Урок 3 Как работает программа от исходного кода до выполнения

(Компиляторы, интерпретаторы, машинный код, байт-код и виртуальные машины)

Когда вы пишете программу на Python, C или Java, компьютер не понимает ваш код напрямую. Сначала он должен быть преобразован в форму, которую процессор может выполнить. Этот процесс называется **трансляцией**, а программы, которые этим занимаются, — **трансляторами** (компиляторами и интерпретаторами).

Транслятор — это программа, которая преобразует код, написанный на одном языке программирования, в другой язык (обычно в машинный код или промежуточное представление).

Трансляторы делятся на три основных типа:

1. **Интерпретаторы**
2. **Компиляторы**
3. **JIT-компиляторы** (Just-In-Time)

Давайте разберёмся, как код превращается в работающую программу, что такое **машинный код, байт-код, объектный файл** и зачем нужны **виртуальные машины**.

---

1. Как компьютер понимает программу?

Компьютер работает на **машинном коде** — наборе инструкций, которые процессор выполняет напрямую. Этот код состоит из **бинарных чисел** (например, `10110000 01100001`), и писать на нём сложно. Поэтому люди придумали языки программирования, а трансляторы переводят их в машинный код.

```
mov eax, 5   ; Загрузить число 5 в регистр EAX (машинный код: B8 05 00 00 00)
add eax, 3   ; Прибавить 3 (машинный код: 83 C0 03)
```

Писать программы в таком виде крайне неудобно, поэтому существуют языки высокого уровня и трансляторы

Два основных подхода к выполнению программ:

- **Компиляция** — код преобразуется в машинный код **до запуска** (C, C++, Go).
- **Интерпретация** — программа выполняется **построчно во время работы** (Python, JavaScript, Bash).

---

2. Компиляторы: от исходного кода к исполняемому файлу

Как работает компилятор?

1. **Препроцессинг** — обработка макросов и включение заголовочных файлов (в C/C++).
2. **Компиляция** — перевод кода в **ассемблер** (низкоуровневый язык).
3. **Ассемблирование** — преобразование ассемблера в **объектный код** (бинарный, но ещё не исполняемый).
4. **Линковка** — объединение объектных файлов в **исполняемый файл** (например, `.exe` или `.elf`).

Пример на C:

```c
// hello.c
#include <stdio.h>
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Компиляция:

```
gcc hello.c -o hello  # → создаётся бинарный файл hello
./hello               # → программа выполняется
```

Плюсы компиляции:

- Высокая скорость выполнения (машинный код работает напрямую с процессором).
- Защита исходного кода (его можно распространять в бинарном виде).

Минусы:

- Нужно компилировать под каждую платформу (Windows, Linux, ARM).
- Исправление ошибок требует перекомпиляции.

---

3. Интерпретаторы: выполнение кода на лету

Как работает интерпретатор?

Интерпретатор **не создаёт отдельный исполняемый файл**, а читает исходный код и выполняет его **построчно**.

Интерпретатор — это программа, которая выполняет исходный код **построчно**, без предварительного преобразования в машинный код.

Как работает интерпретатор по шагам:

1. **Чтение строки кода** – интерпретатор загружает исходный код.
2. **Лексический анализ** – разбивает строку на токены (ключевые слова, переменные, операторы).
3. **Синтаксический анализ** – проверяет, соответствует ли код правилам языка (грамматике).

4. **Немедленное выполнение** – интерпретатор выполняет команду напрямую.
5. **Переход к следующей строке** – повторяет процесс до конца программы.

Пример на Python:

```
# hello.py
print("Hello, World!")
```

Запуск:

```
python hello.py  # интерпретатор читает и выполняет код
```

Плюсы интерпретации:

- Кроссплатформенность (один и тот же код работает везде, где есть интерпретатор).
- Быстрое тестирование (не нужно компилировать).

Минусы:

- Медленнее компилируемых языков (каждая строка обрабатывается отдельно).
- Исходный код должен быть доступен при запуске.

3.1. Как на самом деле работает интерпретатор?**

(С отсылкой к А. Столярову)

Распространённая ошибка:

> "Интерпретатор переводит программу в машинный код построчно и сразу выполняет"

Почему это неверно:
На самом деле, **интерпретатор — это обычная программа**, которая:

1. Читает исходный код (например, Python-скрипт).
2. **Анализирует его структуру** (разбирает на токены, строит синтаксическое дерево).
3. **Выполняет соответствующие действия**, **уже будучи скомпилированной в машинный код**.

Где здесь машинный код?
Сам интерпретатор (например, `python.exe`) — это **скомпилированная программа** на C. Когда вы запускаете скрипт, происходит примерно следующее:

