метод в котором они были созданы.

Когда вызывается любой метод в Java, создается фрейм или область памяти в стеке, и метод кладется на его вершину. Когда метод завершает выполнение, он удаляется из памяти, тем самым освобождая память для следующих методов. Если память стека будет заполнена, Java бросит исключение *java.lang.StackOverFlowError*. К примеру, это может произойти, если у нас будет рекурсивная функция, которая будет вызывать сама себя и памяти в стеке не будет хватать.

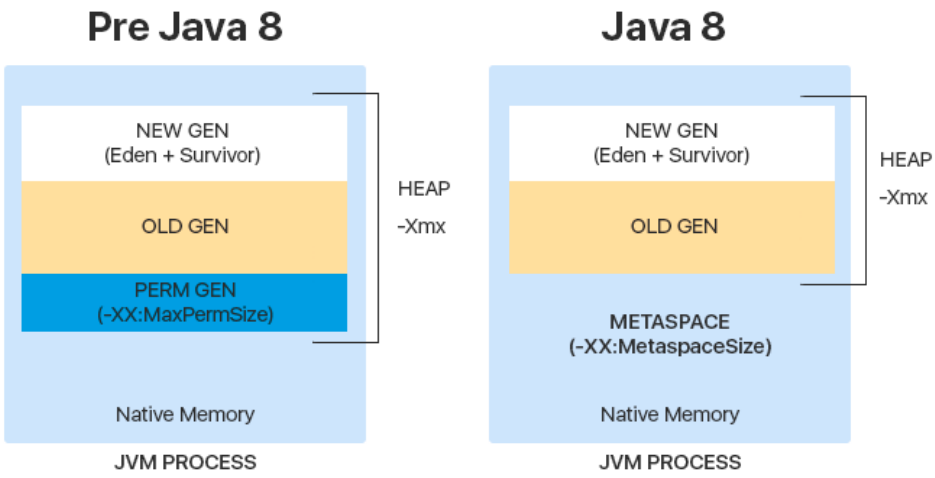
Ключевые особенности стека:

- Стек заполняется и освобождается по мере вызова и завершения новых методов
- Доступ к этой области памяти осуществляется быстрее, чем к куче
- Размер стека определяется операционной системой
- Является потокобезопасным, поскольку для каждого потока создается свой отдельный стек

Ещё одна область памяти в Java — **Heap** или **куча**. Она используется для хранения объектов и классов. Новые объекты всегда создаются в куче, а ссылки на них хранятся в стеке. Все объекты в куче имеют глобальный доступ, то есть к ним можно получить доступ из любой точки приложения.

Куча разбита на несколько более мелких частей, называемых поколениями:

- *Young generation* — область, где размещаются недавно созданные объекты
- *Old (tenured) generation* — область, где хранятся "долгоживущие" объекты
- До Java 8 существовала ещё одна область — *Permanent generation* — которая содержит метаинформацию о классах, методах, статических переменных. После появления Java 8 было решено хранить эту информацию отдельно, вне кучи, а именно в Meta space



Почему отказались от Permanent generation? В первую очередь, это из-за ошибки, которая была связана с переполнением области: так как Perm имел константный размер и не мог расширяться динамически, рано или поздно память заканчивалась, кидалась ошибка, и приложение падало.

Meta space же имеет динамический размер, и во время исполнения он может расширяться до размеров памяти JVM.

Ключевые особенности кучи:

- Когда эта область памяти заполняется полностью, Java бросает *java.lang.OutOfMemoryError*
- Доступ к куче медленнее, чем к стеку
- Для сбора неиспользуемых объектов работает сборщик мусора
- Куча, в отличие от стека, не является потокобезопасной, так как любой поток может получить к ней доступ

