开发框架—基于Nodejs的express

基于 Node.js 平台，快速、开放、极简的 web 开发框架。
创建一个 Express 应用。express() 是一个由 express 模块导出的入口（top-level）函数。

```
var express = require('express');
var app = express();
```

内置方法
express.static(root, [options])
express.static 是 Express 内置的唯一一个中间件。是基于 serve-static 开发的，负责托管 Express 应用内的静态资源。

root 参数指的是静态资源文件所在的根目录。

options 对象是可选的，支持以下属性：

属性	描述	类型	默认值
dotfiles	Option for serving dotfiles. Possible values are "allow", "deny", and "ignore"	String	"ignore"
etag	Enable or disable etag generation	Boolean	true
extensions	Sets file extension fallbacks.	Boolean	false
index	Sends directory index file. Set false to disable directory indexing.	Mixed	"index.html"
lastModified	Set the Last-Modified header to the last modified date of the file on the OS. Possible values are true or false.	Boolean	true
maxAge	Set the max-age property of the Cache-Control header in milliseconds or a string in ms format	Number	0
redirect	Redirect to trailing "/" when the pathname is a directory.	Boolean	true
setHeaders	Function for setting HTTP headers to serve with the file.	Function	

Application

The app object conventionally denotes the Express application. Create it by calling the top-level express() function exported by the Express module:

```
var express = require('express');
var app = express();

app.get('/', function(req, res){
  res.send('hello world');
});

app.listen(3000);
```

The app object has methods for

Routing HTTP requests; see for example, app.METHOD and app.param.
Configuring middleware; see app.route.
Rendering HTML views; see app.render.
Registering a template engine; see app.engine.
It also has settings (properties) that affect how the application behaves; for more information, see Application settings.

Properties
app.locals
The app.locals object is a JavaScript object, and its properties are local variables within the application.

```
app.locals.title
// => 'My App'

app.locals.email
// => 'me@myapp.com'
```

Once set, the value of app.locals properties persist throughout the life of the application, in contrast with res.locals properties that are valid only for the lifetime of the request.

You can access local variables in templates rendered within the application. This is useful for providing helper functions to templates, as well as app-level data. Note, however, that you cannot access local variables in middleware.

```
app.locals.title = 'My App';
app.locals.strftime = require('strftime');
app.locals.email = 'me@myapp.com';
```

app.mountpath
The app.mountpath property is the path pattern(s) on which a sub app was mounted.

A sub app is an instance of express which may be used for handling the request to a route.

```
var express = require('express');

var app = express(); // the main app
var admin = express(); // the sub app

admin.get('/', function (req, res) {
  console.log(admin.mountpath); // /admin
  res.send('Admin Homepage');
})

app.use('/admin', admin); // mount the sub app
```

It is similar to the baseUrl property of the req object, except req.baseUrl returns the matched URL path, instead of the matched pattern(s).

If a sub-app is mounted on multiple path patterns, app.mountpath returns the list of patterns it is mounted on, as shown in the following example.

```
var admin = express();

admin.get('/', function (req, res) {
  console.log(admin.mountpath); // [ '/adm*n', '/manager' ]
  res.send('Admin Homepage');
})

var secret = express();
secret.get('/', function (req, res) {
  console.log(secret.mountpath); // /secr*t
  res.send('Admin Secret');
});

admin.use('/secr*t', secret); // load the 'secret' router on '/secr*t', on the 'admin' sub app
```

app.use(['/adm*n', '/manager'], admin); // load the 'admin' router on '/adm*n' and '/manager', on the parent app
Events
app.on('mount', callback(parent))
The mount event is fired on a sub-app, when it is mounted on a parent app. The parent app is passed to the callback function.

```
var admin = express();

admin.on('mount', function (parent) {
  console.log('Admin Mounted');
  console.log(parent); // refers to the parent app
});

admin.get('/', function (req, res) {
  res.send('Admin Homepage');
});

app.use('/admin', admin);
```

Methods
app.all(path, callback [, callback ...])
This method is like the standard app.METHOD() methods, except it matches all HTTP verbs.

