
Оглавление

Отзывы	10
Пролог	12
Структура книги	12
Стиль	13
Использование примеров кода	15
Благодарности.....	16
От издательства	17
Глава 1. Вступление.....	18
Глава 2. TypeScript с высоты птичьего полета	21
Компилятор	21
Система типов	23
Настройка редактора кода.....	27
index.ts	30
Упражнения к главе 2.....	32
Глава 3. Подробно о типах	33
О типах	34
Типы от а до я.....	35
Итоги.....	65
Упражнения к главе 3.....	66
Глава 4. Функции	67
Объявление и вызов функций.....	67
Полиморфизм.....	90
Разработка на основе типов	108
Итоги.....	109
Упражнения к главе 4.....	110

Глава 5. Классы и интерфейсы	111
Классы и наследование	111
super	117
Использование this в качестве возвращаемого типа	117
Интерфейсы	119
Классы структурно типизированы	126
Классы объявляют и значения, и типы	127
Полиморфизм	131
Примеси	132
Декораторы	135
Имитация финальных классов	138
Паттерны проектирования	139
Итоги	142
Упражнения к главе 5	144
Глава 6. Продвинутое типы	145
Связи между типами	145
Тотальность	165
Продвинутое типы объектов	167
Продвинутое функциональные типы	177
Условные типы	180
Запасные решения	185
Имитация номинальных типов	191
Безопасное расширение прототипа	193
Итоги	196
Упражнения к главе 6	197
Глава 7. Обработка ошибок	198
Возврат null	199
Выбрасывание исключений	200
Возврат исключений	203

Тип Option	205
Итоги.....	213
Упражнение к главе 7.....	214
Глава 8. Асинхронное программирование, конкурентность и параллельная обработка.....	215
Цикл событий	216
Работа с обратными вызовами.....	218
Промисы как здоровая альтернатива.....	221
asunc и await	227
Asunc-поток	228
Типобезопасная многопоточность	231
Итоги.....	245
Упражнения к главе 8.....	246
Глава 9. Фронтенд- и бэкенд-фреймворки	247
Фронтенд-фреймворки	247
Типобезопасные API	260
Бэкенд-фреймворки.....	262
Итоги.....	264
Глава 10. Пространства имен и модули	265
Краткая история модулей JavaScript	266
import, export.....	269
Пространства имен	274
Слияние деклараций	279
Итоги.....	281
Упражнение к главе 10.....	282
Глава 11. Взаимодействие с JavaScript	283
Декларации типов	283
Поэтапная миграция из JavaScript в TypeScript	292
Поиск типов для JavaScript	298

Использование стороннего кода JavaScript	301
Итоги.....	305
Глава 12. Создание и запуск TypeScript	306
Создание проекта в TypeScript	306
Запуск TypeScript на сервере	317
Запуск TypeScript в браузере	318
Публикация TypeScript-кода на NPM.....	321
Директивы с тремя слешами.....	322
Итоги.....	326
Глава 13. Итоги.....	327
Приложение А. Операторы типов	329
Приложение Б. Утилиты типов.....	330
Приложение В. Область действия деклараций	331
Генерирует ли декларация тип	331
Допускает ли декларация слияние.....	331
Приложение Г. Правила написания файлов деклараций для сторонних модулей JavaScript.....	333
Типы экспорта.....	334
Расширение модуля.....	337
Приложение Д. Директивы с тремя слешами	342
Внутренние директивы	343
Нежелательные директивы.....	343
Приложение Е. Флаги безопасности компилятора TSC	344
Приложение Ж. TSX	346
Об авторе	349
Об обложке	350

Обработка ошибок

Физик, инженер и программист ехали в машине, как вдруг в ней отказали тормоза. Произошло это на крутом альпийском перевале. Машина продолжала набирать скорость, водитель изо всех сил пытался вписаться в повороты, и от падения спасали лишь оградительные барьеры. Пассажиры были перед лицом неминуемой гибели и уже смирились со своей участью, но внезапно показалась аварийная полоса, на которой удалось благополучно остановиться.

Физик сказал: «Нужно смоделировать трение в тормозных колодках и вызванный этим рост температуры. Так мы сможем разобраться, что произошло».

Инженер продолжил: «У меня есть с собой кой-какие инструменты. Пойду покопаюсь в колодках».

На что программист ответил им: «А давайте проверим воспроизводимость?»

Аноним

TypeScript предоставляет массу возможностей, позволяющих сместить ошибки среды выполнения в среду компиляции: начиная с богатой системы типов и заканчивая мощным статическим и символическим анализом. Он работает изо всех сил, чтобы вам не пришлось коротать пятничные вечера за исправлением опечаток в именах переменных и исключений нулевых указателей (и чтобы ваш коллега не опоздал из-за этого на день рождения двоюродной бабушки).

