;; # FunC 標準工具庫

;; 繁體中文註解版 by Y.C.

{-

這不是單純的中文註解翻譯版本，除了可讀性，此文檔更着重可理解性，把相同類型的函數分類解釋，
希望令 TON 智能合約工程師在不查閱 官方文檔 / 只閱讀此文件的情況下亦能完全理解 FunC 的使用方法，
所有中文均經過人腦編譯，不存在 AI 製造的奇怪語法，並可直接複製貼上取代原有的 stdlib.fc 服用。
此文檔同時參照了官方 USDT 及 NOT 代幣當中所使用的最新版 stdlib.fc，
部份函式解說在 官方文檔 或 blueprint 預載的 FunC 標準工具庫當中未見更新，
因此在這份檔案當中補充了對相應函數用法的理解，以配合 TON 的狀態更新以及 TVM 的最新升級。

-}

{-

這文件是 TON FunC 標準庫的一部分。

FunC 標準庫是免費軟體：您可以重新分發和/或修改
它遵循”自由軟體基金會”所發布的 GNU Lesser General Public License 條款，
無論是版本 2 或（您所選擇的）任何更新的版本均被許可使用。
FunC 標準庫發布是希望被受用，但並不提供任何擔保；
甚至沒有附帶默示保證，適銷性或特定用途的適用性。
請參閱 GNU Lesser General Public License 了解更多詳細資訊。

元組操作原語

名稱和類型大多是不言自明的，請參閱 "forall" 的多態性 研究有關多態函數的詳細資訊。
forall 代表了函數宣告或定義之前，可以隨着情況選擇任何類型的變量。
注意原類型 “tuple” 的值無法轉換為複合元組類型（例如 (int, cell)）反之亦然。
意思是 tuple 自支援單一類型的多項數值序列。

Lisp 風格的列表

列表可理解為嵌套的 2 維元組。
空列表通常表示為 TVM `null` 值（可以透過呼叫 [null()]取得）。
例如，元組 (1,(2,(3, null)))象徵列表 [1,2,3]。列表當中的元素可以是不同的類型。

-}

{-
    將一個元素加入 Lisp 樣式列表的開頭

        forall X -> :
            表示函數當中的 X 可以是任何類型的值，例如 int, slice, cell 等。
        tuple :
            表示回傳的結果是一個 tuple，類似於 (1, (2, (3, null)))。
        cons(X head, tuple tail) :
            表示這個 function 稱為 "cons" 發動需要輸入兩個參數，包括任何類型的 X 作為頭部以及 tuple 類型的作為尾部。
        asm "CONS":
            引用 fift 的彙編指令/字典詞操作邏輯。

    可以理解 tuple(1, (2, (3, null))) 是 list（列表）的在 stack machine 的底層具體結構
    更多 fift 底層語言資料可參閱 zh_fiftbase_dark_繁中註解導讀版
-}
    forall X -> tuple cons(X head, tuple tail) asm "CONS";

