



Node.js

讲师：李立超



Node.js 简介

服务器端，js 运行环境

- Node.js 是一个能够在服务器端运行 JavaScript 的开放源代码、跨平台 **JavaScript 运行环境**。
- Node 采用 Google 开发的 V8 引擎运行 js 代码，使用 **事件驱动、非阻塞和异步 I/O 模型** 等技术来提高性能，可优化应用程序的传输量和规模。
- Node 大部分基本模块都用 JavaScript 编写。在 Node 出现之前，JS 通常作为客户端程序设计语言使用，以 JS 写出的程序常在用户的浏览器上运行。
- 目前，Node 已被 IBM、Microsoft、Yahoo!、Walmart、Groupon、SAP、LinkedIn、Rakuten、PayPal、Voxer 和 GoDaddy 等企业采用。

简介

web服务器可以处理浏览器等Web客户端的请求并返回相应响应

- Node主要用于编写像Web服务器一样的网络应用，这和PHP和Python是类似的。
- 但是Node与其他语言最大的不同之处在于，PHP等语言是阻塞的而Node是非阻塞的。
- Node是事件驱动的。开发者可以在不使用线程的情况下开发出一个能够承载高并发的服务器。其他服务器端语言难以开发高并发应用，而且即使开发出来，性能也不尽人意。
- Node正是在这个前提下被创造出来。
- Node把JS的易学易用和Unix网络编程的强大结合到了一起。

简介

- Node.js允许通过JS和一系列模块来编写服务器端应用和网络相关的应用。
- 核心模块包括文件系统I/O、网络（HTTP、TCP、UDP、DNS、TLS/SSL等）、二进制数据流、加密算法、数据流等等。Node模块的API形式简单，降低了编程的复杂度。
- 使用框架可以加速开发。常用的框架有Express.js、Socket.IO和Connect等。Node.js的程序可以在Microsoft Windows、Linux、Unix、Mac OS X等服务器上运行。
- Node.js也可以使用CoffeeScript、TypeScript、Dart语言，以及其他能够编译成JavaScript的语言编程。

瑞安·达尔 (Ryan Dahl)

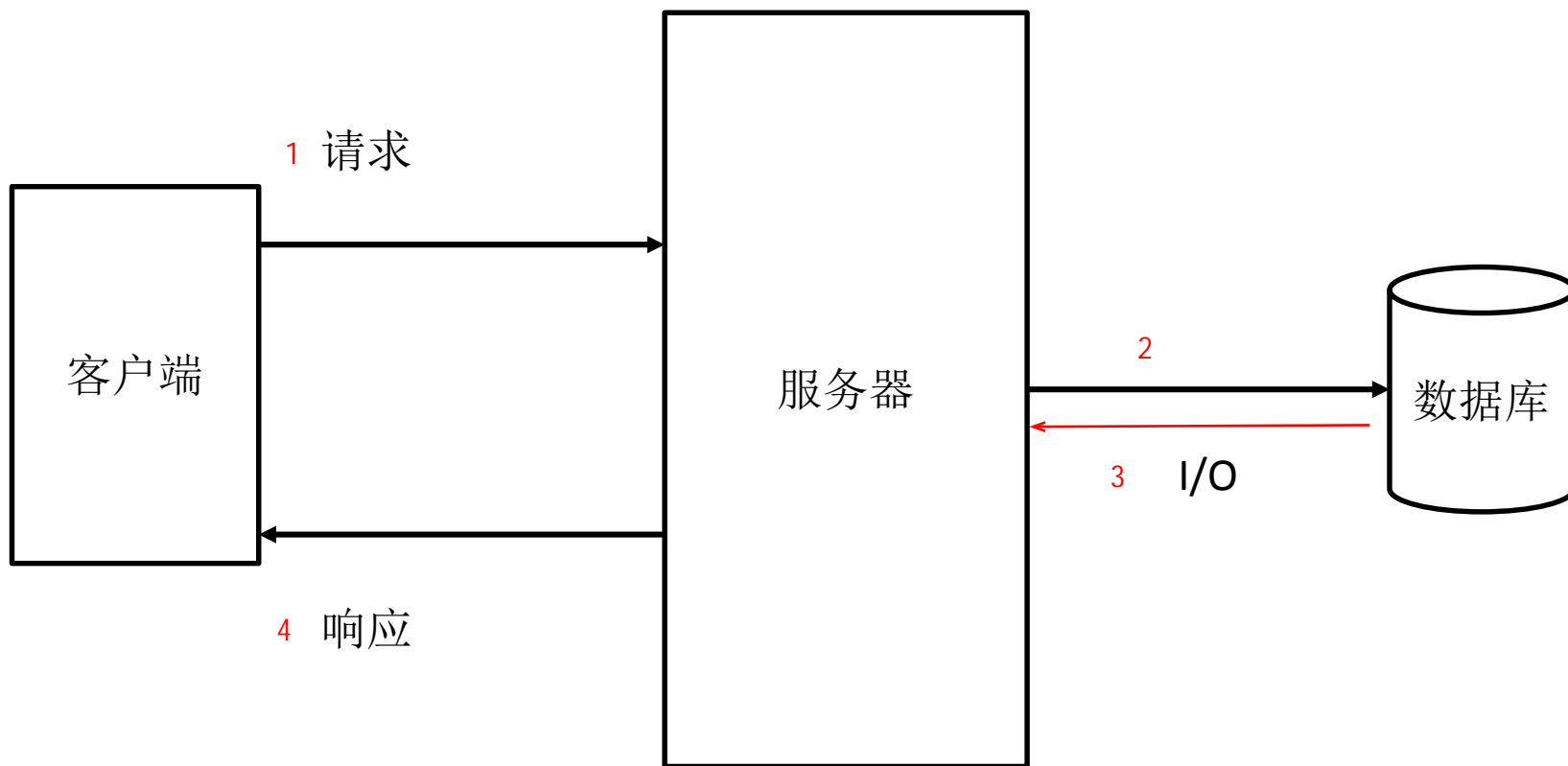
- Ryan Dahl并非科班出身的开发者，在2004年的时候他还在纽约的罗彻斯特大学数学系读博士。
- 2006年，也许是厌倦了读博的无聊，他产生了『世界那么大，我想去看看』的念头，做出了退学的决定，然后一个人来到智利的Valparaiso小镇。
- 从那时起，Ryan Dahl不知道是否因为生活的关系，他开始学习网站开发了，走上了码农的道路。
- 那时候Ruby on Rails很火，他也不例外的学习了它。
- 从那时候开始，Ryan Dahl的生活方式就是接项目，然后去客户的地方工作，在他眼中，拿工资和上班其实就是去那里旅行。

