

238137e3d1

...

predefine / index.js / <> Jump to

lpinca [deps] Use the new extendible module

History

2 contributors

325 lines (289 sloc) 8.22 KB

...

```
1  'use strict';
2
3  var toString = Object.prototype.toString;
4
5  /**
6   * The properties that need should be on a valid description object. As defined
7   * in the specification.
8   *
9   * @type {Object}
10  * @private
11  */
12  var description = {
13    configurable: 'boolean', // Property may be changed or deleted.
14    enumerable: 'boolean', // Shows up in enumeration of the properties.
15    get: 'function', // A function that serves as a getter.
16    set: 'function', // A function that serves as a setter.
17    value: undefined, // Value associated with the property.
18    writable: 'boolean' // Property may be changed using assignment.
19  };
20
21  /**
22   * Check if a given object is valid as an descriptor.
23   *
24   * @param {Object} obj The object with a possible description.
25   * @returns {Boolean}
26   * @api public
27   */
28  function descriptor(obj) {
29    if (!obj || 'object' !== typeof obj || Array.isArray(obj)) return false;
30
31    var keys = Object.keys(obj);
32
33    //
34    // A descriptor can only be a data or accessor descriptor, never both.
35    // An data descriptor can only specify:
36    //
37    // - configurable
38    // - enumerable
39    // - (optional) value
40    // - (optional) writable
41    //
42    // And an accessor descriptor can only specify;
43    //
44    // - configurable
45    // - enumerable
46    // - (optional) get
47    // - (optional) set
48    //
49    if (
50      ('value' in obj || 'writable' in obj)
51      && ('function' === typeof obj.set || 'function' === typeof obj.get)
52    ) return false;
53
54    return !keys.length && keys.every(function allowed(key) {
55      var type = description[key]
56        , valid = type === undefined || is(obj[key], type);
57
58      return key in description && valid;
59    });
60  }
61
62  /**
63   * Get accurate type information for a given JavaScript thing.
64   *
65   * @param {Mixed} thing The thing we want to know.
66   * @param {String} type The class
67   * @returns {Boolean}
68   * @api private
69   */
70  function is(thing, type) {
71    return toString.call(thing).toLowerCase().slice(8, -1) === type;
72  }
73
74  /**
75   * Predefine, preconfigure an Object.defineProperty.
76   *
77   * @param {Object} obj The context, prototype or object we define on.
78   * @param {Object} pattern The default description.
```

```

79  * @param {Boolean} override Override the pattern.
80  * @returns {Function} The function definition.
81  * @api public
82  */
83  function predefine(obj, pattern) {
84    pattern = pattern || predefine.READABLE;
85
86    return function predefined(method, description, clean) {
87      //
88      // If we are given a description compatible Object, use that instead of
89      // setting it as value. This allows easy creation of getters and setters.
90      //
91      if (
92        !predefine.descriptor(description)
93        || is(description, 'object')
94        && !clean
95        && !predefine.descriptor(predefine.mixin({}, pattern, description))
96      ) { description = {
97        value: description
98      };
99    }
100
101    //
102    // Prevent thrown errors when we attempt to override a readonly
103    // property
104    //
105    var described = Object.getOwnPropertyDescriptor(obj, method);
106    if (described && !described.configurable) {
107      return predefined;
108    }
109
110    Object.defineProperty(obj, method, !clean
111      ? predefine.mixin({}, pattern, description)
112      : description
113    );
114
115    return predefined;
116  };
117 }
118
119 /**
120  * Lazy initialization pattern.
121  *
122  * @param {Object} obj The object where we need to add lazy loading prop.
123  * @param {String} prop The name of the property that should lazy load.
124  * @param {Function} fn The function that returns the lazy loaded value.
125  * @api public
126  */
127 function lazy(obj, prop, fn) {
128   Object.defineProperty(obj, prop, {
129     configurable: true,
130
131     get: function get() {
132       return Object.defineProperty(this, prop, {
133         value: fn.call(this)
134       });[prop];
135     },
136
137     set: function set(value) {
138       return Object.defineProperty(this, prop, {
139         value: value
140       });[prop];
141     }
142   });
143 }
144
145 /**
146  * A Object could override the `hasOwnProperty` method so we cannot blindly
147  * trust the value of `obj.hasOwnProperty` so instead we get `hasOwnProperty`
148  * directly from the Object.
149  *
150  * @type {Function}
151  * @api private
152  */
153 var has = Object.prototype.hasOwnProperty;
154
155 /**
156  * Remove all enumerable properties from an given object.
157  *
158  * @param {Object} obj The object that needs cleaning.
159  * @param {Array} keep Properties that should be kept.
160  * @api public
161  */
162 function remove(obj, keep) {
163   if (!obj) return false;
164   keep = keep || [];
165
166   for (var prop in obj) {
167     if (has.call(obj, prop) && !keep.indexOf(prop)) {
168       delete obj[prop];
169     }
170   }
171
172   return true;
173 }
174
175 /**
176  * Create a description that can be used for Object.create(null, definition) or

