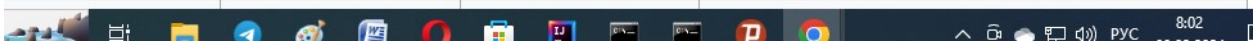
Кроссплатформенность (межплатформенность) — способность программного обеспечения работать с несколькими аппаратными платформами или операционными системами. Обеспечивается благодаря использованию высокоуровневых языков программирования, сред разработки и выполнения, поддерживающих условную компиляцию, компоновку и выполнение кода для различных платформ. Типичным примером является программное обеспечение, предназначенное для работы в операционных системах Linux и Windows одновременно.

Кроссплатформенными можно назвать большинство современных высокоуровневых языков программирования. Например, Си, C++, Free Pascal, FreeBASIC, PureBasic — **кроссплатформенные языки на уровне компиляции**, то есть для этих языков есть компиляторы под различные платформы. Это позволяет — при надлежащем качестве кода — не переписывать основной движок программы, меняются только особые системозависимые части.

Книга "Современные операционные системы" Таненбаум Э.

Платформа — это комплекс аппаратных и программных средств, на котором функционирует программное обеспечение пользователя ЭВМ. Основа аппаратной платформы (hardware-платформы) - процессор. Тип процессора определяет архитектуру аппаратных средств - аппаратную платформу, т. е. тип и характеристики компьютера. (https://studfile.net/preview/4235646/)

Архитектура	Целочисленных регистров	FP-регистров	Примечания
x86-32	8	8	
x86-64	16	16	
IBM System/360	16	4	
z/Architecture	16	16	
Itanium	128	128	
SPARC	31	32	Регистр 0 (глобальный) всегда запущен
IBM Cell	4~16	1~4	
IBM POWER	32	32	
Power Architecture	32	32	
Alpha	32	32	
6502	3	0	
W65C816S	5	0	
PIC	1	0	
AVR	32	0	
ARM 32-bit[4]	16	различное	
ARM 64-bit[5]	31	32	
MIPS	31	32	Регистр 0 всегда занулён



Программная платформа - программная среда, которая используется для написания приложений и их запуска. Он включает в себя программные инструменты, такие как конструкторы графического интерфейса пользователя, компиляторы, библиотеки классов и утилиты для разработки приложений, а также механизм выполнения для выполнения приложений, поскольку они не могут работать сами по себе.

С другой стороны, различия в API различных операционных систем существенно затрудняют перенос приложений между платформами. Существуют различные методы обхода этой сложности — написание «промежуточных» API (API графических интерфейсов wxWidgets, GTK и т. п.), написание библиотек, которые отображают системные вызовы одной ОС в системные вызовы другой ОС (такие среды исполнения, как Wine, cygwin и т. п.), введение стандартов кодирования в языках программирования (например, стандартная библиотека языка C), написание интерпретируемых языков, реализуемых на разных платформах (sh, Python, Perl, PHP, Tcl, JavaScript, Ruby и т. д.).

Начало GCC было положено Ричардом Столлманом, который реализовал первый вариант GCC в 1985 году на нестандартном и непереносимом диалекте языка Паскаль; позднее компилятор был переписан на языке Си Леонардом Тауэром и Ричардом Столлманом[5] и выпущен в 1987 году[6] как компилятор для проекта GNU, который сам по себе являлся свободным программным обеспечением. Разработка GCC курируется Free Software Foundation[7].

В настоящее время GCC поддерживается группой программистов со всего мира. GCC является лидером по количеству процессоров и операционных систем, которые он поддерживает.

Будучи официальным компилятором системы GNU, GCC также является главным компилятором для сборки ряда других операционных систем; среди них — различные варианты Linux и BSD (ранее, в настоящее время используется Clang LLVM), а также ReactOS, macOS, OpenSolaris, NeXTSTEP, BeOS и Haiku.

GCC часто выбирается для разработки программного обеспечения, которое должно работать на большом числе различных аппаратных платформ. Различия между «родными» для каждой из аппаратных платформ компиляторами приводят к трудностям при разработке кода, который бы корректно компилировался разными компиляторами, а кроме того, при использовании различных компиляторов сильно усложняются сборочные скрипты, которые должны собирать ПО для всех аппаратных платформ. При использовании GCC для компиляции кода под разные платформы будет использован один и тот же синтаксический анализатор. Поэтому, если удалось собрать программу для одной из целевых платформ, то велика вероятность, что программа нормально соберётся и для других платформ.

Языки

Стандартный компилятор включает в себя front-end'ы для языков:

- Ada (GCC для Ada, или GNAT),
- Си,
- C++ (GCC для C++, или G++),
- Фортран (GCC для Fortran, или gfortran),
- Java (GCC для Java, или GCJ, исключена из состава GCC начиная с версии 7[8]),
- Objective-C (GCC для Objective-C, или gobjc),
- Objective-C++ (GCC для Objective-C++, или gobjc++),
- Go (GCC для Go, или gccgo) (с версии 4.6[9]).
- D (GCC для D, или GDC[10], начиная с версии 9.1[11])

Front-end для CHILL был добавлен ранее, но из-за недостаточной поддержки был исключён из набора. До выхода версии 4.0 front-end'ом для Fortran был G77, который поддерживал лишь FORTRAN 77. В новых версиях G77 был исключён в пользу нового GFortran front-end, который поддерживает Fortran 95.

Также существуют сторонние front-end'ы для Pascal, Modula-2, Modula-3, Mercury, VHDL и PL/I.

Архитектуры[править | править код]

Список поддерживаемых GCC (для версии 7.1) процессоров включает в себя

- Alpha
- ARM
- Atmel AVR
- Blackfin
- HC12
- H8/300
- x86 (IA-32 и x86-64)
- IA-64 («Itanium»)
- m68k

- Motorola 88000
- MIPS
- Texas Instruments MSP430
- PA-RISC
- PDP-11
- PowerPC
- RISC-V
- R8C/M16C/M32C
- SPU в Cell
- System/370, System/390
- SuperH
- SPARC
- VAX

Менее известные процессоры, поддерживаемые в стандартном релизе:

- A29K
- ARC
- ETRAX CRIS
- D30V
- DSP16xx
- FR-30
- FR-V
- Intel i960
- IP2000
- M32R
- 68HC11
- MCORE
- MMIX
- MN10200
- MN10300
- Motorola 88000
- NS32K
- ROMP
- Stormy16
- V850
- Xtensa
- AVR32

Дополнительные типы архитектур и процессоров, которые поддерживаются версиями GCC, но поддержкой которых занимаются сторонние организации (не Фонд свободного программного обеспечения):

- D10V
- MeP
- MicroBlaze
- TI MSP430
- TI C6X[12]
- Nios II и Nios
- PDP-10
- TIGCC (вариация Motorola 68000)
- Z8000
- PIC24/dsPIC
- OpenRISC 1000

Целью разработчиков процессора было достигнуть пиковой производительности в 250 Gflops[7].

Кристалл микропроцессора спроектирован по технологии 28 нм, имеет 8 процессорных ядер с улучшенной 64-разрядной архитектурой «Эльбрус» 3-го поколения, кэш-память 2-го уровня общим объёмом 4 мегабайта и 3-го уровня объёмом 16 мегабайт.

- Основная особенность линейки отечественных процессоров «Эльбрус» — заложенный в архитектуру принцип явного параллелизма операций, он даёт возможность выполнять на каждом ядре за один машинный такт до 25 операций неупакованных 32- и 64-разрядных данных и до 41 операции в векторном режиме (упакованные 32-разрядные данные)[8], что обеспечивает высокую производительность при умеренной тактовой частоте;
- технология динамической двоичной трансляции, позволяющая обеспечивать эффективное исполнение приложений и операционных систем, распространяемых в двоичных кодах x86;
- поддержка режима защищённых вычислений с особым аппаратным контролем целостности структуры памяти, которая позволяет обеспечить высокий уровень информационной безопасности использующих его программных систем.

Процессор стал частью политики российского правительства по импортозамещению.[9][7]. Базовой операционной системой для платформы Эльбрус является ОС Эльбрус, построенная на базе ядра Linux. Система программирования платформы поддерживает **языки C, C++, Java, Фортран-77, Фортран-90**[10]. Также на платформе могут работать ОС ALT Linux[11], ОСРВ QNX Нейтрино[12], ЛИНТЕР [13].

Как заявляется в компании: «Помимо создания традиционных вычислительных комплексов, прорабатываются и более масштабные проекты. В частности, производительность серверов на базе „Эльбрус-8С“ позволит в ближайшей перспективе приступить к практическому построению на их основе суперкомпьютера».

Не менее важны для кроссплатформенности **стандартизованные библиотеки среды выполнения**. В частности, стандартом стала библиотека языка Си (POSIX).

POSIX (англ. ***P**ortable **O**perating **S**ystem **I**nterface* — переносимый интерфейс операционных систем) — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный API), библиотеку языка C и набор приложений и их интерфейсов. Стандарт создан для обеспечения совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода, но может быть использован и для не-Unix систем.

Идея POSIX заключается в том, что разработчик должен создать приложение и заставить его работать в любой системе, соответствующей стандарту.

Стандарт POSIX, который затронет большинство конечных пользователей, - это POSIX.2, который регулирует поведение оболочки и различных стандартных служебных программ.

В POSIX-совместимой системе все параметры должны быть одинаковыми, независимо от того, какой вариант операционной системы вы используете.

Другие стандарты POSIX в основном представляют интерес для программистов. Они включают в себя все, от потоков до стандартной библиотеки C.

Из крупных кроссплатформенных библиотек — Qt, GTK+, FLTK, STL, Boost, OpenGL, SDL, OpenAL, OpenCL.

Qt — это библиотека классов C++ и набор инструментального программного обеспечения для создания кросс-платформенных приложений с графическим интерфейсом (GUI). Существуют вариации для других языков: PyQt для Python, QtRuby для Ruby, Qt Jambi для Java.

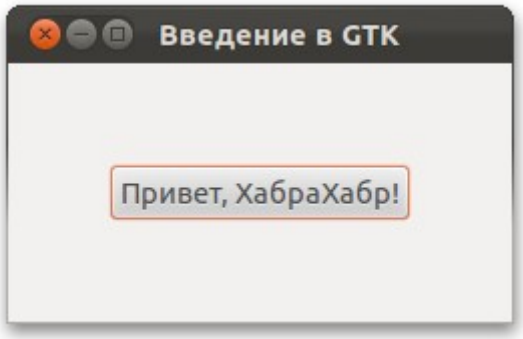
Разработчики на C++, которые создают десктопные и мобильные приложения.

Разработчики ПО для интернета вещей или для микрокомпьютеров.

Специалисты в области специализированного оборудования: embedded-систем, промышленных роботов и другой сложной автоматики.

GTK+ — это фреймворк для создания кроссплатформенного графического интерфейса пользователя (GUI). Наряду с Qt он является одной из двух наиболее популярных на сегодняшний день библиотек для X Window System.

Изначально эта библиотека была частью графического редактора GIMP, но позже стала независимой и приобрела популярность. GTK+ — это свободное ПО, распространяемое на условиях GNU LGPL и позволяющее создавать как свободное, так и проприетарное программное обеспечение.



Как это работает

GTK+ написан на языке Си, однако несмотря на это, является объектно-ориентированным. Также можно использовать обёртки для следующих языков: Ada, C, C++, C#, D, Erlang, Fortran, GOB, Genie, Haskell, FreeBASIC, Free Pascal, Java, JavaScript, Lua, OCaml, Perl, PHP, PureBasic, Python, R, Ruby, Smalltalk, Tcl, Vala.

Внутри GTK+ состоит из двух компонентов: GTK, который содержит набор виджетов (кнопка, метка и т.д.) и GDK, который занят выводом результата на экран.

Внешний вид приложений может меняться программистом и/или пользователем. По-умолчанию приложения выглядят нативно, т.е. так же, как и другие приложение в этой системе. Кроме того, начиная с версии 3.0, можно менять внешний вид элементов с помощью CSS.

Делаем «Hello, World»

Для начала за основу возьмём вот такую заготовку:

```
#include <gtk/gtk.h> /* подключаем GTK+ */

/* эта функция будет выполнена первой */
int main( int argc, char *argv[])
{
        /* тут мы объявим переменные */

        /* запускаем GTK+ */
        gtk_init(&argc, &argv);

        /* тут будет код нашего приложения */

        /* передаём управление GTK+ */
        gtk_main();

        return 0;
}
```

Пожалуйста, не используйте одинарные комментарии (//), если как и я решили писать на Си.

Давайте для начала создадим окно нашего приложения. В GTK существует несколько типов окошек, но нам понадобится обычный GtkWindow. Вставьте в нашу заготовку следующий код:

```
/* это вставьте вначале, хотя на самом деле порядок не так важен */
GtkWidget *window;

/* ... */

/* создать новый виджет - окно */
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
/* дать окну заголовок */
gtk_window_set_title(GTK_WINDOW(window), "Введение в GTK");
/* установить внутренние границы (читайте ниже) */
gtk_container_set_border_width (GTK_CONTAINER(window), 50);
```

Прежде чем продолжить, несколько слов об упаковке.

В GTK+ виджеты принято размещать не по координатам, а упаковывать в специальные контейнеры. Окно, которое мы создали, также является контейнером, который может содержать только **один** виджет. Нам этого хватит, но в ином случае нам бы пришлось добавить специальный виджет-контейнер, который может представлять из себя сетку, а также горизонтальные или вертикальные поля. Но об этом я подробнее расскажу в следующем топике.

Пока остановимся на окне. Напомню, что окно — это контейнер, которому мы указали толщину границ в 50 пикселей. Что это значит?

Это значит, что мы создаём своего рода невидимую рамку вокруг этого контейнера и по этому ничего не сможем разместить в этой области.

