# Merkle树

首先，希望Merkle树能够构建文件与metadata之间的绑定关系。

树的叶子节点可能的情况：

1. 叶子节点是操作的哈希值：
   1. 文件修改操作——操作频繁，数据量大，难以和metadata绑定
   2. metadata修改操作——没有意义
2. 叶子节点是内容的哈希值：
   1. **文件内容**——树根与metadata绑定，但是文件内容修改相对metadata修改更为频繁，这就意味着每次修改文件内容，metadata中的树根都要更换
   2. metadata内容——树根与文件绑定，但是文件内容是可以被有写权限的人修改的，树根一旦被修改就无法验证

所以，使用文件内容的merkle hash tree的树根值是有可能的。

但是这会出现一部分问题。

# 以太坊

将metadata保存在区块链中，相当于metadata作为交易的数据段。

此时，我们的需求有：

metadata要便于查询，便于获取；

metadata的更新要足够安全方便（是否需要保证其他用户不能修改metadata，与签名的取舍）

主要考虑的做法为：

1. 考虑采用智能合约的形式，在metadata的存储、更新和查询时都比较容易

2. 智能合约读取交易中的metadata，将其存入区块链

3. metadata中使用文件的id与文件绑定，这里文件的id的形式还待讨论，如文件路径+作者名等。

5.另外，智能合约可以限定进行交易（也就是调用合约）的用户地址，也就是说，智能合约可以限制只有文件所有者可以进行交易修改metadata。可以在智能合约里维护一个映射表，针对不同的metadata，限制不同的作者进行交易。

# 文件标识

1. merkle tree标识文件——对文件内容hash

文件版本更新——metadata要相应更新

更新metadata里的file\_id需要调用合约函数——修改metadata的用户没有了特殊的限制

任何用户都可调用智能合约中的更新函数进行更新，此时就依靠X.509证书签名来验证metadata是否有效

问题在于： **对不同文件的定义**

2. 分辨两个文件——file\_id = 内容的merkle hash

同一作者在两个目录下创建相同内容的两个文件（policy相同）是一个文件 or 两个文件

对cipher+sig做Merkle树hash，一般不会重复

需要不停更新，所有人都可以调用更新函数对metadata的file\_id进行修改，此时只能依靠证书签名来验证，这就需要每次更新正确的file\_id时都要附带有作者签名，也就意味着作者方要进行许多额外的签名操作。（如果作者不在线？）

3. 在git中，相同内容的两个文件a.txt和b.txt，表现上是两个文件，内部是两个不同的文件指针指向同一个blob

git add时会对文件产生索引，根据路径指向具体内容的blob



4. 如果采用file\_id = file\_path( + user\_id)

当路径改变时，相当于产生了新的metadata，对应的也就是新的文件。

使用路径标识的一个问题：作者A在路径A下创建a.txt，之后删掉了，后来又在路径A下创建新的a.txt。算是两个不同文件，还是同一文件？

5. 还有一点需要确认，当用户对文件进行重命名时，Git显示的操作有两种情况：如果在文件夹中重命名，Git认为是删掉旧文件，产生一个新名字的新文件。如果使用git mv命令，操作上依旧是删除旧的产生新的，但是git会将此操作识别为rename。

新的问题：如何分辨有新文件的产生？监测git status？

如果文件内容没有进行改动：

merkle tree hash——有文件创建的操作，所以要求修改者创建新的metadata，但是git mv应该会认为是同一文件。

file\_path——路径改变是不同的文件，应创建新的metadata

6.在文件内对metadata的标识可以是交易的哈希值或区块号+交易序号，需要获取metadata时按照file\_id来搜索或根据交易标识来搜索即可。

另一个问题：获取文件时是否能确定原作者的id?怎么确定？

7. 交易会产生交易所有参与方的签名

映射表保证只有指定的account可以修改指定的metadata

data owner与account的绑定关系

file\_path——X.509证书签名的必要性？

8. 总结两种文件标识的优缺点：

merkle tree hash——按照文件内容标识文件

1. 很难出现重复现象，即使是相同明文内容的文件，只要read policy和签名的private key有一个不相同最后内容就会不同。
2. 每次文件的修改都需要更新metadata中的file\_id变量，这引发了metadata的有效性验证问题：

