С примерами на TypeScript

Программируй & типизируй

Влад Ришкуция

MANNING

ББК 32.973.2-018
УДК 004.42
Р57

Ришкуция Влад

Р57 Программируй & типизируй. — СПб.: Питер, 2021. — 352 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1692-8

Причиной многих программных ошибок становится несоответствие типов данных. Сильная система типов позволяет избежать целого класса ошибок и обеспечить целостность данных в рамках всего приложения. Разработчик, научившись мастерски использовать типы в повседневной практике, будет создавать более качественный код, а также сэкономит время, которое потребовалось бы для выискивания каверзных ошибок, связанных с данными.

В книге рассказывается, как с помощью типизации создавать программное обеспечение, которое не только было бы безопасным и работало без сбоев, но также обеспечивало простоту в сопровождении.

Примеры решения задач, написанные на TypeScript, помогут развить ваши навыки работы с типами, начиная от простых типов данных и заканчивая более сложными понятиями, такими как функторы и монады.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 004.42

ISBN 978-1617296413 англ.
ISBN 978-5-4461-1692-8

Оглавление

1 Введение в типизацию

В этой главе

- Зачем нужны системы типов.
- Преимущества сильно типизированного кода.
- Разновидности систем типов.
- Распространенные возможности систем типов.

Аппарат Mars Climate Orbiter развалился в атмосфере Марса, поскольку разработанный компанией Lockheed компонент выдавал измерения импульса силы в фунт-силах на секунду (единицы измерения США), а другой компонент, разработанный НАСА, ожидал, что импульс силы будет измеряться в ньютонах на секунду (единицы СИ). Катастрофы можно было избежать, если бы для этих двух величин использовались различные типы данных.

Как мы будем наблюдать на протяжении данной книги, проверки типов позволяют исключать целые классы ошибок при условии наличия достаточной информации. По мере роста сложности программного обеспечения должны обеспечиваться и лучшие гарантии правильности его работы. Мониторинг и тестирование могут продемонстрировать, ведет ли себя ПО в соответствии со спецификациями в заданный момент времени при определенных входных данных. Типы же обеспечивают более общее подтверждение должного поведения кода, независимо от входных данных.

Благодаря научным изысканиям в области языков программирования возникают все более и более эффективные системы типов (см., например, такие языки

программирования, как Elm и Idris). Растет популярность языка Haskell. В то же время продолжаются попытки добиться проверки типов на стадии компиляции в динамически типизированных языках: в Python появилась поддержка указаний ожидаемых типов (type hints) и был создан язык TypeScript, единственная цель которого — обеспечить проверку типов во время компиляции в JavaScript.

Типизация кода, безусловно, важна, и благодаря полному использованию возможностей системы типов, предоставляемой языком программирования, можно писать лучший, более безопасный код.

1.1. Для кого эта книга

Книга предназначена для программистов-практиков. Читатель должен хорошо уметь писать код на одном из таких языков программирования, как Java, C#, C++ или JavaScript/TypeScript. Примеры кода приведены на языке TypeScript, но большая часть излагаемого материала применима к любому языку программирования. На самом деле в примерах далеко не всегда используется характерный TypeScript. По возможности они адаптировались так, чтобы их понимали программисты на других языках программирования. Сборка примеров кода описана в приложении А, а краткая «шпаргалка» по языку TypeScript — в приложении Б.

Если вы по работе занимаетесь разработкой объектно-ориентированного кода, то, возможно, слышали об алгебраических типах данных (algebraic data type, ADT), лямбда-выражениях, обобщенных типах данных (generics), функторах, монадах и хотите лучше разобраться, что это такое и как их использовать в своей работе.

Эта книга расскажет, как использовать систему типов языка программирования для проектирования менее подверженного ошибкам, более модульного и понятного кода. Вы увидите, как превратить ошибки времени выполнения, которые могут привести к отказу всей системы, в ошибки компиляции и перехватить их, пока они еще не натворили бед.

Основная часть литературы по системам типов сильно формализована. Книга же сосредотачивает внимание на практических приложениях систем типов; поэтому математики в ней очень мало. Тем не менее желательно, чтобы вы имели представление об основных понятиях алгебры, таких как функции и множества. Это понадобится для пояснения некоторых из нужных нам понятий.

