

Язык программирования D

Презентация 6.

Управление памятью. Конкурентность и параллелизм. UFCS. Интеграция с C, C++.
Контрактное программирование.

Морщанин Н. 5030102/20201

Управление памятью

Язык D предлагает гибридную модель управления памятью, сочетающую безопасность сборки мусора (Garbage Collector, GC) с возможностями низкоуровневого контроля.

Сборка мусора по умолчанию

Память выделяется оператором `new`. Сборщик мусора автоматически освобождает память, когда на неё не остаётся ссылок.

```
// Выделение памяти под управлением GC
int* ptrToInt = new int;
auto dynamicArray = new int[](100);
```

Сборщик мусора в D: общая архитектура

Сборщик мусора (GC) в D — это неблокирующий, поколенный (generational) сборщик, работающий по принципу mark-and-sweep (помечай и очищай).

Основные характеристики:

- Нет приостановки всех потоков (stop-the-world) — работает конкурентно
- Поколенная структура — объекты делятся по "возрасту"
- Инкрементальная сборка — можно выполнять сборку частями

Поколенная структура (Generational GC)

Объекты разделены на три поколения:

Поколение 0 (Молодое)

Недавно созданные
объекты

Быстрая сборка
(каждые несколько
МБ аллокаций)

Поколение 1 (Среднее)

Пережившие одну
сборку

Средняя частота
сборки

Поколение 2 (Старое)

Долгоживущие
объекты

Редкая сборка
(только при
нехватке памяти)

Конфигурация и управление GC

Настройка через переменные окружения:

```
export DFLAGS="-L--largemode1 -L--minheap=1000000 -L--maxheap=2000000"  
export D_GC_MINCAP=1024  
export D_GC_MAXCAP=1048576
```

Программное управление:

```
import core.memory;

// Ручной запуск сборки
GC.collect();

// Получение статистики
auto stats = GC.stats();
writeln("Память: ", stats.usedSize, "/", stats.totalSize,
      " байт (", stats.usedSize * 100 / stats.totalSize, "%)");

// Отключение GC для критических секций
GC.disable();
// ... критический код ...
GC.enable();
```

Низкоуровневый контроль и `@nogc`

Ключевое слово `@nogc` гарантирует, что функция или блок кода не выделяют память в куче через GC.

```
@nogc void criticalLoop() {  
    int[100] stackArray; // Память в стеке – допустимо  
    // auto ptr = new int; // Ошибка компиляции в @nogc-коде  
}
```

Это критически важно для систем реального времени и низкоуровневых компонент.

Пример ручного управления памятью

```
import core.stdc.stdlib : malloc, free;

@nogc void manualMemory() {
    int* arr = cast(int*)malloc(int.sizeof * 100);
    scope(exit) free(arr);           // Гарантированное освобождение

    arr[0] = 42;                   // Работа с сырой памятью
}
```

Uniform Function Call Syntax (UFCS)

UFCS — синтаксический сахар, позволяющий вызывать функцию `f(a, b)` как `a.f(b)`. Это улучшает читаемость и позволяет создавать цепочки вызовов, похожие на методы.

Пример UFCS

```
import std.string;
import std.stdio;
int add(int a, int b) { return a + b; }
void main() {
    string name = " D Language ";
    // Классический вызов функции
    auto trimmed1 = strip(name);
    auto upper1 = toUpper(trimmed1);

    // Вызов с UFCS, создающий чистую цепочку
    auto result = name.strip().toUpper();
    writeln(result); // "D LANGUAGE"
    int sum = 5.add(3); // sum = 8
    writeln(sum);
    // Также работает со своими функциями
}
```

Контрактное программирование

Система предусловий (`in`), постусловий (`out`) и инвариантов (`invariant`) для проверки корректности.

`assert` vs контракты: `assert` проверяет внутреннюю логику, контракты — формальные условия работы API.