К несчастью, независимо от того, в каком языке вы работаете, иногда исключения все же просачиваются в среду выполнения. TypeScript не под силу препятствовать сбоям в сети и файловой системе, ошибкам обработки пользовательского ввода, переполнению стека или недостаточной памяти. Тем не менее его отличная система типов, несомненно, помогает справляться с ошибками выполнения.

В этой главе я познакомлю вас с наиболее распространенными паттернами представления и обработки ошибок в TypeScript, такими как:

- ❑ Возврат `null`.
- ❑ Выбрасывание исключений.
- ❑ Возврат исключений.
- ❑ Тип `Option`.

Выбор тех или иных механизмов остается за вами и зависит от вашего приложения, а я покажу их плюсы и минусы.

Возврат null

Создадим программу, спрашивающую пользователя о его дне рождения, чтобы интерпретировать эту дату в объект `Date`:

```
function ask() {  
    return prompt('When is your birthday?')  
}
```

```
function parse(birthday: string): Date {  
    return new Date(birthday)  
}
```

```
let date = parse(ask())  
console.info('Date is', date.toISOString())
```

Вероятно, стоит проверить указанную пользователем дату, ведь это текстовый запрос:

```
// ...  
function parse(birthday: string): Date | null {  
    let date = new Date(birthday)  
    if (!isValid(date)) {  
        return null  
    }  
    return date  
}
```

```
// Проверка допустимости указанной даты
function isValid(date: Date) {
    return Object.prototype.toString.call(date) === '[object Date]'
        && !Number.isNaN(date.getTime())
}
```

Получив результат, первым делом проверим, не является ли он `null`, и только потом применим его:

```
// ...
let date = parse(ask())
if (date) {
    console.info('Date is', date.toISOString())
} else {
    console.error('Error parsing date for some reason')
}
```

Возврат `null` — это наиболее легковесный способ обработки ошибок в типобезопасном режиме. Допустимые данные будут представлены как `Date`, а недопустимые — как `null`. Система типов проверит, чтобы оба варианта были обработаны.

Но в этом подходе мы упускаем некоторую информацию, ведь `parse` не сообщает точную причину сбоя операции. Из-за этого инженер, производящий отладку, будет вынужден копаться в логах, а пользователь получит всплывающее сообщение «Ошибка обработки даты по неизвестной причине» вместо «Введите дату в виде ГГГГ/ММ/ДД».

Еще возврат `null` нелегко реализовать: необходимо делать проверку на `null` после каждой операции. Это может вызвать многословность, поскольку именно вам придется делать вложение этих операций и создавать их цепочки.

Выбрасывание исключений

Давайте вместо возврата `null` выбросим исключение, чтобы обработать конкретные признаки отказа и получить необходимые метаданные для упрощения процесса отладки.

```
// ...
function parse(birthday: string): Date {
    let date = new Date(birthday)
```

```
    if (!isValid(date)) {
        throw new RangeError('Enter a date in the form YYYY/MM/DD')
    }
    return date
}
```

Теперь при использовании этого кода можно с осторожностью перехватывать исключения, чтобы их обработать, не порушив все приложение:

```
// ...
try {
    let date = parse(ask())
    console.info('Date is', date.toISOString())
} catch (e) {
    console.error(e.message)
}
```

Следует внимательно перебросить другие исключения, чтобы не упустить возможные ошибки:

```
// ...
try {
    let date = parse(ask())
    console.info('Date is', date.toISOString())
} catch (e) {
    if (e instanceof RangeError) {
        console.error(e.message)
    } else {
        throw e
    }
}
```

Можно выделить подкласс более конкретных ошибок, чтобы, когда другой инженер изменит `parse` или `ask` для выбрасывания других `RangeError`, отличить наши ошибки от его ошибок:

```
// ...

// Кастомизированные типы ошибок
class InvalidDateFormatError extends RangeError {}
class DateIsInTheFutureError extends RangeError {}

function parse(birthday: string): Date {
```



```
let date = new Date(birthday)
if (!isValid(date)) {
  throw new InvalidDateFormatError('Enter a date in the form
  YYYY/MM/DD')
}
if (date.getTime() > Date.now()) {
  throw new DateIsInTheFutureError('Are you a timelord?')
}
return date
}

try {
  let date = parse(ask())
  console.info('Date is', date.toISOString())
} catch (e) {
  if (e instanceof InvalidDateFormatError) {
    console.error(e.message)
  } else if (e instanceof DateIsInTheFutureError) {
    console.info(e.message)
  } else {
    throw e
  }
}
```

Выглядит неплохо. Теперь можно не только выдавать неконкретный сигнал о сбое, но и использовать кастомизированные ошибки для отображения причины сбоя. Это полезно при вычитывании серверных логов во время отладки или при выдаче пользователям диалоговых окон, содержащих данные о неверных действиях и рекомендации по их устранению. Мы сможем эффективно строить цепочки и делать вложения операций, оборачивая любое их число в один `try... catch` (без проверки после каждой операции).

