可插拔交易背书与交易验证

动机

交易在提交时会被验证，在应用交易带来的状态改变之前，节点会对交易进行以下检查：

- 验证交易签名者的身份；
- 验证交易背书者的签名；
- 确认交易满足对应链码命名空间的相关背书策略。

在某些情况下，需要自定义不同于 Fabric 默认的交易验证规则，例如：

- **UTXO（未花费的交易输出）**： 当需要验证账户是否有双花的情况时；
- **匿名交易**： 当背书中不包含节点身份，并且共享的签名和公钥也无法与节点的身份联系起来时。

可插拔的背书和验证逻辑

Fabric 支持以可插拔的方式在 Peer 节点中实现和部署与链码相关的自定义的背书和验证逻辑。这种逻辑既可作为可选逻辑内置到 Peer 节点中，也可作为一个 Golang 插件 与 Peer 节点一起编译和部署。

注解

Go plugins have a number of practical restrictions that require them to be compiled and linked in the same build environment as the peer. Differences in Go package versions, compiler versions, tags, and even GOPATH values will result in runtime failures when loading or executing the plugin logic.

默认情况下，链码将使用内置的背书和验证逻辑。不过，用户可以选择使用自定义的背书和验证插件来作为链码定义的一部分。管理员可通过自定义 Peer 节点的本地配置来扩展背书或验证逻辑。

配置

每个 Peer 节点都有一个本地配置（`core.yaml`），其中包括了背书或验证逻辑的名称与具体实现的映射关系。

默认的逻辑叫做 `ESCC` （其中“E”代表 endorsement，背书）和 `VSCC` （validation，验证）， Peer 节点本地配置中的 `handlers` 部分包含了该默认逻辑：

```
handlers:
    endorsers:
      escc:
        name: DefaultEndorsement
    validators:
      vscc:
        name: DefaultValidation
```

当背书或验证的实现被编译到 Peer 节点中时， `name` 属性就代表了即将运行的初始化函数，以便获得生成背书或验证逻辑相关实例的工厂。

该函数是 `core/handlers/library/library.go` 中 `HandlerLibrary` 结构的实例方法，而且为了添加自定义的背书或验证逻辑，就需要额外的方法对该结构进行扩展。

If the custom code is built as a Go plugin, the `library` property must be provided and set to the location of the shared library.

比如，我们以插件来实现自定义背书和验证逻辑，那么 `core.yaml` 的配置中就会有以下内容：

```
handlers:
    endorsers:
      escc:
        name: DefaultEndorsement
      custom:
        name: customEndorsement
```

```
        library: /etc/hyperledger/fabric/plugins/customEndorsement.so
    validators:
      vscc:
        name: DefaultValidation
      custom:
        name: customValidation
        library: /etc/hyperledger/fabric/plugins/customValidation.so
```

并且我们需要把 `.so` 插件文件放置在 Peer 节点的本地文件系统中。

自定义插件的名称需要在使用的链码定义中引用。如果你使用 CLI 来执行链码定义，请使用 `--escc` 和 `--vscc` 标识来选择自定义的背书或者验证库。如果使用 Fabric Node.js SDK，请访问 如何安装和启动你的链码 。更多信息查阅 chaincode4noah 。

注解

后边内容中，自定义背书和验证逻辑的实现都将表述为“插件”，即使被编译到 Peer 节点中。

背书插件的实现

要实现一个背书插件，用户必须实现 `core/handlers/endorsement/api/endorsement.go` 中的 `Plugin` 接口。

```
// Plugin endorses a proposal response
type Plugin interface {
    // Endorse signs the given payload(ProposalResponsePayload bytes), and optionally mutates it.
    // Returns:
    // The Endorsement: A signature over the payload, and an identity that is used to verify the s
    // The payload that was given as input (could be modified within this function)
    // Or error on failure
    Endorse(payload []byte, sp *peer.SignedProposal) (*peer.Endorsement, []byte, error)

    // Init injects dependencies into the instance of the Plugin
    Init(dependencies ...Dependency) error
}
```

当 Peer 节点调用 `PluginFactory` 接口中的 `New` 方法时，会为每个通道创建给定类型（无论是 `HandlerLibrary` 示例方法的方法名还是 `.so` 文件路径）的背书插件实例， `PluginFactory` 接口需要由插件开发者实现：

```
// PluginFactory creates a new instance of a Plugin
type PluginFactory interface {
    New() Plugin
}
```

`Init` 方法将接收 `core/handlers/endorsement/api/` 中声明的所有依赖项作为输入，这些依赖项会被识别为内嵌 `Dependency` 接口。

创建了 `Plugin` 实例后，Peer 节点在实例上调用 `Init` 方法，并把 `dependencies` 作为参数传递。

目前，Fabric 存在以下背书插件的依赖项：

```
// SigningIdentity signs messages and serializes its public identity to bytes
type SigningIdentity interface {
    // Serialize returns a byte representation of this identity which is used to verify
    // messages signed by this SigningIdentity
    Serialize() ([]byte, error)

    // Sign signs the given payload and returns a signature
    Sign([]byte) ([]byte, error)
}
```

