TON 项目开发教程（一）：源码角度看如何在 TON Chain 上创建一个 NFT

作者：@Web3Mario

摘要：承接上一篇关于 TON 技术介绍的文章，这段时间深入研究了一下 TON 官方开发文档，感觉学习起来还是有些门槛，当前的文档内容似乎更像是一个内部开发文档，对新入门的开发者来说不太友好，因此试着以自己的学习轨迹，梳理一系列关于 TON Chain 项目开发的文章，希望可以对大家快速入门 TON DApp 开发有一些帮助。行文有误也欢迎大家指正，一起学习。

## 在 EVM 中开发 NFT 和在 TON Chain 上开发 NFT 有哪些不同

发行一个 FT 或 NFT 对于 DApp 开发者来说通常是最基本的需求。因此我也以此作为学习入口。首先让我们来了解以下在 EVM 技术栈中开发一个 NFT 和在 TON Chain 中的区别。基于 EVM 的 NFT 通常会选择继承 ERC-721 的标准。所谓 NFT，指的是不可分割的加密资产类型，且每个资产具有唯一性，即存在某些专属的特性。而 ERC-721 就是对这个类型的资产的一种通用的开发范式。让我们看一个常见的 ERC721 合约需要实现哪些函数以及记录哪些信息。下图是一个 ERC721 接口。可以看到与 FT 不同，在转账接口中需要输入的是待转账的 tokenId 而非数量。这个 tokenId 也是 NFT 资产唯一性最基本的体现，当然为了承载更多的属性，通常会为每个 tokenId 记录一个 metadata，这个 metadata 是一个外部链接，保存了该 NFT 的其他可扩展数据，例如一张 PFP 图片的链接，某些属性名称等。



对于熟悉 Solidity 或者熟悉面向对象的开发者来说，实现这样一个智能合约是件容易的事，只要定义好合约中需要的数据类型，例如一些关键的映射关系 mapping，并根据所需功能开发相应的对这些数据的修改逻辑，即可实现一个 NFT。

然而在 TON Chain 中这一切变的不太相同，造成不同的核心原因有两个：

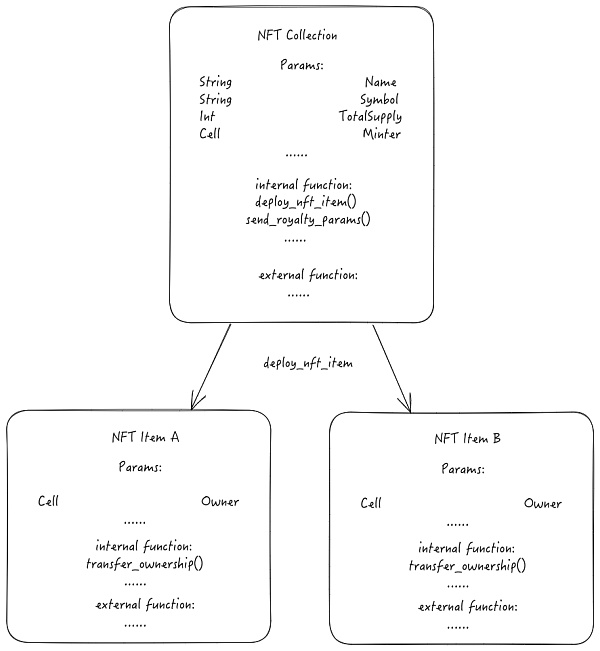
在 TON 中数据的存储是基于 Cell 实现的，而同一个账户的 Cell 通过有向无环图来实现。这样就导致需要之久化存储的数据不能无边界的增长下去，因为一个有向无环图来说，数据深度决定的查询成本，当深度无限延伸之后，有可能造成查询成本过高，从而导致合约陷入死锁问题。

为了追求高并发性能，TON 舍弃了串行执行的架构，转而采用了一个专为并行而生的开发范式，Actor 模型，来重构执行环境。这就造成了一个影响，智能合约之间只能通过发送所谓内部消息的方式异步调用，注意无论是状态修改类型或只读类型的调用都需要遵循这个原则，除此之外，也需要仔细考虑异步调用若失败，如何处理数据回滚的问题。

