TERA 平台上 DApps 的开发

官网：https://terafoundation.org
源码：https://sourceforge.net/p/tera/code/ci/master/tree/
创世帖：https://bitcointalk.org/index.php?topic=4573801.0

在 TERA 平台上开发和构建去中心化应用程序不需要软件开发人员了解网络结构及其协议。
去中心化应用程序开发（DApps）基于两个组件：
1.智能合约（基于 JavaScript 的服务端）
2.前端采用对话框界面（带有 JS，CSS 和其他标准 Web 技术的常规 HTML 文件）

目录

DApps 服务端 - 智能合约

模块

模块要求

智能合约的文本（代码）必须用 JavaScript 编写（规范如下所述），并且必须只包含函数。 要指定函数的外部调用属性，请在函数的形式参数后面使用 public 修饰符。

Example:

```
function ConfirmReceipt(Params) public
{
}
```

为了与常见的 JS 编译器和编辑器兼容（例如，通过格式验证），允许在函数定义之前将该修饰符用纯 JS 语法指定为字符串常量“public”。

Example:

```
"public"
function ConfirmReceipt(Params)
{
}
```

可以通过三种方式调用公共函数：

1.来自支付交易“MONEY_TRANSFER”。验证发件人已签名（context.FromNum - 包含发件人账户的有效值）。

2.来自区块链“SMART_RUN”的特殊交易。 没有发件人的签名（context.FromNum - 包含 undefined）。

3.直接作为静态函数来自内存镜像 - 也就是说，不改变区块链的数据（所有 context 字段都包含 undefined）。

每笔交易(transaction)受限于每次调用的代码执行行数（虚拟机执行 ticks）。 每个系统函数在系统中都有自己的价格。 资源最密集的是状态记录，硬币转账。

预定义函数

预定义函数不能包含 public 修饰符。
如果要取消预定义函数的执行，会抛出异常， 将会导致整个交易取消。
交易（transaction）一词与数据库中的概念一致，暗示了操作的原子性。

函数

OnGet()
收款时调用（收款后，但在交易完成之前）。
Example:
`function OnGet()`
`{`
`  if(context.Description==="MONEY")`
`    Send(context.FromNum,1000,"YES, MONEY");`
`}`

onSend()
付款时调用（付款后，但在交易完成之前）。
Example:
`function OnSend()`
`{`
`  if(context.ToNum !==300 && context.Account.Value.SumCOIN<1000)`
`    throw "There's not much money left in the account (<1000 TERA)";`
`}`

onCreate()
当智能合约被创建时调用。
Example:
`function OnCreate()`
`{`
`  WriteState({StartValue:1000000},context.Smart.Account);`
`}`

OnDeleteSmart()
当删除绑定在当前账户的智能合约（例如，重新绑定到另外一个智能合约）时调用。
Example:
`function OnDeleteSmart()`
`{`
`  Event("UnLink account:"+context.Account.Num);`
`}`

OnSetSmart()
当账户绑定到智能合约时调用。
Example:
`function OnSetSmart()`
`{`
`  Event("Link account:"+context.Account.Num);`
`}`

JavaScript 编程语言

编程语言是标准的 JavaScript。为确保代码执行的安全性，语言中使用了特殊限制：

- 代码执行时间受最大 Tick 数限制（35000）；
- 不允许使用正则表达式；
- 不允许 try-catch（以确保不变性）；
- 标准对象中：

1）禁止以下方法：

数组（**Array**）：concat，toString

字符串（**String**）：localeCompare，match，repeat，search，padStart，padEnd

2）允许以下常数（constants）：

true，false，undefined，Infinity，NaN，null

3）允许以下函数和变量：

标准函数：isFinite，isNaN，parseFloat，parseInt，String，Number，Boolean

JSON - JS 的标准序列化对象

Math - 用于访问数学函数的标准 JS 对象，特例：函数 random()始终返回 0

this - 标准对象 - 函数执行的上下文

arguments - 标准对象 - 函数调用的变量数组

context - 包含智能合约执行的 context 定义为：

```
{
  BlockNum, //number - 当前块的编号
  BlockHash, //32 字节数组 - 当前块的哈希
  BlockAddrHash, //32 字节数组 - 在该数组中含矿工的 id 和公告确定的块的哈希值
  TrNum, //number – 当前交易的编号
  Account, //object - 在当前被调用的智能合约 context 中的账户（参见下面的描述）
  Smart, //object - 当前智能合约（见下面的说明）
  FromNum, //number - 付款人的账户编号
  ToNum, //number - 收款人的账户编号
  Description, //string - 付款说明
  Value:{SumCOIN /*硬币整数部分*/,SumCENT/*硬币小数部分*/}, //支付金额
};
```

对象字段介绍：

Object **Account**:

```
{
 Num, //number – 账户编号
 Currency, //number – 当前币种编号
 PubKey, //33 字节数组 - 公钥
 Name, //string - 账户名称
 BlockNumCreate, // number - 创建账户时的块编号
 Adviser, // number - 顾问的账户编号
 Value:{
        SumCOIN, /*硬币整数部分*/
        SumCENT, /*硬币小数部分*/
        OperationID, /*number - 付款次数*/
        Smart, /*number - 智能合约编号*/
        },
}
```

