yiminghe / async-validator

validate form asynchronous

92 commits | 2 branches | 40 releases | 15 contributors | MIT

Branch: master | New pull request | Find file | Clone or download

yiminghe fix unicode length | Latest commit `062470d` 4 days ago

__tests__	add unicode test case	5 days ago
examples	fix date	3 years ago
src	修改对Unicode码点大于U+FFFF的字符求长度的testcase	11 days ago
.editorconfig	1.0	3 years ago
.gitignore	allow message for required. Fixes #23	2 years ago
.travis.yml	bump	11 months ago
HISTORY.md	bump for promise	11 months ago
LICENSE.md	Added LICENSE.md	2 years ago
README.md	bump for promise	11 months ago
index.js	fix es	a year ago
package-lock.json	add Unicode test case	11 days ago
package.json	fix unicode length	4 days ago

README.md

async-validator

Validate form asynchronous. A variation of https://github.com/freeformsystems/async-validate

npm v1.8.4 | build passing | Test coverage | gemnasium no longer available | node.js >=4.0.0 | downloads 479k/month

API

The following is modified from earlier version of async-validate.

Usage

Basic usage involves defining a descriptor, assigning it to a schema and passing the object to be validated and a callback function to the `validate` method of the schema:

```
var schema = require('async-validator');
var descriptor = {
  name: {type: "string", required: true}
}
var validator = new schema(descriptor);
validator.validate({name: "muji"}, (errors, fields) => {
  if(errors) {
    // validation failed, errors is an array of all errors
    // fields is an object keyed by field name with an array of
    // errors per field
    return handleErrors(errors, fields);
  }
  // validation passed
});
```

Validate

```
function(source, [options], callback)
```

- `source` : The object to validate (required).
- `options` : An object describing processing options for the validation (optional).
- `callback` : A callback function to invoke when validation completes (required).

Options

- `first` : Boolean, Invoke `callback` when the first validation rule generates an error, no more validation rules are processed. If your validation involves multiple asynchronous calls (for example, database queries) and you only need the first error use this option.

- `firstFields` : Boolean|String[], Invoke `callback` when the first validation rule of the specified field generates an error, no more validation rules of the same field are processed. `true` means all fields.

Rules

Rules may be functions that perform validation.

```
function(rule, value, callback, source, options)
```

- `rule` : The validation rule in the source descriptor that corresponds to the field name being validated. It is always assigned a `field` property with the name of the field being validated.
- `value` : The value of the source object property being validated.
- `callback` : A callback function to invoke once validation is complete. It expects to be passed an array of `Error` instances to indicate validation failure.
- `source` : The source object that was passed to the `validate` method.
- `options` : Additional options.
- `options.messages` : The object containing validation error messages, will be deep merged with defaultMessages.

The options passed to `validate` are passed on to the validation functions so that you may reference transient data (such as model references) in validation functions. However, some option names are reserved; if you use these properties of the options object they are overwritten. The reserved properties are `messages` , `exception` and `error` .

```
var schema = require('async-validator');
var descriptor = {
  name(rule, value, callback, source, options) {
    var errors = [];
    if(!/^[a-z0-9]+$/.test(value)) {
      errors.push(
        new Error(
          util.format("%s must be lowercase alphanumeric characters",
            rule.field)));
    }
    callback(errors);
  }
}
var validator = new schema(descriptor);
validator.validate({name: "Firstname"}, (errors, fields) => {
  if(errors) {
    return handleErrors(errors, fields);
  }
  // validation passed
});
```

It is often useful to test against multiple validation rules for a single field, to do so make the rule an array of objects, for example:

```
var descriptor = {
  email: [
    {type: "string", required: true, pattern: schema.pattern.email},
    {validator(rule, value, callback, source, options) {
      var errors = [];
      // test if email address already exists in a database
      // and add a validation error to the errors array if it does
      callback(errors);
    }}
  ]
}
```

Type

Indicates the `type` of validator to use. Recognised type values are:

- `string` : Must be of type `string`. `This is the default type.`
- `number` : Must be of type `number`.
- `boolean` : Must be of type `boolean`.
- `method` : Must be of type `function`.
- `regexp` : Must be an instance of `RegExp` or a string that does not generate an exception when creating a new `RegExp`.
- `integer` : Must be of type `number` and an integer.
- `float` : Must be of type `number` and a floating point number.
- `array` : Must be an array as determined by `Array.isArray`.
- `object` : Must be of type `object` and not `Array.isArray`.
- `enum` : Value must exist in the `enum`.
- `date` : Value must be valid as determined by `Date`
- `url` : Must be of type `url`.
- `hex` : Must be of type `hex`.
- `email` : Must be of type `email`.

Required

The `required` rule property indicates that the field must exist on the source object being validated.

Pattern

The `pattern` rule property indicates a regular expression that the value must match to pass validation.

Range

A range is defined using the `min` and `max` properties. For `string` and `array` types comparison is performed against the `length`, for `number` types the number must not be less than `min` nor greater than `max`.