当然关于技术上其他不同点在上一篇文章中有过详细的论述，本篇文章希望可以聚焦在智能合约开发上，所以不展开讨论。上述两条设计原则让 TON 中智能合约开发与 EVM 产生了很大区别。在开始的论述中，我们知道一个 NFT 合约中需要定义一些映射关系，也就是 mapping，来保存 NFT 相关的数据。其中最重要的就是 owners，这个 mapping 存储了某个 tokenID 对应的 NFT 的所有者地址的映射关系，决定了 NFT 的所有权，转账就是对该所有权的修改。由于理论上这是一个可以无边界的数据结构，需要尽量避免。因此官方推荐以是否存在无边界数据结构作为分片的标准。即当有类似的数据存储需求时，通过主从合约的范式来替代，通过创建子合约的方式来管理每个 key 对应的数据。并通过主合约管理全局参数，或帮助处理子合约之间的内部信息交互。

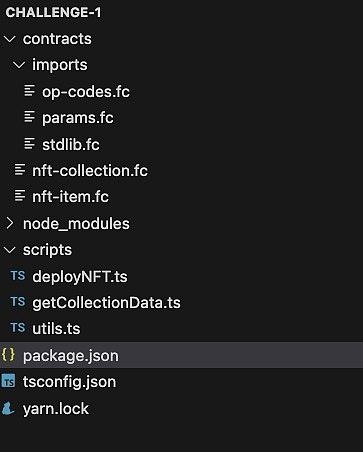
这也就意味着在 TON 中的 NFT 也需要采用类似的架构来设计，每个 NFT 都是一个独立的子合约，保存了诸如所有者地址，metadata 等专属数据，并通过一个主合约来管理全局数据，例如 NFT name，symbol，总供应量等。

在明确了架构后，接下来就需要解决核心功能的需求了，由于采用了这个主从合约的方式，因此就需要明确哪些功能由主合约承载，哪些功能由子合约承载，并且两者之间通过什么内部信息沟通，同时当出现执行错误时，如何回滚之前的数据。通常情况下，在开发复杂的大型项目之前，通过一个类图并明确彼此之间的信息流，并仔细思考内部调用失败后的回滚逻辑是必要的，当然上述 NFT 开发虽然简单，但也可以做类似验证。

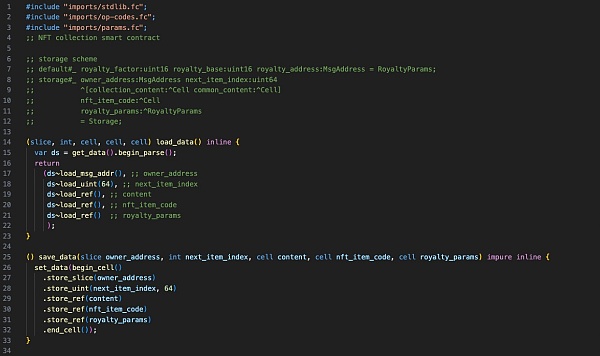


## 从源码学习开发 TON 智能合约

TON 选择了设计一种类 C 语言的、静态类型语言，名为 Func 来作为智能合约开发语言，那么接下来就让我们从源码来学习如何开发 TON 智能合约，我选择了 TON 官方文档中的 NFT 示例来进行介绍，感兴趣的小伙伴可以自行去查阅。在这个 case 中实现了一个简单的 TON NFT 例。让我们看下合约结构，共分为两个功能合约以及三个必要的库。



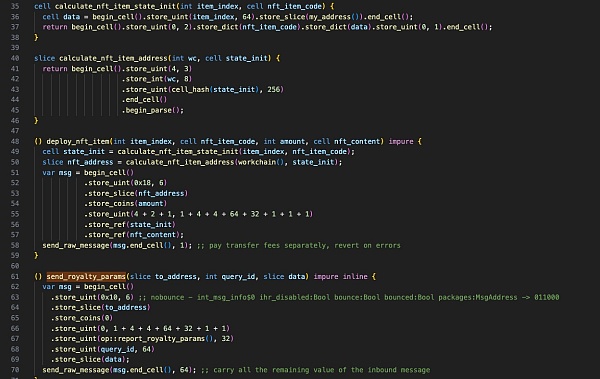
这两个主要的功能合约即按照上述的原则进行设计，首先让我们来看下主合约 nft-collection 的代码：