```
# Ваш script.py
print("Hello")
```

1. Машинный код интерпретатора (`python.exe`) читает файл `script.py`.
2. Видит строку `print("Hello")`.
3. Вызывает **уже скомпилированную** функцию `print()` из стандартной библиотеки Python (которая тоже машинный код!).

Вывод:
Интерпретатор **не генерирует** новый машинный код из вашего скрипта. Он сам является машинным кодом, который "понимает" Python-синтаксис и действует соответственно.

3.2. Как интерпретатор превращает исходный код в байт-код?

Шаг 1: Запуск программы

Допустим, у нас есть файл `hello.py`:

```
print("Hello, World!")
```

Ты запускаешь его:

```
python hello.py
```

Шаг 2: Чтение и компиляция в байт-код

1. **Интерпретатор** читает `hello.py`.
2. **Лексический анализ**: разбивает код на токены (`print`, `"Hello, World!"`).
3. **Синтаксический анализ**: строит абстрактное синтаксическое дерево (AST).
4. **Генерация байт-кода**:
   - Код компилируется в низкоуровневые инструкции для **PVM** (Python Virtual Machine).
   - Результат сохраняется в `.pyc`-файл (если это модуль).

Где лежит байт-код?

- Для скриптов (вроде `hello.py`) байт-код **не сохраняется** (если только не используется `-O` или `__pycache__`).
- Для импортируемых модулей — в папке `__pycache__`:

```
__pycache__/
  hello.cpython-310.pyc  # Байт-код для Python 3.10
```

Как посмотреть байт-код?

```
import dis
dis.dis('print("Hello")')

# Выведет:
#  1           0 LOAD_NAME                0 (print)
#              2 LOAD_CONST               0 ('Hello')
#              4 CALL_FUNCTION            1
#              6 RETURN_VALUE
```

3.3. Где физически находится интерпретатор Python?

Когда ты устанавливаешь Python (например, с python.org), на компьютер попадают:

- **Исполняемый файл интерпретатора** (например, `python.exe` в Windows или `/usr/bin/python3` в Linux).
- **Стандартная библиотека** (пакеты типа `os`, `math`, `json` — лежат в папке `Lib`).
- **Компилятор байт-кода** (скрыто внутри интерпретатора).

Пример (Windows):

```
C:\Users\Вася\AppData\Local\Programs\Python\Python310\
├── python.exe          # Интерпретатор
├── python310.dll       # Основная библиотека
└── Lib\                # Стандартные модули (os.py, math.py и т.д.)
```

Как проверить?

```
# В командной строке:
where python   # Windows
which python3  # Linux/Mac
```

4. Гибридный подход: байт-код и виртуальные машины

Некоторые языки (Java, C#, Python (частично)) используют **промежуточный байт-код**.

Как это работает?

1. **Компиляция в байт-код** (не машинный код, а универсальный для виртуальной машины).
2. **Выполнение виртуальной машиной** (JVM для Java, CPython для Python).

Пример на Java:

```
// Hello.java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Компиляция и запуск:

```
javac Hello.java  # → Hello.class (байт-код)
java Hello        # → выполнение JVM
```

Байт-код Python (.pyc)

Python тоже компилируется в байт-код (файлы `.pyc`), но обычно это происходит автоматически.

Можно посмотреть байт-код:

```
import dis
def hello():
    print("Hello!")
dis.dis(hello)  # → вывод ассемблероподобных инструкций
```

Плюсы байт-кода:

- **Переносимость** (один байт-код работает на всех платформах с виртуальной машиной).
- **Оптимизация** (JIT-компиляция ускоряет выполнение, как в Java и PyPy).