Удобно ли использовать такой код? Представьте, что большой `try... catch` находится в одном файле, а оставшаяся часть кода — в библиотеке, импортированной извне. Откуда инженер узнает о необходимости перехвата конкретных видов ошибок (`InvalidDateFormatError` и `DateInTheFutureError`) или просто о том, что нужно проверить `RangeError`? (Вспомните, что TypeScript не кодирует исключения как часть сигнатуры функции.) Можно указать это в имени функции (`parseThrows`) или включить в документацию:

```
/**
 * @throws {InvalidDateFormatError} Пользователь некорректно ввел
 дату рождения.
 * @throws {DateIsInTheFutureError} Пользователь ввел дату рождения
 из будущего.
 */
function parse(birthday: string): Date {
  // ...
}
```

Но на практике инженер не стал бы оборачивать этот код в `try... catch` и вовсе не проверял бы его на исключения, потому что инженеры ленивы (я — точно), а система типов не сообщает им, что они что-то упустили. Однако иногда, как в нашем примере, ошибки настолько ожидаемы, что последующий код действительно должен их обрабатывать, чтобы они не привели к сбою всей программы.

Как еще мы можем указать потребителям, что они должны обработать случаи как успеха, так и провала?

Возврат исключений

TypeScript не Java, и он не поддерживает спецификаторы `throws`¹. Но мы можем смоделировать их возможности с помощью типов объединений:

```
// ...
function parse(
  birthday: string
): Date | InvalidDateFormatError | DateIsInTheFutureError {
  let date = new Date(birthday)
  if (!isValid(date)) {
    return new InvalidDateFormatError('Enter a date in the form
    YYYY/MM/DD')
  }
  if (date.getTime() > Date.now()) {
    return new DateIsInTheFutureError('Are you a timelord?')
  }
  return date
}
```

¹ В Java спецификаторы `throws` указывают, какие типы исключений среды выполнения может выбросить метод, а потребитель должен обработать.

Теперь потребитель вынужден обработать все три случая: `InvalidDateFormatError`, `DateIsInTheFutureError` и удачное считывание. В противном случае при компиляции появится `TypeError`:

```
// ...
let result = parse(ask()) // Либо дата, либо ошибка.
if (result instanceof InvalidDateFormatError) {
  console.error(result.message)
} else if (result instanceof DateIsInTheFutureError) {
  console.info(result.message)
} else {
  console.info('Date is', result.toISOString())
}
```

Мы успешно воспользовались преимуществами системы типов, чтобы:

- ❑ Кодировать вероятные исключения в сигнатуре `parse`.
- ❑ Сообщить потребителям, какие конкретно исключения могут возникнуть.
- ❑ Вынудить потребителей обработать (или перебросить) каждое из исключений.

Ленивые потребители могут избежать обработки каждой отдельной ошибки, но им придется сделать это явно:

```
// ...
let result = parse(ask()) // Либо дата, либо ошибка.
if (result instanceof Error) {
  console.error(result.message)
} else {
  console.info('Date is', result.toISOString())
}
```

Конечно, программа по-прежнему может дать сбой из-за недостаточной памяти или исключения переполнения стека, но для подобных случаев нет эффективных решений.

Возврат исключений — это тоже легковесный подход, который не требует мудреных структур данных, но при этом он достаточно информативен, чтобы потребители понимали, какой вид сбоя представляет ошибка и куда обратиться для получения дополнительной информации.

Обратная же сторона в том, что связывание в цепочку и вложение операций, выдающих ошибки, может превратиться в громоздкий код. Если функция возвращает `T | Error`, то любая ее функция-потребитель имеет два выхода.

1. Явно обработать `Error1`.
2. Обработать `T` (успешный случай) и передать `Error1` далее для обработки ее потребителями. Если делать это в достаточном объеме, список ошибок, требующих обработки потребителем, быстро вырастет:

```
function x(): T | Error1 {
  // ...
}
function y(): U | Error1 | Error2 {
  let a = x()
  if (a instanceof Error) {
    return a
  }
  // Сделать что-нибудь с a
}
function z(): U | Error1 | Error2 | Error3 {
  let a = y()
  if (a instanceof Error) {
    return a
  }
  // Сделать что-нибудь с a
}
```

Это громоздкий код, но он дает высокий уровень безопасности.

Тип Option

Исключения можно описывать с помощью особых типов данных. У этого подхода есть свои проблемы (в частности, код может не распознавать особые типы), но он дает возможность связывать цепочки операций над потенциально ошибочными вычислениями. Три наиболее популярные опции — это типы `Try`, `Option`¹ и `Either`. В этой главе мы рассмотрим только тип `Option`², потому что остальные типы с ним схожи.

¹ Альтернативное название — тип `Maybe`.

² Погуглите «тип `try`» или «тип `either`» для более подробного ознакомления с ними.