Контракты: пример

```
class BankAccount {  
    private double balance;  
    // Предусловие: сумма должна быть положительной  
    void withdraw(double amount) in {  
        assert(amount > 0 && amount <= balance);  
    } body {  
        balance -= amount;  
    }  
    // Постусловие: баланс не должен стать отрицательным  
    void deposit(double amount) out(result) {  
        assert(balance >= 0);  
    } body {  
        balance += amount;  
    }  
    // Инвариант класса (проверяется в конце публичных методов)  
    invariant {  
        assert(balance >= 0, "Баланс не может быть отрицательным");  
    }  
}
```

Interfacing с другими языками

D имеет нативную поддержку взаимодействия с C и C++.

Совместимость с C

```
// Вызов функции из стандартной библиотеки С
import core/stdc.stdio;

extern(C) void main() {
    printf("Hello from D and C!\n");
}
```

Использование библиотек С и С++

```
// Объявление С-функции  
extern(C) int open(const char* pathname, int flags);  
  
// Объявление С++-класса (упрощенно)  
extern(C++) class MyCPPClass {  
    int doWork(int param);  
}
```

Ключевые атрибуты:

- `extern(C)` : стандартный С ABI (по умолчанию для функций)
- `extern(C++)` : для С++ связывания (платформозависимо)
- `extern(Objective-C)` : для macOS/iOS экосистемы

Это позволяет напрямую использовать огромное количество существующих библиотек.

Конкурентность и параллелизм

Д предоставляет встроенную поддержку многопоточности как часть языка и стандартной библиотеки.

Конкурентность в D: обмен сообщениями

Канонический подход к конкурентности в D — **обмен сообщениями без разделяемой памяти** через модуль `std.concurrency`.

```
import std.concurrency;
import std.stdio;

void worker(Tid ownerTid) {
    // Поток получает сообщение
    receive((int value) {
        writeln("Получено: ", value);
        ownerTid.send("Готово!");
    });
}
```

```
void main() {
    // Порождение потока
    auto workerTid = spawn(&worker, thisTid);
    // Отправка сообщения
    workerTid.send(42);
    // Ожидание ответа
    receive((string reply) {
        writeln(reply);
    });
}
```

Этот подход предотвращает гонки данных.

Параллелизм в D: `std.parallelism`

Для параллельного выполнения вычислений предназначен модуль `std.parallelism`.

Параллельный цикл `foreach`:

```
import std.parallelism;
import std.range;

void main() {
    auto numbers = iota(0, 100).array;
    foreach (i; parallel(numbers)) {
        numbers[i] = process(numbers[i]); // Выполняется параллельно
    }
}
```

Использование пула потоков (Task):

```
import std.parallelism;

auto task1 = task!heavyCalculation(1);
// !heavyCalculation – передача функции как шаблонного параметра
// (1) - аргумент для функции
auto task2 = task!heavyCalculation(2);
task1.executeInNewThread();
// запуск в отдельном потоке
task2.executeInNewThread();
auto result1 = task1.yieldForce;
// получение результата
auto result2 = task2.yieldForce;
```

Синхронизация при работе с разделяемой памятью

Когда обмена сообщениями недостаточно, D предоставляет примитивы синхронизации.

- **Мьютекс (Mutex)**: Гарантирует, что критическая секция кода выполняется только одним потоком.

```
void deposit(double amount) {
    // Явная синхронизация с использованием мьютекса
    synchronized(mutex) {
        balance += amount;
        writeln("Внесено: ", amount, ", баланс: ", balance);
    }
}
```

- **Атомарные операции:** Для простых типов доступны атомарные чтение/запись.

```
shared int atomicCounter;  
import core.atomic : atomicOp;  
atomicCounter	atomicOp! "+="(1);
```

Ключевое слово `shared` помечает данные, доступные из нескольких потоков.

Использованные ресурсы

1. [Официальный учебник по D](#)
2. [Язык программирования D: взгляд с высоты \(Habr\)](#)
3. [Awesome D \(библиотеки и ресурсы\)](#)
4. [Введение в D \(Tour of D\)](#)
5. [Викиучебник по языку D](#)
6. [D Reference: Interfacing to C++](#)
7. [std.parallelism documentation](#)

Спасибо за внимание!