Интерфейсы и типы данных (часть 2)

Алексей Островский

Физико-технический учебно-научный центр НАН Украины

16 апреля 2015 г.

Типобезопасность

Определение

Типобезопасность (англ. *type safety*) — мера, в которой язык программирования или стиль написания программы предотвращает ошибки типов.

Определение

Ошибка типа (англ. *type error*) — дефектное или нежелательное поведение программы, вызванное различием между ожидаемым и действительным смыслом данных, связанных с переменной или функцией / методом.

Виды типобезопасности:

- статическая (определение ошибок во время компиляции);
- динамическая (поиск ошибок во время выполнения).

В «хорошем» ЯП результат любого выражения является корректным значением типа, который может быть определен на основе выражения во время компиляции.

Безопасность памяти

Определение

Безопасность памяти (англ. *memory safety*) — мера, в которой язык или стиль программирования защищены от ошибок доступа к памяти.

Ошибки, связанные с памятью:

- переполнение буфера, кучи в целом или стека;
- использование неинициализированных переменных;
- некорректная работа с динамической памятью:
- работа с указателем после освобождения памяти;
- многократное высвобождение одного указателя;
- неправильная обработка нулевых указателей.

Сильная и слабая типизация

Определение

Сильная / слабая типизация (англ. *strong / weak type system*) — степень соблюдения языком программирования безопасности типов и памяти.

Сильная типизация:

- отсутствие указателей / ссылок, арифметики указателей;
- отсутствие различающихся представлений одних и тех же данных (таких как **`union`** в C / C++);
- минимальное количество неявных приведений типов;
- отсутствие неочевидных для программиста спецификаций операций (таких как перегрузка операторов в C++).

ЯП с сильной типизацией: Python, Java, функциональные ЯП.

ЯП со слабой типизацией: C, C++, Visual Basic.

Статическая и динамическая типизация

Статическая типизация — определение типов всех конструкций языка на этапе компиляции программы (слабая форма верификации программы).

Виды статической типизации:

- явная — типы конструкций декларируются программистом (напр., при объявлении переменных);
- неявная — тип переменных выводится в процессе компиляции. Примеры:
  - `var x = 5` в C#;
  - `List<> list = new ArrayList<String>()` в Java 7+.

ЯП со статической типизацией: C++, Pascal, Java, C#.

Динамическая типизация — определение типов некоторых конструкций и проверка соответствующих ограничений во время выполнения программы.

ЯП с динамической типизацией: Python, PHP, Perl, JavaScript.

Совместимость типов

Задача

Проверить соответствие типа всех выражений ожидаемому в конкретной ситуации. Понятие соответствия специфично для конкретного ЯП.

Конексты, где необходимо согласование:

- присвоение <переменная> = <выражение> (типы переменной и выражения);
- вызов функции / метода (соответствие аргументов сигнатуре).

Методы определения совместимости:

- **«Плоская» система типов:** совместимость = эквивалентность; определяется исходя из деклараций или структуры данных.
- **Иерархическая система типов:** совместимость определяется отношениями тип — подтип (задекларированными или неявными).

Номинальная и структурная типизация

Номинальная типизация — вид системы типов, при которой совместимость и эквивалентность типов определяется на основе явных деклараций (наследования, имплементации интерфейсов и т. п.).

ЯП с номинальной типизацией: C#, Java; C++ (основные типы).

Структурная типизация — вид системы типов, при которой совместимость и эквивалентность типов определяется на основе внутренней структуры объектов *во время компиляции*.

ЯП со структурной типизацией: C++ (шаблоны); функциональные ЯП (Haskell, ML).

Утиная типизация — вид системы типов, при которой совместимость и эквивалентность типов определяется на основе структуры объектов *во время выполнения*.

ЯП с утиной типизацией: Python, JavaScript.

Утиная типизация

Определение

Утиная типизация (англ. *duck typing*) — вид динамической типизации, при которой корректность использования объекта определяется набором его методов и свойств, а не типом.

