### 全局对象

#### 全局变量global

在Node.js环境中，也有唯一的全局对象，但不叫window，而叫global，这个对象的属性和方法也和浏览器环境的window不同。

#### 进程对象process

这说明传入process.nextTick()的函数不是立刻执行，而是要等到下一次事件循环。

Node.js进程本身的事件就由process对象来处理。

### http模块

应用程序并不直接和HTTP协议打交道，而是操作http模块提供的request和response对象。

request对象封装了HTTP请求，我们调用request对象的属性和方法就可以拿到所有HTTP请求的信息；

response对象封装了HTTP响应，我们操作response对象的方法，就可以把HTTP响应返回给浏览器。

var http = require('http');

// 创建http server,并传入回调函数

var server = http.createServer(function(req, resp) {

    console.log(req, resp);

    // 得到请求的method, url

    console.log(req.method + ':' + req.url);

    // 将状态码200和响应类型 写入响应头

    resp.writeHead(200, {'Content-Type' : 'text/html'});

    // 将HTTP响应的HTML内容写入response

    resp.end('<h1>Hello World</h1>')

});

server.listen(8080);

console.log('Server is running at http://127.0.0.1.8080');

### url模块

通过parse()将一个字符串解析为一个Url对象： 列出对象一些基本属性

search: '?name=zzj',

query: 'name=zzj',

pathname: '/01\_1http.js',

path: '/01\_1http.js?name=zzj',

href: '/01\_1http.js?name=zzj'

### path模块

path模块，它可以方便地构造目录：

var path = require('path');

// 解析当前目录的绝对路径

var workDir = path.resolve('.');

console.log(workDir);

// 组合完整路径

var filePath = path.join(workDir, 'pub', 'index.html');

### Node.js 回调函数

Node.js 异步编程的直接体现就是回调。

异步编程依托于回调来实现，但不能说使用了回调后程序就异步化了。

这样在执行代码时就没有阻塞或等待文件 I/O 操作。这就大大提高了 Node.js 的性能，可以处理大量的并发请求。

### Node.js 事件循环

Node.js 基本上所有的事件机制都是用设计模式中观察者模式实现。

Node.js 单线程类似进入一个while(true)的事件循环，直到没有事件观察者退出，每个异步事件都生成一个事件观察者，如果有事件发生就调用该回调函数.

#### 事件驱动程序

Node.js 使用事件驱动模型，当web server接收到请求，就把它关闭然后进行处理，然后去服务下一个web请求。

当这个请求完成，它被放回处理队列，当到达队列开头，这个结果被返回给用户。

因为 webserver 一直接受请求而不等待任何读写操作。（这也称之为非阻塞式IO或者事件驱动IO）

### Node.js EventEmitter

Node.js 所有的异步 I/O 操作在完成时都会发送一个事件到事件队列。

Node.js 里面的许多对象都会分发事件：一个 net.Server 对象会在每次有新连接时触发一个事件， 一个 fs.readStream 对象会在文件被打开的时候触发一个事件。 所有这些产生事件的对象都是 events.EventEmitter 的实例。

一个事件可有多个监听

### Node.js Buffer

JavaScript 语言自身只有字符串数据类型，没有二进制数据类型。

Buffer 类是随 Node 内核一起发布的核心库。Buffer 库为 Node.js 带来了一种存储原始数据的方法，可以让 Node.js 处理二进制数据，每当需要在 Node.js 中处理I/O操作中移动的数据时，就有可能使用 Buffer 库。原始数据存储在 Buffer 类的实例中。一个 Buffer 类似于一个整数数组，但它对应于 V8 堆内存之外的一块原始内存。

建议使用 Buffer.from() 接口去创建Buffer对象。而不是new Buffer()构造函数

### taf接口调用服务实例

1. 创建项目文件安装依赖

npm install @taf/taf-rpc

npm install nodemon -D

1. 编辑Hello.jce文件

定义入参

struct HelloWorldReq

    {

        0 optional  string              data;           // 入参

    };

定义返回参数

   struct HelloWorldRsp

    {

        0 optional  int                 iRet;           // 返回码

        1 optional  string              message;        // 返回信息

    };

定义接口并设置传入参数，和 返回值

 interface Hello

    {

        int test();

        // 接口调用方法名

        int helloWorld(HelloWorldReq stReq, out HelloWorldRsp stRsp);

    };