- Ryan Dahl经过两年的工作后，成为了高性能Web服务器的专家，从接开发应用到变成专门帮客户解决性能问题的专家。
- 期间他开始写一些开源项目帮助客户解决Web服务器的高并发性能问题，他尝试了很多种语言，但是最终都失败了。
- 在他快绝望的时候，V8引擎来了。V8满足他关于高性能Web服务器的想象。于是在2009年2月它开始着手编写Node.js



传统服务器使用多线程处理请求。一个线程处理一个请求，然后另一个线程向数据库发送读取数据的请求，只有数据库响应后，处理请求的线程再响应，否则就一直阻塞。

而nodejs使用单线程处理。这个线程处理所有请求，在等待数据库响应时，它会处理其他请求，而不是阻塞。



Node的历史

时间	事件
2009年	瑞安·达尔（Ryan Dahl）在GitHub上发布node的最初版本
2010年1月	Node的包管理器npm诞生
2010年底	Joyent公司赞助Node的开发，瑞安·达尔加入旗下，全职负责Node
2011年7月	Node在微软的帮助下发布了windows版本
2011年11月	Node超越Ruby on Rails，称为GitHub上关注度最高的项目
2012年1月	瑞安·达尔离开Node项目
2014年12月	Fedor Indutny在2014年12月制作了分支版本，并起名“io.js”
2015年初	Node.js基金会成立（IBM、Intel、微软、Joyent）
2015年9月	Node.js和io.js合并，Node 4.0发布
2016年	Node 6.0发布
2017年	Node 8.0发布

奇数版本是开发版，偶数版本是稳定版，使用偶数版

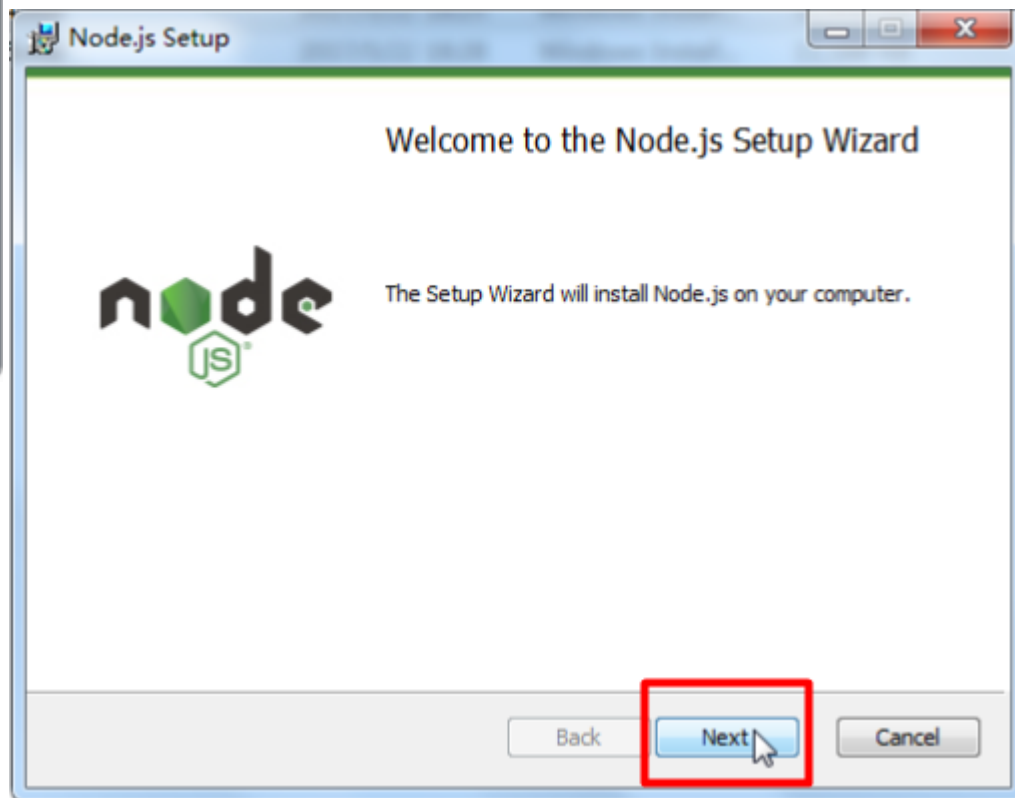
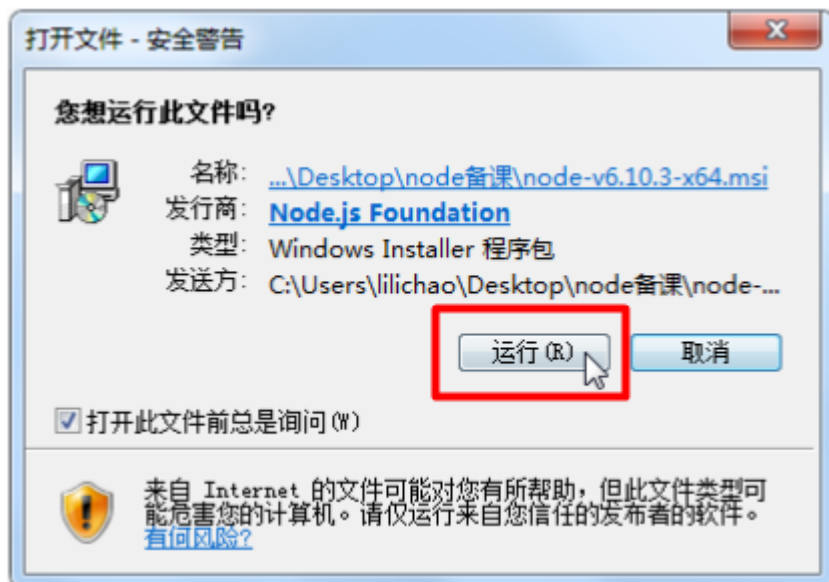
Node的用途