;;;   分拆並提取 Lisp 格式列表的頭尾兩部份，uncons 先回傳頭部再回傳尾部的 tuple，list_next 先回傳尾部 tuple 後才回傳頭部。
    forall X -> (X, tuple) uncons(tuple list) asm "UNCONS";
    forall X -> (tuple, X) list_next(tuple list) asm(-> 1 0) "UNCONS";

```
;;;     大致功能如 cons，但 car 只是回傳頭部的值，cdr 只是回傳尾部的 tuple。
        forall X -> X car(tuple list) asm "CAR";
        tuple cdr(tuple list) asm "CDR";


;;;     創造一個沒有任何元素的 tuple，通常用作新增事項並在之後擴展。
        tuple empty_tuple() asm "NIL";


;;;     將一個元素放置於 tuple 的末端，但需確保添加元素後的 tuple 不能超過 255 個元素，否則會導致類型檢查異常。
;;      ~tpush 返回了一個額外的空元組 () 作為附加返回值。這可能用於一些需要額外返回值的場景
        forall X -> tuple tpush(tuple t, X value) asm "TPUSH";
        forall X -> (tuple, ()) ~tpush(tuple t, X value) asm "TPUSH";


;;;     傳入相關數量的值，創造帶有 1 / 2 / 3 / 4 個元素 tuple。
        forall X -> [X] single(X x) asm "SINGLE";
        forall X, Y -> [X, Y] pair(X x, Y y) asm "PAIR";
        forall X, Y, Z -> [X, Y, Z] triple(X x, Y y, Z z) asm "TRIPLE";
        forall X, Y, Z, W -> [X, Y, Z, W] tuple4(X x, Y y, Z z, W w) asm "4 TUPLE";


;;;     把帶有 1 / 2 / 3 / 4 個元素 tuple 拆解回元素類別。
        forall X -> X unsingle([X] t) asm "UNSINGLE";
        forall X, Y -> (X, Y) unpair([X, Y] t) asm "UNPAIR";
        forall X, Y, Z -> (X, Y, Z) untriple([X, Y, Z] t) asm "UNTRIPLE";
        forall X, Y, Z, W -> (X, Y, Z, W) untuple4([X, Y, Z, W] t) asm "4 UNTUPLE";


;;;     回傳 tuple 第 1 / 2 / 3 / 4 個元素。
        forall X -> X first(tuple t) asm "FIRST";
        forall X -> X second(tuple t) asm "SECOND";
        forall X -> X third(tuple t) asm "THIRD";
        forall X -> X fourth(tuple t) asm "3 INDEX";


;;;     回傳 2 位元素 tuple 中的第 1 / 2 個元素，回傳 3 位元素 tuple 中的第 1 / 2 / 3 個元素。
        forall X, Y -> X pair_first([X, Y] p) asm "FIRST";
        forall X, Y -> Y pair_second([X, Y] p) asm "SECOND";
        forall X, Y, Z -> X triple_first([X, Y, Z] p) asm "FIRST";
        forall X, Y, Z -> Y triple_second([X, Y, Z] p) asm "SECOND";
        forall X, Y, Z -> Z triple_third([X, Y, Z] p) asm "THIRD";


;;;     建立一個沒有值的元素，在 TVM 中，Null 類型表示原類型值的缺失，所以可理解為暫時不指定特定類型。
        forall X -> X null() asm "PUSHNULL";


;;;     當知道變量 x 已確認處於 stack 的某一個位置時，將 x 推到 stack 的頂部。
        forall X -> (X, ()) ~impure_touch(X x) impure asm "NOP";


;;;     回傳目前 Unix 整數時間，可理解為從 c7 寄存器 當中取得 Unix timestamp。
        int now() asm "NOW";


;;;     以 Slice 格式回傳當前智能合約地址。
        slice my_address() asm "MYADDR";


;;;     回傳目前智能合約的餘額，包括 TON 代幣以及其他 Jetton 代幣的餘額。
;;      當中的 cell 在理論上可以表示多達 2^32（約 43 億）種不同的 Jetton 代幣。
        [int, cell] get_balance() asm "BALANCE";
```

```
;;;     回傳當前交易的邏輯時間 / 回傳當前區塊的起始邏輯時間。
        int cur_lt() asm "LTIME";
        int block_lt() asm "BLOCKLT";


;;;     回傳一個 256 位無符號整數的 representation hash，用作簽署和檢查用途。
;;      cell_hash 及 slice_hash 分別存入 cell 及 slice 類別進行操作
;;      string_hash 的位元長度必須是 8 的倍數，否則將會拋出 cell 下溢異常
        int cell_hash(cell c) asm "HASHCU";
        int slice_hash(slice s) asm "HASHSU";
        int string_hash(slice s) asm "SHA256U";


;;;     找出兩個整數的最小值 (min) 和最大值 (max)，將兩個整數排序 (minmax)，以及計算整數的絕對值 (abs)。
        int min(int x, int y) asm "MIN";
        int max(int x, int y) asm "MAX";
        (int, int) minmax(int x, int y) asm "MINMAX";
        int abs(int x) asm "ABS";


;;;     顯示堆疊中最多 255 個值（從堆疊頂部開始），並顯示堆疊的總深度，用於 local 檢查當前堆疊狀態和追踪問題。（TVM 中愈多程序，深度值愈大）
        () dump_stack() impure asm "DUMPSTK";


;;;     強制保存寄存器c4（“持久性資料”）和c5（“動作”）的當前狀態，以便即使拋出異常，執行也會“成功”地執行。
        () commit() impure asm "COMMIT";


;;;     判斷某個 cell 是否為空值（即 null）。
        int cell_null?(cell c) asm "ISNULL";


;;;     my_code 將檢索並回傳當前存儲在智能合約的 c7 寄存器中的智能合約代碼。
        cell my_code() asm "MYCODE";


;;;     set_code 將會把智能合約代碼更改為 new_code 提供的新代碼，更改操作只在智能合約的執行成功完成後才生效。如合約執行過程中出現錯誤或失敗，代碼將不會被更改。
        () set_code(cell new_code) impure asm "SETCODE";


;;;     常量 / 布爾值 / 特殊地址 / 操作碼（op）/ 查詢 ID（query_id）在 stdlib.fc 的定義及基本操作。
        const int TRUE = -1;
        const int FALSE = 0;
        const int MASTERCHAIN = -1;
        const int BASECHAIN = 0;
        const int MSG_OP_SIZE = 32;
        const int MSG_QUERY_ID_SIZE = 64;
        (slice, ()) ~skip_maybe_ref(slice s) asm "SKIPOPTREF";
        (slice, int) ~load_bool(slice s) inline { return s.load_int(1); }
        builder store_bool(builder b, int value) inline { return b.store_int(value, 1); }
        builder store_address_none(builder b) inline { return b.store_uint(0, 2);}
        slice address_none() asm "<b 0 2 u, b> <s PUSHSLICE";
        int is_address_none(slice s) inline { return s.preload_uint(2) == 0; }
        ;; addr_none$00 = MsgAddressExt;


{-
        用於 Ed25519 的簽名驗證 check_signature 是針對 hash 進行驗證，
        而 check_data_signature 則直接對 slice data 進行驗證，
        回傳 -1 表示有效，或 0 表示無效。
        若 check_data_signature 當中的 data 位長不是 8 的倍數，則拋出下溢異常。
-}
               int check_signature(int hash, slice signature, int public_key) asm "CHKSIGNU";
               int check_data_signature(slice data, slice signature, int public_key) asm "CHKSIGNS";
```

```
{-
    用於確認智能合約或數據結構中實際使用的存儲容量，用作計算存儲費用，
    它們使用了深度優先搜索 DAG 單元格結構，並使用 hash table 防止重複訪問同一個 cell。
    compute_data_size 及 slice_compute_data_size 將傳回（x, y, z），
    分別代表( cell的總量，佔用數據位元，cell的分支量 )，
    而帶有問號的 compute_data_size? 及 slice_compute_data_size? 稱為非靜默版本，
    將傳回（x, y, z, e）在遇到錯誤時會立即拋出異常及返回特定的錯誤識別碼 e 。
-}
        (int, int, int) compute_data_size(cell c, int max_cells) impure asm "CDATASIZE";
        (int, int, int) slice_compute_data_size(slice s, int max_cells) impure asm "SDATASIZE";
        (int, int, int, int) compute_data_size?(cell c, int max_cells)
        asm "CDATASIZEQ NULLSWAPIFNOT2 NULLSWAPIFNOT";
