Variable Inspection - Going Deeper with the Truffle Solidity Debugger

Note : This tutorial requires Truffle version 4.1.8 or newer.

The integrated Solidity debugger in Truffle is a powerful tool for inspecting your contracts.

When initially released, the debugger had the ability to step through Solidity code, but that was it.

But development has continued apace, and new functionality has been added to make your contract inspection more powerful. Specifically,you now have the ability to perform variable inspection on your contracts . With this, you can know exactly the state of your variables at every given point in the instruction list, giving you a much greater ability to truly inhabit the current state of your contracts, and making debugging a breeze.

In this tutorial, we're going to take a look at a simple contract and inspect it using the Truffle Solidity debugger. We'll investigate three scenarios:

1. A working contract
2. A working contract with unexpected output
3. A broken contract

A basic smart contract¶

The Fibonacci sequence is an integer sequence where each successive number in the sequence is the sum of the two previous numbers.

With the first two numbers set to 1, you can determine every number in the sequence through iteration.

The Fibonacci sequence is related to the "golden ratio", which is found in certain areas of nature, such as governing the arrangement of leaves on branches and petals on flowers.

The golden ratio in nature: flower petals. (SourceFlickr)

Generating the Fibonacci sequence with a smart contract can show off the debugger and its variable inspection without getting too bogged down in details. Let's do it.

1. Create a new project directory calledfibonacci
2. and change into it.

mkdir fibonaccicd

fibonacci 1. Create a bare truffle project:

truffle init 1. In thecontracts/ 2. directory, create a file namedFibonacci.sol 3. and add the following content:

pragma solidity

^ 0.4.22 ; contract

Fibonacci

{ uint []

fibseries;

// n = how many in the series to return

function

generateFib (uint

n)

public

{

// set 1st and 2nd entries

fibseries. push(1);

fibseries. push(1);

// generate subsequent entries

for

(uint

i

2 ;

i <

n ;

i++)

{

fibseries. push(fibseries[i- 1]

+

fibseries[i- 2]);

}

} } Let's take a look at this contract to see what's going on with it:

- First, we see the standardpragma
- declaration, which states that the contract s compatible with any 0.4.x version of Solidity newer than 0.4.22.
- The contract name is calledFibonacci
- .
- We're defining an array of integers calledfibseries
- . This will house our Fibonacci series. Note that the variable declaration is happeningoutside
- of any function, and therefore the array will be saved in storage (instead of memory), provoking a transaction to occur when the contract is run.
- The function is calledgenerateFib
- and takes a single argument, which is the number of integers in the sequence to generate.
- The next two commands add an element each to the array via the.push()
- method. As we know ourfibseries
- array is defined in storage, but size isn't known until runtime, the.push()
- method is used to add entries to (the end of) our array. This starts the sequence with the number 1 twice.
- The for loop iterates through the rest of the array (as determined by the integern
-) filling each entry with the appropriate value.
- Inside themigrations/
- directory of your project, create a file called2_deploy_contracts.js
- and populate it with the following content:

var

Fibonacci

=

artifacts . require ("Fibonacci"); module . exports

=

function

(deployer)

{

deployer . deploy (Fibonacci); }; This file allows us to deploy theFibonacci contract (shown above) to the blockchain.

1. Now launchGanache
2. . This will be the personal blockchain we'll use to deploy our contract.

Ganache

Note : You can also run this tutorial withTruffle Develop and the results will be the same.

1. In the root of your project, open yourtruffle.js
2. file and add the following content:

module . exports

=

{

networks :

{

development :

{

host :

"127.0.0.1" ,

port :

7545 ,

network_id :

"*" ,

},

}, }; This allows us to connect our project to Ganache.

1. Launch theTruffle console
2. :

truffle console You will see a prompt:

truffle(development)

We'll run all of our commands from here.