Object **Smart**:
{
Num, //number – 账户编号
Version, //number – 智能合约版本编号
TokenGenerate, //number -表示智能合约在系统中生成新币种
BlockNum, //number -创建智能合约的块编号
TrNum, //number -创建智能合约的交易编号
Owner, //number -创建智能合约的交易的账户编号（默认，不提供任何其他权限）
Account, //number – 智能合约账户的基础账户编号（base account）
AccountLength, //number - 从基数开始的账户数量（1-50）
ISIN, //string 12 字符串（国际证券识别号码）
ShortName, //string 5 字符串，代币的短名称（用于显示）
Name, //string – 智能合约的名称
Description, //string – 智能合约的描述
Code, //string – Javascript 代码
HTML, //string – HTML 中的接口代码
StateFormat, //string - 状态字段的格式
}

系统函数

函数	Tick
SetValue*(Num,Sum)* 设置 SumStruct 的值（只能从生成代币的智能合约中访问）。 Sum 是一个数值结构：{SumCOIN,SumCENT} *Example:* *SetValue(100,COIN_FROM_FLOAT(1000000));*	3000
Send*(ToNum,CoinSum,Description)* 从当前账户转账到指定账户 *ToNum*。	3000
Move*(FromNum, ToNum,CoinSum,Description)* 从与当前智能合约绑定的账户 *FromNum* 转账到指定账户 *ToNum*。	3000
ReadState*(Num)* 按账户返回状态对象。 创建智能合约时，状态对象的格式在特殊字段 StateFormat 中设置。 在这种情况下，状态对象包含预定义属性 *Num* - 包含账户编号。	1000
ReadAccount*(Num)* 通过账户编号返回账户对象。	900
WriteState*(Obj,Num)* 将状态对象写入指定的账户（仅能从绑定到该账户的智能合约访问）。 如果未指定 *Num* 参数，则该值取自 *Obj.Num*。	3000
Event*(Description)* 将字符串消息发送到当前的钱包 API。	50

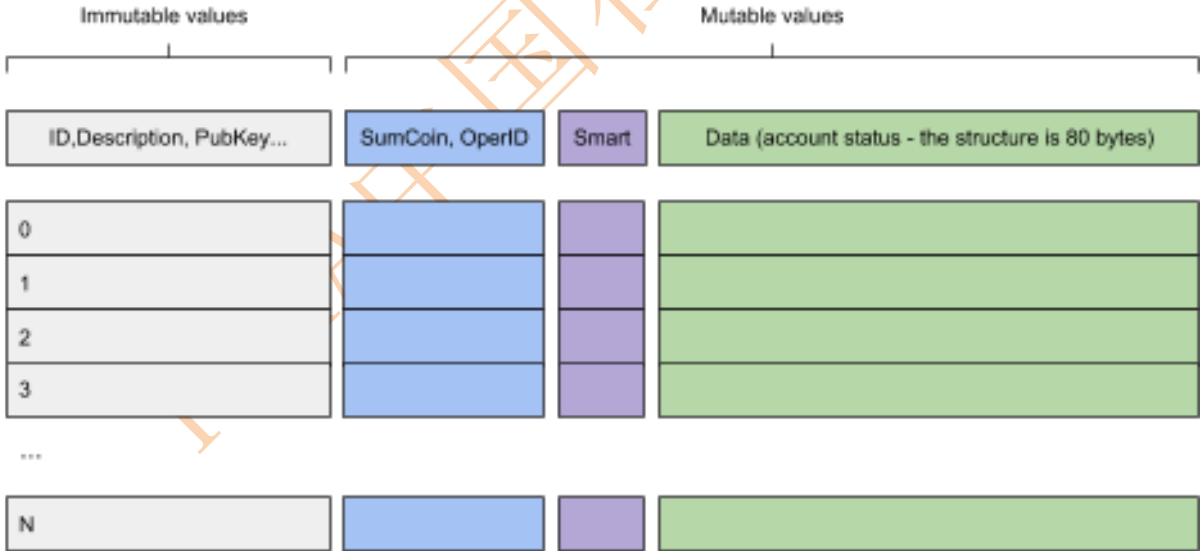
GetMaxAccount*()* 获取最大账户编号。	20
FLOAT_FROM_COIN*(Coin)* 从对象{**SumCOIN,SumCENT**}中返回双精度硬币数量。 *Example:* *Event(**"Get from "+context**.FromNum+**" count="**+FLOAT_FROM_COIN(**context**.Value));*	5
COIN_FROM_FLOAT*(Sum)* 从双精度数值返回对象 {**SumCOIN,SumCENT**}。	20
ADD*(Coin,Value2)* 向 Coin 上增加参数数值，所有参数必须是对象结构{**SumCOIN,SumCENT**}。 返回：**true**	5
SUB*(Coin,Value2)* 从 Coin 中减去参数数值，所有参数必须是对象结构{**SumCOIN,SumCENT**}。 如果成功（获得非零数值）返回：**true**，否则返回：**false。**	5
ISZERO*(Coin)* 检查 Coin 值严格等于 0，所有参数必须是对象结构{**SumCOIN,SumCENT**}。 如果值严格等于 0 返回：**true，** 否则返回：**false。**	5
GetHexFromArr*(Arr)* 从字节数组返回一个 16 进制字符串。	20
GetArrFromHex*(Str)* 从 16 进制字符串返回一个字节数组。	20
sha*(Val)* 返回 SHA3 哈希数组（参数为数组或字符串）。	1000
require(SmartNum) 返回可以调用公共函数的智能合约模块。 将智能合约当作代码库。在库函数中调用 context 变量与当前交易 context 匹配。 *Example:* *var lib=require(21);* *lib.DeleteItem(Item);*	2000
parseUint(Value) 解析并确保返回无符号整数。 所有无效值都设置为零。 *Example:* ***var** AddNum=parseUint(Params.**AddNum**);*	10

