

```
;; 第 5 代錢包代碼

;; 繁體中文註解版 by Y.C.

;; 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 wallet_v5.fc 服用。
```

## {- # 錢包的功能擴展

在第 5 代 Wallet 中，extensions 的功能和用途與 Wallet v4 中的 plug-ins 基本相同，這些 extensions 允許其他鏈上合約擴展錢包功能，並在需要時被添加或移除。

每個 extension 都是一個外部合約的引用，其中包括常用的 Jetton Master 合約，extensions 允許錢包通過 Jetton Master 合約中的功能來管理帳戶中 Jetton Token 的狀態和操作，如餘額的查詢和管理代幣轉帳以及其他合約狀態。因此錢包將可以透過 extensions 動態調用主合約的功能，進而管理個別帳戶的狀態變動。

```
-}
```

```
#pragma version =0.4.4;
```

```
;; 如直接複製貼上，需要修改路徑至工具庫擺放的資料夾位置。
```

```
#include "stdlib_zh.fc";
```

```
;; 定義各種情況下的錯誤碼，可用於回彈信息讓帳戶/合約邏輯了解報錯原因。
```

```
const int error::signature_disabled = 132;
const int error::invalid_seqno = 133;
const int error::invalid_wallet_id = 134;
const int error::invalid_signature = 135;
const int error::expired = 136;
const int error::external_send_message_must_have_ignore_errors_send_mode = 137;
const int error::invalid_message_operation = 138;
const int error::add_extension = 139;
const int error::remove_extension = 140;
const int error::unsupported_action = 141;
const int error::disable_signature_when_extensions_is_empty = 142;
const int error::this_signature_mode_already_set = 143;
const int error::remove_last_extension_when_signature_disabled = 144;
const int error::extension_wrong_workchain = 145;
const int error::only_extension_can_change_signature_mode = 146;
const int error::invalid_c5 = 147;
```

```
;; 定義資料格式大小（所佔的 bits）。
```

```
const int size::bool = 1;
const int size::seqno = 32;
const int size::wallet_id = 32;
const int size::public_key = 256;
const int size::valid_until = 32;
const int size::message_flags = 4;
const int size::signature = 512;
const int size::message_operation_prefix = 32;
```

```

const int size::address_hash_size = 256;

const int size::query_id = 64;

;; 定義前綴編碼，用於識別來源信息的目的。

const int prefix::signed_external = 0x7369676E;
const int prefix::signed_internal = 0x73696E74;
const int prefix::extension_action = 0x6578746E;

;; -----

;; 根據前綴識別消息類型，針對添加 / 移除 / 設置簽名允許模式對 extensions 進行處理。

(slice, int) check_and_remove_add_extension_prefix(slice body) impure asm "{02} SDBEGINSQ";
(slice, int) check_and_remove_remove_extension_prefix(slice body) impure asm "{03} SDBEGINSQ";
(slice, int) check_and_remove_set_signature_allowed_prefix(slice body) impure asm "{04} SDBEGINSQ";

;; 計算 slice 位元尾部 0 的數量，用於驗證信息中的特定格式和結構。

int count_trailing_zeroes(slice s) asm "SDCNTTRAIL0";

;; 從 slice "s" 中提取最後的 "n" 個位元 (get_last_bits)，或者刪除最後的 "n" 個位元 (remove_last_bits)
;; 用於解析消息或數據中的特定部分。（因 cell 的格式限制，所以  $0 \leq n \leq 1023$ ，而 n 與以下公式的 1 同義）

slice get_last_bits(slice s, int l) asm "SDCUTLAST";
slice remove_last_bits(slice s, int l) asm "SDSKIPLAST";

;; 從信息主體移除 action_send_msg 的前綴 (0x0ec3c86d)，用於識別發送操作並進行相應處理。

slice enforce_and_remove_action_send_msg_prefix(slice body) impure asm "{0ec3c86d} SDBEGINSQ";

;; 將原始操作隊列放進 C5 寄存器，通常是代幣操作或有關管理 extensions 的操作流程。
() set_c5_actions(cell action_list) impure asm "c5 POP";

;; 將 cell 轉換為 slice，並另外回傳一個 flag，表示該 cell 是否異常，用於開始讀取信息數據；
;; 異常的類型可於稍後可以從 slice 的前八位元反序列化，從而得到其信息的操作類型。

(slice, int) begin_parse_raw(cell c) asm "XCTOS";

;; -----

;; 負責檢查和驗證存儲在 C5 寄存器中的所有操作流程，確保每個操作都是有效的 action_send_msg，
;; 並檢查信息是否設置了「忽略錯誤」模式，如果「忽略錯誤」模式沒有設置則拋出錯誤。

cell verify_c5_actions(cell c5, int is_external) inline {