1. Compile the contract:

compile You should see the following output:

Compiling .\contracts\Fibonacci.sol... Compiling .\contracts\Migrations.sol... Writing artifacts to .\build\contracts Note : Make sure to examine the output for any errors or warnings.

1. Migrate the contract to our blockchain.

migrate You will see output that looks like this, though the addresses and transaction IDs will be different:

Using network 'development'.

Running migration: 1_initial_migration.js Replacing Migrations... ... 0xd29465deee7a5d60aed89520807432ef8a2fbbb665611882277ec8ca6fc9c622 Migrations: 0x5cc77b19b8e14e4d6074562bdbe1e13e1b793693 Saving successful migration to network... ... 0xba700ad46880ab2a9bdd961e66c9313fc541f91c439243df6ab2b97920cf2c4a Saving artifacts... Running migration: 2_deploy_contracts.js Replacing Fibonacci... ... 0xed0d1b736e948d926ee31f959b013f67d71e08f294a87e86e41ccbd8a62ce908 Fibonacci: 0xa3f4063ffbdc5cfd8ecbd99424378a5f056eb81d Saving successful migration to network... ... 0xcaeb44f5e28689c29cb3189c7b42fa1094021b4e5008f720fef2f9b867e6be23 Saving artifacts...

Interacting with the basic smart contract¶

Our contract is now on the blockchain. Ganache has automatically mined the transactions that came from the contract call and creation, as you can see by clicking the "Transactions" button in Ganache:

Ganache transactions

Now it's time to interact with the contract. First we'll check to make sure that it's working correctly.

1. In the Truffle console, enter the following command:

Fibonacci . deployed (). then (function

(instance)

{

return

instance . generateFib (10); }); Before we run the command, let's take a look at it in greater detail. Displayed in a more easily-readable manner, it reads:

Fibonacci . deployed (). then (function

(instance)

{

return

instance . generateFib (10); }); This command usesJavaScript promises . Specifically, the command says that given a deployed version of the Fibonacci contract, run an instance of that contract, and then run thegenerateFib function from that contract, passing it the argument10 .

1. Run the above command. A transaction will be created on the blockchain because our array that holds the generated Fibonacci sequence is in storage. Because of this, the output of the console

will be the transaction details, which will look similar to this:

{

tx :

'0xf47f01da1a6991f4dc168928cf4c490cb9bb57ca403b428f40a333ea65d1c41c' ,

receipt :

{

transactionHash :

'0xf47f01da1a6991f4dc168928cf4c490cb9bb57ca403b428f40a333ea65d1c41c' ,

transactionIndex :

0 ,

blockHash :

'0x88d45cb84d1f26001b0e7ae2931a56dfc0a592a1d0437fe0a37540d65f49a2ed' ,

blockNumber :

5 ,

gasUsed :

298373 ,

cumulativeGasUsed :

298373 ,

contractAddress :

null ,

logs :

[],

status :

'0x01' ,

logsBloom :

'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

},

logs :

[]

} The important part of this output for us is the transaction ID (listed as the value oftx:). Because even though we've run our function, we don't actually know what happened. Did it work correctly? Did something unexpected happen? We'll need the Truffle Solidity debugger to find out.

Note : Your transaction ID will be different from the above.

Debugging a transaction¶

You can debug a transaction in the Truffle console by typingdebug. We'll now do just that.

1. Typedebug
2. and then the value oftx:
3. found in your transaction details. For example:

debug 0xf47f01da1a6991f4dc168928cf4c490cb9bb57ca403b428f40a333ea65d1c41c Note : Your transaction ID will be different. Do not copy the above example exactly.

