# 概述

由于 Rust 编译为 WebAssembly，它为编写在 Internet 计算机上运行的应用程序提供了丰富的开发环境。 为了帮助在 Rust 中编写可以部署在 Internet 计算机上的应用程序铺平道路，DFINITY 提供了一些工具来简化过程。

# StartUp

## 环境准备

安装rust，执行命令：

|  |
| --- |
| curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh |

安装cmake，执行命令：

|  |
| --- |
| brew install cmake |

安装dfinity环境，执行命令：

|  |
| --- |
| sh -ci "$(curl -fsSL https://sdk.dfinity.org/install.sh)" |

## 创建项目

使用如下命令创建一个dfinity项目：

|  |
| --- |
| dfx new rust\_hello |

项目会有如下目录结构：

|  |
| --- |
| rust\_hello/  ├── README.md # default project documentation  ├── dfx.json # project configuration file  ├── node\_modules # libraries for front-end development  ├── package-lock.json  ├── package.json  ├── src # source files directory  │ ├── rust\_hello  │ │ └── main.mo  │ └── rust\_hello\_assets  │ ├── assets  │ │ ├── logo.png  │ │ ├── main.css  │ │ └── sample-asset.txt  │ └── src  │ ├── index.html  │ └── index.js  └── webpack.config.js |

其中：

* dfx.json是整个工程的配置信息。
* package.json中是前端项目的依赖以及打包信息。
* src目录下存放整个工程的源文件。
* 约定src/xxx/中存放后端项目的源文件。
* 约定src/xxx\_assets/中存放前端项目的源文件。
* 约定src/xxx\_assets/assets/中存放前端项目的资源文件。
* 约定src/xxx\_assets/src/index.html为前端项目的入口html文件。
* 约定src/xxx\_assets/src/index.js为前端项目的会自动载入的且唯一的JavaScript脚本。

删除rust\_hello目录，并使用如下命令创建rust lib crate：

|  |
| --- |
| cargo init --lib src/rust\_hello |

编辑src/rust\_hello/cargo.toml文件，修改lib类型，并添加ic依赖：

|  |
| --- |
| [lib]  crate-type = ["cdylib"]  [dependencies]  ic-cdk = "0.3"  ic-cdk-macros = "0.3" |

添加接口描述文件src/rust\_hello/src/rust\_hello.did：

|  |
| --- |
| touch src/rust\_hello/src/rust\_hello.did |

在根目录添加workspace cargo.toml文件：

|  |
| --- |
| [workspace]  members = [  "src/rust\_hello",  ] |

编辑dfx.json中的canisters.rust\_hello，使其具有类似如下内容：

|  |
| --- |
| {  "canisters": {  "rust\_hello": {  "build": "cargo build --target wasm32-unknown-unknown --package rust\_hello --release",  "wasm": "target/wasm32-unknown-unknown/release/rust\_hello.wasm",  "candid": "src/rust\_hello/rust\_hello.did",  "type": "custom"  },  ...  },  ...  } |

注意：需要使用rustup target add wasm32-unknown-unknown命令添加wasm32-unknown-unknown target支持。

## 编辑项目

编辑src/rust\_hello/src/lib.rs文件内容为：

|  |
| --- |
| #[ic\_cdk\_macros::query]  fn print() {  ic\_cdk::print("Hello World from DFINITY!");  } |

编辑src/rust\_hello/src/rust\_hello.did文件内容为：

|  |
| --- |
| service : {  "print": () -> () query;  } |

## 部署项目

执行dfx start --background启动dfx本地环境。

执行dfx deploy rust\_hello部署rust\_hello canister。

# dfx.json配置

对于rust编写的canister，需要在dfx.json中的canisters.<canister\_name>进行相应配置，相关配置项如下：

|  |
| --- |
| {  "canisters": {  "my\_rust\_program": {  "type": "custom",  "candid": "src/my\_rust\_program.did",  "wasm": "target/wasm32-unknown-unknown/debug/my\_rust\_program.wasm",  "build": "cargo build --target wasm32-unknown-unknown --package my\_rust\_program"  },  ...  },  ...  } |

* type键设置为custom，表示此容器不是当前识别的（motoko 或assets）容器类型之一。
* candid 键指定用于该canister的 Candid 接口描述文件的名称和位置。
* wasm 键指定了由 cargo build 命令生成的 WebAssembly 文件的路径。
* build 键指定用于构建crate命令。

