# Telegram 開放網路區塊鏈

Nikolai Durov
譯者：Dr. Awesome Doge

November 1, 2025

**Abstract**

本文旨在提供 Telegram 開放網路（TON）區塊鏈的詳細描述。

# 引言

本文件提供 TON 區塊鏈的詳細描述，包括其精確的區塊格式、有效性條件、TON 虛擬機（TVM）呼叫細節、智慧合約建立過程和密碼學簽章。在這方面，它是 TON 白皮書（參見 [3]）的延續，因此我們自由使用該文件中引入的術語。

章節 **1** 提供 TON 區塊鏈及其設計原則的一般概述，特別關注相容性和有效性條件的引入以及訊息傳遞保證的實作。更詳細的資訊，例如描述將所有必需資料結構序列化為單元樹或單元集合（「袋」）的 TL-B 方案，在後續章節中提供，最終在章節 **5** 中完整描述 TON 區塊鏈（分片鏈和主鏈）區塊布局。

附錄 **A** 中提供了用於簽署區塊和訊息的橢圓曲線密碼學的詳細描述，這些密碼學也可透過 TVM 原語存取。TVM 本身在單獨的文件中描述（參見 [4]）。

某些主題已刻意排除在本文件之外。一個是驗證者用來確定主鏈或分片鏈下一個區塊的拜占庭容錯（BFT）協議；該主題留待即將發佈的專門介紹 TON 網路的文件討論。雖然本文件描述了 TON 區塊鏈區塊的精確格式，並討論了區塊鏈的有效性條件和序列化的無效性證明，[1] 它沒有提供

[1]截至 2018 年 8 月，本文件不包含序列化無效性證明的詳細描述，因為它們可能會在驗證者軟體開發期間發生重大變化。僅討論一致性條件和序列化無效性證明的一般設計原則。

用於傳播這些區塊、候選區塊、整理區塊和無效性證明的網路協議的詳細資訊。

同樣，本文件不提供主鏈智慧合約的完整原始碼，這些合約用於選舉驗證者、變更可設定參數或取得其當前值，或懲罰驗證者的不當行為，儘管這些智慧合約構成總區塊鏈狀態和主鏈區塊零的重要部分。相反，本文件描述了這些智慧合約的位置及其正式介面。[2] 這些智慧合約的原始碼將作為帶有註解的可下載檔案單獨提供。

請注意，本文件的當前版本描述了 TON 區塊鏈的初步測試版本；在開發、測試和部署階段啟動之前，某些小細節可能會發生變化。

[2]這不包含在本文件的當前版本中，但將在未來修訂版的單獨附錄中提供。

# Contents

# 1 概述

本章概述 TON 區塊鏈的主要特性和設計原則。每個主題的更多詳細資訊在後續章節中提供。

## 1.1 一切皆為單元集合

TON 區塊鏈的區塊和狀態中的所有資料都表示為單元的集合（參見 [3, 2.5]）。因此，本章從單元的一般討論開始。

**1.1.1. TVM 單元.** 回顧一下，TON 區塊鏈以及 TON 虛擬機（TVM；參見 [4]）將所有永久儲存的資料表示為所謂單元的集合或袋。每個單元由最多 1023 個資料位元和最多四個對其他單元的參照組成。不允許循環單元參照，因此單元通常組織為單元樹，或更確切地說是單元有向無環圖（*DAG*）。[3] 抽象代數（依賴）資料型別的任何值都可以表示（序列化）為單元樹。將抽象資料型別的值表示為單元樹的精確方式透過 *TL-B* 方案來表達。[4] 關於不同類型單元的更詳盡討論可以在 [4, 3.1] 中找到。

**1.1.2. 應用於 TON 區塊鏈區塊和狀態.** 上述內容特別適用於 TON 區塊鏈的區塊和狀態，它們也是某些（相當複雜的）依賴代數資料型別的值。因此，它們根據各種 TL-B 方案進行序列化（這些方案在本文件中逐步呈現），並表示為單元的集合或袋。

**1.1.3. 單個單元的布局.** 每個單元由最多 1023 個資料位元和最多四個對其他單元的參照組成。當單元保存在記憶體中時，其確切表示取決於實作。然而，有一個單元的標準表示，例如對於序列化單元以進行檔案儲存或網路傳輸很有用。單元 $c$ 的這種「標準表示」或「標準布局」$\textsc{CellRepr}(c)$ 由以下內容組成：

- 首先是兩個描述符位元組，有時用 $d_1$ 和 $d_2$ 表示。第一個位元組 $d_1$ 等於（在最簡單的情況下）單元中參照的數量 $0 \le r \le 4$。第二個描述符位元組 $d_2$ 編碼單元資料部分的位元長度 $l$，如下所示：$d_2$ 的前七個位元等於 $\lfloor l/8 \rfloor$，即單元中存在的完整資料位元組的數量，而 $d_2$ 的最後一個位元是完成標記，如果 $l$ 不能被 8 整除，則等於 1。因此，

$$d_2 = 2\lfloor l/8 \rfloor + [l \bmod 8 \neq 0] = \lfloor l/8 \rfloor + \lceil l/8 \rceil \tag{1}$$

[3] 完全相同的單元通常在記憶體和磁碟儲存中被識別；這就是為什麼單元樹透明地轉換為單元 DAG 的原因。從這個角度來看，DAG 只是底層單元樹的儲存最佳化，與大多數考慮無關。

[4] 參見 [4, 3.3.3–4]，其中給出並解釋了一個例子，待更完整的參考

其中 $[A]$ 在條件 $A$ 為真時等於 1，否則等於 0。

- 接下來，跟隨 $\lceil l/8 \rceil$ 個資料位元組。這意味著單元的 $l$ 個資料位元被分成八個一組，每組被解釋為大端序 8 位元整數並儲存到一個位元組中。如果 $l$ 不能被 8 整除，則在資料位元後附加一個二進位 1 和適當數量的二進位 0（最多六個），並設定完成標記（描述符位元組 $d_2$ 的最低有效位元）。
- 最後，跟隨 $r$ 個對其他單元的參照。每個參照通常由 32 個位元組表示，包含被參照單元的 SHA256 雜湊，如下面 **1.1.4** 中所述計算。

這樣，具有 $l$ 個資料位元和 $r$ 個參照的單元 $c$ 的標準表示 $\textsc{CellRepr}(c)$ 的長度為 $2 + \lfloor l/8 \rfloor + \lceil l/8 \rceil + 32r$ 個位元組。

**1.1.4. 單元的 sha256 雜湊.** 單元 $c$ 的 SHA256 雜湊遞迴定義為相關單元的標準表示 $\textsc{CellRepr}(c)$ 的 SHA256：

$$\textsc{Hash}(c) := \textsc{sha256}(c) := \textsc{sha256}\big(\textsc{CellRepr}(c)\big) \tag{2}$$

因為不允許循環單元參照（所有單元之間的關係必須構成有向無環圖或 DAG），單元的 SHA256 雜湊總是明確定義的。

此外，因為 SHA256 被隱含地假定為抗碰撞的，我們假設我們遇到的所有單元完全由它們的雜湊決定。特別地，單元 $c$ 的單元參照完全由包含在標準表示 $\textsc{CellRepr}(c)$ 中的被參照單元的雜湊決定。

**1.1.5. 奇異單元.** 除了目前為止考慮的普通單元（也稱為簡單或資料單元）之外，稱為奇異單元的其他類型的單元有時會出現在 TON 區塊鏈區塊和其他資料結構的實際表示中。它們的表示有所不同；它們透過第一個描述符位元組 $d_1 \geq 5$ 來區分（參見 [4, 3.1]）。

**1.1.6. 外部參照單元.** （外部）參照單元包含「真實」資料單元 $c$ 的 32 位元組 $\textsc{sha256}(c)$，而不是資料單元本身，是奇異單元的一個例子。這些單元可用於對應於 TON 區塊鏈區塊的單元袋的序列化中，以參照不在區塊序列化本身中但假定存在於其他地方（例如，在區塊鏈的先前狀態中）的資料單元。

**1.1.7. 參照單元相對於大多數操作的透明性.** 大多數單元操作不觀察任何參照單元或其他「奇異」類型的單元；它們只看到資料單元，任何參照單元都透明地替換為所參照的單元。例如，當遞迴計算透明單元雜湊 $\textsc{Hash}^\flat(c)$ 時，參照單元的雜湊設定為等於所參照單元的雜湊，而不是參照單元的標準表示的雜湊。

**1.1.8. 單元的透明雜湊和表示雜湊.** 這樣，$\text{SHA256}^\flat(c) = \text{HASH}^\flat(c)$ 是單元 $c$（或以 $c$ 為根的單元樹）的透明雜湊。

然而，有時我們需要推理區塊中存在的單元樹的確切表示。為此，定義了表示雜湊 $\text{HASH}^\sharp(c)$，它對於參照單元和其他奇異類型的單元不是透明的。我們經常說 $c$ 的表示雜湊是「 $c$ 的」雜湊，因為它是最常用的單元雜湊。

**1.1.9. 使用表示雜湊進行簽章.** 簽章是表示雜湊應用的絕佳例子。例如：

- 驗證者簽署區塊的表示雜湊，而不僅僅是其透明雜湊，因為他們需要證明區塊確實包含所需的資料，而不僅僅是對它們的一些外部參照。
- 當外部訊息由鏈外各方（例如，使用應用程式發起區塊鏈交易的人類客戶端）簽署和傳送時，如果這些訊息中的某些可能存在外部參照，則必須簽署訊息的表示雜湊。

**1.1.10. 單元的更高階雜湊.** 除了單元 $c$ 的透明和表示雜湊之外，還可以定義一系列更高階雜湊 $\text{HASH}_i(c)$，$i = 1, 2, \ldots$，它最終穩定在 $\text{HASH}_\infty(c)$。(更多細節可以在 [4, 3.1] 中找到。)

## 1.2 區塊和區塊鏈狀態的主要組件

本節簡要描述區塊和區塊鏈狀態的主要組件，而不深入細節。

**1.2.1. 應用於區塊鏈區塊和狀態的無限分片範式（ISP）.** 回顧一下，根據無限分片範式，每個帳戶可以被視為位於其單獨的「帳戶鏈」中，然後這些帳戶鏈的（虛擬）區塊為了效率目的而分組到分片鏈區塊中。具體來說，分片鏈的狀態大致由其所有「帳戶鏈」(即分配給它的所有帳戶）的狀態組成；類似地，分片鏈的區塊本質上由分配給分片鏈的某些帳戶的虛擬「區塊」的集合組成。[5]

我們可以如下總結：

$$\mathit{ShardState} \approx \mathit{Hashmap}(n, \mathit{AccountState}) \tag{3}$$

$$\mathit{ShardBlock} \approx \mathit{Hashmap}(n, \mathit{AccountBlock}) \tag{4}$$

其中 $n$ 是 $account\_id$ 的位元長度，$\mathit{Hashmap}(n, X)$ 描述從長度為 $n$ 的位元串到類型 $X$ 的值的部分映射 $\mathbf{2}^n \dashrightarrow X$。

[5]如果沒有與帳戶相關的交易，則相應的虛擬區塊為空並在分片鏈區塊中省略

回顧一下，每個分片鏈——或更準確地說，每個分片鏈區塊[6]——對應於屬於同一「工作鏈」（即具有相同 $workchain\_id = w$）並且 $account\_id$ 以相同二進位前綴 $s$ 開頭的所有帳戶鏈，因此 $(w, s)$ 完全確定一個分片。因此，上述雜湊映射必須僅包含以前綴 $s$ 開頭的鍵。

我們將在稍後看到，上述描述只是一個近似值：分片鏈的狀態和區塊需要包含一些額外的資料，這些資料不會按照 (3) 所建議的那樣根據 $account\_id$ 進行分割。

**1.2.2. 分片鏈區塊和狀態的分割和非分割部分.** 分片鏈區塊及其狀態可以分別分類為兩個不同的部分。具有 (3) 的 ISP 規定形式的部分將被稱為區塊及其狀態的分割部分，而其餘部分將被稱為非分割部分。

**1.2.3. 與其他區塊和外部世界的互動。全域和局部一致性條件.** 分片鏈區塊及其狀態的非分割部分主要與此區塊與某些其他「相鄰」區塊的互動有關。區塊鏈整體的全域一致性條件簡化為單獨區塊本身的內部一致性條件以及某些區塊之間的外部局部一致性條件（參見 **1.3**）。

這些局部一致性條件中的大多數與不同分片鏈之間的訊息轉發、涉及多個分片鏈的交易以及訊息傳遞保證有關。然而，另一組局部一致性條件將區塊與其在分片鏈內的直接前驅和後繼聯繫起來；例如，區塊的初始狀態通常必須與其直接前驅的最終狀態一致。[7]

**1.2.4. 區塊的入站和出站訊息.** 分片鏈區塊的非分割部分的最重要組件如下：

- *InMsgDescr* ——所有「匯入」到此區塊的訊息的描述（即，由區塊中包含的交易處理，或轉發到輸出佇列，在沿著超立方體路由規定的路徑行進的轉接訊息的情況下）。

- *OutMsgDescr* ——所有由區塊「匯出」或「生成」的訊息的描述（即，由區塊中包含的交易生成的訊息，或目的地不屬於當前分片鏈的轉接訊息，從 *InMsgDescr* 轉發）。

**1.2.5. 區塊標頭.** 分片鏈區塊的另一個非分割組件是區塊標頭，它包含一般資訊，例如 $(w, s)$（即 $workchain\_id$ 和分配給當前分片鏈的所有 $account\_id$ 的公共二進位前綴）、區塊的序號（定義為大於其前驅序號的最

[6]回顧一下，TON 區塊鏈支援動態分片，因此由於分片合併和分割事件，分片設定可能會在區塊之間發生變化。因此，我們不能簡單地說每個分片鏈對應於一組固定的帳戶鏈。

[7]如果恰好有一個直接前驅（即，如果在所討論的區塊之前沒有立即發生分片鏈合併事件），則此條件適用；否則，此條件變得更加複雜。

小非負整數)、邏輯時間和生成 *unixtime*。它還包含區塊的直接前驅的雜湊(或在前面的分片鏈合併事件的情況下其兩個直接前驅的雜湊)、其初始和最終狀態的雜湊(即,處理當前區塊之前和之後的分片鏈狀態),以及生成分片鏈區塊時已知的最近主鏈區塊的雜湊。

**1.2.6. 驗證者簽章,已簽署和未簽署的區塊.** 到目前為止描述的區塊是未簽署的區塊;它由驗證者整體生成並作為一個整體考慮。當驗證者最終簽署它時,創建已簽署的區塊,由未簽署的區塊以及驗證者簽章列表(未簽署區塊的某個表示雜湊的簽章,參見 **1.1.9**)組成。此簽章列表也是(已簽署)區塊的非分割組件;然而,由於它位於未簽署區塊之外,它與區塊中保存的其他資料有所不同。

**1.2.7. 分片鏈的出站訊息佇列.** 類似地,分片鏈狀態的最重要非分割部分是 *OutMsgQueue*,即出站訊息佇列。它包含由導致此狀態的最後一個分片鏈區塊或其前驅之一包含在 *OutMsgDescr* 中的未傳遞訊息。

最初,每個出站訊息都包含在 *OutMsgQueue* 中;它僅在已包含在「相鄰」分片鏈(相對於超立方體路由的下一個)的區塊的 *InMsgDescr* 中,或已透過即時超立方體路由傳遞到(即已出現在 *InMsgDescr* 中)其最終目的地分片鏈之後,才從佇列中移除。在這兩種情況下,從 *OutMsgQueue* 中移除訊息的原因在發生這種狀態轉換的區塊的 *OutMsgDescr* 中明確說明。

**1.2.8. *InMsgDescr*、*OutMsgDescr* 和 *OutMsgQueue* 的布局.** 與訊息相關的所有最重要的非分割分片鏈資料結構都組織為雜湊映射或字典(透過序列化為單元樹的 Patricia 樹實作,如 [4, 3.3] 中所述),具有以下鍵:

- 入站訊息描述 *InMsgDescr* 使用 256 位元訊息雜湊作為鍵。
- 出站訊息描述 *OutMsgDescr* 使用 256 位元訊息雜湊作為鍵。
- 出站訊息佇列 *OutMsgQueue* 使用 32 位元目的地 *workchain_id*、目的地地址 *account_id* 的前 64 個位元和 256 位元訊息雜湊的 352 位元串接作為鍵。

**1.2.9. 區塊的分割部分:交易鏈.** 分片鏈區塊的分割部分由雜湊映射組成,該雜湊映射將分配給分片鏈的某些帳戶映射到「虛擬帳戶鏈區塊」*AccountBlock*,參見 (3)。這樣的虛擬帳戶鏈區塊由與該帳戶相關的交易的順序列表組成。

**1.2.10. 交易描述.** 每個交易在區塊中由 *Transaction* 型別的實例描述,其中特別包含以下資訊:

- 對恰好一個已被交易處理的入站訊息（也必須存在於 *InMsgDescr* 中）的參照。
- 對數個（可能為零）已被交易生成的出站訊息（也存在於 *OutMsgDescr* 中，並且很可能包含在 *OutMsgQueue* 中）的參照。

交易包括呼叫 TVM（參見 [4]），將對應於相關帳戶的智慧合約程式碼載入虛擬機，並將智慧合約的資料根單元載入虛擬機的暫存器 `c4`。入站訊息本身作為智慧合約的 `main()` 函數的引數在堆疊中傳遞，以及一些其他重要資料，例如附加到訊息的 TON Gram 和其他定義貨幣的數量、傳送者帳戶地址、智慧合約的當前餘額等等。

除了上面列出的資訊外，*Transaction* 實例還包含帳戶（即智慧合約）的原始和最終狀態，以及一些 TVM 執行統計資訊（消耗的 gas、gas 價格、執行的指令、建立/銷毀的單元、虛擬機終止碼等）。

**1.2.11. 分片鏈狀態的分割部分：帳戶狀態.** 回顧一下，根據 (3)，分片鏈狀態的分割部分由雜湊映射組成，該雜湊映射將每個「已定義」帳戶識別碼（屬於所討論的分片鏈）映射到相應帳戶的狀態，由 *AccountState* 型別的實例給出。

**1.2.12. 帳戶狀態.** 帳戶狀態本身大致由以下資料組成：

- 其在 Gram 和（可選地）在某些其他定義的加密貨幣/代幣中的餘額。
- 智慧合約程式碼，或智慧合約程式碼的雜湊，如果它將由單獨的訊息稍後提供（上傳）。
- 持久的智慧合約資料，對於簡單的智慧合約可以為空。它是一個單元樹，其根在智慧合約執行期間載入到暫存器 `c4` 中。
- 其儲存使用統計資訊，包括保存在智慧合約的持久儲存中（即在區塊鏈狀態內）的單元和位元組數量，以及上次從此帳戶收取儲存使用費的時間。
- 可選的正式介面描述（用於智慧合約）和/或使用者公開資訊（主要用於人類使用者和組織）。

請注意，在 TON 區塊鏈中，「智慧合約」和「帳戶」之間沒有區別。相反，通常由人類使用者及其加密貨幣錢包應用程式用於簡單加密貨幣轉帳的「簡單」或「錢包」帳戶，只是具有標準（共享）程式碼的簡單智慧合約，持久資料由錢包的公鑰組成（或在多重簽章錢包的情況下由數個公鑰組成；參見 **1.7.6** 以獲取更多細節）。

**1.2.13. 主鏈區塊.** 除了分片鏈區塊及其狀態之外，TON 區塊鏈還包含主鏈區塊和主鏈狀態（也稱為全域狀態）。主鏈區塊和狀態與到目前為止考慮的分片鏈區塊和狀態非常相似，但有一些顯著的差異：

- 主鏈不能分割或合併，因此主鏈區塊通常恰好有一個直接前驅。唯一的例外是「主鏈區塊零」，其特徵在於序號等於零；它根本沒有前驅，並包含整個 TON 區塊鏈的初始設定（例如，原始驗證者集合）。
- 主鏈區塊包含另一個重要的非分割結構：*ShardHashes*，一個二元樹，包含所有已定義分片鏈的列表以及列出的每個分片鏈內最新區塊的雜湊。將分片鏈區塊包含在此結構中使分片鏈區塊成為「標準」，並使其他分片鏈的區塊能夠參照分片鏈區塊中包含的資料（例如，出站訊息）。
- 主鏈的狀態包含整個 TON 區塊鏈的全域設定參數，例如最小和最大 gas 價格、TVM 的支援版本、驗證者候選人的最低質押、除 Gram 之外支援的替代加密貨幣列表、迄今發行的 Gram 總量，以及負責建立和簽署新區塊的當前驗證者集合及其公鑰。
- 主鏈的狀態還包含用於選舉後續驗證者集合和修改全域設定參數的智慧合約程式碼。這些智慧合約的程式碼本身是全域設定參數的一部分，可以相應地修改。在這方面，此程式碼（連同這些參數的當前值）的功能類似於 TON 區塊鏈的「憲法」。它最初在主鏈區塊零中建立。
- 沒有透過主鏈的轉接訊息：每個入站訊息必須在主鏈內有目的地，每個出站訊息必須在主鏈內有來源。

## 1.3 一致性條件

除了包含在區塊和區塊鏈狀態中的資料結構（根據稍後詳細解釋的某些 TL-B 方案序列化為單元袋（參見章節 **3**–**5**））之外，區塊鏈布局的一個重要組件是保存在一個區塊或不同區塊內的資料之間的一致性條件（如 **1.2.3** 中所述）。本節詳細描述一致性條件在區塊鏈中的功能。

**1.3.1. 表達一致性條件.** 原則上，依賴資料型別（例如 TL-B 中使用的那些）不僅可用於描述區塊資料的序列化，還可用於表達對此類資料型別的組件施加的條件。(例如，可以定義資料型別 *OrderedIntPair*，具有整數對 $(x, y)$ 作為值，使得 $x < y$。) 然而，TL-B 目前還不夠表達能力來編碼我們需要的所有一致性條件，因此我們在本文中選擇半形式化的方法。將來，我們可能會在合適的證明助手（如 Coq）中提出後續的完整形式化。

**1.3.2. 一致性條件的重要性.** 一致性條件最終至少與施加在其上的「不受限制」資料結構一樣重要，特別是在區塊鏈環境中。例如，一致性條件確保帳戶的狀態在區塊之間不會改變，並且它只能在區塊內由於交易而改變。這樣，一致性條件確保區塊鏈內加密貨幣餘額和其他資訊的安全儲存。

**1.3.3. 一致性條件的種類.** 對 TON 區塊鏈施加了幾種類型的一致性條件：

- 全域條件 ——表達整個 TON 區塊鏈中的不變數。例如，訊息傳遞保證斷言每個生成的訊息必須傳遞到其目的地帳戶並且恰好傳遞一次，是全域條件的一部分。
- 內部（局部）條件 ——表達對保存在一個區塊內的資料施加的條件。例如，區塊中包含的每個交易（即存在於某個帳戶的交易列表中）恰好處理一個入站訊息；此入站訊息也必須在區塊的 *InMsgDescr* 結構中列出。
- 外部（局部）條件 ——表達對不同區塊的資料施加的條件，通常屬於相同或相鄰的分片鏈（相對於超立方體路由）。因此，外部條件有幾種類型：
  - 前驅/後繼條件 ——表達對某個區塊及其直接前驅的資料或（在前面的分片鏈合併事件的情況下）兩個直接前驅的資料施加的條件。這些條件中最重要的是，陳述分片鏈區塊的初始狀態必須與直接前驅區塊的最終分片鏈狀態一致，前提是在此期間沒有發生分片鏈分割/合併事件。
  - 主鏈/分片鏈條件 ——表達對分片鏈區塊和在其 *ShardHashes* 列表中參照它的主鏈區塊或在分片鏈區塊的標頭中被參照的主鏈區塊施加的條件。
  - 相鄰（區塊）條件 ——表達相對於超立方體路由的相鄰分片鏈的區塊之間的關係。這些條件中最重要的表達區塊的 *InMsgDescr* 和相鄰區塊狀態的 *OutMsgQueue* 之間的關係。

**1.3.4. 將全域和局部條件分解為更簡單的局部條件.** 全域一致性條件，例如訊息傳遞保證，對於區塊鏈正常工作確實是必要的；然而，它們難以直接執行和驗證。因此，我們引入了許多更簡單的局部一致性條件，這些條件更容易執行和驗證，因為它們只涉及一個區塊，或可能兩個相鄰的區塊。這些局部條件的選擇方式使得所需的全域條件是（所有局部條件的合取的）邏輯結果。在這方面，我們說全域條件已「分解」為更簡單的局部條件。

有時局部條件仍然過於繁瑣而無法執行或驗證。在這種情況下，它進一步分解為更簡單的局部條件。

**1.3.5. 分解可能需要額外的資料結構和額外的內部一致性條件.** 將條件分解為更簡單的局部一致性條件有時需要引入額外的資料結構。例如，*InMsgDescr* 明確列出在區塊中處理的所有入站訊息，即使此列表可能已透過掃描區塊中存在的所有交易的列表獲得。然而，*InMsgDescr* 大大簡化了與訊息轉發和路由相關的相鄰條件，最終加起來成為全域訊息傳遞保證。

請注意，引入此類額外資料結構是一種「資料庫反正規化」（即，它導致某些冗餘，或某些資料出現不止一次），因此需要施加更多的內部一致性條件（例如，如果某些資料現在以兩個副本存在，我們必須要求這兩個副本一致）。例如，一旦我們引入 *InMsgDescr* 以促進分片鏈之間的訊息轉發，我們需要引入將 *InMsgDescr* 與同一區塊的交易列表聯繫起來的內部一致性條件。

**1.3.6. 正確序列化條件.** 除了高階內部一致性條件（將區塊的內容視為抽象資料型別的值）之外，還有一些低階內部一致性條件，稱為「(正確）序列化條件」，確保區塊中存在的單元樹確實是預期抽象資料型別的值的有效序列化。可以從描述抽象資料型別及其到單元樹的序列化的 TL-B 方案自動生成這樣的序列化條件。

請注意，序列化條件是單元或單元切片上的一組相互遞迴的謂詞。例如，如果型別 $A$ 的值由 32 位元魔術數字 $m_A$、64 位元整數 $l$ 和分別包含型別 $B$ 和 $C$ 的值的單元的兩個參照組成，則型別 $A$ 的值的正確序列化條件將要求單元或單元切片恰好包含 96 個資料位元和兩個單元參照 $r_1$ 和 $r_2$，並附加要求前 32 個資料位元包含 $m_A$，並且由 $r_1$ 和 $r_2$ 參照的兩個單元分別滿足型別 $B$ 和 $C$ 的值的序列化條件。

**1.3.7. 構造性消除存在量詞.** 人們可能想施加的局部條件有時是非構造性的，這意味著它們不一定包含它們為真的原因的解釋。這種條件 $C$ 的典型例子是

$$C :\equiv \forall_{(x:X)} \exists_{(y:Y)} A(x,y) \quad , \tag{5}$$

「對於 $X$ 中的任何 $x$，存在 $Y$ 中的 $y$ 使得條件 $A(x,y)$ 成立」。即使我們知道 $C$ 為真，對於給定的 $x : X$，我們也沒有快速找到 $y : Y$ 使得 $A(x,y)$ 的方法。因此，驗證 $C$ 可能相當耗時。

為了簡化局部條件的驗證，透過添加一些見證資料結構使它們成為可構造的（即，在有界時間內可驗證）。例如，(5) 的條件 $C$ 可以透過添加新的資料結構 $f : X \to Y$（從 $X$ 到 $Y$ 的映射 $f$）並施加以下條件 $C'$ 來轉換：

$$C' :\equiv \forall_{(x:X)} A\big(x, f(x)\big) \quad . \tag{6}$$

當然，「見證」值 $f(x) : Y$ 可以包含在（修改的）資料型別 $X$ 內，而不是保存在單獨的表 $f$ 中。

**1.3.8. 範例：*InMsgDescr* 的一致性條件.** 例如，$X := InMsgDescr$（區塊中處理的所有入站訊息的列表）和 $Y := Transactions$（區塊中存在的所有交易的列表）之間的一致性條件就是上述類型：「對於 $InMsgDescr$ 中存在的任何輸入訊息 $x$，區塊中必須存在交易 $y$ 使得 $y$ 處理 $x$」。[8] **1.3.7** 中描述的 ∃-消除過程導致我們在 $InMsgDescr$ 的入站訊息描述符中引入一個額外的欄位，包含對實際處理訊息的交易的參照。

**1.3.9. 構造性消除邏輯析取.** 與 **1.3.7** 中描述的轉換類似，條件

$$D :\equiv \forall_{(x:X)}\big(A_1(x) \vee A_2(x)\big) \quad , \tag{7}$$

「對於 $X$ 中的所有 $x$，$A_1(x)$ 和 $A_2(x)$ 中至少有一個成立」，可以轉換為函數 $i : X \to \mathbf{2} = \{1, 2\}$ 和新條件

$$D' :\equiv \forall_{(x:X)} A_{i(x)}(x) \tag{8}$$

這是先前考慮的存在量詞消除的特殊情況，其中 $Y = \mathbf{2} = \{1, 2\}$。當 $A_1(x)$ 和 $A_2(x)$ 是無法快速驗證的複雜條件時，這可能很有用，因此提前知道它們中的哪一個實際上為真是有用的。

例如，如 **1.3.8** 中所考慮的，$InMsgDescr$ 可以包含在區塊中處理的訊息和轉接訊息。我們可以在入站訊息描述中引入一個欄位來指示訊息是否為轉接訊息，並且在後一種情況下，包含處理訊息的交易的見證欄位。

**1.3.10. 條件的構造化.** 透過引入額外的資料結構和欄位來消除非構造性邏輯繫結器 ∃（存在量詞）和（有時）∨（邏輯析取）的過程——即使條件可構造的過程——將被稱為構造化。如果將其推向理論極限，此過程將導致僅包含全稱量詞和邏輯合取的邏輯公式，代價是在某些資料結構中添加一些見證欄位。

**1.3.11. 區塊的有效性條件.** 最終，區塊的所有內部條件，以及涉及此區塊和另一個先前生成的區塊的局部前驅和相鄰條件，構成分片鏈或主鏈區塊的有效性條件。如果區塊滿足有效性條件，則該區塊是有效的。驗證者的責任是生成有效區塊，以及檢查其他驗證者生成的區塊的有效性。

