一：

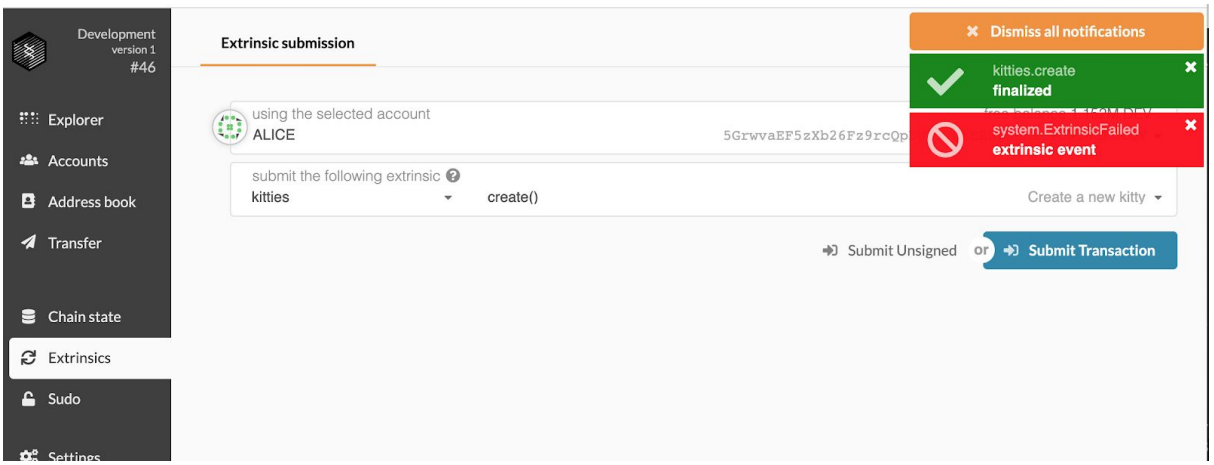
Crate里面的kitties_count有溢出的可能性，修复这个问题

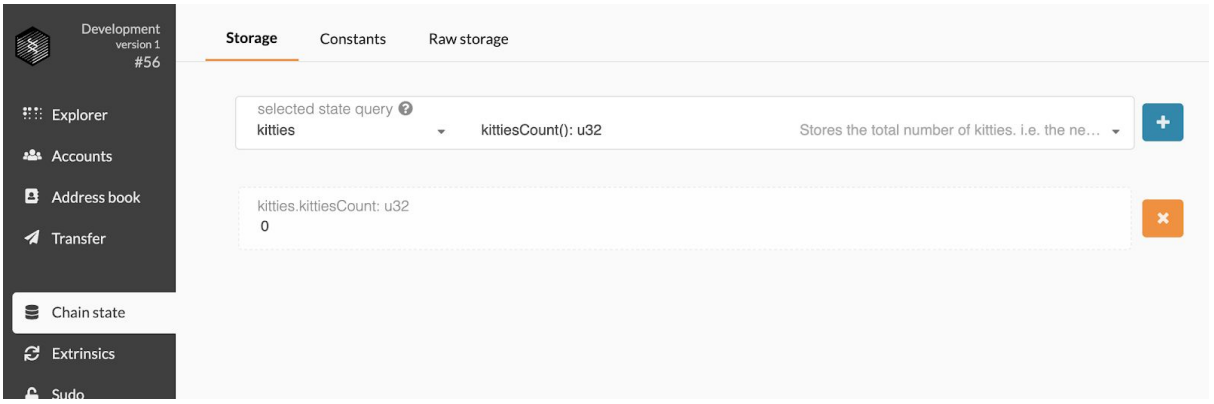
思路：利用Result 与 overflow checker（checked_add），当没有overflow时，才会执行
Kitties::insert(count, kitty);
KittiesCount::put(count + 1);
验证方式：
kitties.rs:

1. 当故意产生overflow时（当Line33注解掉， Line 34 正常执行时）：

```
21  decl_module! {
22    pub struct Module<T: Trait> for enum Call where origin:
      T::Origin {
23      /// Create a new kitty
24      pub fn create(origin)  -> Result<(), &'static str> {
25        let sender = ensure_signed(origin)?;
26        let payload = (
27          <randomness_collective_flip::Module<T> as Randomness<
            T::Hash>>::random_seed(),
28          sender,
29          <system::Module<T>>::extrinsic_index(), <system::Module<
            T>>::block_number()
30        );
31        let dna = payload.using_encoded(blake2_128);
32        let kitty = Kitty(dna);
33        //let count = Self::kitties_count();
34        let count = u32::max_value();
35        let temp = count.checked_add(1)
36                  .ok_or("Overflow adding a new kitty")?;
37        Kitties::insert(count, kitty);
38        KittiesCount::put(count + 1);
39        Ok(())
40      }
41    }
42  }
```