# Candid类型映射

rust类型与candid类型映射如下：

* text：candid的text映射为rust中的String或&str。
* blob：candid的blob映射为rust中的Vec<u8>或&[u8]。
* nat：candid的nat映射为rust中的candid::Nat或u128。
* int：candid的int映射为rust中的candid::Int或i128。
* natN：candid的nat8、nat16、nat32、nat64对应映射为rust中的u8、u16、u32、u64。
* intN：candid的int8、int16、int32、int64对应映射为rust中的i8、i16、i32、i64。
* float32：candid的float32对应映射为rust中的f32。
* float64：candid的float64对应映射为rust中的f64。
* bool：candid的bool类型对应映射为rust中的bool。
* null：candid中的null类型对应映射为rust中的()。
* vec t：candid中的vec t类型对应映射为rust中的Vec<T>或&[T]。
* opt t：candid中的opt t类型对应映射为rust中的Option<T>。
* record { n : t, … }：candid中的record { n : t, … }类型对应映射rust中使用#[derive(CandidType, Deserialize)]属性注解的struct，且其中字段都对应映射，字段前可以使用#[serde(rename = "DifferentFieldName")]属性来进行字段重命名。如果record中的n是以0开头的连续数值，则其代表一个元组，那么对应映射rust中使用#[derive(CandidType, Deserialize)]属性注解的元组。
* variant { n : t, … }：candid中的variant { n : t, … }类型对应映射rust中使用#[derive(CandidType, Deserialize)]属性注解的enum，其中tag一一对应，字段前可以使用#[serde(rename = "DifferentFieldName")]属性来进行字段重命名。
* principal：candid中的principal类型对应映射rust中的candid::Principal或ic\_types::Principal。
* reserved：candid中的reserved类型对应映射rust中的candid::Reserved。
* empty：candid中的empty类型对应映射rust中的candid::Empty。
* func (…) → (…)：candid中的func (…) → (…)类型对应映射rust中的candid::IDLValue::Func(Principal, String)。
* service {…}：candid中的service {…}类型对应映射rust中的candid::IDLValue::Service(Principal)。

# 库

Rust提供了以下工具，这些工具统称为 Rust 的 DFINITY Canister Development Kit (CDK)：

| **Package** | **Description** |
| --- | --- |
| [ic-types](https://docs.rs/ic-types/0.2.2/ic_types/) | ic-types 库定义了用于与 Internet 计算机副本通信以及构建要部署为 IC上的容器的应用程序时的类型。 |
| [ic-agent](https://docs.rs/ic-agent/0.8.0/ic_agent/) | ic-agent 库可实现与 Internet 计算机副本的直接通信。 |
| [ic-utils](https://docs.rs/ic-utils/0.6.0/ic_utils/) | ic-utils 库提供了用于管理call和部署为容器的应用程序的实用程序代码。 |
| [ic-cdk](https://docs.rs/ic-cdk/0.3.1/ic_cdk/index.html) | ic-cdk 库提供了使 Rust 程序能够与 IC 主系统 API 交互的核心方法。这个库是 Rust CDK 的runtime core。 |
| [ic-cdk-macros](https://docs.rs/ic-cdk-macros/0.3.1/ic_cdk_macros/) | ic-cdk-macros 库定义了有助于构建操作端点和 API 的过程宏。 该库包括用于更新、查询、等宏。 |
| [ic-cdk-optimizer](https://docs.rs/crate/ic-cdk-optimizer/0.3.1) | ic-cdk-optimizer 是一个帮助库，用于减少 WebAssembly 模块的大小。 |

# ic\_cdk\_macros

ic\_cdk\_macros提供了如下几个属性宏，其分为两类：

* 导入宏：
  + import：引用一个actor。
* 导出宏：
  + init：actor初始化方法。
  + pre\_upgrade：在进行actor升级时，在旧actor的stable变量转移到新的stable变量前调用。
  + post\_upgrade：在进行actor升级时，在旧actor的stable变量转移到新的stable变量后立即调用。
  + query：将会生成一个可以进行query call的导出。
  + update：将会生成一个可以进行update call的导出。

