util 是一个Node.js 核心模块，提供常用函数的集合，用于弥补核心 JavaScript 的功能 过于精简的不足。

使用方法如下：

const util = require('util');

**util.callbackify**

util.callbackify(original) 将 async 异步函数（或者一个返回值为 Promise 的函数）转换成遵循异常优先的回调风格的函数，例如将 (err, value) => ... 回调作为最后一个参数。 在回调函数中，第一个参数为拒绝的原因（如果 Promise 解决，则为 null），第二个参数则是解决的值。

**实例**

const util = require('util');

async function fn() {

return 'hello world';

}

const callbackFunction = util.callbackify(fn);

callbackFunction((err, ret) => {

if (err) throw err;

console.log(ret);

});

以上代码输出结果为：

hello world

回调函数是异步执行的，并且有异常堆栈错误追踪。 如果回调函数抛出一个异常，进程会触发一个 'uncaughtException' 异常，如果没有被捕获，进程将会退出。

null 在回调函数中作为一个参数有其特殊的意义，如果回调函数的首个参数为 Promise 拒绝的原因且带有返回值，且值可以转换成布尔值 false，这个值会被封装在 Error 对象里，可以通过属性 reason 获取。

function fn() {

return Promise.reject(null);

}

const callbackFunction = util.callbackify(fn);

callbackFunction((err, ret) => {

// 当 Promise 被以 `null` 拒绝时，它被包装为 Error 并且原始值存储在 `reason` 中。

err && err.hasOwnProperty('reason') && err.reason === null; // true

});

original 为 async 异步函数。该函数返回传统回调函数。

**util.inherits**

util.inherits(constructor, superConstructor) 是一个实现对象间原型继承的函数。

JavaScript 的面向对象特性是基于原型的，与常见的基于类的不同。JavaScript 没有提供对象继承的语言级别特性，而是通过原型复制来实现的。

在这里我们只介绍 util.inherits 的用法，示例如下：

var util = require('util');

function Base() {

    this.name = 'base';

    this.base = 1991;

    this.sayHello = function() {

    console.log('Hello ' + this.name);

    };

}

Base.prototype.showName = function() {

    console.log(this.name);

};

function Sub() {

    this.name = 'sub';

}

util.inherits(Sub, Base);

var objBase = new Base();

objBase.showName();

objBase.sayHello();

console.log(objBase);

var objSub = new Sub();

objSub.showName();

//objSub.sayHello();

console.log(objSub);

我们定义了一个基础对象 Base 和一个继承自 Base 的 Sub，Base 有三个在构造函数内定义的属性和一个原型中定义的函数，通过util.inherits 实现继承。运行结果如下：

base

Hello base

{ name: 'base', base: 1991, sayHello: [Function] }

sub

{ name: 'sub' }

注意：Sub 仅仅继承了Base 在原型中定义的函数，而构造函数内部创造的 base 属 性和 sayHello 函数都没有被 Sub 继承。

同时，在原型中定义的属性不会被 console.log 作 为对象的属性输出。如果我们去掉 objSub.sayHello(); 这行的注释，将会看到：

node.js:201

throw e; // process.nextTick error, or 'error' event on first tick

^

TypeError: Object #&lt;Sub&gt; has no method 'sayHello'

at Object.&lt;anonymous&gt; (/home/byvoid/utilinherits.js:29:8)

at Module.\_compile (module.js:441:26)

at Object..js (module.js:459:10)

at Module.load (module.js:348:31)

at Function.\_load (module.js:308:12)

at Array.0 (module.js:479:10)

at EventEmitter.\_tickCallback (node.js:192:40)

**util.inspect**

util.inspect(object,[showHidden],[depth],[colors]) 是一个将任意对象转换 为字符串的方法，通常用于调试和错误输出。它至少接受一个参数 object，即要转换的对象。

showHidden 是一个可选参数，如果值为 true，将会输出更多隐藏信息。

depth 表示最大递归的层数，如果对象很复杂，你可以指定层数以控制输出信息的多 少。如果不指定depth，默认会递归 2 层，指定为 null 表示将不限递归层数完整遍历对象。 如果 colors 值为 true，输出格式将会以 ANSI 颜色编码，通常用于在终端显示更漂亮 的效果。

特别要指出的是，util.inspect 并不会简单地直接把对象转换为字符串，即使该对 象定义了 toString 方法也不会调用。

var util = require('util');

function Person() {

    this.name = 'byvoid';

    this.toString = function() {

    return this.name;

    };

}

var obj = new Person();

console.log(util.inspect(obj));

console.log(util.inspect(obj, true));

运行结果是：

Person { name: 'byvoid', toString: [Function] }

Person {

name: 'byvoid',

toString:

{ [Function]

[length]: 0,

[name]: '',

[arguments]: null,

[caller]: null,

[prototype]: { [constructor]: [Circular] } } }

**util.isArray(object)**

如果给定的参数 "object" 是一个数组返回 true，否则返回 false。

var util = require('util');

util.isArray([])