Сферы применения

В основном Qt используют для создания очень быстрых и высокопроизводительных приложений. Это мессенджеры, игры или сложные ресурсоемкие программы. Он популярен в сферах, имеющих повышенные требования к безопасности ПО. Среди них:

промышленность и транспортная отрасль. На C++ и на Qt пишут программы для роботов, которые используются на производствах, в перевозке грузов и других похожих отраслях. Qt используют при написании программного обеспечения для автомобилей, кораблей и других видов транспорта;

MedTech. Фреймворк применяют при создании программных систем и интерфейсов для медицинского оборудования;

IoT. На C++ с Qt пишут логику для «умных» приборов, которые подключаются к интернету вещей.

С использованием Qt написаны мессенджер Telegram, продукты Autodesk, окружение рабочего стола для многих систем под ядром Linux и пр.

Модули Qt

Тут перечислена часть основных модулей — блоков программного кода библиотеки Qt. В них содержатся классы и функции для создания приложений и работы с данными.

QtCore — ядро фреймворка.

QtGUI — компоненты для создания интерфейсов.

QtNetwork — функции для работы с сетевыми соединениями.

QtSql — компоненты для работы с базами данных на основе SQL.

QtWidgets — модуль для работы с виджетами.

QtXml — компоненты для обработки XML, специального формата хранения файлов.

QtXmlPatterns — инструменты для работы с языками, которые обрабатывают данные XML и организуют к нему доступ.

QtScript — классы внутреннего скриптового языка Qt Scripts.

QtOpenGL — инструменты для работы с библиотеками, написанными по спецификации OpenGL.

QtSvg — компоненты для обработки векторной графики.

QtMultimedia — инструменты для работы с мультимедиа-файлами.

QtWebEngine — ядро браузера Chromium, адаптированное под Qt.

QtTest — компоненты для тестирования приложений.

Qt3Support — поддержка старых версий фреймворка.

QtCLucene — инструменты для автоматического поиска.

Что входит в Qt

Кроме библиотеки и ее модулей, Qt содержит дополнительное ПО, утилиты, справочники и внутренние языки.

Qt Creator. Это IDE, среда программирования. Внутри Qt Creator можно писать, компилировать и запускать код, тестировать его и выполнять отладку. Среда работает в Windows, Linux и macOS.

Qt Assistant. Большой справочник и библиотека документации. Он добавляет в среду возможность открывать и читать документы, сохраненные в QCH — внутреннем формате Qt для справочных документов. Ассистент позволяет быстро разобраться в работе нужного модуля.

QT Linguist. Инструмент помогает быстрее локализовать приложение на разных языках. Используется при создании программ, которые рассчитаны на мультиязычную аудиторию.

Qt Designer. Инструмент позволяет быстро создавать графические интерфейсы (GUI). Он поставляется вместе с фреймворком и подходит для разработки приложений, где большую роль играют визуальные компоненты. Интерфейс создается внутри инструмента с помощью C++, сохраняется в файл и подключается к проекту, написанному на Qt.

Qt Quick. Еще один инструмент для интерфейсов. Он отличается от предыдущего: GUI создается не на C++, а с использованием специального языка QML. Отличается и стиль описания компонентов. Qt Quick предназначен для быстрого и простого создания интерфейсов. Его часто применяют при разработке мобильных приложений и игр.

QML. Это язык для создания интерфейсов от команды Qt. Он основан на среде JavaScript и помогает быстро описывать графические интерфейсы. В Qt реализована полная поддержка QML, а сам язык встроен в инструмент Qt Quick.

Преимущества Qt

Кросс-платформенность. Qt — кросс-платформенный фреймворк. Это значит, что он существует для всех популярных операционных систем: Windows, Linux, iOS и Android. Фреймворк используют при разработке под любые устройства: от микроконтроллеров до суперкомпьютеров.

Высокая скорость. Программы на C++ быстро обрабатываются и запускаются. Также C++ — компилируемый язык программирования. Это значит, что компилятор транслирует исходный код на C++ в исполняемый файл, который содержит набор машинных инструкций, что тоже влияет на скорость.

Удобная среда разработки. Qt Creator — среда, в которой легко разобраться. В ней есть все необходимое, важные компоненты находятся под рукой, а сам инструмент интуитивно понятен. В нем удобно организована отладка, поэтому разработчику легче находить проблемные участки кода.

Быстрое создание GUI. Дополнительные инструменты помогают быстро спроектировать интерфейс и разработать дизайн. Благодаря Qt Creator и его возможностям фреймворк отлично подходит для создания приложений с упором на графический интерфейс.

Взаимодействие процессов. Благодаря метаобъектной системе Qt может более гибко управлять межпроцессным взаимодействием, чем «чистый» C++. Сейчас это преимущество не так актуально, потому что появились версии C++ 11 и выше. Но много проектов пользуется легаси-кодом, написанным на старых версиях языка. Там особенности взаимодействия все так же важны.

Документация. На официальном сайте представлена подробная документация, которая поможет разобраться с особенностями работы с Qt.

Недостатки Qt

Сложности с лицензией. У Qt тройное лицензирование. Существуют три варианта библиотеки, каждый из них — под своей лицензией. Один предназначен для коммерческой разработки, второй — для проектов с открытым исходным кодом, третий — для собственнических проектов. Для коммерческих проектов нельзя использовать бесплатную версию.

Большой вес приложений. Qt добавляет много новых сущностей, все они занимают место. Итоговый проект быстро работает, но много весит. Для десктопных приложений это не так критично, как для мобильных.

Обратная совместимость. Из-за обратной совместимости со старыми версиями разработчики поддерживают в том числе неоптимальные решения.

Сложность. C++ — сложный язык. В нем много абстракций, он не интуитивно понятный. Новичкам бывает трудно в нем разобраться.

Распространенность. В основном C++ используется только там, где нужна высокая скорость работы. Это программное обеспечение для сложных промышленных систем, медицинской техники, автомобилей, роботов. Для пользовательских приложений язык применяется реже, чем раньше. Поэтому Qt встречается не так часто, как другие фреймворки.

Как начать работу с Qt

Бесплатную версию фреймворка можно скачать на официальном сайте. Небольшая программа-инсталлятор Qt Installer подгрузит и установит необходимые компоненты. Можно воспользоваться бесплатной версией либо приобрести лицензию. Она подходит для крупных коммерческих проектов. Для тестирования инструмента достаточно триальной версии (полная версия Qt со сроком действия 10 дней).

FLTK - пишем маленькие кросс-платформенные приложения с олдскульным интерфейсом

Иногда требуется написать кросс-платформенное приложение с небольшим размером. Для управления микроконтроллером, например. Или рисования иконок для дисплеев. Но использовать кросс-платформенные библиотеки вроде Qt, WxWidgets не имеет смысла - весят они ну очень много. Неудобно получается, когда приложение весит 100кб, а графическая библиотека для него – под 30Мб.

На помощь к нам приходит FLTK – Fast, Light Toolkit.

Сейчас всё можно сделать проще!

Обновлённая статья - FLTK - упрощаем себе жизнь с msys2.
Библиотека распространяется в виде исходного кода и скачивается с сайта fltk.org. Значит, сейчас будем её собирать.

Boost — собрание библиотек классов, использующих функциональность языка C++ и предоставляющих удобный кроссплатформенный высокоуровневый интерфейс для лаконичного кодирования различных повседневных подзадач программирования (работа с данными, алгоритмами, файлами, потоками, регулярными выражениями, линейная алгебра, генерация псевдослучайных чисел, обработка изображений, модульное тестирование и т. п.). Версия 1.76 содержит 164 отдельные библиотеки.[2]

Свободно распространяются по лицензии Boost Software License, разработанной для того, чтобы Boost можно было использовать как со свободным, так и с проприетарными программными проектами, вместе с исходным кодом.[3] Проект был создан после принятия стандарта C++, когда многие были недовольны отсутствием некоторых библиотек в STL. Проект является своего рода «испытательным полигоном» для различных расширений языка и части библиотек[4], которые являются кандидатами на включение в следующий стандарт C++. Многие из основателей Boost входят в комитет по стандартизации C++, и несколько библиотек Boost были приняты для включения в C++ Technical Report 1, стандарт C++11 (например, интеллектуальные указатели, потоки, регулярные выражения, random, ratio, tuple) и стандарт C++17 (например, filesystem, any, optional, variant, string_view). **Boost** имеет заметную направленность на исследования и расширяемость (метапрограммирование и обобщённое программирование с активным использованием шаблонов).

- Алгоритмы
- Обход ошибок в компиляторах, не соответствующих стандарту
- Многопоточное программирование
- Контейнеры
- Юнит-тестирование
- Структуры данных
- Функциональные объекты
- Обобщённое программирование
- Графы
- Работа с геометрическими данными
- Ввод-вывод
- Межъязыковая поддержка
- Итераторы
- Математические и числовые алгоритмы
- Работа с памятью
- Синтаксический и лексический разбор
- Метапрограммирование на основе препроцессора
- «Умные указатели»
- Обработка строк и текста
- Метапрограммирование на основе шаблонов

OpenGL является кроссплатформенным, независимым от языка программирования API для работы с графикой. OpenGL — низкоуровневый API, поэтому для работы с ним неплохо иметь некоторое представление о графике в целом и знать основы линейной алгебры.

Знакомство с OpenGL нужно начать с того, что OpenGL — это *спецификация*. Т.е. OpenGL лишь определяет набор обязательных возможностей. Реализация же зависит от конкретной платформы.

OpenGL - это графический стандарт в области компьютерной графики. На данный момент он является одним из самых популярных графических стандартов во всём мире. Ещё в 1982 г. в Стенфордском университете была разработана концепция графической машины, на основе которой фирма Silicon Graphics в своей рабочей станции Silicon IRIS реализовала конвейер рендеринга. Таким образом была разработана графическая библиотека IRIS GL. На основе библиотеки IRIS GL, в 1992 году был разработан и утверждён графический стандарт OpenGL. Разработчики OpenGL - это крупнейшие фирмы разработчики как оборудования так и программного обеспечения: Silicon Graphics, Inc., Microsoft, IBM Corporation, Sun Microsystems, Inc., Digital Equipment Corporation (DEC), Evans & Sutherland, Hewlett-Packard Corporation, Intel Corporation и Intergraph Corporation.

OpenGL переводится как Открытая Графическая Библиотека (Open Graphics Library), это означает, что OpenGL - это открытый и мобильный стандарт. Программы, написанные с помощью OpenGL можно переносить практически на любые платформы, получая при этом одинаковый результат, будь это графическая станция или суперкомпьютер. OpenGL освобождает программиста от написания программ для конкретного оборудования. Если устройство поддерживает какую-то функцию, то эта функция выполняется аппаратно, если нет, то библиотека выполняет её программно.

Что же представляет из себя OpenGL? С точки зрения программиста OpenGL - это программный интерфейс для графических устройств, таких как графические ускорители. Он включает в себя около 150 различных команд, с помощью которых программист может определять различные объекты и производить рендеринг. Говоря более простым языком, вы определяете объекты, задаёте их местоположение в трёхмерном пространстве, определяете другие параметры (поворот, масштаб, ...), задаёте свойства объектов (цвет, текстура, материал, ...), положение наблюдателя, а библиотека OpenGL позаботится о том чтобы отобразить всё это на экране. Поэтому можно сказать, что библиотека OpenGL является только воспроизводящей (Rendering), и занимается только отображением 3Д обьектов, она не работает с устройствами ввода (клавиатуры, мыши). Также она не поддерживает менеджер окон.

OpenGL имеет хорошо продуманную внутреннюю структуру и довольно простой процедурный интерфейс. Несмотря на это с помощью OpenGL можно создавать сложные и мощные программные комплексы, затрачивая при этом минимальное время по сравнению с другими графическими библиотеками.

В некоторых библиотеках OpenGL (например под X Windows) имеется возможность изображать результат не только на локальной машине, но также и по сети. Приложение, которое вырабатывает команды OpenGL называется клиентом, а приложение, которое получает эти команды и отображает результат - сервером. Таким образом можно строить очень мощные

воспроизводящие комплексы на основе нескольких рабочих станций или серверов, соединённых сетью.

Основные возможности OpenGL.

Что предоставляет библиотека в распоряжение программиста? Основные возможности:

- **Геометрические и растровые примитивы.** На основе геометрических и растровых примитивов строятся все объекты. Из геометрических примитивов библиотека предоставляет: точки, линии, полигоны. Из растровых: битовый массив(bitmap) и образ(image)
- **Использование B-сплайнов.** B-сплайны используются для рисования кривых по опорным точкам.
- **Видовые и модельные преобразования.** С помощью этих преобразований можно располагать обьекты в пространстве, вращать их, изменять форму, а также изменять положение камеры из которой ведётся наблюдение.
- **Работа с цветом.** OpenGL предоставляет программисту возможность работы с цветом в режиме RGBA (красный-зелёный-синий-альфа) или используя индексный режим, где цвет выбирается из палитры.
- **Удаление невидимых линий и поверхностей. Z-буферизация.**
- **Двойная буферизация.** OpenGL предоставляет как одинарную так и двойную буферизацию. Двойная буферизация используется для того, чтобы устранить мерцание при мультипликации, т.е. изображение каждого кадра сначала рисуется во втором(невидимом) буфере, а потом, когда кадр полностью нарисован, весь буфер отображается на экране.
- **Наложение текстуры.** Позволяет придавать объектам реалистичность. На объект, например шар, накладывается текстура(просто какое-то изображение), в результате чего наш объект теперь выглядит не просто как шар, а как разноцветный мячик.
- **Сглаживание.** Сглаживание позволяет скрыть ступенчатость, свойственную растровым дисплеям. Сглаживание изменяет интенсивность и цвет пикселей около линии, при этом линия смотрится на экране без всяких зигзагов.
- **Освещение**. Позволяет задавать источники света, их расположение, интенсивность, и т.д.
- **Атмосферные эффекты.** Например туман, дым. Всё это также позволяет придать объектам или сцене реалистичность, а также "почувствовать" глубину сцены.
- **Прозрачность объектов.**
- **Использование списков изображений.**

Кроссплатформенные среды исполнения

PHP, Perl, Python, Tcl и Ruby — кроссплатформенные интерпретируемые языки, их интерпретаторы существуют для многих платформ.