1.2. Для чего существуют типы

На низком уровне аппаратного обеспечения и машинного кода логика программы (код) и данные, которыми она оперирует, представлены в виде битов. На этом уровне нет разницы между кодом и данными, так что вполне могут возникнуть ошибки, при которых система путает одно с другим. Их диапазон простирается от фатальных сбоев программы до серьезных уязвимостей, когда злоумышленник обманом заставляет систему считать входные данные кодом, подлежащим выполнению.

Пример подобной нестрогой интерпретации — функция `eval()` языка JavaScript, выполняющая строковое значение как код. Она отлично работает, если переданная ей строка представляет собой допустимый код на языке JavaScript, но вызывает ошибку времени выполнения в противном случае, как показано в листинге 1.1.

Листинг 1.1. Попытка интерпретировать данные как код

```
console.log(eval("40+2"));                    ← Выводит в консоль 42

console.log(eval("Hello world!"));            ← Порождает исключение SyntaxError:
                                                unexpected token: identifier
```

1.2.1. Нули и единицы

Необходимо не только различать код и данные, но и интерпретировать элементы данных. Состоящая из 16 бит последовательность `1100001010100011` может соответствовать беззнаковому 16-битному целому числу `49827`, 16-битному целому числу со знаком `-15709`, символу `'£'` в кодировке UTF-8 или чему-то совершенно другому, как можно видеть на рис. 1.1. Аппаратное обеспечение, на котором работают наши программы, хранит все в виде последовательностей битов, так что необходим дополнительный слой для осмысления этих данных.

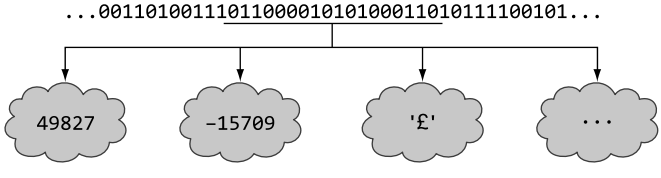


Рис. 1.1. Последовательность битов можно интерпретировать по-разному

Типы придают смысл подобным данным и указывают программному обеспечению, как интерпретировать заданную последовательность битов в установленном контексте, чтобы она сохранила задуманный автором смысл.

Кроме того, типы ограничивают множество допустимых значений переменных. Шестнадцатибитное целое число со знаком может отражать любое из целочисленных значений от `-32768` до `32767` и только их. Благодаря ограничению диапазона допустимых значений исключаются целые классы ошибок, поскольку не допускается возникновения неправильных значений во время выполнения, как показано на рис. 1.2. Чтобы понять многие из приведенных в этой книге концепций, важно рассматривать типы как множества возможных значений.

В разделе 1.3 мы увидим: система обеспечивает также соблюдение многих других мер безопасности при добавлении возможностей в код, например обозначение значений как `const` или членов как `private`.

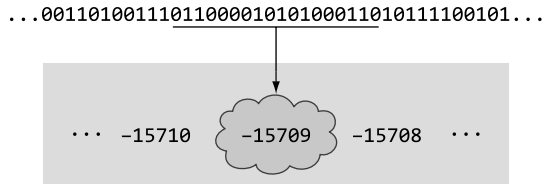


Рис. 1.2. Последовательность битов с типом 16-битного знакового целого. Информация о типе (16-битное знаковое целое число) указывает компилятору и/или среде выполнения, что эта битная последовательность представляет собой целочисленное значение в диапазоне от –32 768 до 32 767, благодаря чему она правильно интерпретируется как –15 709

1.2.2. Что такое типы и их системы

Раз уж книга посвящена типам и их системам, дам определения этих терминов, прежде чем идти дальше.

ЧТО ТАКОЕ ТИП

> Тип (type) — классификация данных, определяющая допустимые операции над ними, смысл этих данных и множество допустимых значений. Компилятор и/или среда выполнения производят проверку типов, чтобы обеспечить целостность данных и соблюдение ограничений доступа, а также интерпретацию данных в соответствии с замыслом разработчика.

В некоторых случаях ради простоты мы будем игнорировать относящуюся к операциям часть этого определения и рассматривать типы просто как множества, отражающие все возможные значения экземпляра данного типа.