        (int, int, int, int) slice_compute_data_size?(cell c, int max_cells)
        asm "SDATASIZEQ NULLSWAPIFNOT2 NULLSWAPIFNOT";



{-
    get_data 及 get_c3 分別以 cell / cont 形式回傳智能合約的 persistent storage 及 continuation
    get_data 得到的 cell 可通過切片（slice）和構建器（builder）進行解析或修改
    get_c3 得到的 cont 可用於記錄或管理協程，或後期重新執行部分邏輯，甚至回到某個狀態
    set_data 及 set_c3 分別存入 cell / cont 作為參數更改合約暫存器更新合約
    set_data 將指定的 cell 設置為持久的合約數據，更新合約的持久存儲
    set_c3 將更新 c3，合約可以在運行時動態修改其行為，進而處理複雜的邏輯控制，實現動態和靈活的程式行為
    執行 set_c3 後，當前 stack 不會立即改變，但後續任何的函數調用都將會更新
    bless 將一個 slice 轉換為成 ordinary continuation，包含一個空 stack 以及 savelist
-}
        cell get_data() asm "c4 PUSH";
        cont get_c3() impure asm "c3 PUSH";
        () set_data(cell c) impure asm "c4 POP";
        () set_c3(cont c) impure asm "c3 POP";
        cont bless(slice s) impure asm "BLESS";



{-
    accept_message 將當前的 Gas 限制設置為其最大允許值 gm，並將 Gas 信用 gc 設為零。同時，會將 gc 的值從 gr 中減去。意味智能合約同意收購 Gas 來完成當前的交易，因為外部消息不會攜帶任何 Gas，因此需要通過這個函數來購買額外的 Gas。
    set_gas_limit 將 limit 設為 gl 和 gm，並將 gas 信用 gc 設為零。如果消耗的 gas（包括目前指令）超過 gl，則會拋出 gas 不足異常。 limit ≥ 2^63 - 1 的 set_gas_limit 將等同 accept_message。
    若 check_data_signature 當中的 data 位長不是 8 的倍數，則拋出下溢異常。詳情請參考接受消息操作。
-}
        () accept_message() impure asm "ACCEPT";
        () set_gas_limit(int limit) impure asm "SETGASLIMIT";



{-
    begin_parse 將 cell 轉換為 slice / end_parse 檢查 slice 是否為空，如果不為空則拋出異常。
    begin_cell 創建一個新 builder 構建和組裝資料並存儲數據；end_cell 結束 builder 構建，並封裝成 cell。
    回傳 cell 的最長深度值，沒有引用則深度為 0， A 引用了 B，而 B 又引用了 C，深度即為 2。
-}
        slice begin_parse(cell c) asm "CTOS";
        () end_parse(slice s) impure asm "ENDS";
        builder begin_cell() asm "NEWC";
        cell end_cell(builder b) asm "ENDC";
        int cell_depth(cell c) asm "CDEPTH";



{-
    load_grams 從 slice 中加載 TonCoins 數值及剩餘的 slice。 load_coins 用於加載更長的無符號整數。
    store_grams 將 TonCoins 的金額存儲到 builder。 store_coins 用於存儲更長的無符號整數。
    需要確保金額在 0 到 2^128 - 1 之間的整數範圍內，超出範圍則會產生報錯。
-}
        (slice, int) load_grams(slice s) asm(-> 1 0) "LDGRAMS";
        (slice, int) load_coins(slice s) asm(-> 1 0) "LDVARUINT16";
        builder store_grams(builder b, int x) asm "STGRAMS";
        builder store_coins(builder b, int x) asm "STVARUINT16";



{-
    load_ref 回傳加載的頭一個 cell 引用和經過修改後的剩餘 slice，
    preload_ref 僅回傳加載的 cell 引用， slice 本身不受影響。
```

```
    load_maybe_ref 回傳剩餘的 slice 和可能存在的 cell 引用，而 preload_maybe_ref 只回傳這個引用；
    如果引用不存在，則返回 null。
-}
        (slice, cell) load_ref(slice s) asm(-> 1 0) "LDREF";
        cell preload_ref(slice s) asm "PLDREF";
        (slice, cell) load_maybe_ref(slice s) asm(-> 1 0) "LDOPTREF";
        cell preload_maybe_ref(slice s) asm "PLDOPTREF";



{-
    first_bits 回傳頭 "len" 個位元的資料； slice_last 回傳由尾開始數起的 "len" 個位元嘅資料；
    skip_bits 及 ~skip_bits 無視 slice 頭部的 "len" 個位元，只回傳尾部的資料；
    skip_last_bits 及 ~skip_last_bits 無視 slice 第 "len" 個位元後的資料，只回傳頭部的資料；
    其中 0 ≤ len ≤ 1023。
-}
        slice first_bits(slice s, int len) asm "SDCUTFIRST";
        slice slice_last(slice s, int len) asm "SDCUTLAST";
        slice skip_bits(slice s, int len) asm "SDSKIPFIRST";
        (slice, ()) ~skip_bits(slice s, int len) asm "SDSKIPFIRST";
        slice skip_last_bits(slice s, int len) asm "SDSKIPLAST";
        (slice, ()) ~skip_last_bits(slice s, int len) asm "SDSKIPLAST";



{-
    load_dict 回傳剩餘的 slice 及加載的字典（cell 類型），沒有字典會返回 null；
    preload_dict 回傳字典，但不改變 slice 狀態；
    skip_dict 及 ~skip_dict 回傳沒有字典數據的 slice。
-}
        (slice, cell) load_dict(slice s) asm(-> 1 0) "LDDICT";
        cell preload_dict(slice s) asm "PLDDICT";
        slice skip_dict(slice s) asm "SKIPDICT";
        (slice, ()) ~skip_dict(slice s) asm "SKIPDICT";



{-
    slice_refs 回傳 slice 中對其他 cell 的引用數量； slice_bits_refs 回傳位元數據量和 cell 引用數量；
    slice_data_empty? 檢查 slice 是否沒有任何數據位元（但可包含有引用）；
    slice_refs_empty? 檢查 slice 是否沒有任何引用（但可能包含數據位元）；
    slice_empty? 檢查 slice 是否完全沒有任何可用數據或引用； slice_depth 回傳 slice 的最長深度值；
    equal_slices_bits 檢查兩個 slice 的位元數據部分是否完全一致，一致返回 1， 否則返回 0。
-}
        int slice_refs(slice s) asm "SREFS";
        (int, int) slice_bits_refs(slice s) asm "SBITREFS";
        int slice_empty?(slice s) asm "SEMPTY";
        int slice_data_empty?(slice s) asm "SDEMPTY";
        int slice_refs_empty?(slice s) asm "SREMPTY";
        int slice_depth(slice s) asm "SDEPTH";
        int equal_slices_bits(slice a, slice b) asm "SDEQ";



{-
    builder_refs 回傳 builder 中對其他 cell 的引用數量； builder_bits 回傳位元數據總量； builder_depth 回傳最長深度值；
    builder_null? 檢查 builder 是否為空，空或 null 回傳 1，有數值則回傳 0。
    store_ref 在 builder 尾部添加 cell 引用，並回傳更新後的 builder；
    store_slice 把 slice 數據添加到 builder 的尾部；
    store_dict 將字典結構序列化並存入 builder，如果 cell 為 null，則存入位元 0，否則存入位元 1 及 cell 引用。
    store_maybe_ref 將 cell 引用存入 builder，如果 cell 為 null，則存入位元 0，否則存入位元 1 及 cell 引用。
    store_builder 將兩個 builder 進行串聯，將 from 中的數據添加到 to 的末尾，用於將多段數據合併成一個更大的數據結構。
-}
        int builder_refs(builder b) asm "BREFS";
        int builder_bits(builder b) asm "BBITS";
        int builder_depth(builder b) asm "BDEPTH";
        int builder_null?(builder b) asm "ISNULL";
        builder store_ref(builder b, cell c) asm(c b) "STREF";
        builder store_slice(builder b, slice s) asm "STSLICER";
        builder store_dict(builder b, cell c) asm(c b) "STDICT";
        builder store_maybe_ref(builder b, cell c) asm(c b) "STOPTREF";
        builder store_builder(builder to, builder from) asm "STBR";
        ;; builder store_uint(builder b, int x, int len) asm(x b len) "STUX";
        ;; builder store_int(builder b, int x, int len) asm(x b len) "STIX";



{-
```