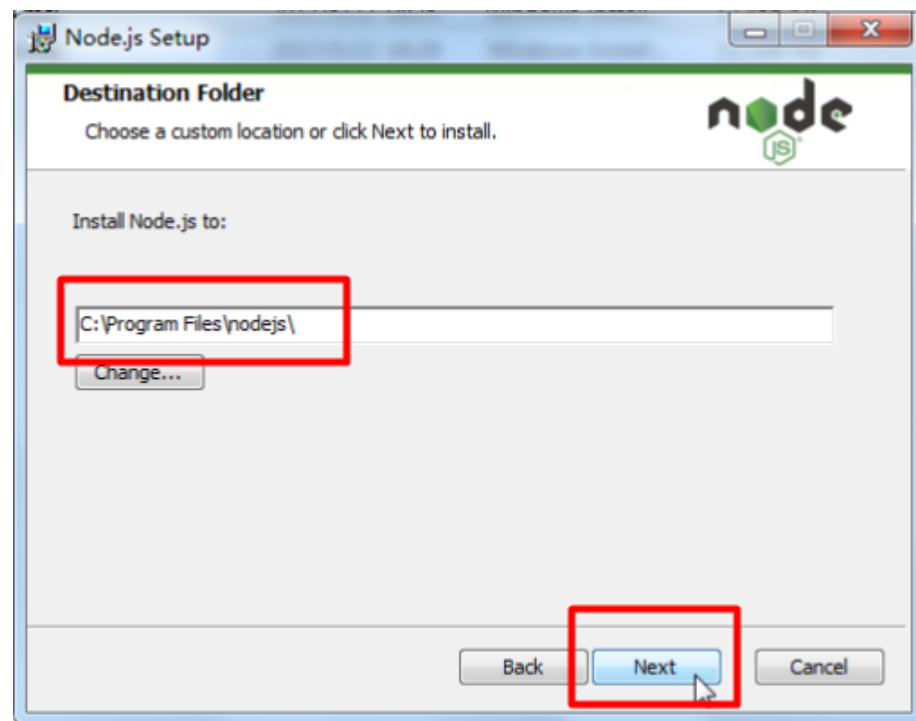
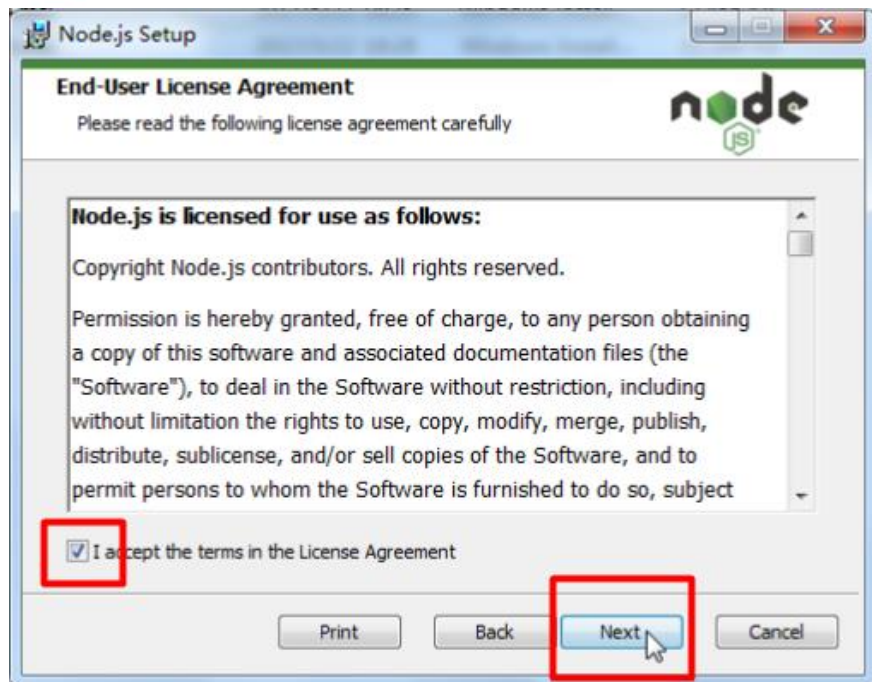
- Web服务API，比如REST
- 实时多人游戏
- 后端的Web服务，例如跨域、服务器端的请求
- 基于Web的应用
- 多客户端的通信，如即时通信

