Компиляторы, Интерпретаторы и Байт-код

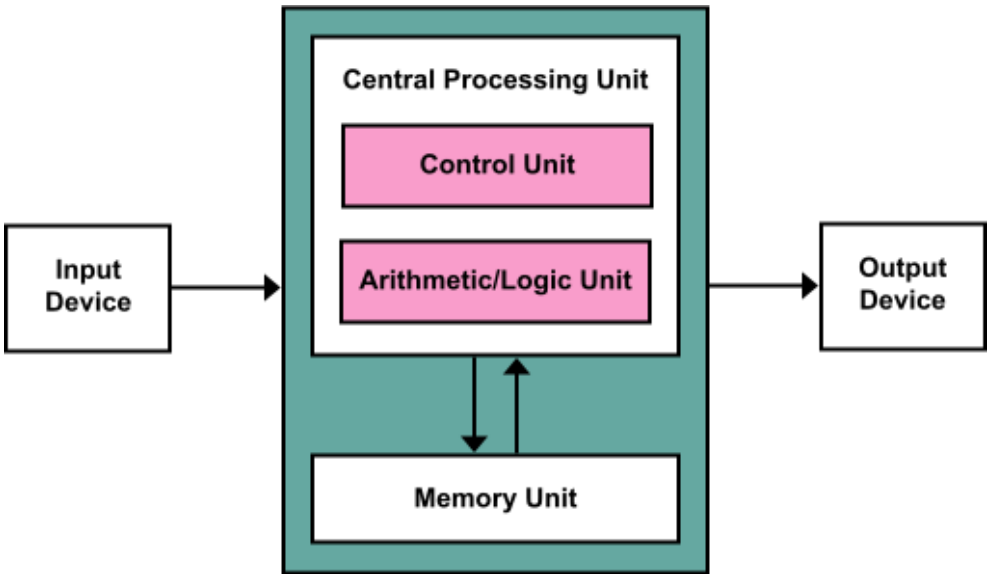
- Архитектуры компьютеров
- Байт-код
- Виртуальная Машина
- Внутренний процесс трансляции
- Компилятор и Интерпретатор, JIT
- CPython / PVM / Cython

Архитектура компьютера

Архитектура фон Неймана и **архитектура Гарварда** — это две основные модели компьютерной архитектуры, которые описывают, как организованы память и процессорные устройства компьютера. Эти архитектуры являются основой большинства современных вычислительных систем и их понимание важно для дальнейшего перехода к трансляторам.

Архитектура фон Неймана

также известная как Принстонская архитектура, была впервые описана математиком и физиком Джоном фон Нейманом в 1940-х годах. Эта модель описывает архитектуру компьютера, в которой и программные инструкции, и данные хранятся в одной и той же области памяти.



Основные характеристики

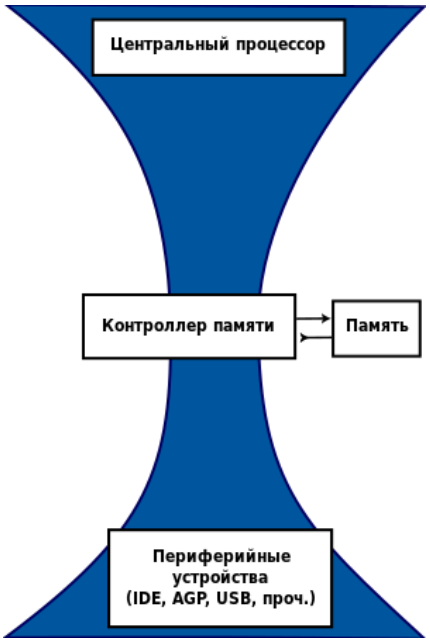
- **Единое пространство памяти:** Инструкции (код) и данные хранятся в одной и той же памяти. Это означает, что процессор извлекает и инструкции, и данные из одной общей памяти.
- **Последовательное выполнение**: Инструкции извлекаются из памяти и выполняются одна за другой в последовательном порядке. Следующая инструкция извлекается только после выполнения текущей.
- **Управляющее устройство и ALU (Arithmetic Logic Unit)**: Процессор обычно включает управляющее устройство, которое интерпретирует инструкции, и ALU, которое выполняет математические операции. Оба устройства используют общий шину памяти для доступа к инструкциям и данным.
- **Бутылочное горлышко фон Неймана**: Существенное ограничение заключается в том, что инструкции и данные не могут извлекаться одновременно, поскольку они используют один и тот же путь передачи данных. Это создает узкое место, ограничивая скорость выполнения инструкций процессором, так как ему постоянно приходится переключаться между доступом к данным и извлечением инструкций.