隨機數生成的核心工具，允許開發者靈活地生成隨機數，控制隨機數的序列，並混合新的隨機性。
random 成生一個 256 位元無符號偽隨機整數，而 rand 則是基於 random 產生一個在指定範圍內的偽隨機整數；
get_seed 用於獲取當前隨機數生成器當前的內部狀態接返回當前的隨機種子 r，set_seed 則用於設置隨機種子；
randomize 混合一個整數製作隨機種子，變更隨機數生成的起始狀態。 而 randomize_lt 則使用當前的邏輯時間戳記作為生成的起始。
-}

```
int random() impure asm "RANDU256";
int rand(int range) impure asm "RAND";
int get_seed() impure asm "RANDSEED";
() set_seed(int x) impure asm "SETRAND";
() randomize(int x) impure asm "ADDRAND";
() randomize_lt() impure asm "LTIME" "ADDRAND";
```

{-
用於加載和預加載特定位長的整數或位元的 slice 數據結構，load 類函數會截斷 slice，而 preload 僅進行讀取而不改變 slice 的狀態；
這些函數的實現已在編譯器層面進行優化，因而被註釋掉。
-}

```
;; (slice, int) ~load_int(slice s, int len) asm(s len -> 1 0) "LDIX";
;; (slice, int) ~load_uint(slice s, int len) asm( -> 1 0) "LDUX";
;; int preload_int(slice s, int len) asm "PLDIX";
;; int preload_uint(slice s, int len) asm "PLDUX";
;; (slice, slice) load_bits(slice s, int len) asm(s len -> 1 0) "LDSLICEX";
;; slice preload_bits(slice s, int len) asm "PLDSLICEX";
```

{-

TL-B 地址結構說明：

每一種地址類型在 TL-B 反序列化後均會轉換為一個元組，元組的結構對應著地址的不同部分，
包括 anycast 信息（部分含有）、工作鏈 ID 與實際的地址數據，準確表示不同類型的地址信息。

```
anycast_info$_ depth:(#<= 30) { depth >= 1 } rewrite_pfx:(bits depth) = Anycast;
```

depth 表示 1-30 位深度，rewrite_pfx 是匹配目標地址的字段。

```
addr_none$00 = MsgAddressExt;
```

addr_none：表示「無地址」情況，在反序列化後，它被表示為一個只包含一個整數 0 的 tuple。

```
addr_extern$01 len:(## 8) external_address:(bits len)  = MsgAddressExt;
```

addr_extern：在反序列化後，結構表示為 t = (1 , s)，其中 s 是 external_address 的位元組。

```
addr_std$10 anycast:(Maybe Anycast) workchain_id:int8 address:bits256 = MsgAddressInt;
```

addr_std：反序列化後表示為 t = (2, u, x, s)，其中 u 是可能存在的 anycast 的信息，x 是 workchain_id，s 是實際地址。

```
addr_var$11 anycast:(Maybe Anycast) addr_len:(## 9) workchain_id:int32 address:(bits addr_len) = MsgAddressInt;
```

addr_var：是可變長度的內部地址格式，被表示為 t = (3, u, x, s)，含義大致上與 addr_std 相同。

```
_ _:MsgAddressInt = MsgAddress; _ _:MsgAddressExt = MsgAddress;
```

MsgAddressInt & MsgAddressExt 同樣是 MsgAddress 的子類型。

```
int_msg_info$0 ihr_disabled:Bool bounce:Bool bounced:Bool src:MsgAddress dest:MsgAddressInt
   value:CurrencyCollection ihr_fee:Grams fwd_fee:Grams
   created_lt:uint64 created_at:uint32 = CommonMsgInfoRelaxed;
```

int_msg_info 描述內部消息的結構，涵蓋了消息在區塊鏈內部傳遞過程中的詳細信息。

```
ext_out_msg_info$11 src:MsgAddress dest:MsgAddressExt
   created_lt:uint64 created_at:uint32 = CommonMsgInfoRelaxed;
```

ext_out_msg_info 描述從區塊鏈外部消息的結構，主要包含了來源地址、目標地址以及創建時間等信息。
-}

{-
load_msg_addr 回傳提取的 MsgAddress 及剩餘的未解析部分；
parse_addr 回傳 MsgAddress 及反序列化後的元組結構 ： (0) / (1,s) / (2, u, x, s) / (3, u, x, s) - 視乎地址類型；
parse_std_addr 回傳工作鏈 ID 及 256 位的地址數據； parse_var_addr 回傳值工作鏈 ID 及實際的地址切片。
-}

```
(slice, slice) load_msg_addr(slice s) asm(-> 1 0) "LDMSGADDR";
tuple parse_addr(slice s) asm "PARSEMSGADDR";
(int, int) parse_std_addr(slice s) asm "REWRITESTDADDR";
(int, slice) parse_var_addr(slice s) asm "REWRITEVARADDR";
```