## 导入宏

导入宏指，不会产生导出的宏。

### import

import用于引用一个actor。

import属性必须位于struct上，由于import属性只会使用其名称，因此单元struct即可。

import属性会生成该struct的固有实现，其中存在关联函数(IDL定义)和关联常量(canister\_id)，

其核心代码如下：

|  |
| --- |
| pub(crate) fn ic\_import(  attr: proc\_macro::TokenStream,  item: proc\_macro::TokenStream,  ) -> Result<proc\_macro::TokenStream, Error> {  let config = from\_tokenstream::<ImportAttributes>(&proc\_macro2::TokenStream::from(attr))?;  // 可以使用#[import(canister = "multiply\_deps")]来指定一个project内的其他项目创建的actor  // 也可以使用#[import(canister\_id = "xxx", candid\_path = "xxx")]来指定外部项目创建的actor  let (canister\_id, candid\_path) = {  if let Some(canister\_name) = config.canister {  get\_env\_id\_and\_candid(&canister\_name)?  } else if let Some(canister\_id) = config.canister\_id {  if let Some(candid\_path) = config.candid\_path {  (canister\_id, candid\_path)  } else {  return Err(Error::new(  Span::call\_site(),  "Must specify both candid and canister\_id.",  ));  }  } else {  return Err(Error::new(  Span::call\_site(),  "Must specify both candid and canister\_id.",  ));  }  };  let item = syn::parse2::<syn::Item>(proc\_macro2::TokenStream::from(item))?;  // 属性必须位于struct上。  let item = match item {  syn::Item::Struct(item) => item,  \_ => {  return Err(Error::new(  Span::call\_site(),  "import must be used on a struct.",  ))  }  };  let visibility = {  let vis = item.vis;  format!("{}", quote! { #vis })  };  let struct\_name = item.ident.to\_string();  let candid\_str = std::fs::read\_to\_string(&candid\_path).unwrap();  let prog = candid::IDLProg::from\_str(&candid\_str).map\_err(|e| {  Error::new(  Span::call\_site(),  format!("Could not parse the candid file: {}", e),  )  })?;  let bindings = Box::new(RustLanguageBinding {  visibility,  canister\_id,  });  let config = candid::codegen::rust::Config::default()  .with\_actor\_name(struct\_name)  .with\_biguint\_type("candid::Nat".to\_string())  .with\_bigint\_type("candid::Int".to\_string())  .with\_bindings(bindings);  let rust\_str = candid::codegen::idl\_to\_rust(&prog, &config)  .map\_err(|e| Error::new(Span::call\_site(), e.to\_string()))?;  let rust\_str = format!("{} {}", "type principal = Vec<u8>;", rust\_str);  Ok(proc\_macro::TokenStream::from\_str(&rust\_str).unwrap())  } |

#### 引用同project的actor

例子：

|  |
| --- |
| use ic\_cdk\_macros::\*;  use ic\_cdk::export::candid;  #[import(canister = "multiply\_deps")]  struct CounterCanister;  #[update]  async fn read() -> candid::Nat {  CounterCanister::read().await.0  } |

#### 引用外部project的actor

例子：

|  |
| --- |
| use ic\_cdk\_macros::\*;  use ic\_cdk::export::candid;  #[import(canister\_id = "el4ku-6qaaa-aaaah-aaqgq-cai", candid\_path = "multiply\_deps.did")]  struct CounterCanister;  #[update]  async fn read() -> candid::Nat {  CounterCanister::read().await.0  } |

candid\_path可以相对路径或者绝对路径。

## 导出宏

导出宏用于向外部导出一个函数。其核心代码如下，对于不同的属性宏，最终都是引用该段代码：

|  |
| --- |
| fn dfn\_macro(  method: MethodType,  attr: TokenStream,  item: TokenStream,  ) -> Result<TokenStream, Error> {  let attrs = from\_tokenstream::<ExportAttributes>(&attr)?;  // 宏属性必须注解在函数项上。  let fun: ItemFn = syn::parse2::<syn::ItemFn>(item.clone()).map\_err(|e| {  Error::new(  item.span(),  format!("#[{0}] must be above a function, \n{1}", method, e),  )  })?;  let signature = &fun.sig;  let generics = &signature.generics;  // 不允许存在泛型  if !generics.params.is\_empty() {  return Err(Error::new(  generics.span(),  format!(  "#[{}] must be above a function with no generic parameters",  method  ),  ));  }  let is\_async = signature.asyncness.is\_some();  let return\_length = match &signature.output {  ReturnType::Default => 0,  ReturnType::Type(\_, ty) => match ty.as\_ref() {  Type::Tuple(tuple) => tuple.elems.len(),  \_ => 1,  },  };  // init属性宏、pre\_upgrade属性宏、post\_upgrade属性宏注解的函数不允许存在返回值。  match method {  MethodType::Init | MethodType::PreUpgrade | MethodType::PostUpgrade  if return\_length > 0 =>  {  return Err(Error::new(  Span::call\_site(),  format!("#[{}] function cannot have a return value.", method),  ));  }  \_ => {}  }  let (arg\_tuple, \_): (Vec<Ident>, Vec<Box<Type>>) =  get\_args(method, signature)?.iter().cloned().unzip();  let name = &signature.ident;  // 构造出调用函数名  let outer\_function\_ident = Ident::new(  &format!("{}\_{}\_", name.to\_string(), crate::id()),  Span::call\_site(),  );  // 构造出导出函数名  let export\_name = if method.is\_lifecycle() {  format!("{}", method)  } else {  format!(  "{0} {1}",  method,  attrs.name.unwrap\_or\_else(|| name.to\_string())  )  };  // 被包装的函数调用，即被注解的函数调用  let function\_call = if is\_async {  quote! { #name ( #(#arg\_tuple),\* ) .await }  } else {  quote! { #name ( #(#arg\_tuple),\* ) }  };  let arg\_count = arg\_tuple.len();  // 返回内容  let return\_encode = if method.is\_lifecycle() {  quote! {}  } else {  match return\_length {  0 => quote! { ic\_cdk::api::call::reply(()) },  1 => quote! { ic\_cdk::api::call::reply((result,)) },  \_ => quote! { ic\_cdk::api::call::reply(result) },  }  };  // On initialization we can actually not receive any input and it's okay, only if  // we don't have any arguments either.  // If the data we receive is not empty, then try to unwrap it as if it's DID.  // 入参内容  let arg\_decode = if method.is\_lifecycle() && arg\_count == 0 {  quote! {}  } else {  quote! { let ( #( #arg\_tuple, )\* ) = ic\_cdk::api::call::arg\_data(); }  };  // 函数调用守护者，asyn函数没啥用  let guard = if let Some(guard\_name) = attrs.guard {  let guard\_ident = syn::Ident::new(&guard\_name, Span::call\_site());  quote! {  let r: Result<(), String> = #guard\_ident ();  if let Err(e) = r {  ic\_cdk::api::call::reject(&e);  return;  }  }  } else {  quote! {}  };  Ok(quote! {  // 生成了导出函数  #[export\_name = #export\_name]  fn #outer\_function\_ident() {  // 设置环境  ic\_cdk::setup();    // 守护函数调用  #guard  ic\_cdk::block\_on(async {  // 解码入参  #arg\_decode  // 调用函数  let result = #function\_call;  // 返回调用返回值  #return\_encode  });  }  // 原始项  #item  })  } |