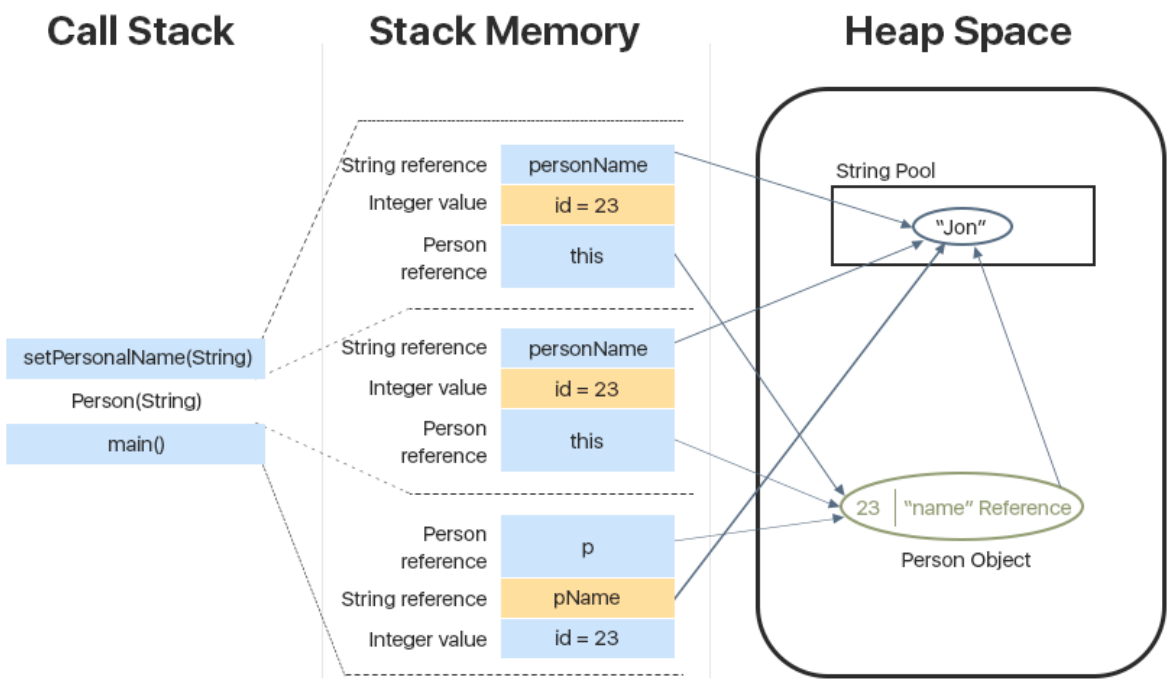
Основываясь на информации выше, рассмотрим, как происходит управление памятью на простом примере:

```
public class App {
    public static void main(String[] args) {
        int id = 23;
        String pName = "Jon";
        Person p = null;
        p = new Person(id, pName);
    }
}
class Person {
    int pid;
    String name;

    // constructors, getters/setters
}
```

У нас есть класс App, в котором единственный метод *main* состоит из:

— примитивной переменой **id** типа **int** со значением **23**
— ссылочной переменной **pName** типа **String** со значением **Jon**
— ссылочной переменной **p** типа **person**



Как уже упоминалось, при вызове метода на вершине стека создаётся область памяти, в которой хранятся данные, необходимые этому методу для выполнения.
В нашем случае, это ссылка на класс *person*: сам объект хранится в куче, а в стеке хранится ссылка. Также в стек кладется ссылка на строку, а сама строка хранится в куче в String pool. Примитив хранится непосредственно в стеке.

Для вызова конструктора с параметрами *Person (String)* из метода *main()* в стеке, поверх предыдущего вызова *main()* создается в стеке отдельный фрейм, который хранит:

— **this** — ссылка на текущий объект
— примитивное значение **id**
— ссылочную переменную **personName**, которая указывает на строку в String Pool.

После того, как мы вызвали конструктор, вызывается *setPersonName()*, после чего снова создается новый фрейм в стеке, где хранятся те же данные: ссылка на объект, ссылка на строку, значение переменной.

Таким образом, когда выполнится метод *setter*, фрейм пропадет, стек очистится. Далее выполняется конструктор, очищается фрейм, который был создан под конструктор, после чего метод *main()* завершает свою работу и тоже удаляется из стека.