// true

util.isArray(new Array)

// true

util.isArray({})

// false

**util.isRegExp(object)**

如果给定的参数 "object" 是一个正则表达式返回true，否则返回false。

var util = require('util');

util.isRegExp(/some regexp/)

// true

util.isRegExp(new RegExp('another regexp'))

// true

util.isRegExp({})

// false

**util.isDate(object)**

如果给定的参数 "object" 是一个日期返回true，否则返回false。

var util = require('util');

util.isDate(new Date())

// true

util.isDate(Date())

// false (without 'new' returns a String)

util.isDate({})

// false

更多详情可以访问 [http://nodejs.org/api/util.html](https://nodejs.org/api/util.html) 了解详细内容。

## util.debuglog(section)

* section{字符串} 被调试的程序节点部分
* Returns: {Function} 日志函数

用来创建一个有条件的写到stderr的函数（基于NODE\_DEBUG环境变量）。如果section出现在环境变量里，返回函数将会和console.error()类似。否则，返回一个空函数。

例如：

javascript

var debuglog = util.debuglog('foo');

var bar = 123;

debuglog('hello from foo [%d]', bar);

如果这个程序以NODE\_DEBUG=foo的环境运行，将会输出：

FOO 3245: hello from foo [123]

3245是进程ID。如果没有运行在这个环境变量里，将不会打印任何东西。

可以用逗号切割多个NODE\_DEBUG环境变量。例如：NODE\_DEBUG=fs,net,tls。

## util.format(format[, ...])

使用第一个参数返回一个格式化的字符串，类似printf。

第一个参数是字符串，它包含0或更多的占位符。每个占位符被替换成想要参数转换的值。支持的占位符包括：

* %s- 字符串.
* %d- 数字 (整数和浮点数).
* %j- JSON。如果参数包含循环引用，将会用字符串替换R
* %%- 单独一个百分号 ('%')。 不会消耗一个参数。

如果占位符没有包含一个相应的参数，占位符不会被替换。

util.format('%s:%s', 'foo'); // 'foo:%s'

如果参数超过占位符，多余的参数将会用util.inspect()转换成字符串，并拼接在一起，用空格隔开。

util.format('%s:%s', 'foo', 'bar', 'baz'); // 'foo:bar baz'

如果第一个参数不是格式化字符串，那么util.format()会返回所有参数拼接成的字符串（空格分割）。每个参数都会用util.inspect()转换成字符串。

util.format(1, 2, 3); // '1 2 3'

## util.log(string)

在stdout输出并带有时间戳：

require('util').log('Timestamped message.');

## util.inspect(object[, options])

返回一个对象的字符串表现形式，在代码调试的时候非常有用。

通过加入一些可选选项，来改变对象的格式化输出形式：

* showHidden- 如果为true，将会显示对象的不可枚举属性。默认为false。
* depth- 告诉inspect格式化对象时递归多少次。这在格式化大且复杂对象时非常有用。默认为 2。如果想无穷递归的话，传null。
* colors- 如果为true，输出内容将会格式化为有颜色的代码。默认为false， 颜色可以自定义，参见下文。
* customInspect- 如果为false，那么定义在被检查对象上的inspect(depth, opts) 方法将不会被调用。 默认为true。

检查util对象上所有属性的例子：

var util = require('util');

console.log(util.inspect(util, { showHidden: true, depth: null }));

当被调用的时候，参数值可以提供自己的自定义inspect(depth, opts)方法。该方法会接收当前的递归检查深度，以及传入util.inspect()的其他参数。

### 自定义util.inspect颜色

util.inspect通过util.inspect.styles和util.inspect.colors对象，自定义全局的输出颜色，

util.inspect.styles和util.inspect.colors组成风格颜色的一对映射。

高亮风格和他们的默认值：

* 数字 (黄色)
* boolean (黄色)
* 字符串 (绿色)
* date (洋红)
* regexp (红色)
* null (粗体)
* undefined (斜体)
* special (青绿色)
* name (内部用，不是风格)

预定义的颜色为: white、斜体、black、blue、cyan、绿色、洋红、红色和黄色以及粗体、斜体、下划线和反选风格。

### 对象自定义inspect()函数

对象也能自定义inspect(depth)函数， 当使用util.inspect()检查该对象的时候，将会执行对象自定义的检查方法：

var util = require('util');

var obj = { name: 'nate' };

obj.inspect = function(depth) {

return '{' + this.name + '}';

};

util.inspect(obj);

// "{nate}"

你可以返回另外一个对象，返回的字符串会根据返回的对象格式化。这和JSON.stringify()的工作流程类似。您还可以完全返回另一个对象，返回的字符串将根据返回的对象格式化。这与JSON.stringify()的工作方式类似：

var obj = { foo: 'this will not show up in the inspect() output' };

obj.inspect = function(depth) {

return { bar: 'baz' };

};

util.inspect(obj);

// "{ bar: 'baz' }"