Среды исполнения ActionScript Virtual Machine, Java Virtual Machine и .NET также кроссплатформенны, однако на их вход подаётся не исходный текст, а промежуточный код. Поэтому программы, написанные на ActionScript, Java и C#, можно запускать под разными операционными системами без предварительной перекомпиляции.

Ошибки верификации байт-кода стековых машин приводили к появлению множества экстремально опасных уязвимостей, в частности десятков в виртуальной машине AVM2, используемой в Adobe Flash для исполнения скриптов ActionScript[14][15][16] и нескольких в ранних популярных системах исполнения Java (JVM)[17][18]

В конце 2000-х — начале 2010-х авторы компиляторов V8 (для языка JavaScript, часто реализуемого через байт-код)[19] и Dart[20] усомнились в том, что промежуточные байт-коды обязательны для быстрых и эффективных виртуальных машин. В этих проектах была реализована непосредственная JIT-компиляция (компиляция во время исполнения) из исходных кодов сразу в машинный код.[21]

V8 — движок JavaScript с открытым исходным кодом, распространяемый по лицензии BSD. Разработан датским отделением компании Google.

Ларс Бак считал[12], что краеугольными камнями V8 являются:

- Компиляция исходного кода JavaScript непосредственно в собственный машинный код, минуя стадию промежуточного байт-кода.
- Эффективная система управления памятью, приводящая к быстрому объектному выделению и маленьким паузам сборки «мусора»[13].
  - V8 приостанавливает исполнение кода во время выполнения сборки «мусора».
  - Уменьшает влияние и воздействие приостановки приложения при сборке «мусора».
  - V8 может точно определять, где находятся в памяти объекты и указатели, что позволяет избежать утечки памяти при ложной идентификации объектов в качестве указателей.
- Введение скрытых классов и встроенных кэшей, ускоряющих доступ к свойствам и вызовам функций.

V8 исполняет JavaScript-сценарии в особых «контекстах», которые по сути являются отдельными виртуальными машинами. Правда в одном процессе может работать только одна виртуальная машина, несмотря на возможность использования нескольких потоков[14]. В Chromium это обходится мультипроцессовой архитектурой, повышающей также стабильность и безопасность, реализуя таким образом механизм «песочницы»[15]. Таким образом, несмотря на динамическую природу JavaScript, разработчикам удалось применить методы,

характерные для реализации классических объектно-ориентированных языков, такие как компиляция кода «на лету», внутреннее кэширование, точный процесс сборки мусора, снэпшоттинг при создании контекстов[9][14].

Браузеры **Chromium — веб-браузер с открытым исходным кодом, на основе которого создаётся ряд браузеров, наиболее популярным из которых является Chrome — веб-браузер компании Google**

- Maxthon — веб-браузер со встроенным блокиратором рекламы, использующий два движка рендеринга: WebKit и Trident[21];
- Браузер Android[22] — мобильный браузер, входящий в Android OS.

Операционные системы[

Android — операционная система от Google, предназначенная для коммуникаторов, нетбуков и планшетов, V8 используется начиная с Android Froyo.

- HP webOS — операционная система от Hewlett-Packard для коммуникаторов, нетбуков и планшетов, движок V8 используется во встроенном браузере.
- Google Chrome OS — операционная система от Google на базе проекта Chromium, ориентированная на облачные сервисы, движок является важным компонентом всей операционной системы.
  - **ActionScript** — объектно-ориентированный язык программирования, один из диалектов ECMAScript, который добавляет интерактивность, обработку данных и многое другое в содержимое Flash-приложений. ActionScript исполняется виртуальной машиной (ActionScript Virtual Machine), которая является составной частью Flash Player. ActionScript компилируется в байт-код, который включается в SWF-файл.
  - SWF-файлы исполняются Flash Player'ом

В этом посте «виртуальная машина» относится к виртуальным машинам процессов, а не к системным виртуальным машинам, таким как Qemu или Virtualbox. Виртуальная машина процесса — это просто программа, предоставляющая общую среду программирования, программа, которую можно запрограммировать.

В Java есть не только виртуальная машина, но и интерпретатор, а в Python есть не только виртуальная машина, но и интерпретатор. Причина, по которой «виртуальная машина» является более распространенным термином в Java, а «интерпретатор» — более распространенным термином в Python, во многом связана с основным различием между двумя языками: статической типизацией (Java) и динамической типизацией (Python). В этом контексте «тип» относится к примитивным типам данных — типам, которые предполагают размер хранения данных в памяти. С виртуальной машиной Java это легко. Это требует, чтобы программист указал примитивный тип данных каждой переменной. Это обеспечивает достаточную информацию для того, чтобы байт-код Java не только мог интерпретироваться и выполняться виртуальной машиной Java, но даже компилироваться в машинные инструкции . Виртуальная машина Python более сложна в том смысле, что она берет на себя дополнительную задачу по приостановке перед выполнением каждой операции для определения примитивных типов данных для каждой переменной или структуры данных, участвующих в

операции. Python освобождает программиста от мышления в терминах примитивных типов данных и позволяет выражать операции на более высоком уровне. Цена этой свободы — производительность. «Интерпретатор» — предпочтительный термин для Python, поскольку ему приходится делать паузу для проверки типов данных, а также потому, что сравнительно краткий синтаксис динамически типизированных языков хорошо подходит для интерактивных интерфейсов. Технических препятствий для создания интерактивного интерфейса Java нет, но попытка написать любой статически типизированный код в интерактивном режиме была бы утомительной, поэтому так не делается.
В мире Java виртуальная машина затмевает всех, потому что она запускает программы, написанные на языке, который фактически может быть скомпилирован в машинные инструкции, и в результате достигается скорость и эффективность использования ресурсов. Байт-код Java может выполняться виртуальной машиной Java с производительностью, приближающейся, условно говоря, к производительности скомпилированных программ. Это связано с наличием информации о примитивном типе данных в байт-коде. Виртуальная машина Java выделяет Java в отдельную категорию:

переносимый интерпретируемый статически типизированный язык
Следующий ближайший объект — LLVM, но LLVM работает на другом уровне:

переносимый интерпретируемый язык ассемблера
Термин «байт-код» используется как в Java, так и в Python, но не весь байт-код одинаковый. байт-код — это всего лишь общий термин для промежуточных языков, используемых компиляторами/интерпретаторами. Даже компиляторы C, такие как gcc, используют промежуточный язык (или несколько) для выполнения своей работы. Байт-код Java содержит информацию о примитивных типах данных, тогда как байт-код Python этого не делает. В этом отношении виртуальная машина Python (а также Bash, Perl, Ruby и т. д.) действительно существенно медленнее, чем виртуальная машина Java, или, скорее, у нее просто больше работы. Полезно рассмотреть, какая информация содержится в разных форматах байт-кода:

- **llvm:** регистры процессора
- **Java:** примитивные типы данных
- **Python:** определяемые пользователем типы

Проведем аналогию из реального мира: LLVM работает с атомами, виртуальная машина Java работает с молекулами, а виртуальная машина Python работает с материалами. Поскольку все должно в конечном итоге разложиться на субатомные частицы (реальные машинные операции), перед виртуальной машиной Python стоит самая сложная задача.

Интерпретаторы/компиляторы статически типизированных языков просто не имеют того багажа, который есть у интерпретаторов/компиляторов динамически типизированных языков. Программистам статически типизированных языков приходится брать на себя слабину, за что платой становится производительность. Однако точно так же, как все недетерминированные функции являются тайно детерминированными, все динамически типизированные языки тайно типизированы статически. Таким образом, различия в производительности между

двумя языковыми семействами должны нивелироваться примерно в то время, когда Python меняет свое название на HAL 9000.

Виртуальные машины динамических языков, таких как Python, реализуют некую идеализированную логическую машину и не обязательно соответствуют какому-либо реальному физическому оборудованию. Виртуальная машина Java, напротив, по функциональности больше похожа на классический компилятор C, за исключением того, что вместо выдачи машинных инструкций она выполняет встроенные процедуры. В Python целое число — это объект Python с набором прикрепленных к нему атрибутов и методов. В Java int — это определенное количество бит, обычно 32. Это не совсем корректное сравнение. Целые числа Python действительно следует сравнивать с классом Java Integer. Примитивный тип данных Java «int» нельзя сравнивать ни с чем в языке Python, потому что в языке Python просто отсутствует этот уровень примитивов, как и байт-код Python.

Поскольку переменные Java явно типизированы, можно разумно ожидать, что что-то вроде производительности Jython будет на том же уровне, что и cPython . С другой стороны, виртуальная машина Java, реализованная на Python, почти гарантированно будет медленнее грязи. И не ждите, что Ruby, Perl и т. д. будут работать лучше. Они не были созданы для этого. Они были разработаны для «сценариев», как это называется программированием на динамическом языке.
Каждая операция, выполняемая на виртуальной машине, в конечном итоге должна выполняться на реальном оборудовании. Виртуальные машины содержат предварительно скомпилированные процедуры, которые являются достаточно общими для выполнения любой комбинации логических операций. Виртуальная машина, возможно, и не генерирует новые машинные инструкции, но она определенно выполняет свои собственные процедуры снова и снова в произвольно сложных последовательностях. Виртуальная машина Java, виртуальная машина Python и все другие виртуальные машины общего назначения равны в том смысле, что их можно заставить выполнять любую логику, которую вы можете придумать, но они различаются с точки зрения того, какие задачи они выполняют. берут на себя и какие задачи оставляют программисту.

Psyco for Python — это не полноценная виртуальная машина Python, а JIT-компилятор, который перехватывает обычную виртуальную машину Python в тех точках, где, по ее мнению, она может скомпилировать несколько строк кода — в основном это циклы, где он думает, что примитивный тип некоторых переменная останется постоянной, даже если значение меняется с каждой итерацией. В этом случае можно отказаться от некоторых процедур непрерывного определения типа обычной виртуальной машины. Однако нужно быть немного осторожным, чтобы не вырвать шрифт из-под ног Псико. Pysco, однако, обычно знает, что нужно просто вернуться к обычной виртуальной машине, если он не полностью уверен, что тип не изменится.

Мораль этой истории в том, что информация о примитивных типах данных действительно полезна для компилятора/виртуальной машины.

Наконец, чтобы представить все это в перспективе, рассмотрим следующее: программа Python, выполняемая интерпретатором/виртуальной машиной

Python, реализованной на Java, работающей на интерпретаторе/виртуальной машине Java, реализованной в LLVM, работающей на виртуальной машине qemu, работающей на iPhone.

T-компиляция – это метод повышения производительности интерпретируемых программ. JIT расшифровывается как Just-in-time. Во время выполнения программа может быть скомпилирована в машинный код для повышения ее производительности. Также этот метод известен как динамическая компиляция.

Динамическая компиляция имеет несколько преимуществ перед статической. При запуске приложений на JAVA или C# среда выполнения может профилировать приложение во время его исполнения. Это позволяет создавать более оптимизированный код. Если поведение приложения меняется во время его исполнения, то среда выполнения может перекомпилировать код.

Есть некоторые недостатки, заключающиеся в задержках при запуске или непроизводительных издержках при компиляции во время выполнения. Чтобы ограничить эти издержки, многие JIT-компиляторы компилируют только пути кода, которые часто используются.

Обзор

Традиционно существует два метода преобразования исходного кода в форму, которую можно запустить на платформе. Статистическая компиляция преобразует код в язык для конкретной платформы. Интерпретатор непосредственно выполняет исходный код.

JIT-компиляция пытается использовать преимущества обоих. В то время как выполняется интерпретируемая программа, JIT-компилятор определяет участки часто используемого кода и компилирует его в машинный код. В зависимости от компилятора это можно сделать для метода или меньшего участка кода.

Впервые динамическая компиляция была описана в статье о языке LISP Дж. Маккарти в 1960 году.

Компиляция на лету, JIT или динамическая компиляция – это компиляция, которая выполняется непосредственно во время исполнения программы, а не до этого. Что в этот момент происходит? Перевод в машинный код. Преимущества JIT-компиляции заключаются в том, что поскольку компиляция происходит во время выполнения, то JIT-компилятор имеет доступ к динамической информации времени выполнения, а это в свою очередь позволяет ему оптимизировать процесс (например, встраивать функции).

Что важно понимать, когда речь идет о JIT-компиляции? Она скомпилирует байт-код в инструкции машинного кода работающего компьютера. Это означает, что полученный машинный код оптимизирован для архитектуры процессора конкретного компьютера.

В качестве примеров JIT-компиляторов можно привести JVM (Java Virtual Machine - виртуальная машина Java) на Java и CLR (Common Language Runtime – общеязыковая исполняющая среда) на C#.

История

Изначально компилятор отвечал за преобразование языка высокого уровня (выше, чем ассемблер) в объектный код (машинные инструкции), который затем должен был быть связан (линкером) с исполняемой программой.

В какой-то момент эволюции языков компиляторы начали компилировать язык высокого уровня в псевдокод, который затем интерпретировался (интерпретатором) для запуска программы. Это исключило объектный код и исполняемые программы и позволило перенести эти языки на несколько операционных систем и аппаратных платформ. Одним из первых был Pascal (который скомпилирован в P-Code); более современными примерами являются Java и C#. Со временем термин P-Code был заменен на байт-код, поскольку большинство псевдоопераций имеют длину в один байт.