Преимущества

- **Простота модели:** Хранение данных и инструкций в одном месте упрощает управление памятью. Не нужно думать о разделении или организации различных типов памяти, что уменьшает сложность системы. Процессор использует одну шину для доступа к обоим типам информации.
    - **Пример:** Представьте, что у вас есть книга рецептов, в которой и сами рецепты (инструкции), и ингредиенты (данные) хранятся на одной полке. Вам не нужно идти в два разных места, чтобы получить инструкции по приготовлению и ингредиенты — всё, что нужно, находится вместе, в одном удобном месте.
- **Экономичность:** Использование одной общей памяти и единых каналов передачи данных означает, что нет необходимости в установке дополнительных модулей памяти или сложных маршрутов передачи данных. Это сокращает количество аппаратных деталей,что уменьшает затраты на производство таких систем.
- **Гибкость:** Гибкость архитектуры заключается в том, что единая модель памяти упрощает проектирование и программирование универсальных компьютеров. Благодаря этому одна и та же архитектура может справляться с разными типами задач и инструкций без необходимости использования специализированного оборудования для каждой операции.

Недостатки

- **Bottleneck**: Совместное использование шины (канала передачи данных) для инструкций и данных создает узкое место, ограничивая доступ к памяти, так как в каждый момент можно получить доступ только к одному элементу данных или инструкции, что снижает производительность процессора. Это решается усовершенствованием систем кэширования, что усложняет архитектуру и увеличивает риск ошибок, таких как проблема когерентности памяти.
    - **Кеширование** — эффективная стратегия управления памятью, используемая для повышения производительности компьютерных систем. Оно временно сохраняет часто используемые данные в быстро доступной памяти, что сокращает время доступа к ним и улучшает общую скорость работы системы.
        - **Кэш процессора (L1, L2, L3):** Это многоуровневые кеши, встроенные в сам процессор или расположенные рядом с ним. L1 кэш самый быстрый, но самый маленький. L2 и L3 кеши больше по размеру, но немного медленнее.
            - Почему нельзя создать кэш размером 1ТБ?



- **Стоимость**: Кэш-память основана на технологиях быстрой памяти (обычно SRAM: static random-access memory), которые значительно дороже в производстве, чем стандартная DRAM, используемая для основной памяти (RAM).
- **Энергопотребление**: Кэш-память потребляет больше энергии, чем основной тип оперативной памяти. Большой кэш, особенно размером 1ТБ, привел бы к значительному увеличению потребления энергии.
- **Замедление доступа**: Несмотря на то что кэш быстрее оперативной памяти, увеличение его объема до терабайта сделало бы его менее эффективным. Доступ к кэшу требует поиска нужных данных, и чем больше кэш, тем дольше будет процесс поиска нужной информации.
- **Управление кэшем**: С увеличением размера кэша сложность управления данными и координации между кэшем и оперативной памятью также возрастает. Для работы с кэшем необходимы алгоритмы вытеснения (например, LRU — Least Recently Used), которые отслеживают, какие данные должны быть сохранены, а какие — удалены. В кэше объемом 1ТБ эти алгоритмы станут сложнее и будут занимать больше ресурсов.

- Зачем нужны три уровня кэша (L1, L2, L3)?
  Три уровня кэша используются для оптимального баланса между скоростью, стоимостью и размером.
    - **L1 (самый быстрый, но самый маленький)**: L1-кэш находится ближе всего к процессору и работает на очень высокой скорости, почти на той же частоте, что и сам процессор. Однако его размер ограничен (обычно от 32КБ до 128КБ), так как он должен быть очень быстрым и компактным. Этот кэш используется для хранения данных, к которым процессор обращается чаще всего.
    - **L2 (больше и медленнее)**: L2-кэш больше по размеру (обычно от 256КБ до 1МБ) и медленнее, чем L1, но всё равно значительно быстрее основной памяти (RAM). Он используется для хранения данных, к которым процессор обращается не так часто, как к данным в L1, но всё же достаточно регулярно, чтобы не загружать их из основной памяти.
    - **L3 (самый медленный и самый большой)**: L3-кэш имеет наибольший объем (обычно от нескольких мегабайт до десятков мегабайт) и является самым медленным из всех уровней кэша, но всё ещё значительно быстрее основной памяти. Он служит общим буфером для всех ядер процессора и хранит данные, которые могут быть полезны для выполнения задач в ближайшем будущем, но еще не были загружены в L1 или L2.