[8] 這個例子有點簡化，因為它沒有考慮 $InMsgDescr$ 中轉接訊息的存在，這些訊息不由任何明確的交易處理。

**1.3.12. 區塊無效性的見證.** 如果區塊不滿足所有有效性條件 $C_1$、…、$C_n$（即有效性條件的合取 $V :\equiv \bigwedge_i C_i$），則它是無效的。這意味著它滿足「無效性條件」$\neg V = \bigvee_i \neg C_i$。如果所有 $C_i$——因此也包括 $V$——都已在 **1.3.10** 中描述的意義上「構造化」，使得它們僅包含邏輯合取和全稱量詞（以及簡單的原子命題），則 $\neg V$ 僅包含邏輯析取和存在量詞。然後可以定義 $\neg V$ 的構造化，這將涉及無效性見證，從失敗的特定有效性條件 $C_i$ 的索引 $i$ 開始。

這樣的無效性見證也可以序列化並呈現給其他驗證者或提交到主鏈，以證明特定區塊或候選區塊實際上是無效的。因此，無效性見證的建構和序列化是權益證明（PoS）區塊鏈設計的重要部分。[9]

**1.3.13. 最小化見證的大小.** 局部條件的設計、它們分解為更簡單條件以及它們的構造化的一個重要考慮是使每個條件的驗證盡可能簡單。然而，另一個要求是我們應該最小化條件的見證（使得區塊大小在構造化過程中不會增長太多）和其否定的見證（使得無效性證明具有有界大小，這簡化了它們的驗證、傳輸和包含到主鏈中）的大小。這兩個設計原則有時是矛盾的，必須尋求妥協。

**1.3.14. 最小化梅克證明的大小.** 一致性條件最初旨在由已經擁有所有相關資料（例如，條件中提到的所有區塊）的一方處理。然而，在某些情況下，它們必須由不擁有所有相關區塊的一方驗證，但只知道它們的雜湊。例如，假設區塊無效性證明被簽署了無效區塊的驗證者的簽章所增強（因此必須受到懲罰）。在這種情況下，簽章將僅包含錯誤簽署的區塊的雜湊；在驗證區塊無效性證明之前，必須從其他地方恢復區塊本身。

在僅提供假定無效區塊的雜湊和提供整個無效區塊以及無效性見證之間的折衷是透過從區塊的雜湊（即區塊的根單元的雜湊）開始的梅克證明來增強無效性見證。這樣的證明將包括無效性見證中參照的所有單元，以及從這些單元到根單元的路徑上的所有單元以及它們的兄弟單元的雜湊。然後，無效性證明變得足夠自包含，可以單獨提供足夠的理由來懲罰驗證者。例如，上面建議的無效性證明可以呈現給駐留在主鏈中的智慧合約，該合約懲罰驗證者的不正確行為。

由於這樣的無效性證明必須由梅克證明增強，因此編寫一致性條件使得它們的否定的梅克證明盡可能小是有意義的。特別是，每個單獨的條件必須盡可能「局部」（即，涉及最少數量的單元）。這也最佳化了無效性證明的驗證時間。

**1.3.15. 外部條件的整理資料.** 當驗證者向分片鏈的其他驗證者建議未簽署的區塊時，這些其他驗證者必須檢查此候選區塊的有效性——即驗證它滿

[9]有趣的是，這部分工作幾乎可以自動完成。

足所有內部和外部局部一致性條件。雖然內部條件不需要除候選區塊本身之外的任何額外資料，但外部條件需要一些其他區塊，或至少需要這些區塊的一些資訊。可以從這些區塊中提取此類額外資訊，以及從包含所需額外資訊的單元到相應區塊的根單元的路徑上的所有單元以及這些路徑上的單元的兄弟單元的雜湊，以呈現可以在不知道被參照的區塊本身的情況下處理的梅克證明。

這些額外資訊稱為整理資料，序列化為單元袋，並由驗證者與未簽署的候選區塊本身一起呈現。候選區塊與整理資料一起稱為整理區塊。

**1.3.16. 整理區塊的條件.** 因此，候選區塊的外部一致性條件（自動）轉換為整理區塊的內部一致性條件，這大大簡化並加快了其他驗證者的驗證。然而，某些資料——例如正在驗證的區塊的直接前驅的最終狀態——不會被整理。相反，所有驗證者都應該保留此資料的本機副本。

**1.3.17. 表示條件和表示雜湊.** 請注意，一旦梅克證明包含在整理區塊中，一致性條件必須考慮哪些資料（即哪些單元）實際上存在於整理區塊中，而不僅僅是透過它們的雜湊參照。這導致了一組新的條件，稱為表示條件，必須能夠區分外部單元參照（通常由其 256 位元雜湊表示）和單元本身。驗證者可能會因為建議不包含所有預期整理資料的整理區塊而受到懲罰，即使候選區塊本身是有效的。

這也導致使用表示雜湊而不是透明雜湊來處理整理區塊。

**1.3.18. 在沒有整理資料的情況下驗證.** 請注意，區塊仍然必須在沒有整理資料的情況下可驗證；否則，除驗證者之外沒有任何一方能夠透過自己的方式檢查先前提交的區塊。特別是，見證不能包含在整理資料中：它們必須駐留在區塊本身中。整理資料必須僅包含在主區塊中參照的相鄰區塊的某些部分以及合適的梅克證明，這些證明可以由擁有被參照的區塊本身的任何人重建。

**1.3.19. 將梅克證明包含在區塊本身中.** 請注意，在某些情況下，梅克證明必須嵌入區塊本身，而不僅僅是嵌入整理資料。例如：

- 在即時超立方體路由（IHR）期間，訊息可以直接包含在目的地分片鏈的區塊的 *InMsgDescr* 中，而無需沿著超立方體的邊緣一直行進。在這種情況下，訊息在發起分片鏈的區塊的 *OutMsgDescr* 中存在的梅克證明必須與訊息本身一起包含在 *InMsgDescr* 中。

- 無效性證明或驗證者不當行為的另一個證明可以透過將其包含在傳送給特殊智慧合約的訊息的主體中而提交到主鏈。在這種情況下，無效性證明必須包含一些單元以及梅克證明，因此必須包含在訊息主體中。

- 同樣，定義支付通道或另一種側鏈的智慧合約可以接受包含合適的梅克證明的終局化訊息或不當行為證明訊息。
- 分片鏈的最終狀態不包含在分片鏈區塊中。相反，僅包含已修改的單元；從舊狀態繼承的那些單元透過它們的雜湊參照，以及由從舊狀態的根到所參照的舊狀態的單元的路徑上的單元組成的合適的梅克證明。

**1.3.20. 處理不完整資料的規定.** 如我們所見，有必要將不完整的資料和梅克證明包含到區塊的主體中、包含在區塊中的某些訊息的主體中以及狀態中。這種必要性透過一些額外的表示條件以及訊息（以及透過擴展，TVM 處理的單元樹）包含不完整資料（外部單元參照和梅克證明）的規定來反映。在大多數情況下，這樣的外部單元參照僅包含單元的 256 位元 SHA256 雜湊以及一個旗標；如果智慧合約嘗試透過 `CTOS` 原語（例如，用於反序列化）檢查此類單元的內容，則會觸發異常。然而，對此類單元的外部參照可以儲存到智慧合約的持久儲存中，並且可以計算此類單元的透明和表示雜湊。

## 1.4 邏輯時間和邏輯時間區間

本節更仔細地查看所謂的邏輯時間，在 TON 區塊鏈中廣泛用於訊息轉發和訊息傳遞保證等用途。

**1.4.1. 邏輯時間.** TON 區塊鏈的一個組件，也在訊息傳遞中扮演重要角色的是邏輯時間，通常用 Lt 表示。它是一個非負 64 位元整數，大致如下分配給某些事件：

> 如果事件 $e$ 在邏輯上依賴於事件 $e_1$、…、$e_n$，則 $\mathrm{Lt}(e)$ 是大於所有 $\mathrm{Lt}(e_i)$ 的最小非負整數。

特別地，如果 $n = 0$(即，如果 $e$ 不依賴於任何先前的事件)，則 $\mathrm{Lt}(e) = 0$。

**1.4.2. 邏輯時間的放寬變體.** 在某些情況下，我們放寬邏輯時間的定義，僅要求

$$\mathrm{Lt}(e) > \mathrm{Lt}(e') \quad \text{當 } e \succ e' \text{ 時（即，} e \text{ 在邏輯上依賴於 } e'\text{），} \tag{9}$$

而不堅持 $\mathrm{Lt}(e)$ 是具有此屬性的最小非負整數。在這種情況下，我們可以談論放寬的邏輯時間，與上面定義的嚴格邏輯時間（參見 **1.4.1**）相對。然而，請注意，條件 (9) 是邏輯時間的基本屬性，不能進一步放寬。

**1.4.3. 邏輯時間區間.** 為某些事件或事件集合 $C$ 分配邏輯時間的區間 $\textsc{Lt}^\bullet(C) = [\textsc{Lt}^-(C), \textsc{Lt}^+(C))$ 是有意義的，這意味著事件集合 $C$ 在指定的邏輯時間「區間」中發生，其中 $\textsc{Lt}^-(C) < \textsc{Lt}^+(C)$ 是某些整數（實際上是 64 位元整數）。在這種情況下，我們可以說 $C$ 在邏輯時間 $\textsc{Lt}^-(C)$ 開始，並在邏輯時間 $\textsc{Lt}^+(C)$ 結束。

預設情況下，對於簡單或「原子」事件，我們假設 $\textsc{Lt}^+(e) = \textsc{Lt}(e) + 1$ 和 $\textsc{Lt}^-(e) = \textsc{Lt}(e)$，假設它們恰好持續一個邏輯時間單位。一般來說，如果我們有單個值 $\textsc{Lt}(C)$ 以及邏輯時間區間 $\textsc{Lt}^\bullet(C) = [\textsc{Lt}^-(C), \textsc{Lt}^+(C))$，我們總是要求

$$\textsc{Lt}(C) \in [\textsc{Lt}^-(C), \textsc{Lt}^+(C)) \tag{10}$$

或等效地，

$$\textsc{Lt}^-(C) \leq \textsc{Lt}(C) < \textsc{Lt}^+(C) \tag{11}$$

在大多數情況下，我們選擇 $\textsc{Lt}(C) = \textsc{Lt}^-(C)$。

**1.4.4. 邏輯時間區間的要求.** 邏輯時間區間的三個主要要求是：

- $0 \leq \textsc{Lt}^-(C) < \textsc{Lt}^+(C)$ 對於任何事件集合 $C$ 都是非負整數。
- 如果 $e' \prec e$（即，如果原子事件 $e$ 在邏輯上依賴於另一個原子事件 $e'$），則 $\textsc{Lt}^\bullet(e') < \textsc{Lt}^\bullet(e)$（即，$\textsc{Lt}^+(e') \leq \textsc{Lt}^-(e)$）。
- 如果 $C \supset D$（即，如果事件集合 $C$ 包含另一個事件集合 $D$），則 $\textsc{Lt}^\bullet(C) \supset \textsc{Lt}^\bullet(D)$，即

  $$\textsc{Lt}^-(C) \leq \textsc{Lt}^-(D) < \textsc{Lt}^+(D) \leq \textsc{Lt}^+(C) \tag{12}$$

  特別地，如果 $C$ 由原子事件 $e_1$、…、$e_n$ 組成，則 $\textsc{Lt}^-(C) \leq \inf_i \textsc{Lt}^-(e_i) \leq \inf_i \textsc{Lt}(e_i)$ 和 $\textsc{Lt}^+(C) \geq \sup_i \textsc{Lt}^+(e_i) \geq 1 + \sup_i \textsc{Lt}(e_i)$。

**1.4.5. 嚴格或最小的邏輯時間區間.** 可以為任何由因果關係（偏序）$\prec$ 相關的有限原子事件集合 $E = \{e\}$ 和所有子集 $C \subset E$ 分配最小邏輯時間區間。也就是說，在滿足 **1.4.4** 中列出的條件的所有邏輯時間區間分配中，我們選擇使所有 $\textsc{Lt}^+(C) - \textsc{Lt}^-(C)$ 盡可能小的那個，如果存在具有此屬性的數個分配，我們選擇也具有最小 $\textsc{Lt}^-(C)$ 的那個。

可以透過首先如 **1.4.1** 中所述為所有原子事件 $e \in E$ 分配邏輯時間 $\textsc{Lt}(e)$，然後為任何 $C \subset E$ 設定 $\textsc{Lt}^-(C) := \inf_{e \in C} \textsc{Lt}(e)$ 和 $\textsc{Lt}^+(C) := 1 + \sup_{e \in C} \textsc{Lt}(e)$ 來實現這樣的分配。

在大多數情況下，當我們需要分配邏輯時間區間時，我們使用剛剛描述的最小邏輯時間區間。

**1.4.6. TON 區塊鏈中的邏輯時間.** TON 區塊鏈為其數個組件分配邏輯時間和邏輯時間區間。

例如，在交易中建立的每個出站訊息都被分配其邏輯建立時間；為此目的，出站訊息的建立被視為原子事件，在邏輯上依賴於由同一交易建立的先前訊息，以及同一帳戶的先前交易，以及由同一交易處理的入站訊息，以及包含在具有同一交易的區塊中透過雜湊包含的區塊中包含的所有事件。因此，由同一智慧合約建立的出站訊息具有嚴格遞增的邏輯建立時間。交易本身被視為原子事件的集合，並被分配邏輯時間區間（參見 **4.2.1** 以獲取更精確的描述）。

每個區塊都是交易和訊息建立事件的集合，因此被分配邏輯時間區間，在區塊的標頭中明確提及。

## 1.5 總區塊鏈狀態

本節討論 TON 區塊鏈的總狀態，以及單獨的分片鏈和主鏈的狀態。例如，相鄰分片鏈的狀態的精確定義對於正確形式化一致性條件至關重要，該條件斷言分片鏈的驗證者必須從所有相鄰分片鏈的狀態中取得的 $OutMsgQueue$ 的聯合中匯入最舊的訊息（參見 **2.2.5**）。

**1.5.1. 由主鏈區塊定義的總狀態.** 每個主鏈區塊都包含所有當前活躍分片的列表以及每個分片的最新區塊。在這方面，每個主鏈區塊定義 $TON$ 區塊鏈的相應總狀態，因為它固定每個分片鏈的狀態以及主鏈的狀態。

對所有分片鏈區塊的最新區塊列表施加的一個重要要求是，如果主鏈區塊 $B$ 將 $S$ 列為某個分片鏈的最新區塊，並且更新的主鏈區塊 $B'$（以 $B$ 為其前驅之一）將 $S'$ 列為同一分片鏈的最新區塊，則 $S$ 必須是 $S'$ 的前驅之一。[10] 此條件使由後續主鏈區塊 $B'$ 定義的 TON 區塊鏈的總狀態與由先前區塊 $B$ 定義的總狀態相容。

**1.5.2. 由分片鏈區塊定義的總狀態.** 每個分片鏈區塊在其標頭中包含最近的主鏈區塊的雜湊。因此，在該主鏈區塊中參照的所有區塊以及它們的前驅都被視為對分片鏈區塊「已知」或「可見」，並且沒有其他區塊對它可見，唯一的例外是其在其自己的分片鏈內的前驅。

特別是，當我們說區塊必須在其 $InMsgDescr$ 中匯入來自所有相鄰分片鏈的狀態的 $OutMsgQueue$ 的訊息時，這意味著必須考慮對該區塊可見的其他分片鏈的區塊，同時區塊不能包含來自「不可見」區塊的訊息，即使它們在其他方面是正確的。

[10] 為了在動態分片存在的情況下正確表達此條件，應該固定某個帳戶 $\xi$，並考慮在 $B$ 和 $B'$ 的分片設定中包含 $\xi$ 的分片鏈的最新區塊 $S$ 和 $S'$，因為在 $B$ 和 $B'$ 中包含 $\xi$ 的分片可能不同。

## 1.6 可設定參數和智慧合約

回顧一下，TON 區塊鏈有數個所謂的「可設定參數」（參見 [3]），它們是某些值或駐留在主鏈中的某些智慧合約。本節討論這些可設定參數的儲存和存取。

**1.6.1. 可設定參數的範例.** 由可設定參數控制的區塊鏈的屬性包括：

- 驗證者的最低質押。
- 選舉的驗證者組的最大規模。
- 同一組驗證者負責的區塊的最大數量。
- 驗證者選舉過程。
- 驗證者懲罰過程。
- 當前活躍和下一個選舉的驗證者集合。
- 變更可設定參數的過程，以及負責保存可設定參數的值和修改其值的智慧合約 $\gamma$ 的地址。

**1.6.2. 可設定參數值的位置.** 可設定參數保存在駐留在 TON 區塊鏈主鏈中的特殊設定智慧合約 $\gamma$ 的持久資料中。更準確地說，該智慧合約的持久資料的根單元的第一個參照是一個字典，將 64 位元鍵（參數編號）映射到相應參數的值；每個值根據該值的型別序列化為單元切片。如果值是「智慧合約」（必須駐留在主鏈中），則使用其 256 位元帳戶地址。

**1.6.3. 透過主鏈區塊的標頭快速存取.** 為了簡化對可設定參數當前值的存取，並縮短包含對它們的參照的梅克證明，每個主鏈區塊的標頭包含智慧合約 $\gamma$ 的地址。它還包含對包含所有可設定參數值的字典的直接單元參照，該字典位於 $\gamma$ 的持久資料中。額外的一致性條件確保此參照與透過檢查智慧合約 $\gamma$ 的最終狀態獲得的參照一致。

**1.6.4. 透過 get 方法取得可設定參數的值.** 設定智慧合約 $\gamma$ 透過「get 方法」提供對某些可設定參數的存取。智慧合約的這些特殊方法不會更改其狀態，而是在 TVM 堆疊中回傳所需的資料。

**1.6.5. 透過 get 訊息取得可設定參數的值.** 類似地，設定智慧合約 $\gamma$ 可以定義一些「普通」方法（即特殊的入站訊息）來請求某些設定參數的值，這些值將在處理此類入站訊息的交易生成的出站訊息中傳送。這對於需要知道某些設定參數值的其他基本智慧合約可能很有用。

**1.6.6. 透過 get 方法獲得的值可能與透過區塊標頭獲得的值不同.** 請注意，設定智慧合約 $\gamma$ 的狀態，包括可設定參數的值，可能會在主鏈區塊內多次更改，如果該區塊中有數個由 $\gamma$ 處理的交易。因此，透過呼叫 $\gamma$ 的 get 方法或向 $\gamma$ 傳送 get 訊息獲得的值，可能與透過檢查區塊標頭中的參照（參見 **1.6.3**）獲得的值不同，該參照參照區塊中可設定參數的最終狀態。

**1.6.7. 更改可設定參數的值.** 更改可設定參數值的過程在智慧合約 $\gamma$ 的程式碼中定義。對於大多數稱為普通的可設定參數，任何驗證者都可以透過向 $\gamma$ 傳送帶有參數編號及其建議值的特殊訊息來建議新值。如果建議的值有效，則智慧合約將收集來自驗證者的進一步投票訊息，如果當前和下一組驗證者中的三分之二以上支援該提案，則更改該值。

某些參數，例如當前的驗證者集合，不能以這種方式更改。相反，當前設定包含一個帶有負責選舉下一組驗證者的智慧合約 $\nu$ 的地址的參數，智慧合約 $\gamma$ 僅接受來自此智慧合約 $\nu$ 的訊息以修改包含當前驗證者集合的設定參數的值。

**1.6.8. 更改驗證者選舉過程.** 如果驗證者選舉過程需要更改，可以透過首先將新的驗證者選舉智慧合約提交到主鏈，然後更改包含驗證者選舉智慧合約地址 $\nu$ 的普通可設定參數來實現。這將需要三分之二的驗證者在如上 **1.6.7** 中所述的投票中接受提案。

**1.6.9. 更改變更可設定參數的過程.** 類似地，設定智慧合約本身的地址是可設定參數，並可以以這種方式更改。這樣，TON 區塊鏈的大多數基本參數和智慧合約可以在驗證者的合格多數同意的任何方向上修改。

**1.6.10. 可設定參數的初始值.** 大多數可設定參數的初始值出現在主鏈的區塊零中，作為主鏈初始狀態的一部分，在此區塊中明確存在，沒有遺漏。所有基本智慧合約的程式碼也存在於初始狀態中。這樣，TON 區塊鏈的原始「憲法」和設定，包括原始驗證者集合，在區塊零中明確說明。

## 1.7 新智慧合約及其地址

本節討論新智慧合約的建立和初始化——特別是它們的初始程式碼、持久資料和餘額的來源。它還討論為新智慧合約分配帳戶地址。

**1.7.1. 僅對主鏈和基礎工作鏈有效的描述.** 本節中描述的建立新智慧合約和分配其地址的機制僅對基礎工作鏈和主鏈有效。其他工作鏈可能定義自己的機制來處理這些問題。

**1.7.2. 將加密貨幣轉移到未初始化的帳戶.** 首先，可以向先前未提及的帳戶傳送訊息，包括承載價值的訊息。如果入站訊息到達分片鏈，目的地地址 $\eta$ 對應於未定義的帳戶，則它由交易處理，就好像智慧合約的程式碼是空的（即，由隱含的 RET 組成）。如果訊息承載價值，這將導致建立「未初始化的帳戶」，該帳戶可能具有非零餘額（如果承載價值的訊息已傳送給它），[11] 但沒有程式碼和資料。因為即使是未初始化的帳戶也佔用一些持久儲存（需要保存其餘額），所以會不時從帳戶的餘額中收取一些小的持久儲存費用，直到它變為負數。

**1.7.3. 透過建構器訊息初始化智慧合約.** 帳戶或智慧合約透過向其地址 $\eta$ 傳送特殊的建構器訊息 $M$ 來建立。此類訊息的主體包含具有智慧合約初始程式碼的單元樹（在某些情況下可能被其雜湊替換），以及智慧合約的初始資料（可能為空；它可以被其雜湊替換）。建構器訊息中包含的程式碼和資料的雜湊必須與智慧合約的地址 $\eta$ 一致；否則，它將被拒絕。

在從建構器訊息的主體初始化智慧合約的程式碼和資料之後，建構器訊息的其餘部分由交易（智慧合約 $\eta$ 的建立交易）處理，透過以類似於處理普通入站訊息的方式呼叫 TVM。

**1.7.4. 智慧合約的初始餘額.** 請注意，建構器訊息通常必須承載一些價值，這些價值將轉移到新建立的智慧合約的餘額；否則，新智慧合約的餘額將為零，並且無法支付將其程式碼和資料儲存在區塊鏈中的費用。新建立的智慧合約所需的最小餘額是其使用的儲存的線性（更準確地說，仿射）函數。此函數的係數可能取決於工作鏈；特別是，它們在主鏈中高於基礎工作鏈。

**1.7.5. 透過外部建構器訊息建立智慧合約.** 在某些情況下，有必要透過不能承載任何價值的建構器訊息建立智慧合約——例如，透過「無處而來」的建構器訊息（外部入站訊息）。然後應該首先如 **1.7.2** 中所述將足夠的資金轉移到未初始化的智慧合約，然後才傳送「無處而來」的建構器訊息。

**1.7.6. 範例：建立加密貨幣錢包智慧合約.** 上述情況的一個例子由人類使用者的加密貨幣錢包應用程式提供，該應用程式必須在區塊鏈中建立特殊的錢包智慧合約以保存使用者的資金。這可以如下實現：

- 加密貨幣錢包應用程式生成新的密碼學公開/私密金鑰對（通常用於 Ed25519 橢圓曲線密碼學，由特殊 TVM 原語支援）以簽署使用者的未來交易。

[11] 設定了 bounce 旗標的承載價值的訊息將不會被未初始化的帳戶接受，而是會被「彈回」。

- 加密貨幣錢包應用程式知道要建立的智慧合約的程式碼（通常對於所有使用者都相同），以及資料，該資料通常由錢包的公鑰（或其雜湊）組成，並在一開始就生成。此資訊的雜湊是要建立的錢包智慧合約的地址 $\xi$。
- 錢包應用程式可以顯示使用者的地址 $\xi$，使用者可以開始接收到其未初始化帳戶 $\xi$ 的資金——例如，透過在交易所購買一些加密貨幣，或透過要求朋友轉移少量金額。
- 錢包應用程式可以檢查包含帳戶 $\xi$ 的分片鏈（在基礎工作鏈帳戶的情況下）或主鏈（在主鏈帳戶的情況下），無論是透過自己還是使用區塊鏈瀏覽器，並檢查 $\xi$ 的餘額。
- 如果餘額足夠，錢包應用程式可以建立並簽署（使用使用者的私鑰）建構器訊息（「無處而來」），並將其提交給相應區塊鏈的驗證者或整理者以包含在內。
- 一旦建構器訊息包含在區塊鏈的區塊中並由交易處理，錢包智慧合約最終建立。
- 當使用者想要將一些資金轉移到其他使用者或智慧合約 $\eta$，或想要向 $\eta$ 傳送承載價值的訊息時，她使用其錢包應用程式建立她想要其錢包智慧合約 $\xi$ 傳送給 $\eta$ 的訊息 $m$，將 $m$ 封裝到帶有目的地 $\xi$ 的特殊「無處而來的訊息」$m'$ 中，並使用其私鑰簽署 $m'$。必須採取一些防止重放攻擊的規定，如 **2.2.1** 中所述。
- 錢包智慧合約接收訊息 $m'$ 並在儲存在其持久資料中的公鑰的幫助下檢查簽章的有效性。如果簽章正確，它從 $m'$ 中提取嵌入的訊息 $m$ 並將其傳送到其預期的目的地 $\eta$，並附加指定數量的資金。
- 如果使用者不需要立即開始轉移資金，而只想被動地接收一些資金，她可以根據需要保持其帳戶未初始化（前提是持久儲存費用不會導致其餘額耗盡），從而最小化帳戶的儲存設定檔和持久儲存費用。
- 請注意，錢包應用程式可以為人類使用者創造資金保存在應用程式本身中的錯覺，並提供介面來「直接」從使用者的帳戶 $\xi$ 轉移資金或傳送任意訊息。實際上，所有這些操作都將由使用者的錢包智慧合約執行，該智慧合約有效地充當此類請求的代理。我們看到加密貨幣錢包是混合應用程式的簡單例子，具有鏈上部分（錢包智慧合約，用作出站訊息的代理）和鏈下部分（在使用者裝置上執行並保存私密帳戶金鑰的外部錢包應用程式）。

當然，這只是處理最簡單的使用者錢包智慧合約的一種方式。可以建立多重簽章錢包智慧合約，或為其每個單獨使用者建立具有內部餘額的共享錢包等等。

**1.7.7. 智慧合約可以由其他智慧合約建立.** 請注意，智慧合約可以在處理任何交易時生成和傳送建構器訊息。這樣，如果需要，智慧合約可以自動建立新的智慧合約，而無需任何人類干預。

**1.7.8. 智慧合約可以由錢包智慧合約建立.** 另一方面，使用者可以編譯其新智慧合約 $\nu$ 的程式碼，生成相應的建構器訊息 $m$，並使用錢包應用程式強制其錢包智慧合約 $\xi$ 向 $\nu$ 傳送訊息 $m$ 並附加足夠數量的資金，從而建立新的智慧合約 $\nu$。

## 1.8 智慧合約的修改和刪除

本節解釋如何更改智慧合約的程式碼和狀態，以及如何和何時可以銷毀智慧合約。

**1.8.1. 修改智慧合約的資料.** 智慧合約的持久資料通常是在處理交易時在 TVM 中執行智慧合約的程式碼而修改的，該交易由智慧合約的入站訊息觸發。更具體地說，智慧合約的程式碼透過 TVM 控制暫存器 `c4` 存取智慧合約的舊持久儲存，並可以在正常終止之前透過將另一個值儲存到 `c4` 中來修改持久儲存。

通常，沒有其他方法可以修改現有智慧合約的資料。如果智慧合約的程式碼不提供任何修改持久資料的方法（例如，如果它是如 **1.7.6** 中所述的簡單錢包智慧合約，它使用使用者的公鑰初始化持久資料並且不打算更改它），那麼它將有效地不可變——除非首先修改智慧合約的程式碼。

**1.8.2. 修改智慧合約的程式碼.** 類似地，現有智慧合約的程式碼只有在當前程式碼中存在此類升級的規定時才能修改。透過呼叫 TVM 原語 `SETCODE` 來修改程式碼，該原語從 TVM 堆疊中的頂部值設定當前智慧合約的程式碼根。僅在當前交易正常終止後才應用修改。

通常，如果智慧合約的開發者希望能夠在將來升級其程式碼，她在智慧合約的原始程式碼中提供特殊的「程式碼升級方法」，該方法響應某些入站「程式碼升級」訊息呼叫 `SETCODE`，使用訊息本身中傳送的新程式碼作為 `SETCODE` 的引數。必須採取一些規定來保護智慧合約免受未經授權的程式碼替換；否則，可能會失去對智慧合約及其餘額上的資金的控制。例如，可能僅接受來自受信任來源地址的程式碼升級訊息，或者可以透過要求有效的密碼學簽章和正確的序號來保護它們。

**1.8.3. 將智慧合約的程式碼或資料保存在區塊鏈之外.** 智慧合約的程式碼或資料可以保存在區塊鏈之外，並僅由它們的雜湊表示。在這種情況下，只能處理空的入站訊息，以及在特殊欄位中攜帶智慧合約程式碼（或與處理特定訊息相關的其部分）及其資料的正確副本的訊息。這種情況的一個例子由 **1.7** 中描述的未初始化智慧合約和建構器訊息給出。

**1.8.4. 使用程式碼庫.** 某些智慧合約可能共享相同的程式碼，但使用不同的資料。這方面的一個例子是錢包智慧合約（參見 **1.7.6**），它們可能使用相同的程式碼（在由相同軟體建立的所有錢包中），但使用不同的資料（因為每個錢包必須使用自己的密碼學金鑰對）。在這種情況下，所有錢包智慧合約的程式碼最好由開發者提交到共享庫中；此庫將駐留在主鏈中，並使用特殊的「外部庫單元參照」作為每個錢包智慧合約的程式碼根（或作為該程式碼內的子樹）透過其雜湊參照。

請注意，即使庫程式碼變得不可用——例如，因為其開發者停止支付其在主鏈中的儲存費用——仍然可以使用參照此庫的智慧合約，無論是透過將庫再次提交到主鏈，還是透過在傳送給智慧合約的訊息中包含其相關部分。稍後在 **4.4.3** 中更詳細地討論此外部單元參照解析機制。