1. 安装@up/oem-cli - oem通用脚手架，可在本地编译jce

up jce2node --client 　MyTest.jce 生成xxxProxy.js，开发者引入该文件之后，可以直接调用服务端的服务 RPC: 远程过程调用

up jce2node --server MyTest.jce 生成xx.jc和 xxImp.js 完成xxImp.js具体函数

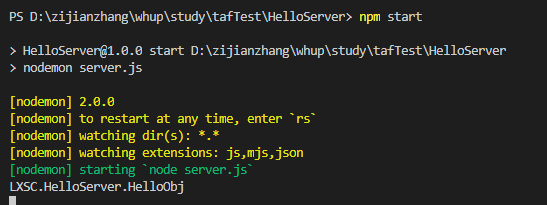
up jce2node MyTest.jce

在服务端使用 up jce2node --server Hello.jce

在client 客户端中 使用 up jce2node Hello.jce --client



1. 创建编写入口启动文件server.js, 本地配置文件local.conf, 实现HelloImp.js中的接口， 编写客户端文件，调用服务代码 client.js
2. 修改 package.json启动脚本
3. npm start 启动服务



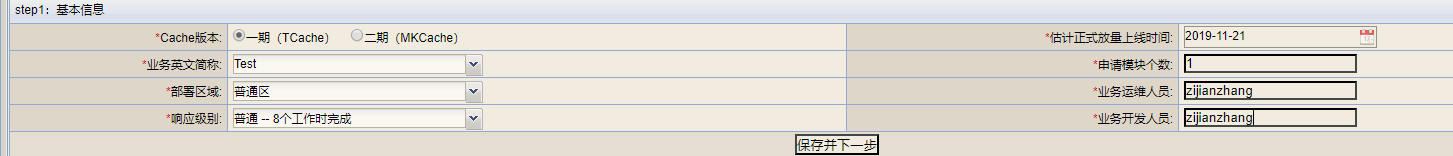
1. node client.js 允许客户端



### DCache起步

#### DCache安装

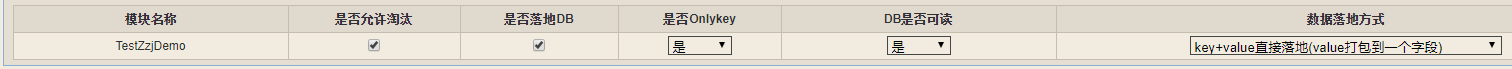
1. 进入taf管理平台，点击 运维工具-> 缓存部署 -> Cache安装
2. 找到安装DCache和DB的业务，申请一个模块，其他默认填写保存下一步



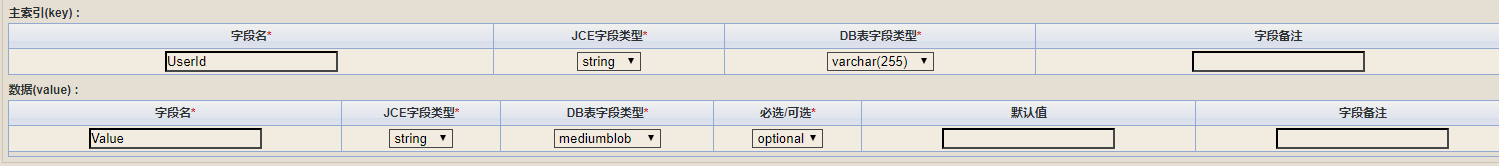
1. 取一个有标识性的模块名称，不热备，接口调用为批量+单次，确定进入下一步



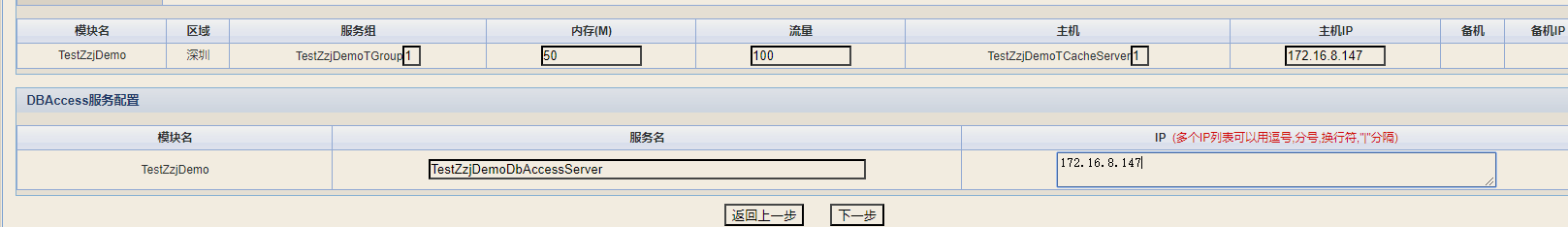
1. 允许淘汰，允许落地DB，允许Onlykey,DB可读，落地方式为key+value直接落地



