



Superstruct

Javascript 运行时接口验证工具

设计原则

- 可组合的
- 可自定义
- 有用的运行时错误
- 语法自然



为什么需要运行时验证

众所周知，JS是一门对隐式类型转换容忍度极高的语言 一不注意就会写出意想不到的代码，比如

```
1  const a = '1';
2  const b = 1;
3
4  function add(x, y) {
5      return (x + y)/2;
6  }
7
8  // some code ...
9  add(a,b)
```

我们这里期待函数应当是 `(x:number, y:number) ⇒ number` 但是实参是字符串和数字，并且数字和字符串可以相互隐式转换，从而导致意料之外的结果，而这种错误不像某些语言，是不会通过抛出的类型错误暴露给用户的

Superstruct

因此，我们需要一个工具来帮助我们验证运行时的接口,Superstruct就是为了应对这种情况而产生的工具

基本类型

最简单的情况就是基本类型的验证

```
1  import { string } from 'superstruct'
2
3  const Struct = string()
4
5  assert('a string', Struct) // passes
6  assert(42, Struct) // throws!
```

这里`assert`充当的是一个断言函数，如果和Struct不匹配，那么将会抛出一个StructError的运行时错误

错误描述

```
1 file:///home/bjorn/test/fuccc/node_modules/superstruct/lib/index.es.js:385
2   const error = new StructError(tuple[0], function* () {
3
4   StructError: Expected a string, but received: 42
5     at validate (file:///home/bjorn/test/fuccc/node_modules/superstruct/lib/index.es.js:385:19)
6     at assert (file:///home/bjorn/test/fuccc/node_modules/superstruct/lib/index.es.js:326:18)
7     at file:///home/bjorn/test/fuccc/examples/assert.mjs:6:1
8     at ModuleJob.run (node:internal/modules/esm/module_job:198:25)
9     at async Promise.all (index 0)
10    at async ESMLoader.import (node:internal/modules/esm/loader:385:24)
11    at async loadESM (node:internal/process/esm_loader:88:5)
12    at async handleMainPromise (node:internal/modules/run_main:61:12) {
13  value: 42,
14  key: undefined,
15  type: 'string',
16  refinement: undefined,
17  path: [],
18  branch: [ 42 ],
19  failures: [Function (anonymous)]
20 }
```

其他类型

除了基本类型意外还有其他高级类型供用户使用

- any
- bigint
- date
- func
- set
- map
- optional
- ...

将基本类型组合，来构造更复杂的类型

```
1  const User = object({
2    id: tt(),
3    email: string(),
4    name: string(),
5  })
6
7
8  // passes
9  assert(
10   {
11     id: 1,
12     email: 'jane@example.com',
13     name: 'Jane',
14   },
15   User
16 )
```

将会抛出一个运行时错误

```
1  // also throws! (email is missing)
2  assert(
3    {
4      id: 1,
5      name: 'Jane',
6    },
7    User
8  )
```


可选类型

可以通过optional函数来指定某个属性可选

```
1  import { optional } from "superstruct"
2  const User = object({
3    id: number(),
4    name: string(),
5    email: optional(string()), //可选属性
6  })
```

`define`

除了object,superstruct还可以使用`define<T>(name: string, validator: Validator): Struct<T, null>`进行更细粒度的验证

```
1 import { define } from 'superstruct'
2 import isEmail from 'is-email'
3
4 const email = () => define('email', (value) => isEmail(value))
```

修改输入的数据

有时候，为了让数据通过验证，我们需要对数据的数据做处理（类型转换，计算，trim...）

默认值

superstruct提供了 ``defaulted`` 函数来完成这一功能

```
1  import { defaulted, create } from 'superstruct'
2
3  let i = 0
4
5  const User = object({
6    id: defaulted(number(), () => i++),
7    email: string(),
8    name: string(),
9  })
10
11  const data = {
12    name: 'Jane',
13    email: 'jane@example.com',
14  }
15
16  const user = create(data, User)
```

除了提供默认值，还可以转换输入数据

```
1  import {  
2    coerce,  
3    number,  
4    string,  
5    create  
6  } from 'superstruct'  
7  
8  const MyNumber = coerce(  
9    number(),  
10   string(),  
11   (value) => parseFloat(value)  
12 )
```

运行结果

```
1  import { create } from 'superstruct'  
2  
3  const data = '3.14'  
4  const output = create(data, MyNumber)  
5  // 3.14
```

Typescript支持

配合typescript 使用，需要激活strictNullChecks获取设置"strict"选项,以便支持optional方法的使用

这里`is`可以充当类型保护,它会让TS自动推断此处的`data`是`User`类型,和`assert`不同的是,它会返回一个布尔值,而非直接抛出一个错误

```
1  const User = object({
2    id: number(),
3    email: email(),
4    name: string(),
5  })
6
7  if (is(data, User)) {
8    // TS会自动推断出块内的代码是User类型, TS已经知道data通过了类型校验
9  }
```

`assert` Function

```
1  export function assert<T, S>(
2    value: unknown,
3    struct: Struct<T, S>
4  ): asserts value is T {
5    const result = validate(value, struct)
6
7    if (result[0]) {
8      throw result[0]
9    }
10 }
```

`is` Function

```
1  export function is<T, S>(
2    value: unknown, struct: Struct<T, S>
3  ): value is T {
4    const result = validate(value, struct)
5    return !result[0]
6  }
```


可以使用TS定义的类型来确保正确的属性类型

```
1  import { Describe } from 'superstruct'
2  type User = {
3    id: number
4    name: string
5  }
6
7  const User: Describe<User> = object({
8    id: string(), // 不会通过类型检查, 应该为`number`
9    name: string(),
10  })
```

TS也可以从superstruct定义的对象构造类型

```
1  import { Infer } from 'superstruct'
2
3  const User = object({
4    id: number(),
5    email: email(),
6    name: string(),
7  })
8
9  type User = Infer<typeof User>
```