**1.8.5. 銷毀智慧合約.** 請注意，在智慧合約的餘額變為零或負數之前，無法真正銷毀智慧合約。由於收取持久儲存費用，或在傳送承載價值的出站訊息以轉移其先前餘額的幾乎所有內容之後，它可能變為負數。

例如，使用者可以決定將其錢包中的所有剩餘資金轉移到另一個錢包或智慧合約。例如，如果想要升級錢包，但錢包智慧合約沒有任何未來升級的規定，這可能很有用；然後可以簡單地建立一個新錢包並將所有資金轉移到它。

**1.8.6. 凍結帳戶.** 當帳戶的餘額在交易後變為非正數，或小於某個依賴於工作鏈的最小值時，透過用單個 32 位元組雜湊替換其所有程式碼和資料來凍結該帳戶。此雜湊之後保留一段時間（例如，幾個月）以防止透過其原始建立交易（其仍具有等於帳戶地址的正確雜湊）重新建立智慧合約，並允許其所有者透過轉移一些資金並傳送包含帳戶的程式碼和資料的訊息來重新建立帳戶，以在區塊鏈中恢復。在這方面，凍結帳戶類似於未初始化帳戶；然而，凍結帳戶的正確程式碼和資料的雜湊不一定等於帳戶地址，而是單獨保存。

請注意，凍結帳戶可能具有負餘額，表示持久儲存費用到期。在其餘額變為正數並大於規定的最小值之前，無法解凍帳戶。

# 2 訊息轉發和傳遞保證

本章討論 TON 區塊鏈內的訊息轉發，包括超立方體路由（HR）和即時超立方體路由（IHR）協議。它還描述實作訊息傳遞保證和 FIFO 排序保證所需的規定。

## 2.1 訊息地址和下一跳計算

本節解釋由 TON 區塊鏈採用的超立方體路由演算法的變體計算轉接和下一跳地址。超立方體路由協議本身，使用本節中引入的概念和下一跳地址計算演算法，在下一節中呈現。

**2.1.1. 帳戶地址.** 來源地址和目的地地址始終存在於任何訊息中。通常，它們是（完整）帳戶地址。完整帳戶地址由 $workchain\_id$（定義工作鏈的有符號 32 位元大端序整數）組成，後跟（通常）256 位元內部地址或帳戶識別碼 $account\_id$（也可以解釋為無符號大端序整數）定義所選工作鏈內的帳戶。

不同的工作鏈可能使用比主鏈（$workchain\_id = -1$）和基礎工作鏈（$workchain\_id = 0$）中使用的「標準」256 位元更短或更長的帳戶識別碼。為此，主鏈狀態包含迄今為止定義的所有工作鏈的列表，以及它們的帳戶識別碼長度。一個重要的限制是任何工作鏈的 $account\_id$ 必須至少為 64 位元長。

在下文中，為了簡單起見，我們通常僅考慮 256 位元帳戶地址的情況。對於訊息路由和分片鏈分割的目的，只有 $account\_id$ 的前 64 個位元是相關的。

**2.1.2. 訊息的來源和目的地地址.** 任何訊息都有來源地址和目的地地址。其來源地址是在處理某些交易時建立訊息的帳戶（智慧合約）的地址；來源地址不能更改或任意設定，智慧合約嚴重依賴於此屬性。相比之下，當建立訊息時，可以選擇任何格式良好的目的地地址；之後，目的地地址不能更改。

**2.1.3. 沒有來源或目的地地址的外部訊息.** 某些訊息可以沒有來源或沒有目的地地址（儘管其中至少一個必須存在），如訊息標頭中的特殊旗標所示。這些訊息是用於 TON 區塊鏈與外部世界互動的外部訊息——人類使用者及其加密錢包應用程式、鏈下和混合應用程式和服務、其他區塊鏈等等。

外部訊息永遠不會在 TON 區塊鏈內路由。相反，「無處而來的訊息」（即沒有來源地址）直接包含在目的地分片鏈區塊的 $InMsgDescr$ 中（前提是滿

足某些條件）並由該區塊中的交易處理。類似地，「無處可去的訊息」（即沒有 TON 區塊鏈目的地地址），也稱為日誌訊息，也僅存在於包含生成此類訊息的交易的區塊中。[12]

因此，外部訊息對於討論訊息路由和訊息傳遞保證幾乎無關。實際上，出站外部訊息的訊息傳遞保證是微不足道的（至多，訊息必須包含在區塊的 *LogMsg* 部分中），而對於入站外部訊息則沒有保證，因為分片鏈區塊的驗證者可以自由決定包含或忽略建議的入站外部訊息（例如，根據訊息提供的處理費用）。[13]

在下文中，我們專注於「通常」或「內部」訊息，它們同時具有來源和目的地地址。

**2.1.4. 轉接和下一跳地址.** 當訊息需要透過中間分片鏈路由才能到達其預期目的地時，除了（不可變的）來源和目的地地址之外，還為其分配轉接地址和下一跳地址。當訊息的副本駐留在轉接分片鏈內等待中繼到其下一跳時，轉接地址是其在轉接分片鏈中的中間地址，就好像屬於特殊訊息中繼智慧合約，其唯一工作是將未更改的訊息中繼到路由上的下一個分片鏈。下一跳地址是相鄰分片鏈（或在某些罕見情況下，在同一分片鏈中）中訊息需要中繼到的地址。在訊息被中繼之後，下一跳地址通常成為包含在下一個分片鏈中的訊息副本的轉接地址。

在出站訊息在分片鏈（或主鏈）中建立之後，其轉接地址立即設定為其來源地址。[14]

**2.1.5. 超立方體路由的下一跳地址計算.** TON 區塊鏈採用超立方體路由的變體。這意味著下一跳地址從轉接地址（最初等於來源地址）計算如下：

1. 轉接地址和目的地地址的（大端序有符號）32 位元 $workchain\_id$ 組件被分成 $n_1$ 位元的組（目前，$n_1 = 32$），並從左（即最高有效位元）到右掃描。如果轉接地址中的一個組與目的地地址中的相應組不同，則將轉接地址中此組的值替換為目的地地址中的值以計算下一跳地址。

[12]「無處可去的訊息」可能在其主體中有一些特殊欄位，指示其在 TON 區塊鏈之外的目的地——例如，某個其他區塊鏈中的帳戶，或 IP 地址和埠——這可能由第三方軟體適當解釋。TON 區塊鏈忽略這些欄位。

[13]繞過可能的驗證者審查的問題——例如，如果所有驗證者串謀不包含傳送給屬於某些黑名單帳戶集合的帳戶的外部訊息——在其他地方單獨處理。主要想法是驗證者可能被迫承諾在未來區塊中包含具有已知雜湊的訊息，而不知道傳送者或接收者的身份；之後當呈現具有預先同意的雜湊的訊息本身時，他們將必須遵守此承諾。

[14]然而， **2.1.11** 中描述的內部路由過程立即應用於此，這可能進一步修改轉接地址。

2. 如果轉接和目的地地址的 $workchain\_id$ 部分匹配，則對地址的 $account\_id$ 部分應用類似的過程：$account\_id$ 部分，或更確切地說是其前（最高有效）64 位元，從最高有效位元開始被分成 $n_2$ 位元的組（目前，使用 $n_2 = 4$ 位元組，對應於地址的十六進位數字），並從左開始比較。第一個不同的組在轉接地址中被其在目的地地址中的值替換以計算下一跳地址。

3. 如果轉接和目的地地址的 $account\_id$ 部分的前 64 位元也匹配，則目的地帳戶屬於當前分片鏈，並且訊息根本不應該轉發到當前分片鏈之外。相反，它必須由其內部的交易處理。

**2.1.6. 下一跳地址的符號表示.** 我們用

$$\textsc{NextHop}(\xi, \eta) \tag{13}$$

表示為當前（來源或轉接）地址 $\xi$ 和目的地地址 $\eta$ 計算的下一跳地址。

**2.1.7. 支援任播地址.** 「大型」智慧合約可以在不同的分片鏈中有單獨的實例，可以使用任播目的地地址到達。這些地址如下支援。

任播地址 $(\eta, d)$ 由通常的地址 $\eta$ 及其「分割深度」$d \leq 31$ 組成。其想法是訊息可以傳遞到僅在內部地址部分的前 $d$ 位元中與 $\eta$ 不同的任何地址（即，不包括工作鏈識別碼，它必須完全匹配）。這如下實現：

- 有效目的地地址 $\tilde{\eta}$ 從 $(\eta, d)$ 計算，透過將 $\eta$ 的內部地址部分的前 $d$ 位元替換為從來源地址 $\xi$ 中取得的相應位元。

- 所有 $\textsc{NextHop}(\nu, \eta)$ 的計算都被 $\textsc{NextHop}(\nu, \tilde{\eta})$ 替換，對於 $\nu = \xi$ 以及所有其他中間地址 $\nu$。這樣，超立方體路由或即時超立方體路由最終將訊息傳遞到包含 $\tilde{\eta}$ 的分片鏈。

- 當訊息在其目的地分片鏈（包含地址 $\tilde{\eta}$ 的分片鏈）中處理時，它可以由同一分片鏈的帳戶 $\eta'$ 處理，該帳戶僅在內部地址部分的前 $d$ 位元中與 $\eta$ 和 $\tilde{\eta}$ 不同。更準確地說，如果公共分片地址前綴是 $s$，因此只有以二進位字串 $s$ 開頭的內部地址屬於目的地分片，則 $\eta'$ 從 $\eta$ 計算，透過將 $\eta$ 的內部地址部分的前 $\min(d, |s|)$ 位元替換為 $s$ 的相應位元。

也就是說，在下面的討論中，我們默默地忽略任播地址的存在以及它們需要的額外處理。

**2.1.8. 下一跳地址演算法的漢明最優性.** 請注意， **2.1.5** 中解釋的特定超立方體路由下一跳計算演算法可能被另一個演算法替換，前提是它滿足某些屬性。這些屬性之一是漢明最優性，意味著從 $\xi$ 到 $\eta$ 的漢明（$L_1$）距離等於從 $\xi$ 到 $\textsc{NextHop}(\xi,\eta)$ 的漢明距離和從 $\textsc{NextHop}(\xi,\eta)$ 到 $\eta$ 的漢明距離之和：

$$\|\xi-\eta\|_1 = \big\|\xi-\textsc{NextHop}(\xi,\eta)\big\|_1 + \big\|\textsc{NextHop}(\xi,\eta)-\eta\big\|_1 \tag{14}$$

這裡 $\|\xi-\eta\|_1$ 是 $\xi$ 和 $\eta$ 之間的漢明距離，等於 $\xi$ 和 $\eta$ 不同的位元位置的數量：[15]

$$\|\xi-\eta\|_1 = \sum_i |\xi_i-\eta_i| \tag{15}$$

請注意，一般來說，應該期望 (14) 中僅有不等式，這遵循 $L_1$-度量的三角不等式。漢明最優性本質上意味著 $\textsc{NextHop}(\xi,\eta)$ 位於從 $\xi$ 到 $\eta$ 的（漢明）最短路徑之一上。它也可以透過說 $\nu=\textsc{NextHop}(\xi,\eta)$ 總是透過將某些位置的位元的值更改為它們在 $\eta$ 中的值從 $\xi$ 獲得來表達：對於任何位元位置 $i$，我們有 $\nu_i=\xi_i$ 或 $\nu_i=\eta_i$。[16]

**2.1.9. NextHop 的非停止性.** NextHop 的另一個重要屬性是其非停止性，意味著 $\textsc{NextHop}(\xi,\eta)=\xi$ 只有在 $\xi=\eta$ 時才可能。換句話說，如果我們還沒有到達 $\eta$，下一跳不能與我們的當前位置一致。

此屬性意味著從 $\xi$ 到 $\eta$ 的路徑——即中間地址的序列 $\xi^{(0)}:=\xi$、$\xi^{(n)}:=\textsc{NextHop}(\xi^{(n-1)},\eta)$——將逐漸穩定在 $\eta$：對於某個 $N\geq 0$，對於所有 $n\geq N$，我們有 $\xi^{(n)}=\eta$。實際上，總是可以取 $N:=\|\xi-\eta\|_1$。

**2.1.10. HR 路徑相對於分片的凸性.** 漢明最優性屬性 (14) 的一個結果是我們稱之為從 $\xi$ 到 $\eta$ 的路徑相對於分片的凸性。也就是說，如果 $\xi^{(0)}:=\xi$、$\xi^{(n)}:=\textsc{NextHop}(\xi^{(n-1)},\eta)$ 是從 $\xi$ 到 $\eta$ 的計算路徑，$N$ 是使得 $\xi^{(N)}=\eta$ 的第一個索引，$S$ 是任何分片設定中某個工作鏈的分片，則 $\xi^{(i)}$ 駐留在分片 $S$ 中的索引 $i$ 構成 $[0,N]$ 中的子區間。換句話說，如果整數 $0\leq i\leq j\leq k\leq N$ 使得 $\xi^{(i)}$、$\xi^{(k)}\in S$，則 $\xi^{(j)}\in S$ 也成立。

這種凸性屬性對於與動態分片存在時的訊息轉發相關的某些證明很重要。

[15]當涉及的地址具有不同的長度時（例如，因為它們屬於不同的工作鏈），應該在上述公式中僅考慮地址的前 96 個位元。

[16]我們可能考慮 *Kademlia* 最優性的等效屬性，而不是漢明最優性，針對由 $\|\xi-\eta\|_K:=\sum_i 2^{-i}|\xi_i-\eta_i|$ 給出的 Kademlia（或加權 $L_1$）距離而不是漢明距離編寫。

**2.1.11. 內部路由.** 請注意，根據 **2.1.5** 中定義的規則計算的下一跳地址可能屬於與當前分片鏈（即包含轉接地址的分片鏈）相同的分片鏈。在這種情況下，「內部路由」立即發生，轉接地址被計算的下一跳地址的值替換，並重複下一跳地址計算步驟，直到獲得位於當前分片鏈之外的下一跳地址。然後根據其計算的下一跳地址將訊息保留在轉接輸出佇列中，其最後計算的轉接地址作為轉接訊息的「中間擁有者」。如果當前分片鏈在訊息進一步轉發之前分割為兩個分片鏈，則包含中間擁有者的分片鏈繼承此轉接訊息。

或者，我們可能繼續計算下一跳地址，只是為了發現目的地地址已經屬於當前分片鏈。在這種情況下，訊息將在此分片鏈內由（交易）處理，而不是進一步轉發。

**2.1.12. 相鄰分片鏈.** 如果分片設定中的兩個分片——或兩個相應的分片鏈——其中一個包含至少一個允許的來源和目的地地址組合的下一跳地址，而另一個包含同一組合的轉接地址，則稱它們為鄰居或相鄰分片鏈。換句話說，如果訊息可以透過超立方體路由從其中一個直接轉發到另一個，則兩個分片鏈是鄰居。

主鏈也包含在此定義中，就好像它是 $workchain\_id = -1$ 的工作鏈的唯一分片鏈。在這方面，它是所有其他分片鏈的鄰居。

**2.1.13. 任何分片都是其自身的鄰居.** 請注意，分片鏈始終被視為其自身的鄰居。這似乎是多餘的，因為我們總是重複 **2.1.5** 中描述的下一跳計算，直到我們獲得當前分片鏈之外的下一跳地址（參見 **2.1.11**）。然而，這種安排至少有兩個原因：

- 某些訊息在同一分片鏈內具有來源和目的地地址，至少在建立訊息時是這樣。然而，如果此類訊息未在建立它的同一區塊中立即處理，則必須將其添加到其分片鏈的出站訊息佇列中，並作為入站訊息（在 *InMsgDescr* 中有條目）匯入到同一分片鏈的後續區塊之一中。[17]
- 或者，下一跳地址最初可能在某個其他分片鏈中，該分片鏈後來與當前分片鏈合併，因此下一跳變為同一分片鏈內。然後訊息將必須從合併的分片鏈的出站訊息佇列匯入，並根據其下一跳地址相應地轉發或處理，即使它們現在駐留在同一分片鏈內。

**2.1.14. 超立方體路由和 ISP.** 最終，無限分片範式（ISP）在這裡適用：分片鏈應被視為帳戶鏈的臨時聯合，僅為了最小化區塊生成和傳輸開銷而組合在一起。

[17]請注意，下一跳和內部路由計算仍然應用於此類訊息，因為在處理訊息之前可能會分割當前分片鏈。在這種情況下，包含目的地地址的新子分片鏈將繼承訊息。

訊息的轉發經過數個中間帳戶鏈，其中一些可能恰好位於同一分片中。在這種情況下，一旦訊息到達位於此分片中的帳戶鏈，它立即（「內部」）在該分片內路由，直到到達位於同一分片中的最後一個帳戶鏈（參見 **2.1.11**）。然後將訊息排入該最後一個帳戶鏈的輸出佇列。[18]

**2.1.15. 轉接和下一跳地址的表示.** 請注意，轉接和下一跳地址僅在 *workchain_id* 和帳戶地址的前（最高有效）64 位元中與來源地址不同。因此，它們可以由 96 位元字串表示。此外，它們的 *workchain_id* 通常與來源地址或目的地地址的 *workchain_id* 一致；可以使用幾個位元來指示這種情況，從而進一步減少表示轉接和下一跳地址所需的空間。

實際上，透過觀察到 **2.1.5** 中描述的特定超立方體路由演算法總是生成在其前 $k$ 位元中與目的地地址一致、在其其餘位元中與來源地址一致的中間（即轉接和下一跳）地址，可以進一步減少所需的儲存。因此，可能僅使用值 $0 \le k_{\rm tr}, k_{\rm nh} \le 96$ 來完全指定轉接和下一跳地址。還可能注意到 $k' := k_{\rm nh}$ 結果是 $k := k_{\rm tr}$ 的固定函數（例如，對於 $k \ge 32$，$k' = k + n_2 = k + 4$），因此在序列化中僅包含 $k$ 的一個 7 位元值。

這種最佳化的明顯缺點是它們過度依賴所使用的特定路由演算法，該演算法將來可能會更改，因此在 **3.1.15** 中使用它們時提供了在必要時指定更一般的中間地址的規定。

**2.1.16. 訊息封包.** 轉發訊息的轉接和下一跳地址不包含在訊息本身中，而是保存在特殊的訊息封包中，該封包是包含具有上述最佳化的轉接和下一跳地址、與轉發和處理相關的某些其他資訊以及對包含未修改的原始訊息的單元的參照的單元（或單元切片）。這樣，訊息可以輕鬆地從其原始封包（例如，*InMsgDescr* 中存在的封包）中「提取」，並放入另一個封包中（例如，在包含在 *OutMsgQueue* 中之前）。

在區塊表示為單元樹或更確切地說是 DAG 時，兩個不同的封包將包含對包含原始訊息的共享單元的參照。如果訊息很大，這種安排避免了在區塊中保留訊息的多個副本的需要。

## 2.2 超立方體路由協議

本節揭示 TON 區塊鏈採用的超立方體路由協議的細節，以實現駐留在任意分片鏈中的智慧合約之間的訊息保證傳遞。出於本文件的目的，我們將 TON 區塊鏈採用的超立方體路由變體稱為超立方體路由（HR）。

[18]我們可以將帳戶(鏈)的(虛擬)輸出佇列定義為當前包含該帳戶的分片的 *OutMsgQueue* 的子集，該子集由轉接地址等於帳戶地址的訊息組成。

**2.2.1. 訊息唯一性.** 在繼續之前，讓我們觀察到任何（內部）訊息都是唯一的。回顧一下，訊息包含其完整來源地址及其邏輯建立時間，並且由同一智慧合約建立的所有出站訊息都具有嚴格遞增的邏輯建立時間（參見 **1.4.6**）；因此，完整來源地址和邏輯建立時間的組合唯一定義訊息。由於我們假設選擇的雜湊函數 SHA256 是抗碰撞的，訊息由其雜湊唯一確定，因此如果我們知道它們的雜湊一致，我們可以識別兩個訊息。

這不擴展到「無處而來」的外部訊息，它們沒有來源地址。必須特別小心以防止與此類訊息相關的重放攻擊，特別是由使用者錢包智慧合約的設計者。一個可能的解決方案是在此類訊息的主體中包含序號，並在智慧合約持久資料中保留已處理的外部訊息的計數，如果其序號與此計數不同，則拒絕處理外部訊息。

**2.2.2. 識別具有相同雜湊的訊息.** TON 區塊鏈假設具有相同雜湊的兩個訊息一致，並將其中任何一個視為另一個的冗餘副本。如上 **2.2.1** 中所述，這不會對內部訊息產生任何意外影響。然而，如果向智慧合約傳送兩個一致的「無處而來的訊息」，可能只有其中一個會被傳遞——或兩者都會。如果它們的操作不應該是冪等的（即，如果處理訊息兩次與處理一次具有不同的效果），則應採取一些規定來區分這兩個訊息，例如透過在其中包含序號。

特別是，*InMsgDescr* 和 *OutMsgDescr* 使用（未封包的）訊息雜湊作為鍵，默默地假設不同的訊息具有不同的雜湊。這樣，可以透過在不同區塊的 *InMsgDescr* 和 *OutMsgDescr* 中查找訊息雜湊來追蹤訊息在不同分片鏈中的路徑和命運。

**2.2.3. *OutMsgQueue* 的結構.** 回顧一下，出站訊息——包括在分片鏈內建立的訊息和先前從相鄰分片鏈匯入以中繼到下一跳分片鏈的轉接訊息——累積在 *OutMsgQueue* 中，它是分片鏈的狀態的一部分（參見 **1.2.7**）。與 *InMsgDescr* 和 *OutMsgDescr* 不同，*OutMsgQueue* 中的鍵不是訊息雜湊，而是其下一跳地址——或至少其前 96 位元——與訊息雜湊串接。

此外，*OutMsgQueue* 不僅僅是字典（雜湊映射），將其鍵映射到（封包的）訊息。相反，它是相對於邏輯建立時間的最小增強字典，這意味著表示 *OutMsgQueue* 的 Patricia 樹的每個節點都有一個額外的值（在這種情況下，是無符號 64 位元整數），並且每個分叉節點中的此增強值設定為等於其子節點的增強值的最小值。葉的增強值等於該葉中包含的訊息的邏輯建立時間；它不需要明確儲存。

**2.2.4. 檢查鄰居的 *OutMsgQueue*.** *OutMsgQueue* 的這種結構使相鄰分片鏈的驗證者能夠檢查它以找到與它們相關的部分（Patricia 子樹）（即，由下一跳地址屬於所討論的相鄰分片的訊息組成——或具有給定二進位前綴

的下一跳地址），以及快速計算該部分中「最舊」（即具有最小邏輯建立時間）的訊息。

此外，分片驗證者甚至不需要追蹤所有相鄰分片鏈的總狀態——他們只需要保持和更新其 *OutMsgQueue* 的副本，或甚至僅更新與它們相關的子樹。

**2.2.5. 邏輯時間單調性：從相鄰分片匯入最舊的訊息.** 訊息轉發的第一個基本局部條件，稱為（訊息匯入）（邏輯時間）單調性條件，可概括如下：

> 在將訊息從相鄰分片鏈的 *OutMsgQueue* 匯入到分片鏈區塊的 *InMsgDescr* 時，驗證者必須按訊息的邏輯時間遞增順序匯入訊息；如果時間相同，則先匯入雜湊值較小的訊息。

更準確地說，每個分片鏈區塊都包含主鏈區塊的雜湊（假定為分片鏈區塊建立時的「最新」主鏈區塊），而主鏈區塊又包含最新分片鏈區塊的雜湊。這樣，每個分片鏈區塊間接地「知道」所有其他分片鏈的最新狀態，特別是其相鄰分片鏈，包括它們的 *OutMsgQueue*。[19]

現在，單調性條件的另一個等價表述如下：

> 如果訊息被匯入到新區塊的 *InMsgDescr* 中，其邏輯建立時間不能大於任何相鄰分片鏈最新狀態的 *OutMsgQueue* 中任何未匯入訊息的邏輯建立時間。

正是這種形式的單調性條件出現在 TON 區塊鏈區塊的局部一致性條件中，並由驗證者強制執行。

**2.2.6. 訊息匯入邏輯時間單調性條件違反的見證.** 請注意，如果不滿足此條件，可以構造一個見證其失敗的小型梅克證明。這樣的證明將包含：

- 相鄰分片鏈的 *OutMsgQueue* 中從根到某個邏輯建立時間較小的訊息 $m$ 的路徑。
- 所考慮區塊的 *InMsgDescr* 中的路徑，顯示鍵等於 $\textsc{Hash}(m)$ 在 *InMsgDescr* 中不存在（即 $m$ 未被包含在當前區塊中）。
- 使用包含匯入到區塊中的所有訊息的最小和最大邏輯時間的區塊標頭資訊，證明 $m$ 未包含在同一分片鏈的前一個區塊中（參見 **2.3.4**–**2.3.7** 了解更多資訊）。

[19]特別是，如果相鄰分片鏈最新區塊的雜湊尚未反映在最新主鏈區塊中，則不得考慮其對 *OutMsgQueue* 的修改。

- *InMsgDescr* 中到另一個包含的訊息 $m'$ 的路徑，使得 $\text{LT}(m') > \text{LT}(m)$，或 $\text{LT}(m') = \text{LT}(m)$ 且 $\text{HASH}(m') > \text{HASH}(m)$。

**2.2.7. 從 *OutMsgQueue* 刪除訊息.** 訊息必須遲早從 *OutMsgQueue* 中刪除；否則，*OutMsgQueue* 使用的儲存空間將無限增長。為此，引入了幾個「垃圾回收規則」。它們允許在評估區塊期間從 *OutMsgQueue* 刪除訊息，但前提是該區塊的 *OutMsgDescr* 中存在明確的特殊「傳遞記錄」。此記錄包含將訊息包含在其 *InMsgDescr* 中的相鄰分片鏈區塊的引用（區塊的雜湊就足夠了，但區塊的整理材料可能包含相關的梅克證明），或透過即時超立方體路由將訊息傳遞到其最終目的地的梅克證明。

**2.2.8. 透過超立方體路由保證訊息傳遞.** 這樣，除非訊息已中繼到其下一跳分片鏈或已傳遞到其最終目的地（參見 **2.2.7**），否則不能從出站訊息佇列中刪除訊息。同時，訊息匯入單調性條件（參見 **2.2.5**）確保任何訊息遲早會被中繼到下一個分片鏈，考慮到其他條件，這些條件要求驗證者至少使用區塊空間或 gas 限制的一半來匯入入站內部訊息（否則驗證者可能選擇建立空區塊或僅匯入外部訊息，即使其相鄰節點的出站訊息佇列非空）。

**2.2.9. 訊息處理順序.** 當區塊內的交易處理多個匯入的訊息時，訊息處理順序條件確保較舊的訊息先被處理。更準確地說，如果區塊包含同一帳戶的兩個交易 $t$ 和 $t'$，分別處理入站訊息 $m$ 和 $m'$，並且 $\text{LT}(m) < \text{LT}(m')$，那麼我們必須有 $\text{LT}(t) < \text{LT}(t')$。

**2.2.10. 超立方體路由的 FIFO 保證.** 訊息處理順序條件（參見 **2.2.9**）與訊息匯入單調性條件（參見 **2.2.5**）一起，意味著超立方體路由的 *FIFO* 保證。也就是說，如果智慧合約 $\xi$ 建立兩個具有相同目的地 $\eta$ 的訊息 $m$ 和 $m'$，並且 $m'$ 的生成晚於 $m$（意味著 $m \prec m'$，因此 $\text{LT}(m) < \text{LT}(m')$），則 $m$ 將在 $m'$ 之前被 $\eta$ 處理。這是因為兩條訊息都將在從 $\xi$ 到 $\eta$ 的路徑上遵循相同的路由步驟（ **2.1.5**中描述的超立方體路由演算法是確定性的），並且在所有出站佇列和入站訊息描述中，$m'$ 將出現在 $m$「之後」。[20]

如果訊息 $m'$ 可以透過即時超立方體路由傳遞到 $B$，這不一定再成立。因此，確保一對智慧合約之間 FIFO 訊息傳遞規則的一個簡單方法是在訊息標頭中設定一個特殊位元，防止其透過 IHR 傳遞。

[20]這個陳述並不像乍看起來那麼簡單，因為涉及的某些分片鏈可能在路由期間分裂或合併。正確的證明可以透過採用 **2.1.14**中解釋的 HR 的 ISP 觀點並觀察到 $m'$ 總是會在 $m$ 之後獲得，無論是在達到的中間帳戶鏈方面，或者，如果它們恰好在同一帳戶鏈中，則在邏輯建立時間方面。

一個關鍵的觀察是，「在任何給定時刻」（邏輯上；更準確的描述是「在處理任何因果閉合的區塊子集 $\mathscr{F}$ 後獲得的總狀態中」），屬於同一分片的中間帳戶鏈在從 $\xi$ 到 $\eta$ 的路徑上是連續的（即不能在它們之間有屬於其他分片的帳戶鏈）。這是 **2.1.5**中描述的超立方體路由演算法的「凸性性質」（參見 **2.1.10**）。

**2.2.11. 超立方體路由的傳遞唯一性保證.** 請注意，訊息匯入單調性條件也意味著透過超立方體路由傳遞任何訊息的唯一性——即它不能被目的地智慧合約匯入和處理多次。我們將在 **2.3**中稍後看到，當超立方體路由和即時超立方體路由都處於活動狀態時，強制執行傳遞唯一性會更複雜。

**2.2.12. 超立方體路由概述.** 讓我們總結一下傳遞由來源帳戶 $\xi_0$ 建立到目的地帳戶 $\eta$ 的內部訊息 $m$ 所執行的所有路由步驟。我們用 $\xi_{k+1} := \textsc{NextHop}(\xi_k, \eta)$，$k = 0, 1, 2, \ldots$ 表示 HR 指示的用於將訊息 $m$ 轉發到其最終目的地 $\eta$ 的中間地址。設 $S_k$ 為包含 $\xi_k$ 的分片。

- [誕生] — 具有目的地 $\eta$ 的訊息 $m$ 由駐留在某個分片鏈 $S_0$ 中的帳戶 $\xi_0$ 的交易 $t$ 建立。邏輯建立時間 $\textsc{Lt}(m)$ 在此時固定並包含在訊息 $m$ 中。

- [立即處理?] — 如果目的地 $\eta$ 駐留在同一分片鏈 $S_0$ 中，則訊息可能在生成它的同一區塊中被處理。在這種情況下，$m$ 被包含在 *OutMsgDescr* 中，帶有一個旗標，指示它已在該區塊中被處理且不需要進一步轉發。$m$ 的另一個副本包含在 *InMsgDescr* 中，以及描述入站訊息處理的通常資料。(請注意，$m$ 不包含在 $S_0$ 的 *OutMsgQueue* 中。)

