Язык программирования D

Презентация 5. Парадигмы программирования в языке D

Объектно-ориентированное программирование. Функциональное программирование. Обобщенные типы (шаблоны). Мета-программирование.

Морщинин Н. 5030102/20201

Объектно-ориентированное программирование

Язык D предоставляет полную поддержку ООП с классами, интерфейсами, наследованием и полиморфизмом. Классы в D ссылочные, размещаются в куче и управляются сборщиком мусора.

ООП Пример

```
class Animal {
    string name;
    int age;

    this(string name, int age) {
        this.name = name;
        this.age = age;
    }

    void makeSound() {
        writeln("Some sound");
    }
}
```

```
class Dog : Animal {
    override void makeSound() {
        writeln("Woof!");
    }
}

interface Flyable {
    void fly();
}
```

Объектно-ориентированное программирование

Особенности ООП в D:

- Единое наследование классов, множественное — интерфейсов
- Модификаторы доступа: `private`, `package`, `protected`, `public`
- Абстрактные классы и методы с ключевым словом `abstract`
- Виртуальные методы, переопределение через `override`
- Автоматическое управление памятью (GC)
- `final` классы и методы для запрета наследования/переопределения
- Свойства (`@property`) для геттеров/сеттеров (использование `obj.field`)

Пример полиморфизма:

```
Shape[] shapes = [new Circle(5), new Square(3)];
foreach(shape; shapes) {
    shape.draw(); // Вызовется соответствующая реализация
}
```

Функциональное программирование

D поддерживает функциональную парадигму с функциями высшего порядка, чистыми функциями, лямбдами и неизменяемыми данными.

Чистые функции (pure):

```
pure int add(int a, int b) {
    return a + b; // Нет побочных эффектов
}
```

Иммутабельные данные:

```
immutable string greeting = "Hello";
immutable int[] numbers = [1, 2, 3];
```

Лямбда-выражения:

```
auto square = (int x) => x * x;
auto sum = (int a, int b) => a + b;
```

Функциональное программирование

Функции высшего порядка и цепочки обработки:

```
import std.algorithm, std.range;

auto numbers = iota(1, 11); // 1..10

auto result = numbers
    .filter!(n => n % 2 == 0)
    .map!(n => n * 2)
    .array;
```

Рекурсия и оптимизация хвостовой рекурсии:

```
int factorial(int n) {
    return n == 0 ? 1 : n * factorial(n - 1);
}

// Хвостовая рекурсия (компилятор может оптимизировать)
int tailFactorial(int n, int acc = 1) {
    return n == 0 ? acc : tailFactorial(n - 1, acc * n);
}
```

Обобщенные типы (шаблоны)

D имеет мощную систему шаблонов, аналогичную C++, но с улучшенным синтаксисом и возможностями.

Шаблон функции:

```
T max(T)(T a, T b) {
    return a > b ? a : b;
}

auto m = max!int(5, 3);
auto m2 = max(5.0, 3.0); // Вывод типа
```

Шаблон класса:

```
class Stack(T) {
    private T[] elements;

    void push(T element) {
        elements ~= element;
    }

    T pop() {
        return elements[$ - 1];
    }
}

auto intStack = new Stack!int;
```

Обобщенные типы (шаблоны)

Ограничения шаблонов (концепты):

```
template isNumeric(T) {
    enum bool isNumeric = is(T == int) || is(T == double) || is(T == float);
}

T add(T)(T a, T b) if (isNumeric!T) {
    return a + b;
}
```

Шаблоны с переменным числом параметров (Variadic templates):

```
void print(T...)(T args) {
    foreach(arg; args) {
        writeln(arg);
    }
}

print(1, "hello", 3.14);
```

Параметризация по значению и типу:

```
template Matrix(int rows, int cols, T) {
    alias Matrix = T[rows][cols];
}

alias Mat3x3 = Matrix!(3, 3, float);
```

Мета-программирование

D обладает одними из самых мощных возможностей метапрограммирования среди компилируемых языков.

