项目开发

学习目标

1. 独立完成测试用chaincode编写
2. 独立完成客户端后端项目编写
3. 独立完成客户端前端项目编写

测试案例

chaincode编写

目标：区块链的数据读写

chaincode编写步骤：

第一步：管理依赖（shim、peer）

第二步：实现Chaincode接口

第三步：主函数中启动Chaincode

第四步：设计读写方法

第五步：完成读写方法入口函数编写（Invoke）

管理依赖

1、从`https://github.com/hyperledger/fabric`下载fabric项目代码放到$GOPATH下。

2、使用Golang开发工具创建一个新项目，在新项目中创建test.go文件。

3、添加chaincode开发需要的最核心的依赖文件。

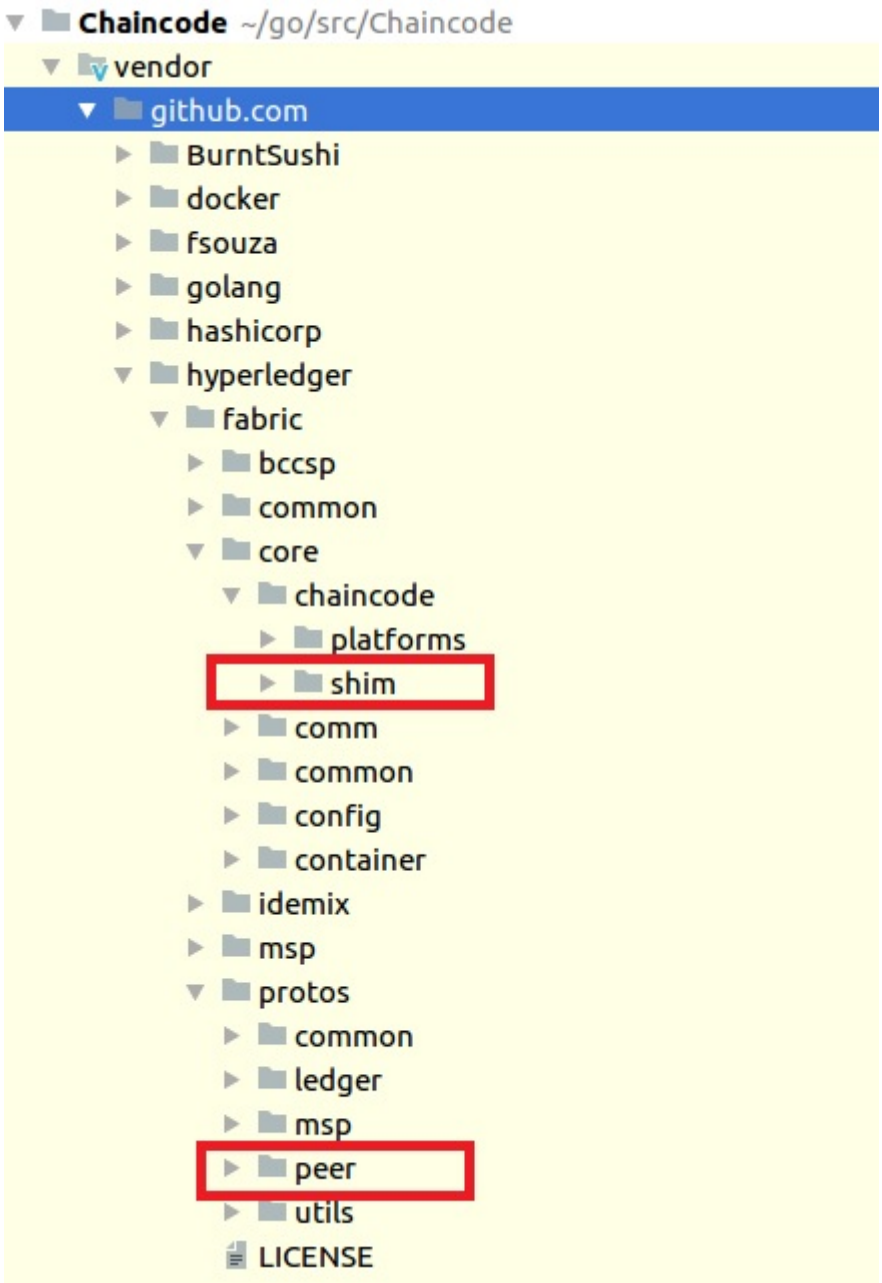
```
import (
    _ "github.com/hyperledger/fabric/core/chaincode/shim"
    _ "github.com/hyperledger/fabric/protos/peer"
)
```