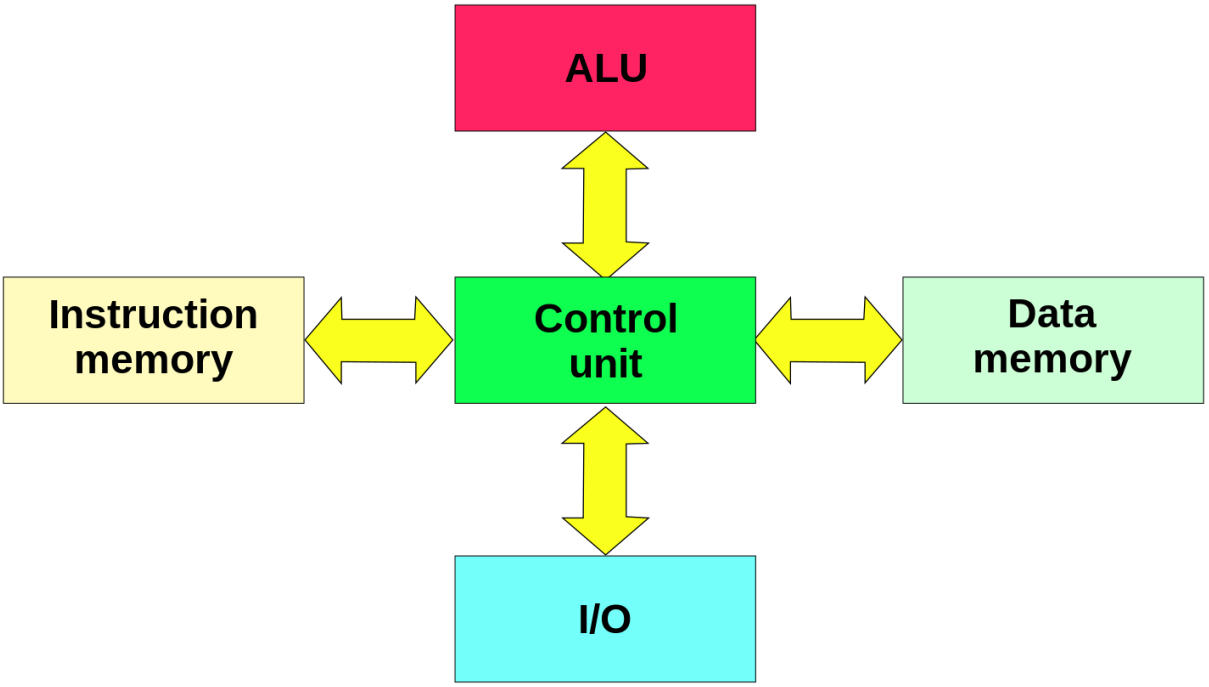
  - **Когерентность памяти** – это свойство многопроцессорных систем, где изменения в ячейке памяти, сделанные одним ядром, становятся видимыми для других. В одноядерных системах изменения видны сразу. В многопроцессорных системах используется протокол когерентности, чтобы обновления одним ядром синхронизировались с кешами других ядер, обеспечивая согласованность данных.
- **Риски безопасности:** Совместное хранение кода и данных в одной памяти упрощает вредоносному коду изменение инструкций программы или данных: Injection Attacks, Buffer Overflow и другие.

Где используется

- Архитектура фон Неймана широко используется в компьютерах общего назначения, таких как настольные компьютеры, ноутбуки и серверы.
- Большинство современных процессоров, включая процессоры x86, основаны на модели фон Неймана.

Гарвардская архитектура

альтернативная архитектура компьютера, в которой инструкции и данные хранятся и передаются отдельно. Она появилась на основе компьютера Harvard Mark I, где память для инструкций и данных была разделена.



Пример:
Представьте, что процессор выполняет программу, в которой необходимо вычислить сумму двух чисел. Инструкции (например, сложение) будут извлекаться из памяти инструкций, а сами числа (данные) — из памяти данных. Эти два типа информации поступают в CPU через разные шины, что позволяет процессору эффективно параллелить работу с инструкциями и данными.

Основные характеристики

- **Раздельная память:** Инструкции (код) и данные хранятся в разных местах (память) и используют отдельные шины (каналы передачи данных) и адресные пространства (диапазон адресов памяти).

Преимущества

- **Параллельность и скорость:** CPU может одновременно получать инструкции и данные, устраняя узкие места, как в архитектуре фон Неймана, что позволяет быстрее обрабатывать информацию, поскольку не нужно ждать доступа к памяти.
- **Большая безопасность:** поскольку инструкции и данные хранятся отдельно, вредоносному коду сложнее изменить инструкции программы, что обеспечивает более безопасное выполнение.
- **Специализация:** различные архитектуры памяти и методы доступа могут быть оптимизированы для инструкций и данных независимо друг от друга, что обеспечивает большую гибкость и настройку производительности.

