325 lines (289 sloc) | 8.22 KB

```
'use strict';

var toString = Object.prototype.toString;

/**
 * The properties that need should be on a valid description object. As defined
 * in the specification.
 *
 * @type {Object}
 * @private
 */
var description = {
  configurable: 'boolean',  // Property may be changed or deleted.
  enumerable: 'boolean',    // Shows up in enumeration of the properties.
  get: 'function',          // A function that serves as a getter.
  set: 'function',          // A function that serves as a setter.
  value: undefined,         // Value associated with the property.
  writable: 'boolean'       // Property may be changed using assignment.
};

/**
 * Check if a given object is valid as an descriptor.
 *
 * @param {Object} obj The object with a possible description.
 * @returns {Boolean}
 * @api public
 */
function descriptor(obj) {
  if (!obj || 'object' !== typeof obj || Array.isArray(obj)) return false;

  var keys = Object.keys(obj);

  //
  // A descriptor can only be a data or accessor descriptor, never both.
  // An data descriptor can only specify:
  //
  // - configurable
  // - enumerable
  // - (optional) value
  // - (optional) writable
  //
  // And an accessor descriptor can only specify;
  //
  // - configurable
  // - enumerable
  // - (optional) get
  // - (optional) set
  //
  if (
       ('value' in obj || 'writable' in obj)
    && ('function' === typeof obj.set || 'function' === typeof obj.get)
  ) return false;

  return !!keys.length && keys.every(function allowed(key) {
    var type = description[key]
      , valid = type === undefined || is(obj[key], type);

    return key in description && valid;
  });
}

/**
 * Get accurate type information for a given JavaScript thing.
 *
 * @param {Mixed} thing The thing we want to know.
 * @param {String} type The class
 * @returns {Boolean}
 * @api private
 */
function is(thing, type) {
  return toString.call(thing).toLowerCase().slice(8, -1) === type;
}

/**
 * Predefine, preconfigure an Object.defineProperty.
 *
 * @param {Object} obj The context, prototype or object we define on.
 * @param {Object} pattern The default description.
```

```
 * @param {Boolean} override Override the pattern.
 * @returns {Function} The function definition.
 * @api public
 */
function predefine(obj, pattern) {
  pattern = pattern || predefine.READABLE;

  return function predefined(method, description, clean) {
    //
    // If we are given a description compatible Object, use that instead of
    // setting it as value. This allows easy creation of getters and setters.
    //
    if (
         !predefine.descriptor(description)
      || is(description, 'object')
         && !clean
         && !predefine.descriptor(predefine.mixin({}, pattern, description))
    ) { description = {
        value: description
      };
    }

    //
    // Prevent thrown errors when we attempt to override a readonly
    // property
    //
    var described = Object.getOwnPropertyDescriptor(obj, method);
    if (described && !described.configurable) {
      return predefined;
    }

    Object.defineProperty(obj, method, !clean
      ? predefine.mixin({}, pattern, description)
      : description
    );

    return predefined;
  };
}

/**
 * Lazy initialization pattern.
 *
 * @param {Object} obj The object where we need to add lazy loading prop.
 * @param {String} prop The name of the property that should lazy load.
 * @param {Function} fn The function that returns the lazy laoded value.
 * @api public
 */
function lazy(obj, prop, fn) {
  Object.defineProperty(obj, prop, {
    configurable: true,

    get: function get() {
      return Object.defineProperty(this, prop, {
        value: fn.call(this)
      })[prop];
    },

    set: function set(value) {
      return Object.defineProperty(this, prop, {
        value: value
      })[prop];
    }
  });
}

/**
 * A Object could override the `hasOwnProperty` method so we cannot blindly
 * trust the value of `obj.hasOwnProperty` so instead we get `hasOwnProperty`
 * directly from the Object.
 *
 * @type {Function}
 * @api private
 */
var has = Object.prototype.hasOwnProperty;

/**
 * Remove all enumerable properties from an given object.