状态对象
写入或读取状态对象时字段格式由字符串字段 StateFormat 确定，该字段是在创建智能合约时设置的。它应该包含一个 JSON 类型的字符串，含“key:type”字段格式。允许以下名称作为类型：
{} – object 对象
[] – array 数组
uint - 无符号整数，长度 6 字节
uint32 - 无符号整数，长度为 4 个字节
uint16 - 无符号整数，长度为 2 个字节
byte - 无符号整数，长度为 1 个字节
double - 双精度数，长度为 8 个字节
str - 可变长度字符串
strxx - 固定长度为 xx 长的字符串
arrxx - 固定长度为 xx 长的字节数组

每个状态对象都有一个保留的预定义 Num 属性，账户编号被写入其中。
Example:
{Name:str10, Value:uint, PubKey:arr33}
{Type:byte,Account:uint,SumCOIN:uint,SumCENT:uint32, arr:[uint]}

数据存储

主要思想是将相关和可变信息存储在智能合约链接到的账户的数据结构中。 并且及时存储永久和不变的信息为区块链的一部分。 在执行智能合约期间，只能访问第一类信息 - 账户状态。
每个账户都具有以下结构：



创建账户时设置常量参数：

- 账户编号（ID），自动分配，在发送硬币时使用；
- 账户名称，方便付款（发送时，指定账户的同时显示其名称）；
- 顾问账户编号，在推荐计划进行期间获得采矿的双重奖励， 释放 30M TERA 后终止；
- 币种 - 管理代币发行的智能合约。 注意：只能在相同币种的账户间发送硬币。

可量化区域：

- SumCoin - 包含硬币数量（硬币整数部分为 6 个字节，小部分为 4 个字节）；
- OperationID - 付款次数（每笔交易，包括零金额，增加 1）；

- Смарт (Smart) - 管理账户的智能合约（收款和转账的操作可由代码控制）；
- Data - 智能合约状态对象的存储（80 字节长的任意数据）

在账户的币种账户字段中注册的智能合约可以任意更改区域：SumCoin，OperationID；
在账户的智能合约字段中指定的智能合约可以任意更改区域：Data。

创建智能合约时，可以创建 1 到 50 个账户。 第一个账户的引用被记录在智能合约中。 除了这些账户，用户还可以创建指定智能合约绑定的账户，这使得智能合约存储的信息量不受限制。
如果智能合约允许，用户可以更改与账户绑定的智能合约，称为事件：OnDeleteSmart。

智能合约样例

一个简单的智能合约，如果在支付描述中写入 MONEY 字样，则向发送方发送 1000 个硬币。

```
function OnGet()
{
  if(context.Description==="MONEY")
    Send(context.FromNum,1000,"YES, MONEY");
}
```

一个最小余额限制的例子：

```
function OnSend()
{
  if(context.Account.Value.SumCOIN<1000)
    throw "ERROR MIN REST = 1000";
}
```

如何实现加密猫：
在加密猫或类似的游戏中，可以这样做：用户创建一个新的账户来存储他的动物，绑定一个智能合约。用户只能向具有相同智能合约的账户出售或购买，因为在这种情况下，智能合约可以访问两个账户中的状态对象（Data 区域）。

一些交易费用

10 TERA - 在限制性队列外创建新账户；
100 TERA - 创建智能合约 DApps；
10 000 TERA - 创建智能代币（即自有货币）；
付款进入账户 0，即返回存钱罐用于采矿。

DApps 前端 – HTML

该接口是一个常规 HTML 文件，具有额外的 API 调用服务端函数，包括由智能合约定义的函数。
如果使用 Web 应用程序的术语，可以说智能合约是后端，该接口是前端。
与智能合约不同，DApps 前端 HTML 可以读取区块链中的所有信息 - 不仅可以读取账户状态，还可以读取区块中的数据（交易内容）。

系统函数

函数
SetStorage(Key,Value) 在浏览器的本地存储中设置智能合约值。 相当于普通 Web 编程中的表达式： **localStorage**.setItem(**Key**,***JSON***.stringify(**Value**)); *Example:* *SetStorage*(**"Name"**,*$*(**"idName"**).value);
GetStorage(Key,F) 从浏览器的本地存储中获取智能合约的值， F - 回调函数定义为：F(Key,Value){} *Example:* *GetStorage(**"Name"**,**function**(Key,Value)* *{* *$(**"idName"**).**value**=Value* *});*
SetCommon(Key,Value) 在浏览器的本地存储中设置值。 键值区域与其他智能合约共享。 *Example:* *SetCommon*(**"Amount"**,*$*(**"idAmount"**).value);
GetCommon(Key,F) 从浏览器的本地存储中获取其他智能合约共享的值， F - 回调函数定义为：F(Key,Value){} *Example:* *GetCommon(**"Amount"**,**function**(Key,Value)* *{* *$(**"idAmount"**).**value**=Value* *});*
OpenRefFile(Path) 从区块链打开 HTML 链接。链接的格式：/file/<区块编号>/<交易编号> *Example:* ***var** refs=**document**.getElementsByTagName(**"A"**)* ***for** (**var** i=0, L=refs.**length**; i<L; i++)* *{* * **if**(refs[i].**href**.indexOf(**"/file/"**)>=0)* * {* *   refs[i].onclick=**function**()*

```
    {
      OpenRefFile(this.href);
    }
  }
}
```

SendPay(Data)
将支付信息发送到钱包以进一步确认并由用户发送到区块链。
Data 结构:

```
{
  From:number – 付款账户编号（可选）,
  To:number – 收款账号编号,
  Amount:number – 支付数量,
  name:string – 项目头信息,
  Description:string – 支付描述
}
```