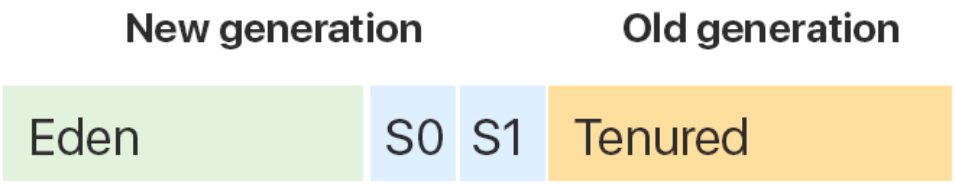
Если будут вызваны другие методы, для них будут также созданы новые фреймы с контекстом этих конкретных методов.

8. Garbage collector

В куче работает Garbage collector — программа, работающая на виртуальной машине Java, которая избавляется от объектов, к которым невозможно получить доступ.

Разные JVM могут иметь различные алгоритмы сборки мусора, также существуют разные сборщики мусора.

Мы поговорим о самом простом сборщике **Serial GC**. Сборку мусора мы запрашиваем при помощи *System.gc()*.



Как уже было упомянуто выше, куча разбита на 2 области: New generation и Old generation.

New generation (младшее поколение) включает в себя 3 региона: *Eden*, *Survivor 0* и *Survivor 1*.

Old generation включает в себя регион *Tenured*.

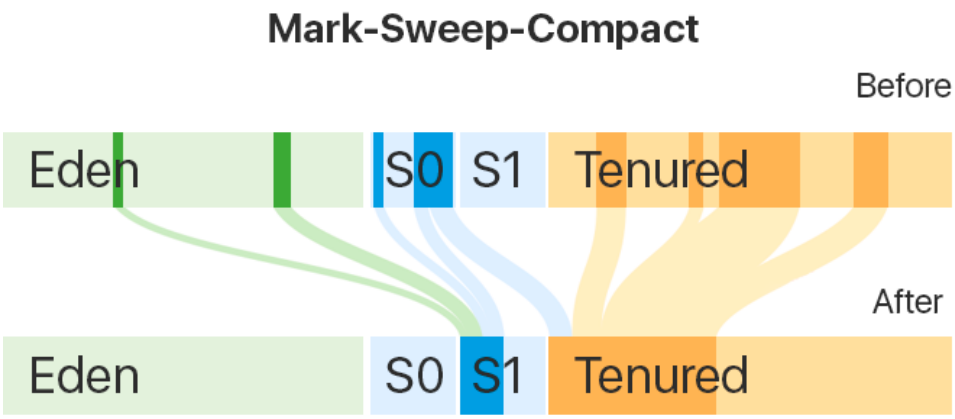
Что происходит, когда мы создаем в Java объект?

В первую очередь объект попадает в *Eden*. Если мы создали уже много объектов и в *Eden* уже нет места, срабатывает сборщик мусора и освобождает память. Это, так называемая, *малая сборка мусора* — на первом проходе он очищает область *Eden* и кладёт "выжившие" объекты в регион *Survivor 0*. Таким образом регион *Eden* полностью высвобождается.

Если произошло так, что область *Eden* снова была заполнена, garbage collector начинает работу с областью *Eden* и областью *Survivor 0*, которая занята на данный момент. После очищения выжившие объекты попадут в другой регион — *Survivor 1*, а два остальных останутся чистыми. При последующей сборке мусора в качестве региона назначения опять будет выбран *Survivor 0*. Именно поэтому важно, чтобы один из регионов *Survivor* всегда был пустым.

JVM следит за объектами, которые постоянно копируются и перемещаются из одного региона в другой. И для того, чтобы оптимизировать данный механизм, после определённого порога сборщик мусора перемещает такие объекты в регион *Tenured*.

Когда в *Tenured* места для новых объектов не хватает, происходит полная сборка мусора — **Mark-Sweep-Compact**.



Во время этого механизма определяется, какие объекты больше не используются, регион очищается от этих объектов, и область памяти *Tenured* дефрагментируется, т.е. последовательно заполняется нужными объектами.