> 现阶段只是导入依赖，所以前面使用"_"。后续的开发中用到了对应的工具删除"_"即可

4、运行govendor的init和add命令，添加项目用到的依赖信息。

```
govendor init
govendor add +e
```

> 注意核对下图标红信息



实现Chaincode接口

在shim目录下面，找到interfaces_stable.go文件，Chaincode接口内容如下：

```
type Chaincode interface {
    // Init is called during Instantiate transaction after the chaincode container
    // has been established for the first time, allowing the chaincode to
    // initialize its internal data
    Init(stub ChaincodeStubInterface) pb.Response

    // Invoke is called to update or query the ledger in a proposal transaction.
    // Updated state variables are not committed to the ledger until the
    // transaction is committed.
    Invoke(stub ChaincodeStubInterface) pb.Response
}
```

我们需要实现Init和Invoke方法，现阶段仅仅有简单的返回值即可。

```
type Test struct {
    // 测试用chaincode,用于信息的简单读写
}
/*
 * 实现Chaincode接口
 */
func (this *Test) Init(stub shim.ChaincodeStubInterface) pb.Response {
    // 初始化,调用一次
    return shim.Success(nil)
}

func (this *Test) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    // 更新或查询ledger的入口
    return shim.Success(nil)
}
```

主函数中启动Chaincode

在我们的 Chaincode中有一个主函数（main），我们直接在里面启动Chaincode即可。

```
func main(){
    shim.Start(new(Test))
}
```

> 测试：Start出现异常时的服务器反馈。

Start方法中需要传递Chaincode类型的参数。

```
func Start(cc Chaincode) error
```

设计读写方法

我们通过get和set方法来完成数据的读写操作（依据个人喜好：read、write）。

读取数据方法：

1. 利用ChaincodeStubInterface获取key对应的数据
2. 非空和异常处理
3. 返回读取的结果

```
/*
 * 设计读写方法
 * get/set方法
 */
func (this *Test) get(stub shim.ChaincodeStubInterface,key string) pb.Response {
    // 依据key值进行读取
    // 注意:仅读取已经提交到ledger中的数据,如果state database中不含有key信息,会返回空的数据和异常
    data, err := stub.GetState(key)
    // 异常处理
    if err!=nil{
        return shim.Error(err.Error())
    }
    // data处理
    if len(data)==0{
        // 数据不存在
        return shim.Error("Data not Available")
    }
    return shim.Success(data)
}
```

ChaincodeStubInterface接口部分代码信息如下：

```
type ChaincodeStubInterface interface {
    // GetArgs returns the arguments intended for the chaincode Init and Invoke
    // as an array of byte arrays.
    // 以byte数组的数组的形式获得传入的参数列表
    GetArgs() [][]byte

    // 以字符串数组的形式获得传入的参数列表
    GetStringArgs() []string

    // 将字符串数组的参数分为两部分，数组第一个字是Function，剩下的都是Parameter
    GetFunctionAndParameters() (string, []string)

    // 以byte切片的形式获得参数列表
    GetArgsSlice() ([]byte, error)

    // 核心的操作就是对State Database的增删改查
    // 增加和修改数据是统一的操作
    // PutState puts the specified `key` and `value` into the transaction's
    // writeset as a data-write proposal. PutState doesn't effect the ledger
    // until the transaction is validated and successfully committed.
    // Simple keys must not be an empty string and must not start with null
    // character (0x00), in order to avoid range query collisions with
    // composite keys, which internally get prefixed with 0x00 as composite
    // key namespace.
    PutState(key string, value []byte) error
```

```
    // 根据Key删除State DB的数据
    // DelState records the specified `key` to be deleted in the writeset of
    // the transaction proposal. The `key` and its value will be deleted from
    // the ledger when the transaction is validated and successfully committed.
    DelState(key string) error

    // GetState returns the value of the specified `key` from the
    // ledger. Note that GetState doesn't read data from the writeset, which
    // has not been committed to the ledger. In other words, GetState doesn't
    // consider data modified by PutState that has not been committed.
    // If the key does not exist in the state database, (nil, nil) is returned.
    GetState(key string) ([]byte, error)
……
}
```