- `StateFetcher`：获取一个与世界状态交互的 **状态** 对象

```
// State defines interaction with the world state
type State interface {
    // GetPrivateDataMultipleKeys gets the values for the multiple private data items in a single
    GetPrivateDataMultipleKeys(namespace, collection string, keys []string) ([][]byte, error)

    // GetStateMultipleKeys gets the values for multiple keys in a single call
    GetStateMultipleKeys(namespace string, keys []string) ([][]byte, error)

    // GetTransientByTXID gets the values private data associated with the given txID
    GetTransientByTXID(txID string) ([]*rwset.TxPvtReadWriteSet, error)
```

```
    // Done releases resources occupied by the State
    Done()
}
```

验证插件的实现

要实现一个验证插件，用户必须实现 `core/handlers/validation/api/validation.go` 中的 `Plugin` 接口：

```
// Plugin validates transactions
type Plugin interface {
    // Validate returns nil if the action at the given position inside the transaction
    // at the given position in the given block is valid, or an error if not.
    Validate(block *common.Block, namespace string, txPosition int, actionPosition int, contextDat

    // Init injects dependencies into the instance of the Plugin
    Init(dependencies ...Dependency) error
}
```

每个 `ContextDatum` 都是运行时派生的额外元数据，由节点负责传递给验证插件。目前，代表链码背书策略的 `ContextDatum` 是唯一被传递的数据 。

```
// SerializedPolicy defines a serialized policy
type SerializedPolicy interface {
      validation.ContextDatum

      // Bytes returns the bytes of the SerializedPolicy
      Bytes() []byte
}
```

当 Peer 节点调用 `PluginFactory` 接口中的 `New` 方法时，会为每个通道创建给定类型（无论是 `HandlerLibrary` 示例方法的方法名还是 `.so` 文件路径）的验证插件实例， `PluginFactory` 接口需要由插件开发者实现。

```
// PluginFactory creates a new instance of a Plugin
type PluginFactory interface {
    New() Plugin
}
```

`Init` 方法将接收 `core/handlers/validation/api/` 中声明的所有依赖项作为输入，这些依赖项会被识别为内嵌 `Dependency` 接口。

创建了 `Plugin` 实例后，Peer 节点在实例上调用 `Init` 方法，并把 `dependencies` 作为参数传递。

目前，Fabric 存在以下验证插件的依赖项：

- `IdentityDeserializer`：将表示身份的字节转换为 `Identity` 对象，该对象可用于验证由这些身份的签名，并根据各自的 MSP 对自身进行验证，以查看它们是否满足给定的 **MSP 准则**。`core/handlers/validation/api/identities/identities.go` 中包含了全部的规范。
- `PolicyEvaluator`：评估是否满足给定的策略：

```
// PolicyEvaluator evaluates policies
type PolicyEvaluator interface {
    validation.Dependency

    // Evaluate takes a set of SignedData and evaluates whether this set of signatures satisfies
    // the policy with the given bytes
    Evaluate(policyBytes []byte, signatureSet []*common.SignedData) error
}
```

- `StateFetcher`：获取一个与世界状态中的 `State` 对象：

```
// State defines interaction with the world state
type State interface {
    // GetStateMultipleKeys gets the values for multiple keys in a single call
    GetStateMultipleKeys(namespace string, keys []string) ([][]byte, error)

    // GetStateRangeScanIterator returns an iterator that contains all the key-values between give
    // startKey is included in the results and endKey is excluded. An empty startKey refers to the
    // and an empty endKey refers to the last available key. For scanning all the keys, both the s
    // can be supplied as empty strings. However, a full scan should be used judiciously for perfo
    // The returned ResultsIterator contains results of type *KV which is defined in fabric-protos
    GetStateRangeScanIterator(namespace string, startKey string, endKey string) (ResultsIterator,
```

```
    // GetStateMetadata returns the metadata for given namespace and key
    GetStateMetadata(namespace, key string) (map[string][]byte, error)

    // GetPrivateDataMetadata gets the metadata of a private data item identified by a tuple <name
    GetPrivateDataMetadata(namespace, collection, key string) (map[string][]byte, error)

    // Done releases resources occupied by the State
    Done()
}
```

重要提示

- **各节点上的验证插件保持一致：** 在以后的版本中，Fabric 通道基础设施将确保在给定区块链高度上，通道内所有节点对给定链码使用相同的验证逻辑，以消除可能导致节点间状态分歧的错误配置风险，若发生错配置，则可能会致使节点运行不同的实现。但就目前来说，系统操作员和管理员的唯一责任就是确保以上问题不会发生。
- **验证插件错误处理：** 当因发生某些暂时性执行问题（比如无法访问数据库）而导致验证插件不能确定一笔交易是否有效时，插件应返回 `core/handlers/validation/api/validation.go` 中定义的 **ExecutionFailureError** 类型的错误。任何其他被返回的错误将被视为背书策略错误，并且被验证逻辑标记为无效。但是，如果返回的错误是 `ExecutionFailureError`，链处理程序不会将该交易标志为无效，而是暂停处理。目的是防止不同节点之间发生状态分歧。
- **私有元数据索取的错误处理：** 当一个插件利用 `StateFetcher` 接口来为私有数据索取元数据时，错误处理需要按一下方式来处理：`CollConfigNotDefinedError'' 和 ``InvalidCollNameError''，表明指定的集合不存在，应该按照确定性的错误来处理，而不是 ``Executi`
- **将 Fabric 代码导入插件：** 强烈不建议将 Fabric 代码导入插件而不使用 protobufs，这样做会在 Fabric 代码更新时出现问题，或者当运行不同版本的节点时，引起操作问题。理想情况下，插件代码应该只使用提供的依赖，并除了 protobufs 之外的最小化导入项。