{-

Dictionary 在 FunC 中的概念

字典（Dictionary）與 mapping 的觀念類似，可理解為於二維數據庫的資料結構，可存儲多組鍵（key）和值（value）的對應關係。
每個鍵都與特定值相聯，而鍵和值可以是不同的數據類型。字典用於快速查找、插入、更新和刪除數據，根據鍵訪問相應的值。
鍵（Key）：在 FunC 中，鍵可以是有符號或無符號的整數。鍵的位長（key_len）指定了鍵的長度，例如 16 位、32 位等。
值（Value）：字典中的值是不同類型的數據，例如 cell、slice 或 builder。這些值類型是 FunC 語言中常用的數據結構。
-}

```
;;;     設置新的鍵值對添加到字典中。如果鍵已經存在，則不會覆蓋原有值。
        (cell, int) udict_add?(cell dict, int key_len, int index, slice value) asm(value index dict key_len) "DICTUADD";
        (cell, int) idict_add?(cell dict, int key_len, int index, slice value) asm(value index dict key_len) "DICTIADD";
        (cell, int) udict_add_builder?(cell dict, int key_len, int index, builder value) asm(value index dict key_len) "DICTUADDB";
        (cell, int) idict_add_builder?(cell dict, int key_len, int index, builder value) asm(value index dict key_len) "DICTIADDB";

;;;     設置替換字典中已有的鍵值對。如果鍵不存在，操作將無效。
        (cell, int) udict_replace?(cell dict, int key_len, int index, slice value) asm(value index dict key_len) "DICTUREPLACE";
        (cell, int) idict_replace?(cell dict, int key_len, int index, slice value) asm(value index dict key_len) "DICTIREPLACE";
        (cell, int) udict_replace_builder?(cell dict, int key_len, int index, builder value) asm(value index dict key_len) "DICTUREPLACEB";
        (cell, int) idict_replace_builder?(cell dict, int key_len, int index, builder value) asm(value index dict key_len) "DICTIREPLACEB";

;;;     無論鍵是否已存在，都會設置字典中指定鍵對應的值。如果鍵已存在，則直接覆蓋原有值；如果鍵不存在，則添加該鍵值對。
        cell udict_set(cell dict, int key_len, int index, slice value) asm(value index dict key_len) "DICTUSET";
        (cell, ()) ~udict_set(cell dict, int key_len, int index, slice value) asm(value index dict key_len) "DICTUSET";
        cell idict_set(cell dict, int key_len, int index, slice value) asm(value index dict key_len) "DICTISET";
        (cell, ()) ~idict_set(cell dict, int key_len, int index, slice value) asm(value index dict key_len) "DICTISET";
        cell dict_set(cell dict, int key_len, slice index, slice value) asm(value index dict key_len) "DICTSET";
        (cell, ()) ~dict_set(cell dict, int key_len, slice index, slice value) asm(value index dict key_len) "DICTSET";
        cell idict_set_ref(cell dict, int key_len, int index, cell value) asm(value index dict key_len) "DICTISETREF";
        (cell, ()) ~idict_set_ref(cell dict, int key_len, int index, cell value) asm(value index dict key_len) "DICTISETREF";
        cell udict_set_ref(cell dict, int key_len, int index, cell value) asm(value index dict key_len) "DICTUSETREF";
        (cell, ()) ~udict_set_ref(cell dict, int key_len, int index, cell value) asm(value index dict key_len) "DICTUSETREF";
        cell udict_set_builder(cell dict, int key_len, int index, builder value) asm(value index dict key_len) "DICTUSETB";
        (cell, ()) ~udict_set_builder(cell dict, int key_len, int index, builder value) asm(value index dict key_len) "DICTUSETB";
        cell idict_set_builder(cell dict, int key_len, int index, builder value) asm(value index dict key_len) "DICTISETB";
        (cell, ()) ~idict_set_builder(cell dict, int key_len, int index, builder value) asm(value index dict key_len) "DICTISETB";
        cell dict_set_builder(cell dict, int key_len, slice index, builder value) asm(value index dict key_len) "DICTSETB";
        (cell, ()) ~dict_set_builder(cell dict, int key_len, slice index, builder value) asm(value index dict key_len) "DICTSETB";

;;;     更新字典及回傳更新後的字典及設置前的 cell。
        (cell, cell) idict_set_get_ref(cell dict, int key_len, int index, cell value) asm(value index dict key_len) "DICTISETGETOPTREF";
        (cell, cell) udict_set_get_ref(cell dict, int key_len, int index, cell value) asm(value index dict key_len) "DICTUSETGETOPTREF";

;;;     回傳字典對應的值， "?" 性質操作回傳的 int 指示該鍵是否存在。存在 -1；否則為 0。
        cell idict_get_ref(cell dict, int key_len, int index) asm(index dict key_len) "DICTIGETOPTREF";
        (cell, int) idict_get_ref?(cell dict, int key_len, int index) asm(index dict key_len) "DICTIGETREF" "NULLSWAPIFNOT";
        (cell, int) udict_get_ref?(cell dict, int key_len, int index) asm(index dict key_len) "DICTUGETREF" "NULLSWAPIFNOT";
        (slice, int) idict_get?(cell dict, int key_len, int index) asm(index dict key_len) "DICTIGET" "NULLSWAPIFNOT";
        (slice, int) udict_get?(cell dict, int key_len, int index) asm(index dict key_len) "DICTUGET" "NULLSWAPIFNOT";

;;;     從字典中刪除與特定位長的鍵（key）所對應的值，回傳更新後的字典及成功狀態，-1 表示成功，0 表示鍵不存在或刪除失敗 。
        (cell, int) idict_delete?(cell dict, int key_len, int index) asm(index dict key_len) "DICTIDEL";
        (cell, int) udict_delete?(cell dict, int key_len, int index) asm(index dict key_len) "DICTUDEL";

;;;     從字典中刪除指定的鍵並返回刪除的值，cell：更新後的字典，slice:被刪除的值，int:成功狀態。
        (cell, slice, int) idict_delete_get?(cell dict, int key_len, int index) asm(index dict key_len) "DICTIDELGET" "NULLSWAPIFNOT";