Length

To validate an exact length of a field specify the `len` property. For `string` and `array` types comparison is performed on the `length` property, for the `number` type this property indicates an exact match for the `number`, ie, it may only be strictly equal to `len`.

If the `len` property is combined with the `min` and `max` range properties, `len` takes precedence.

Enumerable

To validate a value from a list of possible values use the `enum` type with a `enum` property listing the valid values for the field, for example:

```
var descriptor = {
  role: {type: "enum", enum: ['admin', 'user', 'guest']}
}
```

Whitespace

It is typical to treat required fields that only contain whitespace as errors. To add an additional test for a string that consists solely of whitespace add a `whitespace` property to a rule with a value of `true`. The rule must be a `string` type.

You may wish to sanitize user input instead of testing for whitespace, see transform for an example that would allow you to strip whitespace.

Deep Rules

If you need to validate deep object properties you may do so for validation rules that are of the `object` or `array` type by assigning nested rules to a `fields` property of the rule.

```
var descriptor = {
  address: {
    type: "object", required: true,
    fields: {
      street: {type: "string", required: true},
      city: {type: "string", required: true},
      zip: {type: "string", required: true, len: 8, message: "invalid zip"}
    }
  },
  name: {type: "string", required: true}
}
var validator = new schema(descriptor);
validator.validate({ address: {} }, (errors, fields) => {
  // errors for street, address.city, address.zip and address.name
});
```

Note that if you do not specify the `required` property on the parent rule it is perfectly valid for the field not to be declared on the source object and the deep validation rules will not be executed as there is nothing to validate against.

Deep rule validation creates a schema for the nested rules so you can also specify the `options` passed to the `schema.validate()` method.

```
var descriptor = {
  address: {
    type: "object", required: true, options: {single: true, first: true},
    fields: {
      street: {type: "string", required: true},
      city: {type: "string", required: true},
      zip: {type: "string", required: true, len: 8, message: "invalid zip"}
    }
  },
  name: {type: "string", required: true}
}
var validator = new schema(descriptor);
validator.validate({ address: {} }, (errors, fields) => {
  // now only errors for street and name
});
```

The parent rule is also validated so if you have a set of rules such as:

```
var descriptor = {
  roles: {
    type: "array", required: true, len: 3,
    fields: {
      0: {type: "string", required: true},
      1: {type: "string", required: true},
      2: {type: "string", required: true}
    }
  }
}
```

And supply a source object of `{roles: ["admin", "user"]}` then two errors will be created. One for the array length mismatch and one for the missing required array entry at index 2.

defaultField

The `defaultField` property can be used with the `array` or `object` type for validating all values of the container. It may be an `object` or `array` containing validation rules. For example:

```
var descriptor = {
  urls: {
    type: "array", required: true,
    defaultField: {type: "url"}
  }
}
```

Note that `defaultField` is expanded to `fields`, see deep rules.

Transform

Sometimes it is necessary to transform a value before validation, possibly to coerce the value or to sanitize it in some way. To do this add a `transform` function to the validation rule. The property is transformed prior to validation and re-assigned to the source object to mutate the value of the property in place.

```
var schema = require('async-validator');
var sanitize = require('validator').sanitize;
var descriptor = {
  name: {
    type: "string",
    required: true, pattern: /^[a-z]+$/,
    transform(value) {
      return sanitize(value).trim();
    }
  }
}
var validator = new schema(descriptor);
var source = {name: " user  "};
validator.validate(source, (errors, fields) => {
  assert.equal(source.name, "user");
});
```

Without the `transform` function validation would fail due to the pattern not matching as the input contains leading and trailing whitespace, but by adding the transform function validation passes and the field value is sanitized at the same time.

Messages

Depending upon your application requirements, you may need i18n support or you may prefer different validation error messages.

The easiest way to achieve this is to assign a `message` to a rule:

```
{name:{type: "string", required: true, message: "Name is required"}}
```

Message can be any type, such as jsx format.

```
{name:{type: "string", required: true, message: <b>Name is required</b>}}
```

Potentially you may require the same schema validation rules for different languages, in which case duplicating the schema rules for each language does not make sense.

In this scenario you could just provide your own messages for the language and assign it to the schema:

```
var schema = require('async-validator');
var cn = {
  required: '%s 必填',
};
var descriptor = {name:{type: "string", required: true}};
var validator = new schema(descriptor);
// deep merge with defaultMessages
validator.messages(cn);
...
```

If you are defining your own validation functions it is better practice to assign the message strings to a messages object and then access the messages via the `options.messages` property within the validation function.

validator

you can custom validate function for specified field:

```
const fields = {
  asyncField:{
    validator(rule,value,callback){
      ajax({
        url:'xx',
        value:value
      }).then(function(data){
        callback();
      },function(error){
        callback(new Error(error))
      });
    }
  },

  promiseField:{
      validator(rule, value){
        return ajax({
          url:'xx',
          value:value
        });
      }
    }
};
```

Test Case

```
npm test
npm run chrome-test
```

Coverage

```
npm run coverage
```

open coverage/ dir

License

Everything is MIT.