Недостатки

- **Сложность**: конструкция требует отдельных блоков памяти и шин для инструкций и данных, что увеличивает сложность и стоимость.
- **Меньшая гибкость:** изменение набора инструкций или конфигурации хранилища данных может быть более сложным и менее гибким по сравнению с архитектурой фон Неймана.

Где используется

- Широко используется во встраиваемых системах, микроконтроллерах и цифровых сигнальных процессорах (DSP), где эффективность и скорость имеют решающее значение.
- Распространено в средах обработки в реальном времени, где требуется предсказуемое выполнение и высокая производительность.

Сравнение архитектур

Feature	Von Neumann	Harvard
Memory Structure	Единая память для инструкций и данных	Раздельные памяти для инструкций и данных
Data Path	Общий путь передачи данных для инструкций и данных	Отдельные пути передачи данных для инструкций и данных
Execution Speed	Ограничено узким местом фон Неймана	Быстрее за счет параллельного доступа
Design Complexity	Более простая и экономичная конструкция	Более сложная и дорогая конструкция
Security	Менее безопасно; общая память может быть взломана	Более безопасно; отдельные области памяти защищают целостность
Flexibility	Более гибкий для вычислений общего назначения	Специализируется на конкретных задачах
Use Cases	Компьютеры общего назначения (настольные компьютеры, ноутбуки, серверы)	Встроенные системы, микроконтроллеры, DSP
Examples	x86, ARM (процессоры общего назначения)	AVR, ARM Cortex-M (микроконтроллеры), DSP

Выводы

- **Архитектура фон Неймана** подходит для вычислений общего назначения благодаря своей простоте и экономической эффективности. Однако она страдает от узкого места фон Неймана, которое может ограничивать производительность.
- **Гарвардская архитектура** более эффективна для конкретных задач, требующих высокой производительности и обработки в реальном времени, благодаря раздельным путям памяти и данных. Она широко используется в специализированных приложениях, таких как встроенные системы и DSP.

Язык и компьютер

Компьютеры — это комбинация аппаратного и программного обеспечения. Аппаратное обеспечение (**hardware**) — это физические компоненты компьютера, такие как процессор, память, жесткий диск и т.д. Эти компоненты являются механическими и электронными устройствами, которые выполняют определенные функции. Однако, чтобы аппаратное обеспечение могло выполнять задачи, оно должно быть управляемо программным обеспечением (**software**).

Программное обеспечение представляет собой набор инструкций, которые управляют аппаратными компонентами. Эти инструкции передаются компьютеру в виде последовательности электронных зарядов, которые мы отображаем в **двоичном представлении** символами 0 и 1. Вся информация, которую компьютер получает или отправляет, представлена в виде длинной последовательности 0 и 1.

Почему мы используем языки программирования?

Написание программ в двоичном коде (только 0 и 1) было бы очень сложным и трудоемким. Чтобы сделать процесс программирования более удобным и понятным для людей, были разработаны языки программирования высокого уровня, такие как Python, Java, C++. Эти языки используют синтаксис, который ближе к человеческому языку, что упрощает написание и понимание кода.

Каждый язык программирования имеет свою собственную **грамматику** (правила построения предложений), **синтаксис** (правила написания кода) и другие особенности, которые определяют, как правильно писать код, чтобы компьютер мог его понять.

Пример кода на языке высокого уровня (Python):

```
def hello_world():
    print("Hello, World!")
hello_world()
```

Этот код выводит текст "Hello, World!" на экран. Он написан на понятном языке программирования, но компьютер не может выполнить его напрямую.

Программы, написанные на языке высокого уровня, не могут быть непосредственно поняты компьютером. Эти программы проходят через процесс трансляции. Программы часто сначала компилируются в промежуточный формат, называемый байт-кодом. Байт-код — это более простой и оптимизированный код, который затем интерпретируется виртуальной машиной (VM). Виртуальная машина берет байт-код и выполняет его, используя ресурсы компьютера.

Байт-код

это промежуточное представление программы, находящееся между исходным кодом на высокоуровневом языке, таком как Python или Java, и машинным кодом, который может выполняться процессором напрямую. Байт-код представляет собой набор инструкций, более простых и оптимизированных по сравнению с исходным кодом, что делает их легче интерпретировать и выполнять.