This will enter the debugger. You will see the following output:

Gathering transaction data... Addresses affected:

0x33b217190208f7b8d2b14d7a30ec3de7bd722ac6

-

Fibonacci

Commands: (enter)

last command entered (step next) (o)

step over,

(i)

step into,

(u)

step out,

(n)

step next(;)

step instruction,

(p)

print instruction,

(h)

print this

help,

(q)

quit(b)

toggle breakpoint,

(c)

continue

until breakpoint(+)

add watch expression (+:< expr>),

(

)

remove watch expression (-:< expr>) (?)

list existing watch expressions(v)

print variables and values,

(:)

evaluate expression -

see v Fibonacci. sol: 1 :

pragma solidity

^ 0.4.22 ; 2 : 3 :

contract

Fibonacci

{

^^^^^^^^^^^^^^^^^^ debug(development: 0xf47f01da ...)

Let's examine what's going on here.

- The debugger names the address of the contract in question that is related to the transaction, as well as the name of that contract. In our case, we're only dealing with a single contract, though if our transaction dealt with multiple contracts, all addresses would be shown.
- A full list of commands for the debugger is shown. Many of these mirror other code debuggers. We'll use a few of these throughout the tutorial, but if you ever want to see the list again, typeh
- .
- The debugger starts at the first instruction of the transaction, and shows you the relevant code for that instruction, highlighted with carets (^^^
-).
- Typen
- and
- to step next. This will move to the next instruction:

6 : 7 :

// n = how many in the series to return 8 :

function

generateFib (uint

n)

public

{

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ debug(development: 0xf47f01da ...)

1. We've seen in the[previous tutorial on debugging]
2. how you can step through the instructions to debug your contract. But here, we have an additional concern, which is that we don't actually know the outcome of our function call; we want to know what avariable
3. is set to. You can view the state of all known variables by pressingv
4. :

fibseries:

[] debug(development: 0xf47f01da ...)

Here we see one empty array. The arrayfibseries will have then numbers in the Fibonacci sequence.

1. PressEnter
2. to repeat the last command, and step next to the next instruction:

9 : 10 :

// set 1st and 2nd entries 11 :

fibseries. push(1);

^^^^^^^^ debug(development: 0xf47f01da ...)

This instruction moves on to the initial seeding of thefibseries variable.

1. Pressv
2. to see the current state of the variables:

i:

0

n:

10

fibseries:

[] Now we see three variables, ourfibseries array and two others:

- i
- is an index variable, used to determine the location of the next number in the sequence
- n
- is the integer we passed to the function (10
- in this case) indicating the number of entries in our Fibonacci sequence

Here we see thatn has been passed the value from the contract, whilei hasn't received its value from the for loop yet, since we haven't gotten to that part of the function.

1. PressEnter
2. to step next to the next instruction:

9 : 10 :

// set 1st and 2nd entries 11 :

fibseries. push(1);

^ 1. It's rather tedious to constantly pressv 2. after every state change. Thankfully, you can set certain variables to be "watched", so that they will display after every movement in the debugger. The syntax to watch a variable is+: 3. . So let's watch the variables we care about (i 4. andfibseries 5.) with the following two commands:

+: i+: fibseries After each command, the current value of the variable will display. But once done, we'll get a persistent display of the variables and their values.

1. PressEnter
2. to move to the next instruction:

9 : 10 :

// set 1st and 2nd entries 11 :

fibseries. push(1);

^^^^^^^^^^^^^^^^ : i 0 : fibseries [] Notice that after the current instruction, the watched variables are displayed automatically.

1. You can also watch expressions, not just variables. Run the following expression to make our output a little more compact.

+: {

i,

fibseries } This will output as follows:

{

i:

0 ,

fibseries:

[]

} 1. Since we have all we need in this one expression, we can unwatch the individual variables. The syntax to watch a variable is-2. .

-: i-: fibseries Now we'll only be watching the single expression comprising the two variables.

1. PressEnter
2. to move to the next instruction and see the output:

10 :

// set 1st and 2nd entries 11 :

fibseries. push(1); 12 :

fibseries. push(1);

^^^^^^^^ : {

i,

fibseries }

{

i:

0 ,

fibseries:

[

1

]

} Notice that we have now populated the first entry in the sequence.