СИСТЕМА ТИПОВ

> Система типов (type system) представляет собой набор правил присвоения типов элементам языка программирования и обеспечения соблюдения этих присвоений. Такими элементами могут быть переменные, функции и другие высокоуровневые конструкции языка. Системы типов производят присвоение типов с помощью задаваемой в коде нотации или неявным образом, путем вывода типа конкретного элемента по контексту. Системы типов разрешают одни преобразования типов друг в друга и запрещают другие.

Теперь, когда мы узнали определения типов и систем типов, посмотрим, как обеспечивается соблюдение правил системы типов. На рис. 1.3 показано на высоком уровне выполнение исходного кода.

Если описывать на очень высоком уровне, то создаваемый нами исходный код преобразуется компилятором или интерпретатором в инструкции для машины *(среды выполнения)*. Ее роль может играть физическая машина (в этом случае роль инструкций играют инструкции CPU) или виртуальная с собственным набором инструкций и функций.

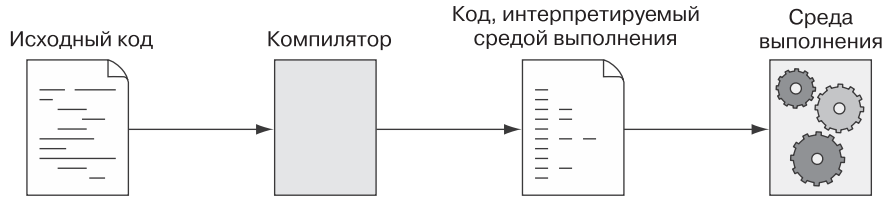


Рис. 1.3. С помощью компилятора или интерпретатора исходный код преобразуется в код, запускаемый средой выполнения. Ее роль может играть физический компьютер или виртуальная машина, например JVM Java или движок JavaScript браузера

ПРОВЕРКА ТИПОВ

Процесс проверки типов (type checking) обеспечивает соблюдение программой правил системы типов. Проверка производится компилятором во время преобразования кода или средой выполнения при его работе. Компонент компилятора, обеспечивающий соблюдение правил типизации, называется модулем проверки типов (type checker).

Если проверка типов завершается неудачно, то есть программа не соблюдает правила системы типов, то возникает ошибка на этапе компиляции или выполнения. Разницу между проверкой типа на этапе компиляции и на этапе выполнения мы обсудим подробнее в разделе 1.4.

Проверка типов и доказательства

В основе систем типов лежит формальная теория. Замечательное соответствие Карри-Ховарда (Curry-Howard correspondence), известное также как эквивалентность между математическими доказательствами и программами (proofs-as-programs), демонстрирует родственность логики и теории типов. Оно показывает, что тип можно рассматривать как логическое высказывание, а функцию, принимающую на входе один тип и возвращающую другой, — как логическую импликацию. Значение типа эквивалентно факту справедливости высказывания.

Возьмем для примера функцию, принимающую на входе `boolean` и возвращающую `string`.

Из булева значения в строковое

```
function booleanToString(b: boolean): string {
    if (b) {
        return "true";
    } else {
        return "false";
    }
}
```

Эту функцию можно интерпретировать как «из `boolean` следует `string`». По заданному факту высказывания типа `boolean` данная функция (импликация) выдает факт

высказывания типа `string`. Факт `boolean` представляет собой значение этого типа, `true` или `false`. По нему указанная функция (импликация) выдает факт `string` в виде строки `"true"` или `"false"`.

Тесная связь между логикой и теорией типов показывает: соблюдающая правила системы типов программа эквивалентна логическому доказательству. Другими словами, система типов — язык написания этих доказательств. Соответствие Карри-Ховарда важно тем, что правильность работы программы гарантируется с логической строгостью.

1.3. Преимущества систем типов

Все данные, по сути, представляют собой нули и единицы, поэтому все свойства данных, например их интерпретация, неизменяемость и видимость, относятся к уровню типа. Переменная объявляется с числовым типом, и модуль проверки типа гарантирует, что ее значение не будет интерпретировано как строковое. Переменная объявляется как приватная или предназначенная только для чтения. И хотя сами данные в памяти ничем не отличаются от аналогичных публичных изменяемых данных, модуль проверки типа гарантирует, что мы не будем обращаться к приватной переменной вне ее области видимости или пытаться изменить данные, предназначенные только для чтения.