```

```

;; 由於 begin_parse_raw 引用的 XCTOS 不會自動展開或加載特殊 cell
;; 因此以 0x02、0x03 或 0x04 開頭的 cell 不會通過 action_send_msg 前綴檢查
;; 所以開始讀取的 c5 仍然保持原整形態，用以進一步檢查操作的有效性
(slice cs, _) = c5.begin_parse_raw();
;; 計算有效的程序數量，並在後續操作確保數量不超過 255。
int count = 0;
;; 當 slice cs 還有未處理完的資料程序，就繼續解析操作，確認每個 c5 中存儲的操作有效。
while (~ cs.slice_empty?()) {
    ;; 只持有 action_send_msg 前綴的程序會被讀取 (0x0ec3c86d) ,
    ;; 確認每個操作均為合規格的信息發送操作，然後移除前綴作進一步處理；
    ;; `action_set_code`/`action_reserve_currency`/`action_change_library` 不受理。
    cs = cs.enforce_and_remove_action_send_msg_prefix();
    ;; 檢查程序操作的 send_mode 長度是否為 8 bits，並確認一帶有 2 個引用：
    ;; 下一個程序的引用 & MessageRelaxed 信息引用。檢查失敗則會拋出 error::invalid_c5 錯誤。
    throw_unless(error::invalid_c5, cs.slice_bits() == 8);    throw_unless(error::invalid_c5,
    cs.slice_refs() == 2);
    ;; 確認 send_mode 設置了「忽略錯誤」模式 (含 +2 flag)，避免信息在失敗時反復重試。
    ;; (如果餘額不足又沒設忽略錯誤，seqno 將不會增加，並且將被一次又一次地處理。)
    ;; action_send_msg#0ec3c86d_mode:(## 8) out_msg:^(MessageRelaxed Any) = OutAction;
    throw_if(error::external_send_message_must_have_ignore_errors_send_mode, is_external &
    (count_trailing_zeroes(cs.preload_bits(7)) > 0));
    ;; 每檢查完一個程序後，開始解析下一個引用並將操作計量器 count 增加 1。
    (cs, _) = cs.preload_ref().begin_parse_raw();
    count += 1;
}
throw_unless(error::invalid_c5, count <= 255);
throw_unless(error::invalid_c5, cs.slice_refs() == 0);
;; 檢查操作數量不能超過 255 及沒有更多的引用，否則會拋出 error::invalid_c5 錯誤。
return c5; ;; 回傳檢查完畢的 c5。
}

;; -----

;; 管理 extensions 的核心部分，並處理從 C5 寄存器中加載的操作。
() process_actions(slice cs, int is_external, int is_extension) impure inline_ref {
    ;; 嘗試從參數傳入的 slice 中加載 extensions 的相關操作程序。
    cell c5_actions = cs~load_maybe_ref();
    ;; 確認 c5_actions 中有操作程序，再經 verify_c5_actions 驗證，並將其設置到 C5 寄存器中。
    ifnot (cell_null?(c5_actions)) {
        set_c5_actions(c5_actions.verify_c5_actions(is_external));
    }
    ;; 檢查其他處理程序，如果沒有則完結並退出合約操作。
    if (cs~load_int(1) == 0) { ;; has_other_actions
        return ();
    }
    ;; 進入擴展功能的處理迴圈，負責 (extensions) 的添加、移除和其他相關操作。
    while (true) {
        ;; 檢查當前程序是否是一個添加或移除 extension 的操作，檢查後移除識別的前綴。
        int is_add_extension = cs~check_and_remove_add_extension_prefix();
        int is_remove_extension = is_add_extension ? 0 : cs~check_and_remove_remove_extension_prefix();
        ;; 如果是以上兩種情況 ...
        if (is_add_extension | is_remove_extension) {