JIT-компилятор – это функция интерпретатора, которая вместо интерпретации байт-кода при каждом вызове компилирует байт-код в инструкции машинного кода работающей машины, а затем вызывает этот объектный код. В идеальном варианте эффективность выполнения объектного кода должна превзойти неэффективность перекомпиляции программы при каждом ее запуске.

Обычный сценарий

Исходный код полностью преобразуется в машинный код.

JIT-сценарий

Исходный код преобразуется в структуру на языке ассемблера, например, IL (промежуточный язык) для C#, ByteCode для Java.

Промежуточный код преобразуется в машинный только тогда, когда приложение нуждается в том, чтобы необходимые коды были преобразованы в машинный код.

JIT или не JIT

При JIT-компиляции не весь код преобразуется в машинный код. Для начала преобразуется только необходимая часть кода. Затем, если вызываемый метод или выполняемые функции находятся не в виде машинного кода, то они тоже будут преобразованы в машинный код. Это снижает нагрузку на ЦП. Поскольку машинный код будет генерироваться во время выполнения, то JIT-компилятор создаст машинный код, оптимизированный для запуска архитектуры ЦП машины.

Ниже приведены некоторые примеры JIT-компиляторов:

Java: JVM (Java Virtual Machine – виртуальная машина Java)

C#: CLR (Common Language Runtime – общеязыковая исполняющая среда)

Android: DVM (Dalvik Virtual Machine – виртуальная машина Dalvik) или ART (Android RunTime – среда выполнения Android-приложений) в новых версиях

Виртуальная машина Java (JVM) выполняет байт-код и ведет подсчет времени выполнения функции. Если это значение превышает предустановленный порог, то JIT-компилятор компилирует код в машинный код, который в дальнейшем может быть выполнен непосредственно процессором (в отличие от случая, когда javac компилирует код в байт-код, а затем интерпретатор интерпретирует этот байт-код построчно, переводя его в машинный код, и выполняет его).

Кроме того, при следующем вычислении функции тот же скомпилированный код выполняется снова, в отличие от обычной интерпретации, когда код повторно интерпретируется построчно. Это значительно ускоряет процесс выполнения программы.

LLVM против JVM

Последнее обновление: 25 апреля 2022 г.

Время компиляции:

- **JVM** : компилирует исходный код (например, Java/Kotlin/Scala) в **байт-код** ;
- **ЛЛВМ** :
  - интерфейс (например, Clang/Flang) компилирует исходный код в **IR** (промежуточное представление, IR с самого начала разрабатывался как переносимая сборка).
  - серверная часть превращает IR в собственный двоичный исполняемый файл.

Время выполнения:

- **JVM** : сбор мусора, доступ к ресурсам и т. д. JVM — это интерпретатор байт-кода Java. Он должен запускаться во время выполнения программы. (Большой размер и более высокие накладные расходы).
- **LLVM** : не требуется во время выполнения (поскольку он заранее генерирует исполняемые файлы для конкретной архитектуры)

Точно в срок и раньше времени

Как обсуждалось выше, LLVM работает в первую очередь **с опережением времени** , JVM — прежде всего **с опережением времени** .
LLVM также поддерживает компиляцию «точно в срок» (на основе сгенерированного IR), поскольку в некоторых случаях код необходимо генерировать «на лету», например, при использовании REPL в Julia. (`lli`: непосредственно выполняет программы в формате битового кода LLVM. Он принимает программу в формате битового кода LLVM и выполняет ее с использованием JIT-компилятора или интерпретатора.)
GraalVM может заранее скомпилировать JAVA-приложение в собственные двоичные файлы.

на основе регистров или на основе стека

- **LLVM** : низкоуровневая виртуальная машина на основе регистров. Он предназначен для абстрагирования базового оборудования и проведения четкой границы между серверной частью компилятора (генерация машинного кода) и внешней частью (анализ и т. д.).
- **JVM** : виртуальная машина более высокого уровня на основе стека (вместо загрузки значений в регистры байт-код JVM загружает значения в стек и вычисляет значения оттуда).

Кросс-язык

Благодаря IR и байт-коду и LLVM, и JVM могут поддерживать несколько языков.

- **LLVM** : C, C++, Rust, Swift, Fortran, Kotlin/Native и т. д.
- **JVM** : Java, Kotlin/JVM, Scala, Groovy, Closure и т. д.
  - **GraalVM** : Java, JavaScript, Python, Ruby, R, WASM и т. д.

LLVM предоставляет примитивы для общих функций языков программирования. Например, функции, глобальные переменные, сопрограммы и интерфейсы внешних функций C. Многие из них есть в LLVM в качестве стандартных элементов в его IR. Разделение клиентской и серверной части освобождает компиляторы языков высокого уровня от необходимости ориентироваться на каждую платформу (им нужно только создавать промежуточное представление LLVM).
Интерфейсы LLVM:

- Clang: для семейства языков C, например C, C++, Objective-C.
- Флаг: для Фортрана, добавлен в LLVM 13.

Выполнение

- Сама LLVM написана на C++; он предоставляет API C и C++.
- OpenSDK написан на C++.
- GraalVM написан на Java.

```
public class bcode {
    public static void main(String[] args){
        String str1 = new String("test");
        String str2 = new String("test");
    }
}
```

```
// class version 60.0 (60)
// access flags 0x21
public class bcode {

  // compiled from: bcode.java

  // access flags 0x1
  public <init>()V
   L0
    LINENUMBER 1 L0
    ALOAD 0
    INVOKESPECIAL java/lang/Object.<init> ()V
```

```
 RETURN
L1
 LOCALVARIABLE this Lbcode; L0 L1 0
 MAXSTACK = 1
 MAXLOCALS = 1

// access flags 0x9
public static main([Ljava/lang/String;)V
 L0
 LINENUMBER 5 L0
 NEW java/lang/String
 DUP
 LDC "test"
 INVOKESPECIAL java/lang/String.<init> (Ljava/lang/String;)V
 ASTORE 1
L1
 LINENUMBER 6 L1
 NEW java/lang/String
 DUP
 LDC "test"
 INVOKESPECIAL java/lang/String.<init> (Ljava/lang/String;)V
 ASTORE 2
L2
```

```
    LINENUMBER 7 L2
    RETURN
   L3
    LOCALVARIABLE args [Ljava/lang/String; L0 L3 0
    LOCALVARIABLE str1 Ljava/lang/String; L1 L3 1
    LOCALVARIABLE str2 Ljava/lang/String; L2 L3 2
    MAXSTACK = 3
    MAXLOCALS = 3
}
```

Соответственно появилась идея разобрать байткоды для двух вариантов main-метода с инициализацией одинаковых строк, и попытаться прояснить почему они дают разные результаты:

Вариант 1:(если сравнить str1==str2, то получаем false)

```
    public static void main(String[] args){
        String str1 = new String("test");
        String str2 = new String("test");
    }
```

Вариант 2:(если сравнить str3==str4, то получаем true)

```
    public static void main(String[] args){
        String str3 = "test";
        String str4 = "test";
    }
```

Байткод первого варианта(в квадратных скобках[] будет состояние стека после выполнения команды(вершина стека справа)):

public static main([Ljava/lang/String;)V

L0

LINENUMBER 5 L0

NEW java/lang/String

//новая строка - кладем ссылку на класс "String "на вершину стека: [value_string]

DUP

//копируем ссылку на вершине стека: [value_string, value_string]

LDC "test"

//кладём в стек ссылку на "test" из constant pool(а если в нём еще нет ссылки на "test",

//то она также остается в constant pool'e): [value_string, value_string, value_test]

INVOKESPECIAL java/lang/String. (Ljava/lang/String;)V

//вызов new для String с аргументами с вершины стека -

//после вызова инициализирующего метода для стринг(и его внутренних манипуляций со стеком,

//которые аналогично можно посмотреть в байткоде java.lang.String)

//- в стеке будет ссылка на проинициализированную строку: [value_result_string]

//Почему для инициализации String(String) нужно три аргумента в стеке:

//1)ссылка на входной аргумент 2)новая строка, 3)хешкод строки

ASTORE 1

//сохраняем ссылку с вершины стека в локальную переменную(1) с вершины стека: [ ]

L1

LINENUMBER 6 L1

NEW java/lang/String //аналогично первой строке

DUP

LDC "test"

INVOKESPECIAL java/lang/String. (Ljava/lang/String;)V

ASTORE 2

L2

LINENUMBER 7 L2

RETURN //return void - компилятор дописывает за нас если метод возвращает void

L3

LOCALVARIABLE args [Ljava/lang/String; L0 L3 0 // список локальных переменных метода

LOCALVARIABLE str3 Ljava/lang/String; L1 L3 1 // L1(метка где операции с локальной переменной) L3(метка где описание что это за локальная переменная) 1(номер локальной переменной)

LOCALVARIABLE str4 Ljava/lang/String; L2 L3 2 //

MAXSTACK = 3 //максимальная глубина стека метода

MAXLOCALS = 3 //количество локальных переменных метода

//

//То есть сначала JRE будет выделять память при

//вызове метода учитывая всю вложенность вызовов методов и new - инициализаций

//(и если её не хватит - то всё вылетит с каким-нибудь OutOfMemory

//эксепшном), а только потом будут производиться операции

Байткод второго варианта:

public static main([Ljava/lang/String;)V

L0

LINENUMBER 5 L0

LDC "test" //берем ссылку на "test" из constant_pool и кладем на вершину стека: [value_test]

ASTORE 1 //забираем ссылку с вершины стека в переменную(1) str1 : [ ]

L1

LINENUMBER 6 L1

LDC "test" //берем ссылку на "test" из constant_pool и кладем на вершину стека: [value_test]

ASTORE 2 //забираем ссылку с вершины стека в переменную(2) str2: [ ]

L2

LINENUMBER 7 L2

RETURN //return void (который компилятор дописывает за нас как успешный конец метода)

L3

LOCALVARIABLE args [Ljava/lang/String; L0 L3 0 //список локальных переменных, под которые выделяется память

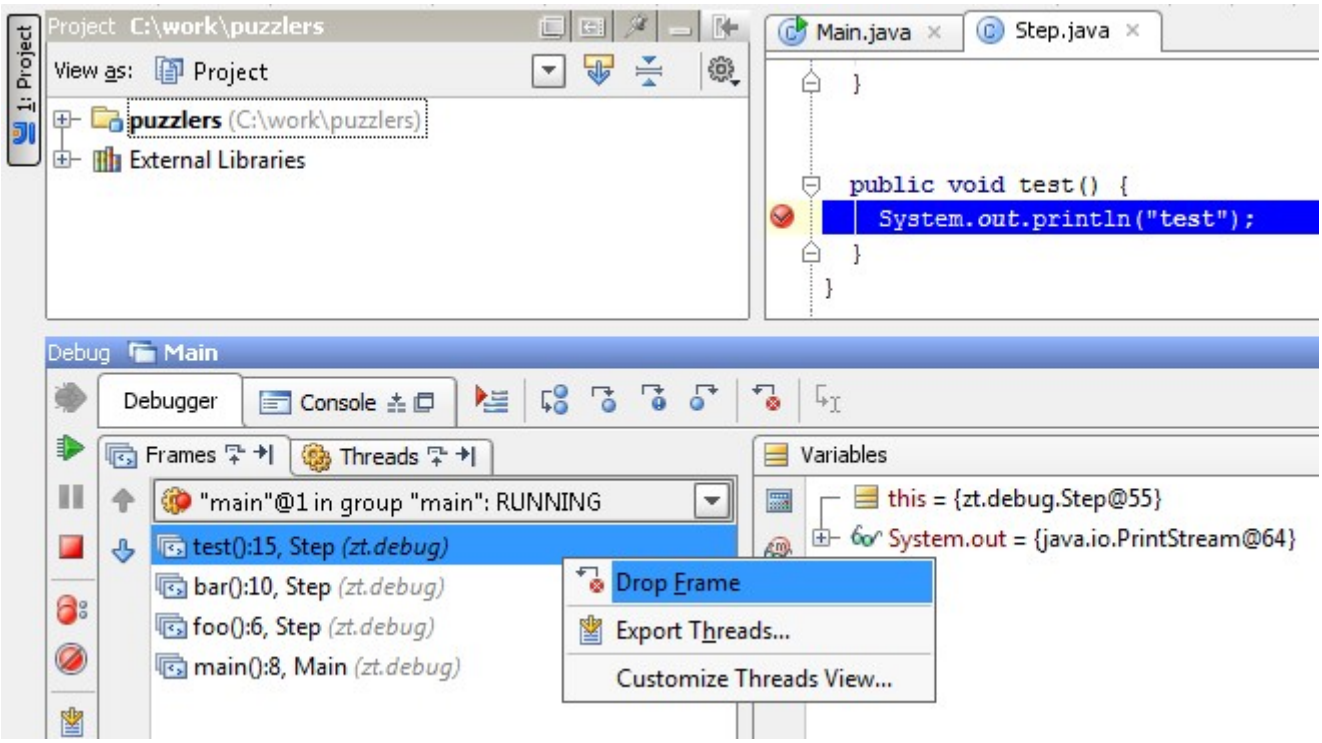
LOCALVARIABLE str1 Ljava/lang/String; L1 L3 1 //например: L1(метка где записан код) L3(метка где описан тип объекта и выделяется память) 1(локальный номер объекта)

LOCALVARIABLE str2 Ljava/lang/String; L2 L3 2 //

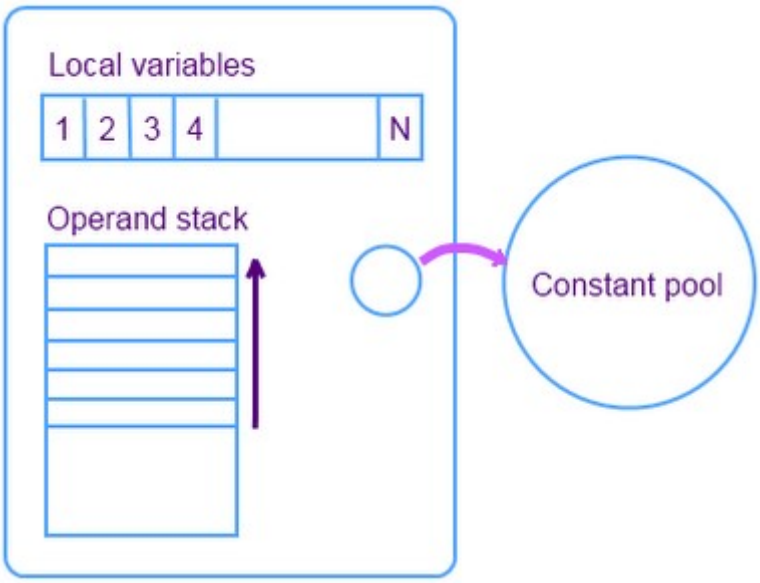
MAXSTACK = 1 //максимальная глубина стека для метода

MAXLOCALS = 3 //количество локальных переменных метода

Чтобы понять, как работает байт-код, достаточно взглянуть на модель выполнения. JVM использует модель выполнения на основе стеков. Каждый тред имеет JVM-стек, содержащий фреймы. Например, если мы запустим приложение в дебаггере, то увидим следующие фреймы:



При каждом вызове метода создается новый фрейм. Фрейм состоит из стека операнда, массива локальных переменных и ссылку на пул констант класса выполняемого метода.