1. Because the debugger steps through each instruction one at a time, it's going to take a long time to see results if we don't pick up our pace. Luckily, the debugger can "step over", which steps over the current line, moving to the next line, as long as it's at the same function depth. This will allow us to make progress much more quickly. So typeo
2. to step over the current instruction set. The output will be:

13 : 14 :

// generate subsequent entries 15 :

for

(uint

i

2 ;

i <

n ;

i++)

{

^ : {

i,

fibseries }

{

i:

0 ,

fibseries:

[

1 ,

1

]

} Here we see that we have populated the first two entries in the sequence.

1. SinceEnter
2. will replay the previous command, pressEnter
3. now to step over again:

14 :

// generate subsequent entries 15 :

for

(uint

i

2 ;

i <

n ;

i++)

{ 16 :

fibseries. push(fibseries[i- 1]

+

fibseries[i- 2]);

^^^^^^^^ : {

i,

fibseries }

{

i:

2 ,

fibseries:

[

1 ,

1

]

} Now that we have two entries in our sequence, we've moved into our for loop.

1. Continue pressingEnter
2. and watching the variable output. You should notice that wheni
3. gets to10
4. , the for loop ends (as the loop terminates ati < n
5.). The final array is:

[1 ,

1 ,

2 ,

3 ,

5 ,

8 ,

13 ,

21 ,

34 ,

55] It can be verified that this is the correct first ten entries in the Fibonacci sequence.

Debugging unexpected output¶

Our contract as we have created it is working as expected. That's great, but we can't always get so lucky. Sometimes a contract appears to work fine (no errors are thrown) but the output is not what you'd expect. Let's examine that here.

It's amazing what switching a plus sign for a minus sign can do. Instead of the Fibonacci sequence:

F[1]

=

1 F[2]

=

1 F[n]

=

F[n- 1]

+

F[n- 2] Let's switch the plus for a minus sign:

F[1]

=

1 F[2]

=

1 F[n]

=

F[n- 1]

-

F[n- 2] (I don't know if this series has a name, so we'll just call it the "Trufflenacci" sequence.)

Let's edit our contract and see what happens.

1. Open the Fibonacci.sol file. Edit the definition of ourfibseries[]
2. array to be of typeuint8[]
3. .

uint8 []

fibseries; Note : A uint8 is an unsigned integer with 8 bits, with a maximum value of 2^8 = 255. Compare this to auint, which is a 256 bit number with a maximum value of 2^256.

1. Edit the for loop so that the terms are subtracted instead of added:

fibseries. push(fibseries[i- 1]

-

fibseries[i- 2]); 1. Save this file. 2. Back on the console, recompile the contract.

compile --all Note : The --all flag with force recompile all of the contracts.

1. Remigrate the contract to the blockchain.

migrate --reset 1. We can now run the same command as before, which will generate a new transaction:

Fibonacci . deployed (). then (function

(instance)

{

return

instance . generateFib (10); }); As before, the output of the console will be the transaction details. Note the transaction hash (the value oftx:).

1. Typedebug
2. and then the value oftx:
3. found in your transaction details. This will enter the debugger again.

Fibonacci. sol: 1 :

pragma solidity

^ 0.4.22 ; 2 : 3 :

contract

Fibonacci

{

^^^^^^^^^^^^^^^^^^^ 1. We're curious about what ourfibseries 2. array is going to look like, so let's watch it:

+: fibseries The response will beundefined , because we haven't yet gotten to the point in the code where that array was defined.

1. Let's "step over" execution, so as to speed things up ("step next" takes much longer, as it's a more granular process). You can "step over" by running theo
2. command.

Fibonacci. sol: 6 : 7 :

// n = how many in the series to return 8 :

function

generateFib (uint

n)

public

{

^^ : fibseries [] The array is defined, but is still empty.

1. SinceEnter
2. repeats the previous command (in our case, "step over"), press Enter twice. You'll see that the first element offibseries
3. has been populated.