* 如果保证只有作者可以修改metadata(限制账户地址)，那每次file\_id的更新（文件更新）都需要作者调用合约；
* 如果是文件更新方自动发起交易更新合约，则合约中的更新函数是所有人都可以调用的（没有账户限制），此时metadata的有效性只能依赖于证书签名，但是file\_id一更新，证书签名就要相应更新，同样需要作者参与。
* 作者离线？限制了分布式存储的使用？
* merkle tree hash周期性更新？需要一个机构产生并维护周期性的hash值，否则周期的长短，包括用户获取最新版本文件但是要独自计算上个周期的hash都是比较复杂的操作。

file\_path——按照文件路径标识文件

1. 搜索和更新都简单，直接搜索路径字符串；路径不同就被认为是不同的文件需要创建新metadata，没有hash值的更新开销。
2. 可以使用合约限制只有metadata最初创建者的账户可以修改metadata
3. 类似于git对于文件指针的定义，保护的就是当前的这个文件指针对应的权限，即使有其他文件指针和我指向同一个blob，则这个文件指针可以对应另外的权限（metadata）。
4. 文件重命名后或文件路径被修改后会认为是新创建的文件，要求修改者创建新的metadata。

# 原论文

分布式版本控制系统：

用户获取文件时将metadata一起获取，metadata与文件绑定，要么全拷要么都不拷

分布式Server存储

base hash 由Data Owner修改，

文件格式中是否有字段 只能由拥有者修改

1. 不能只用path的原因

分布式的Server，repository存在于每个用户，path只能保证相对路径，而绝对路径——repository的基地址是不同的，但是每个repository都应该对应同样的metadata

2. 不能只用hash的原因——不能周期性签署的原因

hash需要不断地修改，所有用户都可以调用合约，需要作者每次都进行签名。

如果改为周期性更新，用户修改文件产生新的hash值，只是传到合约中而不写入区块链，在一定时间后由作者同一将这些hash签名修改，这会导致在更新metadata前，下载最新文件版本的用户需要计算旧版本的hash，比较麻烦。

3. 不能只用base hash的原因

既然hash需要一直修改，那如果将文件初始创建时的内容（cipher+sig）哈希后作为base ash，将其看作file\_id，是否可以解决这个问题呢？

此时metadata中有（oid, base hash, read policy, write policy）

文件中有作者签名的base hash。

问题在于，恶意攻击者可以将base hash修改后用自己的私钥签名，针对修改后的base hash创建新的metadata。这样下载此版本的其他用户就会顺着假的base hash找到攻击者的metadata，从而认为其有写权限。

4.使用path和repo hash的方案 威胁模型

针对base hash的安全问题，攻击者想让其他用户相信他的修改是合法的，前提是他修改了一个已经存在的文件，而不是他自己创建的新文件。而攻击成功的另一大前提是攻击者可以生成针对这个已存在文件的新的metadata，这是因为文件的metadata只用base hash确定了。那如果加入文件外部的信息——文件路径呢?

如果加入了文件路径，这个是在违法修改中不会改变的，而攻击者想产生新的metadata，在创建时合约会搜索到此path已经存在，而攻击者不是作者，所以拒绝其进行metadata的修改和创建。

但是path方法的约束呢？这可以用repo hash来代替每个文件的base hash来解决。这就相当于所有的repository，即使存在不同的用户主机上，但是base repo hash是不变的，这时候加上文件在repo中的相对路径，就可以唯一的标识这个文件。

检查时需要get到最初版本的metadata作者做比对。

攻击模型：攻击者攻击文件中的（base repo hash+transaction id）或修改区块链中的meatdata

为什么部放在git中存： 文件唯一性。如果有恶意用户很早就在某个路径下建立了a.txt文件，之后删除了。但是Git中此路径的metadata初作者是此用户。若后面有其他用户创建了a.txt，页创建了相应的metadata，但是其他用户在获取metadata时，会认为当前作者与原作者不符。但是在区块链上就不一样了，按照repo hash和path查询，若用户不是初始建立的用户，会不允许进行metadata的改动，要求其更改文件名称建立新的metadata。

5. 分布式的约束