 *
 * @param {Object} obj The object that needs cleaning.
 * @param {Array} keep Properties that should be kept.
 * @api public
 */
function remove(obj, keep) {
  if (!obj) return false;
  keep = keep || [];

  for (var prop in obj) {
    if (has.call(obj, prop) && !~keep.indexOf(prop)) {
      delete obj[prop];
    }
  }

  return true;
}

/**
 * Create a description that can be used for Object.create(null, definition) or
```

```
 * Object.defineProperties.
 *
 * @param {String} property The name of the property we are going to define.
 * @param {Object} description The object's description.
 * @param {Object} pattern Optional pattern that needs to be merged in.
 * @returns {Object} A object compatible with Object.create & defineProperties.
 */
function create(property, description, pattern) {
  pattern = pattern || {};

  if (!predefine.descriptor(description)) description = {
    enumberable: false,
    value: description
  };

  var definition = {};
  definition[property] = predefine.mixin(pattern, description);

  return definition;
}

/**
 * Mix multiple objects in to one single object that contains the properties of
 * all given objects. This assumes objects that are not nested deeply and it
 * correctly transfers objects that were created using `Object.defineProperty`.
 *
 * @returns {Object} target
 * @api public
 */
function mixin(target) {
  Array.prototype.slice.call(arguments, 1).forEach(function forEach(o) {
    Object.getOwnPropertyNames(o).forEach(function eachAttr(attr) {
      Object.defineProperty(target, attr, Object.getOwnPropertyDescriptor(o, attr));
    });
  });

  return target;
}
/**
 * Iterate over a collection. When you return false, it will stop the iteration.
 *
 * @param {Mixed} collection Either an Array or Object.
 * @param {Function} iterator Function to be called for each item.
 * @param {Mixed} context The context for the iterator.
 * @api public
 */
function each(collection, iterator, context) {
  if (arguments.length === 1) {
    iterator = collection;
    collection = this;
  }

  var isArray = Array.isArray(collection || this)
    , length = collection.length
    , i = 0
    , value;

  if (context) {
    if (isArray) {
      for (; i < length; i++) {
        value = iterator.apply(collection[ i ], context);
        if (value === false) break;
      }
    } else {
      for (i in collection) {
        value = iterator.apply(collection[ i ], context);
        if (value === false) break;
      }
    }
  } else {
    if (isArray) {
      for (; i < length; i++) {
        value = iterator.call(collection[i], i, collection[i]);
        if (value === false) break;
      }
    } else {
      for (i in collection) {
        value = iterator.call(collection[i], i, collection[i]);
        if (value === false) break;
      }
    }
  }

  return this;
}

/**
 * Merge in objects, deeply nested objects.
 *
 * @param {Object} target The object that receives the props.
 * @param {Object} additional Extra object that needs to be merged in the target.
 * @returns {Object} The first argument, target, which is fully merged.
 * @api public
 */
function merge(target, additional) {
  var result = target
    , undefined;

```

```
  if (Array.isArray(target)) {
    each(additional, function arrayForEach(index) {
      if (JSON.stringify(target).indexOf(JSON.stringify(additional[index])) === -1) {
        result.push(additional[index]);
      }
    });
  } else if ('object' === typeof target) {
    each(additional, function objectForEach(key, value) {
      if (target[key] === undefined) {
        result[key] = value;
      } else {
        result[key] = merge(target[key], additional[key]);
      }
    });
  } else {
    result = additional;
  }

  return result;
}

//
// Attach some convenience functions.
//
predefine.extend = require('extendible');
predefine.descriptor = descriptor;
predefine.create = create;
predefine.remove = remove;
predefine.merge = merge;
predefine.mixin = mixin;
predefine.each = each;
predefine.lazy = lazy;

//
// Predefined description templates.
//
predefine.WRITABLE = {
  configurable: true,
  enumerable: false,
  writable: true
};

predefine.READABLE = {
  enumerable: false,
  writable: false
};

//
// Expose the module.
//
module.exports = predefine;
```