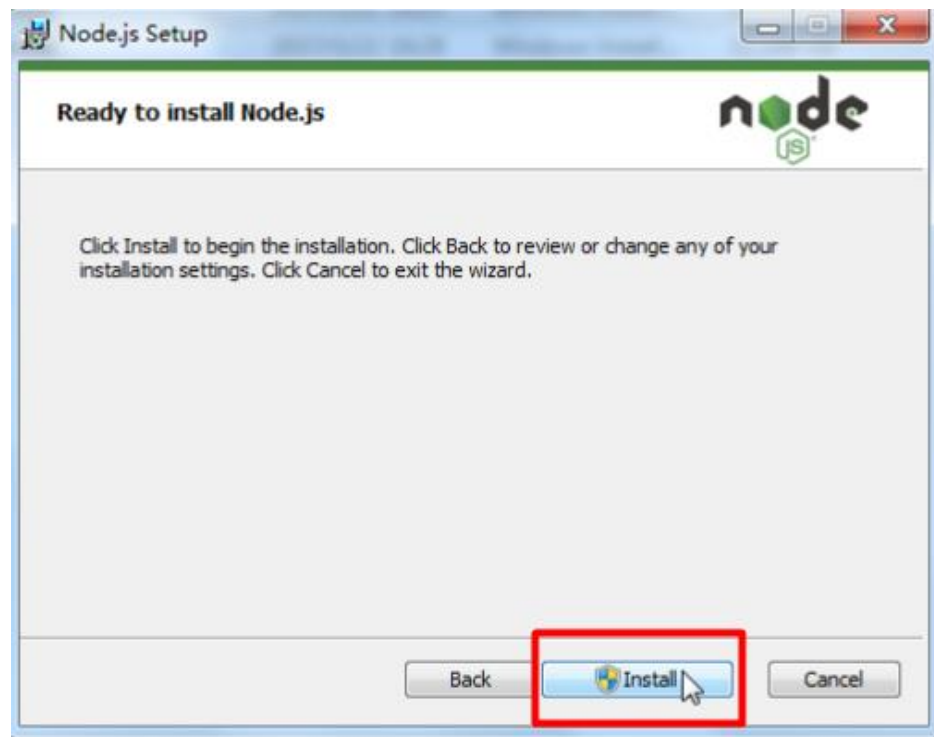
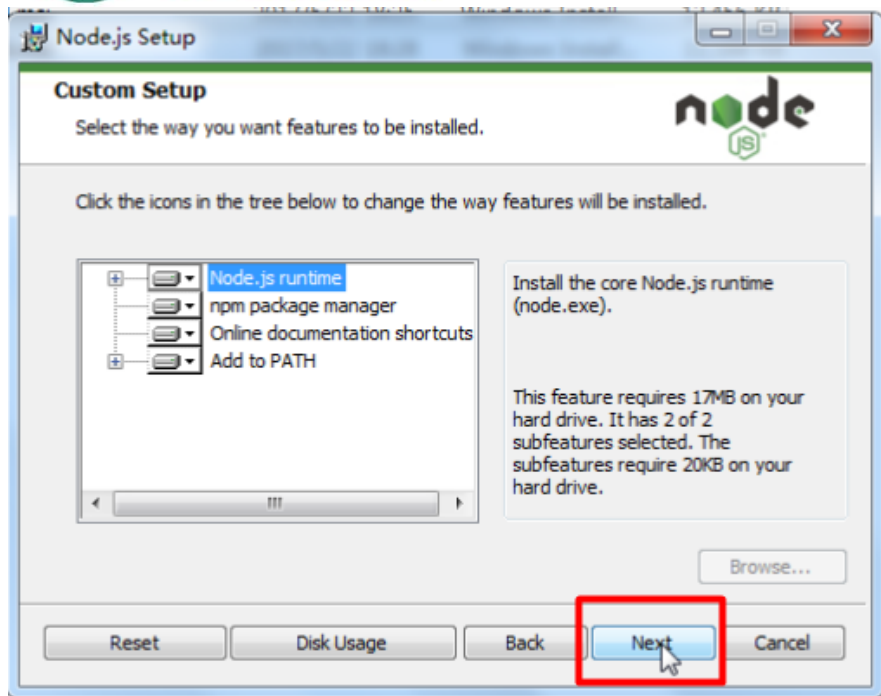
万事开头难