Example:

```
var Data=
  {
    From:10,
    To:15,
    Amount:90,
    name:"GAME",
    Description:"For game"
  };
SendPay(Data);
```

Call(Account,MethodName,Params, F)
在区块链交易之外调用智能合约函数（静态）。
Account - 账户号码（如果为 0，智能合约的基本账户）；
MethodName - 函数的名称；
Params – 含传递参数的对象；
F - 回调函数定义为：F(*Err,RetValue*){}

Example:

```
Call(60, "Run1",{A:100}, function (Err,RetValue)
{
  SetStatus("Err="+Err+"  RetValue="+JSON.stringify(RetValue))
});
```

SendCall(Account,MethodName,Params, FromNum)
向区块链上的智能合约发送函数调用。
Account - 账户号码（如果为 0，智能合约的基本账户）；
MethodName - 函数的名称;
Params – 含传递参数的对象;
FromNum - 调用函数的交易发送方的账户编号获取方法：*context.FromNum*。 智能合约必须具有从此账户发送的权限，如果未设置参数，则交易将在没有签名的情况下发送。

GetInfo(F)
获取最新的区块链信息。

F - 回调函数定义为：F(***Err***,***Data***){}
Data:
{
Smart:object – 当前智能合约，
Account:object – 智能合约的基础账号，
BlockNumDB:number – 写入数据库的最新区块编号，
CurBlockNum:number – 当前区块高度，
MaxAccID:number – 最大账号编号，
MaxDAppsID:number – 最大 DApps 编号
}
Example:

```
GetInfo(function (Err,Data)
{
 if(Err)
    return;
 $("idInfo").innerText="Info: "+JSON.stringify(Data);
});
```

GetWalletAccounts(F)
获取链接到当前智能合约的账户列表作为对象数组。
F - 回调函数定义为：F(***Err***,***Arr***){}
Example:

```
GetWalletAccounts(function (Err,arr)
{
  if(Err)
    return;
  var Str="";
  for(var i=0;i<arr.length;i++)
    Str+=arr[i].Name+" ";
  $("idInfo2").innerText="Accounts:"+Str;
});
```

GetAccountList(Params,F)
获取一个由账户列表构成的数组对象。
Params 包含参数:
{
StartNum:number – 起始的账户编号,
CountNum:number – 项目数量
}
F – 回调函数定义为：F(***Err***,***Arr***){}
Example:

```
GetAccountList({StartNum:19,CountNum:2},function (Err,Arr)
{
  if(Err)
  {
    $("idInfo").innerText="Error AccountList";
    return;
  }
  if(Arr.length)
  {
```

```
    $("idInfo").innerHTML="SmartState="+JSON.stringify(Arr[0].SmartState);
  }
  else
  {
    $("idInfo").innerHTML="Count items="+Arr.length;
  }
});
```

GetSmartList(Params,F)
获取一个由智能合约列表构成的数组对象。
Params 包含参数:

```
{
    StartNum:number – 起始的智能合约编号,
    CountNum:number – 项目数量
}
```

F – 回调函数定义为：F(*Err*,*Arr*){}
Example:

```
GetSmartList({StartNum:0,CountNum:20},function (Err,Arr)
{
  if(Err)
  {
    $("idInfo").innerText="Error SmartList";
    return;
  }
  $("idInfo").innerText="Count items="+Arr.length;
})
```

GetBlockList(Params,F)
获取一个由区块列表构成的数组对象。
Params 包含参数:

```
{
    StartNum:number – 起始的区块编号,
    CountNum:number – 项目数量
}
```

F – 回调函数定义为：F(*Err*,*Arr*){}
Example:

```
GetBlockList({StartNum:0,CountNum:30},function (Err,Arr)
{
  if(Err)
  {
    $("idInfo").innerText="Error BlockList";
    return;
  }
  $("idInfo").innerText="Count items="+Arr.length;
})
```

GetTransactionList(Params,F)
获取一个由交易列表构成的数组对象。
Params 包含参数:

```
{
```

BlockNum:number – 区块编号,
StartNum:number – 起始的交易编号,
CountNum:number – 项目数量
}
F – 回调函数定义为：F(*Err*,*Arr*){}

Example:

```
GetTransactionList({BlockNum:74585,StartNum:0,CountNum:30},function (Err,Arr)
{
  if(Err)
  {
    $("idInfo").innerText="Error TransactionList";
    return;
  }
  if(Arr.length)
  {
    $("idInfo").innerHTML="Script="+Arr[0].Script;
  }
  else
  {
    $("idInfo").innerText="Count items="+Arr.length;
  }
})
```

window.OnEvent(Data)
当通过 Event(Description)函数从代码发送的智能合约接收到事件时，将调用此函数。

Example:

```
window.OnEvent=function(Data)
{
  $("idInfo").innerHTML="OnEvent: Block="+Data.BlockNum+"/"+Data.TrNum+"
Description="+JSON.stringify(Data.Description);
}
```

DApps 特色:

如果智能合约具有以下状态字段，也包括其它：
HTMLBlock:uint，HTMLTr:uint16，并且基本账户具有非零 HTMLBlock 值，HTML 文件从区块链加载，即使用地址：/file/HTMLBlock/HTMLTr
这允许你动态更改 DApps 前端页面。 在智能合约中通过这样提供一个新版本的控制界面：

```
function OnGet()
{
  var BaseNum=context.Smart.Account;
  if(context.FromNum===context.Smart.Owner && context.Account.Num===BaseNum)
  {
    var CurItem=ReadState(BaseNum);
    var Data=JSON.parse(context.Description);
    CurItem.HTMLBlock=Data.HTMLBlock;
    CurItem.HTMLTr=Data.HTMLTr;
    WriteState(CurItem);
    Event("Set new HTML  to: "+CurItem.HTMLBlock+"/"+CurItem.HTMLTr);
    return;
  }
}
```