- [初始內部路由] — 如果 $m$ 的目的地在 $S_0$ 之外，或未在生成它的同一區塊中處理，則應用 **2.1.11**中描述的內部路由程序，直到找到索引 $k$，使得 $\xi_k$ 位於 $S_0$ 中，但 $\xi_{k+1} = \textsc{NextHop}(\xi_k, \eta)$ 不在 (即 $S_k = S_0$，但 $S_{k+1} \neq S_0$)。或者，如果 $\xi_k = \eta$ 或 $\xi_k$ 在其前 96 位元與 $\eta$ 相同，則此過程停止。

- [出站佇列] — 訊息 $m$ 被包含在 *OutMsgDescr* 中（鍵等於其雜湊），其封包包含其轉接地址 $\xi_k$ 和下一跳地址 $\xi_{k+1}$，如 **2.1.16**和 **2.1.15**中所述。相同的封包訊息也包含在 $S_k$ 狀態的 *OutMsgQueue* 中，鍵等於其下一跳地址 $\xi_{k+1}$（如果 $\eta$ 屬於 $S_k$，則可能等於 $\eta$）的前 96 位元與訊息雜湊 $\textsc{Hash}(m)$ 的串接。

- [佇列等待] — 訊息 $m$ 在分片鏈 $S_k$ 的 *OutMsgQueue* 中等待進一步轉發。同時，分片鏈 $S_k$ 可能與其他分片鏈分裂或合併；在這種情況下，包含轉接地址 $\xi_k$ 的新分片 $S'_k$ 在其 *OutMsgQueue* 中繼承 $m$。

- [匯入入站] — 在未來的某個時刻，包含下一跳地址 $\xi_{k+1}$ 的分片鏈 $S_{k+1}$ 的驗證者掃描分片鏈 $S_k$ 狀態中的 *OutMsgQueue*，並決定根據單調性條件（參見 **2.2.5**）和其他條件匯入訊息 $m$。為分片鏈 $S_{k+1}$ 生

成一個新區塊，其 *InMsgDescr* 中包含 $m$ 的封包副本。*InMsgDescr* 中的條目還包含將 $m$ 匯入此區塊的原因，包含分片鏈 $S'_k$ 最新區塊的雜湊，以及先前的下一跳和轉接地址 $\xi_k$ 和 $\xi_{k+1}$，以便可以輕鬆定位 $S'_k$ 的 *OutMsgQueue* 中的相應條目。

- [確認] — $S_{k+1}$ 的 *InMsgDescr* 中的此條目也作為 $S'_k$ 的確認。在 $S'_k$ 的後續區塊中，必須從 $S'_k$ 的 *OutMsgQueue* 中刪除訊息 $m$；此修改反映在執行此狀態修改的 $S'_k$ 區塊的 *OutMsgDescr* 中的特殊條目中。

- [轉發?] — 如果 $m$ 的最終目的地 $\eta$ 不駐留在 $S_{k+1}$ 中，則訊息被轉發。應用超立方體路由直到獲得某個 $\xi_l$，$l > k$，和 $\xi_{l+1} = \textsc{NextHop}(\xi_l, \eta)$，使得 $\xi_l$ 位於 $S_{k+1}$ 中，但 $\xi_{l+1}$ 不在（參見 **2.1.11**）。之後，$m$ 的新封包副本（轉接地址設定為 $\xi_l$，下一跳地址設定為 $\xi_{l+1}$）被包含在 $S_{k+1}$ 當前區塊的 *OutMsgDescr* 和 $S_{k+1}$ 新狀態的 *OutMsgQueue* 中。*InMsgDescr* 中 $m$ 的條目包含一個旗標，指示訊息已被轉發；*OutMsgDescr* 中的條目包含新封包的訊息和一個旗標，指示這是轉發的訊息。然後從 [出站佇列] 開始的所有步驟都重複，用 $l$ 代替 $k$。

- [處理?] — 如果 $m$ 的最終目的地 $\eta$ 駐留在 $S_{k+1}$ 中，則匯入該訊息的 $S_{k+1}$ 區塊必須由包含在同一區塊中的交易 $t$ 處理它。在這種情況下，*InMsgDescr* 包含對 $t$ 的引用（透過其邏輯時間 $\textsc{Lt}(t)$），以及一個旗標，指示訊息已被處理。

上述訊息路由演算法未考慮實施即時超立方體路由（IHR）所需的一些進一步修改。例如，訊息可能在被匯入（列入 *InMsgDescr*）到其最終或中間分片鏈區塊後被丟棄，因為提供了透過 IHR 傳遞到最終目的地的證明。在這種情況下，必須將這樣的證明包含在 *InMsgDescr* 中，以解釋為什麼訊息未進一步轉發或處理。

## 2.3 即時超立方體路由和組合傳遞保證

本節描述即時超立方體路由協議，通常由 TON 區塊鏈與先前討論的超立方體路由協議並行應用，以實現更快的訊息傳遞。然而，當超立方體路由和即時超立方體路由並行應用於同一訊息時，實現傳遞和唯一傳遞保證會更複雜。本節還討論了這個主題。

**2.3.1. 即時超立方體路由概述.** 讓我們解釋當即時超立方體路由（IHR）機制應用於訊息時所應用的主要步驟。（請注意，通常對於同一訊息，普通 HR 和 IHR 並行工作；必須採取一些措施來保證任何訊息傳遞的唯一性。）

考慮 **2.2.12**中討論的具有來源 $\xi$ 和目的地 $\eta$ 的相同訊息 $m$ 的路由和傳遞：

- [網路傳送] — 在 $S_0$ 的驗證者就包含 $m$ 的建立交易 $t$ 的區塊達成一致並簽署後，並觀察到 $m$ 的目的地 $\eta$ 不駐留在 $S_0$ 內，他們可能會發送一個資料包（加密的網路訊息），其中包含訊息 $m$ 以及其包含在剛生成區塊的 $OutMsgDescr$ 中的梅克證明，發送到當前擁有目的地 $\eta$ 的分片鏈 $T$ 的驗證者群組。

- [網路接收] — 如果分片鏈 $T$ 的驗證者收到這樣的訊息，他們從最新的主鏈區塊和其中列出的分片鏈區塊雜湊開始檢查其有效性，包括分片鏈 $S_0$ 的最新「規範」區塊。如果訊息無效，他們會默默地丟棄它。如果分片鏈 $S_0$ 的該區塊具有比最新主鏈區塊中列出的序號更大的序號，他們可能會丟棄它或推遲驗證，直到下一個主鏈區塊出現。

- [包含條件] — 驗證者檢查訊息 $m$ 的包含條件。特別是，他們必須檢查此訊息之前未被傳遞，並且相鄰節點的 $OutMsgQueue$ 中沒有邏輯建立時間小於 $\mathrm{LT}(m)$ 的未處理出站訊息，其目的地在 $T$ 中。

- [傳遞] — 驗證者傳遞並處理訊息，方法是將其包含在當前分片鏈區塊的 $InMsgDescr$ 中，以及一個位元，指示它是 IHR 訊息、其包含在原始區塊的 $OutMsgDescr$ 中的梅克證明，以及處理此入站訊息的交易 $t'$ 的邏輯時間，該交易被包含在當前生成的區塊中。

- [確認] — 最後，驗證者向從 $\xi$ 到 $\eta$ 路徑上的中間分片鏈的所有驗證者群組發送加密資料包，其中包含訊息 $m$ 包含在其最終目的地的 $InMsgDescr$ 中的梅克證明。中間分片鏈的驗證者可以使用此證明來丟棄按照 HR 規則傳播的訊息 $m$ 的副本，方法是將訊息匯入其 $InMsgDescr$ 以及最終傳遞的梅克證明，並設定一個旗標，指示訊息已被丟棄。

整體程序甚至比超立方體路由的程序更簡單。然而請注意，IHR 不提供傳遞或 FIFO 保證：網路資料包可能在傳輸過程中丟失，或目的地分片鏈的驗證者可能決定不對其採取行動，或由於緩衝區溢位而丟棄它。這就是為什麼 IHR 被用作 HR 的補充，而不是替代品。

**2.3.2. 整體最終傳遞保證.** 請注意，HR 和 IHR 的組合保證了任何內部訊息最終傳遞到其最終目的地。實際上，HR 本身保證最終傳遞任何訊息，並且訊息 $m$ 的 HR 只能在中間階段被 $m$ 傳遞到其最終目的地（透過 IHR）的梅克證明取消。

**2.3.3. 整體唯一傳遞保證.** 然而，對於 HR 和 IHR 的組合，訊息傳遞的唯一性更難實現。特別是，必須檢查以下條件，並在必要時能夠提供它們成立或不成立的簡短梅克證明：

- 當透過 HR 將訊息 $m$ 匯入其下一個中間分片鏈區塊時，我們必須檢查 $m$ 是否尚未透過 HR 匯入。
- 當 $m$ 被匯入並在其最終目的地分片鏈中處理時，我們必須檢查 $m$ 是否尚未被處理。如果已處理，則有三種子情況：
  - 如果正在考慮透過 HR 匯入 $m$，並且它已經透過 HR 匯入，則根本不得匯入。
  - 如果正在考慮透過 HR 匯入 $m$，並且它已經透過 IHR（但不是 HR）匯入，則必須匯入並立即丟棄（不由交易處理）。這是必要的，以便從其先前中間分片鏈的 $OutMsgQueue$ 中刪除 $m$。
  - 如果正在考慮透過 IHR 匯入 $m$，並且它已經透過 IHR 或 HR 匯入，則根本不得匯入。

**2.3.4. 檢查訊息是否已傳遞到其最終目的地.** 考慮以下檢查訊息 $m$ 是否已傳遞到其最終目的地 $\eta$ 的一般演算法：可以簡單地掃描屬於包含目的地地址的分片鏈的最後幾個區塊，從最新區塊開始，透過先前區塊引用向後工作。（如果有兩個先前區塊——即如果在某個時刻發生了分片鏈合併事件——則遵循包含目的地地址的鏈。）可以檢查這些區塊中每個的 $InMsgDescr$，查找鍵為 $\textsc{Hash}(m)$ 的條目。如果找到這樣的條目，則訊息 $m$ 已被傳遞，我們可以輕鬆構造此事實的梅克證明。如果在到達具有 $\textsc{Lt}^+(B) < \textsc{Lt}(m)$ 的區塊 $B$ 之前沒有找到這樣的條目（這意味著 $m$ 不可能在 $B$ 或其任何前身中被傳遞），則訊息 $m$ 肯定尚未被傳遞。

此演算法的明顯缺點是，如果訊息 $m$ 非常舊（並且很可能很久以前就被傳遞了），意味著它具有較小的 $\textsc{Lt}(m)$ 值，則在產生答案之前需要掃描大量區塊。此外，如果答案是否定的，則此事實的梅克證明的大小將隨著掃描的區塊數量線性增加。

**2.3.5. 檢查 IHR 訊息是否已傳遞到其最終目的地.** 要檢查 IHR 訊息 $m$ 是否已傳遞到其目的地分片鏈，我們可以應用上述描述的一般演算法（參見 **2.3.4**），修改為僅檢查最後 $c$ 個區塊，其中 $c$ 是一個小常數（例如，$c = 8$）。如果在檢查這些區塊後無法得出結論，則目的地分片鏈的驗證者可以簡單地丟棄 IHR 訊息，而不是在此檢查上花費更多資源。

**2.3.6. 檢查 HR 訊息是否已透過 HR 傳遞到其最終目的地或中間分片鏈.** 要檢查 HR 接收的訊息 $m$（或更確切地說，正在考慮透過 HR 匯入的訊息 $m$）是否已透過 HR 匯入，我們可以使用以下演算法：設 $\xi_k$ 為 $m$ 的轉接地址（屬於相鄰分片鏈 $S_k$），$\xi_{k+1}$ 為其下一跳地址（屬於正在考慮的分片鏈）。由於我們正在考慮包含 $m$，$m$ 必須存在於分片鏈 $S_k$ 最新狀態的 *OutMsgQueue* 中，其封包中指示了 $\xi_k$ 和 $\xi_{k+1}$。特別是，(a) 訊息已包含在 *OutMsgQueue* 中，我們甚至可能知道何時，因為 *OutMsgQueue* 中的條目有時包含它被添加的區塊的邏輯時間，以及 (b) 它尚未從 *OutMsgQueue* 中刪除。

現在，相鄰分片鏈的驗證者被要求一旦觀察到訊息（其封包中的轉接和下一跳地址為 $\xi_k$ 和 $\xi_{k+1}$）已被匯入訊息的下一跳分片鏈的 *InMsgDescr* 中，就從 *OutMsgQueue* 中刪除訊息。因此，(b) 意味著訊息只有在這個前一個區塊非常新（即尚未被最新的相鄰分片鏈區塊知道）的情況下，才可能被匯入到前一個區塊中。因此， **2.3.4**中描述的演算法只需掃描非常有限數量的前一個區塊（通常是一個或兩個，最多）就可以得出訊息尚未被匯入的結論。[21]實際上，如果此檢查由當前分片鏈的驗證者或整理者自己執行，則可以透過在記憶體中保留最新幾個區塊的 *InMsgDescr* 來優化它。

**2.3.7. 檢查 HR 訊息是否已透過 IHR 傳遞到其最終目的地.** 最後，要檢查 HR 訊息是否已透過 IHR 傳遞到其最終目的地，可以使用 **2.3.4**中描述的一般演算法。與 **2.3.5**相反，我們不能在掃描目的地分片鏈中固定數量的最新區塊後中止驗證過程，因為 HR 訊息不能無故丟棄。

相反，我們透過禁止將 IHR 訊息 $m$ 包含在其目的地分片鏈的區塊 $B$ 中（如果目的地分片鏈中已經有超過，比如說，$c = 8$ 個區塊 $B'$ 滿足 $\text{LT}^+(B') \geq \text{LT}(m)$）來間接限制要檢查的區塊數量。

這樣的條件有效地限制了訊息 $m$ 建立後可以透過 IHR 傳遞的時間間隔，因此只需檢查目的地分片鏈的少量區塊（最多 $c$ 個）。

請注意，此條件與 **2.3.5**中描述的修改演算法很好地對齊，有效地禁止驗證者在需要超過 $c = 8$ 個步驟來檢查新接收的 IHR 訊息是否尚未匯入時匯入該訊息。

[21]不僅必須在這些區塊的 *InMsgDescr* 中查找鍵 $\text{HASH}(m)$，而且如果找到，還必須檢查相應條目的封包中的中間地址。

# 3 訊息、訊息描述符和佇列

本章介紹單個訊息、訊息描述符（如 *InMsgDescr* 或 *OutMsgDescr*）和訊息佇列（如 *OutMsgQueue*）的內部佈局。封包訊息（參見 **2.1.16**）也在此處討論。

請注意，與訊息相關的大多數一般慣例必須由所有分片鏈遵守，即使它們不屬於基礎分片鏈；否則，不同工作鏈之間的訊息傳遞和互動將不可能。訊息內容的解釋和訊息的處理（通常由某些交易處理）在工作鏈之間有所不同。

## 3.1 地址、貨幣和訊息佈局

本章從一些一般定義開始，然後是用於在訊息中序列化來源和目的地地址的地址的精確佈局。

**3.1.1. 一些標準定義.** 為了讀者方便，我們在此重現幾個一般的 TL-B 定義。[22]這些定義在下面關於地址和訊息佈局的討論中使用，但在其他方面與 TON 區塊鏈無關。

```
unit$_ = Unit;
true$_ = True;
// EMPTY False;
bool_false$0 = Bool;
bool_true$1 = Bool;
nothing$0 {X:Type} = Maybe X;
just$1 {X:Type} value:X = Maybe X;
left$0 {X:Type} {Y:Type} value:X = Either X Y;
right$1 {X:Type} {Y:Type} value:Y = Either X Y;
pair$_ {X:Type} {Y:Type} first:X second:Y = Both X Y;

bit$_ _:(## 1) = Bit;
```

**3.1.2. 地址的 TL-B 方案.** 來源和目的地地址的序列化由以下 TL-B 方案定義：

```
addr_none$00 = MsgAddressExt;
addr_extern$01 len:(## 9) external_address:(len * Bit)
```

[22]TL 的舊版本描述可以在 https://core.telegram.org/mtproto/TL 找到。或者，可以在 [4, 3.3.4] 中找到 TL-B 方案的非正式介紹。

```
              = MsgAddressExt;
anycast_info$_ depth:(## 5) rewrite_pfx:(depth * Bit) = Anycast;
addr_std$10 anycast:(Maybe Anycast)
   workchain_id:int8 address:uint256  = MsgAddressInt;
addr_var$11 anycast:(Maybe Anycast) addr_len:(## 9)
   workchain_id:int32 address:(addr_len * Bit) = MsgAddressInt;
_ MsgAddressInt = MsgAddress;
_ MsgAddressExt = MsgAddress;
```

最後兩行將類型 MsgAddress 定義為類型 MsgAddressInt 和 MsgAddressExt 的內部聯集(不要與它們的外部聯集 Either MsgAddressInt MsgAddressExt 混淆，如 **3.1.1** 中定義的)，就好像前面的四行已用右側替換為 MsgAddress 重複了一樣。這樣，類型 MsgAddress 有四個建構器，類型 MsgAddressInt 和 MsgAddressExt 都是 MsgAddress 的子類型。

**3.1.3. 外部地址.** 前兩個建構器 addr_none 和 addr_extern 用於「無處而來的訊息」（入站外部訊息）的來源地址，以及「無處而去的訊息」（出站外部訊息）的目的地地址。addr_extern 建構器定義了一個「外部地址」，TON 區塊鏈軟體完全忽略它（將 addr_extern 視為 addr_none 的較長變體），但外部軟體可能會為其自己的目的使用它。例如，特殊的外部服務可能會檢查 TON 區塊鏈所有區塊中找到的所有出站外部訊息的目的地地址，如果 external_address 欄位中存在特殊的魔術數字，則將剩餘部分解析為 IP 地址和 UDP 埠或（TON 網路）ADNL 地址，並將帶有訊息副本的資料包發送到由此獲得的網路地址。

**3.1.4. 內部地址.** 剩餘的兩個建構器 addr_std 和 addr_var 表示內部地址。其中第一個 addr_std 表示有符號的 8 位元 *workchain_id*（足夠用於主鏈和基礎工作鏈）和所選工作鏈中的 256 位元內部地址。第二個 addr_var 表示具有「大」*workchain_id* 的工作鏈中的地址，或長度不等於 256 的內部地址。這兩個建構器都有一個可選的 anycast 值，預設情況下不存在，當存在時啟用「地址重寫」。[23]

驗證者必須在可能的情況下使用 addr_std 而不是 addr_var，但必須準備好在入站訊息中接受 addr_var。addr_var 建構器用於未來的擴充。

請注意，*workchain_id* 必須是當前主鏈配置中啟用的有效工作鏈識別碼，並且內部地址的長度必須在指定工作鏈允許的範圍內。例如，不能

[23]地址重寫是一個用於實現所謂大型或全域智慧合約（參見 [3, 2.3.18]）採用的「任播地址」的功能，這些智慧合約可以在多個分片鏈中有實例。當啟用地址重寫時，訊息可能會被路由到地址與目的地地址的前 $d$ 位元一致的智慧合約並由其處理，其中 $d \leq 32$ 是 anycast.depth 欄位中指示的智慧合約的「分割深度」（參見 **2.1.7**）。否則，地址必須完全匹配。

對不完全是 256 位元長的地址使用 $workchain\_id = 0$（基礎工作鏈）或 $workchain\_id = -1$（主鏈）。

**3.1.5. 表示 Gram 貨幣金額.** Gram 的金額藉助兩種表示可變長度無符號或有符號整數的類型來表達，加上一個明確專用於表示非負 nanogram 金額的類型 `Grams`，如下所示：

```
var_uint$_ {n:#} len:(#< n) value:(uint (len * 8))
         = VarUInteger n;
var_int$_ {n:#} len:(#< n) value:(int (len * 8))
        = VarInteger n;
nanograms$_ amount:(VarUInteger 16) = Grams;
```

如果想表示 $x$ nanogram，則選擇整數 $\ell < 16$ 使得 $x < 2^{8\ell}$，並首先將 $\ell$ 序列化為無符號 4 位元整數，然後將 $x$ 本身序列化為無符號 $8\ell$ 位元整數。請注意，四個零位元表示零 Gram 金額。

回顧（參見 [3, A]）Gram 的原始總供應量固定為五十億（即 $5 \cdot 10^{18} < 2^{63}$ nanogram），並預計增長非常緩慢。因此，實際中遇到的所有 Gram 金額都將適合無符號或甚至有符號的 64 位元整數。驗證者可以在其內部計算中使用 Gram 的 64 位元整數表示；然而，這些值在區塊鏈中的序列化是另一回事。

**3.1.6. 表示任意貨幣的集合.** 回顧一下，TON 區塊鏈允許其使用者定義除 Gram 之外的任意加密貨幣或代幣，前提是滿足某些條件。此類額外的加密貨幣由 32 位元 $currency\_id$ 識別。已定義的額外加密貨幣列表是區塊鏈配置的一部分，儲存在主鏈中。

當需要表示一種或多種此類加密貨幣的一些金額時，使用字典（參見 [4, 3.3]），以 32 位元 $currency\_id$ 作為鍵，以 `VarUInteger 32` 值：

```
extra_currencies$_ dict:(HashmapE 32 (VarUInteger 32))
                 = ExtraCurrencyCollection;
currencies$_ grams:Grams other:ExtraCurrencyCollection
           = CurrencyCollection;
```

附加到內部訊息的值由 `CurrencyCollection` 類型的值表示，它可以描述一定的（非負）（nano）gram 金額以及一些額外的貨幣（如果需要）。請注意，如果不需要額外的貨幣，`other` 僅減少為一個零位元。

**3.1.7. 訊息佈局.** 訊息由其標頭後跟其主體或有效載荷組成。主體基本上是任意的，由目的地智慧合約解釋。訊息標頭是標準的，組織如下：

```
int_msg_info$0 ihr_disabled:Bool bounce:Bool
  src:MsgAddressInt dest:MsgAddressInt
  value:CurrencyCollection ihr_fee:Grams fwd_fee:Grams
  created_lt:uint64 created_at:uint32 = CommonMsgInfo;
ext_in_msg_info$10 src:MsgAddressExt dest:MsgAddressInt
  import_fee:Grams = CommonMsgInfo;
ext_out_msg_info$11 src:MsgAddressInt dest:MsgAddressExt
  created_lt:uint64 created_at:uint32 = CommonMsgInfo;

tick_tock$_ tick:Bool tock:Bool = TickTock;

_ split_depth:(Maybe (## 5)) special:(Maybe TickTock)
  code:(Maybe ^Cell) data:(Maybe ^Cell)
  library:(Maybe ^Cell) = StateInit;

message$_ {X:Type} info:CommonMsgInfo
  init:(Maybe (Either StateInit ^StateInit))
  body:(Either X ^X) = Message X;
```

此方案的含義如下。

類型 `Message` $X$ 描述具有類型 $X$ 的主體（或有效載荷）的訊息。其序列化從類型 `CommonMsgInfo` 的 `info` 開始，它有三種風格：分別用於內部訊息、入站外部訊息和出站外部訊息。它們都有來源地址 `src` 和目的地地址 `dest`，根據所選建構器，它們是外部或內部的。除此之外，內部訊息可能承載一些 Gram 和其他已定義貨幣的 `value`（參見 **3.1.6**），並且在 TON 區塊鏈內生成的所有訊息都有邏輯建立時間 `created_lt`（參見 **1.4.6**）和建立 unixtime `created_at`，兩者都由生成交易自動設定。建立 unixtime 等於包含生成交易的區塊的建立 unixtime。

**3.1.8. 轉發和 IHR 費用。內部訊息的總值.** 內部訊息定義了 Gram 的 `ihr_fee`，如果目的地分片鏈的驗證者透過 IHR 機制包含訊息，則從附加到訊息的值中扣除並授予驗證者。`fwd_fee` 是使用 HR 機制支付的原始總轉發費用；它在訊息生成時根據一些配置參數和訊息的大小自動計算。

請注意，新建立的內部出站訊息攜帶的總值等於 `value`、`ihr_fee` 和 `fwd_fee` 的總和。此總和從來源帳戶的餘額中扣除。在這些組件中，只有 `value` 在訊息傳遞時總是記入目的地帳戶。`fwd_fee` 由從來源到目的地的 HR 路徑上的驗證者收取，`ihr_fee` 要麼由目的地分片鏈的驗證者收取（如果訊息透過 IHR 傳遞），要麼記入目的地帳戶。

**3.1.9. 訊息中包含的程式碼和資料部分.** 除了儲存在 `info` 中的常見訊息資訊外，訊息還可以包含目的地智慧合約的程式碼和資料部分。例如，此功能用於所謂的建構器訊息（參見 **1.7.3**），它們只是在其 `init` 部分定義了 `code` 和可能 `data` 欄位的內部或入站外部訊息。如果這些欄位的雜湊正確，並且目的地智慧合約沒有程式碼或資料，則使用訊息中的值。[24]

**3.1.10. 將 `code` 和 `data` 用於其他目的.** 主鏈和基礎工作鏈以外的工作鏈可以自由地將 `code`、`data` 和 `library` 欄位中引用的單元樹用於自己的目的。訊息傳遞系統本身不對其內容做任何假設；它們僅在訊息由交易處理時才變得相關。

**3.1.11. 缺少明確的 gas 價格和 gas 限制.** 請注意，訊息沒有明確的 gas 價格和 gas 限制。相反，gas 價格由驗證者為每個工作鏈全域設定（它是一個特殊的可配置參數），每個交易的 gas 限制也有一個預設值，這是一個可配置參數；如果需要，智慧合約本身可以在執行期間降低 gas 限制。

對於內部訊息，初始 gas 限制不能超過訊息的 Gram 值除以當前 gas 價格。對於入站外部訊息，初始 gas 限制非常小，真正的 gas 限制由接收智慧合約本身設定，當它透過相應的 TVM 原語接受入站訊息時。

**3.1.12. 訊息有效載荷的反序列化.** 訊息的有效載荷或主體在 TVM 執行時由接收智慧合約反序列化。訊息傳遞系統本身不對有效載荷的內部格式做任何假設。然而，透過具有 32 位元建構器標籤的 TL 或 TL-B 方案描述支援的入站訊息的序列化是有意義的，這樣其他智慧合約的開發者就會知道特定智慧合約支援的介面。

訊息在區塊鏈內總是作為單元中的最後一個欄位序列化。因此，區塊鏈軟體可以假設在解析 `Message` 的 `body` 之前的欄位之後未解析的任何位元和引用都屬於有效載荷 `body` : $X$，而無需知道類型 $X$ 的序列化的任何內容。

**3.1.13. 具有空有效載荷的訊息.** 訊息的有效載荷可能恰好是空單元切片，沒有資料位元也沒有引用。按照慣例，這樣的訊息用於簡單的值轉帳。接收智慧合約通常應該安靜地處理此類訊息並成功終止（退出代碼為零），儘管某些智慧合約即使在接收具有空有效載荷的訊息時也可能執行重要操作。例如，智慧合約可能會檢查結果餘額，如果它對於先前推遲的操作變得足夠，則觸發此操作。或者，智慧合約可能希望在其持久儲存中記住收到的金額和相應的發送者，例如，以便稍後按轉帳資金的比例向每個發送者分配一些代幣。

[24] 更準確地說，當接收帳戶未初始化或凍結時，帳戶預期的 *StateInit* 雜湊相等，或者當接收帳戶處於活動狀態，並且其程式碼或資料是與訊息的 *StateInit* 中接收的程式碼或資料的雜湊匹配的外部雜湊引用時，使用入站訊息的 `init` 欄位中的資訊。

請注意，即使智慧合約沒有對具有空有效載荷的訊息做特殊規定，並在處理此類訊息時拋出例外，收到的值（減去 gas 支付）仍將添加到智慧合約的餘額中。

**3.1.14. 訊息來源地址和邏輯建立時間決定其生成區塊.** 請注意，內部或出站外部訊息的來源地址和邏輯建立時間唯一地確定訊息生成的區塊。實際上，來源地址確定來源分片鏈，並且該分片鏈的區塊被分配不相交的邏輯時間間隔，因此只有其中一個可能包含指示的邏輯建立時間。這就是為什麼訊息中不需要明確提及生成區塊的原因。

**3.1.15. 封包訊息.** 訊息封包用於將路由資訊（例如當前（轉接）地址和下一跳地址）附加到入站、轉接和出站訊息（參見 **2.1.16**）。訊息本身保存在單獨的單元中，並透過單元引用從訊息封包引用。

```
interm_addr_regular$0 use_src_bits:(#<= 96)
  = IntermediateAddress;
interm_addr_simple$10 workchain_id:int8 addr_pfx:(64 * Bit)
  = IntermediateAddress;
interm_addr_ext$11 workchain_id:int32 addr_pfx:(64 * Bit)
  = IntermediateAddress;
msg_envelope cur_addr:IntermediateAddress
  next_addr:IntermediateAddress fwd_fee_remaining:Grams
  msg:^(Message Any) = MsgEnvelope;
```

`IntermediateAddress` 類型用於描述訊息的中間地址——即其當前(或轉接)地址 `cur_addr` 和其下一跳地址 `next_addr`。第一個建構器 `interm_addr_regular` 使用 **2.1.15**中描述的優化來表示中間地址，方法是儲存與來源地址相同的中間地址的第一位元數量；另外兩個明確儲存工作鏈識別碼和該工作鏈內地址的前 64 位元（剩餘位元可以從來源地址獲取）。`fwd_fee_remaining` 欄位用於明確表示在剩餘的 HR 步驟期間可以從訊息值中扣除的訊息轉發費用的最大金額；它不能超過訊息本身中指示的 `fwd_fee` 值。

## 3.2 入站訊息描述符

本節討論 *InMsgDescr*，包含匯入到區塊中的所有入站訊息的描述的結構。[25]

[25] 嚴格來說，*InMsgDescr* 是此結構的類型；我們故意使用相同的符號來描述區塊中此類型的唯一實例。

**3.2.1. 入站訊息的類型和來源.** *InMsgDescr* 中提到的每條入站訊息都由類型 *InMsg*（「入站訊息描述符」）的值描述，該值指定訊息的來源、其匯入到此區塊的原因，以及有關其「命運」的一些資訊——其由交易處理或在區塊內轉發。