这引入了第一个知识点，如何在 TON 智能合约中持久化存储数据，我们知道在 Solidity 中数据的持久化存储是由 EVM 根据参数的类型自动处理的，通常情况下，智能合约的状态变量将在执行结束后根据最新值自动被持久化存储，开发者并不需要考虑这个过程。但在 Func 中情况并不如此，开发者需要自己来实现相应的处理逻辑，这个情况有点类似于 C 和 C++ 需要考虑 GC 的过程，但其他新的开发语言通常将这部分逻辑自动化处理。我们来看下代码，首先引入一些需要的库，然后看到第一个函数 load\_data 用于读取被持久化存储的数据，其逻辑为首先通过 get\_data 返回持久化合约存储 cell，注意这是由标准库 stdlib.fc 实现的，通常情况下可以将其中的一些函数视为系统函数来使用。

该函数的返回值类型为 cell，这是 TVM 中的 cell 类型。在之前的介绍中，我们已经知道 TON 区块链中的所有持久数据都存储在 cell 树中。每个 cell 最多有 1023 位任意数据和最多四个对其他 cell 的引用。cell 在基于堆栈的 TVM 中用作内存。cell 中保存的是紧编码后的数据，要想获取其中具体的明文数据，需要将 cell 转换为被称为 slice 的类型。cell 可以通过 begin\_parse 函数转换成为 slice 类型，然后可以通过从 slice 加载数据位和对其他 cell 的引用来获得 cell 中的数据。注意 15 行代码中的这种调用方法是一个 func 中的语法糖，可以直接调用第一个函数的返回值的第二个函数。并在最后按照数据持久化顺序依次加载相应的数据。注意这个过程和 solidity 不同，并不是根据 hashmap 调用，所以这个调用顺序不能乱。

在 save\_data 函数中，逻辑与之类似，只不过这是一个反向的过程，这就引入了下一个知识点，一个新的类型 builder，这是 cell 构建器的类型。数据位和对其他 cell 的引用可以存储在构建器中，然后构建器可以最终化为新 cell。首先通过标准函数 begin\_cell 创建一个 builder，并依次通过 store 相关函数存储相关函数，注意上文中调用顺序与此处存储顺序需要保持一致。最后通过 end\_cell 完成新 cell 构建，这时该 cell 被管理在内存中，最后通过最外层的 set\_data，就可以完成对该 cell 的持久化存储。



接下来让我们来看下业务相关函数，首先需要先介绍下一个知识点，如何通过合约创建一个新的合约，这在刚刚介绍的主从架构中将被经常用到。我们知道在 TON 中，智能合约之间的调用是通过发送内部消息的方式来实现的。这是通过一个名为 send\_raw\_message 来实现的，注意第一个参数是 message 编码后的 cell，第二个参数是标识位，用于表明该交易的执行方式的区别，在 TON 中设置了不同的内部消息发送的执行方式，目前有 3 种消息 Modes 和 3 种消息 Flags。可以将单一 Mode 与多个（也许没有）标志组合以获得所需的 mode。组合只是意味着将它们值的和填入即可。下面给出了 Modes 和 Flags 的描述表格:



那么让我们来看第一个主要函数，deploy\_nft\_item，顾名思义，这是一个用于创建或者说铸造新 NFT 实例的函数，经过一番操作编码一个 msg 后，通过 send\_raw\_message 发送该内部合约，并选择了 flag 1 的发送标识位，仅将编码中指定的 fee 作为本次执行的 gas fee。经过上文的介绍我们很容易意识到，这个编码规则应该是对应创建一个新的智能合约的方式。那让我们来看看具体是怎么实现的。