写入数据方法：

1. 利用ChaincodeStubInterface写入数据，value的类型为[]byte
2. 回复写入数据结果（异常处理）

```
func (this *Test) set(stub shim.ChaincodeStubInterface,key string,value []byte) pb.Response {
    err := stub.PutState(key, value)
    if err != nil {
        return shim.Error(err.Error())
    }
    return shim.Success(nil)
}
```

入口函数编写

Invoke方法是数据库增删改查的统一入口，我们需要依据传递的参数进行调用方法的控制。我们需要处理两个函数的调用，即读写方法的调用。

> 约定：传递的参数中方法名称为"get"调用读方法，方法名称为"set"调用写方法。所以大家在完成客户端代码的时候需要注意传递的方法名称，一旦错误，chaincode中将视为非法参数，返回错误。

操作步骤：

1. 获取传递的方法名称和参数信息
2. 依据方法名称完成调用函数的确认
3. 方法异常处理
4. 传递参数并处理返回值

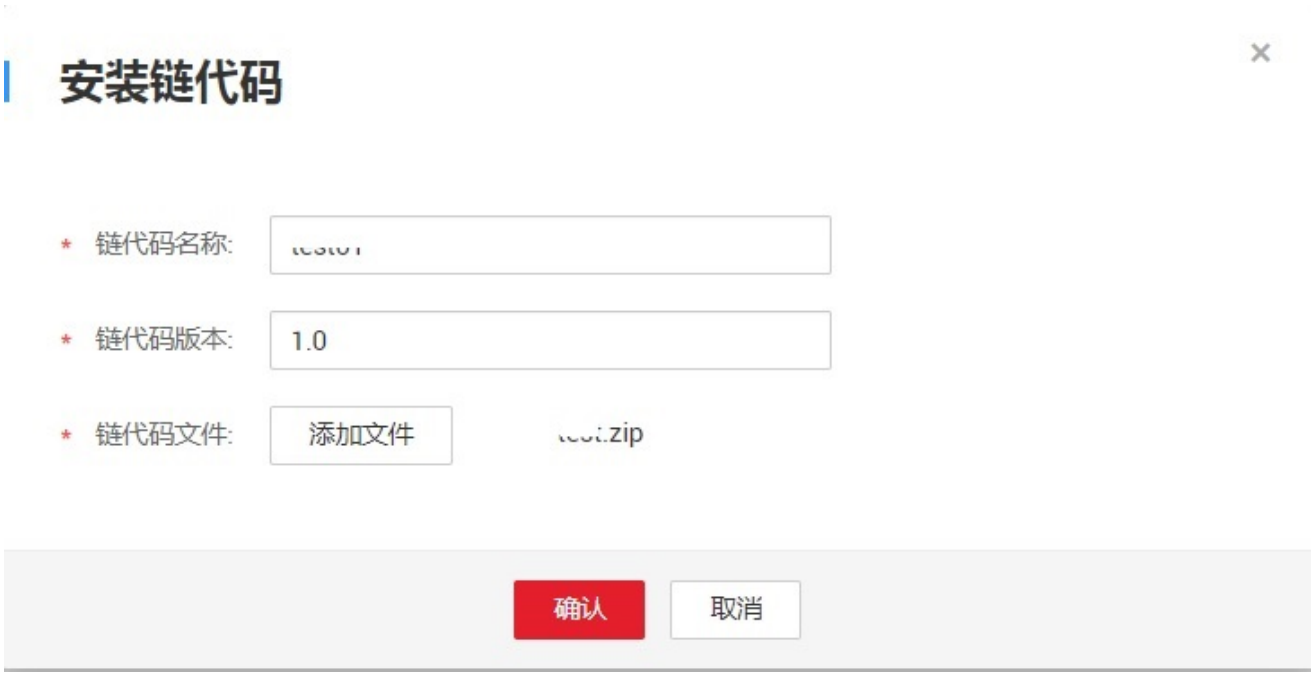
```
func (this *Test) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    // 更新或查询ledger的入口
    function, parameters := stub.GetFunctionAndParameters()

    // 依据不同的function完成读写方法调用和参数的传递
    if function=="get"{
        // 约定:key放在第一个参数的位置
        return this.get(stub,parameters[0])
```

```
    }else if function=="set" {
        // 约定:value放在第二个参数的位置
        return this.set(stub,parameters[0],[]byte(parameters[1]))
    }
    // 无效的智能合约方法名称
    return shim.Error("Invalid Smart Contract function name.")
}
```