Python

```
import dis

def hello_world():
    print('Hello, World!')

dis.dis(hello_world)
```

Байт-код

```
  2           0 LOAD_GLOBAL              0 (print)
              2 LOAD_CONST               1 ('Hello, World!')
              4 CALL_FUNCTION            1
              6 RETURN_VALUE
```

Байт-код не зависит от конкретной архитектуры компьютера [x86, ARM], однако сам по себе не может напрямую исполняться процессором, так как не является машинным кодом. Для его выполнения требуется специальная программа — **виртуальная машина.**

Virtual Machine

Virtual Machine (VM) – программная эмуляция физического компьютера:

- действует как промежуточный слой между байт-кодом (промежуточным представлением программы) и оборудованием (физическим процессором и памятью)
- интерпретирует инструкции байт-кода одну за другой
- преобразует инструкции в действия, понятные оборудованию

Instruction	Action		Effect
`0 LOAD_GLOBAL 0 (print)`	Загрузить `print` функцию в стек		Подготовка к вызову `print` функции
`2 LOAD_CONST 1 ('Hello, World!')`	Загрузить константу `'Hello, World!'`		Подготовить аргумент для `print`

4 CALL_FUNCTION 1	Вызвать print с 1 аргументом		Вывести Hello, World! в console
6 RETURN_VALUE	Конец выполнения функции		Отметить завершение сценария

- управляет распределением памяти и сборкой мусора (**garbage collector**), гарантируя, что программа имеет необходимые ей ресурсы, избегая при этом утечек памяти *(когда данные становятся доступными для неавторизованных пользователей или приложений без намерения или осведомленности владельца данных)*
- предоставляет изолированную среду для программ, что предотвращает прямое взаимодействие вредоносного кода с ресурсами хост-системы. Например, запуск Python программы на Python VM гарантирует, что программа не сможет получить доступ к системным файлам и сетевым ресурсам без явного разрешения.

Внутренний процесс:

Python

```
a = 5
b = 10
c = a + b

print(c)
```

1. **Лексический анализатор (Lexer):**
   Лексический анализатор разбивает код на токены – элементарные компоненты программы, такие как ключевые слова, идентификаторы (имена переменных), операторы и литералы *(константы, включаемые непосредственно в текст программы)*.

```
a (идентификатор), = (оператор присваивания), 5 (целое число),
b (идентификатор), = (оператор присваивания), 10 (целое число),
c (идентификатор), = (оператор присваивания), a (идентификатор), + (оператор
сложения), b (идентификатор),
print (функция), ( (открывающая скобка), c (идентификатор), ) (закрывающая скобка)
```

2. **Синтаксический анализатор (Parser)**:
   Затем синтаксический анализатор строит абстрактное синтаксическое дерево (AST) на основе этих токенов. AST представляет структуру программы и иерархию операций. Это дерево помогает точно определить последовательность выполнения программы и выявить логические связи между различными компонентами программы. AST делает последующую компиляцию в байт-код более управляемой и эффективной.

```
Assign (присваивание)
├── Name (a)
└── Constant (5)

Assign (присваивание)
├── Name (b)
└── Constant (10)

Assign (присваивание)
├── Name (c)
└── BinOp (бинарная операция)
    ├── Name (a)
    └── Name (b)

Call (вызов функции)
```

```
├── Name (print)
└── Name (c)
```

3. **Семантический анализ (Semantic Analysis):**
   Этот шаг проверяет AST на логическую согласованность и смысл. Он гарантирует, что операции имеют смысл в контексте (например, проверка типов, обеспечение объявления переменных перед использованием). В CPython некоторые семантические проверки могут перемежаться во время синтаксического анализа.

4. **Компиляция в байт-код**:
   Транслятор (CPython) внутренним компилятором создаёт байт-код на основе AST. В CPython байт-код хранится в файлах `.pyc` (скомпилированные Python файлы).

```
1           0 LOAD_CONST               1 (5)
            2 STORE_NAME               0 (a)
2           4 LOAD_CONST               2 (10)
            6 STORE_NAME               1 (b)
3           8 LOAD_NAME                0 (a)
           10 LOAD_NAME                1 (b)
           12 BINARY_ADD
           14 STORE_NAME               2 (c)
4          16 LOAD_NAME                3 (print)
           18 LOAD_NAME                2 (c)
           20 CALL_FUNCTION            1
           22 POP_TOP
           24 LOAD_CONST               0 (None)
           26 RETURN_VALUE
```

Этот байт-код — это последовательность инструкций, которые говорят интерпретатору Python, какие действия выполнять. Например, `LOAD_CONST` загружает константу в стек, `STORE_NAME` сохраняет значение в переменную, а `CALL_FUNCTION` вызывает функцию.

5. **Исполнение байт-кода виртуальной машиной**:
   Виртуальная машина (PVM, являющийся частью CPython) интерпретирует по отдельности инструкции байт-кода и апеллирует к полю стандартных команд в CPU, где и происходит исполнение. Виртуальная машина также может включать механизмы управления памятью (например, сборку мусора) и поддерживать JIT-компиляцию, чтобы ускорить выполнение путем динамической компиляции байт-кода в машинный код.