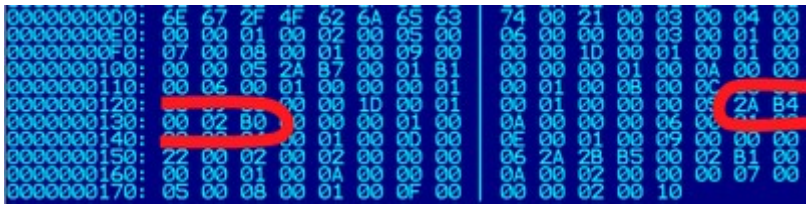
Размер массива локальных переменных определяется во время компиляции в зависимости от количества и размера локальных переменных и параметров метода. Стек операндов — LIFO-стек для записи и удаления значений в стеке; размер также определяется во время компиляции. Некоторые опкоды добавляют значения в стек, другие берут из стека операнды, изменяют их состояние и возвращают в стек. Стек операндов также используется для получения значений, возвращаемых методом (return values).

```
public String getBar(){
    return bar;
  }
```

```
public java.lang.String getBar();
  Code:
    0:   aload_0
    1:   getfield        #2; //Field bar:Ljava/lang/String;
    4:   areturn
```

Байткод для этого метода состоит из трёх опкодов. Первый опкод, aload_0, проталкивает в стек значение с индексом 0 из таблицы локальных переменных. Ссылка *this* в таблице локальных переменных для конструкторов и instance-методов всегда имеет индекс 0. Следующий опкод, getfield, достает поле объекта. Последняя инструкция, areturn, возвращает ссылку из метода.

Каждый метод имеет соответствующий байткод-массив. Смотря на содержимое .class-файла в hex-редакторе, вы увидите в байткод-массиве следующие значения:



Так, байткод для метода getBar — 2A B4 00 02 B0. 2A относится к инструкции aload_0, B0 — к areturn. Может показаться странным, что байткод для метода имеет три инструкции, а в массиве байт 5 элементов. Это связано с тем, что getfield (B4) нуждается в двух параметрах (00 02), занимающих позиции 2 и 3 в массиве, отсюда и 5 элементов в массиве. Инструкция areturn сдвигается на 4 позицию.

Таблица локальных переменных

Машинный код (байткод) всегда предназначен для исполнения каким-либо процессором, физическим либо виртуальным. Процессоры (машины) могут быть стековыми либо регистровыми:

- Стековая машина для вычислений использует только стек, постоянно добавляя на него новые значения либо снимая старые, что приводит к непрерывному изменению вершины стека.
- Регистровая может использовать стек, но кроме него использует также независимые регистры, распределяя и переиспользуя регистры для последовательных вычисления.

В стековом байткоде виртуальной машины инструкции короче, а сам подход позволяет написать компилятор значительно легче. Регистровый позволяет проводить операции с памятью намного эффективнее, особенно с учётом современных процессоров, которые практически всегда являются регистровыми с программным стеком в оперативной памяти (более медленной, чем регистры).

До появления Java, примерно 25 лет назад, многие приложения писали на C или C++. У этих языков есть проблема: разработчику приходится думать, на какой операционке и архитектуре процессора будет работать его код. Например, если он пишет программу под Linux, то, скорее всего, она не запустится на Windows или MacOS. Поэтому код приходилось пичкать **директивами условной компиляции** или **писать отдельную версию** для каждой операционки.

Самую сильную боль вызывали приложения с GUI — код графических компонентов для разных операционок был совершенно разным. Причина в том, что C и C++ довольно близки к железу, а значит, сишник должен учитывать архитектуру процессора и тип операционной системы. Понятно, что никакой кроссплатформенностью здесь и не пахнет.

Ещё в 1960-е годы у инженеров появилась идея: писать программы не для конкретного железа, а для абстрактного «исполнителя». Программы на Java как раз пишутся для такого исполнителя — виртуальной машины, или Java Virtual Machine (JVM). Java-разработчик не задумывается, на какой платформе будет запускаться его код. В то же время виртуальная машина не знает, что исполняет инструкции на Java, ведь она принимает и исполняет байт-код.

Кроссплатформенность в Java обеспечивается явным разделением уровней языка и реализации.

Языковой уровень. Разработчики пишут код на языке Java, синтаксис и семантика которого описаны в Java Language Specification. После этого специальным инструментом, который называется javac, исходный код компилируется в байт-код Java. При этом происходит проверка синтаксиса, и в случае его нарушения разработчик получает сообщение об ошибке от javac.

Что здесь важно:

- На этом этапе нет ничего платформенно-специфичного, весь код на языке Java (как и байт-код Java) универсален.
- Байт-код Java — это язык, предназначенный не для людей, а для машин. Обычному разработчику его читать не нужно.

Уровень реализации. Полученный байт-код Java передаётся на вход виртуальной машины Java. И вот как именно она будет его исполнять, описано уже в другой спецификации — Java Virtual Machine Specification. Со всеми особенностями конкретной операционной системы или архитектуры процессора тоже разбирается JVM, без влияния на исходный код на языке Java. Таким образом, происходит перенос ответственности: разработчики на Java о таких неприятных вещах больше не думают, им достаточно просто взять правильную JVM (например, для Linux x64). А все OS/arch-специфические нюансы решают разработчики JVM.

Язык Java и сама JVM разрабатываются параллельно, при этом новые фичи языка зачастую требуют поддержки внутри JVM. С другой стороны, различные реализации JVM развиваются и сами по себе, например в них появляются новые, более эффективные алгоритмы сборки мусора или оптимизации кода.

У разделения языка и его реализации есть замечательное следствие: разработчику не обязательно ограничиваться языком Java, ведь виртуальная машина понятия не имеет, откуда взяли байт-код, который пришёл к ней на вход, был ли это изначально Java-код или что-то другое. То есть программу можно писать на любом языке, который транслируется в байт-код. И таких языков — целое семейство: Kotlin, Clojure, Groovy и так далее. Например, программы на Kotlin компилируются с помощью kotlinc в class-файлы, а затем подаются на вход JVM.

Как соотносятся JDK и Open JDK

Java Development Kit *(JDK)* — это комплект инструментов для разработки приложений на Java. В него входят несколько компонентов, которые позволяют разрабатывать и запускать приложения:

- **Javac.** Это компилятор, который преобразует исходники Java в class-файлы или формат jar.
- **JVM.** Виртуальная машина, которая исполняет байт-код.
- **Стандартная библиотека.** Набор модулей, классов, объектов и функций, которые доступны в языке.
- **Документация.** В ней содержится справочная информация об инструментах данной версии JDK.

Когда появляется новая версия языка, все эти компоненты обновляются.

В случае стандартной JVM (например, HotSpot) запуск Java-приложения выглядит так: сначала с помощью javac вы компилируете исходный код в class-файлы (или jar-файлы), и именно их вы можете поставлять в качестве приложения.

Чтобы приложение запустилось на хосте, там должна быть установлена виртуальная машина. Тогда пользователь пишет в командной строке: java -jar <*имя вашего jar-файла*> — и программа запускается.

С нашей виртуальной машиной сценарий запуска приложения был другой: мы заранее компилировали class-файлы в один исполняемый exe-файл. Поэтому он работал на компьютере пользователя без каких-либо предустановленных JVM. Но, по сути, JVM никуда не исчезла — просто все её части были слинкованы в один экзешник.
В спецификации Java Virtual Machine почти ничего не сказано о том, как именно её реализовывать — только что должно получиться на выходе. Например, вы не можете явно освободить память из под объекта, когда он вам уже не нужен. Этим занимается специальный компонент JVM — Garbage Collector. Но вот как именно он работает, в спецификации не сказано.

Так и было задумано: спецификация позволяет создавать разные JVM, реализации которых в чём-то отличаются, но при этом удовлетворяют требованиям спецификации. И это замечательно, пусть расцветают сто цветов! Такие различия в реализации дают разным виртуальным машинам особые конкурентные преимущества: у кого-то более крутой сборщик мусора, у кого-то быстрый стартап, а кто-то реализовал более мощные оптимизации в компиляторе.

Первую JVM, как и язык Java, разработали в компании Sun Microsystems. Когда Sun опубликовала спецификацию, Oracle стала работать над своей JRockit, а IBM — над OpenJ9. Наша Excelsior JET вышла в свет в 2000 году.

Изначально все JVM были закрытыми, но в 2006 году Sun опубликовала исходники своей JVM HotSpot и компилятора javac, а потом и стандартной библиотеки Java. Это дало старт проекту OpenJDK — полностью открытой реализации JDK. Через какое-то время Sun была поглощена корпорацией Oracle, но при этом проект OpenJDK никуда не делся и до сих пор остаётся главной open-source-площадкой для разработки Java и JVM.

OpenJDK — это эталонная реализация, на которую ориентируются разработчики других виртуальных машин. Проект распространяется под GPL 2 + Classpath Exception, благодаря чему любая компания или разработчик, которым хватит квалификации, смогут форкнуть OpenJDK и начать разрабатывать свою собственную JVM. Конечно же, по условию лицензии их реализация тоже будет open source.

Существуют и реализации JVM, которые никак не связаны с OpenJDK. Ведь не только Sun Microsystems и Oracle создавали JVM. Другой пример — это IBM J9, которая, кстати, тоже открыла исходный код и стала называться OpenJ9.

Некоторые компании продают свои виртуальные машины. Например, у Azul есть свободная сборка Zulu и закрытая коммерческая Azul Zing. В последней есть уникальный сборщик мусора C4, который гарантировал низкие паузы ещё до того, как это стало мейнстримом. А также технология ReadyNow!, которая позволяет «запомнить» список всех оптимизаций кода и «проиграть» его на старте JVM, выводя машину на пиковую производительность.

Excelsior JET тоже была закрытой и платной. Она обеспечивала быстрый стартап, низкое потребление памяти и хорошую пиковую производительность во многих

сценариях. Всё это мы получали за счёт статической компиляции, в отличие от динамической компиляции на лету в HotSpot.

Однако сегодня люди чаще пользуются бесплатными сборками, основанными на OpenJDK. Существуют как стандартные сборки OpenJDK от Oracle, так и сборки от других компаний и разработчиков, имеющих отношение к разработке OpenJDK (зачастую их публикуют контрибьюторы в OpenJDK не из Oracle).

В таких сборках могут быть дополнительные фичи, не включённые по умолчанию в стандартные сборки от Oracle. Либо такие альтернативные сборки прошли дополнительную сертификацию, например для использования с «КриптоПро». При этом такие сторонние билды проходят все стандартные тесты, поэтому гарантируется, что ваши программы будут работать как надо и пользоваться ими можно абсолютно безопасно и бесплатно.

Некоторые компании не только выкладывают свои билды, но и оказывают платную поддержку пользователям. Допустим, возникла проблема с виртуальной машиной: ваше приложение работает неожиданно медленно или даже крашится! Всякое бывает, ведь JVM — это тоже программа, в которой могут быть свои баги. Тогда инженеры этой компании оперативно разберутся в проблеме и решат её.

В целом вариантов виртуальных машин и их сборок очень много — в рамках этой статьи все не перечислить. Есть сборки от Oracle, Red Hat, Azul, IBM, BellSoft, Amazon и Microsoft. Сообщество Java-чемпионов написало замечательный документ, в котором перечислило все варианты и в целом разъяснило ситуацию с многообразием JVM и их сборок. Документ называется Java Is Still Free, рекомендую его всем к прочтению.

Как и зачем другие языки переходят на JVM

Java — это не просто язык, а целая философия. И она оказалась настолько востребованной, что у других языков программирования стали появляться реализации под JVM. Например, Jython и JRuby.

Главная причина появления таких реализаций — перформанс. За 25 лет человечество научилось делать хорошо оптимизированные JVM. Так почему бы за счёт этого не повысить производительность программ на Python и Ruby?

С другой стороны, так можно получить хорошую связь с экосистемой Java. Если мы запускаем код на JVM, он лучше взаимодействует с другими модулями на JVM. Значит, можно пользоваться и Java-библиотеками.

У Kotlin немного другая история. Когда язык только появился, было решено компилировать его в байт-код Java и запускать на JVM. Таким образом, реализация языка заключалась в написании хорошего транслятора — kotlinc — из исходного кода .kt в class-файлы. А реализация низкоуровневых компонентов типа сборщиков мусора или взаимодействия с операционной системой делегировалась уже существующим JVM.