入站訊息可分類如下：

- 入站外部訊息 — 不需要額外的理由就可以匯入到區塊中，但必須立即由同一區塊中的交易處理。
- 目的地地址在此區塊中的內部 *IHR* 訊息 — 它們被匯入到區塊中的原因包括其生成的梅克證明（即它們包含在其原始區塊的 *OutMsgDescr* 中）。這樣的訊息必須立即傳遞到其最終目的地並由交易處理。
- 目的地在此區塊中的內部訊息 — 它們被包含的原因是它們存在於相鄰分片鏈最新狀態的 *OutMsgQueue* 中，[26]或它們存在於該區塊的 *OutMsgDescr* 中。此相鄰分片鏈完全由轉發訊息封包中指示的轉接地址確定，該地址也在 *InMsg* 中複製。此訊息的「命運」再次透過對當前區塊內處理交易的引用來描述。
- 立即路由的內部訊息 — 本質上是前一類訊息的子類。在這種情況下，匯入的訊息是該區塊中生成的出站訊息之一。
- 轉接內部訊息 — 具有與前一類訊息相同的包含原因。然而，它們不在區塊內處理，而是內部轉發到 *OutMsgDescr* 和 *OutMsgQueue*。此事實以及對轉接訊息新封包的引用必須在 *InMsg* 中註冊。
- 目的地在此區塊中的丟棄內部訊息 — 目的地在此區塊中的內部訊息可能被匯入並立即丟棄，而不是由交易處理，如果它已經在該分片鏈的前一個區塊中透過 IHR 接收和處理。在這種情況下，必須提供對先前處理交易的引用。
- 丟棄的轉接內部訊息 — 類似地，轉接訊息可能在匯入後立即被丟棄，如果它已經透過 IHR 傳遞到其最終目的地。在這種情況下，需要其在最終區塊中作為 IHR 訊息處理的梅克證明。

**3.2.2. 入站訊息的描述符.** 每條入站訊息由 `InMsg` 類型的實例描述，該類型具有六個建構器，對應於 **3.2.1**中列出的情況：

[26]回顧一下，分片鏈被視為自身的相鄰鏈。

```
msg_import_ext$000 msg:^(Message Any) transaction:^Transaction
              = InMsg;
msg_import_ihr$010 msg:^(Message Any) transaction:^Transaction
    ihr_fee:Grams proof_created:^Cell = InMsg;
msg_import_imm$011 in_msg:^MsgEnvelope
    transaction:^Transaction fwd_fee:Grams = InMsg;
msg_import_fin$100 in_msg:^MsgEnvelope
    transaction:^Transaction fwd_fee:Grams = InMsg;
msg_import_tr$101  in_msg:^MsgEnvelope out_msg:^MsgEnvelope
    transit_fee:Grams = InMsg;
msg_discard_fin$110 in_msg:^MsgEnvelope transaction_id:uint64
    fwd_fee:Grams = InMsg;
msg_discard_tr$111 in_msg:^MsgEnvelope transaction_id:uint64
    fwd_fee:Grams proof_delivered:^Cell = InMsg;
```

請注意，處理交易在前四個建構器中直接透過對 Transaction 的單元引用引用，即使交易的邏輯時間 transaction_lt:uint64 就足夠了。內部一致性條件確保引用的交易確實屬於訊息中指示的目的地智慧合約，並且該交易處理的入站訊息確實是此 *InMsg* 實例中描述的那個。

此外，請注意，msg_import_imm 可以透過觀察到它是正在處理的訊息的邏輯建立時間大於或等於匯入訊息的區塊的（最小）邏輯時間的唯一情況來與 msg_import_fin 區分。

**3.2.3. 從匯入的訊息中收取轉發和轉接費用.** *InMsg* 結構也用於指示從入站訊息中收取的轉發和轉接費用。費用本身在 ihr_fee、fwd_fee 或 transit_fee 欄位中指示；它僅在入站外部訊息中不存在，後者使用其他機制來獎勵驗證者匯入它們。費用必須滿足以下內部一致性條件：

- 對於外部訊息（msg_import_ext），沒有轉發費用。
- 對於 IHR 匯入的內部訊息（msg_import_ihr），費用等於 ihr_fee，它必須與訊息本身中指示的 ihr_fee 值一致。請注意，從 IHR 匯入的訊息中永遠不會收取 fwd_fee 或 fwd_fee_remaining。
- 對於傳遞到其目的地的內部訊息(msg_import_fin 和 msg_import_imm)，費用等於封包入站訊息 in_msg 的 fwd_fee_remaining。請注意，它不能超過訊息本身中指示的 fwd_fee 值。
- 對於轉接訊息（msg_import_tr），費用等於 in_msg 和 out_msg 封包中指示的 fwd_fee_remaining 值之間的差。

- 對於丟棄的訊息，費用也等於 in_msg 中指示的 fwd_fee_remaining。

**3.2.4. 入站訊息的匯入值.** 每條匯入的訊息都會將一些值——一定數量的一種或多種加密貨幣——匯入到區塊中。此匯入值的計算如下：

- 外部訊息不匯入任何值。
- IHR 匯入的訊息匯入其 value 加上其 ihr_fee。
- 傳遞的或轉接的內部訊息匯入其 value 加上其 ihr_fee 加上其 in_msg 封包的 fwd_fee_remaining 值。
- 丟棄的訊息匯入其 in_msg 封包的 fwd_fee_remaining。

請注意，從匯入訊息中收取的轉發和轉接費用不超過其匯入值。

**3.2.5. 增強雜湊映射或字典.** 在繼續之前，讓我們討論增強雜湊映射或字典的序列化。

增強雜湊映射是具有 $n$ 位元鍵和某種類型 $X$ 值的鍵值儲存結構，類似於 [4, 3.3] 中描述的普通雜湊映射。然而，表示增強雜湊映射的 Patricia 樹的每個中間節點都透過類型 $Y$ 的值增強。

這些增強值必須滿足某些聚合條件。通常，$Y$ 是整數類型，聚合條件是分叉的增強值必須等於其兩個子節點的增強值之和。一般而言，使用分叉評估函數 $S : Y \times Y \to Y$ 或 $S : Y \to Y \to Y$ 代替總和。葉的增強值通常透過葉評估函數 $L : X \to Y$ 從該葉中儲存的值計算。葉的增強值可能與值一起明確儲存在葉中；然而，在大多數情況下不需要這樣做，因為葉評估函數 $L$ 非常簡單。

**3.2.6. 增強雜湊映射的序列化.** 具有 $n$ 位元鍵、類型 $X$ 值和類型 $Y$ 增強值的增強雜湊映射的序列化由以下 TL-B 方案給出，這是 [4, 3.3.3] 中提供的擴充：

```
ahm_edge#_ {n:#} {X:Type} {Y:Type} {l:#} {m:#}
  label:(HmLabel ~l n) {n = (~m) + l}
  node:(HashmapAugNode m X Y) = HashmapAug n X Y;
ahmn_leaf#_ {X:Type} {Y:Type} extra:Y value:X
  = HashmapAugNode 0 X Y;
ahmn_fork#_ {n:#} {X:Type} {Y:Type}
    left:^(HashmapAug n X Y) right:^(HashmapAug n X Y) extra:Y
  = HashmapAugNode (n + 1) X Y;
```

```
ahme_empty$0 {n:#} {X:Type} {Y:Type} extra:Y
          = HashmapAugE n X Y;
ahme_root$1 {n:#} {X:Type} {Y:Type} root:^(HashmapAug n X Y)
  extra:Y = HashmapAugE n X Y;
```

**3.2.7. *InMsgDescr* 的增強.** 入站訊息描述符的集合由兩個貨幣值的向量增強，表示從訊息或訊息集合中匯入的值以及收取的轉發和轉接費用：

```
import_fees$_ fees_collected:Grams
  value_imported:CurrencyCollection = ImportFees;
```

**3.2.8. *InMsgDescr* 的結構.** 現在 *InMsgDescr* 本身被定義為增強雜湊映射，具有 256 位元鍵（等於匯入訊息的表示雜湊）、類型 `InMsg` 的值（參見 **3.2.2**）和類型 `ImportFees` 的增強值（參見 **3.2.7**）：

```
_ (HashmapAugE 256 InMsg ImportFees) = InMsgDescr;
```

此 TL-B 記號使用匿名建構器 `_` 將 `InMsgDescr` 定義為另一個類型的同義詞。

**3.2.9. *InMsgDescr* 的聚合規則.** 分叉評估和葉評估函數（參見 **3.2.5**）未在上述記號中明確包含，因為 TL-B 的依賴類型不足以表達此目的。用文字來說，分叉評估函數只是兩個 `ImportFees` 實例的逐組件相加，葉評估函數由 **3.2.3**和 **3.2.4**中列出的規則定義。這樣，表示 *InMsgDescr* 實例的 Patricia 樹的根包含一個 *ImportFees* 實例，其中包含所有入站訊息匯入的總值，以及從它們收取的總轉發費用。

## 3.3 出站訊息佇列和描述符

本節討論 *OutMsgDescr*，該結構表示區塊的所有出站訊息，以及它們的封包和將它們包含在 *OutMsgDescr* 中的原因的簡要描述。此結構還描述 *OutMsgQueue* 的所有修改，它是分片鏈狀態的一部分。

**3.3.1. 出站訊息的類型.** 出站訊息可分類如下：

- 外部出站訊息，或「無處而去的訊息」— 由該區塊內的交易生成。將此類訊息包含在 *OutMsgDescr* 中的原因只是對其生成交易的引用。
- 立即處理的內部出站訊息 — 在該區塊中生成和處理，不包含在 *OutMsgQueue* 中。包含此類訊息的原因是對其生成交易的引用，其「命運」透過對 *InMsgDescr* 中相應條目的引用來描述。

- 普通（內部）出站訊息 — 在此區塊中生成並包含在 *OutMsgQueue* 中。

- 轉接（內部）出站訊息 — 匯入到同一區塊的 *InMsgDescr* 中，並透過 HR 路由，直到獲得當前分片之外的下一跳地址。

**3.3.2. 訊息出列記錄.** 除了上述類型的出站訊息外，*OutMsgDescr* 還可以包含特殊的「訊息出列記錄」，它指示訊息已在此區塊中從 *OutMsgQueue* 中刪除。刪除的原因在訊息刪除記錄中指示；它包含對被刪除的封包訊息的引用，以及在其 *InMsgDescr* 中具有此封包訊息的相鄰分片鏈區塊的邏輯時間。

請注意，在某些情況下，訊息可能會從當前分片鏈的 *OutMsgQueue* 匯入，內部路由，然後再次包含在 *OutMsgDescr* 和 *OutMsgQueue* 中，但使用不同的封包。[27]在這種情況下，使用轉接出站訊息描述的變體，它也充當訊息出列記錄。

**3.3.3. 出站訊息的描述符.** 每條出站訊息由 *OutMsg* 的實例描述：

```
msg_export_ext$000 msg:^(Message Any)
    transaction:^Transaction = OutMsg;
msg_export_imm$010 out_msg:^MsgEnvelope
    transaction:^Transaction reimport:^InMsg = OutMsg;
msg_export_new$001 out_msg:^MsgEnvelope
    transaction:^Transaction = OutMsg;
msg_export_tr$011  out_msg:^MsgEnvelope
    imported:^InMsg = OutMsg;
msg_export_deq$110 out_msg:^MsgEnvelope
    import_block_lt:uint64 = OutMsg;
msg_export_tr_req$111 out_msg:^MsgEnvelope
    imported:^InMsg = OutMsg;
```

最後兩個描述具有從 *OutMsgQueue* 中刪除（出列）訊息而不是插入它的效果。最後一個在執行內部路由後（參見 **2.1.11**）使用新封包將訊息重新插入 *OutMsgQueue*。

**3.3.4. 出站訊息的匯出值.** *OutMsg* 描述的每條出站訊息都會從區塊中匯出一些值——一定數量的一種或多種加密貨幣。此匯出值的計算如下：

[27]這種情況很少見，僅在分片鏈合併事件之後發生。通常，從同一分片鏈的 *OutMsgQueue* 匯入的訊息的目的地在該分片鏈內，並相應地被處理，而不是重新排隊。

- 外部出站訊息不匯出任何值。
- 在此區塊中生成的內部訊息匯出其 `value` 加上其 `ihr_fee` 加上其 `fwd_fee`。請注意，`fwd_fee` 必須等於 `out_msg` 封包中指示的 `fwd_fee_remaining`。
- 轉接訊息匯出其 `value` 加上其 `ihr_fee` 加上其 `out_msg` 封包的 `fwd_fee_remaining` 值。
- 對於 `msg_export_tr_req`，用於重新插入的出列訊息的 *OutMsg* 建構器，同樣適用。
- 訊息出列記錄（`msg_export_deq`；參見 **3.3.2**）不匯出任何值。

**3.3.5. *OutMsgDescr* 的結構.** *OutMsgDescr* 本身只是一個增強雜湊映射（參見 **3.2.5**），具有 256 位元鍵（等於訊息的表示雜湊）、類型 *OutMsg* 的值和類型 *CurrencyCollection* 的增強值：

```
_ (HashmapAugE 256 OutMsg CurrencyCollection) = OutMsgDescr;
```

增強是相應訊息的匯出值，透過總和聚合，並在葉處計算，如 **3.3.4**中所述。這樣，總匯出值出現在表示 *OutMsgDescr* 的 Patricia 樹的根附近。

*OutMsgDescr* 最重要的一致性條件是其鍵為 $k$ 的條目必須是描述表示雜湊 $\text{HASH}^b(m) = k$ 的訊息 $m$ 的 *OutMsg*。

**3.3.6. *OutMsgQueue* 的結構.** 回顧（參見 **1.2.7**）*OutMsgQueue* 是區塊鏈狀態的一部分，而不是區塊的一部分。因此，區塊僅包含對其初始和最終狀態以及其新建立的單元的雜湊引用。

*OutMsgQueue* 的結構很簡單：它只是一個具有 352 位元鍵和類型 *OutMsg* 值的增強雜湊映射：

```
_ (HashmapAugE 352 OutMsg uint64) = OutMsgQueue;
```

用於出站訊息 $m$ 的鍵是其 32 位元下一跳 *workchain_id*、該工作鏈內下一跳地址的前 64 位元和訊息 $m$ 本身的表示雜湊 $\text{HASH}^b(m)$ 的串接。增強在葉處透過訊息 $m$ 的邏輯建立時間 $\text{LT}(m)$ 進行，在分叉處透過子節點增強值的最小值進行。

*OutMsgQueue* 最重要的一致性條件是鍵 $k$ 處的值確實必須包含具有預期下一跳地址和表示雜湊的封包訊息。

**3.3.7. *OutMsg* 的一致性條件.** 對 *OutMsgDescr* 中存在的 *OutMsg* 實例施加幾個內部一致性條件。它們包括以下內容：

- 出站訊息描述的前三個建構器中的每一個都包含對生成交易的引用。此交易必須屬於訊息的來源帳戶，它必須包含對指定訊息的引用作為其出站訊息之一，並且它必須透過透過其 `account_id` 和 `transaction_id` 查找來可恢復。
- `msg_export_tr` 和 `msg_export_tr_req` 必須引用描述相同訊息（在不同的原始封包中）的 *InMsg* 實例。
- 如果使用前四個建構器之一，則訊息必須在區塊的初始 *OutMsgQueue* 中不存在；否則，它必須存在。
- 如果使用 `msg_export_deq`，則訊息必須在區塊的最終 *OutMsgQueue* 中不存在；否則，它必須存在。
- 如果訊息未在 *OutMsgDescr* 中提及，則它在區塊的初始和最終 *OutMsgQueue* 中必須相同。

# 4 帳戶和交易

本章討論 TON 區塊鏈中帳戶（或智慧合約）及其狀態的佈局。它還考慮了交易，這是修改帳戶狀態、處理入站訊息和生成新出站訊息的唯一方式。

## 4.1 帳戶及其狀態

回顧一下，在 TON 區塊鏈的背景下，智慧合約和帳戶是同一回事，並且這些術語可以互換使用，至少在僅考慮小型（或「通常」）智慧合約時是如此。大型智慧合約可能會使用位於同一工作鏈的不同分片鏈中的多個帳戶以進行負載平衡。

帳戶由其完整地址識別，並由其狀態完全描述。換句話說，帳戶中除了其地址和狀態外沒有其他任何東西。

**4.1.1. 帳戶地址.** 一般而言，帳戶完全由其完整地址識別，該地址由 32 位元 $workchain\_id$ 和所選工作鏈內（通常為 256 位元）內部地址或帳戶識別碼 $account\_id$ 組成。在基礎工作鏈（$workchain\_id = 0$）和主鏈（$workchain\_id = -1$）中，內部地址始終為 256 位元。在這些工作鏈中，[28]$account\_id$ 不能任意選擇，但必須等於智慧合約的初始程式碼和資料的雜湊；否則，將無法使用預期的程式碼和資料初始化帳戶（參見 **1.7.3**），並且無法對帳戶餘額中累積的資金做任何事情。

**4.1.2. 零帳戶.** 按照慣例，零帳戶或具有零地址的帳戶累積處理、轉發和轉接費用，以及由主鏈或工作鏈的驗證者收取的任何其他付款。此外，零帳戶是「大型智慧合約」，意味著每個分片鏈都有其零帳戶的實例，地址的最高有效位元經過調整以位於分片中。任何有意或無意轉移到零帳戶的資金實際上都是給驗證者的禮物。例如，智慧合約可能透過將其所有資金發送到零帳戶來銷毀自己。

**4.1.3. 小型和大型智慧合約.** 預設情況下，智慧合約是「小型」的，意味著它們有一個帳戶地址，在任何給定時刻恰好屬於一個分片鏈。然而，可以建立「分割深度為 $d$ 的大型智慧合約」，意味著最多可以建立 $2^d$ 個智慧合約實例，智慧合約原始地址的前 $d$ 位元被任意位元序列替換。[29]可以使用 `anycast` 設定為 $d$ 的內部任播地址向此類智慧合約發送訊息（參見 **3.1.2**）。此外，大型智慧合約的實例被允許使用此任播地址作為其生成訊息的來源地址。

[28]為簡單起見，我們有時將主鏈視為僅具有 $workchain\_id = -1$ 的另一個工作鏈。

[29]實際上，前 $d$ 位元以這樣的方式被替換：每個分片最多包含大型智慧合約的一個實例，並且前綴 $s$ 長度為 $|s| \le d$ 的分片 $(w, s)$ 恰好包含一個實例。

大型智慧合約的實例是具有非零最大分割深度 $d$ 的帳戶。

**4.1.4. 三種帳戶.** 有三種帳戶：

- 未初始化 — 帳戶僅有餘額；其程式碼和資料尚未初始化。
- 活動 — 帳戶的程式碼和資料也已初始化。
- 凍結 — 帳戶的程式碼和資料已被雜湊替換，但餘額仍明確儲存。凍結帳戶的餘額可能實際上變為負數，反映應付的儲存費用。

**4.1.5. 帳戶的儲存概況.** 帳戶的儲存概況是描述該帳戶使用的持久區塊鏈狀態儲存量的資料結構。它描述使用的單元總數、資料位元、內部和外部單元引用。

```
storage_used$_ cells:(VarUInteger 7) bits:(VarUInteger 7)
  ext_refs:(VarUInteger 7) int_refs:(VarUInteger 7)
  public_cells:(VarUInteger 7) = StorageUsed;
```

相同類型 `StorageUsed` 可以表示訊息的儲存概況，例如，根據需要計算 `fwd_fee`，即超立方體路由的總轉發費用。帳戶的儲存概況有一些額外的欄位，指示上次徵收儲存費用的時間：

```
storage_info$_ used:StorageUsed last_paid:uint32
               due_payment:(Maybe Grams) = StorageInfo;
```

`last_paid` 欄位包含最近一次收取儲存付款的 unixtime（通常這是最近一次交易的 unixtime），或帳戶建立時的 unixtime（同樣，由交易建立）。`due_payment` 欄位（如果存在）累積無法從帳戶餘額中收取的儲存付款，由嚴格正的 nanogram 金額表示；它只能存在於餘額為零 Gram 的未初始化或凍結帳戶中（但可能在其他加密貨幣中有非零餘額）。當 `due_payment` 變得大於區塊鏈的可配置參數值時，帳戶將被完全銷毀，其餘額（如果有）將轉移到零帳戶。

**4.1.6. 帳戶描述.** 帳戶的狀態由類型 *Account* 的實例表示，透過以下 TL-B 方案描述：[30]

```
account_none$0 = Account;
account$1 addr:MsgAddressInt storage_stat:StorageInfo
          storage:AccountStorage = Account;
```

[30]此方案使用匿名建構器和匿名欄位，兩者都由底線 `_` 表示。

```
account_storage$_ last_trans_lt:uint64
    balance:CurrencyCollection state:AccountState
  = AccountStorage;

account_uninit$00 = AccountState;
account_active$1 _:StateInit = AccountState;
account_frozen$01 state_hash:uint256 = AccountState;

acc_state_uninit$00 = AccountStatus;
acc_state_frozen$01 = AccountStatus;
acc_state_active$10 = AccountStatus;
acc_state_nonexist$11 = AccountStatus;

tick_tock$_ tick:Bool tock:Bool = TickTock;

_ split_depth:(Maybe (## 5)) special:(Maybe TickTock)
  code:(Maybe ^Cell) data:(Maybe ^Cell)
  library:(Maybe ^Cell) = StateInit;
```

請注意，account_frozen 包含 *StateInit* 實例的表示雜湊，而不是該實例本身，否則該實例將包含在 account_active 中；account_uninit 類似於 account_frozen，但它不包含明確的 state_hash，因為假定它等於帳戶的內部地址（*account_id*），該地址已存在於 addr 欄位中。split_depth 欄位僅在大型智慧合約的實例中存在且非零。special 欄位只能存在於主鏈中——並且在主鏈中，只能存在於整個系統運作所需的某些基礎智慧合約中。

storage_stat 中保存的儲存統計資料反映單元切片 storage 的總儲存使用情況。特別是，用於儲存 balance 的位元和單元也反映在 storage_stat 中。

當需要表示不存在的帳戶時，使用 account_none 建構器。

**4.1.7. 帳戶狀態作為從帳戶到其未來自身的訊息.** 請注意，帳戶狀態與從帳戶發送到其未來自身參與下一個交易的訊息非常相似，原因如下：

- 帳戶狀態在同一帳戶的兩個連續交易之間不會改變，因此在這方面它與從較早交易發送到較晚交易的訊息完全相似。
- 當處理交易時，其輸入是入站訊息和先前的帳戶狀態；其輸出是生成的出站訊息和下一個帳戶狀態。如果我們將狀態視為一種特殊類型的

訊息，我們會看到每個交易恰好有兩個輸入（帳戶狀態和入站訊息）和至少一個輸出。

- 訊息和帳戶狀態都可以在 *StateInit* 實例中攜帶程式碼和資料，以及在其 `balance` 中攜帶一些值。
- 帳戶由建構器訊息初始化，該訊息本質上攜帶帳戶的未來狀態和餘額。
- 在某些情況下，訊息會轉換為帳戶狀態，反之亦然。例如，當發生分片鏈合併事件，並且需要合併同一大型合約的兩個實例時，其中一個將轉換為發送到另一個的訊息（參見 **4.2.11**）。類似地，當發生分片鏈分裂事件，並且需要將大型智慧合約的實例分裂為兩個時，這是透過特殊交易實現的，該交易透過從先前存在的實例發送到新實例的建構器訊息來建立新實例（參見 **4.2.10**）。
- 可以說，訊息涉及跨空間轉移一些資訊（在不同的分片鏈之間，或至少在帳戶鏈之間），而帳戶狀態跨時間轉移資訊（從同一帳戶的過去到未來）。

**4.1.8. 訊息和帳戶狀態之間的差異.** 當然，也有重要的差異。例如：

- 帳戶狀態僅「及時」轉移（從分片鏈區塊到其後繼），但永遠不會「在空間中」轉移（從一個分片鏈到另一個）。因此，此轉移是隱式完成的，無需在區塊鏈中的任何地方建立帳戶狀態的完整副本。
- 驗證者為保留帳戶狀態而收取的儲存付款通常遠小於相同資料量的訊息轉發費用。
- 當入站訊息傳遞到帳戶時，呼叫的是帳戶中的程式碼，而不是訊息中的程式碼。

**4.1.9. 分片中所有帳戶的組合狀態.** 分片鏈狀態的分割部分(參見 **1.2.1**和 **1.2.2**)由下式給出：

```
_ (HashmapAugE 256 Account CurrencyCollection) = ShardAccounts;
```

這只是一個字典，以 256 位元 $account\_id$ 作為鍵，相應的帳戶狀態作為值，由帳戶餘額總和增強。這樣就計算出分片鏈中所有帳戶的餘額總和，以便可以輕鬆檢查分片中「儲存」的加密貨幣總量。

內部一致性條件確保 *SmartAccounts* 中鍵 $k$ 引用的帳戶地址確實等於 $k$。額外的內部一致性條件要求所有鍵 $k$ 都以分片前綴 $s$ 開頭。

**4.1.10. 帳戶擁有者和介面描述.** 人們可能希望在受控帳戶中包含一些可選資訊。例如，個人使用者或公司可能希望在其錢包帳戶中添加文字描述欄位，包含使用者或公司的名稱或地址（或它們的雜湊，如果資訊不應公開）。或者，智慧合約可能會提供其支援方法及其預期應用的機器可讀或人類可讀的描述，進階錢包應用程式可能會使用這些描述來建構下拉選單和表單，幫助人類使用者建立有效的訊息發送到該智慧合約。

包含此類資訊的一種方法是保留，比如說，帳戶狀態的 `data` 單元中的第二個引用用於具有 64 位元鍵（對應於可能想要儲存的額外資料的標準類型的某些識別碼）和相應值的字典。然後區塊鏈瀏覽器將能夠提取所需的值，如果需要還可以提供梅克證明。

更好的方法是在智慧合約中定義一些 *get* 方法。

**4.1.11. 智慧合約的 get 方法.** *Get* 方法由獨立的 TVM 實例執行，帳戶的程式碼和資料載入其中。所需參數在堆疊上傳遞（例如，指示要獲取的欄位或要呼叫的特定 get 方法的魔術數字），結果也在 TVM 堆疊上返回（例如，包含具有帳戶擁有者名稱的字串序列化的單元切片）。

作為獎勵，get 方法可用於獲取比僅獲取常數物件更複雜的查詢的答案。例如，TON DNS 註冊智慧合約提供 get 方法來在註冊表中查找網域字串並返回相應的記錄（如果找到）。

按照慣例，get 方法使用大型負 32 位元或 64 位元索引或魔術數字，智慧合約的內部函數使用連續的正索引，用於 TVM 的 `CALLDICT` 指令。智慧合約的 `main()` 函數用於在普通交易中處理入站訊息，始終具有索引零。

## 4.2 交易

根據無限分片範式和 actor 模型，TON 區塊鏈的三個主要組件是帳戶（及其狀態）、訊息和交易。前面的章節已經討論了前兩個；本節考慮交易。

與訊息相反（訊息在 TON 區塊鏈的所有工作鏈中基本上具有相同的標頭）以及帳戶（帳戶至少具有一些共同部分（地址和餘額）），我們對交易的討論必然僅限於主鏈和基礎工作鏈。其他工作鏈可能定義完全不同類型的交易。

**4.2.1. 交易的邏輯時間.** 每個交易 $t$ 都有分配給它的邏輯時間間隔 $\text{Lt}^\bullet(t) = [\text{Lt}^-(t), \text{Lt}^+(t))$（參見 **1.4.6**和 **1.4.3**）。按照慣例，生成 $n$ 個出站訊息 $m_1$, ..., $m_n$ 的交易 $t$ 被分配長度為 $n+1$ 的邏輯時間間隔，因此

$$\text{Lt}^+(t) = \text{Lt}^-(t) + n + 1 \quad . \tag{16}$$

我們還設定 $\text{LT}(t) := \text{LT}^-(t)$，並透過以下方式分配訊息 $m_i$(其中 $1 \le i \le n$)的邏輯建立時間：

$$\text{LT}(m_i) = \text{LT}^-(m_i) := \text{LT}^-(t) + i, \quad \text{LT}^+(m_i) := \text{LT}^-(m_i) + 1 \quad . \tag{17}$$

這樣，每個生成的出站訊息在交易 $t$ 的邏輯時間間隔 $\text{LT}^\bullet(t)$ 內被分配其自己的單位間隔。

**4.2.2. 邏輯時間唯一識別帳戶的交易和出站訊息.** 回顧一下，對邏輯時間施加的條件意味著對於同一帳戶 $\xi$ 的任何前一個交易 $t'$，$\text{LT}^-(t) \ge \text{LT}^+(t')$，並且如果 $m$ 是由交易 $t$ 處理的入站訊息，則 $\text{LT}^-(t) > \text{LT}(m)$。這樣，同一帳戶的交易的邏輯時間間隔彼此不相交，因此，帳戶生成的所有出站訊息的邏輯時間間隔也彼此不相交。換句話說，當 $m$ 遍歷同一帳戶 $\xi$ 的所有出站訊息時，所有 $\text{LT}(m)$ 都不同。

這樣，當與帳戶識別碼 $\xi$ 結合時，$\text{LT}(t)$ 和 $\text{LT}(m)$ 唯一地確定該帳戶的交易 $t$ 或出站訊息 $m$。此外，如果有一個帳戶的所有交易及其邏輯時間的有序列表，則很容易找到生成給定出站訊息 $m$ 的交易，只需查找邏輯時間 $\text{LT}(t)$ 從下方最接近 $\text{LT}(m)$ 的交易 $t$ 即可。

**4.2.3. 交易的通用組件.** 每個交易 $t$ 包含或間接引用以下資料：

- 交易所屬的帳戶 $\xi$。
- 交易的邏輯時間 $\text{LT}(t)$。
- 交易處理的一個或零個入站訊息 $m$。
- 生成的出站訊息數量 $n \ge 0$。
- 出站訊息 $m_1$, ..., $m_n$。
- 帳戶 $\xi$ 的初始狀態（包括其餘額）。
- 帳戶 $\xi$ 的最終狀態（包括其餘額）。
- 驗證者收取的總費用。
- 包含驗證所需的全部或部分資料的交易詳細描述，包括交易的類型（參見 **4.2.4**）和執行的一些中間步驟。