链代码安装

操作详见课件1。



项目配置信息如下：

```
#chaincode config
chaincode_id = test02
channel_id = testchannel
CORE_XXX1_CONFIG_FILE=conf/xxx1.yaml
useid=User1
```

客户端开发

目标：客户端完成区块链的数据读写

技术分析：前后端分离方式进行案例开发，后端使用beego完成案例开发。前端使用Vue完成案例开发。

后端项目

该项目无界面部分，我们只需要接收前端请求，将结果回复给前端即可。

从区块中读数据

操作步骤：

1、构建项目，配置路由

2、在controller中，处理读数据请求

3、在models中，使用fabric-sdk-go完成读数据操作（重点）

4、在controller中，调用models完成读数据操作，并将结果返回给调用者

5、测试数据读取

第一步：构建项目，配置路由

构建beego项目`bee new test`，我们不需要在这个项目中管理界面，所以可以删除视图部分内容。配置路由，在router.go文件中填写如下内容：

```
beego.Router("/test",&controllers.MainController{},"get:GetValue")
```

将/test的get请求交给控制器的GetValue方法进行处理。

第二步：处理读数据请求

在controller中编写GetValue进行数据读取请求处理，现阶段我们仅仅对参数进行校验，还无法读取区块数据。

> 读源码查看如何进行非空判断

```
/*
 * 读取区块链中数据
 */
func (this *MainController) GetValue()  {
    // 获取请求中的key信息
    key := this.GetString("key")
    // 没有key无法进行查询
    if key==""{
        handleResponse(this,400,"Request parameter key can't be empty")
        return
    }
    beego.Info("key:"+key)

    // TODO 使用fabric-sdk-go将调用的chaincode方法和参数传递到服务器,我们需要在models中完成与区块链服务器的交换,大家可以把这个过程看做将数据库中的数据读写
    var response []byte
    // 获取结果成功后,将结果返回给前端
    handleResponse(this, 200, response)
}
```

```
/*
    处理数据回复
 */
func handleResponse(this *MainController, code int, msg interface{}) {
    if code >= 400 {
        beego.Error(msg)
    }else{
        beego.Info(msg)
    }
    this.Ctx.ResponseWriter.WriteHeader(code)
    b, ok := msg.([]byte)
    if ok {
        this.Ctx.ResponseWriter.Write(b)
    } else {
        s := msg.(string)
        this.Ctx.ResponseWriter.Write([]byte(s))
    }
}
```

第三步：用fabric-sdk-go完成读数据(*)

操作步骤：

1、构建读写方法，借鉴一下Demo中的信息

```
/*
    需要处理的方法有两种:update和query
    需要传递数据:方法名称和一组参数
 */

type ChainCodeSpec struct {
    // 操作智能合约
}

/*
    查询:依据方法和参数组
 */
func (this *ChainCodeSpec) ChaincodeQuery(function string, chaincodeArgs [][]byte) (response
[]byte, err error) {
    return nil, nil
}

/*
    更新:依据方法和参数组
 */
func (this *ChainCodeSpec) ChaincodeUpdate(function string, chaincodeArgs [][]byte) (response
[]byte, err error) {
    return nil, nil
}
```

2、依赖添加

> 注意：需要将demo中的vendor信息拷贝到我们的项目中，部分依赖直接下载会失败。
>
> 另：api和def中的信息是在hyperledger的fabric-sdk-go的基础上开发工具。

两个重要的工具如下：

`fabric-sdk-go/api/apitxn`

`fabric-sdk-go/def/fabapi`

```
import (
    _ "github.com/hyperledger/fabric-sdk-go/api/apitxn"
    _ "github.com/hyperledger/fabric-sdk-go/def/fabapi"
)
```

3、完善读方法

```
type ChainCodeSpec struct {
    // 操作智能合约
    client apitxn.ChannelClient
    chaincodeId string
}

/*
    查询:依据方法和参数组
 */
func (this *ChainCodeSpec) ChaincodeQuery(function string, chaincodeArgs [][]byte) (response
[]byte, err error) {
    request := apitxn.QueryRequest{this.chaincodeId, function, chaincodeArgs}
    return this.client.Query(request)
}
```