检测到overflow就停下，不会执行
Kitties::insert(count, kitty);
KittiesCount::put(count + 1);
Create时不会产生新的kitty

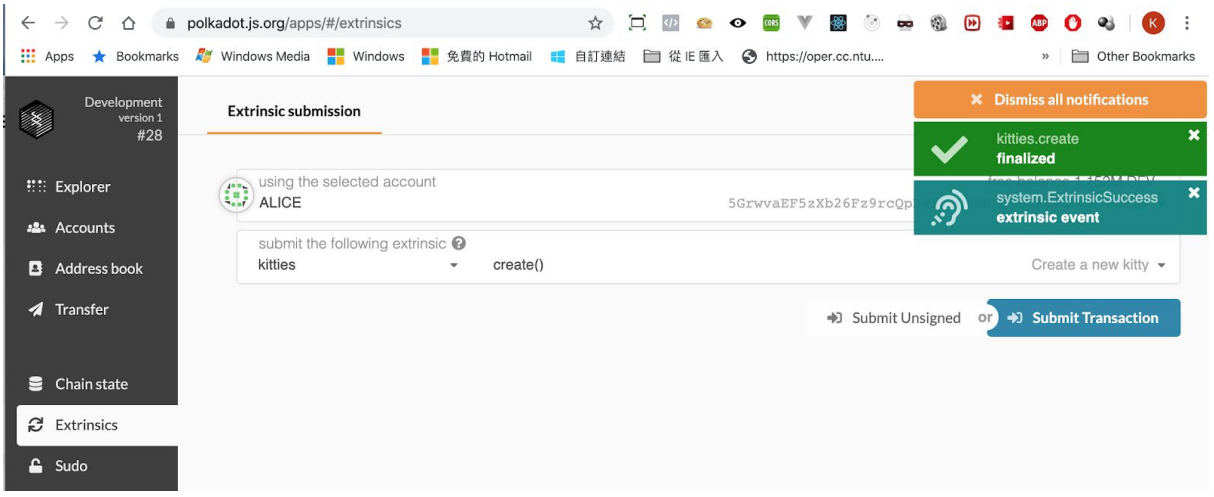


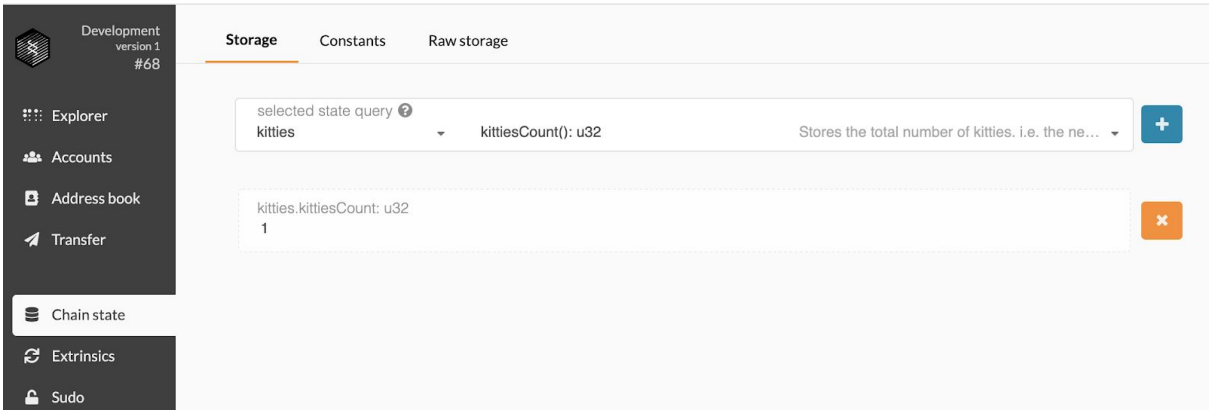


2. 当正常执行时（当Line33正常执行， Line 34 注解掉时）：

```
20
21  decl_module! {
22    pub struct Module<T: Trait> for enum Call where origin:
      T::Origin {
23      /// Create a new kitty
24      pub fn create(origin)  -> Result<(), &'static str> {
25        let sender = ensure_signed(origin)?;
26        let payload = (
27          <randomness_collective_flip::Module<T> as Randomness<
            T::Hash>>::random_seed(),
28          sender,
29          <system::Module<T>>::extrinsic_index(), <system::Module<
            T>>::block_number()
30        );
31        let dna = payload.using_encoded(blake2_128);
32        let kitty = Kitty(dna);
33        let count = Self::kitties_count();
34        //let count = u32::max_value();
35        let temp = count.checked_add(1)
36                  .ok_or("Overflow adding a new kitty")?;
37        Kitties::insert(count, kitty);
38        KittiesCount::put(count + 1);
39        Ok(())
40      }
41    }
42  }
```