Относительно недавно создатели Kotlin сделали Kotlin Native — технологию компиляции Kotlin в нативный код напрямую, без JVM. В нём низкоуровневые компоненты, такие как менеджер памяти, уже реализованы самостоятельно.

Хотя на словах кажется, что любой язык программирования легко перевести на JVM (достаточно написать транслятор в class-файлы), на деле всё оказывается намного сложнее. Грамотно отобразить фичи языка в байт-код Java и при этом получить хорошую производительность — нетривиальная задача. Посмотрите хотя бы на JRuby.

Чарльз Наттер часто делится опытом поддержки различных фич Ruby в проекте JRuby и рассказывает о сложностях, с которыми сталкивается. Чем дальше исходный язык от Java, тем сложнее задача трансляции. Но, думаю, всё возможно, и JRuby — наглядный тому пример.

Какие конкуренты есть у JVM

Самый известный конкурент Java Virtual Machine — платформа .NET и их виртуальная машина для реализации C#.

В начале 2000-х Microsoft делала свою виртуальную машину Java — Microsoft J++. Но из-за того, что корпорация не соблюдала спецификацию, Sun подала на неё судебный иск. Microsoft проиграла все суды и лишилась права делать виртуальную машину для Java. Есть мнение, что это стало одной из причин появления C#.
Java и C# решают одни и те же задачи и обладают одними и теми же преимуществами: строгая типизация, сборка мусора, безопасность — всё это совпадает. Это managed-языки широкого применения.

Раньше между платформами была одна принципиальная разница: Java был кроссплатформенным, а .NET и C# работали только на Windows. Но с тех пор, как появился .NET Core, C# тоже стал мультиплатформенным.

Отмечу, что сами языки Java и C# довольно разные, и C# развивается быстрее: там регулярно появляются интересные фичи, которые в Java приходят с большим опозданием или не приходят вообще. Поэтому как язык C# в целом выглядит интереснее.

Зато у Java мощная реализация и хорошо оптимизированные виртуальные машины. Это стало возможным благодаря «гонке вооружений» между разработчиками виртуальных машин. Вендоры на протяжении 20 лет соревновались, кто сделает самую крутую оптимизацию, лучший алгоритм сборки мусора, более быстрый стартап и так далее. Поэтому по многим показателям реализация Java выглядит куда лучше C#. Некоторые вещи, которые уже давно есть в JVM, только-только появляются в .NET.

Есть и другие языки, которые претендуют на роль конкурента Java, но они отличаются гораздо сильнее. Например, Python, который недавно занял 1-е место в рейтинге TIOBE. Он популярен в Data Science, AI, машинном обучении, скриптах

и так далее. К сожалению, динамическая типизация не позволяет держать на нём слишком большую кодовую базу.

Ещё есть безумно популярный JavaScript, но его интересы с Java не очень пересекаются, так как сегодня Java во фронтенде не используется. Хотя когда-то были и такие планы (привет устаревшей технологии Java-апплетов).

Где Java хорош, а где — нет

Java широко используют в бэкенде, то есть во всех приложениях, которые работают на стороне сервера. Его главный конкурент — C#, но Java всё-таки значительно лидирует, потому что считается более кроссплатформенным. Хотя последнее уже не совсем правда. Также стоит отметить, что у Java банально есть фора по времени по сравнению с C#. Он появился на 10 лет раньше, поэтому успел захватить умы разработчиков и кодовые базы.

Почему Java, а не «швейцарский нож» C++? Просто Java удобнее и безопаснее для жизни, или, как принято говорить, не даст выстрелить себе в ногу. Разработчику не нужно думать об управлении памятью и сегфолтах, которые могу прилететь хоть откуда. Поэтому, когда дело касается будничных задач бэкенда, скорость разработки на Java и C# гораздо выше, чем на C++.

Java плохо подходит для приложений, которым требуется высокая производительность и гарантированное время отклика. Тем не менее некоторые пишут на Java и такие программы, если готовы мириться с небольшими задержками в микросекунды. Для них выигрыш от скорости разработки и надёжности важнее. Но писать на Java программное обеспечение для кардиостимуляторов, где все элементы должны работать точно, как швейцарские часы, — не стоит.

Кадр: фильм «Человек-паук 2»

С другой стороны, приложения для трейдинга всё чаще пишут именно на Java. Казалось бы, где, если не в трейдинге, важен быстрый отклик — чтобы не потерять миллионы из-за скачка цен на акции Tesla? Однако на практике надёжность здесь важнее скорости: лучше торговать медленнее, но спокойно и надёжно, чем написать на C++ быструю биржу, которая через пять минут торгов крашнется. Java гарантирует безопасность, и это круто.

Что не так с JVM

В истории развития любого языка программирования всегда находятся тонкие места или сомнительные решения. Java здесь не исключение, встречаются такие спорные вопросы и в спецификациях Java и JVM.

Система модулей

Одна из спорных тем — **Project Jigsaw и система модулей**, появившаяся в девятой версии Java. Создатели Java долго делали систему, которая позволяет разбивать

Java-программу на отдельные логические составляющие — модули. Это дополнительный уровень абстракции над давно существующими packages в Java.

На неё потратили много времени — ведь нужно было организовать поддержку в самом языке, виртуальной машине и других частях JDK. Система долго не попадала в релизную версию, а когда всё-таки попала, оказалась, мягко говоря, не слишком востребованной пользователями.

Как в языке появляются новые фичи и спецификации? Есть специальный совет Java Community Process (JCP), который рассматривает предложения Java Specification Request (JSR) и голосует за или против. В этом совете есть представители крупных компаний, есть представители user group и даже отдельные люди — в общем, представлены интересы разных сторон Java Community.
Идея новой модульной системы в Java была несколько раз отвергнута JCP (что вообще случается довольно редко), но всё-таки вошла в язык после значительных доработок и послаблений. Тем не менее спустя несколько лет мы видим, что идея так и не пришлась по вкусу Java-разработчикам и модулями пользуются редко.

Дженерики

Дженерики в Java, в отличие от многих других языков, — стираемые. Это означает, что все типовые параметры стираются через javac до Object. В рантайме вы не сможете понять, какой тип вам на самом деле пришёл. У этого подхода есть свои плюсы и минусы, а споры о том, что лучше — стираемые или нестираемые дженерики — продолжаются и по сей день. Тем не менее, выбор в пользу стираемых дженериков в Java сделан не случайно. Подробное обоснование, почему дженерики в Java такие, можно почитать вот в этой статье Брайана Гетца «Background: how we got the generics we have».

Но как бы вы не относились к реализации дженериков в Java, сейчас есть существенная проблема: нельзя использовать примитивные типы в качестве типовых параметров (ведь они несовместимы с Object). Это сокращает выразительную силу языка и вынуждает либо использовать Boxed-типы (что может ухудшить производительность), либо и вовсе специализировать функции под примитивные типы вручную. Исправить это собираются с помощью специализированных дженериков в рамках проекта Valhalla.

Обратная совместимость

Обратная совместимость — один из главных принципов Java, которым совет JCP не пожертвует даже ради самой передовой фичи. От некоторых вещей иногда отказываются, но при этом старый код всё равно должен продолжать работать.

Например, реализация исключений через инструкции JSR/RET встречается в старом байт-коде, а в новом — нет. Но до сих пор во всех тулах нужно уметь поддерживать старый байт-код ради библиотеки, написанной 20 лет назад, но которая так и продолжает использоваться в современных проектах в виде того же старого формата .jar.

С одной стороны, этот принцип позволяет построить стабильную экосистему. Бизнес может не бояться, что выйдет новая версия, которая всё сломает и навсегда

заблокирует путь к обновлению. С другой стороны, это тормозит инновации. Возможно, Java развивается медленнее, чем C#, потому, что у нас обратная совместимость — это священная корова, которую нельзя трогать.

Насколько глубоко разработчику нужно знать JVM

На самом деле это очень холиварный вопрос. В основном ответ на него зависит от задач, которые хочет решать джавист.

Думаю, 30% программистов могут спокойно работать и не понимать, что происходит внутри виртуальной машины. Другим 50% достаточно на среднем уровне знать, как устроена JVM, — тогда они будут отличными мидлами и сеньорами. Их работа не в том, чтобы копаться в кишках виртуальной машины, но некоторые знания о её устройстве помогут им писать более эффективный код и избежать многих ошибок. В оставшихся 20% задач без знания деталей всё-таки не обойтись. Получается, чем лучше вы разбираетесь в виртуальной машине, тем более широкий класс задач способны решать.

Если в проекте важна производительность и повысить её с помощью выбора более оптимального алгоритма нельзя, надо разбираться, как работает сборщик мусора и как повысить его эффективность, какие настройки можно поменять, что такое деоптимизация, почему она происходит и так далее.

Java-разработчик, который регулярно погружается в эту тему, со временем сможет стать Performance Engineer. Эти специалисты находят проблемы производительности, анализируют глубинные процессы в JVM и исправляют их. Они решают проблемы не дописыванием кода в виртуальную машину, а настраивают её, как часовщики.

С точки зрения культуры всем разработчикам на managed-языках полезно знать, что происходит под капотом машины, когда они нажимают кнопку Run в IDE. Конечно, не нужно запоминать, что и в какой момент делает каждая деталь JVM. Но пройти вводный курс и понять на концептуальном уровне, что такое интерпретатор, Just In Time Compiler, что значит «умер объект» и так далее, — полезно.

Разработчик, который не знает ответов на все эти вопросы, способен решить довольно много задач. Но однажды он столкнётся с ошибками, причины которых даже не сможет понять. Например, если не задумываться, откуда берётся память и как она возвращается операционной системе, то можно получить от IDE сообщение *Out of memory*. А всё потому, что программа потребляет слишком много памяти. Разработчик, который знает, как устроена память, постарается оптимизировать код и не создавать горы объектов.

Куда развивается JVM

Сейчас довольно активно развиваются три проекта, которые рано или поздно войдут в новые версии Java.

Loom

Loom — это проект по добавлению виртуальных тредов в Java. С ним треды не мапятся на треды операционной системы, а живут сами по себе. И лишь когда их нужно исполнить, они будут садиться на треды операционки.

Раньше каждому экземпляру класса java.lang.Thread (через которые реализуется многопоточное исполнение) соответствовал один поток операционной системы. Вполне логичное решение, но есть проблема: потоков операционной системы не слишком много, поэтому вряд ли у вас получится создать больше нескольких тысяч потоков, реализованных таким образом. Project Loom предлагает альтернативу — виртуальные потоки, которые уже не связаны соотношением 1 к 1 с потоками виртуальной системы. Вместо этого они время от времени садятся на эти OS threads, чтобы выполнить очередной квант работы.
Идея не новая. Например, в Go так было всегда. Да и вообще о виртуальных тредах говорили уже много десятилетий. И даже в самом Java в 1990-е были грин-треды — схожая идея, где N зелёных потоков назначались на один поток операционной системы (тогда в процессорах было по одному ядру и проблема легковесной многопоточности не стояла так остро). Можно сказать, что Loom — это в некотором смысле реинкарнация зелёных тредов, но сделанная на более серьёзном уровне.

Этот проект очень важен, так как в современном мире микросервисов и серверов с огромным количеством одновременных подключений вам часто хотелось бы создавать десятки и сотни тысяч тредов. Со старым подходом к реализации через OS threads это просто невозможно, а вот с Loom всё получится.

Panama

Panama — это проект, посвящённый взаимодействию между Java и нативным кодом, написанным, например, на C или C++. Конечно, Java мог взаимодействовать с нативным кодом с самого рождения. Основное решение в этой области — Java Native Interface, очень мощный, но медленный и неудобный в использовании механизм. Чтобы взаимодействовать с нативами, приходилось писать некрасивый, громоздкий и, что самое главное, небезопасный код. Я уверен, что любому джависту, который хоть что-нибудь делал с JNI, как минимум не понравилось, а как максимум — он до сих видит кошмары про развалы в JNI.

Чтобы избавить Java-разработчиков от ночных кошмаров, авторы Panama уже много лет работают над своим проектом, который, как Панамский канал, соединяет два мира: нативный и Java. Panama предлагает новый интерфейс и его оптимизированную реализацию на стороне JVM, которая позволит приятно и безопасно работать с нативным кодом, в том числе с off-heap.

Valhalla

Проект Valhalla посвящён введению в Java новых типов объектов, так называемых value-типов, или inline-типов, или же primitive-value-типов — это текущее название. Да, проект настолько сложный, что даже основная его концепция часто меняет имя!

Если коротко, то смысл в том, чтобы выделить подкласс объектов, у которых нет identity. Что это такое? В Java вы можете создать два абсолютно одинаковых объекта с помощью оператора new. Потом сравнить их на == и получить false. Это разные объекты! Они сравниваются не по содержимому (которое одинаковое), а по некоторому уникальному свойству, назначаемому каждому объекту при создании. Вот возможность сравнивать не по содержимому, а по чему-то ещё — это и есть identity.

С другой стороны, можно объявить две переменные типа int, записать в каждую из них значение 42, сравнить на == и получить результат true. Всё потому, что у int никакого identity нет, они сравниваются по содержимому.

Цель Valhalla — дать пользователям возможность самим описывать классы, у которых нет identity, и подготовить к этому уже существующую типовую систему. Как следствие, можно будет, например, заводить плоские поля таких типов внутри других классов (то есть фактически инлайнить их поля внутрь других объектов), работать с плоскими массивами таких объектов и даже получить специализированные дженерики.

Это огромное, фундаментальное изменение всей платформы Java, за которым очень интересно следить и которого стоит ожидать, ведь оно принесёт множество возможностей обычным разработчикам.

