通道配置（configtx）

Hyperledger Fabric 区块链网络的共享配置存储在一个集合配置交易中，每个通道一个。 配置交易通常简称为 *configtx*。

通道配置有以下重要特性：

1. **版本化**：配置中的所有元素都有一个关联的、每次修改都提高的版本。此外，每个 提交的配置都有一个序列号。
2. **许可的**：配置中的所有元素都有一个关联的、控制对此元素的修改是否被允许的策略。 任何拥有之前 configtx 副本（没有附加信息）的人均可以这些策略为基础来验证新配置 的有效性。
3. **分层的**：根配置组包含子组，而且层次结构中的每个组都有关联的值和策略。这些 策略可以利用层次结构从较低层级的策略派生出一个层级的策略。

配置解析

配置，作为一个类型为 `HeaderType_CONFIG` 的交易，被存储在一个没有其他交易的 区块中。这些区块被称为 *配置区块*，其中的第一个就是 *创世区块*。

配置的原型结构在 `fabric-protos/common/configtx.proto` 中。类型为 `HeaderType_CONFIG` 的信封将 `ConfigEnvelope` 消息编码为 `Payload` `data` 字段。 `ConfigEnvelope` 的原型定义如下：

```
message ConfigEnvelope {
    Config config = 1;
    Envelope last_update = 2;
}
```

`last_update` 字段在下面的 **配置更新** 一节定义，但只有当验证配置时才是必需的， 读取时则不是。当前提交的配置存储在 `config` 字段，包含 `Config` 消息。

```
message Config {
    uint64 sequence = 1;
    ConfigGroup channel_group = 2;
}
```

`sequence` 数字每次提交配置时增加1。 `channel_group` 字段是包含配置的根组。 `ConfigGroup` 结构是递归定义的，并构建了一个组的树，每个组都包含值和策略。其 定义如下：

```
message ConfigGroup {
    uint64 version = 1;
    map<string,ConfigGroup> groups = 2;
    map<string,ConfigValue> values = 3;
    map<string,ConfigPolicy> policies = 4;
    string mod_policy = 5;
}
```

因为 `ConfigGroup` 是递归结构，所以它有层次结构。为清楚起见，下面的例子使用 golang 展现。

```
// Assume the following groups are defined
var root, child1, child2, grandChild1, grandChild2, grandChild3 *ConfigGroup

// Set the following values
root.Groups["child1"] = child1
root.Groups["child2"] = child2
child1.Groups["grandChild1"] = grandChild1
child2.Groups["grandChild2"] = grandChild2
child2.Groups["grandChild3"] = grandChild3

// The resulting config structure of groups looks like:
// root:
//     child1:
//         grandChild1
//     child2:
//         grandChild2
//         grandChild3
```

每个组定义了配置层次中的一个级别，每个组都有关联的一组值（按字符串键索引）和策略 （也按字符串键索引）。

值定义：

Values are defined by:

```
message ConfigValue {
    uint64 version = 1;
    bytes value = 2;
    string mod_policy = 3;
}
```

策略定义：

```
message ConfigPolicy {
    uint64 version = 1;
    Policy policy = 2;
    string mod_policy = 3;
}
```

注意，值、策略和组都有 `version` 和 `mod_policy`。一个元素每次修改时 `version` 都会增长。`mod_policy` 被用来控制修改元素时所需要的签名。 对于组，修改就是增加或删除值、策略或组映射中的元素（或改变 `mod_policy`）。 对于值和策略，修改就是分别改变值和策略字段（或改变 `mod_policy`）。每个元素 的 `mod_policy` 都在当前层级配置的上下文中被评估。下面是一个定义在 `Channel.Groups["Application"]` 中的修改策略示例（这里，我们使用 golang map 语法，所以，`Channel.Groups["Application"].Policies["policy1"]` 表示根组 `Channel` 的子组 `Application` 的 `Policies` 中的 `policy1` 所对应的 策略。）

- `policy1` 对应 `Channel.Groups["Application"].Policies["policy1"]`
- `Org1/policy2` 对应 `Channel.Groups["Application"].Groups["Org1"].Policies["policy2"]`
- `/Channel/policy3` 对应 `Channel.Policies["policy3"]`

注意，如果一个 `mod_policy` 引用了一个不存在的策略，那么该元素不可修改。

Note that if a `mod_policy` references a policy which does not exist, the item cannot be modified.