```

```

;; 確認信息發送地址與當前錢包在同一個 workchain 上，否則會拋出錯誤。
(int address_wc, int address_hash) = parse_std_addr(cs~load_msg_addr());
(int my_address_wc, _) = parse_std_addr(my_address());
throw_unless(error::extension_wrong_workchain, my_address_wc == address_wc);
;; 從自身帳戶中提取 extension 字典，並記錄錢包其他關鍵數據。
slice data_slice = get_data().begin_parse();
;; 讀取並重新寫入 extension 的前端資料
slice data_slice_before_extensions
    = data_slice~load_bits(size::bool + size::seqno + size::wallet_id + size::public_key);
;; 提取原有的字典庫 - 先前安裝的 extensions 集成
cell extensions = data_slice.preload_dict();
if (is_add_extension) {
    ;; 把新 extension 添加到字典庫，操作成功後將存儲到合約狀態中。
    (extensions, int is_success) = extensions.udict_add_builder?(
        size::address_hash_size, address_hash, begin_cell().store_int(-1, 1));
    throw_unless(error::add_extension, is_success);
} else {
    ;; 從字典中刪除指定 extension，如果 extension 不存在及簽名不允許，則拋出錯誤。
    (extensions, int is_success) = extensions.udict_delete?(
        size::address_hash_size, address_hash);
    throw_unless(error::remove_extension, is_success);
    int is_signature_allowed = data_slice_before_extensions.preload_int(size::bool);
    ;; 條件 1：檢查 extension 字典在刪除該 extension 後是否變為空；
    ;; 條件 2：檢查是否禁用了簽名模式；
    ;; 如果兩者同時成立（沒有 extensions 且簽名模式禁用），則拋出錯誤；
    ;; 防止帳戶進入"沒有簽名許可" + "沒有任何 extension" 管理帳戶的危險狀態。
    ;; 這是代表整個 wallet v5 進入「死鎖」狀態，既不能接受簽名的外部信息，
    ;; 也無法通過 extension 來執行操作，帳戶中的資產將被凍結並無法恢復。
    throw_if(error::remove_last_extension_when_signature_disabled,
        null?(extensions) & (~ is_signature_allowed));
}
;; 添加或移除 extension 的操作完成後，將修改過的 extension 字典覆蓋，更新合約的狀態。
set_data(begin_cell()
    .store_slice(data_slice_before_extensions)
    .store_dict(extensions)
    .end_cell());
;; 如信息操作被設為簽名模式 ...
} elseif (cs~check_and_remove_set_signature_allowed_prefix()) { ;; allow/disallow signature
    ;; 檢查是否是由 extension 改變簽名模式，如不是由 extension 更改簽名狀態則拋出錯誤。
    throw_unless(error::only_extension_can_change_signature_mode, is_extension);
    ;; 從信息中加載簽名模式的設定值（允許或禁用簽名）。
    int allow_signature = cs~load_int(1);
    ;; 從自身帳戶中提取錢包關鍵數據。
    slice data_slice = get_data().begin_parse();
    ;; 從關鍵數據中檢測簽署狀態。
    int is_signature_allowed = data_slice~load_int(size::bool);
    ;; 當簽名狀態和傳入的 allow_signature 相同則拋出錯誤，因無需重複設置相同的簽名模式。
    throw_if(error::this_signature_mode_already_set, is_signature_allowed == allow_signature);
    ;; 沒有拋出錯誤，則進行簽名模式配對。
    is_signature_allowed = allow_signature;
    ;; 把還沒有讀取的剩餘資料定義為新 slice (seqno, wallet_id, public_key, extensions)
    slice data_tail = data_slice;
    ifnot (allow_signature) { ;; 如果禁用簽名，進一步檢查 extension：
        ;; 跳過序列號、錢包 ID 和公鑰，檢查 extensions 的狀態。
        int is_extensions_not_empty = data_slice.skip_bits(
            size::seqno + size::wallet_id + size::public_key).preload_int(1);
        ;; 禁用了簽名許可 + 沒有 extensions 則拋出錯誤，
        ;; 防止合約在沒有簽名許可和 extensions 的情況下運行。
        ;; 即是防止「死鎖」的檢查。
        throw_unless(error::disable_signature_when_extensions_is_empty,
            is_extensions_not_empty);
    }
    ;; 將修改過的簽名許可狀態，及 extension 字典更新為最新的合約狀態。
    set_data(begin_cell()