It's useful for mapping "global" logic for specific path prefixes or arbitrary matches. For example, if you put the following at the top of all other route definitions, it requires that all routes from that point on require authentication, and automatically load a user. Keep in mind that these callbacks do not have to act as end-points: loadUser can perform a task, then call next() to continue matching subsequent routes.

```
app.all('*', requireAuthentication, loadUser);
```

Or the equivalent:

```
app.all('*', requireAuthentication)
app.all('*', loadUser);
```

Another example is white-listed "global" functionality. The example is much like before, however it only restricts paths that start with "/api":

```
app.all('/api/*', requireAuthentication);
```

app.delete(path, callback [, callback ...])
Routes HTTP DELETE requests to the specified path with the specified callback functions. For more information, see the routing guide.

You can provide multiple callback functions that behave just like middleware, except these callbacks can invoke next('route') to bypass the remaining route callback(s). You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there's no reason to proceed with the current route.

```
app.delete('/', function (req, res) {
  res.send('DELETE request to homepage');
});
```

app.disable(name)
Sets the Boolean setting name to false, where name is one of the properties from the app settings table. Calling app.set('foo', false) for a Boolean property is the same as calling app.disable('foo').

For example:

```
app.disable('trust proxy');
app.get('trust proxy');
// => false
```
app.disabled(name)
Returns true if the Boolean setting name is disabled (false), where name is one of the properties from the app settings table.

```
app.disabled('trust proxy');
// => true

app.enable('trust proxy');
app.disabled('trust proxy');
// => false
```
app.enable(name)
Sets the Boolean setting name to true, where name is one of the properties from the app settings table. Calling app.set('foo', true) for a Boolean property is the same as calling app.enable('foo').

```
app.enable('trust proxy');
app.get('trust proxy');
// => true
```
app.enabled(name)
Returns true if the setting name is enabled (true), where name is one of the properties from the app settings table.

```
app.enabled('trust proxy');
// => false

app.enable('trust proxy');
app.enabled('trust proxy');
// => true
```
app.engine(ext, callback)
Registers the given template engine callback as ext.

By default, Express will require() the engine based on the file extension. For example, if you try to render a “foo.jade” file, Express invokes the following internally, and caches the require() on subsequent calls to increase performance.

```
app.engine('jade', require('jade').__express);
```
Use this method for engines that do not provide .__express out of the box, or if you wish to “map” a different extension to the template engine.

For example, to map the EJS template engine to “.html” files:

```
app.engine('html', require('ejs').renderFile);
```
In this case, EJS provides a .renderFile() method with the same signature that Express expects: (path, options, callback), though note that it aliases this method as ejs.__express internally so if you’re using “.ejs” extensions you don’t need to do anything.

Some template engines do not follow this convention. The consolidate.js library maps Node template engines to follow this convention, so they work seemlessly with Express.

```
var engines = require('consolidate');
app.engine('haml', engines.haml);
app.engine('html', engines.hogan);
```
app.get(name)

Returns the value of name app setting, where name is one of strings in the app settings table. For example:

```
app.get('title');
// => undefined

app.set('title', 'My Site');
app.get('title');
// => "My Site"
```

app.get(path, callback [, callback ...])
Routes HTTP GET requests to the specified path with the specified callback functions. For more information, see the routing guide.

You can provide multiple callback functions that behave just like middleware, except these callbacks can invoke next('route') to bypass the remaining route callback(s). You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there's no reason to proceed with the current route.

```
app.get('/', function (req, res) {
  res.send('GET request to homepage');
});
```

app.listen(port, [hostname], [backlog], [callback])
Binds and listens for connections on the specified host and port. This method is identical to Node's http.Server.listen().

```
var express = require('express');
var app = express();
app.listen(3000);
```

The app returned by express() is in fact a JavaScript Function, designed to be passed to Node's HTTP servers as a callback to handle requests. This makes it easy to provide both HTTP and HTTPS versions of your app with the same code base, as the app does not inherit from these (it is simply a callback):

```
var express = require('express');
var https = require('https');
var http = require('http');
var app = express();

http.createServer(app).listen(80);
https.createServer(options, app).listen(443);
```

The app.listen() method is a convenience method for the following (for HTTP only):

```
app.listen = function() {
  var server = http.createServer(this);
  return server.listen.apply(server, arguments);
};
```

app.METHOD(path, callback [, callback ...])
Routes an HTTP request, where METHOD is the HTTP method of the request, such as GET, PUT, POST, and so on, in lowercase. Thus, the actual methods are app.get(), app.post(), app.put(), and so on. See below for the complete list.