4、加载配置，完成client创建

通过xxx1.yaml配置文件创建sdk

```
/*
    依据配置文件加载SDK
 */
func getSDK(config string) (*fabapi.FabricSDK,error)  {
    options := fabapi.Options{ConfigFile:config}
    sdk, err := fabapi.NewSDK(options)
    if err!=nil{
        beego.Error(err.Error())
        return nil,err
    }
    return sdk,nil
}
```

通过sdk创建client

```
func Initialize(channelID, chainCodeId, userId string)  (*ChainCodeSpec,error) {
    var config string=  beego.AppConfig.String("CORE_XXX1_CONFIG_FILE")
    sdk, err := getSDK(config)
    if err != nil {
        return nil, err
    }

    client, err := sdk.NewChannelClient(channelID,userId)
    if err != nil {
        return nil, err
    }

    return &ChainCodeSpec{client: client, chainCodeId: chainCodeId}, nil
}
```

第四步：controllers读数据代码完善

```
/*
 * 读取区块链中数据
 */
func (this *MainController) GetValue() {
    // 获取请求中的key信息
    key := this.GetString("key")
    // 没有key无法进行查询
    if key == "" {
        beego.Error("Request parameter key can't be empty")
        handleResponse(this,400,[]byte("Request parameter key can't be empty"))
        return
    }
    beego.Info("key:" + key)

    // 使用fabric-sdk-go将调用的chaincode方法和参数传递到服务器,我们需要在models中完成与区块链服务器的交换,大家可以把这个过程看做将数据库中的数据读写
    var (
        channelID   = beego.AppConfig.String("channel_id")
        chainCodeId = beego.AppConfig.String("chaincode_id")
        userId      = beego.AppConfig.String("use_id")
    )
    ccs, err := models.Initialize(channelID, chainCodeId, userId)
    if err != nil {
        handleResponse(this, 500, []byte(err.Error()))
        return
    }
    var args [][]byte
    args = append(args, []byte(key))
    response, err := ccs.ChaincodeQuery("get", args)
    if err != nil {
        handleResponse(this, 500, []byte(err.Error()))
        return
    }
    // 获取结果成功后,将结果返回给前端
    handleResponse(this, 200, response)
```

```
}
```

第五步：测试数据读取

key=test01的数据已经写到区块中了，值为itcast，大家可以使用下面的链接测试一下返回内容。

```
http://localhost:8080/test?key=test01
```

向区块中写数据

操作步骤：

1、配置路由

2、在controller中，处理写数据请求

3、在models中，使用fabric-sdk-go完成写数据操作（重点）

4、在controller中，调用models完成写数据操作，并将结果返回给调用者

5、测试数据读取

第一步：配置路由

在router.go文件中填写如下内容：

```
beego.Router("/test", &controllers.MainController{},"post:SetValue")
```

第二步：处理写数据请求

在controller中编写SetValue进行数据读取请求处理，现阶段我们仅仅对参数进行校验

```
func (this *MainController) SetValue() {
    key := this.GetString("key")
    value := this.GetString("value")

    if key == "" || value == "" {
        handleResponse(this, 400, "Request parameter key(or value) can't be empty")
        return
    }
    beego.Info(key + ":" + value)
    // TODO 写操作
    var response []byte
    // 操作成功后,将结果返回给前端
    handleResponse(this, 200, response)
}
```

第三步：用fabric-sdk-go完成写数据(*)

此处，我们只需要完善ChaincodeUpdate方法即可。

> 读源码查看返回值内容

```
/*
    更新:依据方法和参数组
 */
func (this *ChainCodeSpec) ChaincodeUpdate(function string, chaincodeArgs [][]byte) (response
[]byte, err error) {
    request := apitxn.ExecuteTxRequest{ChaincodeID: this.chainCodeId, Fcn: function, Args:
chaincodeArgs}
    id, err := this.client.ExecuteTx(request)
    return []byte(id.ID),err
}
```