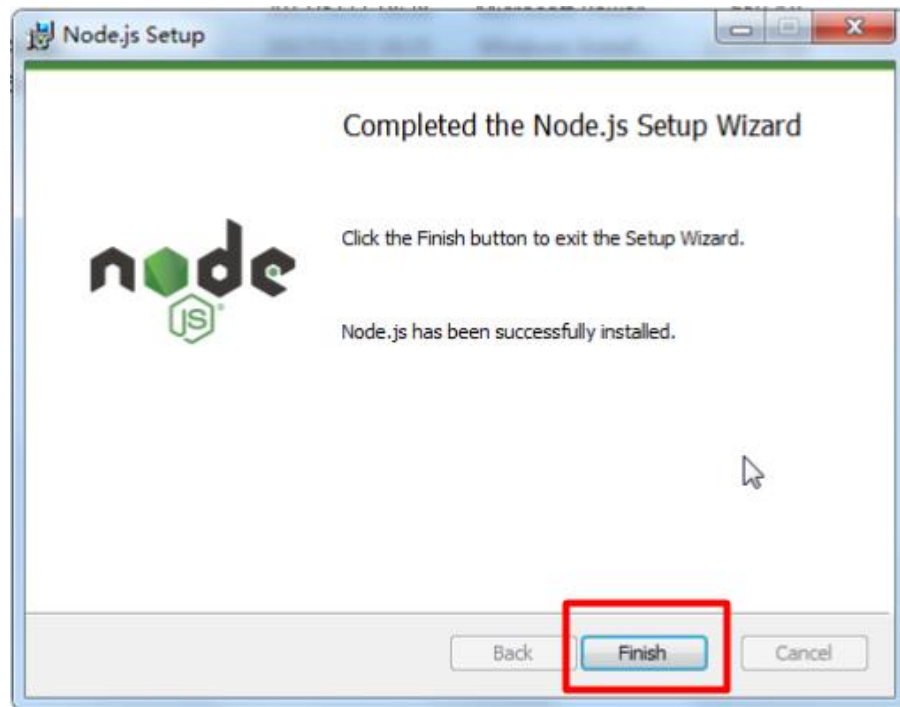
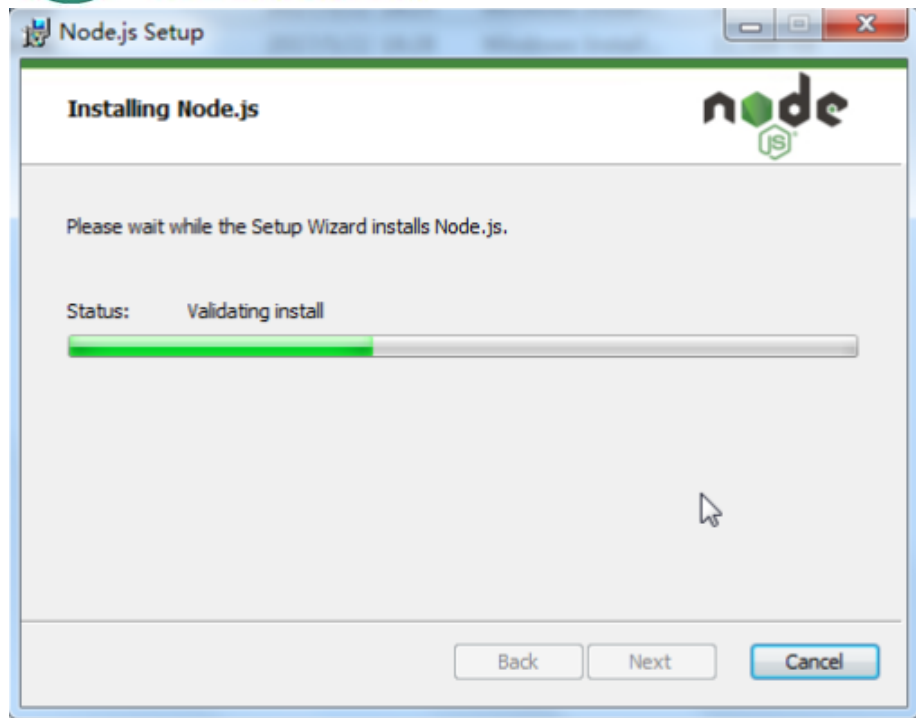
安装NODE.JS