Основные преимущества типизации — *корректность* (correctness), *неизменяемость* (immutability), *инкапсуляция* (encapsulation), *компонуемость* (composability) и *читабельность* (readability). Это фундаментальные признаки хорошей архитектуры и нормального поведения программного обеспечения. С течением времени системы развиваются. Эти признаки противостоят энтропии, которая неизбежно возникает в любой системе.

1.3.1. Корректность

Корректным (correct) является код, который ведет себя в соответствии со спецификациями, выдает ожидаемые результаты без ошибок и сбоев во время выполнения. Благодаря типам растут строгость кода и гарантии его должного поведения.

Для примера предположим, что нам нужно найти позицию строки `"Script"` внутри другой строки. Мы не будем передавать достаточную информацию о типе и разрешим передачу в качестве аргумента нашей функции значения типа `any`. Как показывает листинг 1.2, это приведет к ошибкам во время выполнения.

В этой программе содержится ошибка — `42` не является допустимым аргументом для функции `scriptAt`, но компилятор об этом молчит, поскольку мы не предоставили достаточную информацию о типе данных. Усовершенствуем данный код, ограничив аргумент типом `string` в листинге 1.3.

Теперь компилятор отвергает эту некорректную программу, выдавая следующее сообщение об ошибке: `Argument of type '42' is not assignable to parameter of type 'string'` (невозможно присвоить параметру типа `'string'` аргумент типа `'42'`).

Листинг 1.2. Недостаточная информация о типе данных

```
function scriptAt(s: any): number {
    return s.indexOf("Script");
}

console.log(scriptAt("TypeScript"));
console.log(scriptAt(42));
```

Тип аргумента s — any, то есть разрешается значение произвольного типа

Эта строка выводит в консоль корректное значение 4

Передача в качестве аргумента числового значения приводит к TypeError во время выполнения

Листинг 1.3. Уточненная информация о типе

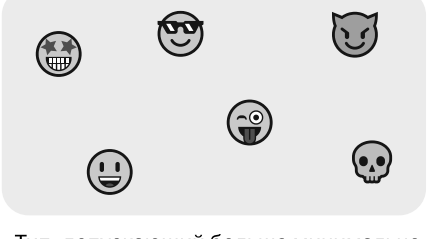
```
function scriptAt(s: string): number {
    return s.indexOf("Script");
}

console.log(scriptAt("TypeScript"));
console.log(scriptAt(42));
```

Теперь у аргумента s тип — string

Код не компилируется и выдает ошибку компиляции на данной строке вследствие несовпадения типов

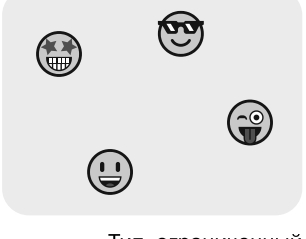
Воспользовавшись системой типов, мы из проблемы времени выполнения, которая могла проявиться при промышленной эксплуатации (и повлиять на наших клиентов), сделали безобидную проблему этапа компиляции, которую просто нужно исправить перед развертыванием кода. Модуль проверки типа гарантирует, что яблоки не будут передаваться в качестве апельсинов; а значит, растет ошибкоустойчивость кода.

Ошибки возникают, когда программа переходит в *некорректное состояние*, то есть текущее сочетание всех ее действующих переменных некорректно по какой-либо причине. Один из методов, позволяющих избавиться от части таких некорректных состояний, — уменьшение пространства состояний за счет ограничения количества возможных значений переменных, как показано на рис. 1.4.



Тип, допускающий больше минимально необходимого количества значений

x = 💀 ; // Плохо



Тип, ограниченный только корректными значениями

x = 💀 ; // Ошибка компиляции

Рис. 1.4. Благодаря правильному объявлению типа можно запретить некорректные значения. Первый тип слишком широк и допускает нежелательные нам значения. Второй тип — более жестко ограниченный — не скомпилируется, если код попытается присвоить переменной нежелательное значение

Пространство состояний (state space) работающей программы можно описать как сочетание всех вероятных значений всех ее действующих переменных. То есть декартово произведение типов всех переменных. Напомню, что тип переменной можно рассматривать как множество ее возможных значений. Декартово произведение двух множеств представляет собой множество, состоящее из всех их упорядоченных пар элементов.