```

```

        .store_int(is_signature_allowed, size::bool)
        .store_slice(data_tail) ;; seqno, wallet_id, public_key, extensions
        .end_cell();

;; 其他模式將會報錯 ...
} else {
    throw(error::unsupported_action);
}

;; 如果沒有更多引用則結束操作，否則繼續處理下一個操作。
ifnot (cs.slice_refs()) {
    return ();
}

cs = cs.preload_ref().begin_parse();
}

}

;; -----

;; 驗證帶有簽名請求的有效性，檢查信息序列編碼 (seqno) 和有效期，並確保該信息來自錢包的擁有者。
() process_signed_request(slice in_msg_body, int is_external) impure inline {
    ;; 從信息主體提取簽名並命名這項資料，將用作待會驗證。
    slice signature = in_msg_body.get_last_bits(size::signature);
    ;; 把提取簽名後的資料命名 (名稱為“所簽署資料片段”)。
    slice signed_slice = in_msg_body.remove_last_bits(size::signature);
    ;; 先跳過操作類型前綴，讀取實質內容數據。
    ;; skip signed_internal or signed_external prefix
    slice cs = signed_slice.skip_bits(size::message_operation_prefix);
    ;; 提取 wallet_id (錢包 ID)、valid_until (消息有效期) 和 seqno (序列號) 用於後續驗證。
    (int wallet_id, int valid_until, int seqno) = (cs~load_uint(size::wallet_id),
        cs~load_uint(size::valid_until), cs~load_uint(size::seqno));
    ;; 從自身帳戶中提取錢包關鍵數據。
    slice data_slice = get_data().begin_parse();
    ;; 本錢包的簽名許可狀態
    int is_signature_allowed = data_slice~load_int(size::bool);
    ;; 儲存在帳戶/錢包內的序列號
    int stored_seqno = data_slice~load_uint(size::seqno);
    ;; 把還沒有讀取的剩餘資料定義為“尾部資料” slice(wallet_id, public_key, extensions)
    slice data_tail = data_slice; ;; wallet_id, public_key, extensions
    ;; 另外再分別命名及識別 3 個尾部資料的內容
    int stored_wallet_id = data_slice~load_uint(size::wallet_id);
    int public_key = data_slice~load_uint(size::public_key);
    int is_extensions_not_empty = data_slice.preload_int(1);
    ;; 檢查簽名是否有效
    int is_signature_valid = check_signature(slice_hash(signed_slice), signature, public_key);
    ;; 如果不是“有效”...
    ifnot (is_signature_valid) {
        if (is_external) {
            ;; 外來信息 + 簽名無效，則拋出錯誤。
            throw(error::invalid_signature);
        } else {
            ;; 內部信息，則立即退出。
            return ();
        }
    }
}

```

```

;; 檢查錢包是否允許簽名操作及配置任何 extensions, 防止帳戶死鎖。
throw_if(error::signature_disabled, (~ is_signature_allowed)
        & is_extensions_not_empty);
;; 由於用戶發起一個交易請求, 必須附帶一個當前帳戶的 seqno, 所以這情況下的 2 個值必須一樣。
throw_unless(error::invalid_seqno, seqno == stored_seqno);
;; 如果傳入的 wallet_id 與帳戶的 wallet_id 不匹配, 說明信息不是針對此錢包, 應拒絕執行。
throw_unless(error::invalid_wallet_id, wallet_id == stored_wallet_id);
;; 檢查消息的有效期 valid_until 是否超過當前時間。
throw_if(error::expired, valid_until <= now());
;; 通過所有驗證的外來信息...
if (is_external) {
    ;; 請參考 stdlib.fc 當中的: () accept_message() impure asm "ACCEPT";
    ;; 簡單而言是進行 gas 設置以進一步交互。
    accept_message();
}
;; 更新帳戶的序列號。
stored_seqno = stored_seqno + 1;
;; 把新的合約狀態覆蓋取代舊有狀態。
set_data(begin_cell()
        .store_int(true, size::bool) ;; is_signature_allowed
        .store_uint(stored_seqno, size::seqno)
        .store_slice(data_tail) ;; wallet_id, public_key, extensions
        .end_cell());
if (is_external) {
    ;; 當更新操作遇到異常或失敗, commit() 可確保序列號等關鍵數據的更改仍被保存, 避免重入。
    commit();
}
;; 驗證信息有效性後的最後一步操作, 調用上一個重點函數進行有關 extensions 的操作。
process_actions(cs, is_external, false);
}

;; -----

;; 處理鏈外信息的入口, 確認信息前綴是否 signed_external, 然後傳遞給 process_signed_request 進行處理。
() recv_external(slice in_msg_body) impure inline {
    ;; 檢查信息的操作類型與簽名前綴是否匹配。
    throw_unless(error::invalid_message_operation,
        in_msg_body.preload_uint(size::message_operation_prefix)
        == prefix::signed_external);
    ;; 通過前綴檢查後, 調用 process_signed_request 函數來進一步處理;
    ;; 這裡傳遞的兩個參數是 in_msg_body (消息主體) 和 true (表示這是一條外部消息)。
    process_signed_request(in_msg_body, true);
}

;; -----

;; 處理內部信息的入口, 檢查信息前綴是否 extension action 或 signed_internal,
;; 根據不同前綴處理不同的操作, 並處理來自其他合約擴展功能 (extensions) 的操作。