配置更新

配置更新被作为一个类型为 `HeaderType_CONFIG_UPDATE` 的 `Envelope` 消息提交。 交易中的 `Payload` `data` 是一个封送的 `ConfigUpdateEnvelope`。 `ConfigUpdateEnvelope` 定义如下:

```
message ConfigUpdateEnvelope {
    bytes config_update = 1;
    repeated ConfigSignature signatures = 2;
}
```

`signatures` 字段包含一组授权配置更新的签名。它的消息定义如下:

```
message ConfigSignature {
    bytes signature_header = 1;
    bytes signature = 2;
}
```

`signature_header` 是为标准交易定义的，而签名是通过 `signature_header` 字节 和 `ConfigUpdateEnvelope` 中的 `config_update` 字节串联而得。

`ConfigUpdateEnvelope` `config_update` 字节是封送的 `ConfigUpdate` 消息，定义如下:

```
message ConfigUpdate {
    string channel_id = 1;
    ConfigGroup read_set = 2;
    ConfigGroup write_set = 3;
}
```

`channel_id` 是更新所绑定的通道 ID，这对于确定支持此重配置的签名的作用域 是必需的。

`read_set` 定义了现有配置的子集，属稀疏指定，其中只设置 `version` 字段， 其他字段不需要填充。尤其 `ConfigValue` `value` 或者 `ConfigPolicy` `policy` 字段不应在 `read_set` 中设置。 `ConfigGroup` 可以有已填充 映射字段的子集，以便引用配置树中更深层次的元素。例如，要将 `Application` 组 包含在 `read-set` 中，其父组（ `Channel` 组）也必须包含在读集合中，但 `Channel` 组不需要填充所有键，例如 `Orderer` `group` 键，或任何 `values` 或 `policies` 键。

`write_set` 指定了要修改的配置片段。由于配置的层次性，对层次结构中深层元素 的写入也必须在其 `write_set` 中包含更高级别的元素。但是，对于 `read-set` 中也指定的 `write-set` 中的任何同一版本的元素，应该像在 ``read-set``中一样 稀疏地指定该元素。

例如，给定配置:

```
Channel: (version 0)
    Orderer (version 0)
    Application (version 3)
       Org1 (version 2)
```

为了提交一个修改 `Org1` 的配置更新， `read_set` 应如：

```
Channel: (version 0)
    Application: (version 3)
```

`write_set` 应如

```
Channel: (version 0)
    Application: (version 3)
        Org1 (version 3)
```

收到 `CONFIG_UPDATE` 后，排序节点按以下步骤计算 `CONFIG` 结果。

1. 验证 `channel_id` 和 `read_set` 。 `read_set` 中的所有元素都必须 以给定的版本存在。
2. 收集 `write_set` 中的所有与 `read_set` 版本不一致的元素以计算更新集。
3. 校验更新集合中版本号刚好增长了1的每个元素。
4. 校验附加到 `ConfigUpdateEnvelope` 的签名集是否满足更新集中每个元素的 `mod_policy` 。
5. 通过应用更新到到当前配置，计算出配置的新的完整版本。
6. 将配置写入 `ConfigEnvelope` ，包含作为 `last_update` 字段的 `CONFIG_UPDATE` ，和编码为 `config` 字段的新配置, 以及递增的 `sequence` 值。
7. 将新 `ConfigEnvelope` 写入类型为 `CONFIG` 的 `Envelope` ，并最终将其 作为唯一交易写入一个新的配置区块。

当节点（或其他任何 `Deliver` 的接收者）收到这个配置区块时，它应该，将 `last_update` 消息应用到当前配置并校验经过排序计算的 `config` 字段包含 当前的新配置，以此来校验这个配置是否得到了适当地验证。

组和值的许可配置

任何有效配置都是以下配置的子集。在这里，我们用符号 `peer.<MSG>` 来定义一个 `ConfigValue` ，其 `value` 字段是一个封送的名为 `<MSG>` 的消息。它定义在 `fabric-protos/peer/configuration.proto` 中。符号 `common.<MSG>` 、 `msp.<MSG>` 和 `orderer.<MSG>` 类似对应，它们的消息依次定义在 `fabric-protos/common/configuration.proto` 、 `fabric-protos/msp/mspconfig.proto` 和 ``fabric-protos/orderer/configuration.proto``中

注意，键 `{{org_name}}` 和 `{{consortium_name}}` 表示任意名称，指示一个 可以用不同名称重复的元素。

Note, that the keys `{{org_name}}` and `{{consortium_name}}` represent arbitrary names, and indicate an element which may be repeated with different names.

```
&ConfigGroup{
    Groups: map<string, *ConfigGroup> {
        "Application":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{org_name}}:&ConfigGroup{
                    Values:map<string, *ConfigValue>{
```

```
                        "MSP":msp.MSPConfig,
                        "AnchorPeers":peer.AnchorPeers,
                    },
                },
            },
        },
        "Orderer":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{org_name}}:&ConfigGroup{
                    Values:map<string, *ConfigValue>{
                        "MSP":msp.MSPConfig,
                    },
                },
            },

            Values:map<string, *ConfigValue> {
                "ConsensusType":orderer.ConsensusType,
                "BatchSize":orderer.BatchSize,
                "BatchTimeout":orderer.BatchTimeout,
                "KafkaBrokers":orderer.KafkaBrokers,
            },
        },
        "Consortiums":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{consortium_name}}:&ConfigGroup{
                    Groups:map<string, *ConfigGroup> {
                        {{org_name}}:&ConfigGroup{
                            Values:map<string, *ConfigValue>{
                                "MSP":msp.MSPConfig,
                            },
                        },
                    },
                    Values:map<string, *ConfigValue> {
                        "ChannelCreationPolicy":common.Policy,
                    }
                },
            },
        },
    },

    Values: map<string, *ConfigValue> {
        "HashingAlgorithm":common.HashingAlgorithm,
        "BlockHashingDataStructure":common.BlockDataHashingStructure,
        "Consortium":common.Consortium,
        "OrdererAddresses":common.OrdererAddresses,
    },
}
```