БЕЗОПАСНОСТЬ

Важный побочный результат запрета на потенциальные некорректные состояния — повышение безопасности кода. В основе множества атак лежит выполнение передаваемых пользователем данных, переполнение буфера и другие подобные методики, опасность которых нередко можно уменьшить за счет достаточно сильной системы типов и хороших определений типов.

Корректность кода не исчерпывается исправлением невинных ошибок в коде с целью предотвратить атаки злоумышленников.

1.3.2. Неизменяемость

Неизменяемость (immutability) — еще одно свойство, тесно связанное с представлением о нашей работающей системе как о движении по пространству состояний. Вероятность ошибок можно снизить, если при нахождении системы в заведомо хорошем состоянии не допускать его изменений.

Рассмотрим простой пример, в котором попытаемся предотвратить деление на ноль с помощью проверки значения делителя и генерации ошибки в случае, когда оно равно `0`, как показано в листинге 1.4. Если же значение может меняться после нашей проверки, то она теряет всякий смысл.

Листинг 1.4. «Плохое» изменение значения

```
function safeDivide(): number {
    let x: number = 42;

    if (x == 0) throw new Error("x should not be 0");   <- Проверяем допустимость x

    x = x - 42;                                         <- Ошибка в программе:
                                                           после проверки x становится равен 0

    return 42 / x;                                      <- Деление на 0 приводит
}                                                          к значению Infinity[1]
```

В настоящих программах подобное случается регулярно, причем часто довольно неожиданным образом: переменная меняется, скажем, конкурентным потоком выполнения или другой вызванной функцией. Как и в этом примере, сразу после изменения значения все гарантии, которые мы надеялись получить от наших проверок,

[1] Стандартный встроенный объект JavaScript (и TypeScript), олицетворяет бесконечное значение. — *Примеч. пер.*

теряются. Если же сделать x константой, как в листинге 1.5, то компилятор вернет ошибку при попытке изменить ее значение.

Листинг 1.5. Неизменяемость

```
function safeDivide(): number {
    const x: number = 42;

    if (x == 0) throw new Error("x should not be 0");

    x = x - 42;

    return 42 / x;
}
```

x объявляется с указанием ключевого слова const вместо let

Эта строка больше не компилируется, поскольку x — неизменяемая и повторное присвоение ей значения недопустимо

Теперь компилятор отвергает некорректный код, выводя следующее сообщение об ошибке: `Cannot assign to 'x' because it is a constant` (Присвоение значения переменной x невозможно, поскольку она является константой).

В смысле представления в оперативной памяти разницы между изменяемой и неизменяемой x нет. Свойство константности значит что-то только для компилятора. Это свойство, обеспечиваемое системой типов.

Указание на неизменяемость состояния с помощью добавления ключевого слова `const` в описание типа предотвращает те изменения значений, при которых теряются гарантии, полученные благодаря предыдущим проверкам. Особенно полезна неизменяемость в случае конкурентного выполнения, поскольку делает невозможной состояние гонки.

Оптимизация компиляторов обеспечивает выдачу более эффективного кода в случае неизменяемых переменных, так как их значения можно встраивать в код. В некоторых функциональных языках программирования все данные — неизменяемые: функции принимают на входе какие-либо данные и возвращают другие, никогда не меняя входных. При этом достаточно один раз проверить значение переменной и убедиться в ее хорошем состоянии с целью гарантировать, что она будет находиться в хорошем состоянии на протяжении всего жизненного цикла. Конечно, при этом приходится идти на (не всегда желательный) компромисс: копировать данные, с которыми в противном случае можно было бы работать, не прибегая к дополнительным структурам данных.

Впрочем, не всегда имеет смысл делать все данные неизменяемыми. Тем не менее неизменяемость как можно большего числа данных может резко снизить вероятность возникновения таких проблем, как несоответствие заранее заданным условиям и состояние гонки по данным.