在這些組件中，除了最後一個組件外，所有組件都相當通用，也可能出現在其他工作鏈中。

**4.2.4. 交易的類型.** 主鏈和分片鏈中允許不同類型的交易。普通交易包括將一條入站訊息傳遞到帳戶，並由該帳戶的程式碼處理；這是最常見的交易類型。此外，還有幾種類型的特殊交易。

總共有六種交易類型：

- 普通交易 — 屬於帳戶 $\xi$。它們恰好處理一條目的地為 $\xi$ 的入站訊息 $m$（在包含區塊的 *InMsgDescr* 中描述），計算帳戶的新狀態，並生成幾個來源為 $\xi$ 的出站訊息（在 *OutMsgDescr* 中註冊）。

- 儲存交易 — 可以由驗證者自行決定插入。它們不處理任何入站訊息，也不呼叫任何程式碼。它們的唯一效果是從帳戶收取儲存付款，影響其儲存統計資料和餘額。如果帳戶的結果 Gram 餘額變得小於某個金額，則帳戶可能會被凍結，其程式碼和資料會被它們的組合雜湊替換。

- *Tick* 交易 — 自動為主鏈中某些在其狀態中設定了 `tick` 旗標的特殊帳戶（智慧合約）呼叫，作為每個主鏈區塊中的第一個交易。它們沒有入站訊息，但可能生成出站訊息並改變帳戶狀態。例如，驗證者選舉由主鏈中特殊智慧合約的 tick 交易執行。

- *Tock* 交易 — 類似地自動呼叫，作為每個主鏈區塊中某些特殊帳戶的最後一個交易。

- 分裂交易 — 在緊接分片鏈分裂事件之前的分片鏈區塊的最後一個交易中呼叫。它們為需要在分裂後產生新實例的大型智慧合約實例自動觸發。

- 合併交易 — 類似地在緊接分片鏈合併事件之後的分片鏈區塊的第一個交易中呼叫，如果大型智慧合約的實例需要與同一智慧合約的另一個實例合併。

請注意，在這六種交易類型中，只有四種可以在主鏈中發生，另一個四種的子集可以在基礎工作鏈中發生。

**4.2.5. 普通交易的階段.** 普通交易分幾個階段執行，可以將其視為緊密綁定為一個的幾個「子交易」：

- 儲存階段 — 收取帳戶狀態（包括智慧合約程式碼和資料，如果存在）的應付儲存付款直到當前時間。智慧合約可能因此被凍結。如果智慧合約之前不存在，則不存在儲存階段。

- 信用階段 — 帳戶記入收到的入站訊息的值。
- 計算階段 — 智慧合約的程式碼在具有適當參數的 TVM 實例內呼叫，包括入站訊息和持久資料的副本，並以退出代碼、新的持久資料和操作列表（例如包括要發送的出站訊息）終止。處理階段可能導致建立新帳戶（未初始化或活動）或啟動先前未初始化或凍結的帳戶。*gas* 付款等於 gas 價格和消耗的 gas 的乘積，從帳戶餘額中收取。
- 操作階段 — 如果智慧合約成功終止（退出代碼為 0 或 1），則執行列表中的操作。如果無法執行所有操作——例如，由於資金不足無法隨出站訊息一起轉移——則交易被中止，帳戶狀態回滾。如果智慧合約沒有成功終止，或者由於它未初始化或凍結而根本無法呼叫智慧合約，則交易也會被中止。
- 退回階段 — 如果交易被中止，並且入站訊息設定了其 `bounce` 旗標，則透過自動生成出站訊息（`bounce` 旗標清除）到其原始發送者來「退回」它。原始入站訊息的幾乎所有值（減去 gas 付款和轉發費用）都轉移到生成的訊息，該訊息在其他方面具有空主體。

**4.2.6. 退回到不存在帳戶的入站訊息.** 請注意，如果將設定了 `bounce` 旗標的入站訊息發送到先前不存在的帳戶，並且交易被中止（例如，因為入站訊息中沒有具有正確雜湊的程式碼和資料，因此根本無法呼叫虛擬機器），則該帳戶甚至不會建立為未初始化帳戶，因為它無論如何都會有零餘額且沒有程式碼和資料。[31]

**4.2.7. 入站訊息的處理分為計算階段和操作階段.** 請注意，入站訊息的處理實際上分為兩個階段：計算階段和操作階段。在計算階段，呼叫虛擬機器並執行必要的計算，但不採取虛擬機器外部的任何操作。換句話說，*TVM* 中智慧合約的執行沒有副作用；智慧合約在執行期間無法直接與區塊鏈互動。相反，`SENDMSG` 等 TVM 原語只是將所需的操作（例如，要發送的出站訊息）儲存到逐漸累積在 TVM 控制暫存器 `c5` 中的操作列表中。操作本身被推遲到操作階段，在此期間根本不呼叫使用者智慧合約。

**4.2.8. 將處理分為計算階段和操作階段的原因.** 這種安排的一些原因是：

- 如果智慧合約最終以 0 或 1 以外的退出代碼終止，則更容易中止交易。

[31]特別是，如果使用者錯誤地在可退回訊息中向不存在的地址發送一些資金，則資金不會浪費，而是會被退回（退回）。因此，除非明確指示，否則使用者錢包應用程式應在所有生成的訊息中預設設定 `bounce` 旗標。然而，在某些情況下不可退回訊息是必不可少的（參見 **1.7.6**）。

- 可以在不修改虛擬機器的情況下更改處理輸出操作的規則。（例如，可以引入新的輸出操作。）
- 虛擬機器本身可以被修改甚至被另一個虛擬機器替換（例如，在新的工作鏈中），而無需更改處理輸出操作的規則。
- 虛擬機器內智慧合約的執行與區塊鏈完全隔離，是純計算。因此，這種執行可以透過 TVM 的 `RUNVM` 原語在虛擬機器本身內部虛擬化，這對於驗證者智慧合約和控制支付通道和其他側鏈的智慧合約是一個有用的功能。此外，虛擬機器可以在其自身或其簡化版本內部模擬，這是用於在 TVM 內驗證智慧合約執行的有用功能。[32]

**4.2.9. 儲存、tick 和 tock 交易.** 儲存交易與普通交易的獨立儲存階段非常相似。Tick 和 tock 交易類似於沒有信用和退回階段的普通交易，因為沒有入站訊息。

**4.2.10. 分裂交易.** 分裂交易實際上由兩個交易組成。如果帳戶 $\xi$ 需要分裂為兩個帳戶 $\xi$ 和 $\xi'$：

- 首先為帳戶 $\xi$ 發出分裂準備交易，類似於 tock 交易（但在分片鏈中而不是在主鏈中）。它必須是分片鏈區塊中 $\xi$ 的最後一個交易。分裂準備交易的處理階段的輸出不僅包括帳戶 $\xi$ 的新狀態，還包括帳戶 $\xi'$ 的新狀態，由到 $\xi'$ 的建構器訊息表示（參見 **4.1.7**）。
- 然後為帳戶 $\xi'$ 添加分裂安裝交易，引用相應的分裂準備交易。分裂安裝交易必須是區塊中先前不存在的帳戶 $\xi'$ 的唯一交易。它有效地按照分裂準備交易定義設定 $\xi'$ 的狀態。

**4.2.11. 合併交易.** 合併交易也每個由兩個交易組成。如果帳戶 $\xi'$ 需要合併到帳戶 $\xi$ 中：

- 首先為 $\xi'$ 發出合併準備交易，將其所有持久狀態和餘額轉換為目的地為 $\xi$ 的特殊建構器訊息（參見 **4.1.7**）。
- 然後為 $\xi$ 發出合併安裝交易，引用相應的合併準備交易，處理該建構器訊息。合併安裝交易類似於 tick 交易，它必須是區塊中 $\xi$ 的第一個交易，但它位於分片鏈區塊中，而不是在主鏈中，並且它有特殊的入站訊息。

[32]在簡化版本的 TVM 中執行的 TVM 模擬器的參考實現可能被提交到主鏈，以便在驗證者對 TVM 的特定執行產生分歧時使用。這樣，可以檢測到 TVM 的有缺陷實現。然後參考實現作為 TVM 操作語義的權威來源。（參見 [4, B.2]）

**4.2.12. 一般交易的序列化.** 任何交易都包含 **4.2.3**中列出的欄位。因此，所有交易中都有一些共同組件：

```
transaction$_ account_addr:uint256 lt:uint64 outmsg_cnt:uint15
  orig_status:AccountStatus end_status:AccountStatus
  in_msg:(Maybe ^(Message Any))
  out_msgs:(HashmapE 15 ^(Message Any))
  total_fees:Grams state_update:^(MERKLE_UPDATE Account)
  description:^TransactionDescr = Transaction;

!merkle_update#02 {X:Type} old_hash:uint256 new_hash:uint256
  old:^X new:^X = MERKLE_UPDATE X;
```

`merkle_update` 的 TL-B 宣告中的驚嘆號表示此類值需要特殊處理。特別是，它們必須保存在單獨的單元中，該單元必須透過其標頭中的位元標記為特殊（參見 [4, 3.1]）。

可以在 **4.3**中找到 *TransactionDescr* 序列化的完整解釋，它根據 **4.2.4**中列出的類型描述一個交易。

**4.2.13. 交易生成的出站訊息的表示.** 交易 $t$ 生成的出站訊息保存在字典 `out_msgs` 中，其 15 位元鍵等於 0, 1, ..., $n-1$，其中 $n =$ `outmsg_cnt` 是生成的出站訊息數量。索引為 $0 \le i < n$ 的訊息 $m_{i+1}$ 必須有 $\mathrm{Lt}(m_{i+1}) = \mathrm{Lt}(t) + i + 1$，並且 $\mathrm{Lt}(t) = \mathrm{Lt}^{-}(t)$ 明確儲存在 `lt` 欄位中。

**4.2.14. 交易的一致性條件.** *Transaction* 中存在的欄位的共同序列化，獨立於其類型和描述，使我們能夠對任何交易施加幾個「外部」一致性條件。其中最重要的涉及交易內的值流：所有輸入的總和（入站訊息的匯入值加上帳戶的原始餘額）必須等於所有輸出的總和（帳戶的結果餘額，加上所有出站訊息的匯出值的總和，加上驗證者收取的所有儲存、處理和轉發費用）。這樣，對交易的表面檢查，該交易處理匯入值為 1 Gram 的入站訊息，由初始餘額為 10 Gram 的帳戶接收，在過程中生成匯出值為 100 Gram 的出站訊息，即使在檢查 TVM 執行的所有細節之前也會揭示其無效性。

其他一致性條件可能略微取決於交易的描述。例如，普通交易處理的入站訊息必須在包含區塊的 *InMsgDescr* 中註冊，並且相應的記錄必須包含對此交易的引用。類似地，所有交易生成的所有出站訊息（除了分裂準備或合併準備交易生成的一條特殊訊息外）都必須在 *OutMsgDescr* 中註冊。

**4.2.15. 帳戶的所有交易的集合.** 區塊中屬於同一帳戶 $\xi$ 的所有交易被收集到「帳戶鏈區塊」*AccountBlock* 中，它本質上是一個字典 `transactions`，具有 64 位元鍵，每個鍵等於相應交易的邏輯時間：

```
acc_trans$_ account_addr:uint256
            transactions:(HashmapAug 64 ^Transaction Grams)
            state_update:^(MERKLE_UPDATE Account)
          = AccountBlock;
```

transactions 字典由 *Grams* 值總和增強，該值聚合從這些交易收取的總費用。

除了這個字典外，*AccountBlock* 還包含帳戶總狀態的梅克更新（參見 [4, 3.1]）。如果帳戶在區塊之前不存在，則其狀態由 account_none 表示。

**4.2.16. *AccountBlock* 的一致性條件.** 對 *AccountBlock* 施加幾個一般一致性條件。特別是：

- 作為增強 transactions 字典中值出現的交易必須其 lt 值等於其鍵。
- 所有交易必須屬於地址 account_addr 在 *AccountBlock* 中指示的帳戶。
- 如果 $t$ 和 $t'$ 是兩個交易，滿足 $\text{LT}(t) < \text{LT}(t')$，並且它們的鍵在 transactions 中是連續的，意味著沒有交易 $t''$ 滿足 $\text{LT}(t) < \text{LT}(t'') < \text{LT}(t')$，則 $t$ 的最終狀態必須對應於 $t'$ 的初始狀態（它們在梅克更新中明確指示的雜湊必須相等）。
- 如果 $t$ 是具有最小 $\text{LT}(t)$ 的交易，則其初始狀態必須與 *AccountBlock* 的 state_update 中指示的初始狀態一致。
- 如果 $t$ 是具有最大 $\text{LT}(t)$ 的交易，則其最終狀態必須與 *AccountBlock* 的 state_update 中指示的最終狀態一致。
- 交易列表必須非空。

這些條件簡單地表達了帳戶的狀態只能由於執行交易而改變的事實。

**4.2.17. 區塊中所有交易的集合.** 區塊中的所有交易由（參見 **1.2.1**）表示：

```
_ (HashmapAugE 256 AccountBlock Grams) = ShardAccountBlocks;
```

**4.2.18. 所有交易集合的一致性條件.** 同樣，對此結構施加一致性條件，要求鍵 $\xi$ 處的值是地址等於 $\xi$ 的 *AccountBlock*。進一步的一致性條件將此結構與區塊中指示的分片鏈的初始和最終狀態相關聯，要求：

- 如果 *ShardAccountBlock* 沒有鍵 $\xi$，則帳戶 $\xi$ 在區塊的初始和最終狀態中的狀態必須一致（或者必須在兩者中都不存在）。
- 如果 $\xi$ 存在於 *ShardAccountBlock* 中，則其在 *AccountBlock* 中指示的初始和最終狀態必須與分片鏈區塊的初始和最終狀態中指示的狀態匹配，由 *ShardAccounts* 實例表示（參見 **4.1.9**）。

這些條件表示分片鏈狀態確實是由各個獨立的帳戶鏈的狀態所組成的。

## 4.3 交易描述

本節根據 **4.2.4** 中提供的分類，呈現交易描述的特定 TL-B 方案。

**4.3.1. 從交易描述中省略資料的原因.** 對於具有圖靈完備虛擬機來執行智慧合約的區塊鏈而言，交易描述必然是不完整的。實際上，一個真正完整的描述將包含虛擬機在執行每條指令後的所有中間狀態，這些資料無法納入合理大小的區塊鏈區塊中。因此，此類交易的描述可能僅包含步驟總數以及虛擬機初始和最終狀態的雜湊值。對此類交易的驗證必然需要執行智慧合約以重現所有中間步驟和最終結果。

如果我們將虛擬機的所有中間步驟序列壓縮為僅包含初始和最終狀態的雜湊值，那麼根本不需要包含任何交易細節：能夠自行檢查虛擬機執行的驗證者也能夠從初始資料開始檢查交易的所有其他動作，而無需這些細節。

**4.3.2. 將資料包含在交易描述中的原因.** 儘管有上述考量，仍有幾個原因需要在交易描述中引入一些細節：

- 我們希望對交易施加外部一致性條件，以便至少可以在不呼叫虛擬機的情況下快速檢查交易內部的價值流動的有效性以及入站和出站訊息的有效性（參見 **4.2.14**）。這至少保證了區塊鏈中每種加密貨幣總量的不變性，即使它不保證其分配的正確性。
- 我們希望能夠透過檢查交易描述中儲存的資料來追蹤帳戶的主要狀態變化（例如其被建立、啟動或凍結），而無需理清交易的遺漏細節。這簡化了區塊中帳戶鏈和分片鏈狀態之間一致性條件的驗證。
- 最後，某些資訊——例如虛擬機的總步驟數、其初始和最終狀態的雜湊值、消耗的總 gas 以及退出碼——可能會大大簡化 TON 區塊鏈軟體的除錯和實現。（這些資訊將幫助程式設計師理解特定區塊鏈區塊中發生了什麼。）

另一方面，我們希望最小化每個交易的大小，因為我們希望最大化能夠納入每個（有界大小的）區塊中的交易數量。因此，所有不是出於上述原因之一所需的資訊都會被省略。

**4.3.3. 儲存階段的描述.** 儲存階段存在於多種類型的交易中，因此使用該階段的通用表示形式：

```
tr_phase_storage$_ storage_fees_collected:Grams
  storage_fees_due:(Maybe Grams)
  status_change:AccStatusChange
  = TrStoragePhase;

acst_unchanged$0 = AccStatusChange;  // x -> x
acst_frozen$10 = AccStatusChange;    // init -> frozen
acst_deleted$11 = AccStatusChange;   // frozen -> deleted
```

**4.3.4. 信用階段的描述.** 信用階段可能導致收取一些到期款項：

```
tr_phase_credit$_ due_fees_collected:(Maybe Grams)
  credit:CurrencyCollection = TrCreditPhase;
```

due_fees_collected 和 credit 的總和必須等於收到的訊息的價值，如果訊息沒有透過 IHR 接收，則加上其 ihr_fee（否則 ihr_fee 將授予驗證者）。

**4.3.5. 計算階段的描述.** 計算階段包括使用正確的輸入呼叫 TVM。在某些情況下，根本無法呼叫 TVM（例如，如果帳戶不存在、未初始化或被凍結，並且正在處理的入站訊息沒有程式碼或資料欄位，或這些欄位的雜湊值不正確）；這由對應的建構器反映。

```
tr_phase_compute_skipped$0 reason:ComputeSkipReason
  = TrComputePhase;
tr_phase_compute_vm$1 success:Bool msg_state_used:Bool
  account_activated:Bool gas_fees:Grams
  _:^[ gas_used:(VarUInteger 7)
  gas_limit:(VarUInteger 7) gas_credit:(Maybe (VarUInteger 3))
  mode:int8 exit_code:int32 exit_arg:(Maybe int32)
  vm_steps:uint32
  vm_init_state_hash:uint256 vm_final_state_hash:uint256 ]
  = TrComputePhase;
```

```
cskip_no_state$00 = ComputeSkipReason;
cskip_bad_state$01 = ComputeSkipReason;
cskip_no_gas$10 = ComputeSkipReason;
```

TL-B 建構 `_:^[...]` 描述了對包含方括號內列出的欄位的單元的參照。透過這種方式，可以將包含大型記錄的單元中的多個欄位移動到單獨的子單元中。

**4.3.6. 跳過的計算階段.** 如果計算階段已被跳過，可能的原因包括：

- 缺乏購買 gas 的資金。
- 帳戶（不存在、未初始化或被凍結）和訊息中都沒有狀態（即智慧合約程式碼和資料）。
- 傳遞給被凍結或未初始化帳戶的訊息中包含無效狀態（即狀態的雜湊值與預期值不同）。

**4.3.7. 有效的計算階段.** 如果沒有理由跳過計算階段，則會呼叫 TVM 並記錄計算結果。可能的參數如下：

- 當且僅當 `exit_code` 為 0 或 1 時，`success` 旗標才會被設定。
- `msg_state_used` 參數反映訊息中傳遞的狀態是否已被使用。如果已設定，`account_activated` 旗標反映這是否導致先前被凍結、未初始化或不存在的帳戶被啟動。
- `gas_fees` 參數反映驗證者為執行此交易而收取的總 gas 費用。它必須等於當前區塊標頭中的 `gas_used` 和 `gas_price` 的乘積。
- `gas_limit` 參數反映此 TVM 實例的 gas 上限。它等於從入站訊息的價值在信用階段入帳的 Grams 除以當前 gas 價格，或全域每交易 gas 上限中的較小值。
- `gas_credit` 參數僅對外部入站訊息可能為非零。它是可以從帳戶餘額支付的 gas 量或最大 gas 信用額度中的較小值。
- `exit_code` 和 `exit_args` 參數表示 TVM 返回的狀態值。
- `vm_init_state_hash` 和 `vm_final_state_hash` 參數是 TVM 原始狀態和結果狀態的表示雜湊值，`vm_steps` 是 TVM 執行的總步驟數（通常等於執行的指令數加二，包括隱式 `RET`）。[33]

[33]請注意，此記錄不代表帳戶狀態的變化，因為交易在動作階段仍可能被中止。在這種情況下，`vm_final_state_hash` 間接參照的新持久性資料將被捨棄。

**4.3.8. 動作階段的描述.** 動作階段發生在有效的計算階段之後。它嘗試執行 TVM 在計算階段中儲存到動作串列中的動作。它可能會失敗，因為動作串列可能過長、包含無效動作，或包含無法完成的動作（例如，由於資金不足而無法建立具有所需價值的出站訊息）。

```
tr_phase_action$_ success:Bool valid:Bool no_funds:Bool
  status_change:AccStatusChange
  total_fwd_fees:(Maybe Grams) total_action_fees:(Maybe Grams)
  result_code:int32 result_arg:(Maybe int32) tot_actions:int16
  spec_actions:int16 msgs_created:int16
  action_list_hash:uint256 tot_msg_size:StorageUsed
  = TrActionPhase;
```

**4.3.9. 退回階段的描述.**

```
tr_phase_bounce_negfunds$00 = TrBouncePhase;
tr_phase_bounce_nofunds$01 msg_size:StorageUsed
  req_fwd_fees:Grams = TrBouncePhase;
tr_phase_bounce_ok$1 msg_size:StorageUsed
  msg_fees:Grams fwd_fees:Grams = TrBouncePhase;
```

**4.3.10. 普通交易的描述.**

```
trans_ord$0000 storage_ph:(Maybe TrStoragePhase)
  credit_ph:(Maybe TrCreditPhase)
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
  aborted:Bool bounce:(Maybe TrBouncePhase)
  destroyed:Bool
  = TransactionDescr;
```

對此結構施加了幾個一致性條件：

- 當且僅當計算階段不成功時，`action` 才會缺失。
- 如果沒有動作階段或動作階段不成功，則會設定 `aborted` 旗標。
- 僅當設定了 `aborted` 旗標且入站訊息是可退回的時，才會發生退回階段。

**4.3.11. 儲存交易的描述.** 儲存交易僅由一個獨立的儲存階段組成：

```
trans_storage$0001 storage_ph:TrStoragePhase
  = TransactionDescr;
```

**4.3.12. Tick 和 tock 交易的描述.** Tick 和 tock 交易類似於沒有入站訊息的普通交易，因此沒有信用階段或退回階段：

```
trans_tick_tock$001 is_tock:Bool storage:TrStoragePhase
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
  aborted:Bool destroyed:Bool = TransactionDescr;
```

**4.3.13. 分裂準備和安裝交易.** 分裂準備交易類似於主鏈中的 tock 交易，但它必須恰好產生一條特殊的建構器訊息；否則，動作階段將被中止。

```
split_merge_info$_ cur_shard_pfx_len:(## 6)
  acc_split_depth:(## 6) this_addr:uint256 sibling_addr:uint256
  = SplitMergeInfo;
trans_split_prepare$0100 split_info:SplitMergeInfo
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
  aborted:Bool destroyed:Bool
  = TransactionDescr;
trans_split_install$0101 split_info:SplitMergeInfo
  prepare_transaction:^Transaction
  installed:Bool = TransactionDescr;
```

請注意，新的兄弟帳戶 $\xi'$ 的分裂安裝交易參照了先前存在的帳戶 $\xi$ 的對應分裂準備交易。

**4.3.14. 合併準備和安裝交易.** 合併準備交易將帳戶的狀態和餘額轉換為訊息，隨後的合併安裝交易處理此狀態：

```
trans_merge_prepare$0110 split_info:SplitMergeInfo
  storage_ph:TrStoragePhase aborted:Bool
  = TransactionDescr;
trans_merge_install$0111 split_info:SplitMergeInfo
  prepare_transaction:^Transaction
  credit_ph:(Maybe TrCreditPhase)
  compute_ph:TrComputePhase action:(Maybe ^TrActionPhase)
  aborted:Bool destroyed:Bool
  = TransactionDescr;
```

## 4.4 在 TVM 中呼叫智慧合約

本節描述在普通交易和其他交易的計算階段呼叫 TVM 時的確切參數。

**4.4.1. 智慧合約程式碼.** 智慧合約的程式碼通常是帳戶持久性狀態的一部分，至少在帳戶是活躍狀態時如此（參見 **4.1.6**）。然而，被凍結或未初始化（或不存在）的帳戶沒有持久性狀態，可能的例外是帳戶的餘額和其預期狀態的雜湊值（對於未初始化的帳戶等於帳戶地址）。在這種情況下，程式碼必須在交易正在處理的入站訊息的 `init` 欄位中提供（參見 **3.1.7**）。

**4.4.2. 智慧合約持久性資料.** 智慧合約的持久性資料與其程式碼一起保存，並且 **4.4.1** 中所做的類似備註也適用。在這方面，智慧合約的程式碼和持久性資料只是其持久性狀態的兩個部分，它們的區別僅在於 TVM 在智慧合約執行期間處理它們的方式。

**4.4.3. 智慧合約函式庫環境.** 智慧合約的函式庫環境是一個雜湊映射，將 256 位元單元（表示）雜湊值映射到對應的單元本身。當智慧合約執行期間存取外部單元參照時，會在函式庫環境中查找所參照的單元，並且外部單元參照會被透明地替換為找到的單元。

智慧合約呼叫的函式庫環境計算如下：

1. 有關工作鏈的全域函式庫環境從主鏈的當前狀態中取得。[34]
2. 接下來，它會被智慧合約的本地函式庫環境增強，該環境儲存在智慧合約狀態的 `library` 欄位中。只考慮等於對應值單元的雜湊值的 256 位元金鑰。如果金鑰同時存在於全域和本地函式庫環境中，在合併兩個函式庫環境時，本地環境具有優先權。
3. 最後，同樣考慮入站訊息的 `init` 欄位的 `library` 欄位中儲存的訊息函式庫。但請注意，如果帳戶被凍結或未初始化，訊息的 `library` 欄位是帳戶建議狀態的一部分，並在前一步驟中代替本地函式庫環境使用。訊息函式庫的優先權低於本地和全域函式庫環境。

**4.4.4. TVM 的初始狀態.** 在執行智慧合約之前，TVM 的新實例按如下方式初始化：

- 原始 `cc`（當前延續）使用從單元 `code` 建立的單元切片進行初始化，該單元包含如 **4.4.1** 中所述計算的智慧合約程式碼。

[34] 為 TVM 建立共享函式庫的最常見方法是在主鏈中發布對函式庫根單元的參照。

- cp（TVM 代碼頁）設定為零。如果智慧合約想要使用另一個 TVM 代碼頁 $x$，它必須透過使用 SETCODEPAGE $x$ 作為其程式碼的第一條指令來切換到該代碼頁。

- 控制暫存器 c0（返回延續）由帶參數 0 的非凡延續 ec_quit 初始化。執行時，此延續導致 TVM 以退出碼 0 終止。

- 控制暫存器 c1（替代返回延續）由帶參數 1 的非凡延續 ec_quit 初始化。呼叫時，它導致 TVM 以退出碼 1 終止。(請注意，以退出碼 0 或 1 終止被視為成功終止。)

- 控制暫存器 c2（例外處理器）由非凡延續 ec_quit_exc 初始化。呼叫時，它從堆疊中取出頂部整數（等於例外編號），並以等於該整數的退出碼終止 TVM。這樣，預設情況下，所有例外都會以等於例外編號的退出碼終止智慧合約執行。

- 控制暫存器 c3（程式碼字典）由包含智慧合約程式碼的單元初始化，類似於初始當前延續（cc）。

- 控制暫存器 c4（持久性資料的根）由智慧合約的持久性資料初始化。[35]

- 控制暫存器 c5（動作的根）由一個空單元初始化。TVM 的「輸出動作」原語（例如 SENDMSG）使用 c5 來累積在智慧合約成功終止時要執行的動作串列（例如出站訊息）（參見 **4.2.7** 和 **4.2.8**）。

- 控制暫存器 c7（臨時資料的根）由一個單例元組初始化，其唯一組件是一個元組，其中包含帶有智慧合約餘額和其他有用資訊的 *SmartContractInfo* 實例（參見 **4.4.10**）。智慧合約可以用它可能需要的任何其他臨時資料替換臨時資料，特別是 c7 處的元組的所有組件（除了第一個組件）。然而，c7 持有的元組的第一個組件中的 *SmartContractInfo* 的原始內容會被 SENDMSG TVM 原語和 TVM 的其他「輸出動作」原語檢查並有時修改。

- $gas$ 上限 gas $= (g_m, g_l, g_c, g_r)$ 按如下方式初始化：

[35]智慧合約的持久性資料不需要完全載入即可實現此目的。相反，載入根，並且 TVM 可能僅在存取時才從根的參照載入其他單元，從而提供一種虛擬記憶體形式。

- 最大 *gas* 上限 $g_m$ 設定為智慧合約的總 Gram 餘額（在信用階段之後——即與入站訊息的價值結合）除以當前 gas 價格，或每次執行全域 gas 上限中的較小值。[36]
- 當前 *gas* 上限 $g_l$ 設定為入站訊息的 Gram 價值除以 gas 價格，或全域每次執行 gas 上限中的較小值。這樣，始終 $g_l \leq g_m$。對於入站外部訊息 $g_l = 0$，因為它們無法攜帶任何價值。
- *gas* 信用額度 $g_c$ 對於入站內部訊息設定為零，對於入站外部訊息設定為 $g_m$ 或固定的小值（預設外部訊息 gas 信用額度，一個可配置參數）中的較小值。
- 最後，剩餘 *gas* 上限 $g_r$ 自動由 $g_l + g_c$ 初始化。

**4.4.5. 處理內部訊息的 TVM 初始堆疊.** 在如 **4.4.4** 所述初始化 TVM 後，其堆疊透過推入智慧合約的 `main()` 函式的引數來初始化，如下所示：

- 智慧合約的 Gram 餘額 $b$（在將入站訊息的價值入帳之後）作為 nanogram 的整數數量傳遞。
- 入站訊息 $m$ 的 Gram 餘額 $b_m$ 作為 nanogram 的整數數量傳遞。
- 入站訊息 $m$ 作為一個單元傳遞，該單元包含類型 *Message* $X$ 的序列化值，其中 $X$ 是訊息主體的類型。
- 入站訊息的主體 $m_b : X$，等於 $m$ 的 `body` 欄位的值，作為單元切片傳遞。
- 最後，函式選擇器 $s$，一個通常等於零的整數，被推入堆疊。

之後，智慧合約的程式碼（等於其 `c3` 的初始值）被執行。它根據 $s$ 選擇正確的函式，該函式應處理函式的剩餘引數並在之後終止。

**4.4.6. 處理入站外部訊息.** 入站外部訊息的處理類似於 **4.4.4** 和 **4.4.5**，具有以下修改：

- 函式選擇器 $s$ 設定為 $-1$，而不是 0。
- 入站訊息的 Gram 餘額 $b_m$ 始終為 0。
- 初始當前 gas 上限 $g_l$ 始終為 0。但是，初始 gas 信用額度 $g_c > 0$。