```

```

() recv_internal(cell in_msg_full, slice in_msg_body) impure inline {
    ;; 請參考新版 stdlib.fc 當中的: int slice_bits(slice s) asm "SBITS";
    ;; 如果信息主體結構位元小於訊息規格前綴, 則退出帳戶操作, 亦表明這只是簡單地接收 TON。
    if (in_msg_body.slice_bits() < size::message_operation_prefix) {
        return (); ;; just receive Toncoins
    }
    ;; 通過前綴規格驗證後, 載入信息的操作類型。
    int op = in_msg_body.preload_uint(size::message_operation_prefix);
    ;; 只要不是 extension_action (擴展功能處理) 及 signed_internal (內部簽名消息),
    ;; 都當作只是簡單地接收 TON 並退出帳戶操作。
    if ((op != prefix::extension_action) & (op != prefix::signed_internal)) {
        return (); ;; just receive Toncoins
    }
    ;; bounced messages has 0xffffffff prefix and skipped by op check
    ;; 如果操作類型是 extension_action (擴展功能處理) ...
    if (op == prefix::extension_action) {
        ;; 跳過操作類型的前綴, 進入處理邏輯。
        in_msg_body~skip_bits(size::message_operation_prefix);
        ;; 提取信息主體的完整數據。
        slice in_msg_full_slice = in_msg_full.begin_parse();
        ;; 跳過信息標誌位元 (message_flags 的 bits)
        in_msg_full_slice~skip_bits(size::message_flags);
        ;; 讀取信息發送地址的所屬工作鏈及其 hash 化的區塊鏈地址。
        (int sender_address_wc, int sender_address_hash)
            = parse_std_addr(in_msg_full_slice~load_msg_addr());
        ;; 讀取自身帳號的所屬工作鏈。
        (int my_address_wc, _) = parse_std_addr(my_address());
        ;; 自身帳號與信息發送地址的所屬工作鏈需要一致, 否則退出帳戶操作。
        if (my_address_wc != sender_address_wc) {
            return ();
        }
        ;; 從帳戶數據中讀取原有的擴展合約 (extensions) 字典。
        cell extensions = get_data().begin_parse()
            .skip_bits(size::bool + size::seqno + size::wallet_id + size::public_key)
            .preload_dict();
        ;; Note that some random contract may have deposited funds with this prefix,
        ;; so we accept the funds silently instead of throwing an error.
        ;; (wallet v4 does the same)
        ;; 簡單檢查是否存有 extension, 沒有的話則退出帳戶操作。
        (_, int extension_found) = extensions.udict_get?(
            size::address_hash_size, sender_address_hash);
        ifnot (extension_found) {
            return ();
        }
        ;; 在這裡無視信息中的 query_id (查詢 ID), 再調用 process_actions 開始處理擴展功能操作。
        in_msg_body~skip_bits(size::query_id); ;; skip query_id
        process_actions(in_msg_body, false, true);
        return ();
    }
    ;; Before signature checking we handle errors silently (return),
    ;; after signature checking we throw exceptions.
    ;; Check to make sure that there are enough bits for reading before signature check
    ;; 信息主體結構位元若小於簽署訊息規格, 則退出帳戶操作, 若通過則進行簽名驗證流程。

```

```
        if (in_msg_body.slice_bits() < size::message_operation_prefix
            + size::wallet_id + size::valid_until + size::seqno + size::signature) {
            return ();
        }
        process_signed_request(in_msg_body, false);
    }
}

;; -----

;; Get methods
;; 檢查當前合約數據是否允許使用簽名。
int is_signature_allowed() method_id {
    return get_data().begin_parse()
        .preload_int(size::bool);
}

;; 返回當前合約的序列號 (seqno) , 防止重入攻擊, 每次處理完信息都會遞增序列號。
int seqno() method_id {
    return get_data().begin_parse()
        .skip_bits(size::bool)
        .preload_uint(size::seqno);
}

;; 返回合約中記錄的 subwallet_id, 這是該錢包的一個 flag, 用於標識多個子錢包。
int get_subwallet_id() method_id {
    return get_data().begin_parse()
        .skip_bits(size::bool + size::seqno)
        .preload_uint(size::wallet_id);
}

;; 返回錢包中的公鑰 (public_key) , 用於驗證來外來信息的簽名。
int get_public_key() method_id {
    return get_data().begin_parse()
        .skip_bits(size::bool + size::seqno + size::wallet_id)
        .preload_uint(size::public_key);
}

;; 以哈希化地址形式, 回傳存儲於錢包的 extensions 字典, 包含所有外部擴展合約的交互接口。
cell get_extensions() method_id {
    return get_data().begin_parse()
        .skip_bits(size::bool + size::seqno + size::wallet_id + size::public_key)
        .preload_dict();
}
```