行为一切正常，create时能产生新的kitty





二：

需求：

繁殖小猫

1.选择两只现有的猫作为父母

2.小猫必须继承父母的基因

3.同样的父母出生的小猫不能相同

要达成条件2与条件3的思路：小猫的dna的某一段random的由父or母而来，且同父母每次生的小猫也不一样，所以必须先产生random，由此random决定dna的某一段由父or母来的

```
fn breed_kitty(origin, kitty_id_1: T::Hash, kitty_id_2: T::Hash) -> Result{

    let sender = ensure_signed(origin)?;

    //验证交配的两只猫是否存在
    //1.选择两只现有的猫作为父母
    ensure!(<Kitties<T>>::exists(kitty_id_1), "This cat 1 does not exist");
    ensure!(<Kitties<T>>::exists(kitty_id_2), "This cat 2 does not exist");

    //获取随机Hash
    let nonce = <Nonce<T>>::get();
    let random_hash = (<system::Module<T>>::random_seed(), &sender, nonce)
        .using_encoded(<T as system::Trait>::Hashing::hash);


    let kitty_1 = Self::kitty(kitty_id_1);
    let kitty_2 = Self::kitty(kitty_id_2);

    //计算小猫的dna
   //小猫的dna的某一段random的由父or母而来，且同父母每次生的小猫也不一样，所以必须先产生random，由此random决定dna的某一段由父or母来的
    let mut final_dna = kitty_1.dna;
```

```
 //2.小猫必须继承父母的基因
 //3.同样的父母出生的小猫不能相同
   //生育过程
   for (i, (dna_2_element, r)) in
kitty_2.dna.as_ref().iter().zip(random_hash.as_ref().iter()).enumerate() {
       if r % 2 == 0 {
           final_dna.as_mut()[i] = *dna_2_element;
       }
   }

   let new_kitty = Kitty {
       id: random_hash,
       dna: final_dna,
       price: <T::Balance as As<u64>>::sa(0),
       gen: cmp::max(kitty_1.gen, kitty_2.gen) + 1,
   };

   Self::mint(sender, random_hash, new_kitty)?;

   <Nonce<T>>::mutate(|n| *n += 1);

   Ok(())
}
```

三：

额外作业

1.解释如何在链上实现（伪）随机数

2.对比不同方案的优缺点

随机性的获取包括但不仅限于：如何在智能合约中引入不可预测的随机数；如何在共识协议中安全地进行随机抽签.....等等。

伪随机数发生器不可以被直接以硬编码的方式或者是智能合约代码的方式应用在区块链系统中来获取安全的随机数。另一方面，随机数获取协议作为区块链系统的一个子协议常常与该系统下的其他协议有着强耦合的特性——例如，共识协议——也就是说，其他协议很有可能会影响随机数获取协议的安全性。

伪随机数发生器产生的随机序列的不可预测性的前提是伪随机数发生器作为一个黑盒，除了它的输出，外界无法得知其他一切信息。但是区块链上的一切都是公开透明的，包括使用的伪随机数发生器及输入到伪随机数发生器里面的种子也是一样公开透明的。在这样的情况下，所有传统的伪随机数发生器都无法在区块链的环境下产出具有不可预测性的随机数序列。

法一：Oracle法
而至于真随机数发生器，是存在将真随机数发生器的结果通过可验证的不可篡改的通道，引入区块链系统内部，这样的通道又被称作 Oracle。以太坊现在常用的随机数发生器就是通过 Oracle，引入 random.org（声称提供真随机数）提供的随机数。

优点：

方便，直接将随机数的不可预测性属性，交给外部去做

缺点：

这种方法的问题在于，所谓的“真随机数发生器”往往是中心化的，拥有这样的硬件或者软件的人或者组织拥有篡改随机数发生器结果并不让用户发觉的能力。这对于主打去中心化的区块链系统来说，是有点奇怪的。

法二：使用block hash值作为seed

直接利用区块链系统中共识过程所天然产生的随机性。e.g.使用未来某个块或者之前某个块的 Hash 值来作为种子之一生成随机数（其他的种子可以是用户的地址、用户支付的以太币数量等，但是这些是用户可控的部分，没有增加不可预测性的作用）。这种做法也常见于各种区块链博彩类游戏以及资金盘游戏当中，

优点：

直接利用hash的collision resistance特性以及hash的avalanche effect的特性

缺点：

用户有可能通过仔细选择交易时间来控制随机数向有利于自己的方向生成；即使用户无法控制，矿工也可以控制随机数的生成，并且这样的攻击成立并不需要太多算力的参与。只要最终随机数牵涉的金额足够，完全可以使用租用算力或者贿赂矿工的方式进行攻击。