[36]全域 gas 上限和 gas 價格都是由主鏈當前狀態決定的可配置參數。

智慧合約必須以 $g_c = 0$ 或 $g_r \geq g_c$ 終止；否則，交易和包含它的區塊無效。建議區塊候選的驗證者或整理者絕不能包含處理無效入站外部訊息的交易。

**4.4.7. 處理 tick 和 tock 交易.** 處理 tick 和 tock 交易（參見 **4.2.4**）的 TVM 堆疊透過推入以下值來初始化：

- 當前帳戶的 Gram 餘額 $b$（以 nanogram 為單位）（一個整數）。
- 當前帳戶在主鏈內的 256 位元地址 $\xi$，由一個無符號整數表示。
- 對於 tick 交易等於 0，對於 tock 交易等於 $-1$ 的整數。
- 函式選擇器 $s$，等於 $-2$。

**4.4.8. 處理分裂準備交易.** 為了處理分裂準備交易（參見 **4.3.13**），TVM 堆疊透過推入以下值來初始化：

- 當前帳戶的 Gram 餘額 $b$。
- 包含 *SplitMergeInfo* 的切片（參見 **4.3.13**）。
- 當前帳戶的 256 位元地址 $\xi$。
- 兄弟帳戶的 256 位元地址 $\tilde{\xi}$。
- 整數 $0 \leq d \leq 63$，等於 $\xi$ 和 $\tilde{\xi}$ 不同的唯一位元的位置。
- 函式選擇器 $s$，等於 $-3$。

**4.4.9. 處理合併安裝交易.** 為了處理合併安裝交易（參見 **4.3.14**），TVM 堆疊透過推入以下值來初始化：

- 當前帳戶的 Gram 餘額 $b$（已與兄弟帳戶的 Gram 餘額結合）。
- 兄弟帳戶的 Gram 餘額 $b'$，取自入站訊息 $m$。
- 來自兄弟帳戶的訊息 $m$，由合併準備交易自動產生。其 `init` 欄位包含兄弟帳戶的最終狀態 $\tilde{S}$。
- 兄弟帳戶的狀態 $\tilde{S}$，由 *StateInit*（參見 **3.1.7**）表示。
- 包含 *SplitMergeInfo* 的切片（參見 **4.3.13**）。

- 當前帳戶的 256 位元地址 $\xi$。
- 兄弟帳戶的 256 位元地址 $\tilde{\xi}$。
- 整數 $0 \leq d \leq 63$，等於 $\xi$ 和 $\tilde{\xi}$ 不同的唯一位元的位置。
- 函式選擇器 $s$，等於 $-4$。

**4.4.10. 智慧合約資訊.** 智慧合約資訊結構 *SmartContractInfo*，傳遞在控制暫存器 `c7` 中包含的元組的第一個組件中，也是一個元組，包含以下資料：

```
[ magic:0x076ef1ea actions:Integer msgs_sent:Integer
  unixtime:Integer block_lt:Integer trans_lt:Integer
  rand_seed:Integer balance_remaining:[Integer (Maybe Cell)]
  myself:MsgAddressInt global_config:(Maybe Cell)
] = SmartContractInfo;
```

換句話說，此元組的第一個組件是一個整數 `magic`，始終等於 `0x076ef1ea`，第二個組件是一個整數 `actions`，最初由零初始化，但每當 TVM 的非 `RAW` 輸出動作原語安裝輸出動作時就遞增一，依此類推。剩餘餘額由一對表示，即一個雙組件元組：第一個組件是 nanogram 餘額，第二個組件是一個具有 32 位元金鑰的字典，表示所有其他貨幣（如果有的話）（參見 **3.1.6**）。

此處的 `rand_seed` 欄位(一個無符號 256 位元整數)從區塊的 `rand_seed`、帳戶地址、正在處理的入站訊息的雜湊值（如果有）和交易邏輯時間 `trans_lt` 開始確定性地初始化。

**4.4.11. 輸出動作的序列化.** 智慧合約的輸出動作累積在儲存於控制暫存器 `c5` 中的連結串列中。輸出動作串列序列化為類型 *OutList* $n$ 的值，其中 $n$ 是串列的長度：

```
out_list_empty$_ = OutList 0;
out_list$_ {n:#} prev:^(OutList n) action:OutAction
  = OutList (n + 1);
action_send_msg#0ec3c86d out_msg:^(Message Any) = OutAction;
action_set_code#ad4de08e new_code:^Cell = OutAction;
```

# 5 區塊佈局

本章呈現 TON 區塊鏈使用的區塊佈局，結合前面章節中分別描述的資料結構，以產生分片鏈區塊的完整描述。除了定義分片鏈區塊由單元樹表示的 TL-B 方案之外，本章還描述了結果單元集合（集合）的確切序列化格式，這對於將分片鏈區塊表示為檔案是必要的。

主鏈區塊類似於分片鏈區塊，但具有一些額外的欄位。必要的修改在 **5.2** 中單獨討論。

## 5.1 分片鏈區塊佈局

本節列出分片鏈區塊和分片鏈狀態中必須包含的資料結構，並以分片鏈區塊的正式 TL-B 方案作為結論。

**5.1.1. 分片鏈狀態的組件.** 分片鏈狀態包括：

- *ShardAccounts*，分片鏈狀態的分割部分（參見 **1.2.2**），包含分配給此分片的所有帳戶的狀態（參見 **4.1.9**）。
- *OutMsgQueue*，分片鏈的輸出訊息佇列（參見 **3.3.6**）。
- *SharedLibraries*，分片鏈的所有共享函式庫的描述（目前僅在主鏈中為非空）。
- 狀態最後修改的邏輯時間和 unixtime。
- 分片的總餘額。
- 對最近主鏈區塊的雜湊參照，間接描述主鏈的狀態，並透過它描述 TON 區塊鏈的所有其他分片鏈的狀態（參見 **1.5.2**）。

**5.1.2. 分片鏈區塊的組件.** 分片鏈區塊必須包含：

- 驗證者簽章串列（參見 **1.2.6**），它相對於區塊的所有其他內容是外部的。
- *BlockHeader*，包含有關區塊的一般資訊（參見 **1.2.5**）
- 對同一分片鏈的緊接在前的一個或多個區塊以及最近的主鏈區塊的雜湊參照。

- *InMsgDescr* 和 *OutMsgDescr*，入站和出站訊息描述器（參見 **3.2.8** 和 **3.3.5**）。
- *ShardAccountBlocks*，區塊中處理的所有交易的集合（參見 **4.2.17**），以及分配給分片的帳戶的狀態的所有更新。這是分片鏈區塊的分割部分（參見 **1.2.2**）。
- 價值流動，描述從同一分片鏈的前面區塊和入站訊息匯入的總價值、由出站訊息匯出的總價值、驗證者收取的總費用以及分片中剩餘的總價值。
- 分片鏈狀態的梅克更新（參見 [4, 3.1]）。此類梅克更新包含相對於區塊的初始和最終分片鏈狀態的雜湊值，以及在處理區塊時建立的最終狀態的所有新單元。[37]

**5.1.3. 所有工作鏈的區塊佈局通用部分.** 回想一下，不同的工作鏈可以定義自己的訊息處理規則、其他類型的交易、狀態的其他組件以及序列化所有這些資料的其他方式。然而，區塊及其狀態的某些組件必須對所有工作鏈通用，以保持不同工作鏈之間的互通性。此類通用組件包括：

- *OutMsgQueue*，分片鏈的出站訊息佇列，由鄰近的分片鏈掃描以查找發送給它們的訊息。
- *InMsgDescr* 的外部結構作為雜湊映射，具有等於匯入訊息的雜湊值的 256 位元金鑰。(入站訊息描述器本身不需要具有相同的結構。)
- 區塊標頭中識別分片鏈和區塊的某些欄位，以及從區塊標頭到此清單中指示的其他資訊的路徑。
- 價值流動資訊。

**5.1.4. 分片鏈狀態的 TL-B 方案.** 分片鏈狀態（參見 **1.2.1** 和 **5.1.1**）根據以下 TL-B 方案序列化：

```
ext_blk_ref$_ start_lt:uint64 end_lt:uint64
  seq_no:uint32 hash:uint256 = ExtBlkRef;

master_info$_ master:ExtBlkRef = BlkMasterInfo;
```

[37]原則上，TON 區塊鏈的實驗版本可能選擇僅保留分片鏈的初始和最終狀態的雜湊值。梅克更新會增加區塊大小，但對於想要保留和更新其分片鏈狀態副本的全節點來說非常方便。否則，全節點將不得不重複區塊中包含的所有計算，以自行計算分片鏈的更新狀態。

```
shard_ident$00 shard_pfx_bits:(## 6)
  workchain_id:int32 shard_prefix:uint64 = ShardIdent;

shard_state shard_id:ShardIdent
  out_msg_queue:OutMsgQueue
  total_balance:CurrencyCollection
  total_validator_fees:CurrencyCollection
  accounts:ShardAccounts
  libraries:(HashmapE 256 LibDescr)
  master_ref:(Maybe BlkMasterInfo)
  custom:(Maybe ^McStateExtra)
  = ShardState;
```

custom 欄位通常僅存在於主鏈中，並包含所有主鏈特定資料。然而，其他工作鏈可以使用相同的單元參照來參照它們的特定狀態資料。

**5.1.5. 共享函式庫描述.** 共享函式庫目前僅可存在於主鏈區塊中。它們由 $HashmapE(256, LibDescr)$ 的實例描述，其中 256 位元金鑰是函式庫的表示雜湊值，並且 $LibDescr$ 描述一個函式庫：

```
shared_lib_descr$00 lib:^Cell publishers:(Hashmap 256 True)
  = LibDescr;
```

此處 publishers 是一個雜湊映射，其金鑰等於已發布對應共享函式庫的所有帳戶的地址。只要至少一個帳戶將共享函式庫保留在其已發布的函式庫集合中，該共享函式庫就會被保留。

**5.1.6. 未簽章分片鏈區塊的 TL-B 方案.** 未簽章（參見 **1.2.6**）分片鏈區塊的精確格式由以下 TL-B 方案給出：

```
block_info version:uint32
  not_master:(## 1)
  after_merge:(## 1) before_split:(## 1) flags:(## 13)
  seq_no:# vert_seq_no:#
  shard:ShardIdent gen_utime:uint32
  start_lt:uint64 end_lt:uint64
  master_ref:not_master?^BlkMasterInfo
  prev_ref:seq_no?^(BlkPrevInfo after_merge)
  prev_vert_ref:vert_seq_no?^(BlkPrevInfo 0)
```

```
  = BlockInfo;

prev_blk_info#_ {merged:#} prev:ExtBlkRef
  prev_alt:merged?ExtBlkRef = BlkPrevInfo merged;

unsigned_block info:^BlockInfo value_flow:^ValueFlow
  state_update:^(MERKLE_UPDATE ShardState)
  extra:^BlockExtra = Block;

block_extra in_msg_descr:^InMsgDescr
  out_msg_descr:^OutMsgDescr
  account_blocks:ShardAccountBlocks
  rand_seed:uint256
  custom:(Maybe ^McBlockExtra) = BlockExtra;
```

custom 欄位通常僅存在於主鏈中，並包含所有主鏈特定資料。然而，其他工作鏈可以使用相同的單元參照來參照它們的特定區塊資料。

**5.1.7. 透過區塊的總價值流動的描述.** 透過區塊的總價值流動根據以下 TL-B 方案序列化：

```
value_flow _:^[ from_prev_blk:CurrencyCollection
  to_next_blk:CurrencyCollection
  imported:CurrencyCollection
  exported:CurrencyCollection ]
  fees_collected:CurrencyCollection
  _:^[
  fees_imported:CurrencyCollection
  created:CurrencyCollection
  minted:CurrencyCollection
  ] = ValueFlow;
```

回想一下，_:^[...] 是一個 TL-B 建構，表示一組欄位已被移動到單獨的單元中。最後三個欄位可能僅在主鏈區塊中為非零。

**5.1.8. 已簽章的分片鏈區塊.** 已簽章的分片鏈區塊只是未簽章區塊加上驗證者簽章的集合：

```
ed25519_signature#5 R:uint256 s:uint256 = CryptoSignature;
```

```
signed_block block:^Block blk_serialize_hash:uint256
  signatures:(HashmapE 64 CryptoSignature)
  = SignedBlock;
```

未簽章區塊 `block` 的序列化雜湊 `blk_serialize_hash` 本質上是區塊特定序列化為八位元組字串的雜湊值（參見 **5.3.12** 以獲得更詳細的解釋）。在 `signatures` 中收集的簽章是使用驗證者私鑰對區塊 `block` 的 256 位元表示雜湊值與其 256 位元序列化雜湊值 `blk_serialize_hash` 的串接的 SHA256 所做的 Ed25519 簽章（參見 **A.3**）。字典 `signatures` 中的 64 位元金鑰表示對應驗證者公鑰的前 64 位元。

**5.1.9. 已簽章區塊的序列化.** 序列化和簽章區塊的整體程序可以描述如下：

1. 產生未簽章區塊 $B$，將其轉換為完整的單元集合（參見 **5.3.2**），並序列化為八位元組字串 $S_B$。
2. 驗證者簽署 256 位元組合雜湊值

$$H_B := \text{SHA256}\big(\text{HASH}_\infty(B).\,\text{HASH}_M(S_B)\big) \tag{18}$$

   該雜湊值是 $B$ 的表示雜湊值和其序列化 $S_B$ 的梅克雜湊值的組合。
3. 如上所述（參見 **5.1.8**），從 $B$ 和這些驗證者簽章產生已簽章的分片鏈區塊 $\tilde{B}$。
4. 此已簽章區塊 $\tilde{B}$ 被轉換為不完整的單元集合，該集合僅包含驗證者簽章，但未簽章區塊本身不在此單元集合中，成為其唯一缺失的單元。
5. 此不完整的單元集合被序列化，並且其序列化被前置到先前建構的未簽章區塊的序列化之前。

結果是已簽章區塊序列化為八位元組字串。它可以透過網路傳播或儲存到磁碟檔案中。

## 5.2 主鏈區塊佈局

主鏈區塊與基礎工作鏈的分片鏈區塊非常相似。本節列出了從 **5.1** 中給出的分片鏈區塊描述獲得主鏈區塊描述所需的一些修改。

**5.2.1. 主鏈狀態中存在的額外組件.** 除了 **5.1.1** 中列出的組件之外，主鏈狀態還必須包含：

- *ShardHashes* — 描述當前分片配置，並包含對應分片鏈的最新區塊的雜湊值。
- *ShardFees* — 描述每個分片鏈的驗證者收取的總費用。
- *ShardSplitMerge* — 描述未來的分片分裂/合併事件。它被序列化為 *ShardHashes* 的一部分。
- *ConfigParams* — 描述 TON 區塊鏈所有可配置參數的值。

**5.2.2. 主鏈區塊中存在的額外組件.** 除了 **5.1.2** 中列出的組件之外，每個主鏈區塊還必須包含：

- *ShardHashes* — 描述當前分片配置，並包含對應分片鏈的最新區塊的雜湊值。(請注意，此組件也存在於主鏈狀態中。)

**5.2.3. *ShardHashes* 的描述.** *ShardHashes* 由一個字典表示，其金鑰為 32 位元 *workchain_id*，值為「分片二元樹」，由 TL-B 類型 *BinTree ShardDescr* 表示。此分片二元樹的每個葉子包含類型 *ShardDescr* 的值，該值透過指示序列號 `seq_no`、邏輯時間 `lt` 和對應分片鏈的最新（已簽章）區塊的雜湊值 `hash` 來描述單個分片。

```
bt_leaf$0 {X:Type} leaf:X = BinTree X;
bt_fork$1 {X:Type} left:^(BinTree X) right:^(BinTree X)
          = BinTree X;

fsm_none$0 = FutureSplitMerge;
fsm_split$10 mc_seqno:uint32 = FutureSplitMerge;
fsm_merge$11 mc_seqno:uint32 = FutureSplitMerge;

shard_descr$_ seq_no:uint32 lt:uint64 hash:uint256
  split_merge_at:FutureSplitMerge = ShardDescr;

_ (HashmapE 32 ^(BinTree ShardDescr)) = ShardHashes;
```

`fsm_split` 和 `fsm_merge` 的 `mc_seqno` 欄位用於通知未來的分片合併或分裂事件。參照主鏈區塊且序列號不超過（但不包括）`mc_seqno` 中指示的序

列號的分片鏈區塊以通常方式產生。一旦達到指示的序列號，就必須發生分片合併或分裂事件。

請注意，主鏈本身從 *ShardHashes* 中省略（即 32 位元索引 $-1$ 在此字典中缺失）。

**5.2.4. *ShardFees* 的描述.** *ShardFees* 是一個主鏈結構，用於反映分片鏈的驗證者到目前為止收取的總費用。此結構中反映的總費用透過將它們入帳到一個特殊帳戶在主鏈中累積，該帳戶的地址是一個可配置參數。通常，此帳戶是計算並向所有驗證者分配獎勵的智慧合約。

```
bta_leaf$0 {X:Type} {Y:Type} leaf:X extra:Y = BinTreeAug X Y;
bta_fork$1 {X:Type} {Y:Type} left:^(BinTreeAug X Y)
           right:^(BinTreeAug X Y) extra:Y = BinTreeAug X Y;

_ (HashmapAugE 32 ^(BinTreeAug True CurrencyCollection)
  CurrencyCollection) = ShardFees;
```

*ShardFees* 的結構類似於 *ShardHashes*(參見 **5.2.3**)，但所涉及的字典和二元樹透過貨幣值增強，等於對應分片鏈區塊的最終狀態的 `total_validator_fees` 值。在 *ShardFees* 根部聚合的值與主鏈狀態的 `total_validator_fees` 一起相加，產生總 TON 區塊鏈驗證者費用。從主鏈區塊的初始狀態到最終狀態，*ShardFees* 根部聚合值的增加反映在該主鏈區塊的價值流動中的 `fees_imported` 中。

**5.2.5. *ConfigParams* 的描述.** 回想一下，可配置參數或配置字典是一個字典 `config`，具有 32 位元金鑰，保存在配置智慧合約 $\gamma$ 的持久性資料的第一個單元參照中（參見 **1.6**）。配置智慧合約的地址 $\gamma$ 和配置字典的副本在 *ConfigParams* 結構的 `config_addr` 和 `config` 欄位中複製，明確包含在主鏈狀態中以便於存取可配置參數的當前值（參見 **1.6.3**）：

```
_ config_addr:uint256 config:^(Hashmap 32 ^Cell)
  = ConfigParams;
```

**5.2.6. 主鏈狀態資料.** 主鏈狀態特定的資料收集到 *McStateExtra* 中，已在 **5.1.4** 中提到：

```
masterchain_state_extra#cc1f
  shard_hashes:ShardHashes
  shard_fees:ShardFees
  config:ConfigParams
= McStateExtra;
```

**5.2.7. 主鏈區塊資料.** 同樣，主鏈區塊特定的資料收集到 *McBlockExtra* 中：

```
masterchain_block_extra#cc9f
  shard_hashes:ShardHashes
= McBlockExtra;
```

## 5.3 單元集合的序列化

前一節中提供的描述定義了分片鏈區塊表示為單元樹的方式。然而，這個單元樹需要序列化為檔案，適合磁碟儲存或網路傳輸。本節討論將單元樹、DAG 或單元集合序列化為八位元組字串的標準方法。

**5.3.1. 將單元樹轉換為單元集合.** 回想一下，任意（相依）代數資料型別的值在 TON 區塊鏈中由單元樹表示。透過識別樹中相同的單元，將此類單元樹轉換為單元的有向無環圖或 *DAG*。之後，我們可以將每個單元的每個參照替換為所參照單元的 32 位元組表示雜湊值，並獲得單元集合。按照慣例，原始單元樹的根是結果單元集合的標記元素，以便接收此單元集合並知道標記元素的任何人都可以重建原始單元 DAG，因此也可以重建原始單元樹。

**5.3.2. 完整的單元集合.** 如果單元集合包含其任何單元所參照的所有單元，我們就說該單元集合是完整的。換句話說，完整的單元集合沒有任何對該單元集合外部單元的「未解析」雜湊參照。在大多數情況下，我們只需要序列化完整的單元集合。

**5.3.3. 單元集合內部的內部參照.** 如果屬於單元集合 $B$ 的單元 $c$ 的參照，如果此參照所參照的單元 $c_i$ 也屬於 $B$，我們就說該參照是內部的（相對於 $B$）。否則，該參照稱為外部參照。當且僅當單元集合的所有組成單元的所有參照都是內部的時，該單元集合才是完整的。

**5.3.4. 為單元集合中的單元分配索引.** 設 $c_0$、…、$c_{n-1}$ 為屬於單元集合 $B$ 的 $n$ 個不同單元。我們可以按某種順序列出這些單元，然後分配從 0 到 $n-1$ 的索引，使得單元 $c_i$ 獲得索引 $i$。排序單元的一些選項是：

- 按表示雜湊值排序單元。那麼，當 $i < j$ 時，$\textsc{Hash}(c_i) < \textsc{Hash}(c_j)$。
- 拓撲順序：如果單元 $c_i$ 參照單元 $c_j$，則 $i < j$。一般來說，同一單元集合有多個拓撲順序。建構拓撲順序有兩種標準方法：

- 深度優先順序：對從根（即標記單元）開始的單元有向無環圖應用深度優先搜尋，並按訪問順序列出單元。
- 廣度優先順序：與上述相同，但應用廣度優先搜尋。

請注意，拓撲順序始終將索引 0 分配給從單元樹建構的單元集合的根單元。在大多數情況下，我們選擇使用拓撲順序，或者如果我們想更具體，則使用深度優先順序。

如果單元按拓撲順序列出，則驗證單元集合中沒有循環參照是立即的。另一方面，按表示雜湊值排序單元簡化了驗證序列化單元集合中沒有重複項的過程。

**5.3.5. 序列化過程概要.** 由 $n$ 個單元組成的單元集合 $B$ 的序列化過程可以概述如下：

1. 按拓撲順序列出 $B$ 中的單元：$c_0$、$c_1$、…、$c_{n-1}$。那麼 $c_0$ 是 $B$ 的根單元。
2. 選擇整數 $s$，使得 $n \leq 2^s$。以標準方式將每個單元 $c_i$ 表示為整數個八位元組（參見 **1.1.3** 或 [4, 3.1.4]），但使用無符號大端序 $s$ 位元整數 $j$ 而不是雜湊值 $\textsc{Hash}(c_j)$ 來表示對單元 $c_j$ 的內部參照（參見下面的 **5.3.6**）。
3. 按 $i$ 的遞增順序串接如此獲得的單元 $c_i$ 的表示。
4. 可選地，可以建構一個索引，該索引由 $n+1$ 個 $t$ 位元整數項目 $L_0$、…、$L_n$ 組成，其中 $L_i$ 是 $j \leq i$ 的單元 $c_j$ 的表示的總長度（以八位元組為單位），並且選擇整數 $t \geq 0$ 使得 $L_n \leq 2^t$。
5. 單元集合的序列化現在由指示序列化精確格式的魔術數字組成，後跟整數 $s \geq 0$、$t \geq 0$、$n \leq 2^s$、由 $\lceil (n+1)t/8 \rceil$ 個八位元組組成的可選索引，以及具有單元表示的 $L_n$ 個八位元組。
6. 可選地，可以將 CRC32 附加到序列化以用於完整性驗證目的。

如果包含索引，序列化單元集合中的任何單元 $c_i$ 都可以透過其索引 $i$ 輕鬆存取，而無需反序列化所有其他單元，甚至無需將整個序列化單元集合載入記憶體。

**5.3.6. 單元集合中一個單元的序列化.** 更精確地說，假設 $s$ 是八的倍數（通常 $s = 8$、16、24 或 32），每個單獨的單元 $c = c_i$ 按如下方式序列化：

1. 兩個描述器位元組 $d_1$ 和 $d_2$ 的計算方式類似於 [4, 3.1.4]，透過設定 $d_1 = r + 8s + 16h + 32l$ 和 $d_2 = \lfloor b/8 \rfloor + \lceil b/8 \rceil$ 來計算，其中：

   - $0 \le r \le 4$ 是單元 $c$ 中存在的單元參照數量；如果 $c$ 不在正在序列化的單元集合中，並且僅由其雜湊值表示，則 $r = 7$。[38]
   - $0 \le b \le 1023$ 是單元 $c$ 中的資料位元數。
   - $0 \le l \le 3$ 是單元 $c$ 的層級（參見 [4, 3.1.3]）。
   - 對於特殊單元，$s = 1$；對於普通單元，$s = 0$。
   - 如果單元的雜湊值明確包含在序列化中，則 $h = 1$；否則 $h = 0$。（當 $r = 7$ 時，我們必須始終有 $h = 1$。）

   對於缺失單元（即外部參照），僅存在 $d_1$，始終等於 $23 + 32l$。

2. 兩個位元組 $d_1$ 和 $d_2$（如果 $r < 7$）或一個位元組 $d_1$（如果 $r = 7$）開始單元 $c$ 的序列化。

3. 如果 $h = 1$，則序列化由 $l + 1$ 個 32 位元組的 $c$ 的更高雜湊值繼續（參見 [4, 3.1.6]）：$\text{HASH}_1(c)$、…、$\text{HASH}_{l+1}(c) = \text{HASH}_\infty(c)$。

4. 之後，透過將 $b$ 個資料位元分成 8 位元組並將每個組解釋為 $0 \ldots 255$ 範圍內的大端序整數來序列化 $\lceil b/8 \rceil$ 個資料位元組。如果 $b$ 不能被 8 整除，則資料位元首先由一個二進位 `1` 和最多六個二進位 `0` 增強，以使資料位元數能被八整除。[39]

5. 最後，透過 $r$ 個 $s$ 位元大端序整數 $j_1$、…、$j_r$ 編碼對單元 $c_{j_1}$、…、$c_{j_r}$ 的 $r$ 個單元參照。[40]

**5.3.7. 單元集合序列化方案的分類.** 單元集合的序列化方案必須指定以下參數：

- 前置到序列化的 4 位元組魔術數字。

[38]請注意，這些「缺失單元」與函式庫參照和外部參照單元不同，後者是特殊單元的類型（參見 [4, 3.1.7]）。相比之下，缺失單元僅為了序列化不完整的單元集合而引入，並且永遠不能由 TVM 處理。

[39]請注意，特殊單元（具有 $s = 1$）始終具有 $b \ge 8$，單元類型編碼在前八個資料位元中（參見 [4, 3.1.7]）。

[40]如果單元集合不完整，這些單元參照中的一些可能參照單元集合中缺失的單元 $c'$。在這種情況下，具有 $r = 7$ 的特殊「缺失單元」被包含在單元集合中並被分配一些索引 $j$。然後使用這些索引來表示對缺失單元的參照。

- 用於表示單元索引的位元數 $s$。通常 $s$ 是八的倍數（例如 8、16、24 或 32）。
- 用於表示單元序列化偏移量的位元數 $t$（參見 **5.3.5**）。通常 $t$ 也是八的倍數。
- 指示是否存在具有單元序列化偏移量 $L_0$、…、$L_n$ 的索引的旗標。此旗標可以透過在索引不存在時設定 $t = 0$ 來與 $t$ 組合。
- 指示整個序列化的 CRC32-C 是否附加到它以用於完整性驗證目的的旗標。

**5.3.8. 單元集合序列化中存在的欄位.** 除了 **5.3.7** 中列出的由單元集合序列化方案的選擇固定的值之外，特定單元集合的序列化還必須指定以下參數：

- 序列化中存在的單元總數 $n$。
- 序列化中存在的「根單元」數量 $k \leq n$。根單元本身是 $c_0$、…、$c_{k-1}$。單元集合中存在的所有其他單元預期可透過從根單元開始的參照鏈到達。
- 「缺失單元」的數量 $l \leq n - k$，它們表示實際上不在此單元集合中但從中參照的單元。缺失單元本身由 $c_{n-l}$、…、$c_{n-1}$ 表示，並且只有這些單元可以（並且也必須）具有 $r = 7$。完整的單元集合具有 $l = 0$。
- 所有單元的序列化的總長度（以位元組為單位）$L_n$。如果存在索引，則 $L_n$ 可能不會明確儲存，因為它可以作為索引的最後一個項目恢復。

**5.3.9. 序列化單元集合的 TL-B 方案.** 可以使用幾個 TL-B 建構器將單元集合序列化為八位元組（即 8 位元位元組）序列。目前用於序列化新單元集合的唯一一個是

```
serialized_boc#b5ee9c72 has_idx:(## 1) has_crc32c:(## 1)
  has_cache_bits:(## 1) flags:(## 2) { flags = 0 }
  size:(## 3) { size <= 4 }
  off_bytes:(## 8) { off_bytes <= 8 }
  cells:(##(size * 8))
  roots:(##(size * 8)) { roots >= 1 }
  absent:(##(size * 8)) { roots + absent <= cells }
```

```
  tot_cells_size:(##(off_bytes * 8))
  root_list:(roots * ##(size * 8))
  index:has_idx?(cells * ##(off_bytes * 8))
  cell_data:(tot_cells_size * [ uint8 ])
  crc32c:has_crc32c?uint32
  = BagOfCells;
```

欄位 `cells` 是 $n$，`roots` 是 $k$，`absent` 是 $l$，`tot_cells_size` 是 $L_n$（所有單元的序列化的總大小，以位元組為單位）。如果存在索引，參數 $s/8$ 和 $t/8$ 分別序列化為 `size` 和 `off_bytes`，並且設定旗標 `has_idx`。索引本身包含在 `index` 中，僅在設定 `has_idx` 時存在。欄位 `root_list` 包含單元集合的根節點的（從零開始的）索引。

單元集合反序列化函式中仍支援兩個較舊的建構器：

```
serialized_boc_idx#68ff65f3 size:(## 8) { size <= 4 }
  off_bytes:(## 8) { off_bytes <= 8 }
  cells:(##(size * 8))
  roots:(##(size * 8)) { roots = 1 }
  absent:(##(size * 8)) { roots + absent <= cells }
  tot_cells_size:(##(off_bytes * 8))
  index:(cells * ##(off_bytes * 8))
  cell_data:(tot_cells_size * [ uint8 ])
  = BagOfCells;

serialized_boc_idx_crc32c#acc3a728 size:(## 8) { size <= 4 }
  off_bytes:(## 8) { off_bytes <= 8 }
  cells:(##(size * 8))
  roots:(##(size * 8)) { roots = 1 }
  absent:(##(size * 8)) { roots + absent <= cells }
  tot_cells_size:(##(off_bytes * 8))
  index:(cells * ##(off_bytes * 8))
  cell_data:(tot_cells_size * [ uint8 ])
  crc32c:uint32 = BagOfCells;
```

**5.3.10. 在檔案中儲存編譯的 TVM 程式碼.** 請注意，上述序列化單元集合的程序可用於序列化編譯的智慧合約和其他 TVM 程式碼。必須定義類似於以下的 TL-B 建構器：

```
compiled_smart_contract
```

```
  compiled_at:uint32 code:^Cell data:^Cell
  description:(Maybe ^TinyString)
  _:^[ source_file:(Maybe ^TinyString)
       compiler_version:(Maybe ^TinyString) ]
  = CompiledSmartContract;

tiny_string#_ len:(#<= 126) str:(len * [ uint8 ]) = TinyString;
```

然後，編譯的智慧合約可以由類型 *CompiledSmartContract* 的值表示，轉換為單元樹，然後轉換為單元集合，然後使用 **5.3.9** 中列出的建構器之一進行序列化。然後可以將產生的八位元組字串寫入帶有後綴.tvc（「TVM 智慧合約」）的檔案，並且可以使用此檔案來分發編譯的智慧合約，將其下載到錢包應用程式中以部署到 TON 區塊鏈中，等等。

**5.3.11. 八位元組字串的梅克雜湊值.** 在某些情況下，我們必須定義長度為 $|s|$ 的任意八位元組字串 $s$ 的梅克雜湊值 $\textsc{Hash}_M(s)$。我們按如下方式執行此操作：

- 如果 $|s| \le 256$ 個八位元組，則 $s$ 的梅克雜湊值只是其 $\textsc{sha256}$：

$$\textsc{Hash}_M(s) := \textsc{sha256}(s) \quad \text{如果 } |s| \le 256。 \tag{19}$$

- 如果 $|s| > 256$，設 $n = 2^k$ 為小於 $|s|$ 的最大二次冪(即 $k := \lfloor \log_2(|s|-1) \rfloor$，$n := 2^k$)。如果 $s'$ 是 $s$ 的長度為 $n$ 的前綴，$s''$ 是 $s$ 的長度為 $|s| - n$ 的後綴，使得 $s$ 是 $s'$ 和 $s''$ 的串接 $s'.s''$，我們定義

$$\textsc{Hash}_M(s) := \textsc{sha256}\big(\textsc{int}_{64}(|s|).\, \textsc{Hash}_M(s').\, \textsc{Hash}_M(s'')\big) \tag{20}$$

  換句話說，我們串接 $|s|$ 的 64 位元大端序表示以及遞迴計算的 $s'$ 和 $s''$ 的梅克雜湊值，並計算結果字串的 $\textsc{sha256}$。

可以檢查，對於長度小於 $2^{64} - 2^{56}$ 的八位元組字串 $s$ 和 $t$，$\textsc{Hash}_M(s) = \textsc{Hash}_M(t)$ 意味著 $s = t$，除非已找到 $\textsc{sha256}$ 的雜湊碰撞。

**5.3.12. 區塊的序列化雜湊值.** **5.3.11** 的建構特別應用於表示未簽章分片鏈或主鏈區塊的單元集合的序列化。驗證者不僅簽署未簽章區塊的表示雜湊值，還簽署未簽章區塊的「序列化雜湊值」，定義為未簽章區塊序列化的 $\textsc{Hash}_M$。這樣，驗證者證明此八位元組字串確實是對應區塊的序列化。

# References

[1] Daniel J. Bernstein, *Curve25519: New Diffie–Hellman Speed Records* (2006), in: M. Yung, Ye. Dodis, A. Kiayas et al, *Public Key Cryptography*, Lecture Notes in Computer Science **3958**, pp. 207–228. Available at `https://cr.yp.to/ecdh/curve25519-20060209.pdf`.