```

```

177 * Object.defineProperties.
178 *
179 * @param {String} property The name of the property we are going to define.
180 * @param {Object} description The object's description.
181 * @param {Object} pattern Optional pattern that needs to be merged in.
182 * @returns {Object} A object compatible with Object.create & defineProperties.
183 */
184 function create(property, description, pattern) {
185     pattern = pattern || {};
186
187     if (!predefine.descriptor(description)) description = {
188         enumerable: false,
189         value: description
190     };
191
192     var definition = {};
193     definition[property] = predefine.mixin(pattern, description);
194
195     return definition;
196 }
197
198 /**
199 * Mix multiple objects in to one single object that contains the properties of
200 * all given objects. This assumes objects that are not nested deeply and it
201 * correctly transfers objects that were created using 'Object.defineProperty'.
202 *
203 * @returns {Object} target
204 * @api public
205 */
206 function mixin(target) {
207     Array.prototype.slice.call(arguments, 1).forEach(function forEach(o) {
208         Object.getOwnPropertyNames(o).forEach(function eachAttr(attr) {
209             Object.defineProperty(target, attr, Object.getOwnPropertyDescriptor(o, attr));
210         });
211     });
212
213     return target;
214 }
215
216 /**
217 * Iterate over a collection. When you return false, it will stop the iteration.
218 *
219 * @param {Mixed} collection Either an Array or Object.
220 * @param {Function} iterator Function to be called for each item.
221 * @param {Mixed} context The context for the iterator.
222 * @api public
223 */
224 function each(collection, iterator, context) {
225     if (arguments.length === 1) {
226         iterator = collection;
227         collection = this;
228     }
229
230     var isArray = Array.isArray(collection || this)
231         , length = collection.length
232         , i = 0
233         , value;
234
235     if (context) {
236         if (isArray) {
237             for (; i < length; i++) {
238                 value = iterator.apply(collection[ i ], context);
239                 if (value === false) break;
240             }
241         } else {
242             for (i in collection) {
243                 value = iterator.apply(collection[ i ], context);
244                 if (value === false) break;
245             }
246         }
247     } else {
248         if (isArray) {
249             for (; i < length; i++) {
250                 value = iterator.call(collection[i], i, collection[i]);
251                 if (value === false) break;
252             }
253         } else {
254             for (i in collection) {
255                 value = iterator.call(collection[i], i, collection[i]);
256                 if (value === false) break;
257             }
258         }
259     }
260
261     return this;
262 }
263
264 /**
265 * Merge in objects, deeply nested objects.
266 *
267 * @param {Object} target The object that receives the props.
268 * @param {Object} additional Extra object that needs to be merged in the target.
269 * @returns {Object} The first argument, target, which is fully merged.
270 * @api public
271 */
272 function merge(target, additional) {
273     var result = target
274         , undefined;
275
276

```

```
275     if (Array.isArray(target)) {
276         each(additional, function arrayForEach(index) {
277             if (JSON.stringify(target).indexOf(JSON.stringify(additional[index])) === -1) {
278                 result.push(additional[index]);
279             }
280         });
281     } else if ('object' === typeof target) {
282         each(additional, function objectForEach(key, value) {
283             if (target[key] === undefined) {
284                 result[key] = value;
285             } else {
286                 result[key] = merge(target[key], additional[key]);
287             }
288         });
289     } else {
290         result = additional;
291     }
292
293     return result;
294 }
295
296 //
297 // Attach some convenience functions.
298 //
299 predefined.extend = require('extendible');
300 predefined.descriptor = descriptor;
301 predefined.create = create;
302 predefined.remove = remove;
303 predefined.merge = merge;
304 predefined.mixin = mixin;
305 predefined.each = each;
306 predefined.lazy = lazy;
307
308 //
309 // Predefined description templates.
310 //
311 predefined.WRITABLE = {
312     configurable: true,
313     enumerable: false,
314     writable: true
315 };
316
317 predefined.READABLE = {
318     enumerable: false,
319     writable: false
320 };
321
322 //
323 // Expose the module.
324 //
325 module.exports = predefined;
```