        (cell, slice, int) udict_delete_get?(cell dict, int key_len, int index) asm(index dict key_len) "DICTUDELGET" "NULLSWAPIFNOT";
        (cell, (slice, int)) ~idict_delete_get?(cell dict, int key_len, int index) asm(index dict key_len) "DICTIDELGET" "NULLSWAPIFNOT";
        (cell, (slice, int)) ~udict_delete_get?(cell dict, int key_len, int index) asm(index dict key_len) "DICTUDELGET" "NULLSWAPIFNOT";

;;;     刪除字典中最小 / 最大的鍵值對，並回傳：更新後的字典 - 刪除的鍵 - 刪除的值 - 成功狀態。
        (cell, int, slice, int) udict_delete_get_min(cell dict, int key_len) asm(-> 0 2 1 3) "DICTUREMMIN" "NULLSWAPIFNOT2";
        (cell, (int, slice, int)) ~udict::delete_get_min(cell dict, int key_len) asm(-> 0 2 1 3) "DICTUREMMIN" "NULLSWAPIFNOT2";
        (cell, int, slice, int) idict_delete_get_min(cell dict, int key_len) asm(-> 0 2 1 3) "DICTIREMMIN" "NULLSWAPIFNOT2";
        (cell, (int, slice, int)) ~idict::delete_get_min(cell dict, int key_len) asm(-> 0 2 1 3) "DICTIREMMIN" "NULLSWAPIFNOT2";
        (cell, slice, slice, int) dict_delete_get_min(cell dict, int key_len) asm(-> 0 2 1 3) "DICTREMMIN" "NULLSWAPIFNOT2";
        (cell, (slice, slice, int)) ~dict::delete_get_min(cell dict, int key_len) asm(-> 0 2 1 3) "DICTREMMIN" "NULLSWAPIFNOT2";
        (cell, int, slice, int) udict_delete_get_max(cell dict, int key_len) asm(-> 0 2 1 3) "DICTUREMMAX" "NULLSWAPIFNOT2";
        (cell, (int, slice, int)) ~udict::delete_get_max(cell dict, int key_len) asm(-> 0 2 1 3) "DICTUREMMAX" "NULLSWAPIFNOT2";
        (cell, int, slice, int) idict_delete_get_max(cell dict, int key_len) asm(-> 0 2 1 3) "DICTIREMMAX" "NULLSWAPIFNOT2";
        (cell, (int, slice, int)) ~idict::delete_get_max(cell dict, int key_len) asm(-> 0 2 1 3) "DICTIREMMAX" "NULLSWAPIFNOT2";
        (cell, slice, slice, int) dict_delete_get_max(cell dict, int key_len) asm(-> 0 2 1 3) "DICTREMMAX" "NULLSWAPIFNOT2";
        (cell, (slice, slice, int)) ~dict::delete_get_max(cell dict, int key_len) asm(-> 0 2 1 3) "DICTREMMAX" "NULLSWAPIFNOT2";

;;;     根據不同的條件來獲取最小或最大鍵值對應的內容，min 和 max 函數負責獲取最小和最大的鍵值，而 ref 函數則返回值的引用。 回傳： 鍵 - 值 - 狀態。
        (int, slice, int) udict_get_min?(cell dict, int key_len) asm (-> 1 0 2) "DICTUMIN" "NULLSWAPIFNOT2";
        (int, slice, int) udict_get_max?(cell dict, int key_len) asm (-> 1 0 2) "DICTUMAX" "NULLSWAPIFNOT2";
        (int, cell, int) udict_get_min_ref?(cell dict, int key_len) asm (-> 1 0 2) "DICTUMINREF" "NULLSWAPIFNOT2";
        (int, cell, int) udict_get_max_ref?(cell dict, int key_len) asm (-> 1 0 2) "DICTUMAXREF" "NULLSWAPIFNOT2";
        (int, slice, int) idict_get_min?(cell dict, int key_len) asm (-> 1 0 2) "DICTIMIN" "NULLSWAPIFNOT2";
        (int, slice, int) idict_get_max?(cell dict, int key_len) asm (-> 1 0 2) "DICTIMAX" "NULLSWAPIFNOT2";
        (int, cell, int) idict_get_min_ref?(cell dict, int key_len) asm (-> 1 0 2) "DICTIMINREF" "NULLSWAPIFNOT2";
        (int, cell, int) idict_get_max_ref?(cell dict, int key_len) asm (-> 1 0 2) "DICTIMAXREF" "NULLSWAPIFNOT2";
```

```
;;;     next 獲取比指定鍵大的下一個鍵；prev 查找比指定鍵小的上一個鍵；回傳： 鍵 - 值 - 狀態。
        (int, slice, int) udict_get_next?(cell dict, int key_len, int pivot) asm(pivot dict key_len -> 1 0 2) "DICTUGETNEXT" "NULLSWAPIFNOT2";
        (int, slice, int) udict_get_nexteq?(cell dict, int key_len, int pivot) asm(pivot dict key_len -> 1 0 2) "DICTUGETNEXTEQ" "NULLSWAPIFNOT2";
        (int, slice, int) udict_get_prev?(cell dict, int key_len, int pivot) asm(pivot dict key_len -> 1 0 2) "DICTUGETPREV" "NULLSWAPIFNOT2";
        (int, slice, int) udict_get_preveq?(cell dict, int key_len, int pivot) asm(pivot dict key_len -> 1 0 2) "DICTUGETPREVEQ" "NULLSWAPIFNOT2";
        (int, slice, int) idict_get_next?(cell dict, int key_len, int pivot) asm(pivot dict key_len -> 1 0 2) "DICTIGETNEXT" "NULLSWAPIFNOT2";
        (int, slice, int) idict_get_nexteq?(cell dict, int key_len, int pivot) asm(pivot dict key_len -> 1 0 2) "DICTIGETNEXTEQ" "NULLSWAPIFNOT2";
        (int, slice, int) idict_get_prev?(cell dict, int key_len, int pivot) asm(pivot dict key_len -> 1 0 2) "DICTIGETPREV" "NULLSWAPIFNOT2";
        (int, slice, int) idict_get_preveq?(cell dict, int key_len, int pivot) asm(pivot dict key_len -> 1 0 2) "DICTIGETPREVEQ" "NULLSWAPIFNOT2";

;;;     初始化一個空字典以便後續操作，回傳字典 cell。
        cell new_dict() asm "NEWDICT";

;;;     檢查字典是否為空，回傳 1 表示字典為空，0 表示字典不是空。
        int dict_empty?(cell c) asm "DICTEMPTY";

;;;     處理一系列擁有相同開頭或共同特徵的 key，針對這個共同的前綴進行高效的查找、設置或刪除；
;;;     pfxdict_get? 回傳匹配的值 - 匹配的鍵 - 原始的前綴鍵 - 狀態； pfxdict_set? & pfxdict_delete? 回傳更新後的字典及成功狀態。
        (slice, slice, slice, int) pfxdict_get?(cell dict, int key_len, slice key) asm(key dict key_len) "PFXDICTGETQ" "NULLSWAPIFNOT2";
        (cell, int) pfxdict_set?(cell dict, int key_len, slice key, slice value) asm(value key dict key_len) "PFXDICTSET";