DApps 示例

创建简单智能合约的示例

让我们创建一个示例，向每个向其账户发送任何金额的人（他所绑定的账户）发送 10 TERA。我们将为服务端编写智能合约的代码：

```
function OnGet()
{
  Send(context.FromNum,10,"YES, MONEY");
}
```

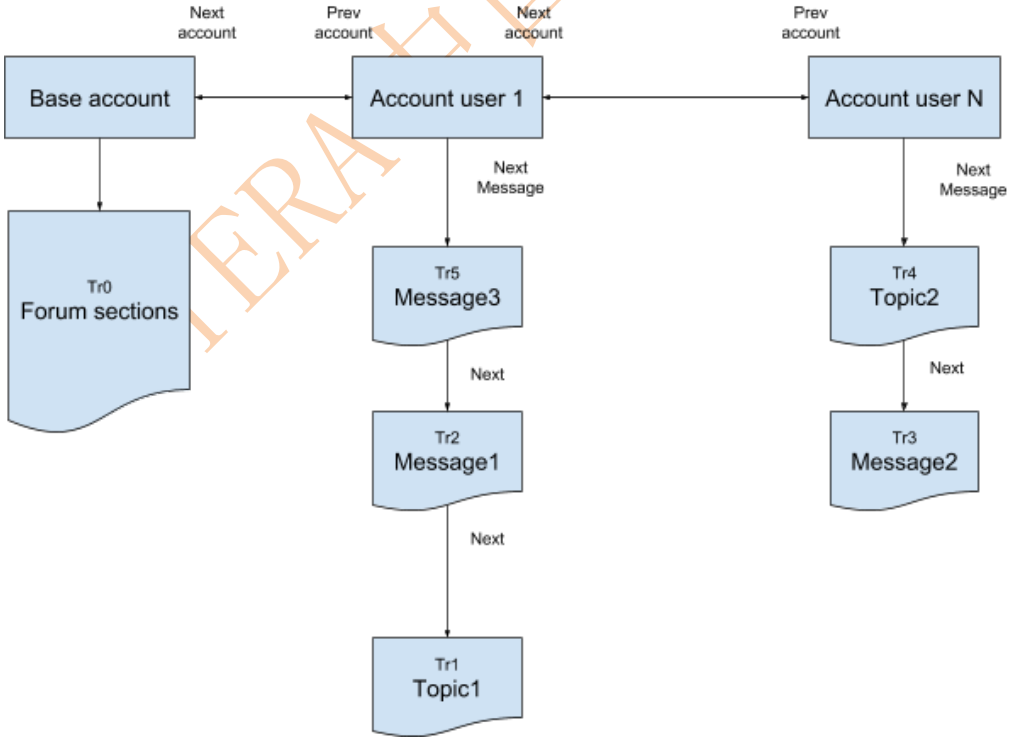
当你点击按钮时，用户可以看到这个应用程序是什么以及它做了什么，在客户端部分我们将编写这样的 HTML 代码：

```
<DIV>
  Send any payment to Base Account<BR>And get 10 TERA*
  <BR>*This is true if the account has enough funds
</DIV>
```

如果我从也附有此智能合约的账户向此智能合约发送付款会怎样？
收到付款后，一个账户会将硬币发送给另一个账户，然后另一个账户将硬币发送回发件人，发件人再次发起发送，依此类推，直到程序执行限制（ticks）结束。 交易将被取消，付款将不会通过，并会弹出错误：停止执行代码。 ticks 的限制结束了。

去中心化论坛

论坛（链接到智能合约）的用户账户和包含消息内容（主题）的交易之间的链接架构：



在这个例子中，我们获得：

Topic1: Message1, Message2
Topic2: Message3

以智能合约基本账户开头，所有用户账户都合并为一个双向链表。
每个新交易都记录消息或主题的内容以及前一个交易的链接（下一个）。 然后更新用户账户的状态区域（下一个消息）中的链接。 由于实际消息位于区块链的末尾，为了获得消息的全貌，您需要将区块链开头方向的交易列表展开到所需的时间段。

要获得常规信息，请执行以下程序步骤：
1.循环浏览用户列表（项目是账户状态）
2.对于每个用户，在消息列表中循环（项目是来自用户的交易）

特有的属性：
当用户将他/她的账户与智能合约断开时，他/她的所有消息都将丢失（更确切地说，在构建实际论坛信息时不会考虑它们）。

格式化的字符串结构：

```
{
  PrevNum:uint,//用户列表循环
  NextNum:uint,
  State:byte,
  InfoBlock:uint,//信息列表循环
  InfoTr:uint16,
  HTMLBlock:uint,
  HTMLTr:uint16
}
```