На каких языках разрабатывают JVM и библиотеки для Java

Сами виртуальные машины Java обычно пишут на C++, стандартная библиотека же в основном написана на самом Java (хотя иногда и приходится уходить в нативный код, то есть снова в плюсы). Вообще, так повелось, что системщину пишут на плюсах, ведь здесь, с одной стороны, важна производительность, а с другой — связь с низким уровнем, с операционной системой и железом. В то же время стало очевидно, что далеко не все компоненты виртуальной машины обязательно нужно писать на C++. Например, зачем делать компилятор на C++, если есть замечательные managed-языки вроде Java или Scala?

Вот, например, в Oracle Labs разрабатывает проект GraalVM. Он состоит из трёх компонентов:

- **Компилятор Graal** — мощный оптимизирующий компилятор, в будущем он, возможно, заменит компилятор C2 из HotSpot. Он не уступает по качеству многих оптимизаций, но написан полностью на Java.
- **Truffle Framework** — специальный фреймворк для написания интерпретаторов других языков, которые потом запускаются на GraalVM. По сути, это быстрый способ реализовать любой язык программирования на базе GraalVM! Опять-таки, интерпретаторы пишутся на managed-языках.
- **Graal Native Image** — отдельный режим сборки приложений, который на самом деле представляет из себя целую виртуальную машину, но написанную с некоторыми нарушениями спецификации. Зато она позволяет получить быстрый стартап и низкое потребление памяти. Рантайм для такой виртуальной

машины тоже частично написан на Java! Понятно, что не полностью и с разными ограничениями языка (на самом деле там используется отдельный диалект — System Java), но тем не менее это возможно.

Вообще, писать виртуальные машины на managed-языках — давняя мечта системщиков. Ведь, например, компилятор C написан на C? Вот и разработчики виртуальных машин для managed-языков хотят писать их на Java или других managed-языках, а не на опасном и болезненном C++.

Как контрибьютить в OpenJDK

Для начала скажу, что я никогда не контрибьютил в OpenJDK, а занимался разработкой другой, проприетарной и закрытой JVM. Поэтому о процедуре коммита в OpenJDK имею только смутное представление, так что написанному ниже доверяйте на свой страх и риск!

Если бы я задался целью контрибьютить в OpenJDK, то думал бы в двух направлениях:

- **Найти что-то небольшое, что получится исправить или улучшить.** Это может быть несложный (и низкоприоритетный) баг в баг-трекере OpenJDK или неэффективность какого-нибудь алгоритма в платформенных классах. Исправьте эту проблему, соберите OpenJDK и проверьте, что, например, алгоритм и правда стал работать лучше на каких-нибудь бенчмарках.

Потом напишите в мейлинг-лист, расскажите о сути улучшения и попросите сделать ревью. Наконец, пройдите его, а затем и остальную процедуру коммита. Насколько я знаю, она довольно стандартная, ведь OpenJDK теперь находится на GitHub. Поздравляю, теперь вы контрибьютор в OpenJDK!

По рассказам людей, которые прошли по этому пути, вся процедура может занять довольно много времени, особенно в первый раз, но в целом это вполне реально.

LLVM vs JVM vs GraalVM

В Kotlin есть отличная возможность — Kotlin Native. Она позволяет компилировать Kotlin код в бинарники под конкретные платформы — которые способны запускаться без виртуальной машины. Эта технология основана на LLVM — Low Level Virtual Machine. Что? Снова виртуальная машина? Не аналог ли это обычной JVM? Или модной молодёжной GraalVM? Давайте разберёмся, в чём же разница между ними.

LLVM

- Регистровая VM (оперирует, в первую очередь, регистрами процессора).

- Очень низкоуровневая, спроектированна для абстрагирования от различий в железе.
- Больше похожа на компилятор, потому что из своего IR (Intermediate Representation) получает бинарник под конкретную платформу.

JVM

- Стэковая ВМ (не пользуясь регистрами, оперирует стэком).
- Предоставляет высокоуровневую инфраструктуру разработки: понятие объектов и ООП, Garbage Collector и др.

GraalVM

- Это не совсем ВМ сама по себе. Это набор компонентов, предоставляющих кучу доп возможностей JVM.
- Поддерживает JIT и AOT компиляцию.
- JIT компилятор просто заменяет имеющийся JIT компилятор в JVM, при этом сама JVM остаётся той же.
- Позволяет запускать LLVM биткод. (В отличие от стандартного его запуска компилирует динамически для interop-совместимости с другими поддерживаемыми языками)

3. Области доминирования Java

Благодаря вышеописанным преимуществам, программы, написанные на *Java*, могут выполняться практически на любых устройствах — компьютерах, телефонах, банкоматах, тостерах, банковских карточках.

Преимуществ такого подхода очень много. Именно поэтому программы на *Android* тоже пишутся на *Java*. А благодаря развитию мобильного сектора, Java занимает доминирующее положение в следующих отраслях программирования:

1. *Enterprise*: тяжелые серверные приложения для банков, корпораций, инвестфондов и т.д.
2. *Mobile*: мобильная разработка (телефоны, планшеты), благодаря Android.
3. *Web*: лидирует PHP, но и Java держит солидный кусок рынка.
4. *Big Data*: распределенные вычисления в кластерах из тысяч серверов.
5. *Smart Devices*: программы для умного дома, электроники, холодильников с выходом в интернет.

Java — это не просто язык, а целая экосистема: миллионы готовых модулей, которые вы можете использовать в своей программе. Тысячи сообществ и форумов в интернете, где можно попросить помощь или совет.

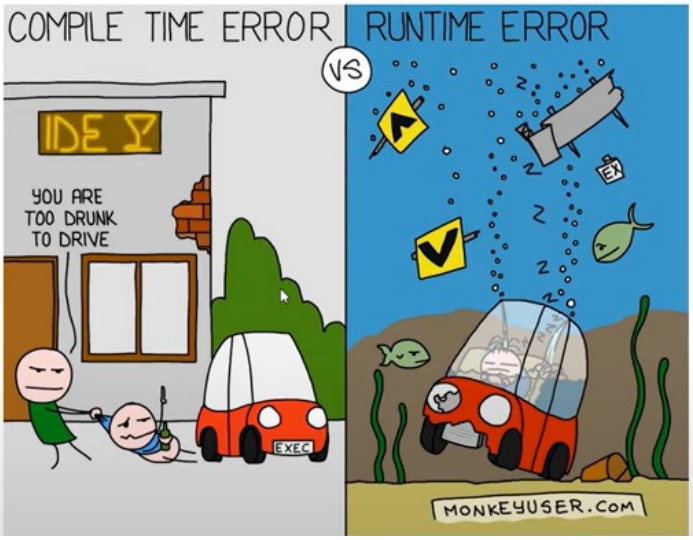
Порядок от старшего к младшему[править | править код]

Порядок *от старшего к младшему* (англ. *big-endian* — с большого конца): . Этот порядок подобен привычному порядку записи (например арабскими цифрами) слева направо , например, число *сто двадцать три* было бы записано при таком порядке как *123*. В этом же порядке принято записывать байты в технической и учебной литературе, если другой порядок явно не обозначен.

Этот порядок является стандартным для протоколов TCP/IP, он используется в заголовках пакетов данных и во многих протоколах более высокого уровня, разработанных для использования поверх TCP/IP. Поэтому порядок байтов от старшего к младшему часто называют «сетевым порядком байтов» (англ. *network byte order*). Этот порядок байтов используется процессорами IBM 360/370/390, SPARC, Motorola 68000 (отсюда третье название — *порядок байтов Motorola*, англ. *Motorola byte order*).

При таком порядке байтов удобно проводить сравнение строк (можно сравнивать их целочисленными полями-частями большей разрядности, каждое из которых содержит несколько символов сразу).

Безопасность в design-time и в run-time



Виртуальная машина

WORA + Две взаимосвязанные задачи:

- Безопасность
- Исключение фатальных сбоев на сервере

Генерируем байт-код, верифицируем и исполняем его в безопасной среде

VM НМ не вылетают

Как например

NAN

Floating point overflow

Порядок от старшего к младшему[править | править код]

Порядок *от старшего к младшему* (англ. *big-endian* — с большого конца): . Этот порядок подобен привычному порядку записи (например арабскими цифрами) слева направо , например, число *сто двадцать три* было бы записано при таком порядке как *123*. В этом же порядке принято записывать байты в технической и учебной литературе, если другой порядок явно не обозначен.

Этот порядок является стандартным для протоколов TCP/IP, он используется в заголовках пакетов данных и во многих протоколах более высокого уровня, разработанных для использования поверх TCP/IP. Поэтому порядок байтов от старшего к младшему часто называют «сетевым порядком байтов» (англ. *network byte order*). Этот порядок байтов используется процессорами IBM 360/370/390, SPARC, Motorola 68000 (отсюда третье название — *порядок байтов Motorola*, англ. *Motorola byte order*).

При таком порядке байтов удобно проводить сравнение строк (можно сравнивать их целочисленными полями-частями большей разрядности, каждое из которых содержит несколько символов сразу).

Сегодняшний день

- Desktop
- Server-side (backend)
- Mobile Devices (Android) — отдельная история (see Google vs. Oracle case)

Сервисы виртуальной машины и платформы

- Byte code, интерпретация и JIT-компиляция.
- Garbage Collection
- MultiThreading & Memory Model
- Загрузка и исполнение кода в runtime
- Рефлексия
- Стандартная библиотека

Рефлексия

- Исследование информации о классах, полях, методах и т. д. во время выполнения
- Чтение/модификация полей (в том числе private!) и динамический вызов методов (в том числе private!)
- Proxy и AOP

Рефлексия в Java осуществляется с помощью Java Reflection API. Что такое эта рефлексия? Существует короткое и точное, а также популярное на просторах интернета определение. **Рефлексия** *(от позднелат. reflexio — обращение назад)* — это механизм исследования данных о программе во время её выполнения. Рефлексия позволяет исследовать информацию о полях, методах и конструкторах классов. Сам же механизм рефлексии позволяет обрабатывать типы, отсутствующие при компиляции, но появившиеся во время выполнения программы. Рефлексия и наличие логически целостной модели выдачи информации об ошибках дает возможность создавать корректный динамический код. Иначе говоря, понимание принципов работы рефлексии в java открывает перед вами ряд удивительных возможностей. Вы буквально можете жонглировать классами и их составляющими.

Вот основной список того, что позволяет рефлексия:

- Узнать/определить класс объекта;
- Получить информацию о модификаторах класса, полях, методах, константах, конструкторах и суперклассах;
- Выяснить, какие методы принадлежат реализуемому интерфейсу/интерфейсам;
- Создать экземпляр класса, причем имя класса неизвестно до момента выполнения программы;
- Получить и установить значение поля объекта по имени;
- Вызвать метод объекта по имени.

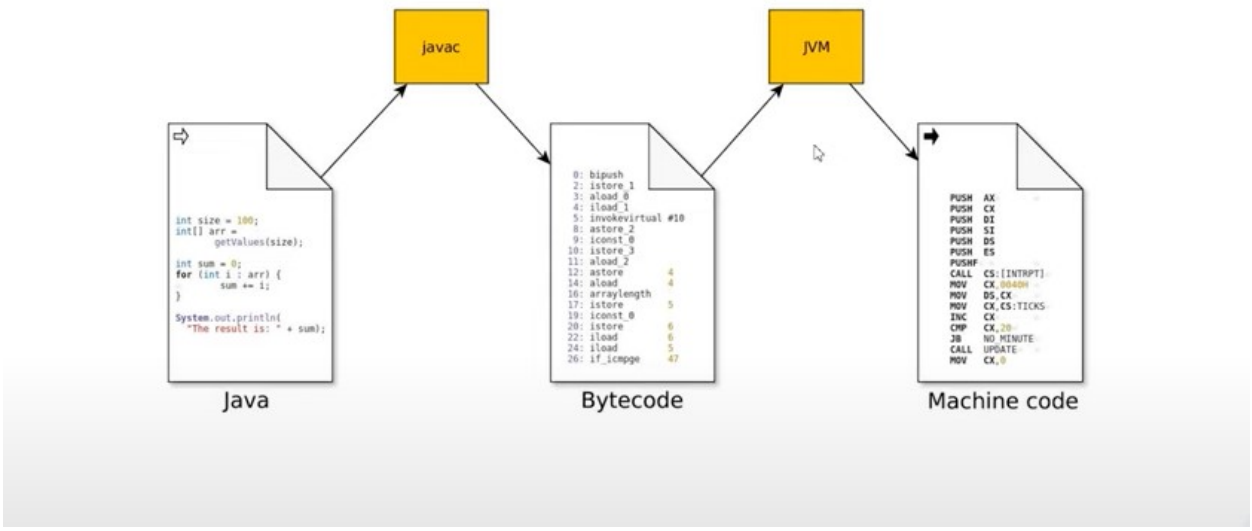
Рефлексия используется практически во всех современных технологиях Java. Сложно себе представить, могла бы Java, как платформа, достигнуть такого огромного распространения без рефлексии. Скорее всего не смогла бы.

Верификация байт-кода

https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10

- There are no operand stack overflows or underflows.
- All local variable uses and stores are valid.
- The arguments to all the Java Virtual Machine instructions are of valid types.

Интерпретация и JIT-компиляция



Статистика

Большинство реализаций JIT имеет последовательную структуру: сначала приложение компилируется в байт-код виртуальной машины среды исполнения (AOT-компиляция), а потом JIT компилирует байт-код непосредственно в машинный код. В итоге при запуске приложения тратится лишнее время, что впоследствии компенсируется более быстрой его работой.

1. Компиляция может осуществляться непосредственно для целевого процессора и операционной системы, на которой запущено приложение. Например, JIT может использовать векторные SSE2 расширения процессора, если он обнаружит их поддержку.
2. Среда может собирать статистику о работающей программе и производить оптимизации с учётом этой информации. Некоторые статические компиляторы также могут принимать на вход информацию о предыдущих запусках приложения.
3. Среда может делать глобальные оптимизации кода (например, встраивание библиотечных функций в код) без потери преимуществ динамической компиляции и без накладных расходов, присущих статическим компиляторам и компоновщикам.
4. Более простое перестраивание кода для лучшего использования кэша.