ЯП с утиной типизацией:

- языки с ООП на основе прототипов (JavaScript, Lua);
- Python;
- Smalltalk.

Пример динамической не утиной типизации: подсказки типов (англ. *data hinting*) в PHP.

Пример: утиная типизация в Python

```
1  def count(iterable):
2      """Подсчитывает число вхождений в коллекцию каждого из элементов."""
3
4      cnt = dict()
5      for elem in iterable:
6          cnt[elem] = (cnt[elem] + 1) if cnt.has_key(elem) else 1
7      return cnt
8
9  # Возможные аргументы:
10 print count([1, 4, 5, 3, 2, 1, 1, 4]) # списки (list)
11 print count((0, 1, 0, 2, 1, 2)) # неизменяемые списки (tuple)
12 print count('foobarbazz') # строки (str)
13 print count({5, 4, 3, 2, 1}) # множества (set)
14 print count({'f': 'oo', 'b': 'ar'}) # ассоциативные массивы (dict)
```

Приведение типов в различных ЯП

Java:

```
1 String str = "4" + 2; // str == "42"
```

Python:

```
1 x = "4" + 2;
2 # TypeError: cannot concatenate 'str' and 'int' objects
```

PHP:

```
1 <?php
2 $number = 2;
3 $x = '4' + $number; // $x == 6
4 $x = '4' . $number; // $x == "42"
```

C:

```
1 printf("%s\n", "4" + 2);
2 char* nextStr = "next string";
3 // выведет (скорее всего) "next string"
```

Неявное приведение типов данных

Определение

Неявное приведение ТД (англ. *implicit conversion, coercion*) — приведение типов данных, осуществляемое автоматически компилятором.

Примеры:

- Преобразование между числовыми типами с повышением разрядности или множества представимых чисел (**`byte`** → **`short`** → **`int`** → **`long`** → **`float`** → **`double`**).

  Код (Java):

  ```
  1 short a = 2; int b = 3;
  2 double x = a + b; // x == 5
  ```

- Приведение объектов к строковому представлению; зачастую — с помощью средств ООП:
  - метод `toString()` в Java;
  - метод `ToString()` в C#;
  - метод `str.format()` и ключевое слово / функция `print` в Python.

Явное приведение типов данных

Определение

Явное приведение ТД (англ. *explicit conversion*) — приведение ТД, специфицируемое в исходном коде программы.

Подвиды:

- С динамической проверкой типа во время выполнения:
  - приведение типов в Java;
  - оператор **`dynamic_cast`**<T> в C++.
- Без динамической проверки типа:
  - **`static_cast`**<T> в C++;
  - оператор **`as`** в C#.

Пример: приведение ТД в C#

```
1  class SomeClass { /* … */ }
2  class Subclass : SomeClass { /* … */ }
3
4  SomeClass obj = // …;
5
6  // Приведение типа с проверкой;
7  // если obj — не экземпляр Subclass, возбуждается исключение
8  try {
9      Subclass checkedSub = (Subclass) obj;
10 } catch (InvalidCastException e) {
11     /* … */
12 }
13
14 // Безусловное приведение;
15 // если obj — не экземпляр Subclass, uncheckedSub == null
16 Subclass uncheckedSub = obj as Subclass;
```

Полиморфизм

Определение

Полиморфизм — использование единого интерфейса для сущностей различных типов.

Виды полиморфизма:

- Специальный (ad hoc) полиморфизм — определение различных реализаций для конечного числа фиксированных наборов входных типов (напр., перегрузка функций / методов).
- Параметрический полиморфизм — определение обобщенной реализации для произвольного типа (напр., шаблоны в C++; generics в Java и C#).
- Полиморфизм подтипов — использование интерфейса класса для любого производного от него подкласса (применяется в ООП).