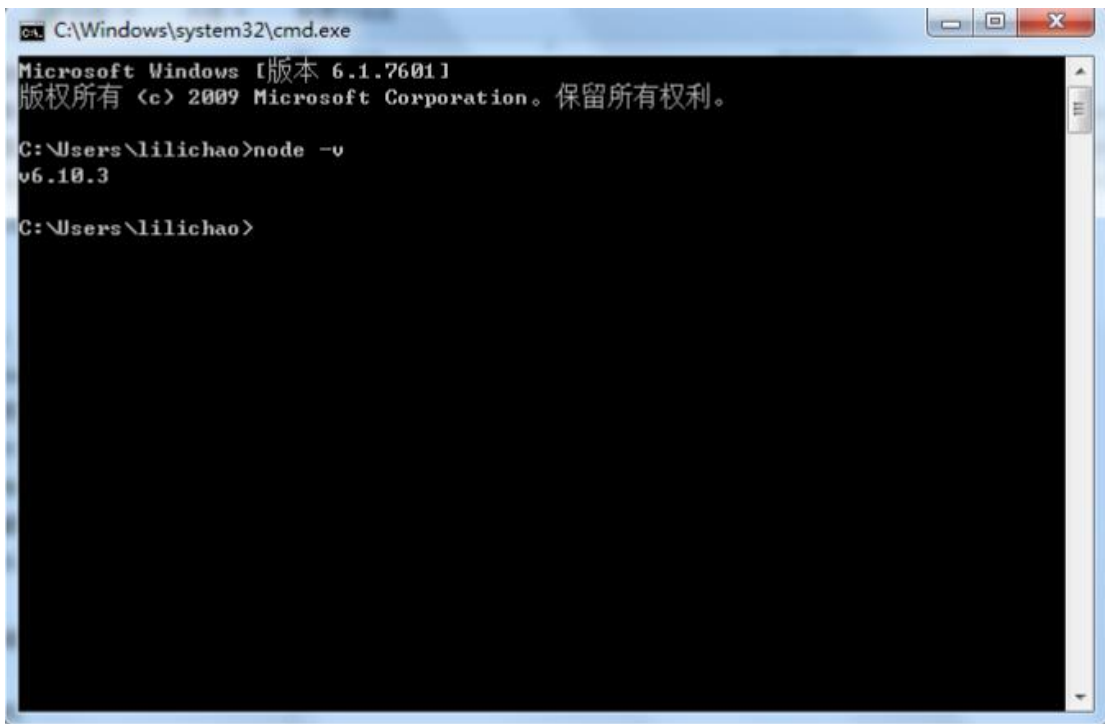
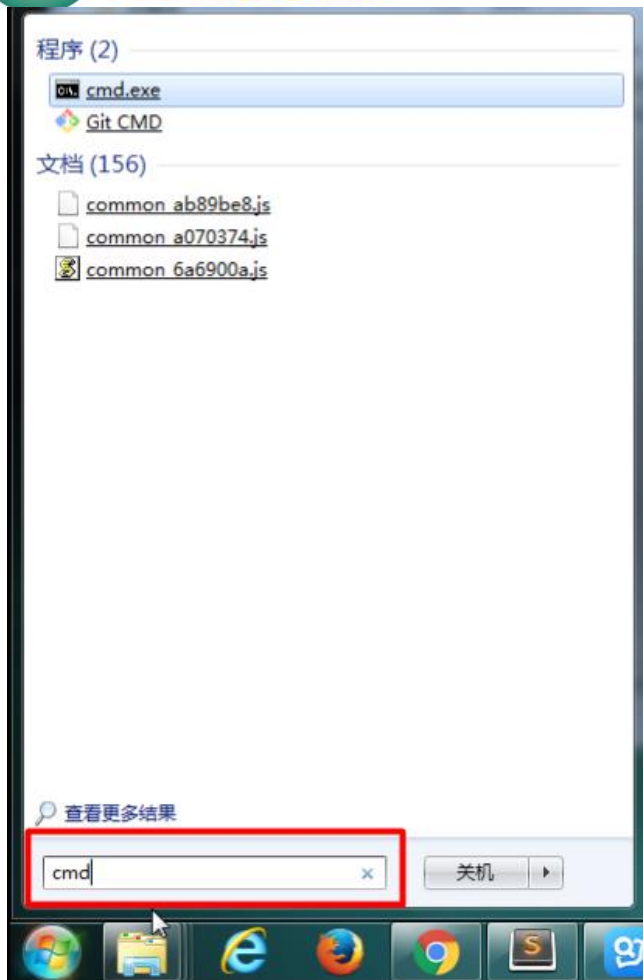
Node安装步骤


















node目录结构

	node_modules	20
	node.exe	20
	node_etw_provider.man	20
	node_perfctr_provider.man	20
	nodevars.bat	20
	npm	20
	npm.cmd	20

node包目录

node启动文件

npm启动文件

node hello.js 执行js文件；

也可以在IDEA中创建ndoejs工程，直接run。
如果没有配置node环境，需要先配置（setting --> Nodejs and NPM）

不得不提的一个东东

COMMONJS规范

ECMAScript标准的缺陷

- 没有模块系统
- 标准库较少
- 没有标准接口
- 缺乏管理系统

模块化

- 如果程序设计的规模达到了一定程度，则必须对其进行模块化。
- 模块化可以有多种形式，但至少应该提供能够将代码分割为多个源文件的机制。
- CommonJS 的模块功能可以帮助我们解决该问题。

CommonJS规范

- CommonJS规范的提出，主要是为了弥补当前JavaScript没有模块化标准的缺陷。
- CommonJS规范为JS指定了一个美好的愿景，希望JS能够在任何地方运行。
- CommonJS对模块的定义十分简单：
 - 模块引用
 - 模块定义
 - 模块标识

模块引用

- 在规范中，定义了require()方法，这个方法接手模块标识，以此将一个模块引入到当前运行环境中。
- 模块引用的示例代码：
 - `var math = require('math');`

模块定义

- 在运行环境中，提供了 **exports** 对象用于导出当前模块的方法或者变量，并且它是唯一的导出的出口。
- 在模块中还存在一个 **module** 对象，它代表模块自身，而 **exports** 是 **module** 的属性。
- 在 Node 中一个文件就是一个模块。

模块定义

```
exports.xxx = function() {};
```

```
module.exports = {};
```

模块标识

- 模块标识其实就是模块的名字，也就是传递给`require()`方法的参数，它必须是符合驼峰命名法的字符串，或者是以`..`开头的相对路径、或者绝对路径。
- 模块的定义十分简单，接口也十分简洁。每个模块具有独立的空间，它们互不干扰，在引用时也显得干净利落。