Code cache

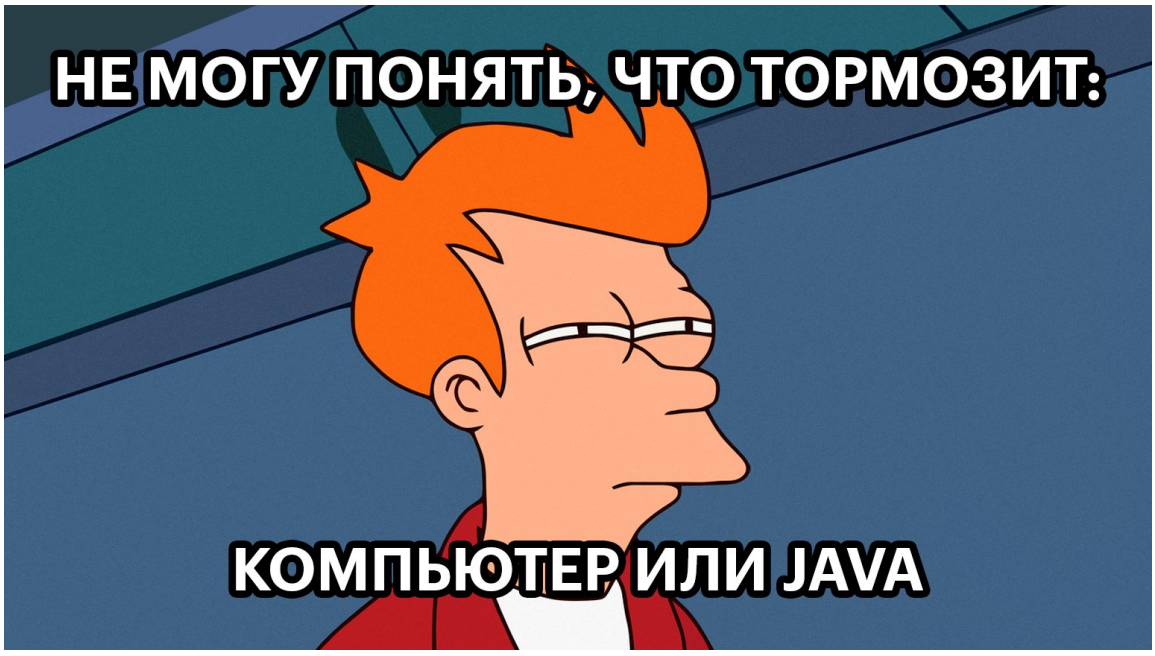
Garbage Collection

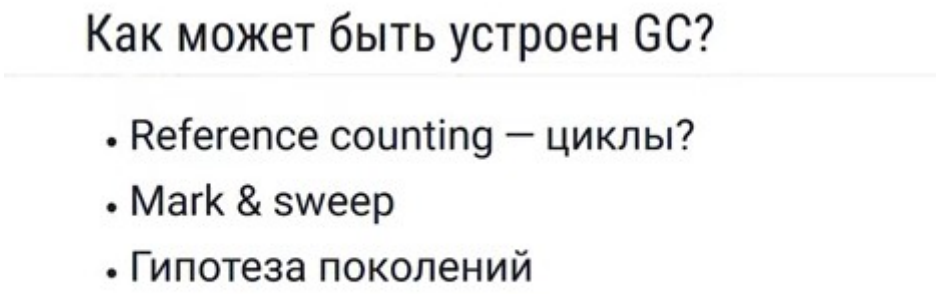
- Разработчику не нужно заботиться о высвобождении памяти
- Ссылкок на объект нет? — garbage collection
- Достоинства:
  - Меньше кода
  - Не бывает некорректных ссылок (на разрушенный объект)
  - Меньше утечек памяти
- Недостатки:
  - Не контролируем время работы GC
  - STW-паузы (бывает, в неподходящий момент)

Не надо писать

Инструкция try-finally — это расширение Майкрософт на языках C и C++, которые позволяют целевым приложениям гарантировать выполнение кода очистки при прерывании выполнения блока кода. Очистка включает такие задачи, как отмена распределения памяти, закрытие файлов и освобождение их дескрипторов.

Нет ссылок на разрушенные объекты



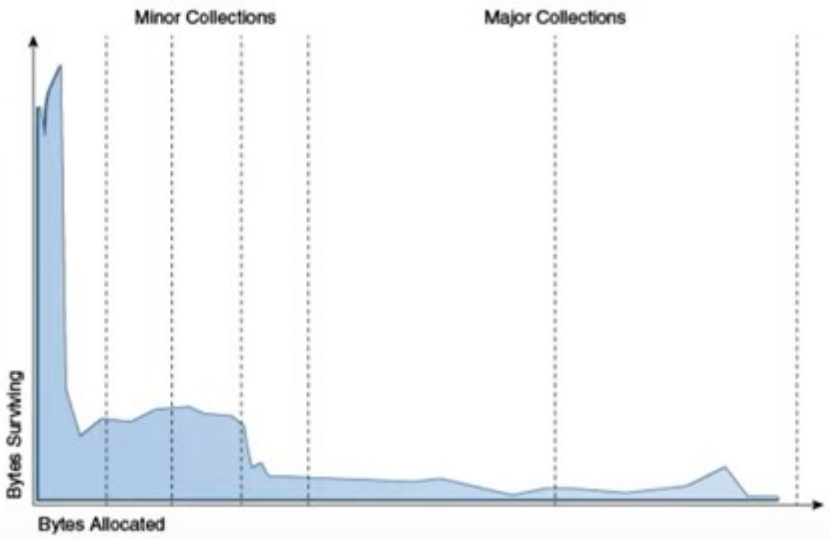


Коротко о Mark-Sweep

Чтобы чистить ненужные объекты GC делает так:

1. Обходит граф всех объектов в хипе (heap) и помечает их
2. Не помеченные объекты никто не использует - их можно удалить.

Гипотеза поколений



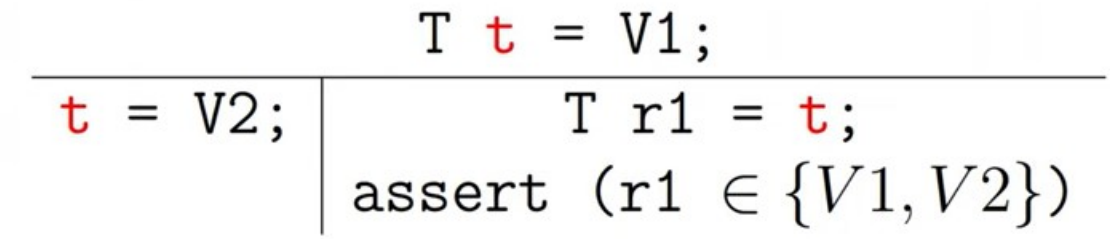
Типы коллекторов

- Serial (for single-threaded, client environments)
- Parallel (default)
- CMS (Concurrent mark and Sweep — shorter pauses, share processor resources)
- G1 (Garbage First — big heaps)
- Shenandoah (submillisecond pauses)
- Epsilon (no GC)

Модель памяти

- Memory Model: часть спецификации VM: модель того, как работает хранилище данных
- Отвечает на вопрос: "какие значения может прочитать конкретный read в программе?"

Access Atomicity



Многопоточность

- Возможность запуска параллельных потоков выполнения встроена в язык с первой версии (`Thread`, `Runnable` etc)
- В версии 1.5 существенные дополнения (executor, semaphore, mutex, barrier, latches, concurrent collections, blocking queues)

Загрузка и исполнение кода в runtime

- Скомпилированный байт-код класса может быть доставлен на любую систему в runtime (чаще всего "подкладыванием файла" на файловую систему).
- Пересборка/линковка не требуется.

JAR-файлы — что это такое ?

JAR расшифровывается как Java ARchive — архив Java. Если не особо вдаваться в подробности (что в общем-то часто и происходит в реальности), то JAR-файл представляет собой обычный ZIP-файл с некоторыми дополнениями. Основная задача этого архива — хранить файлы с классами. Т.е. пакеты/каталоги (мы о них говорили в разделе Пакеты), внутри которых находятся class-файлы архивируются и JVM может их использовать уже в более удобном (компактном) виде.
Выгода очевидна — в реальных проектах вы вряд ли будете ограничиваться использованием библиотек классов только из JDK — скорее всего вы будете подключать сотни (если не тысячи) классов, которые были созданы для решения определенных задач. Если все это огромное количество лежало бы на диске в виде class-файлов, то несложно понять, насколько это неудобно и громоздко. Разработчики Java предложили вполне элегантное решение — вы можете подключать архивные файлы, в которых запакованы тысячи скомпилированных классов. Что конечно же удобнее, чем таскать за собой в каждый проект все директории со всеми файлами.
Для примера можете заглянуть в каталог JAVA_HOME/jre/lib — там находится очень важный архив — rt.jar. Эта библиотека содержит практически все классы, который мы рассматривали — String, JFrame, JButton, Date, Object. Открыть его вы можете с помощью любого архиватора.
Просматривая содержимое JAR-файла, вы рано или поздно обратите внимание на каталог META-INF. В нем содержится файл MANIFEST.MF. Я настоятельно рекомендую вам почитать информацию о нем в документе Working with Manifest Files: The Basics. Файл позволяет расширить функциональность — кроме обычного набора классов, JAR-файл может выполнять функции электронной подписи, поддержки версионности, подключения библиотек к самому архиву, определение класса для запуска (который содержит метод main). В данной статье я не ставлю себе цель разборать эти возможности, так что отложим на будущее. Займемся базовыми вопросами использования JAR-файлов.

Сериализация объектов, встроенная в язык

- Идея, в начале казавшаяся очень хорошей!

Сериализация (в программировании) — процесс перевода структуры данных в битовую последовательность. Обратной к операции сериализации является операция *десериализации* (структуризации) — создание структуры данных из битовой последовательности.

Сериализация используется для передачи объектов по сети и для сохранения их в файлы. Например, нужно создать распределённое приложение, разные части которого должны обмениваться данными со сложной структурой. В таком случае для типов данных, которые предполагается передавать, пишется код, который осуществляет сериализацию и десериализацию.

Стандартная библиотека

- Независимая от системы часть
  - Структуры данных
  - Математика
- Зависимая от системы часть
  - Дата и время
  - Доступ к файловой системе
  - Доступ к сети
  - Пользовательский интерфейс

Стандартная библиотека

Version	Year	New Language Features	Number of Classes and Interfaces
1.0	1996	The language itself	211
1.1	1997	Inner classes	477
1.2	1998	The strictfp modifier	1,524
1.3	2000	None	1,840
1.4	2002	Assertions	2,723
5.0	2004	Generic classes, "for each" loop, varargs, autoboxing, metadata, enumerations, static import	3,279
6	2006	None	3,793
7	2011	Switch with strings, diamond operator, binary literals, exception handling enhancements	4,024
8	2014	Lambda expressions, interfaces with default methods, stream and date/time libraries	4,240

Обратная совместимость

- Любой, как угодно давно скомпилированный .jar-файл может быть запущен на любой современной JVM
- Дар или проклятие?

В данном списке представлены языки программирования, которые используются для создания программного обеспечения, использующего в качестве среды выполнения виртуальную машину Java (JVM). Некоторые из этих языков интерпретируются, а некоторые компилируются в байт-код Java и компилируются «на лету» во время исполнения.

JVM была изначально создана для поддержки исключительно языка программирования Java. Однако, с течением времени, некоторые языки были адаптированы или созданы для исполнения на платформе Java.

Языки, изначально созданные для JVM:

- Clojure — функциональный язык, диалект Lisp;
- Groovy — сценарный язык;
- Kotlin — объектно-ориентированный язык для индустриальной разработки
- Scala — объектно-ориентированный и функциональный язык;
- Ceylon — объектно-ориентированный язык со строгой статической типизацией;
- JRuby — реализация Ruby;
- Jython — реализация Python;
- Nashorn — реализация JavaScript.

Реализация существующих языков программирования:

Язык	Реализация
Ада	JGNAT
awk	Jawk[1]
Бейсик	jScriptBasic — реализация Java для языка ScriptBasic.
BBx	BBj — расширенный BBx, PRO/5 и Visual PRO/5.
Boo	Boojay
Си	различные компиляторы с языка Си для JVM[2]

Язык	Реализация
Кобол	Elastic COBOL Micro Focus Visual COBOL Veryant isCobol
ColdFusion	Adobe ColdFusion Railo Open BlueDragon
Common Lisp	Armed Bear Common Lisp[3] CLforJava Jatha Common Lisp Library
Component Pascal	Gardens Point Component Pascal
Eiffel	liberty-eiffel[4]
Erlang	Erjang[5]
Forth	myForth[6]
Go	jgo[7]
JavaScript	Rhino Nashorn GraalVM
Logo	jLogo[8] XLogo[9]
Lua	Kahlua[10] Luaj[11] Jill[12]
Оберон-2	Canterbury Oberon-2 for JVM JOB
OCaml	OCaml-Java[13]

Язык	Реализация
Object Pascal	Oxygene
Паскаль	Canterbury Pascal for JVM Free Pascal MIDletPascal
PHP	IBM WebSphere sMash PHP (P8)[14] Caucho Quercus[15] JPHP
Пролог	JIProlog Jekejeke Prolog JLog TuProlog Jinniprolog
Python	Jython
R	renjin
REXX	NetRexx
Ruby	JRuby
Scheme	Bigloo Kawa SISC JScheme
Tcl	Jacl JTcl[16]

Spring Framework

DI container and application framework,

Среды разработки

- IDEA (proprietary, JetBrains)
- Eclipse (Eclipse Foundation)
- NetBeans (ASF / Oracle)