Пример: специальный полиморфизм (Java)

```java
public class Arrays {
    /* … */

    // Методы для бинарного поиска в массивах
    public static int binarySearch(byte[] a, byte key);
    public static int binarySearch(int[] a, int key);
    // другие массивы из примитивных ТД

    // используется для всех массивов, не состоящих из примитивных элементов
    public static int binarySearch(Object[] a, Object key);
}

Arrays.binarySearch(new int[] { 2, 3, 5, 8 }, 6);
Arrays.binarySearch(new double[] { 2.1, 3.1, 5.0, 8.0 }, 6.3);
Arrays.binarySearch(new String[] { "bar", "bazz", "foo" }, "element");
```

Пример: параметрический полиморфизм (C++)

```
1  template<typename T> string dump(const void* ptr, int size) {
2      ostringstream oss;
3
4      T* array = (T*) ptr;
5      for (int i = 0; i < size / sizeof(T); i++) {
6          oss << array[i];
7          if (i < array_sz - 1) oss << " ";
8      }
9      return oss.str();
10 }
11
12 const string str("Datatype");
13 const void* ptr = (void*) str.c_str();
14 const int sz = str.size();
15
16 cout << dump<char>(ptr, sz) << endl; // D a t a t y p e
17 cout << dump<short>(ptr, sz) << endl; // 24900 24948 31092 25968
18 cout << dump<int>(ptr, sz) << endl; // 1635017028 1701869940
```

Пример: параметрический полиморфизм (Java)

```java
// Интерфейс для списков, состоящих из элементов типа E
public interface List<E>{
    boolean add(E e);
    E get(int index);
    E remove(int index);
    E set(int index, E element);
    /* … */
}

List<Integer> intList = new ArrayList<Integer>();
intList.add(2); // боксинг int -> Integer
intList.add("foo"); // ошибка компиляции
```

Информация о параметризации типа в Java доступна только во время компиляции; во время выполнения эти сведения стираются:

```java
List<Object> objList = (List<Object>) intList;
objList.add("foo");
int x = intList.get(1); // ошибка времени выполнения
```

Особенности параметрического полиморфизма

C++: шаблоны (англ. *templates*).

- Используется для объявления широкого круга конструкций (функции, структуры, классы);
- аргументы — произвольные переменные (не обязательно типы данных);
- возможность явного указания частных реализаций;
- для каждого набора аргументов генерируется свой код.

Java и C#: шаблонные типы (англ. *generics*).

- Используется для объявления классов / интерфейсов и отдельных методов;
- аргументы — типы данных (в Java — только ссылочные);
- общий код для всех аргументов;
- стирание типов во время выполнения (Java).

Полиморфизм подтипов

Принцип подстановки Барбары Лисков (англ. *Liskov substitution principle, LSP*):
Функции, использующие базовый тип данных (напр., в качестве аргументов), должны уметь использовать произвольные подтипы этого типа, не зная об этом.

(LSP — один из пяти базовых принципов объектно-ориентированного проектирования SOLID.)

NB. Полиморфизм подтипов означает наследование интерфейсов, наследование в ООП — наследование имплементации. Один тип может быть подтипом неродственных типов в ЯП без множественного наследования.

Пример (Java):

- ArrayList — подтип интерфейсов List, Iterable, Collection, Cloneable, Serializable, RandomAccess;
- ArrayList — подтип классов AbstractList, AbstractCollection и Object.

Пример: полиморфизм подтипов (Java)

```java
abstract class Shape {
    public abstract double area();
}

class Rectangle extends Shape {
    private final double width, height;
    public double area() { return width * height; }
}

class Circle extends Shape {
    private final double radius;
    public double area() { return Math.PI * radius * radius; }
}

Shape[] shapes = new Shape[] {
    new Rectangle(3, 4), new Circle(5)
};
for (Shape shape : shapes) System.out.println(shape.area());
```

Подтипы и наследование

Определение

Наследование (= наследование реализации, *code inheritance*) — перенос для использования, расширения или модификации *реализации* методов класса.