Заключение

В данной статье мы разобрали базовые инструменты языка Java: JVM, JRE, JDK, принцип и этапы выполнения кода на JVM, компиляцию, организацию памяти, а также принцип работы сборщика мусора.

Теги: java, java development, bytecode, compilation, jdk, jre, jvm, сборщик мусора, инструменты разработчика

Хабы: Программирование, Java

+8 175 15k 7 Поделиться

12,0 Карма 8,0 Рейтинг

Подписаться

Виталий Сергей @VitaliSergey
Архитектор решений в IntexSoft

ПОХОЖИЕ ПУБЛИКАЦИИ

15 ноября 2017 в 12:10
Анонс Java-конференции JPoint 2018: JDK 9, высокие нагрузки и производительность JVM
+38 8,7k 18 27

4 сентября 2017 в 15:40
Обзор всех инструментов разработчика Chrome DevTools
+10 48,2k 223 1

28 марта 2017 в 13:07
IL2CPP: Интеграция сборщика мусора
+10 2,9k 27 0

Моя лента Все потоки Разработка Администрирование Дизайн Менеджмент Маркетинг Научпоп

Java-разработчик/ Backend Developer (Java)
от 180 000 до 240 000 ₽ • Crypterium • Москва • Можно удаленно

Java developer
от 150 000 до 180 000 ₽ • Granatum.solutions • Можно удаленно

Java разработчик
от 130 000 до 200 000 ₽ • ICL Services • Казань • Можно удаленно

Java-разработчик
от 100 000 до 180 000 ₽ • Crystal Service Integration • Санкт-Петербург

Java developer
от 140 000 до 200 000 ₽ • Эвотор • Москва

Больше вакансий на Хабр Карьере

Комментарии 7

Отслеживать новые в ☐ почте ☐ трекере

Neikist 16 октября 2019 в 17:56 0

Спасибо, было интересно, не планируется чего то подобного по андроиду? Насколько я понимаю d8, ART и Dalvik вносят в описанное коррективы.

Ответить

VitaliSergey 17 октября 2019 в 10:19 0

Рад, что материал оказалася полезным.
По андроиду статья также будет, следите за обновлениями.

Ответить

НЛО прилетело и опубликовало эту надпись здесь

Neikist 17 октября 2019 в 15:43 0

Кстати согласен. Сколько лет бейсику, лиспу? Да и не только им.

Ответить

VitaliSergey 18 октября 2019 в 09:56 0

Спасибо за внимательность, исправил.

Ответить

serjeant 17 октября 2019 в 15:11 0

Спасибо, отличная подача материала! Кратко, ёмко и по существу. Ждем новых статей

Ответить

VitaliSergey 18 октября 2019 в 10:00 0

Рад, что статья зашла. Новый материал уже в процессе

Ответить

САМОЕ ЧИТАЕМОЕ

Сутки | Неделя | Месяц

Что помешало экипажу Crew Dragon выйти из корабля?
+176 89,2k 50 120

Памятка для пострадавшего от слезоточивого газа/перцового баллона
+110 30,9k 194 86

Microsoft Defender начал помечать файл hosts как зловредный, если там блокируется сбор телеметрии Windows 10
+41 17,2k 25 88

Рынок соискателя или работодателя VS возрастная дискриминация

+50 17,7k 51 113

Разбор: вся изнанка онлайн-школ

Мегапост

Ваш аккаунт

Профиль
Трекер
Настройки

Разделы

Публикации
Новости
Хабы
Компании
Пользователи
Песочница

Информация

Устройство сайта
Для авторов
Для компаний
Документы
Соглашение
Конфиденциальность

Услуги

Реклама
Тарифы
Контент
Семинары
Мегапроекты
Мерч

Если нашли опечатку в посте, выделите ее и нажмите Ctrl+Enter, чтобы сообщить автору.

© 2006 – 2020 «Habr» Настройка языка О сайте Служба поддержки Мобильная версия