[2] Daniel J. Bernstein, Niels Duif, Tanja Lange et al., *High-speed high-security signatures* (2012), *Journal of Cryptographic Engineering* **2** (2), pp. 77–89. Available at `https://ed25519.cr.yp.to/ed25519-20110926.pdf`.

[3] N. Durov, *Telegram Open Network*, 2017.

[4] N. Durov, *Telegram Open Network Virtual Machine*, 2018.

# A 橢圓曲線密碼學

本附錄包含目前在 TON 中使用的橢圓曲線密碼學的正式描述，特別是在 TON 區塊鏈和 TON 網路中。

TON 使用兩種形式的橢圓曲線密碼學：Ed25519 用於密碼學 Schnorr 簽章，而 Curve25519 用於非對稱密碼學。這些曲線以標準方式使用（如 D. Bernstein 的原始文章 [1] 和 [2] 以及 RFC 7748 和 8032 中定義）；然而，必須解釋一些 TON 特定的序列化細節。這些曲線對 TON 的一個獨特適應是 TON 支援將 Ed25519 金鑰自動轉換為 Curve25519 金鑰，以便相同的金鑰可用於簽章和非對稱密碼學。

## A.1 橢圓曲線

本節收集了與橢圓曲線密碼學相關的有限體上橢圓曲線的一些一般事實。

**A.1.1. 有限體.** 我們考慮有限體上的橢圓曲線。對於 Curve25519 和 Ed25519 演算法的目的，我們將主要關注質數域 $k := \mathbb{F}_p$ 的餘數體（模 $p$ 的餘數）上的橢圓曲線，其中 $p = 2^{255} - 19$ 是質數，以及 $\mathbb{F}_p$ 的有限擴張 $\mathbb{F}_q$，特別是二次擴張 $\mathbb{F}_{p^2}$。[41]

**A.1.2. 橢圓曲線.** 體 $k$ 上的橢圓曲線 $E = (E, O)$ 是屬 $g = 1$ 的幾何積分光滑射影曲線 $E/k$，以及一個標記的 $k$ 有理點 $O \in E(k)$。眾所周知，體 $k$ 上的橢圓曲線 $E$ 可以用（廣義）Weierstrass 形式表示：

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad \text{對於某些 } a_1、\ldots、a_6 \in k。 \tag{21}$$

更準確地說，這只是橢圓曲線的仿射部分，以座標 $(x, y)$ 寫成。對於 $k$ 的任何體擴張 $K$，$E(K)$ 由方程式 (21) 的所有解 $(x, y) \in K^2$ 組成，稱為$E(K)$ 的有限點，以及無窮遠點，即標記點 $O$。

**A.1.3. 齊次座標中的 Weierstrass 形式.** 在齊次座標 $[X : Y : Z]$ 中，(21) 對應於

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{22}$$

當 $Z \neq 0$ 時，我們可以設定 $x := X/Z$、$y := Y/Z$，並獲得 (21) 的解 $(x, y)$（即 $E$ 的有限點）。另一方面，具有 $Z = 0$ 的 (22) 的唯一解（直到比例）是 $[0 : 1 : 0]$；這是無窮遠點 $O$。

[41] 對於接近二次冪的模數 $p$，模 $p$ 的算術可以非常有效地實現。另一方面，模 $2^{255} - 19$ 的餘數可以由 255 位元整數表示。這就是 D. Bernstein 選擇這個特定 $p$ 值的原因。

**A.1.4. 標準 Weierstrass 形式.** 當體 $k$ 的特徵 $\operatorname{char} k$ 為 $\neq 2$、3 時，可以藉助線性變換 $y' := y + a_1x/2 + a_3/2$、$x' := x + a_2/3$ 來簡化 (21) 或 (22) 的 Weierstrass 形式，從而使 $a_1 = a_3 = a_2 = 0$ 並獲得

$$y^2 = x^3 + a_4x + a_6 \tag{23}$$

和

$$Y^2Z = X^3 + a_4XZ^2 + a_6Z^3 \tag{24}$$

當且僅當三次多項式 $P(x) := x^3 + a_4x + a_6$ 沒有重根時，即當判別式 $D := -4a_4^3 - 27a_6^2$ 非零時，此類方程式定義橢圓曲線。

**A.1.5. 橢圓曲線 $E$ 上點的加法.** 設 $K$ 為體 $k$ 的體擴張，並設 $E = (E, O)$ 為定義在 $k$ 上的 Weierstrass 形式中的任何橢圓曲線。那麼任何直線 $l \subset \mathbb{P}^2_K$ 恰好與橢圓曲線 $E_{(K)}$（這是曲線 $E$ 對體 $K$ 的基變換，即由較大體 $K$ 上的相同方程式定義的曲線）相交於三個點 $P$、$Q$、$R$，按重數考慮。我們透過要求

$$P + Q + R = O \quad \text{當對於某條直線 } l \subset \mathbb{P}^2_K \text{ 有 } \{P, Q, R\} = l \cap E \text{ 時。} \tag{25}$$

來定義橢圓曲線 $E$ 上的點的加法（或更確切地說，其 $K$ 值點 $E(K)$ 的加法）。眾所周知，此要求在橢圓曲線 $E$ 的點上定義了唯一的交換律 $[+] : E \times_k E \to E$，以 $O$ 為其中性元素。當橢圓曲線 $E$ 由其 Weierstrass 形式 (21) 表示時，可以寫出明確的公式，將橢圓曲線 $E$ 的兩個 $K$ 值點 $P$、$Q \in E(K)$ 的和 $P + Q$ 的座標 $x_{P+Q}$、$y_{P+Q}$ 表示為點 $P$ 和 $Q$ 的座標 $x_P$、$y_P$、$x_Q$、$y_Q \in K$ 以及 (21) 的係數 $a_i \in k$ 的有理函數。

**A.1.6. 冪映射.** 由於 $E(K)$ 是阿貝爾群，可以為任何點 $X \in E(K)$ 和任何整數 $n \in \mathbb{Z}$ 定義倍數或冪 $[n]X$。如果 $n = 0$，則 $[0]X = O$；如果 $n > 0$，則 $[n]X = [n-1]X + X$；如果 $n < 0$，則 $[n]X = -[-n]X$。對於 $n \neq 0$，映射 $[n] = [n]_E : E \to E$ 是同源，意味著它是 $E$ 的群律的非常數同態：

$$[n](P + Q) = [n]P + [n]Q \quad \text{對於任何 } P、Q \in E(K) \text{ 和 } n \in \mathbb{Z}。 \tag{26}$$

特別地，$[-1]_E : E \to E$，$P \mapsto -P$，是橢圓曲線 $E$ 的對合自同構。如果 $E$ 是 Weierstrass 形式，$[-1]_E$ 映射 $(x, y) \mapsto (x, -y)$，並且兩個點 $P$、$Q \in E(\mathbb{F}_q)$ 具有相等的 $x$ 座標當且僅當 $Q = \pm P$。

**A.1.7. $E$ 的有理點群的階.** 設 $E$ 為定義在有限基體 $k$ 上的橢圓曲線，並設 $K = \mathbb{F}_q$ 為 $k$ 的有限擴張。那麼 $E(\mathbb{F}_q)$ 是有限阿貝爾群。根據 Hasse 的著名結果，此群的階 $n$ 與 $q$ 的距離不太遠：

$$n = |E(\mathbb{F}_q)| = q - t + 1 \quad \text{其中 } t^2 \leq 4q\text{，即 } |t| \leq 2\sqrt{q}。 \tag{27}$$

我們將主要對 $K = k = \mathbb{F}_p$ 的情況感興趣，其中 $q = p$ 是質數。

**A.1.8. 大質數階的循環子群.** 橢圓曲線密碼學通常使用允許質數階 $\ell$ 的（必然是循環的）子群 $C \subset E(\mathbb{F}_q)$ 的橢圓曲線來執行。等價地，可以給定質數階 $\ell$ 的有理點 $G \in E(\mathbb{F}_q)$；然後 $C$ 可以恢復為由 $G$ 生成的循環子群 $\langle G \rangle$。為了驗證點 $G \in E(\mathbb{F}_q)$ 生成質數階 $\ell$ 的循環群，可以檢查 $G \neq O$，但 $[\ell]G = O$。

根據 Legendre 定理，$\ell$ 必然是有限阿貝爾群 $E(\mathbb{F}_q)$ 的階 $n = |E(\mathbb{F}_q)|$ 的除數：

$$n = |E(\mathbb{F}_q)| = c\ell \quad \text{對於某個整數 } c \geq 1 \tag{28}$$

整數 $c$ 稱為餘因子；通常希望餘因子儘可能小，以使 $\ell = n/c$ 儘可能大。回想一下，根據 (27)，$n$ 始終與 $q$ 具有相同的數量級，因此一旦固定 $q$，透過改變 $E$ 就無法改變太多。

**A.1.9. 橢圓曲線密碼學的資料.** 為了定義特定的橢圓曲線密碼學，必須固定有限基體 $\mathbb{F}_q$（如果 $q = p$ 是質數，則僅需固定質數 $p$）、橢圓曲線 $E/\mathbb{F}_q$（通常由其 Weierstrass 形式 (23) 或 (21) 的係數表示）、基點 $O$（通常是以 Weierstrass 形式寫的橢圓曲線的無窮遠點）以及大質數階 $\ell$ 的循環子群的生成器 $G \in E(\mathbb{F}_q)$（通常由其相對於橢圓曲線方程式的座標 $(x, y)$ 決定）。質數 $\ell$ 和餘因子 $c$ 通常也是橢圓密碼學資料的一部分。

**A.1.10. 橢圓曲線密碼學的主要運算.** 橢圓曲線密碼學通常處理有限體 $\mathbb{F}_q$ 上的橢圓曲線 $E$ 的點群內的大質數階 $\ell$ 的固定循環子群 $C$。通常固定 $C$ 的生成器 $G$。通常假設，給定 $C$ 的一個點 $X$，無法比 $O(\sqrt{\ell})$ 次運算更快地找到其「以 $G$ 為基的離散對數」（即餘數 $n$ 模 $\ell$ 使得 $X = [n]G$）。橢圓曲線密碼學中使用的最重要運算是來自 $C \subset E(\mathbb{F}_q)$ 的點的加法以及它們的冪或倍數的計算。

**A.1.11. 橢圓曲線密碼學的私鑰和公鑰.** 通常，由 **A.1.9** 中列出的資料描述的橢圓曲線密碼學的私鑰是「隨機」整數 $0 < a < \ell$，稱為秘密指數，並且對應的公鑰是點 $A := [a]G$（或僅其 $x$ 座標 $x_A$），經過適當序列化。

**A.1.12. 蒙哥馬利曲線.** 具有特定 Weierstrass 方程式

$$y^2 = x^3 + Ax^2 + x \quad \text{其中 } A = 4a - 2 \text{ 對於某些 } a \in k\text{，} a \neq 0\text{，} a \neq 1 \tag{29}$$

的橢圓曲線稱為蒙哥馬利曲線。它們具有方便的性質，即 $x_{P+Q}x_{P-Q}$ 可以表示為 $x_P$ 和 $x_Q$ 的簡單有理函數：

$$x_{P+Q}x_{P-Q} = \left(\frac{x_P x_Q - A}{x_P - x_Q}\right)^2 \tag{30}$$

這意味著如果已知 $x_{P-Q}$、$x_P$ 和 $x_Q$，則可以計算 $x_{P+Q}$。特別地，如果已知 $x_P$、$x_{[n]P}$ 和 $x_{[n+1]P}$，則可以計算 $x_{[2n]P}$、$x_{[2n+1]P}$ 和 $x_{[2n+2]P}$。使用 $0 < n < 2^s$ 的二進位表示，可以為 $i = 0$、$1$、…、$s$ 計算 $x_{\lfloor\lfloor n/2^{s-i}\rfloor P}$、$x_{\lfloor\lfloor n/2^{s-i}\rfloor+1]P}$，從而獲得 $x_{[n]P}$（這種在蒙哥馬利曲線上從 $x_P$ 開始快速計算 $x_{[n]P}$ 的演算法稱為蒙哥馬利階梯）。因此我們看到蒙哥馬利曲線對於橢圓曲線密碼學的重要性。

## A.2 Curve25519 密碼學

本節描述由 Daniel Bernstein [1] 提出的著名 Curve25519 密碼學及其在 TON 中的使用。

**A.2.1. Curve25519.** *Curve25519* 定義為蒙哥馬利曲線

$$y^2 = x^3 + Ax^2 + x \quad \text{在 } \mathbb{F}_p \text{ 上，其中 } p = 2^{255} - 19 \text{ 且 } A = 486662。 \tag{31}$$

此曲線的階為 $8\ell$，其中 $\ell$ 是質數，$c = 8$ 是餘因子。階為 $\ell$ 的循環子群由具有 $x_G = 9$ 的點 $G$ 生成（這決定了 $G$ 直到 $y_G$ 的符號，這並不重要）。Curve25519 的二次扭曲 $2y^2 = x^3 + Ax^2 + x$ 的階對於另一個質數 $\ell'$ 為 $4\ell'$。[42]

**A.2.2. Curve25519 的參數.** Curve25519 的參數如下：

- 基體：質數有限體 $\mathbb{F}_p$，其中 $p = 2^{255} - 19$。
- 方程式：$y^2 = x^3 + Ax^2 + x$，其中 $A = 486662$。
- 基點 $G$：特徵為 $x_G = 9$（九是 $E(\mathbb{F}_p)$ 的大質數階子群的生成器的最小正整數 $x$ 座標）。
- $E(\mathbb{F}_p)$ 的階：

$$|E(\mathbb{F}_p)| = p - t + 1 = 8\ell, \quad \text{其中} \tag{32}$$

$$\ell = 2^{252} + 27742317777372353535851937790883648493 \quad \text{是質數。} \tag{33}$$

[42] 實際上，D. Bernstein 選擇 $A = 486662$ 是因為它是最小的正整數 $A \equiv 2 \pmod 4$，使得對於 $p = 2^{255} - 19$，$\mathbb{F}_p$ 上的對應蒙哥馬利曲線 (31) 和該曲線的二次扭曲都具有小餘因子。這種安排避免了檢查點 $P$ 的 $x$ 座標 $x_P \in \mathbb{F}_p$ 是否定義了位於蒙哥馬利曲線本身還是其二次扭曲上的點 $(x_P, y_P) \in \mathbb{F}_p^2$ 的必要性。

- $\tilde{E}(\mathbb{F}_p)$ 的階，其中 $\tilde{E}$ 是 $E$ 的二次扭曲：

$$|\tilde{E}(\mathbb{F}_p)| = p + t + 1 = 2p + 2 - 8\ell = 4\ell', \quad \text{其中} \tag{34}$$

$$\ell' = 2^{253} - 55484635554744707071703875581767296995 \quad \text{是質數。} \tag{35}$$

**A.2.3. 標準 Curve25519 密碼學的私鑰和公鑰.** Curve25519 密碼學的私鑰通常定義為秘密指數 $a$，而對應的公鑰是 $x_A$，即點 $A := [a]G$ 的 $x$ 座標。這通常足以執行 ECDH（橢圓曲線 Diffie–Hellman 金鑰交換）和非對稱橢圓曲線密碼學，如下所示：

如果一方想要向另一方發送訊息 $M$，該方具有公鑰 $x_A$（和私鑰 $a$），則執行以下計算。產生一次性隨機秘密指數 $b$，並使用蒙哥馬利階梯計算 $x_B := x_{[b]G}$ 和 $x_{[b]A}$。之後，使用 256 位元「共享秘密」$S := x_{[b]A}$ 作為金鑰，透過諸如 AES 的對稱密碼加密訊息 $M$，並將 256 位元整數（「一次性公鑰」）$x_B$ 前置到加密訊息。一旦具有公鑰 $x_A$ 的一方接收到此訊息，它可以從 $x_B$（與加密訊息一起傳輸）和私鑰 $a$ 開始計算 $x_{[a]B}$。由於 $x_{[a]B} = x_{[ab]G} = x_{[b]A} = S$，接收方恢復共享秘密 $S$ 並能夠解密訊息的其餘部分。

**A.2.4. TON Curve25519 密碼學的公鑰和私鑰.** TON 使用另一種形式的 Curve25519 密碼學的公鑰和私鑰，借用自 Ed25519 密碼學。

TON Curve25519 密碼學的私鑰只是一個隨機的 256 位元字串 $k$。透過計算 $\text{SHA512}(k)$，取結果的前 256 位元，將它們解釋為小端序 256 位元整數 $a$，清除 $a$ 的位元 0、1、2 和 255，並設定位元 254 以獲得值 $2^{254} \le a < 2^{255}$，可被八整除，來使用它。如此獲得的值 $a$ 是對應於 $k$ 的秘密指數；同時，$\text{SHA512}(k)$ 的剩餘 256 位元構成秘密鹽值 $k''$。

對應於 $k$——或秘密指數 $a$——的公鑰只是點 $A := [a]G$ 的 $x$ 座標 $x_A$。一旦計算出 $a$ 和 $x_A$，它們的使用方式與 **A.2.3** 中完全相同。特別地，如果需要序列化 $x_A$，它將作為無符號小端序 256 位元整數序列化為 32 個八位元組。

**A.2.5. Curve25519 用於 TON 網路.** 請注意， **A.2.4** 中描述的非對稱 Curve25519 密碼學被 TON 網路廣泛使用，特別是 ADNL（抽象資料包網路層）協議。然而，TON 區塊鏈主要需要橢圓曲線密碼學用於簽章。為此，使用下一節中描述的 Ed25519 簽章。

## A.3 Ed25519 密碼學

Ed25519 密碼學被 TON 區塊鏈和 TON 網路廣泛用於快速密碼學簽章。本節描述 TON 使用的 Ed25519 密碼學變體。與標準方法（如 D. Bernstein

等人在 [2] 中定義的）的一個重要區別是，TON 提供私鑰和公鑰 Ed25519 金鑰自動轉換為 Curve25519 金鑰的功能，以便相同的金鑰可用於加密/解密和簽署訊息。

**A.3.1. 扭曲 Edwards 曲線.** 體 $k$ 上的具有參數 $a \neq 0$ 和 $d \neq 0, a$ 的扭曲 *Edwards* 曲線 $E_{a,d}$ 由方程式給出

$$E_{a,d} : ax^2 + y^2 = 1 + dx^2y^2 \quad \text{在 } k \text{ 上} \tag{36}$$

如果 $a = 1$，此方程式定義（未扭曲的）Edwards 曲線。點 $O(0, 1)$ 通常被選為 $E_{a,d}$ 的標記點。

**A.3.2. 扭曲 Edwards 曲線雙有理等價於蒙哥馬利曲線.** 扭曲 Edwards 曲線 $E_{a,d}$ 雙有理等價於蒙哥馬利橢圓曲線

$$M_A : v^2 = u^3 + Au^2 + u \tag{37}$$

其中 $A = 2(a+d)/(a-d)$ 且 $d/a = (A-2)/(A+2)$。雙有理等價 $\phi : E_{a,d} \dashrightarrow M_A$ 及其逆 $\phi^{-1}$ 由

$$\phi : (x, y) \mapsto \left(\frac{1+y}{1-y}, \frac{c(1+y)}{x(1-y)}\right) \tag{38}$$

和

$$\phi^{-1} : (u, v) \mapsto \left(\frac{cu}{v}, \frac{u-1}{u+1}\right) \tag{39}$$

給出，其中

$$c = \sqrt{\frac{A+2}{a}} \tag{40}$$

請注意，$\phi$ 將 $E_{a,d}$ 的標記點 $O(0, 1)$ 轉換為 $M_A$ 的標記點（即其無窮遠點）。

**A.3.3. 扭曲 Edwards 曲線上點的加法.** 由於 $E_{a,d}$ 雙有理等價於橢圓曲線 $M_A$，$M_A$ 上點的加法可以透過設定

$$P + Q := \phi^{-1}\big(\phi(P) + \phi(Q)\big) \quad \text{對於任何 } P、Q \in E_{a,d}(k)。 \tag{41}$$

轉移到 $E_{a,d}$。請注意，標記點 $O(0, 1)$ 是關於此加法的中性元素，並且 $-(x_P, y_P) = (-x_P, y_P)$。

**A.3.4. 在扭曲 Edwards 曲線上加點的公式.** 座標 $x_{P+Q}$ 和 $y_{P+Q}$ 允許作為 $x_P$、$y_P$、$x_Q$、$y_Q$ 的有理函數的簡單表達式：

$$x_{P+Q} = \frac{x_P y_Q + x_Q y_P}{1 + d x_P x_Q y_P y_Q} \tag{42}$$

$$y_{P+Q} = \frac{y_P y_Q - a x_P x_Q}{1 - d x_P x_Q y_P y_Q} \tag{43}$$

這些表達式可以有效地計算，特別是如果 $a = -1$。這是扭曲 Edwards 曲線對於快速橢圓曲線密碼學重要的原因。

**A.3.5. Ed25519 扭曲 Edwards 曲線.** Ed25519 是 $\mathbb{F}_p$ 上的扭曲 Edwards 曲線 $E_{-1,d}$，其中 $p = 2^{255} - 19$ 是與 Curve25519 使用的相同質數，並且 $d = -(A-2)/(A+2) = -121665/121666$，其中 $A = 486662$ 與方程式 (31) 中的相同：

$$-x^2 + y^2 = 1 - \frac{121665}{121666} x^2 y^2 \quad \text{對於 } x、y \in \mathbb{F}_p，p = 2^{255} - 19。 \tag{44}$$

這樣，Ed25519 曲線 $E_{-1,d}$ 雙有理等價於 Curve25519 (31)，並且可以使用 $E_{-1,d}$ 和公式 (42)–(43) 在 Ed25519 或 Curve25519 上進行點加法，使用 (38) 和 (39) 將 Ed25519 上的點轉換為 Curve25519 上的對應點，反之亦然。

**A.3.6. Ed25519 的生成器.** Ed25519 的生成器是具有 $y(G') = 4/5$ 且 $0 \le x(G') < p$ 為偶數的點 $G'$。根據 (38)，它對應於 Curve25519 的點 $(u, v)$，其中 $u = (1 + 4/5)/(1 - 4/5) = 9$（即 **A.2.2** 中選擇的 Curve25519 的生成器 $G$）。特別地，$G = \phi(G')$，$G'$ 生成相同大質數階 $\ell$ 的循環子群，該階在 (32) 中給出，並且對於任何整數 $a$，

$$\phi([a]G') = [a]G \quad . \tag{45}$$

這樣，我們可以使用 Curve25519 及其生成器 $G$，或使用 Ed25519 和生成器 $G'$ 執行計算，並獲得本質上相同的結果。

**A.3.7. Ed25519 上點的標準表示.** Ed25519 上的點 $P(x, y)$ 可以由其兩個座標 $x_P$ 和 $y_P$ 表示，即模 $p = 2^{255} - 19$ 的餘數。反過來，這兩個座標都可以由無符號 255 位元或 256 位元整數 $0 \le x_P, y_P < p < 2^{255}$ 表示。

然而，更常用的是（TON 也使用）透過一個小端序無符號 256 位元整數 $\tilde{P}$ 來表示 $P$ 的更緊湊表示。即，$\tilde{P}$ 的 255 個低位元包含 $y_P$，$0 \le y_P < p < 2^{255}$，並且位元 255 用於儲存 $x_P \bmod 2$，即 $x_P$ 的低位元。由於 $y_P$ 始

終將 $x_P$ 決定到符號（即直到將 $x_P$ 替換為 $p - x_P$），$x_P$ 和 $p - x_P$ 始終可以透過它們的低位元區分，$p$ 為奇數。

如果知道 $\pm P$ 直到符號就足夠了，則可以忽略 $x_P \bmod 2$ 並僅考慮小端序 255 位元整數 $y_P$，任意設定位元 255，忽略其先前定義的值，或將其清除。

**A.3.8. Ed25519 的私鑰.** Ed25519 的私鑰只是任意 256 位元字串 $k$。秘密指數 $a$ 和秘密鹽值 $k''$ 從 $k$ 衍生，首先計算 $\text{SHA512}(k)$，然後將此 SHA512 的前 256 位元作為 $a$ 的小端序表示（但清除位元 255、2、1 和 0，並設定位元 254）；然後 $\text{SHA512}(k)$ 的最後 256 位元構成 $k''$。

這本質上與 **A.2.4** 中描述的程序相同，但用雙有理等價曲線 Ed25519 替換 Curve25519。（實際上，反過來：此程序對於基於 Ed25519 的橢圓曲線密碼學是標準的，TON 將該程序擴展到 Curve25519。）

**A.3.9. Ed25519 的公鑰.** 對應於 Ed25519 的私鑰 $k$ 的公鑰是點 $A = [a]G'$ 的標準表示（參見 **A.3.7**），其中 $a$ 是由私鑰 $k$ 定義的秘密指數（參見 **A.3.8**）。

請注意，$\phi(A)$ 是根據 **A.2.4** 和 (45) 由相同私鑰 $k$ 定義的 Curve25519 的公鑰。這樣，我們可以將 Ed25519 的公鑰轉換為 Curve25519 的對應公鑰，反之亦然。私鑰根本不需要轉換。

**A.3.10. 密碼學 Ed25519 簽章.** 如果訊息（八位元組字串）$M$ 需要由定義秘密指數 $a$ 和秘密鹽值 $k''$ 的私鑰 $k$ 簽署，則執行以下計算：

- $r := \text{SHA512}(k''|M)$，解釋為小端序 512 位元整數。此處 $s|t$ 表示八位元組字串 $s$ 和 $t$ 的串接。
- $R := [r]G'$ 是 Ed25519 上的一個點。
- $\tilde{R}$ 是點 $R$ 的標準表示（參見 **A.3.7**）作為 32 個八位元組字串。
- $s := r + a \cdot \text{SHA512}(\tilde{R}|\tilde{A}|M) \bmod \ell$，編碼為小端序 256 位元整數。此處 $\tilde{A}$ 是點 $A = [a]G'$ 的標準表示，即對應於 $k$ 的公鑰。

（Schnorr）簽章是 64 個八位元組字串 $(\tilde{R}, s)$，由點 $R$ 的標準表示和 256 位元整數 $s$ 組成。

**A.3.11. 檢查 Ed25519 簽章.** 為了驗證訊息 $M$ 的簽章 $(\tilde{R}, s)$，據稱由對應於已知公鑰 $A$ 的私鑰 $k$ 的所有者製作，執行以下步驟：

- 計算 Ed25519 的點 $[s]G'$ 和 $R + [\text{SHA512}(\tilde{R}|\tilde{A}|M)]A$。
- 如果這兩個點重合，則簽章是正確的。