Множественное наследование — копирование реализации из нескольких источников.

ЯП с множественным наследованием: C++, Python.

Определение

Отношение «тип — подтип» (= наследование интерфейса, *subtyping*) — копирование *описания* методов класса для определения совместимости типов.

Промежуточные варианты:

типажи (англ. *traits*), примеси (англ. *mixins*) — ∼ интерфейсы с частично реализованными методами.

ЯП с типажами / примесями: методы в интерфейсах по умолчанию (Java 8); **`trait`** в PHP.

Интерфейсы ООП

Определение

Интерфейс в ООП — способ задания отношения подтипов на основе контрактов (т. е. описания требуемой функциональности).

ЯП, поддерживающие интерфейсы: C#, D, Delphi / Object Pascal, Java, PHP.

Эмуляция интерфейсов: C++ (за счет классов с чистыми виртуальными функциями и множественного наследования).

Примеры интерфейсов (Java):

- Cloneable — указывает на то, что класс поддерживает клонирование;
- Serializable — класс поддерживает сохранение данных;
- Comparable<T> — объекты класса сравнимы с объектами класса T;
- List<T> — класс представляет собой список элементов класса T.

Пример: интерфейсы в Java

```java
// Указывает, что фрукты сравнимы с другими фруктами.
// В результате можно, например, сортировать списки и массивы фруктов
// (методы Arrays.sort, Collections.sort)
// и производить в них бинарный поиск (Arrays.binarySearch, Collections.binarySearch).
public class Fruit implements Comparable<Fruit> {

    // реализация единственного метода интерфейса Comparable
    @Override public int compareTo(Fruit other) {
        return Double.compare(this.weight(), other.weight());
    }

    public double weight() { /* … */ }

    // Другие методы
}

List<Fruit> fruitList = // …
Collections.sort(fruitList);
```

Сравнение наследования и подтипов

		Наследование	
		+	−

Подтип	+	Наследование в Java, C#
		`class A { }`
		`class B extends A { }`
			Расширение интерфейсов в Java, C#
			`intefrace A { }`
			`interface B extends A { }`

	−	Наследование в C++ с модификаторами **protected** и **private**
		`class A { };`
		`class B : protected A { };`
			Независимые типы данных
			`class A { }`
			`class B { }`

Ковариантность и контравариантность

Ковариантность: ParametricType<S> — подтип ParametricType<T>, если S — подтип T.

Примеры использования:

- итераторы (C#):

```
1 // Стандартный интерфейс для перечислимых объектов.
2 interface IEnumerable<out T> { /* */ }
3 // Объект, перечисляющий круги — частный случай объекта, перечисляющего фигуры.
4 // IEnumerable<Circle> — подтип IEnumerable<Shape>.
```

- массивы в C#, Java:

```
1 String[] strings = new String[] { "foo", "bar" };
2 // работает, т.к. String[] — подтип Object[]
3 Arrays.sort(strings);
4 // Ковариантность делает опасной операции записи в массивы
5 Object[] objects = (Object[]) strings;
6 objects[0] = new Integer(5);
7 System.out.println(strings[0]); // исключение времени выполнения
```

Ковариантность и контравариантность

Контравариантность: ParametricType<T> — подтип ParametricType<S>, если S — подтип T.

Пример использования: сравнение (C#):

```
1 // Стандартный интерфейс для сравнимых объектов.
2 interface IComparable<in T> { /* … */ }
3 // Объект, сравнимый с произвольными фигурами — частный случай (расширение)
4 // объекта, сравнимого с кругами.
5 // IComparable<Shape> — подтип IComparable<Circle>.
```

Инвариантность: ParametricType<T> и ParametricType<S> не связаны.