10 :

// set 1st and 2nd entries 11 :

fibseries. push(1); 12 :

fibseries. push(1);

^^^^^^^^^ : fibseries [

1

] 1. Keep pressing Enter until four elements have been populated:

: fibseries [

1 ,

1 ,

0 ,

255

] Something seems off here. Obviously, if you were expecting the Fibonacci series, by this point you would realize that you had veered far off course.

But the jump from0 to255 should be suspicious too. And in fact, what we're seeing is abuffer underflow , a situation where we have anunsigned integer , which can hold only positive values, being set to less than zero. Specifically, that fourth value is defined as the difference between the previous two elements, so0 - 1 = -1 . Our type isuint8[] which means that each integer in the array is defined by 8 bits, so its maximum value is 2^8 - 1 = 255. Subtract one from zero, and the value "wraps around" to its maximum value.

Assuming this was the series we wanted (and that our minus sign was correct), the way to change this is to change the definition of ourfibseries array fromuint8[] toint8[] . That would make values "signed integers" and able to accept negative values, giving you an array that would look like this instead:

[

1 ,

1 ,

0 ,

- 1

]

Debugging errors¶

Sometimes, contracts have errors in them. They may compile just fine, but when you go to use them, problems arise.

This is one of the reasons to use Ganache over a public testnet: you can easily redeploy a contract without any penalty in time or ether.

So we're going to introduce a small error, a misnumbering in our for loop that will cause it to have an invalid index.

1. Open theFibonacci.sol
2. file again.
3. Edit the for loop so that thei
4. variable starts from1

for

(uint

i

1 ;

i <

n ;

i++)

{ 1. Save this file. 2. Back on the console, recompile the contract.

compile --all 1. Remigrate the contract to the blockchain.

migrate --reset 1. Run the same command as before:

Fibonacci . deployed (). then (function

(instance)

{

return

instance . generateFib (10); }); You'll see an error, which will start with this:

Error: VM Exception while processing transaction: invalid opcode Well that's not good. Moreover, the output didn't give us a transaction ID to debug, so we're going to need to look elsewhere for it. Thankfully, Ganache can tell us the transaction information.

1. Go back in Ganache and click the "Logs" link at the top:

Click this button to open the logs

1. At the very bottom of the logs, you will see a transaction ID listed near an error sayinginvalid opcode
2. . This is the one we need. Copy this transaction ID.

Ganache logs

1. Back in the console, typedebug
2. and paste in the transaction ID. This will enter the debugger again.
3. Let's watch the same compacted expression as before, showing both the value ofi
4. and the value offibseries
5. .

+: {

i,

fibseries } 1. Continually step next. You will eventually reach a state where the debugger will error out.

14 :

// generate subsequent entries 15 :

for

(uint

i

1 ;

i <

n ;

i++)

{ 16 :

fibseries. push(fibseries[i- 1]

-

fibseries[i- 2]);

^^^^^^^^^^^^^ : {

i,

fibseries }

{

i:

1 ,

fibseries:

[

1 ,

1

]

} debug(development: 0x948da9db ...)

Transaction halted with a RUNTIME ERROR. This is

likely due to an intentional halting expression,

like assert(),

require ()

or revert().

It can also be due to out- of- gas exceptions.

Please inspect your transaction parameters and contract

code

to determine the meaning of this

error. Note that the contract is failing at the point where it tries to determine the value offibseries[i-2] But in our debugger, we know that the current value ofi is1 , soi-2 is going to be-1 (or actually2^256 - 1 , due to the buffer underflow issue we talked about above, and withi defined as auint). Since either one of those values are invalid (you can't have a negative index, and the value at index2^256 - 1 is clearly not defined), the contract halts with an error.

These are just some of the ways that you can use variable inspection to debug your contracts. We're constantly adding functionality of the debugger, so pleaseraise an issue on our Github page or ask a fellow Truffler in ourGitHub Discussions . Happy debugging!