Node的模块实现

- Node中虽然使用的是CommonJS规范，但是其自身也对规范做了一些取舍。
- 在Node中引入模块，需要经历如下3个步骤：
 - 路径分析
 - 文件定位
 - 编译执行
- 在Node中，模块分为三类：一类是底层由C++编写的**内建模块**，一类是Node提供的**核心模块**；还有一类是用户编写的模块，称为**文件模块**。

包 package

- CommonJS的包规范允许我们将一组相关的模块组合到一起，形成一组完整的工具。
- CommonJS的包规范由包结构和包描述文件两个部分组成。
- 包结构
 - 用于组织包中的各种文件
- 包描述文件
 - 描述包的相关信息，以供外部读取分析

包结构

- 包实际上就是一个压缩文件，解压以后还原为目录。符合规范的目录，应该包含如下文件：
 - **package.json** **描述文件**
 - bin 可执行二进制文件
 - lib js代码
 - doc 文档
 - test 单元测试

包描述文件

- 包描述文件用于表达非代码相关的信息，它是一个JSON格式的文件 – package.json，位于包的根目录下，是包的重要组成部分。
- package.json中的字段
 - name、description、version、keywords、maintainers、contributors、bugs、licenses、repositories、dependencies、homepage、os、cpu、engine、builtin、directories、implements、scripts、author、bin、main、devDependencies。

NPM (Node Package Manager)

- CommonJS包规范是理论，NPM是其中一种实践。
- 对于Node而言，NPM帮助其完成了第三方模块的发布、安装和依赖等。借助NPM，Node与第三方模块之间形成了很好的一个生态系统。

NPM命令

- `npm -v`
 - 查看版本
- `npm`
 - 帮助说明
- `npm search 包名`
 - 搜索模块包
- `npm install 包名`
 - 在当前目录安装包
- `npm install 包名 -g`
 - 全局模式安装包

`npm install` 下载当前项目所依赖的包

`npm install 包名 --save` 安装包并添加到依赖中
(这样, 克隆下的项目, 可以执行 `npm install` 一次性
下载项目所需要的包)

(全局安装的包一般都是一些工具)

NPM命令

配置淘宝镜像：

```
npm install -g cnpm --registry=https://registry.npm.taobao.org
```

```
npm config set registry https://registry.npm.taobao.org
```

- npm remove 包名
 - 删除一个模块
- npm install 文件路径
 - 从本地安装
- npm install 包名 --registry=地址
 - 从镜像源安装
- npm config set registry 地址
 - 设置镜像源

Buffer (缓冲区)

- 从结构上看Buffer非常像一个数组，它的元素为16进制的两位数。
- 实际上一个元素就表示内存中的一个字节。
- 实际上Buffer中的内存不是通过JavaScript分配的，而是在底层通过C++申请的。
- 也就是我们可以直接通过Buffer来创建内存中的空间。

Buffer的操作

- 使用Buffer保存字符串

```
let str = "你好 atguigu";  
let buf = Buffer.from(str, "utf-8");
```

- 创建指定大小的Buffer对象

```
let buf3 = Buffer.alloc(1024*8)
```

Buffer 的转换

- Buffer与字符串间的转换
 - 支持的编码:
 - ASCII、UTF-8、UTF-16LE/UCS-2、Base64、Binary、Hex
 - 字符串转Buffer
 - `Buffer.from(str , [encoding]);`
 - Buffer转字符串
 - `buf.toString([encoding] , [start] , [end]);`

写入操作

- 向缓冲区中写入字符串
 - `buf.write(string[, offset[, length]][, encoding])`
- 替换指定索引位置的数据
 - `buf[index]`
- 将指定值填入到缓冲区的指定位置
 - `buf.fill(value[, offset[, end]][, encoding])`

读取操作

- 将缓冲区中的内容，转换为一个字符串返回
 - `buf.toString([encoding[, start[, end]]])`
- 读取缓冲区指定索引的内容
 - `buf[index]`

其他操作

- 复制缓冲区
 - `buf.copy(target[, targetStart[, sourceStart[, sourceEnd]]])`
- 对缓冲区切片
 - `buf.slice([start[, end]])`
- 拼接缓冲区
 - `Buffer.concat(list[, totalLength])`

fs（文件系统）

- 在Node中，与文件系统的交互是非常重要的，服务器的本质就将本地的文件发送给远程的客户端
- Node通过fs模块来和文件系统进行交互
- 该模块提供了一些标准文件访问API来打开、读取、写入文件，以及与其交互。
- 要使用fs模块，首先需要对其进行加载
 - `const fs = require("fs");`

同步和异步调用

- fs模块中所有的操作都有两种形式可供选择**同步**和**异步**。
- 同步文件系统会**阻塞**程序的执行，也就是除非操作完毕，否则不会向下执行代码。
- 异步文件系统**不会阻塞**程序的执行，而是在操作完成时，通过回调函数将结果返回。

打开和关闭文件

- 打开文件
 - `fs.open(path, flags[, mode], callback)`
 - `fs.openSync(path, flags[, mode])`
- 关闭文件
 - `fs.close(fd, callback)`
 - `fs.closeSync(fd)`