### init

init用于初始化canister。其具有如下限制：

* init属性宏只允许使用一次。
* init属性宏必须在一个函数项上。其该函数具有如下限制：
  + 不允许存在泛型。
  + 不允许存在返回值。

init属性宏中可以指定guard属性，例如#[init(guard="check")]，该属性用于声明一个在init函数运行前会运行的守卫函数，该函数必须具有"()->Result<(), String>"样式的签名。

init属性宏注解的函数会在canister进行install或者reinstall时被调用，install或reinstall时使用的参数将会被用作函数的入参。

例子：

|  |
| --- |
| use ic\_cdk\_macros::\*;  use ic\_cdk::export::candid;  static mut COUNTER: Option<candid::Nat> = None;  #[init]  fn init() {  unsafe {  COUNTER = Some(candid::Nat::from(0));  }  } |

### pre\_upgrade

pre\_upgrade用于canister升级。其具有如下限制：

* pre\_upgrade属性宏只允许使用一次。
* pre\_upgrade属性宏必须在一个函数项上。其该函数具有如下限制：
  + 不允许存在泛型。
  + 不允许存在返回值。

pre\_upgrade属性宏中可以指定guard属性，例如#[pre\_upgrade(guard="check")]，该属性用于声明一个在pre\_upgrade函数运行前会运行的守卫函数，该函数必须具有"()->Result<(), String>"样式的签名。

pre\_upgrade属性宏注解的函数会在canister升级时，在旧actor的stable变量转移到新的stable变量前调用。

### post\_upgrade

post\_upgrade用于canister升级。其具有如下限制：

* post\_upgrade属性宏只允许使用一次。
* post\_upgrade属性宏必须在一个函数项上。其该函数具有如下限制：
  + 不允许存在泛型。
  + 不允许存在返回值。

post\_upgrade属性宏中可以指定guard属性，例如#[post\_upgrade(guard="check")]，该属性用于声明一个在post\_upgrade函数运行前会运行的守卫函数，该函数必须具有"()->Result<(), String>"样式的签名。

post\_upgrade属性宏注解的函数会在canister升级时，在旧actor的stable变量转移到新的stable变量后立即调用。canister升级时使用的参数将会被用作函数的入参。

### query

query将会生成一个可以进行query call的导出。其具有如下限制：

* query属性宏必须在一个函数项上。其该函数具有如下限制：
  + 不允许存在泛型。

query属性宏中可以指定guard属性，例如#[query(guard="check")]，该属性用于声明一个在query函数运行前会运行的守卫函数，该函数必须具有"()->Result<(), String>"样式的签名。

query属性宏中可以指定name属性，例如#[query(name = "getSelf")]，该属性声明了导出函数的名称，如果未指定该属性，则使用函数项的名称作为导出名称。

query属性宏注解的函数可以被query call进行调用，query call时指定的入参会作为函数的入参。

例子：

|  |
| --- |
| use ic\_cdk::export::{candid::{CandidType, Deserialize}, Principal};  use ic\_cdk::storage;  use ic\_cdk\_macros::\*;  use std::collections::BTreeMap;  type IdStore = BTreeMap<String, Principal>;  type ProfileStore = BTreeMap<Principal, Profile>;  #[derive(Clone, Debug, Default, CandidType, Deserialize)]  struct Profile {  pub name: String,  pub description: String,  pub keywords: Vec<String>,  }  #[query(name = "getSelf")]  fn get\_self() -> Profile {  let id = ic\_cdk::caller();  let profile\_store = storage::get::<ProfileStore>();  profile\_store  .get(&id)  .cloned()  .unwrap\_or\_else(|| Profile::default())  }  #[query]  fn get(name: String) -> Profile {  let id\_store = storage::get::<IdStore>();  let profile\_store = storage::get::<ProfileStore>();  id\_store  .get(&name)  .and\_then(|id| profile\_store.get(id).cloned())  .unwrap\_or\_else(|| Profile::default())  } |