For more information, see the routing guide.

Express supports the following routing methods corresponding to the HTTP methods of the same names:

checkout
connect
copy
delete
get
head
lock
merge
mkactivity
mkcol
move
m-search
notify
options
patch
post
propfind
proppatch
purge
put
report
search
subscribe
trace
unlock
unsubscribe
To route methods which translate to invalid JavaScript variable names, use the bracket notation. For example, app['m-search']('/', function ....
You can provide multiple callback functions that behave just like middleware, except that these callbacks can invoke next('route') to bypass the remaining route callback(s). You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there is no reason to proceed with the current route.

The API documentation has explicit entries only for the most popular HTTP methods app.get(), app.post(), app.put(), and app.delete(). However, the other methods listed above work in exactly the same way.
There is a special routing method, app.all(), that is not derived from any HTTP method. It loads middleware at a path for all request methods.

In the following example, the handler is executed for requests to “/secret” whether using GET, POST, PUT, DELETE, or any other HTTP request method.

```
app.all('/secret', function (req, res, next) {
  console.log('Accessing the secret section ...')
  next() // pass control to the next handler
})
```

app.param([name], callback)
Add callback triggers to route parameters, where name is the name of the parameter or an array of them, and function is the callback function. The parameters of the callback function are the request object, the response object, the next middleware, and the value of the parameter, in that order.

If name is an array, the callback trigger is registered for each parameter declared in it, in the order in which they are declared. Furthermore, for each declared parameter except the last one, a call to next inside the callback will call the callback for the next declared parameter. For the last parameter, a call to next will call the next middleware in place for the route currently being processed, just like it would if name were just a string.

For example, when :user is present in a route path, you may map user loading logic to automatically provide req.user to the route, or perform validations on the parameter input.

```
app.param('user', function(req, res, next, id) {

  // try to get the user details from the User model and attach it to the request object
  User.find(id, function(err, user) {
    if (err) {
      next(err);
    } else if (user) {
      req.user = user;
      next();
    } else {
      next(new Error('failed to load user'));
    }
  });
});
```

Param callback functions are local to the router on which they are defined. They are not inherited by mounted apps or routers. Hence, param callbacks defined on app will be triggered only by route parameters defined on app routes.

All param callbacks will be called before any handler of any route in which the param occurs, and they will each be called only once in a request-response cycle, even if the parameter is matched in multiple routes, as shown in the following examples.

```
app.param('id', function (req, res, next, id) {
  console.log('CALLED ONLY ONCE');
  next();
})

app.get('/user/:id', function (req, res, next) {
  console.log('although this matches');
  next();
});

app.get('/user/:id', function (req, res) {
  console.log('and this matches too');
  res.end();
});
```

On GET /user/42, the following is printed:

```
CALLED ONLY ONCE
although this matches
and this matches too
app.param(['id', 'page'], function (req, res, next, value) {
  console.log('CALLED ONLY ONCE with', value);
  next();
})

app.get('/user/:id/:page', function (req, res, next) {
  console.log('although this matches');
  next();
});

app.get('/user/:id/:page', function (req, res) {
```

```
  console.log('and this matches too');
  res.end();
});
```
On GET /user/42/3, the following is printed:

```
CALLED ONLY ONCE with 42
CALLED ONLY ONCE with 3
although this matches
and this matches too
```
The following section describes app.param(callback), which is deprecated as of v4.11.0.
The behavior of the app.param(name, callback) method can be altered entirely by passing only a function to app.param(). This function is a custom implementation of how app.param(name, callback) should behave - it accepts two parameters and must return a middleware.

The first parameter of this function is the name of the URL parameter that should be captured, the second parameter can be any JavaScript object which might be used for returning the middleware implementation.

The middleware returned by the function decides the behavior of what happens when a URL parameter is captured.