Вычисление во время компиляции (CTFE):

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}

enum fact5 = factorial(5); // Вычисляется при компиляции
```

Статические условия и проверки:

```
template isIntegral(T) {
    enum bool isIntegral = is(T == int) || is(T == long) || is(T == short);
}

static if (isIntegral!T) {
    // Код только для целочисленных типов
}
```

Мета-программирование

Генерация кода с помощью `mixin` и строковых операций:

```
mixin(`
    int add(int a, int b) {
        return a + b;
    }
`);
```

Мета-программирование

Reflection и интроспекция во время компиляции:

```
import std.traits;

void printInfo(T)() {
    pragma(msg, "Type: ", T.stringof);
    pragma(msg, "Size: ", T.sizeof);
    static if (is(T == class)) {
        pragma(msg, "Is class");
    }
}

// Вызов при компиляции
printInfo!int();
```

Анализ членов класса

```
writeln("=== Информация о классе: ", T.stringof, " ===");

// Получаем все члены класса
foreach(member; __traits(allMembers, T)) {
    writeln("Член: ", member);

    // Получаем тип члена
    alias MemberType = typeof(__traits(getMember, T, member));
    writeln("  Тип: ", MemberType.stringof);
}
```

Пользовательские атрибуты (UDA):

```
@(Serializable, Version(1, 0))
class User {
    string name;
    int age;
}
```

Простые атрибуты-флаги

```
// Пустая структура как флаг
struct Serializable {
}
```

```
// Атрибут с параметрами
struct Version {
    int major;
    int minor;

    this(int major, int minor) {
        this.major = major;
        this.minor = minor;
    }
}
```

```
// Атрибут для документирования
struct Description {
    string text;
}
```

Использование атрибутов

```
@Serializable
@Version(1, 0)
@Description("Класс пользователя системы")
class User {
    @Description("Имя пользователя")
    string name;

    @Description("Возраст пользователя")
    int age;

    @Description("Электронная почта")
    string email;
}
```

Извлечение и использование атрибутов во время компиляции

```
import std.traits;

// Функция для проверки наличия атрибута
bool hasAttribute(T, alias Attr)() {
    static if (hasUDA!(T, Attr)) {
        return true;
    } else {
        return false;
    }
}
```

```d
// Получение значения атрибута
template getAttribute(T, alias Attr) {
    static if (hasUDA!(T, Attr)) {
        alias getAttribute = getUDAs!(T, Attr)[0];
    } else {
        alias getAttribute = void;
    }
}
```

Пример: сериализация с атрибутами

```
struct JsonField {
    string name;
    bool required;

    this(string name, bool required = true) {
        this.name = name;
        this.required = required;
    }
}
```

```
// Класс с аннотациями для JSON сериализации
class Person {
    @JsonField("person_name", true)
    string name;

    @JsonField("years_old", true)
    int age;

    @JsonField("email_address", false)
    string email;

    @JsonField("phone_number", false)
    string phone;
}
```

Обработчик атрибутов

```
string toJson(T)(T obj) {
    import std.conv : to;
    string json = "{";

    // Используем интроспекцию для перебора полей
    foreach(fieldName; FieldNameTuple!T) {
        alias FieldType = typeof(__traits(getMember, T, fieldName));

        // Проверяем наличие атрибута JsonField
        static if (hasUDA!(__traits(getMember, T, fieldName), JsonField)) {
            auto uda = getUDAs!(__traits(getMember, T, fieldName), JsonField)[0];
            string jsonName = uda.name;
            auto fieldValue = __traits(getMember, obj, fieldName);
            json ~= "\"" ~ jsonName ~ "\":";
            // Сериализация в зависимости от типа
            static if (is(FieldType == string)) {
                json ~= "\"" ~ fieldValue ~ "\"";
            } else static if (is(FieldType : int) || is(FieldType : double) || is(FieldType : float)) {
                json ~= fieldValue.to!string;
            } else static if (is(FieldType == bool)) {
                json ~= (fieldValue ? "true" : "false");
            } else {
                json ~= "null";
            }
            json ~= ",";
        }
    }
    // Убираем последнюю запятую
    if (json[$-1] == ',') {
        json = json[0..$-1];
    }
    json ~= "}";
    return json;
}
```

Использованные ресурсы

1. Официальная документация D
2. Язык программирования D: взгляд с высоты
3. Awesome D
4. Tour of D
5. D Programming Language Forum

Спасибо за внимание!