### update

update将会生成一个可以进行update call的导出。其具有如下限制：

* update属性宏必须在一个函数项上。其该函数具有如下限制：
  + 不允许存在泛型。

update属性宏中可以指定guard属性，例如#[update(guard="check")]，该属性用于声明一个在query函数运行前会运行的守卫函数，该函数必须具有"()->Result<(), String>"样式的签名。

update属性宏中可以指定name属性，例如#[update(name = "setSelf")]，该属性声明了导出函数的名称，如果未指定该属性，则使用函数项的名称作为导出名称。

update属性宏注解的函数可以被update call进行调用，update call时指定的入参会作为函数的入参。

例子：

|  |
| --- |
| use ic\_cdk::export::{candid::{CandidType, Deserialize}, Principal};  use ic\_cdk::storage;  use ic\_cdk\_macros::\*;  use std::collections::BTreeMap;  type IdStore = BTreeMap<String, Principal>;  type ProfileStore = BTreeMap<Principal, Profile>;  #[derive(Clone, Debug, Default, CandidType, Deserialize)]  struct Profile {  pub name: String,  pub description: String,  pub keywords: Vec<String>,  }  #[update]  fn update(profile: Profile) {  let principal\_id = ic\_cdk::caller();  let id\_store = storage::get\_mut::<IdStore>();  let profile\_store = storage::get\_mut::<ProfileStore>();  id\_store.insert(profile.name.clone(), principal\_id.clone());  profile\_store.insert(principal\_id, profile);  } |

# ic\_cdk

ic\_cdk提供了如下项：

* api模块
* export模块
* storage模块
* eprintln声明宏：打印格式化错误消息。
* println声明宏：打印格式化消息。
* block\_on函数：以 WASM 友好的方式阻塞promise（没有多线程！）。
* setup：设置 stdlib 钩子。
* call函数：从ic\_cdk::api::call::call重导出的函数，可以用来调用其他canister的函数。
* caller函数：从ic\_cdk::api::caller重导出的函数，可以用来获取调用者的principal。
* id函数：从ic\_cdk::api::id重导出的函数，可以用来获取当前canister的principal。
* print函数：从ic\_cdk::api::print重导出的函数。
* trap函数：从ic\_cdk::api::trap重导出的函数，可以用来产生一个陷阱，导致调用失败。

## setup函数

setup函数会设置当前canister的panic钩子。

该函数会自动由导出宏生成调用，通常不需要手动调用。在使用声明宏时，展开的代码中，该函数在被注解的函数被调用前进行调用。

其核心代码如下：

|  |
| --- |
| /// Sets a custom panic hook, uses debug.trace  pub fn set\_panic\_hook() {  panic::set\_hook(Box::new(|info| {  let file = info.location().unwrap().file();  let line = info.location().unwrap().line();  let col = info.location().unwrap().column();  let msg = match info.payload().downcast\_ref::<&'static str>() {  Some(s) => \*s,  None => match info.payload().downcast\_ref::<String>() {  Some(s) => &s[..],  None => "Box<Any>",  },  };  let err\_info = format!("Panicked at '{}', {}:{}:{}", msg, file, line, col);  api::print(&err\_info);  }));  } |

每个导出函数内部都会进行钩子设置。

|  |
| --- |
| Ok(quote! {  // 生成了导出函数  #[export\_name = #export\_name]  fn #outer\_function\_ident() {  // 设置环境  ic\_cdk::setup();    // 守护函数调用  #guard  ic\_cdk::block\_on(async {  // 解码入参  #arg\_decode  // 调用函数  let result = #function\_call;  // 返回调用返回值  #return\_encode  });  }  // 原始项  #item  }) |

## block\_on函数

block\_on函数用于阻塞一个promise函数的调用，直到其future返回。

该函数会自动由导出宏生成调用，通常不需要手动调用。在使用声明宏时，展开的代码中，该函数会被用于进行一次包装的async函数调用，async函数包装了被注解的函数的调用。