Минусы:

- Требуется виртуальная машина (JVM, .NET CLR, Python interpreter).
- Немного медленнее нативного кода (но быстрее чистого интерпретатора).

---

5. Что такое JIT-компиляция?

Just-In-Time (JIT) — это технология, которая **компилирует код во время выполнения**, сочетая преимущества интерпретации и компиляции.

Где используется?

- **Java (HotSpot JVM)** — сначала интерпретация, затем JIT для "горячего" кода.
- **JavaScript (V8 в Chrome, SpiderMonkey в Firefox)** — ускоряет выполнение скриптов.
- **PyPy для Python** — JIT делает Python в 4-5 раз быстрее стандартного интерпретатора.

Вывод: какой подход лучше?

Критерий	Компиляция (C, Go)	Интерпретация (Python, JS)	Байт-код + VM (Java, C#)
Скорость	⚡ Максимальная	🐢 Медленная	⚡/🐢 (зависит от JIT)
Переносимость	❌ Нужна перекомпиляция	✅ Один код везде	✅ Байт-код работает везде
Защита кода	✅ Исполняемый файл	❌ Нужен исходник	✅ Байт-код можно обфусцировать

Какой язык выбрать?

- **Для скорости и системного программирования** → C, C++, Rust (компиляция).
- **Для кроссплатформенности и скорости разработки** → Python, JavaScript (интерпретация/JIT).
- **Для баланса между скоростью и переносимостью** → Java, C# (байт-код + JIT).

Термины

Заключение

Теперь вы знаете, как код превращается в работающую программу!

- **Компиляторы** делают из кода **машинный** или **объектный код**.
- **Интерпретаторы** выполняют код **построчно**.
- **Байт-код + виртуальная машина** — это компромисс между скоростью и переносимостью.
- **JIT** ускоряет выполнение, компилируя "на лету".

Если хотите глубже разобраться, попробуйте:

- Скомпилировать C-программу и посмотреть ассемблер (`gcc -S hello.c`).
- Изучить байт-код Java или Python.
- Поэкспериментировать с PyPy для ускорения Python-кода.

Программирование — это магия, но теперь вы знаете, как она работает! 🚀

Вот все ключевые термины, упомянутые в статье, с краткими определениями:

Основные термины

1. **Транслятор** — программа, преобразующая код из одного языка в другой (компиляторы, интерпретаторы).
2. **Компилятор** — транслятор, который **полностью** переводит исходный код в машинный/байт-код **до выполнения** (C, C++, Java).
3. **Интерпретатор** — программа, которая **построчно анализирует и выполняет** исходный код без предварительной компиляции (Python, JavaScript).
4. **Исходный код** — текст программы на языке программирования (например, файл `.c` или `.py`).

Типы кода

5. **Машинный код** — бинарные инструкции, которые процессор выполняет напрямую (например, `10110000`).
6. **Бинарный код** — синоним машинного кода или любого представления данных в виде 0 и 1.
7. **Объектный код** — промежуточный бинарный файл, полученный после компиляции (например, `.o` в C), но ещё не исполняемый.
8. **Байт-код** — промежуточный код для виртуальной машины (Java `.class` , Python `.pyc`).
9. **Исполняемый код** — готовый машинный код, который можно запустить (например, `.exe` или `.elf`).

Процессы

10. **Компиляция** — процесс преобразования исходного кода в машинный/байт-код.
11. **Интерпретация** — построчное выполнение кода без компиляции.
12. **Линковка** — объединение объектных файлов в исполняемый (шаг после компиляции в C).

13. **JIT-компиляция** (Just-In-Time) — компиляция во время выполнения (Java, PyPy).

Дополнительные понятия

14. **Виртуальная машина (VM)** — программа, выполняющая байт-код (JVM для Java, CPython для Python).
15. **Ассемблер** — низкоуровневый язык, близкий к машинному коду.
16. **Препроцессинг** — предварительная обработка кода (например, замена `#include` в C).
17. **Декомпиляция** — обратное преобразование машинного/байт-кода в читаемый код.

Эти термины помогут систематизировать понимание работы трансляторов и этапов выполнения программ. 🛠️