第四步：controllers读数据代码完善

```
func (this *MainController) SetValue() {
    key := this.GetString("key")
    value := this.GetString("value")

    if key == "" || value == "" {
        handleResponse(this, 400, "Request parameter key(or value) can't be empty")
        return
    }

    beego.Info(key + ":" + value)

    // 写操作
    var (
        channelID   = beego.AppConfig.String("channel_id")
        chainCodeId = beego.AppConfig.String("chaincode_id")
        userId      = beego.AppConfig.String("use_id")
    )
    ccs, err := models.Initialize(channelID, chainCodeId, userId)
    if err != nil {
        handleResponse(this, 500, err.Error())
        return
    }
    var args [][]byte
    args = append(args, []byte(key))
    args = append(args, []byte(value))
    response, err := ccs.ChaincodeQuery("set", args)
    if err != nil {
        handleResponse(this, 500, err.Error())
        return
    }
    // 操作成功后,将结果返回给前端
    handleResponse(this, 200, response)
}
```

第五步：测试数据读取

通过浏览器模拟一个post请求比较麻烦了，我们在后面完成前端开发的界面后再进行测试。

回收资源

使用client读写操作完成后我们并没有立即的回收资源，在ChannelClient接口中含有Close方法，在后端项目的最后我们完善回收操作。

```
type ChannelClient interface {
    // Query chaincode
    Query(request QueryRequest) ([]byte, error)
    ……
    // ExecuteTx execute transaction
    ExecuteTx(request ExecuteTxRequest) (TransactionID, error)
    ……
    // Close releases channel client resources (disconnects event hub etc.)
    Close() error
}
```

在models中添加Close方法，内容如下：

```
/*
    回收资源
 */
func (this *ChainCodeSpec) Close()  {
    this.client.Close()
}
```

在读写方法中，增加Close方法。

```
ccs, err := models.Initialize(channelID, chainCodeId, userId)
if err != nil {
    handleResponse(this, 500, err.Error())
    return
}
defer ccs.Close()
```

> 注意：需要在判断完err后调用

前端项目

前端界面(写数据)

利用WebStorm快速开发一个前端Vue界面。使用vue-resource发送请求到后台服务。

资源内容：

vue.js

vue-resource.js

界面效果：

创建一个空项目，新建TestSet.html，导入vue.js和vue-resource.js文件。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>写数据到区块</title>
    <script src="vue.js"></script>
    <script src="vue-resource.js"></script>
</head>
<body>
<div id="app">
    <table>
        <tr>
            <td>key:</td>
            <td><input type="text" v-model="key"></td>
        </tr>
        <tr>
            <td>value:</td>
            <td><input type="text" v-model="value"></td>
        </tr>
    </table>
    <button v-on:click="set">写数据</button>
</div>
</body>
<script>
    var vm = new Vue({
        el: '#app',
        data: {
```

```
            key: "test01",
            value:"itcast"
        },
        methods:{
            set:function () {
                var url="http://localhost:8080/test"
                var params={"key":this.key,"value":this.value}
                this.$http.post(
                    url,
                    params,
                    {emulateJSON: true}// 跨域访问
                ).then(
                    function (response) {
                        console.log(response)
                    },
                    function (response) {
                        console.log(response)
                    }
                )
            }
        }
    });
</script>
</html>
```

前端界面(读数据)

界面：

新建TestSet.html，导入vue.js和vue-resource.js文件

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>从区块中读数据</title>
    <script src="vue.js"></script>
    <script src="vue-resource.js"></script>
</head>
<body>
<div id="app">

    <table>
        <tr>
            <td>key:</td>
            <td><input type="text" v-model="key"></td>
        </tr>
        <tr>
            <td>value:</td>
            <td>{{value}}</td>
        </tr>
    </table>
    <button v-on:click="get">读数据</button>

</div>
</body>
<script>
    var vm = new Vue({
        el: '#app',
        data: {
            key: "test01",
            value:""
        },
        methods:{
            get:function () {
                var url="http://localhost:8080/test"
                var params={"key":this.key}
                this.$http.get(
                    url,
                    {params:params},// 注意get方式的params结构与post方式不同
                    {emulateJSON: true}// 跨域访问
                ).then(
                    function (response) {
                        this.value=response
                    },
                    function (response) {
                        console.log(response)
                    }
                )
            }
        }

    });
</script>
```

```
</html>
```

> 试一试：如果需要读取的数据不存在时，查看一下返回结果。