其核心代码如下：

|  |
| --- |
| pub fn block\_on<F: 'static + Future<Output = ()>>(future: F) {  let future\_ptr = Box::into\_raw(Box::new(future));  let future\_ptr\_ptr: \*mut \*mut dyn Future<Output = ()> = Box::into\_raw(Box::new(future\_ptr));  let mut pinned\_future = unsafe { Pin::new\_unchecked(&mut \*future\_ptr) };  if pinned\_future  .as\_mut()  .poll(&mut Context::from\_waker(&waker::waker(  future\_ptr\_ptr as \*const (),  )))  .is\_ready()  {  unsafe {  let \_ = Box::from\_raw(future\_ptr);  let \_ = Box::from\_raw(future\_ptr\_ptr);  }  }  } |

每个导出函数内部都会对要调用的方法进行阻塞式调用。

|  |
| --- |
| Ok(quote! {  // 生成了导出函数  #[export\_name = #export\_name]  fn #outer\_function\_ident() {  // 设置环境  ic\_cdk::setup();    // 守护函数调用  #guard  ic\_cdk::block\_on(async {  // 解码入参  #arg\_decode  // 调用函数  let result = #function\_call;  // 返回调用返回值  #return\_encode  });  }  // 原始项  #item  }) |

## api模块

api模块提供了系统API和底层函数。其提供以下内容：

* call模块：提供了用于在容器中进行和管理调用的 API。
* stable模块：提供了用于进行数据持久化的API。这里不再展开，主要由storage模块进行了封装。
* caller函数：返回当前调用的调用者。
* canister\_balance函数：获取canister中可用的cycle量。
* set\_certified\_data函数：设置此容器的认证数据。
* data\_certificate函数：当在query call中调用该函数时，返回由该canistter设置的证书数据。
* id函数：返回canister ID。
* time函数：返回当前时间戳，纳秒精度。
* trap函数：产生一个陷阱。
* print函数：打印指定信息。

### call模块

call模块提供用于在容器中进行和管理调用的 API。提供内容如下：

* msg\_cycles\_available函数：获取当前call可以accept的最大cycle数量。
* msg\_cycles\_accept函数：accept指定数量的cycle。
* msg\_cycles\_refunded函数：指示已退的cycle数量。
* arg\_data函数：获取当前调用的入参数据。
* call函数：通过 ic0 对另一个容器执行异步调用。
* call\_raw函数：与“call”相同，但数据没有序列化。
* call\_with\_payment函数：与“call”相同，但增加了发送cycle功能。
* reply函数：使用candid参数响应当前调用。
* reject函数：使用一个消息拒绝当前调用。
* reject\_code函数：获取当前调用的拒绝码。
* reject\_message函数：获取当前调用的拒绝消息。
* result函数：返回当前调用的结果，如果调用成功（T 是 arg\_data），则为 Ok(T)，如果失败则为 reject\_message()。

#### msg\_cycles\_available函数

返回当前call的调用者传输的cycle数量，并且在此消息中仍然可用。

在update方法入口点中，调用msg\_cycles\_available会返回调用者传递给canister的cycle数量。 当cycle被接受 (msg\_cycles\_accept) 时，这会减去已经被accept的cycle。当call被响应（reply或reject）时，所有的可用cycle都退还给调用者，此时调用msg\_cycles\_available将返回 0。

代码如下：

|  |
| --- |
| pub fn msg\_cycles\_available() -> u64 {  unsafe { ic0::msg\_cycles\_available() as u64 }  } |

#### msg\_cycles\_accept函数

将cycle从当前调用移动至canister中去。 它移动尽可能多的cycle，直到这些限制：

* 移动的cycle数量不超过 max\_amount。
* 移动的cycle数量不会超过msg\_cycles\_available的值。
* 移动后的canister的cycle数量不能超过MAX\_CANISTER\_BALANCE减去任何可能的未清余额。

msg\_cycles\_accept可以被多次调用，每次都可能为余额增加更多的cycle。

返回值指示实际移动了多少个周期。

该函数调用不会产生trap。

代码如下：

|  |
| --- |
| pub fn msg\_cycles\_accept(max\_amount: u64) -> u64 {  unsafe { ic0::msg\_cycles\_accept(max\_amount as i64) as u64 }  } |

#### arg\_data函数

arg\_data用于获取当前调用的入参数据。

该函数会自动由导出宏生成调用，通常不需要手动调用。

代码如下：

|  |
| --- |
| pub fn arg\_data<R: for<'a> ArgumentDecoder<'a>>() -> R {  let bytes = unsafe { arg\_data\_raw() };  match decode\_args(&bytes) {  Err(e) => trap(&format!("{:?}", e)),  Ok(r) => r,  }  } |

#### reply函数

reply函数用于使用candid参数响应当前调用。

该函数会自动由导出宏生成调用，通常不需要手动调用。

代码如下：

|  |
| --- |
| pub fn reply<T: ArgumentEncoder>(reply: T) {  write\_args(&mut CallReplyWriter, reply).expect("Could not encode reply.");  unsafe {  ic0::msg\_reply();  }  } |