Аналогия для понимания

Представьте, что вы на кухне ресторана, где разные роли и инструменты используются для того, чтобы взять рецепт (исходный код) и превратить его в блюдо (вывод программы):

- **Исходный код** = рецепт на родном языке (Python).
- **Транслятор** (CPython) = подготовительное бюро организующее готовку.
  - **Lexer** = персонал по подготовке ингредиентов и осмыслению действий: овощи, мясо, специи, нарезка, измельчение, …
  - **Parser** = человек-конструктор, преобразует список ингредиентов и действий в пошаговый план.
    - **AST** = структурированное представление рецепта, пошаговый план, дорожная карта.
  - **Семантический анализ** = проверяющий, подтверждает последовательность и логичность.
  - **Компилятор** (в CPython) = переводчик рецептов, берет план рабочего процесса и преобразует его в набор универсальных инструкций по приготовлению пищи понятный любому повару вне зависимости от происхождения.
    - **Байт-код** = универсальные инструкции по приготовлению пищи

    - Виртуальная машина (в CPython):
        - **Интерпретатор** = человек-коммуникатор, поочерёдно читающий универсальные инструкции и отсылающий инструкции на понятном языке иностранным поварам на кухне, привыкший работать на определённом типе кухонь.
- **CPU** = кухня с оборудованием для приготовления и собственной командой, которая на вход получает …

Резюме

Портативность:

- Программы на байт-коде могут выполняться на любом устройстве с соответствующей виртуальной машиной, обеспечивая кроссплатформенную совместимость, в отличие от двоичного кода, который разный под разные архитектуры. Например, Java-код может быть написан один раз и выполнен везде, где есть JVM.

Безопасность:

- Виртуальные машины предоставляют песочницу (sandbox), ограничивающую доступ к критическим системным ресурсам. Это делает выполнение кода более безопасным, так как VM может отслеживать и контролировать доступ программы к файловой системе, памяти и сетевым ресурсам.

Управление памятью:

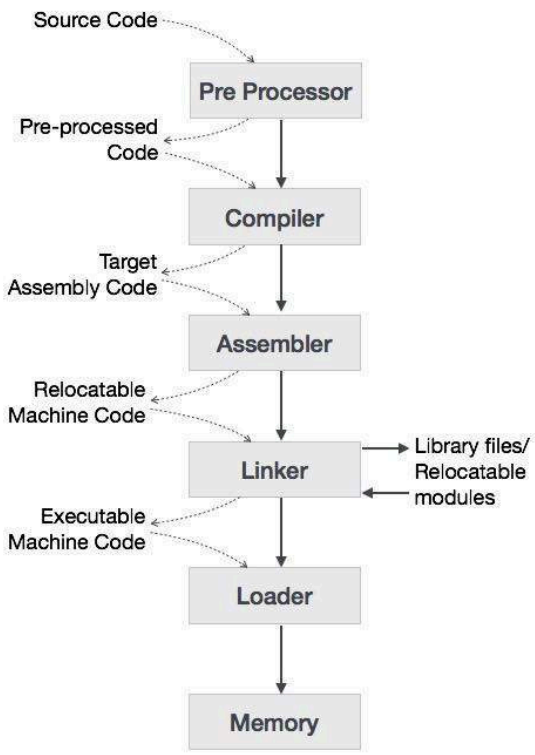
- Виртуальные машины часто включают в себя автоматический сбор мусора, что упрощает управление памятью для разработчиков и снижает вероятность утечек памяти и ошибок сегментации.

Легкость отладки и анализа:

- Байт-код содержит больше информации, чем машинный код, что делает его удобным для отладки и анализа выполнения программ. Виртуальные машины могут предоставлять подробные отчеты об ошибках и состояния выполнения.

Использование виртуальных машин в Python

Большинство интерпретаторов Python, включая CPython, PyPy и MicroPython, используют виртуальную машину для выполнения байт-кода Python. Это обеспечивает эффективное выполнение Python-кода на разных платформах. В некоторых реализациях, таких как Jython и IronPython, Python-код компилируется в байт-код, совместимый с существующими VMs (JVM и CLR (VM for .NET libraries) соответственно), что также обеспечивает кроссплатформенную поддержку.



Компилятор и Интерпретатор

В чём разница и что это такое?

Компилятор

это программа, которая преобразует исходный код, написанный на одном языке программирования, в эквивалентный код на другом языке программирования. Например, компилятор преобразует код на языке высокого уровня (C или Java, Python: ?Nuitka, GraalPython, Cinder) в машинный код, который может быть выполнен процессором.

Компилятор читает весь исходный код сразу, создает токены, проверяет семантику, генерирует промежуточный код, выполняет всю программу.