排序系统通道配置

排序系统通道需要定义一些排序参数，以及创建通道的联盟。一个排序服务有且只能有一个 排序系统通道，它是需要创建的第一个通道（或更准确地说是启动）。建议不要在排序系统 通道的创世配置中定义应用，但在测试时是可以的。注意，任何对排序系统通道具有读权限 的成员可能看到所有的通道创建，所以，这个通道的访问应用受到限制。

排序参数被定义在如下配置子集中：

```
&ConfigGroup{
    Groups: map<string, *ConfigGroup> {
        "Orderer":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{org_name}}:&ConfigGroup{
                    Values:map<string, *ConfigValue>{
                        "MSP":msp.MSPConfig,
                    },
                },
            },

            Values:map<string, *ConfigValue> {
                "ConsensusType":orderer.ConsensusType,
```

```
                "BatchSize":orderer.BatchSize,
                "BatchTimeout":orderer.BatchTimeout,
                "KafkaBrokers":orderer.KafkaBrokers,
            },
        },
    },
```

参与排序的每个组织在 `Order` 组下都有一个组元素。此组定义单个参数 `MSP`， 其中包含该组织的加密身份信息。 `Order` 组的 `Values` 决定了排序节点的工作 方式。它们在每个通道中存在，因此，例如 `orderer.BatchTimeout` 可能在不同通 道上被不同地指定。

在启动时，排序节点将面临一个包含了很通道信息的文件系统。排序节点通过识别带有定义的 联盟组的通道来识别系统通道。联盟组的结构如下。

```
&ConfigGroup{
    Groups: map<string, *ConfigGroup> {
        "Consortiums":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{consortium_name}}:&ConfigGroup{
                    Groups:map<string, *ConfigGroup> {
                        {{org_name}}:&ConfigGroup{
                            Values:map<string, *ConfigValue>{
                                "MSP":msp.MSPConfig,
                            },
                        },
                    },
                    Values:map<string, *ConfigValue> {
                        "ChannelCreationPolicy":common.Policy,
                    }
                },
            },
        },
    },
},
```

注意，每个联盟定义一组成员，正如排序组织里的组织成员一样。每个联盟也定义一个 `ChannelCreationPolicy`。这是一个应用于授权通道创建请求的策略。通常，该值 将被设置为一个 `ImplicitMetaPolicy`，并要求通道的新成员签名以授权通道创建。 更多关于通道创建的细节，请参见下文。

应用通道配置

应用配置适用于为应用类型交易而设计的通道。它定义如下：

```
&ConfigGroup{
    Groups: map<string, *ConfigGroup> {
        "Application":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{org_name}}:&ConfigGroup{
                    Values:map<string, *ConfigValue>{
                        "MSP":msp.MSPConfig,
                        "AnchorPeers":peer.AnchorPeers,
                    },
                },
            },
        },
    },
}
```

正如 `Orderer` 部分，每个组织被编码为组。但是，并非仅仅编码 `MSP` 身份信息， 每个组织额外编码一个 `AnchorPeers` 列表。这个列表允许不同组织的节点互相联系以 建立对等 gossip 网络。

应用通道对排序组织副本和共识选项进行编码，以允许对这些参数进行确定的更新，因此排序 系统通道配置中相同的 `Orderer` 部分也被包括在内。但从应用程序角度来看，这在很大 程度上可能被忽略。

通道创建

当排序节点收到一个尚不存在的通道的 `CONFIG_UPDATE` 时，排序节点假定这是一个通道 创建请求并执行以下内容。

1. 排序节点识别将为之执行通道创建请求的联盟。它通过查看顶级组的 `Consortium` 值 来完成这一操作。
2. 排序节点校验：包含在 `Application` 组中的组织是包含在对应联盟中的组织的一个子集， 并且 `ApplicationGroup` `version` 被设为 `1`。
3. 排序节点校验：如果联盟有成员，那么新通道也要有应用成员（创建没有成员的联盟和通道仅 用于测试）。
4. 排序节点从排序系统通道中获取 `Orderer` 组，使用新指定的成员建立 `Application` 组，并按联盟配置中指定的 `ChannelCreationPolicy` 指定它的 `mod_policy`， 以此建立模板配置。注意，策略会在新配置的上下文被评估，所以，一个要求 `ALL` 成员 的策略，会要求新通道所有成员的签名，而不是联盟所有成员的签名。
5. 然后排序节点将 `CONFIG_UPDATE` 作为一个更新应用到这个模板配置。 因为 `CONFIG_UPDATE` 将修改应用到 `Application` 组（它的 `version` 是 `1`），配置代码按 `ChannelCreationPolicy` 验证这些更新。如果通道的创 建包含任何其他修改，比如个别组织的锚节点，则这个元素相应的修改策略也会被调用。
6. 带有新通道配置的 `CONFIG` 交易被封装并发送给的排序系统通道以进行排序，排序后， 通道被创建了。