#### reject函数

reject函数使用一个消息拒绝当前调用。

该函数会自动由导出宏生成调用，通常不需要手动调用。

代码如下：

|  |
| --- |
| pub fn reject(message: &str) {  let err\_message = message.as\_bytes();  unsafe {  ic0::msg\_reject(err\_message.as\_ptr() as i32, err\_message.len() as i32);  }  } |

#### call\_raw函数

在处理更新调用时，一个canister可以进一步调用另一个canister。通过call\_raw函数，可以进行一次异步的canister间调用。如果需要同步调用，则使用call 函数或call\_with\_payment函数。

代码如下：

|  |
| --- |
| fn callback(state\_ptr: \*const InnerCell<CallFutureState<Vec<u8>>>) {  let state = unsafe { WasmCell::from\_raw(state\_ptr) };  // Make sure to un-borrow\_mut the state.  {  state.borrow\_mut().result = Some(match reject\_code() {  RejectionCode::NoError => unsafe { Ok(arg\_data\_raw()) },  n => Err((n, reject\_message())),  });  }  let w = state.borrow\_mut().waker.take();  if let Some(waker) = w {  // This is all to protect this little guy here which will call the poll() which  // borrow\_mut() the state as well. So we need to be careful to not double-borrow\_mut.  waker.wake()  }  }  pub fn call\_raw(  id: Principal,  method: &str,  args\_raw: Vec<u8>,  payment: u64,  ) -> impl Future<Output = CallResult<Vec<u8>>> {  let callee = id.as\_slice();  let state = WasmCell::new(CallFutureState {  result: None,  waker: None,  });  let state\_ptr = WasmCell::into\_raw(state.clone());  let err\_code = unsafe {  ic0::call\_new(  callee.as\_ptr() as i32,  callee.len() as i32,  method.as\_ptr() as i32,  method.len() as i32,  callback as usize as i32,  state\_ptr as i32,  callback as usize as i32,  state\_ptr as i32,  );  ic0::call\_data\_append(args\_raw.as\_ptr() as i32, args\_raw.len() as i32);  if payment > 0 {  ic0::call\_cycles\_add(payment as i64);  }  ic0::call\_perform()  };  // 0 is a special error code meaning call\_simple call succeeded.  if err\_code != 0 {  let mut state = state.borrow\_mut();  state.result = Some(Err((  RejectionCode::from(err\_code),  "Couldn't send message".to\_string(),  )));  }  CallFuture { state }  } |

#### call函数

通过call函数，可以进行一次同步的canister间调用。

代码如下：

|  |
| --- |
| pub async fn call<T: ArgumentEncoder, R: for<'a> ArgumentDecoder<'a>>(  id: Principal,  method: &str,  args: T,  ) -> CallResult<R> {  let args\_raw = encode\_args(args).expect("Failed to encode arguments.");  let bytes = call\_raw(id, method, args\_raw, 0).await?;  decode\_args(&bytes).map\_err(|err| trap(&format!("{:?}", err)))  } |

#### call\_with\_payment函数

通过call函数，可以进行一次同步的canister间调用并携带一定量的cycle。

代码如下：

|  |
| --- |
| pub async fn call\_with\_payment<T: ArgumentEncoder, R: for<'a> ArgumentDecoder<'a>>(  id: Principal,  method: &str,  args: T,  cycles: u64,  ) -> CallResult<R> {  let args\_raw = encode\_args(args).expect("Failed to encode arguments.");  let bytes = call\_raw(id, method, args\_raw, cycles).await?;  decode\_args(&bytes).map\_err(|err| trap(&format!("{:?}", err)))  } |

#### msg\_cycles\_refunded函数

当canister间调用返回后，通过msg\_cycles\_refunded函数可以获取退回的cycle数量。无论msg\_cycles\_refunded函数是否调用，退回的cycle都会自动进入当前canister。

代码如下：

|  |
| --- |
| pub fn msg\_cycles\_refunded() -> u64 {  unsafe { ic0::msg\_cycles\_refunded() as u64 }  } |

#### result函数

当canister间调用返回后，通过result函数可以获取调用的结果。如果调用成功，则为 Ok(T)，如果失败则为 reject\_message()。

代码如下：

|  |
| --- |
| pub fn result<T: for<'a> ArgumentDecoder<'a>>() -> Result<T, String> {  match reject\_code() {  RejectionCode::NoError => decode\_args(&unsafe { arg\_data\_raw() })  .map\_err(|e| format!("Failed to decode arguments: {}", e)),  \_ => Err(reject\_message()),  }  } |

#### reject\_code函数

当canister间调用返回后，通过reject\_code函数可以获取调用返回码。如果没有被reject，则值为RejectionCode::NoError。

通常不进行单独调用，而是使用封装的result函数。

代码如下：

|  |
| --- |
| pub fn reject\_code() -> RejectionCode {  let code = unsafe { ic0::msg\_reject\_code() };  RejectionCode::from(code)  } |

