Оглавление

ГЛАВА 7

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.