Давайте сначала поймем, как программа, использующая компилятор, выполняется на хост-машине:

- **high-level language**: пользователь пишет программу на языке высокого уровня
- **low-level language**: компилятор компилирует программу и переводит её в ассемблер
- **object**: ассемблер затем переводит ассемблерную программу в машинный код

- **executable machine code**: связывает все части программы для выполнения линкер
- а **loader** загружает все их в память, а затем программа выполняется

Preprocessor

в общем случае считается частью компилятора, это инструмент, который создаёт входные данные для компиляторов. Он занимается макро обработкой, расширением функциональности, включением файлов, расширением языка и т.д. Если происходит ошибка, компилятор читает всю программу, даже если встречает несколько ошибок.

Assembler

переводит программы на языке ассемблера в машинный код. Результат работы ассемблера называется объектным файлом, который содержит комбинацию машинных инструкций, а также данные, необходимые для размещения этих инструкций в памяти

Linker

это компьютерная программа, которая связывает и объединяет различные объектные файлы вместе, чтобы создать исполняемый файл. Все эти файлы могли быть скомпилированы отдельными ассемблерами. Основная задача линкера – искать и определять расположение ссылаемых модулей/процедур в программе и определять адрес в памяти, куда будут загружены эти коды, делая инструкции программы иметь абсолютные ссылки

Loader

является частью операционной системы и отвечает за загрузку исполняемых файлов в память и их выполнение. Он рассчитывает размер программы (инструкции и данные) и создает для нее пространство в памяти. Загрузчик инициализирует различные регистры для начала выполнения

Интерпретатор

как и компилятор, переводит высокоуровневый язык программирования в низкоуровневый машинный язык. Разница заключается в том, как они обрабатывают исходный код или ввод. Компилятор читает весь исходный код сразу, создает токены, проверяет семантику, генерирует промежуточный код, выполняет всю программу и может проходить несколько этапов (проходов) обработки. В отличие от этого, интерпретатор считывает одно выражение из входных данных, преобразует его в промежуточный код, выполняет его, а затем переходит к следующему выражению по порядку. Если происходит ошибка, интерпретатор останавливает выполнение и сообщает об ошибке, что позволяет быстрее находить ошибки, так как они сообщают о ней в момент возникновения. Это делает отладку и тестирование более удобными. Компилятор же продолжает читать весь код программы, даже если встречает несколько ошибок.

JIT (just-in-time) компиляция

это метод выполнения компьютерного кода, который включает компиляцию во время выполнения программы, а не до её выполнения. Идея заключается в том, чтобы скомпилировать код как раз вовремя для его запуска, отсюда и название. JIT-компиляторы работают с промежуточными представлениями кода, такими как байт-код, и компилируют его в нативный машинный код на лету.

Этапы JIT компиляции

1. **компиляция исходного кода в байт-код**
2. **интерпретационное исполнение**
   читает инструкции байт-кода и выполняет их одну за другой на виртуальной машине
3. **выявление часто выполняемого кода**
   Этот этап помогает определить, какие участки кода больше всего выиграют от компиляции в машинный код. Среда выполнения отслеживает “горячие точки”: циклы, часто вызываемые методы или критические пути кода *(частки программы, которые оказывают наибольшее влияние на общую производительность системы, так как они выполняются наиболее часто или требуют значительных вычислительных ресурсов)*.
4. **компиляция “на лету” в машинный код**
   Когда обнаруживается "горячая точка", JIT-компилятор берёт эту часть байт-кода и компилирует её в машинный код во время выполнения программы. Затем машинный код выполняется непосредственно CPU. Скомпилированный код кешируется, поэтому в следующий раз, когда будет выполнена та же самая "горячая точка", среда выполнения сможет напрямую использовать оптимизированный машинный код.

5. **методы оптимизации**
   JIT-компиляторы часто применяют различные техники оптимизации в процессе выполнения, такие как:
   a. Встраивание (**Inlining**): Замена вызова функции на фактический код этой функции для уменьшения накладных расходов на вызов.
   b. Развёртывание циклов (**Loop unrolling**): Расширение цикла для минимизации накладных расходов на проверку условия цикла.
   c. Устранение мёртвого кода (**Dead code elimination**): Удаление кода, который не влияет на результат работы программы.

   Эти оптимизации основаны на фактических данных времени выполнения, что может быть более эффективным по сравнению со статическими оптимизациями на этапе компиляции.

6. **адаптивная перекомпиляция**
   Если среда выполнения обнаруживает изменения в профиле исполнения (например, разные части кода становятся "горячими точками"), JIT-компилятор может повторно перекомпилировать участки кода с использованием различных оптимизаций, адаптируясь к изменяющимся условиям и обеспечивая максимально эффективную работу программы.