Пример: изменяемые коллекции (C#):

```
1 // Стандартный класс списка.
2 class List<T> {
3     T get(int index); // Подразумевает ковариантное определение класса
4     void add(T element); // Подразумевает контравариантное определение класса
5 }
6 // List<Shape> и List<Circle> не связаны
```

Ковариантность и контравариантность (Java)

Декларации ко/контравариантности в Java: во время использования типа, а не во время его декларации:

- `? extends T` — ковариантное определение типа данных в шаблоне;
- `? super T` — контравариантное определение типа данных в шаблоне.

Пример (ковариантное определение):

```
1  public double totalArea(List<Shape> shapes) {
2      double area = 0.0;
3
4      for (Shape shape : shapes) {
5          area += shape.area();
6      }
7      return area;
8  }
9
10 List<Rectangle> rectangles = // …
11 totalArea(rectangles); // Ошибка: List<Rectangle> нельзя привести к List<Shape>
```

Ковариантность и контравариантность (Java)

Декларации ко/контравариантности в Java: во время использования типа, а не во время его декларации:

- `? extends T` — ковариантное определение типа данных в шаблоне;
- `? super T` — контравариантное определение типа данных в шаблоне.

Пример (ковариантное определение):

```
1  public double totalArea(List<? extends Shape> shapes) {
2      double area = 0.0;
3
4      for (Shape shape : shapes) {
5          area += shape.area();
6      }
7      return area;
8  }
9
10 List<Rectangle> rectangles = // …
11 totalArea(rectangles); // OK: List<Rectangle> — подтип List<? extends Shape>
```

Ковариантность и контравариантность (Java)

Декларации ко/контравариантности в Java: во время использования типа, а не во время его декларации:

- `? extends T` — ковариантное определение типа данных в шаблоне;
- `? super T` — контравариантное определение типа данных в шаблоне.

Пример (контравариантное определение):

```
1  // Общий метод поиска максимального элемента в коллекции
2  public static <T> T max(Collection<T> elements,
3      Comparator<T> comparator) {
4      // comparator используется для сравнения элементов из коллекции
5      // с помощью метода public int comparator.compare(T x, T y);
6  }
7
8  List<Rectangle> rectangles = // …
9  Comparator<Shape> areaComp = // Сравнение фигур по площади
10 max(rectangles, areaComp);
11 // Ошибка: Comparator<Shape> нельзя привести к Comparator<Rectangle>
```

Ковариантность и контравариантность (Java)

Декларации ко/контравариантности в Java: во время использования типа, а не во время его декларации:

- `? extends T` — ковариантное определение типа данных в шаблоне;
- `? super T` — контравариантное определение типа данных в шаблоне.

Пример (контравариантное определение):

```
1  // Общий метод поиска максимального элемента в коллекции
2  public static <T> T max(Collection<T> elements,
3      Comparator<? super T> comparator) {
4      // comparator используется для сравнения элементов из коллекции
5      // с помощью метода public int comparator.compare(T x, T y);
6  }
7
8  List<Rectangle> rectangles = // …
9  Comparator<Shape> areaComp = // Сравнение фигур по площади
10 max(rectangles, areaComp);
11 // OK: Comparator<Shape> — подтип Comparator<? super Rectangle>
```

Выводы

1. Основное применение системы типов данных — устранение ошибок, связанных с некорректной интерпретацией данных (слабая форма верификации программы). Для решения этой задачи вводится понятие совместимых типов данных.
2. Совместимость типов может определяться с помощью явных или неявных приведений, а также с помощью полиморфизма, в частности отношений «тип — подтип».
3. Определение подтипов и наследование в ООП — связанные, но различные понятия. В некоторых случаях определение подтипов нетривиально (например, правила ковариантности / контравариантности для параметрических типов).

Материалы

- Лавріщева К. М.

  Програмна інженерія (підручник).

  K., 2008. — 319 с.

- Tratt, Laurence

  Dynamically typed languages.

  http://tratt.net/laurie/research/pubs/html/tratt__dynamically_typed_languages/

- Cardelli Luca, Wegner Peter

  On understanding types, data abstraction, and polymorphism.

  http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf

Спасибо за внимание!