打开状态

模式	说明
r	读取文件，文件不存在则出现异常
r+	读写文件，文件不存在则出现异常
rs	在同步模式下打开文件用于读取
rs+	在同步模式下打开文件用于读写
w	打开文件用于写操作，如果不存在则创建，如果存在则截断
wx	打开文件用于写操作，如果 存在 则打开失败
w+	打开文件用于读写，如果不存在则创建，如果存在则截断
wx+	打开文件用于读写，如果 存在 则打开失败
a	打开文件用于追加，如果不存在则创建
ax	打开文件用于追加，如果路径存在则失败
a+	打开文件进行读取和追加，如果不存在则创建该文件
ax+	打开文件进行读取和追加，如果路径存在则失败

写入文件

- fs中提供了四种不同的方式将数据写入文件
 - 简单文件写入
 - 同步文件写入
 - 异步文件写入
 - 流式文件写入

简单文件写入

- `fs.writeFile(file, data[, options], callback)`
- `fs.writeFileSync(file, data[, options])`
- 参数：
 - file 文件路径
 - data 被写入的内容，可以是String或Buffer
 - options 对象，包含属性（ encoding、 mode、 flag ）
 - callback 回调函数

同步文件写入

- `fs.writeFileSync(fd, buffer, offset, length[, position])`
- `fs.writeFileSync(fd, data[, position[, encoding]])`
- 要完成同步写入文件，先需要通过`openSync()`打开文件来获取一个文件描述符，然后在通过`writeSync()`写入文件。
- 参数
 - `fd` 文件描述符，通过`openSync()`获取
 - `data` 要写入的数据（String 或 Buffer）
 - `offset` buffer写入的偏移量
 - `length` 写入的长度
 - `position` 写入的起始位置
 - `encoding` 写入编码

异步文件写入

- `fs.write(fd, buffer, offset, length[, position], callback)`
- `fs.write(fd, data[, position[, encoding]], callback)`
- 要使用异步写入文件，先需要通过`open()`打开文件，然后在回调函数中通过`write()`写入。
- 参数：
 - `fd` 文件描述符
 - `data` 要写入的数据（String 或 Buffer）
 - `offset` buffer写入的偏移量
 - `length` 写入的长度
 - `position` 写入的起始位置
 - `encoding` 写入编码

流式文件写入

- 往一个文件中写入大量数据时，最好的方法之一是使用流。
- 若要将数据异步传送到文件，首需要使用以下语法创建一个Writable对象：
 - `fs.createWriteStream(path[, options])`
 - path 文件路径
 - options {encoding:"",mode:"",flag:""}
- 一旦你打开了Writable文件流，就可以使用 `write()` 方法来写入它，写入完成后，在调用 `end()` 方法来关闭流。

读取文件

- fs中提供了四种读取文件的方式
 - 简单文件读取
 - 同步文件读取
 - 异步文件读取
 - 流式文件读取

简单文件读取

- `fs.readFile(file[, options], callback)`
- `fs.readFileSync(file[, options])`
 - 参数：
 - `file` 文件路径或文件描述符
 - `options` `<Object> | <String>`
 - `encoding` `<String> | <Null>` 默认 = `null`
 - `flag` `<String>` 默认 = `'r'`
 - `callback` 回调函数，有两个参数 `err`、`data`

同步文件读取

- `fs.readFileSync(fd, buffer, offset, length, position)`
 - 参数：
 - `fd` 文件描述符
 - `buffer` 读取文件的缓冲区
 - `offset` `buffer`的开始写入的位置
 - `length` 要读取的字节数
 - `position` 开始读取文件的位置

异步文件读取

- `fs.read(fd, buffer, offset, length, position, callback)`

– 参数：

- `fd` 文件描述符
- `buffer` 读取文件的缓冲区
- `offset` `buffer`的开始写入的位置
- `length` 要读取的字节数
- `position` 开始读取文件的位置
- `callback` 回调函数 参数`err` , `bytesRead` , `buffer`

流式文件读取

- 从一个文件中读取大量的数据时，最好的方法之一就是流式读取，这样将把一个文件作为Readable流的形式打开。
- 要从异步从文件传输数据，首先需要通过以下语法创建一个Readable流对象：
 - `fs.createReadStream(path[, options])`
 - path 文件路径
 - options {encoding:"",mode:"",flag:""}
- 当你打开Readable文件流以后，可以通过readable事件和read()请求，或通过data事件处理程序轻松地从它读出。

其他操作

- 验证路径是否存在
 - ~~fs.exists(path, callback)~~
 - fs.existsSync(path)
- 获取文件信息
 - fs.stat(path, callback)
 - fs.statSync(path)
- 删除文件
 - fs.unlink(path, callback)
 - fs.unlinkSync(path)

其他操作

- 列出文件
 - `fs.readdir(path[, options], callback)`
 - `fs.readdirSync(path[, options])`
- 截断文件
 - `fs.truncate(path, len, callback)`
 - `fs.truncateSync(path, len)`
- 建立目录
 - `fs.mkdir(path[, mode], callback)`
 - `fs.mkdirSync(path[, mode])`

其他操作

- 删除目录
 - `fs.rmdir(path, callback)`
 - `fs.rmdirSync(path)`
- 重命名文件和目录
 - `fs.rename(oldPath, newPath, callback)`
 - `fs.renameSync(oldPath, newPath)`
- 监视文件更改写入
 - `fs.watchFile(filename[, options], listener)`