In this example, the app.param(name, callback) signature is modified to app.param(name, accessId). Instead of accepting a name and a callback, app.param() will now accept a name and a number.

```
var express = require('express');
var app = express();

// customizing the behavior of app.param()
app.param(function(param, option) {
  return function (req, res, next, val) {
    if (val == option) {
      next();
    }
    else {
      res.sendStatus(403);
    }
  }
});

// using the customized app.param()
app.param('id', 1337);

// route to trigger the capture
app.get('/user/:id', function (req, res) {
  res.send('OK');
})

app.listen(3000, function () {
  console.log('Ready');
})
```
In this example, the app.param(name, callback) signature remains the same, but instead of a middleware callback, a custom data type checking function has been defined to validate the data type of the user id.

```
app.param(function(param, validator) {
```

```
  return function (req, res, next, val) {
    if (validator(val)) {
      next();
    }
    else {
      res.sendStatus(403);
    }
  }
})

app.param('id', function (candidate) {
  return !isNaN(parseFloat(candidate)) && isFinite(candidate);
});
```

The '.' character can't be used to capture a character in your capturing regexp. For example you can't use '/user-.+/' to capture 'users-gami', use [\\s\\S] or [\\w\\W] instead (as in '/user-[\\s\\S]+/'.

Examples:

```
//captures '1-a_6' but not '543-azser-sder'
router.get('/[0-9]+-[[\\w]]*', function);

//captures '1-a_6' and '543-az(ser"-sder' but not '5-a s'
router.get('/[0-9]+-[[\\S]]*', function);

//captures all (equivalent to '.*')
router.get('[[\\s\\S]]*', function);
```

app.path()

Returns the canonical path of the app, a string.

```
var app = express()
  , blog = express()
  , blogAdmin = express();

app.use('/blog', blog);
blog.use('/admin', blogAdmin);

console.log(app.path()); // ''
console.log(blog.path()); // '/blog'
console.log(blogAdmin.path()); // '/blog/admin'
```

The behavior of this method can become very complicated in complex cases of mounted apps: it is usually better to use req.baseUrl to get the canonical path of the app.

app.post(path, callback [, callback ...])

Routes HTTP POST requests to the specified path with the specified callback functions. For more information, see the routing guide.

You can provide multiple callback functions that behave just like middleware, except that these callbacks can invoke next('route') to bypass the remaining route callback(s). You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there's no reason to proceed with the current route.

```
app.post('/', function (req, res) {
  res.send('POST request to homepage');
});
```

app.put(path, callback [, callback ...])

Routes HTTP PUT requests to the specified path with the specified callback functions. For more information, see the routing guide.

You can provide multiple callback functions that behave just like middleware, except that these callbacks can invoke next('route') to bypass the remaining route callback(s). You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there's no reason to proceed with the current route.

```
app.put('/', function (req, res) {
  res.send('PUT request to homepage');
});
```

app.render(view, [locals], callback)
Returns the rendered HTML of a view via the callback function. It accepts an optional parameter that is an object containing local variables for the view. It is like res.render(), except it cannot send the rendered view to the client on its own.

Think of app.render() as a utility function for generating rendered view strings. Internally res.render() uses app.render() to render views.
The local variable cache is reserved for enabling view cache. Set it to true, if you want to cache view during development; view caching is enabled in production by default.

```
app.render('email', function(err, html){
  // ...
});

app.render('email', { name: 'Tobi' }, function(err, html){
  // ...
});
```

app.route(path)
Returns an instance of a single route, which you can then use to handle HTTP verbs with optional middleware. Use app.route() to avoid duplicate route names (and thus typo errors).

```
var app = express();

app.route('/events')
.all(function(req, res, next) {
  // runs for all HTTP verbs first
  // think of it as route specific middleware!
})
.get(function(req, res, next) {
  res.json(...);
})
.post(function(req, res, next) {
  // maybe add a new event...
})
```

app.set(name, value)
Assigns setting name to value, where name is one of the properties from the app settings table.

Calling app.set('foo', true) for a Boolean property is the same as calling app.enable('foo'). Similarly, calling app.set('foo', false) for a Boolean property is the same as calling app.disable('foo').

Retrieve the value of a setting with app.get().

```
app.set('title', 'My Site');
app.get('title'); // "My Site"
```