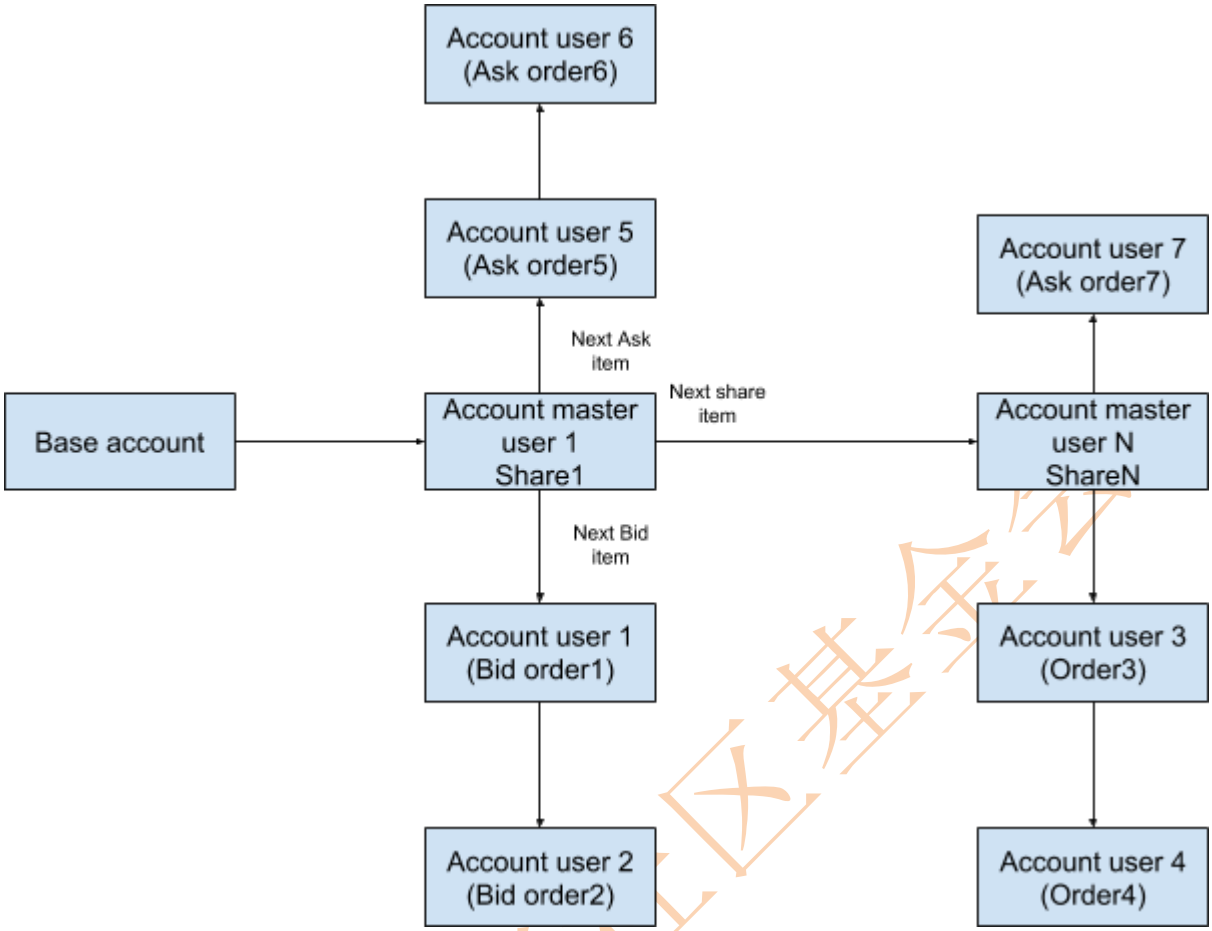
去中心化交易所

任何币种和 TERA 之间进行交换。代币是用户定义的附加货币，由与 TokenGenerate 字段的特殊智能合约管理。在本文中，我们所有代币将称为币种。
要列出交易对，您需要向交易所的智能合约提供任意账户，以将该交易对包含在交易所工具链中。 因此，不需要在此交易所中罗列其它的。
订单使用以下属性设置：
●销售额
o销售币种（由当前账户自动确定）
●收到的金额
o收据货币（由收到的账户自动确定）
●账户接收
o价格由公式自动确定：收到的金额/销售金额

订单信息将存储在用户的账户中，该账户与交易所的智能合约相关联。 因此，需要下多少个订单，就需要开多少个账户与智能合约关联。
当您尝试从订单中花钱时，交易所的智能合约将检查订单金额，而不是让您注销为执行订单而保留的金额。
从账户中删除智能合约时，如果已下订单，则会将其删除。