        (cell, int) pfxdict_delete?(cell dict, int key_len, slice key) asm(key dict key_len) "PFXDICTDEL";

;;;     獲取全局配置參數，它返回的可能是具體的 cell 或者 null。
        cell config_param(int x) asm "CONFIGOPTPARAM";
```

```
{-
    raw_reserve 決定智能合約經操作後的餘額保留方式，amount 是用戶指定的值，mode 決定了值的用法，Bit flag 允許不同情境下的行為。

        raw_reserve 的 mode 決定了具體的保留方式：
            mode = 0 ： 強制保留指定的 amount，如果餘額不足，操作失敗。
            mode = 2 ： 允許保留少於 amount 的餘額，以防止操作失敗。
            mode = 1 或 mode = 3 ： 在這模式，amount 是讓給後續操作所使用的金額，而不是保留的金額。
        raw_reserve 的 mode 中添加 Bit flags 會進一步影響保留行為：
            +2 ： 當賬戶沒有足夠資金保留指定 amount，把賬戶所有餘額保留。即使資金不足，也能保留盡可能多的金額，而不會中止操作。
            +4 ： 在保留操作前，把最終保留在合約的金額設置成 初始餘額加上 amount，確保保留智能合約中的全部資金並額外加上一個特定數額。
            +8 ： 在保留操作前，把最終保留在合約的金額設置成 初始餘額減去 amount，根據某些計算結果或變化來動態調整最終保留的資金量。

    raw_reserve_extra 除了處理保留 TonCoin 的操作，還可以處理其他貨幣的價保留方案。用於從智能合約發送信息
        extra_amount 是一個字典 cell，可包含多種不同類型的貨幣及數量。如果不需要保留額外貨幣，extra_amount 可以設置為 null。

    send_raw_message 用於從智能合約發送信息，信息中某些資訊（如 ihr_fee, fwd_fee, created_lt, created_at）可以設置為任意值，
        這些值會在操作階段自動替換；而源地址 (source address) 可以亦可設為 addr_none，消息發送過程中將會自動替換為當前智能合約的地址。

    send_message 用於發送普通信息，必要內容會被自動設置並填充正確的值，更適合常規操作和資金轉移應用，mode 的用法與 send_raw_message 相同。
        回傳的 int 是信息發送 gas fee，即執行操作時從智能合約中扣除的費用，由系統計算用來支付運行此信息操作的成本。

        send_message & send_raw_message 的 mode 決定信息的操作及處理方式：
            mode = 0 ： 發送常規普通信息，進行簡單的合約間通信或資金轉移，沒有特殊行為。
            mode = 128 ： 指示信息將攜帶當前智能合約的全部餘額，用於清空合約餘額資金，並發送到指定目標地址。
            mode = 64 ： 用戶操作此合約所剩餘的資金，將添加到新信息中一起發送，當 Bit flag 未設置，這意味著 gas fee 將從中扣除。
        send_message & send_raw_message 的 mode 中添加 Bit flags 的作用：
            +1 ： 由發送方單獨支付轉賬費用，而不是從信息攜帶餘額中扣除，有助更精確地控制資金流，確保轉賬的資金數額。
            +2 ： 強行忽略信息處理過程中出現的錯誤，對容錯性要求較高的操作非常有用，將確保操作執行不會因意外錯誤而中斷。
            +32 ： 當信息發送後賬戶餘額為零，銷毀當前賬戶。可與 mode = 128 一起使用，當合約完成預定操作，便立即執行合約的清算和銷毀。
-}
        () raw_reserve(int amount, int mode) impure asm "RAWRESERVE";
        () raw_reserve_extra(int amount, cell extra_amount, int mode) impure asm "RAWRESERVEX";
        () send_raw_message(cell msg, int mode) impure asm "SENDRAWMSG";
        int send_message(cell msg, int mode) impure asm "SENDMSG";

;;; 常見發送模式，適用於普通信息傳遞，gas 會從發送金額扣除，並且不忽略操作階段的任何錯誤。
            const int SEND_MODE_REGULAR = 0;
;;; 轉賬費用將由發送方另外單獨支付，而不是從發送金額中扣除。
            const int SEND_MODE_PAY_FEES_SEPARATELY = 1;
;;; 操作階段中任何錯誤都將被忽略，繼續執行而不報錯失敗。
            const int SEND_MODE_IGNORE_ERRORS = 2;
;;; 如消息發送後賬戶的餘額為零，銷毀此賬戶。
            const int SEND_MODE_DESTROY = 32;
;;; 將所有入站消息所剩餘的資金，將添加到新信息中一起發送。
            const int SEND_MODE_CARRY_ALL_REMAINING_MESSAGE_VALUE = 64;
;;; 將智能合約中的所有餘額添加到新信息中一起轉移。
            const int SEND_MODE_CARRY_ALL_BALANCE = 128;
;;; 操作失敗時不丟失資金，而是退回交易。
            const int SEND_MODE_BOUNCE_ON_ACTION_FAIL = 16;
;;; 是一個新增功能，僅用於估算信息費用，不會實際創建或執行發送信息。
```

```
        const int SEND_MODE_ESTIMATE_FEE_ONLY = 1024;
    ;;; 用於確保特定金額被精確保留在合約的情境。
        const int RESERVE_REGULAR = 0;
    ;;; 當合約餘額不確定，執行潛在高耗費操作前保留盡可能多的資金時使用這個模式，但如不確保能保留精確金額。
        const int RESERVE_AT_MOST = 2;
    ;;; 在操作失敗時會發送反彈交易，確保在錯誤情況下適當地處理資金。
        const int RESERVE_BOUNCE_ON_ACTION_FAIL = 16;



{-
    gas_consumed 回傳智能合約在當前操作消耗的 gas 數量，與合約執行的指令數量和複雜度相關；
    get_compute_fee 根據指定的 workchain 和 gas_used，計算並回傳運算部份的費用；
    get_storage_fee 根據指定的 workchain、以秒為單位的持續存儲時間、使用的 bits 和 cells 數量，計算並回傳存儲費用；
    get_forward_fee 根據 workchain、以位元為單位的信息大小和 cells 總量，計算並回傳信息的轉發費用；
    get_precompiled_gas_consumption 回傳當前優化後的預編譯合約所需消耗的 gas 數量；
    get_simple_compute_fee & get_simple_forward_fee 以較簡單快速的方式計算合約執行成本及轉發成本；
    get_original_fwd_fee 根據 workchain 和 fwd_fee（轉發費用），計算原始的轉發費用，用於追溯或記錄最初的轉發成本；
    get_fee_cofigs 以 tuple 形式回傳當前區塊鏈的費用配置，這些配置包括 gas 費用、存儲費用和轉發費用的基本參數；
    my_storage_due 根據合約佔用的存儲資源和時間計算並回傳當前智能合約應支付的存儲費用 （rent）。
-}
        int gas_consumed() asm "GASCONSUMED";
        int get_compute_fee(int workchain, int gas_used) asm(gas_used workchain) "GETGASFEE";
        int get_storage_fee(int workchain, int seconds, int bits, int cells) asm(cells bits seconds workchain) "GETSTORAGEFEE";
        int get_forward_fee(int workchain, int bits, int cells) asm(cells bits workchain) "GETFORWARDFEE";
        int get_precompiled_gas_consumption() asm "GETPRECOMPILEDGAS";
        int get_simple_compute_fee(int workchain, int gas_used) asm(gas_used workchain) "GETGASFEESIMPLE";
        int get_simple_forward_fee(int workchain, int bits, int cells) asm(cells bits workchain) "GETFORWARDFEESIMPLE";
        int get_original_fwd_fee(int workchain, int fwd_fee) asm(fwd_fee workchain) "GETORIGINALFWDFEE";
        tuple get_fee_cofigs() asm "UNPACKEDCONFIGTUPLE";
        int my_storage_due() asm "DUEPAYMENT";
        ;; () buy_gas(int amount) impure asm "BUYGAS";
        () throw_if(int excno, int cond) impure asm "THROWARGIF";



{-
```