Преимущества JIT-компиляции

- **Оптимизация производительности:**
  JIT-компиляция позволяет программе работать быстрее, поскольку она компилирует часто выполняемый код в машинный код, который процессор выполняет намного быстрее, чем интерпретируемый байт-код.
- **Независимость от платформы:**
  Исходный код сначала компилируется в байт-код, который является универсальным набором инструкций, понятных виртуальной машине. Таким образом, этот байт-код может выполняться на любой платформе, где доступна совместимая JIT среда выполнения, обеспечивая абсолютную независимость от аппаратной архитектуры.
- **Динамическая оптимизация:**
  JIT-компиляция позволяет проводить оптимизацию на основе фактических шаблонов выполнения. Компилятор может принимать решения, используя данные в реальном времени, что приводит к лучшей оптимизации, чем та, которая возможна при статической компиляции.
- **Улучшенное управление ресурсами:**
  Сосредоточившись на горячих точках, JIT-компиляция эффективно использует ресурсы, избегая необходимости компилировать всю программу в машинный код сразу. Это помогает эффективнее управлять памятью и временем обработки.

Недостатки JIT-компиляции

- **Время запуска:**
  Первоначально программы, использующие JIT-компиляцию, могут работать медленнее, чем полностью скомпилированные программы, поскольку интерпретатор используется до того, как будут идентифицированы и скомпилированы «горячие точки».
- **Использование памяти:**
  JIT-компиляция требует дополнительной памяти для хранения скомпилированного машинного кода, что может увеличить общий объем памяти, занимаемый программой.
- **Сложность:**
  Реализация JIT-компиляторов сложна и требует сложных методов обнаружения горячих точек и оптимизации кода в реальном времени. Эта сложность может привести к увеличению времени разработки и отладки следующих версий.

Аналогии для понимания:

- **Компилятор**: Представьте себе переводчика, который переводит целую книгу с одного языка на другой и затем передает вам полностью переведенную книгу. Вы можете сразу читать её на новом языке, но перевод требует времени.

- **Интерпретатор**: Представьте себе другого переводчика, который сидит рядом с вами и переводит каждую страницу книги, пока вы её читаете. Вы начинаете читать сразу, но читаете медленнее, так как переводчик работает в реальном времени.
- **JIT-компилятор**: Представьте себе третьего переводчика, который сначала переводит каждую страницу книги по мере того, как вы её читаете, как интерпретатор. Но если переводчик замечает, что вы возвращаетесь к определённым страницам снова и снова, он начинает переводить эти страницы заранее и делает их доступными сразу в переводе. Таким образом, вы начинаете читать быстрее, потому что те страницы, к которым вы часто возвращаетесь, уже переведены и готовы. Это обеспечивает баланс между скоростью и эффективностью, адаптируясь к вашему стилю чтения и концентрируясь на наиболее важных частях текста.

Каков Python?

Python (CPython) является интерпретируемым языком, что означает, что его код исполняется интерпретатором Python. Однако это не так просто, как может показаться. Python включает в себя два основных этапа исполнения: компиляцию в байт-код и интерпретацию этого байт-кода.

CPython как Интерпретатор:

CPython — это эталонная реализация Python, написанная на языке C. Он называется "CPython", потому что реализован на языке программирования C. Он работает следующим образом: берет исходный код на Python, компилирует его в байт-код (это .pyc файлы), а затем интерпретирует этот байт-код. Как интерпретатор, CPython выполняет код Python, считывая его построчно и преобразуя в исполняемые действия с помощью компонента Python Virtual Machine (PVM).

Компонент Виртуальной Машины CPython:

Внутри CPython есть часть, известная как Python Virtual Machine (PVM). Это виртуальная машина, которая считывает и выполняет байт-код Python, сгенерированный компилятором CPython. PVM отвечает за интерпретацию инструкций байт-кода и управление выполнением программ на Python, включая такие задачи, как управление памятью и обработка исключений.

Различие между CPython и PVM:

Хотя CPython — это полный интерпретатор Python, который включает компилятор, генератор байт-кода и различные библиотеки, PVM — это конкретная часть CPython, которая действует как виртуальная машина для выполнения байт-кода. Другими словами, PVM является компонентом внутри интерпретатора CPython. Таким образом, CPython сам по себе не является виртуальной машиной, но включает виртуальную машину (PVM) для обработки выполнения байт-кода.

Итог:

CPython — это стандартная реализация Python, которая включает как компилятор (для преобразования кода Python в байт-код), так и компонент виртуальной машины (PVM) для выполнения этого байт-кода. Python Virtual Machine (PVM) является частью CPython, ответственной за считывание и выполнение инструкций байт-кода. Следовательно, когда мы говорим о CPython, мы имеем в виду всю среду интерпретатора Python, которая включает виртуальную машину как часть своей архитектуры для выполнения кода Python.