1.3.3. Инкапсуляция

Инкапсуляция (encapsulation) — сокрытие части внутреннего устройства кода в функции, классе или модуле. Как вы, вероятно, знаете, инкапсуляция — желательное свойство, она помогает понижать сложность: код разбивается на меньшие компоненты, каждый из которых предоставляет доступ только к тому, что действительно нужно, а подробности реализации скрываются и изолируются.

В листинге 1.6 мы расширим пример безопасного деления, превратив его в класс, который старается гарантировать отсутствие деления на `0`.

Листинг 1.6. Недостаточная инкапсуляция

```
class SafeDivisor {
    divisor: number = 1;

    setDivisor(value: number) {
        if (value == 0) throw new Error("Value should not be 0");

        this.divisor = value;
    }

    divide(x: number): number {
        return x / this.divisor;
    }
}

function exploit(): number {
    let sd = new SafeDivisor();

    sd.divisor = 0;
    return sd.divide(42);
}
```

Проверяем значение перед присваиванием, чтобы гарантировать ненулевой делитель

Деления на 0 не должно быть

Поскольку член класса divisor — публичный, проверку можно обойти

В результате деления на 0 возвращается Infinity

В данном случае мы не можем сделать делитель неизменяемым, поскольку хотим, чтобы у вызывающего наш API кода была возможность его обновлять. Проблема такова: вызывающая сторона может обойти проверку на `0` и непосредственно задать любое значение для `divisor`, так как он для них доступен. Эту проблему в данном случае можно решить, объявив его в качестве `private` и ограничив его область видимости классом, как показано в листинге 1.7.

Листинг 1.7. Инкапсуляция

```
class SafeDivisor {
    private divisor: number = 1;

    setDivisor(value: number) {
        if (value == 0) throw new Error("Value should not be 0");
        this.divisor = value;
    }

    divide(x: number): number {
        return x / this.divisor;
    }
}

function exploit() {
    let sd = new SafeDivisor();

    sd.divisor = 0;
    sd.divide(42);
}
```

Теперь этот член класса стал приватным

Данная строка не скомпилируется, поскольку на divisor больше нельзя ссылаться вне класса

Представление в оперативной памяти приватных и публичных членов класса одинаково; проблемный код не компилируется во втором примере просто благодаря указанию типа. На самом деле `public`, `private` и другие модификаторы видимости — свойства соответствующего типа.

Инкапсуляция (сокрытие информации) позволяет разбивать логику программы и данные на публичный интерфейс и непубличную реализацию. Это очень удобно в больших системах, поскольку при работе с интерфейсами (абстракциями) требуется меньше умственных усилий, чтобы понять конкретный фрагмент кода. Желательно анализировать и понимать код на уровне интерфейсов компонентов, а не всех их нюансов реализации. Полезно также ограничивать область видимости непубличной информации, чтобы внешний код не мог их модифицировать попросту вследствие отсутствия доступа.

Инкапсуляция существует на множестве уровней: сервис предоставляет доступ к своему API в виде интерфейса, модуль экспортирует свой интерфейс и скрывает нюансы реализации, класс делает видимыми только публичные члены класса и т. д. Чем слабее связь между двумя частями кода, тем меньший объем информации они разделяют. Благодаря этому усиливаются гарантии компонента относительно его внутренних данных, поскольку никакой внешний код не может их модифицировать, не прибегая к использованию интерфейса компонента.

1.3.4. Компонуемость

Допустим, нам требуется найти первое отрицательное число в числовом массиве и первую строку из одного символа в символьном массиве. Не прибегая к разбиению этой задачи на две части и последующему их объединению в единую систему, мы получили бы в итоге две функции: `findFirstNegativeNumber()` и `findFirstOneCharacterString()`, показанные в листинге 1.8.

Листинг 1.8. Некомпонуемая система

```
function findFirstNegativeNumber(numbers: number[])
    : number | undefined {
    for (let i of numbers) {
        if (i < 0) return i;
    }
}

function findFirstOneCharacterString(strings: string[])
    : string | undefined {
    for (let str of strings) {
        if (str.length == 1) return str;
    }
}
```

Эти две функции ищут первое отрицательное число и первую строку из одного символа соответственно. Если подобных элементов не найдено, то функции возвращают `undefined` (неявно, путем выхода из функции без оператора `return`).