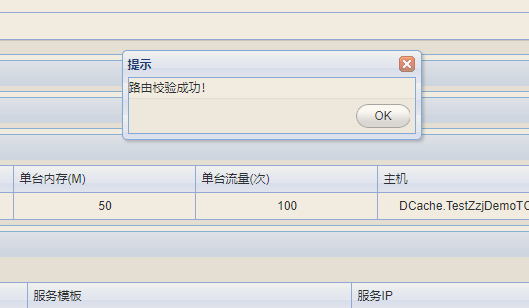
1. 主索引字段类型为varchar(255),数据的JCE类型为string,DB字段类型为mediumblob，可选



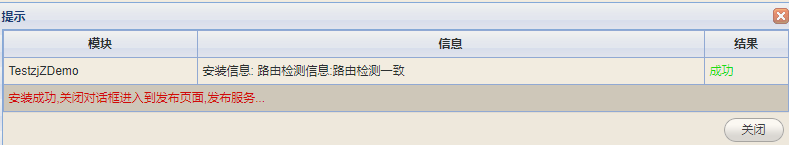
1. 点下一步，进入DB实例填写，
2. 选择数据库ip并填写用户名，密码，端口号默认3306
3. 下一步->建库-》通过->进入服务组配置->配置一个服务组
4. 填写服务器主机ip,内存默认分配50M



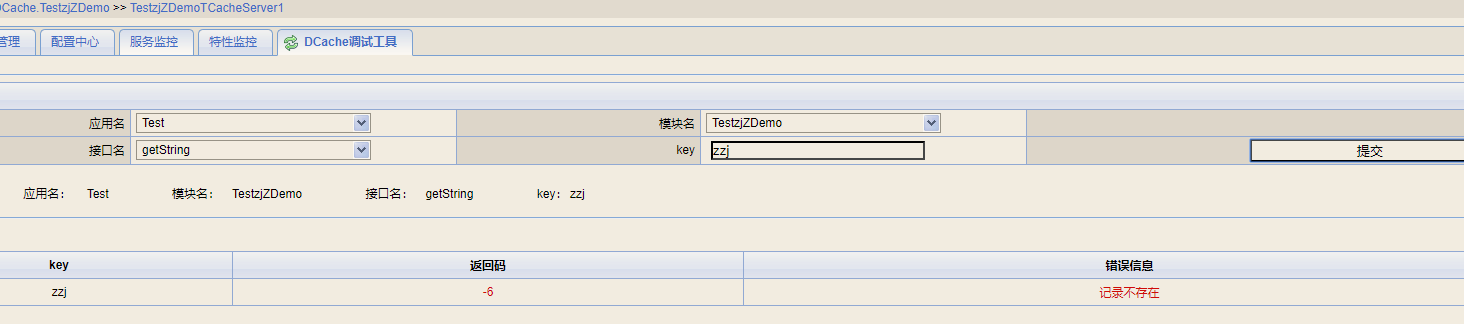
1. 下一步->默认提交->路由校验



1. 安装



1. 在缓存管理里找到对应的Test模块的Cache和DBAccess管理，发布对应的服务，发布后可以调试



#### DCache下线

#### Cache下线

运维工具->缓存部署->非Cache下线-> 找到对应的服务提交下线

#### dbAccessServer下线

运维工具->缓存部署->非Cache下线-> 找到对应的服务提交下线



### node.js.pdf

#### 高阶函数

高阶函数则是可以把函数作为参数，或是将函数最为返回值的函数

#### 偏函数：

偏函数是固定一个函数的一个或多个参数，然后返回一个新函数

// 入参函数

function add(a, b, c) {

    return a + b + c

}

// 生产偏函数

function partial(fn, a) {

    return function(b, c) {

        return fn(a, b, c)

    }

}

// 传入 入参函数和固定参数，并接收返回的新函数

const parAdd = partial(add, 66);

console.log(parAdd(1, 2)); //利用返回的新函数 再传入剩余的参数