TL-B 信息頭部資訊

int_msg_info$0 定義頭部信息結構（$0 表示合約內傳信息）用於 TON 區塊鏈節點之間的傳遞指示，以執行合約或轉發資金。
ihr_disabled: Bool 表示 IHR 是否禁用（Instant Hypercube Routing），IHR 通過減少跨分片鏈通信路徑長度，減少網絡負載，允許信息在幾乎即時地傳遞；
ihr_fee: Grams 和 fwd_fee: Grams 分別為 IHR 費用和轉發費用；bounce: Bool 指示當信息無法送達時是否退回；bounced: Bool 表示消息是否為退回信息；
src: MsgAddressInt 和 dest: MsgAddressInt 分別為內部發送賬戶和內部接收賬戶的地址；value: CurrencyCollection 信息攜帶的資金量 “Grams” 和 “ExtraCurrencyCollection”；
CommonMsgInfoRelaxed 與 CommonMsgInfo 類似，但 "Relaxed" 的 src 地址允許更靈活的消息地址格式，可輸入更通用的 MsgAddress 類型，而不必是 MsgAddressInt。

TL-B 信息結構

message$_ {X:Type} 描述信息整體結構並定義信息實際內容和行為，包括信息頭部資訊（info）、初始化狀態（init）和信息本體（body）；
info 就是以上提及的頭部信息資訊；init 是可選的，在需要初始化或更新合約狀態或代碼時使用；body 用於傳遞具體數據或觸發業務邏輯的操作指令；
init 使用 Maybe (Either StateInit ^StateInit) 來表示，StateInit 就是描述合約初始化狀態的結構，通常會被存放在引用（ref）節省空間；
body 中包含任意類型的 X 數據或引用，通常是用戶的交易指令、數據更新請求、事件通知等。

```
-}


;;;     定義 BOUNCEABLE & NON_BOUNCEABLE MSG FLAGS 為可回彈與不回彈信息的配置設定。
        const int BOUNCEABLE = 0x18; ;; 0b011000 tag - 0, ihr_disabled - 1, bounce - 1, bounced - 0, src = adr_none$00
        const int NON_BOUNCEABLE = 0x10; ;; 0b010000 tag - 0, ihr_disabled - 1, bounce - 0, bounced - 0, src = adr_none$00



;;;     將信息標記（msg_flags）和空地址（address_none）存儲在建構器（b），並通過 store_uint 將 6 位元的 msg_flags 轉換並存儲在指定的建構器中。
        builder store_msg_flags_and_address_none(builder b, int msg_flags) inline {
            return b.store_uint(msg_flags, 6); }



;;;     從切片（slice）中讀取 4 位元信息標誌，回傳提取 msg_flags 後的切片及提取出的標誌 msg_flags 位元。
        (slice, int) ~load_msg_flags(slice s) inline {
            return s.load_uint(4); }



;;;     檢查 msg_flags 資料底部判斷是否回彈信息，如果最底部為 1 則表示該消息是彈回的。
```

```
int is_bounced(int msg_flags) inline {
   return msg_flags & 1; }
```

;;; 處理回彈信息，跳過特定的 32 位回彈前綴部分。

```
(slice, ()) ~skip_bounced_prefix(slice s) inline {
   return (s.skip_bits(32), ()); } ;; skip 0xFFFFFFFF prefix
```

{-

MSG_INFO_REST_BITS 定義標準信息部分 msg_info 的位元分佈常數：

1 bit: 標記 bounce 標誌

4 bits: 標記信息類型或模式

4 bits: 標記信息功能或子類型

64 bits: 標記信息發送時間戳

32 bits: 標記智能合約執行的邏輯時間

MSG_WITH_STATE_INIT_AND_BODY_SIZE 在 MSG_INFO_REST_BITS 的基礎上，加上 3 個額外的位：

1 bit: 標記是否包含狀態初始化 state_init

1 bit: 標記 state_init 是否為外部引用（ref）

1 bit: 標記信息主體是否存有外部引用

MSG_HAVE_STATE_INIT 檢查設置信息是否包括狀態初始化部分；

MSG_STATE_INIT_IN_REF 確認 StateInit 是否以引用的方式包含在消息中；

MSG_BODY_IN_REF 確認信息主體（Body）是否是以引用的形式包含其中，而不是直接嵌入信息內部；

MSG_ONLY_BODY_SIZE 包含信息主體的所需的位大小，對處理不含 StateInit 部分的簡單消息非常有用，確保信息在系統內部的正確解析和操作。

-}

```
const int MSG_INFO_REST_BITS = 1 + 4 + 4 + 64 + 32;
const int MSG_WITH_STATE_INIT_AND_BODY_SIZE = MSG_INFO_REST_BITS + 1 + 1 + 1;
const int MSG_HAVE_STATE_INIT = 4;
const int MSG_STATE_INIT_IN_REF = 2;
const int MSG_BODY_IN_REF = 1;
const int MSG_ONLY_BODY_SIZE = MSG_INFO_REST_BITS + 1 + 1;
```

{-

store_statinit_ref_and_body_ref 構建一個包含狀態初始化信息和信息主體的完整信息， store_only_body_ref 則僅包含信息主體。

store_prefix_only_body 僅佔用 / 構建一個信息主體大小的空間，並不附帶任何信息主體引用；

~retrieve_fwd_fee 從完整的信息片段（slice）解析並提取轉發費用（fwd_fee）數值。

-}

```
builder store_statinit_ref_and_body_ref(builder b, cell state_init, cell body) inline {
   return b
   .store_uint(MSG_HAVE_STATE_INIT + MSG_STATE_INIT_IN_REF + MSG_BODY_IN_REF, MSG_WITH_STATE_INIT_AND_BODY_SIZE)
   .store_ref(state_init)
   .store_ref(body);
  }
builder store_only_body_ref(builder b, cell body) inline {
   return b
   .store_uint(MSG_BODY_IN_REF, MSG_ONLY_BODY_SIZE)
   .store_ref(body);
  }
builder store_prefix_only_body(builder b) inline {
   return b
   .store_uint(0, MSG_ONLY_BODY_SIZE);
  }
(slice, int) ~retrieve_fwd_fee(slice in_msg_full_slice) inline {
   in_msg_full_slice~load_msg_addr(); ;; skip dst
   in_msg_full_slice~load_coins(); ;; skip value
   in_msg_full_slice~skip_dict(); ;; skip extracurrency collection
   in_msg_full_slice~load_coins(); ;; skip ihr_fee
   int fwd_fee = in_msg_full_slice~load_coins();
   return (in_msg_full_slice, fwd_fee);
  }
```

{-

~load_op、~skip_op 和 store_op 函數用於信息主體與操作碼的 讀取 / 跳過 / 寫入操作；

~load_query_id、~skip_query_id 和 store_query_id 函數用於信息主體與查詢 ID 的 讀取 / 跳過 / 寫入操作；

~load_op_and_query_id 同時從信息主體讀取操作碼（op）和查詢 ID（query_id）。

-}

```
(slice, int) ~load_op(slice s) inline {
   return s.load_uint(MSG_OP_SIZE);
  }
(slice, ()) ~skip_op(slice s) inline {
```

```
        return (s.skip_bits(MSG_OP_SIZE), ());
    }
    builder store_op(builder b, int op) inline {
        return b.store_uint(op, MSG_OP_SIZE);
    }
    (slice, int) ~load_query_id(slice s) inline {
        return s.load_uint(MSG_QUERY_ID_SIZE);
    }
    (slice, ()) ~skip_query_id(slice s) inline {
        return (s.skip_bits(MSG_QUERY_ID_SIZE), ());
    }
    builder store_query_id(builder b, int query_id) inline {
        return b.store_uint(query_id, MSG_QUERY_ID_SIZE);
    }
    (slice, (int, int)) ~load_op_and_query_id(slice s) inline {
        int op = s~load_op();
        int query_id = s~load_query_id();
        return (s, (op, query_id));
    }


;;;    TOKEN_METADATA

;;;    將不同的元數據條目（例如名稱、符號、描述等）存儲在同一個字典中，以便在需要時檢索。
    (cell, ()) ~set_token_snake_metadata_entry(cell content_dict, int key, slice value) impure {
        content_dict~udict_set_ref(256, key, begin_cell().store_uint(0, 8).store_slice(value).end_cell());
        return (content_dict, ());
    }

;;;    將多個元數據條目打包為存儲整體的代幣的元數據單元結構。
    cell create_token_onchain_metadata(cell content_dict) inline {
        return begin_cell().store_uint(0, 8).store_dict(content_dict).end_cell();
    }
```