让我们直接看 51 行，上面两个函数是用于生成 message 所需信息的辅助函数，因此我们后面再来看，这是一个用于创建智能合约的内部消息的编码过程，中间的一些数字其实也是一些标识位，用于说明该内部消息的需求，这里要引入下一个知识点，TON 选择了一种名为 TL-B 的二进制语言来描述消息的执行方式，并且根据设置不同的标记位来实现某些特定功能的内部消息，最容易想到的两个使用场景，新合约创建和已部署合约函数调用。而 51 行的这种方式即对应了前者，创建一个新的 nft item 合约，而这主要是通过 55，56，57 三行指定的。首先 55 行这一大串数字是一系列标识位，注意 store\_uint 的第一个入参是数值，第二个是位长，其中决定了该内部消息是合约创建的是后三个标记位，以及相应二进制值位为 111（十进制即为 4+2+1），其中前两个表示该消息将附带 StateInit 数据，这个数据即为新合约的源码，以及初始化所需的数据。而后一个标记位表示内部消息附载，即希望执行相关逻辑以及需要的参数。因此你会看到在第 66 行代码并没有设置该三位数据，则表明的是一次对已部署合约的函数调用。具体的编码规则在这里查看。

那么 StateInit 的编码规则即对应了 49 行代码，通过 calculate\_nft\_item\_state\_init 计算，注意 stateinit 数据的编码也遵循了一种既定的 TL-B 编码规则，除了一些标记位之外，主要涉及到两部分新合约 code 和以及初始化 data。data 的编码顺序需要与新合约指定的持久化 cell 的存储顺序保持一致。在 36 行可以看到，初始化数据有 item\_index，即类似与 ERC721 中的 tokenId，以及由标准函数 my\_address 返回的当前合约地址，即为 collection\_address，这个数据的顺序与 nft-item 中的声明保持一致。

接下来一个知识点就是在 TON 中，所有未生成的智能合约而可以预先计算其生成后的地址，这点与 Solidity 中的 create2 函数类似，在 TON 中新地址的生成由两部分组成，workchain 标识位与 stateinit 的哈希值拼接而成，前者在之前的介绍中我们已经知道是为了相应 TON 无限分片架构而需要被指定的，当前为统一值。由标准函数 workchain 获得。后者由标准函数 cell\_hash 获得。因此回到该例子，calculate\_nft\_item\_address 即为预先计算新合约地址的函数。并将生成值在第 53 行编码到 message 中，作为该内部消息的接收地址。而 nft\_content 则对应了对被创建合约的初始化调用，具体的实现在下一篇文章中介绍。

至于 send\_royalty\_params，则需要是对某只读请求的内部消息的相应，在之前的介绍中，我们特意强调了在 TON 中内部消息不光包含可能会修改数据的操作，只读操作也需要通过这种方式实现，因此该合约即为此类操作，首先值得注意的是 67 行表示响应该请求后对请求者回调函数的标记，记下来即为返回的数据，分别是请求的 item index，以及相应的 royalty 数据。

接下来让我们引入下一个知识点，TON 中智能合约只有两个统一的入口，名为 recv\_internal 和 recv\_external，其中前者为所有内部消息的统一调用入口，后者为所有外部消息的统一调用入口，开发者需要在函数内部根据需求，采用类似 switch 的方式根据 message 指定的不同标记位来响应不同的请求，这里的标记位即为上述 67 行的回调函数标记。回到该例子，首先对 message 进行空位检查，通过后分别解析 message 中的信息，首先在 83 行解析获得 sender\_address，该参数将用于后续的权限检查，注意这里的~操作符，属于另一个语法糖。这里先不展开将。接下来解析 op 操作标记位，而后根据不同的标记位，分别处理相应请求。其中即根据某些逻辑分别调用了上述的函数。例如响应对 royalty 参数的请求，或铸造新的 nft，并自增全局 index。

接下来一个知识点对应了 108 行，想必大家通过命名也可以知道该函数的处理逻辑，与 Solidity 中的 require 函数类似，Func 中通过标准函数 throw\_unless 来抛出异常，第一个入参为错误码，第二个是检查位布尔值，若位 false 则抛出异常，并附带该错误码。而在这行中通过 equal\_slices 来判断上面解析到的 sender\_address 是否等于该合约持久化存储的 owner\_address，做权限判断。



最后为了使代码结构更清晰，开始闲了一系列帮助获取持久化信息的辅助函数，在这里就不展开介绍了，开发者可以参考这种结构来开发自己的智能合约。



TON 生态的 DApp 开发实在是件有趣的事情，与 EVM 的开发范式有很大差异，因此我会通过一系列文章来介绍如何在 TON Chain 中开发 DApp。