存储结构：

基本账户包含按币种编号排序的交易对（币种对）列表。
反过来，这样一个列表的每个元素都包含一个按价格排序的订单列表。



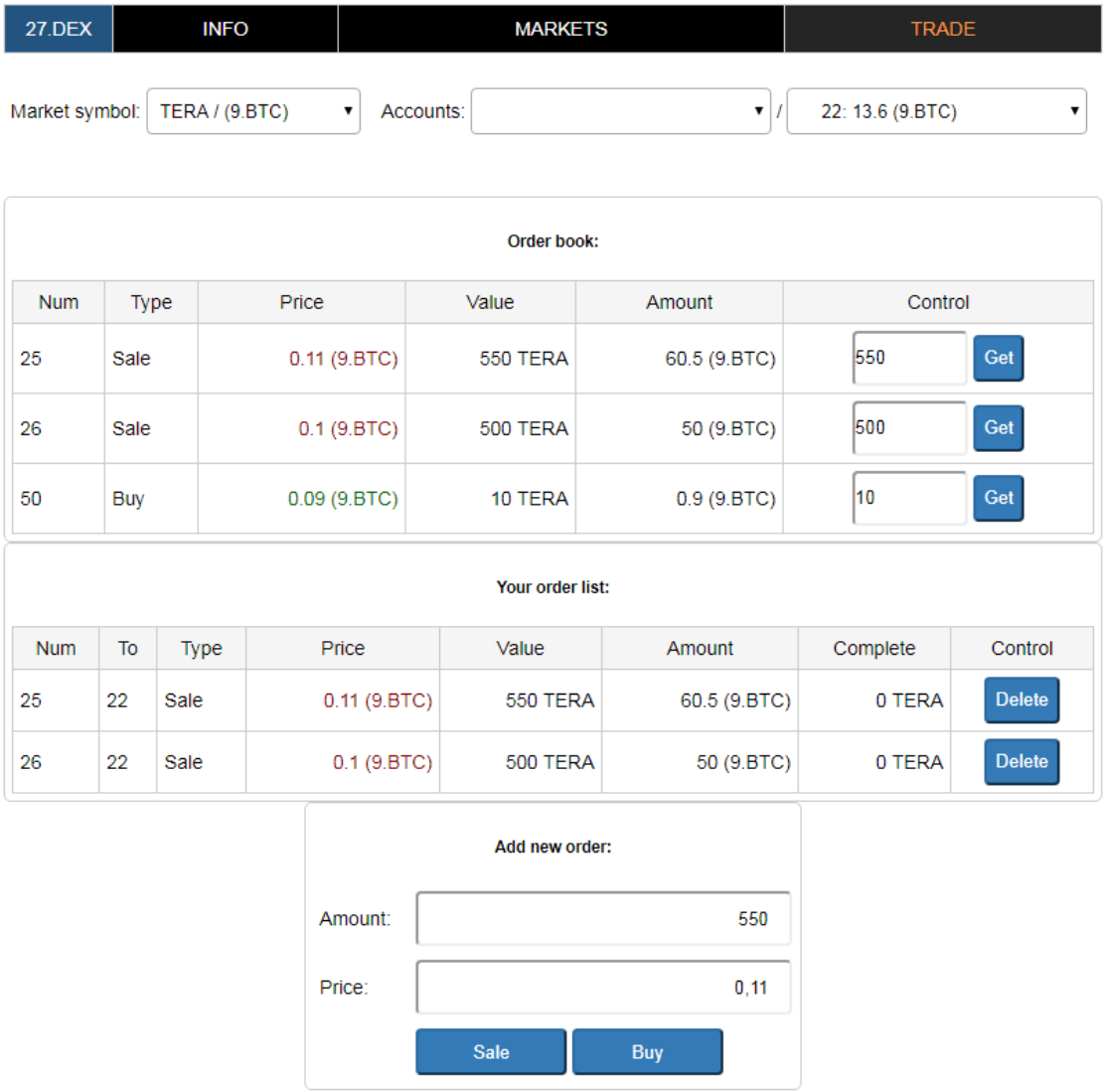
根据排序在列表的合适位置处执行插入操作。合适的位置是在 DApps 侧计算的（即，在 DB 术语的客户端），插入操作在智能合约中进行，智能合约则检查最近节点检验位置的正确性。

格式化字符串结构：

```
{
  Type:str1,
  SaleCurrency:uint32,
  GetCurrency:uint32,
  PrevNum:uint,
  ShareNextNum:uint,
  AskNextNum:uint,
  BidNextNum:uint,
  Order:
    {
      Value:{SumCOIN:uint,SumCENT:uint32},
      CompleteValue:{SumCOIN:uint,SumCENT:uint32},
      Price:double,
      GetNum:uint,
      RefCount:byte,
    },
  HTMLBlock:uint,
  HTMLTr:uint16
}
Structure length = 78 bytes
```

界面

界面由 3 部分组成（屏幕）：
1.能够在交易所添加新账户进行交易
2.增加交易对
3.交易部分



博彩游戏示例

让我们做一个没有 GUI 的程序，即只编写智能合约的代码。
游戏通过智能合约费率的累计进行，description 将指示下注的内容（只允许两个值 0 和 1）。
在游戏中没有界面还有一个优点 - 用户无需在账户上设置智能合约。
第一个过程是在账户上收到硬币时调用的预定义事件。 在事件中，我们检查该账户是智能合约的基本账户，即与智能合约同时创建的账户。 查看 description 字段，如果是下注 0 或 1 开始游戏；如果是存款 - 这是最初的资金转移到智能合约以便游戏有资金启动：

```
function OnGet()
```

```
{
  if(context.Account.Num===context.Smart.Account)
  {
    var Choose;
    if(context.Description==="Deposit")
      return;
    if(context.Description==="1")
      Choose=1;
    else
    if(context.Description==="0")
      Choose=0;
    else
      throw "Error description";

    var SumBet=FLOAT_FROM_COIN(context.Value);
    DoRound(SumBet,Choose);
  }
}
```

现在是游戏的函数。它检查指定的下注限额（至少 1 个硬币，不超过余额的 1%）。用随机值检查用户的下注选择，该随机值来自当前交易区块的 16 字节散列值。在下注时该块尚不存在，所以该值是随机的。如果获胜，将向交易发件人返还双倍金额：

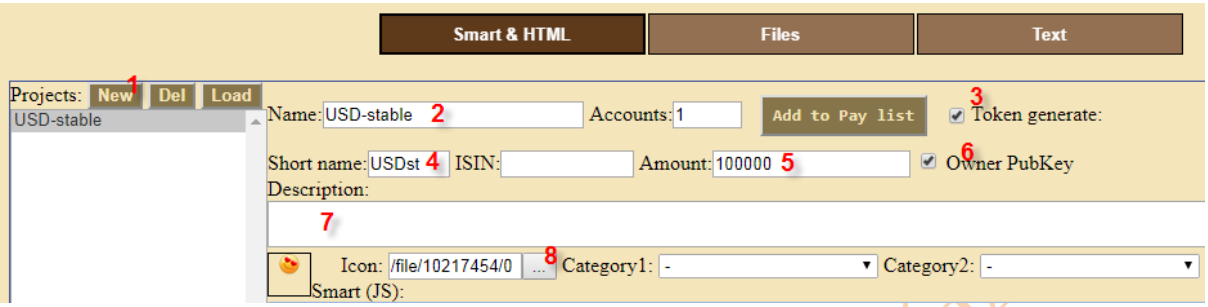
```
function DoRound(SumBet,Choose)
{
  if(SumBet<1)
    throw "The minimum amount must be 1";

  var SumRest=FLOAT_FROM_COIN(context.Account.Value);
  if(SumBet>SumRest/100)
    throw "The maximum amount should be 1% of the remaining amount";

  var Casino=context.BlockHash[16]%2;
  if(Choose===Casino)
  {
    SumBet*=2;
    Event("Win: "+SumBet);
    Move(context.Smart.Account,context.FromNum,SumBet,"Win");
  }
  else
  {
    Event("Loss: "+SumBet);
  }
}
```

发布代币

要创建自己的令牌，需要将智能合约上传到区块链。如果想将所有代币发布到自己钱包，则无需编程即可完成。进入智能合约编辑器，其中：

1. 创建一个新项目；
2. 输入名称（它将显示在区块链智能合约列表中）；
3. 代币生成的标志（币种）；

4. 代币的简称（最多 5 个字节）；
5. 初始发行规模；
6. 表示公共地址与所有者的地址（交易的发件人）匹配；
7. 描述（显示在列表中）；
8. 选择图标；
9. 单击“添加到付款”列表按钮（付款单将显示在钱包中，选择从哪个账户中扣除 10000 TERA 并支付即可）。



扩展代币的功能

对于 TERA 区块链以外的其他货币之间的支付网关，事先并不认可，因此需要添加代码允许受托人根据外部因素执行发布功能：

```
function OnGet()
{
  var BaseNum=context.Smart.Account;
  if(context.FromNum===context.Smart.Owner
        && context.Account.Num===BaseNum
        && context.Description.substr(0,1)==="{")
  {
    var Data=JSON.parse(context.Description);
    if(Data.SUM)
    {
      SetValue(BaseNum,Data.SUM);
      Send(Data.TO,Data.SUM,"MONEY");

      Event("CMD: "+context.Description);
    }
  }
}
```

在这个智能合约中，为了发行额外的硬币，您需要从智能合约的所有者的地址发送付款命令（所有者是智能合约的创建者）。 如果您在此类付款的说明中指定了一行，例如 *{“SUM”:1000000,”TO”:186555}*，那么地址 186555 将被记入 1000000 个硬币的金额。
发行的硬币可以在 TERA 的钱包中自由使用，将它们从一个账户转移到另一个账户（付款人和收款人账户的币种必须相同）。也可以设置为与去中心化交易所（DEX）上的其他币种或 TERA 交换。
我们可以说 TERA 硬币也是一个代码为 0 的智能合约，它具有系统级规定的释放规则。

冻结账户

在 Dap 17 中，智能合约在指定的时间段（年，月，日或块数）中冻结账户中的特定金额。

君子协定

一项智能合约，冻结所有借记交易，直到某人解冻它为止。
智能合约指定条件的文本描述 - 例如，交付货物的条件。并指明买方账户的编号，该账户有权解冻账户。
工作逻辑如下：
1 卖方选择一个空头金额的账户，将智能合约与之挂钩，然后启动冻结：表明他承诺要做什么以及有权解冻的账户
2 买方认为卖方规定的文字条件使他满意。
3 向它汇款并等待货物
4 当货物到达时，它进入智能合约并解冻账户。
5 如果买方不想解冻发票，并且已收货，可以通过 Escrow（智能合约的创建者）完成。

缺点：
1 有一个托管
2 不考虑买方的所有利益。

计划中的 DApps

区块链上的业务流程
传入的流程表
传出的流程表
可选 - 内容加密
想法 - 账户，内容，文本状态中的状态 - 块中的事务。
用户使用此智能合约创建账户，在设置某些状态时，会将其添加到基本账户的根目录，并对传入流程表中的收件人（处理者）可见。

有条件的担保交易

1. 创建一个价格提议（例如，100TERA 对 1USD）并发布；
2. 当有卖家时，买家家为交易创建任意的文本条件-例如，卖家向账号 xxxx 转账 0.001BTC，买家向账号 xxxx 转账 100TERA。为了解除担保，买卖双方互相理解是有必要的；
3. 买家向智能合约发送 100TERA；
4. 卖家在其他区块链完成转账（或者发送订购的土豆）；
5. 如果买家收到了土豆，点击确认收货。这种情况无需担保；
6. 如果买家没收到土豆，或者没有确认收货。这种情况担保人根据第 2 步的条款，决定将 100TERA 支付给谁。

免担保交易

在 BTT 上的文章：https://bitcointalk.org/index.php?topic=4711054.0
1. 输入交易金额；
2. 智能合约向卖家请求支付 50%的货款；
3. 智能合约向买家请求支付 150%的货款；
4. 交易可撤销时间为 1 小时；
5. 卖家向买家发货；
6. 买家收到货物后，在智能合约内确认收货；
7. 卖家收到 50%货款（保证金）+100%货款；

8. 买家收到 50%货款（保证金）；

结论

希望 TERA 区块链平台能够让开发者轻松使用 JavaScript 发布他们的应用程序。