#### reject\_message函数

当canister间调用返回后，如果调用被reject，则可以通过reject\_message函数获取reject消息。

通常不进行单独调用，而是使用封装的result函数。

代码如下：

|  |
| --- |
| pub fn reject\_message() -> String {  let len: u32 = unsafe { ic0::msg\_reject\_msg\_size() as u32 };  let mut bytes = vec![0; len as usize];  unsafe {  ic0::msg\_reject\_msg\_copy(bytes.as\_mut\_ptr() as i32, 0, len as i32);  }  String::from\_utf8\_lossy(&bytes).to\_string()  } |

### caller函数

获取当前调用者的principal。

代码如下：

|  |
| --- |
| pub fn caller() -> Principal {  let len: u32 = unsafe { ic0::msg\_caller\_size() as u32 };  let mut bytes = vec![0; len as usize];  unsafe {  ic0::msg\_caller\_copy(bytes.as\_mut\_ptr() as i32, 0, len as i32);  }  Principal::try\_from(&bytes).unwrap()  } |

### canister\_balance函数

获取当前canister的可用cycle量。

代码如下：

|  |
| --- |
| pub fn canister\_balance() -> u64 {  unsafe { ic0::canister\_cycle\_balance() as u64 }  } |

### set\_certified\_data函数

Internet Computer 允许容器在更新方法处理期间存储少量数据，以便在查询调用处理期间，容器可以获得有关该数据的证书。

canister最多可以存储 32 个字节的数据，可以从查询调用中调用data\_certificate函数获取通过调用此函数设置的证书。

此函数只能从以下上下文中调用：

* "canister\_init", "canister\_pre\_upgrade" 和 "canister\_post\_upgrade" 钩子。
* "canister\_update" 调用。
* 回复或拒绝回调。

以下情况会产生陷阱：

* 如果 data.len() > 32，则此函数会产生陷阱。
* 如果从非法上下文（例如，从查询调用）调用此函数，则会陷入陷阱。

代码如下：

|  |
| --- |
| pub fn set\_certified\_data(data: &[u8]) {  unsafe { ic0::certified\_data\_set(data.as\_ptr() as i32, data.len() as i32) }  } |

### data\_certificate函数

Internet Computer 允许容器在更新方法处理期间存储少量数据，以便在查询调用处理期间，容器可以获得有关该数据的证书。

canister最多可以存储 32 个字节的数据，可以从查询调用中调用data\_certificate函数获取通过调用此函数设置的证书。

代码如下：

|  |
| --- |
| pub fn data\_certificate() -> Option<Vec<u8>> {  if unsafe { ic0::data\_certificate\_present() } == 0 {  return None;  }  let n = unsafe { ic0::data\_certificate\_size() };  let mut buf = vec![0u8; n as usize];  unsafe {  ic0::data\_certificate\_copy(buf.as\_mut\_ptr() as i32, 0i32, n);  }  Some(buf)  } |

### id函数

获取当前canister ID。

代码如下：

|  |
| --- |
| pub fn id() -> Principal {  let len: u32 = unsafe { ic0::canister\_self\_size() as u32 };  let mut bytes = vec![0; len as usize];  unsafe {  ic0::canister\_self\_copy(bytes.as\_mut\_ptr() as i32, 0, len as i32);  }  Principal::try\_from(&bytes).unwrap()  } |

### time函数

获取当前时间戳，纳秒精度。

代码如下：

|  |
| --- |
| pub fn time() -> u64 {  unsafe { ic0::time() as u64 }  } |

### trap函数

产生一个陷阱，使用指定的消息。

代码如下：

|  |
| --- |
| pub fn trap(message: &str) -> ! {  unsafe {  ic0::trap(message.as\_ptr() as i32, message.len() as i32);  }  unreachable!()  } |

### print函数

打印指定的消息。

代码如下：

|  |
| --- |
| pub fn print<S: std::convert::AsRef<str>>(s: S) {  let s = s.as\_ref();  unsafe {  ic0::debug\_print(s.as\_ptr() as i32, s.len() as i32);  }  } |

## export模块

export模块提供以下两个内容：

* Principal结构：principal描述身份的安全上下文，命名了可以与特定角色一起进行身份验证的任何身份。
* candid库：Candid 是一种接口描述语言 (IDL)，用于与运行在 Internet 计算机上的canister进行交互。

### Principal结构

Principal描述身份的安全上下文，命名了可以与特定角色一起进行身份验证的任何身份。

在 Internet 计算机中，这映射到可以由canister验证的身份。 例如，容器 ID 是 Principal，用户也是Principal。

Principal有以下关联函数：

* management\_canister() -> Principal：标记系统容器的空Principal。
* self\_authenticating<P: AsRef<[u8]>>(public\_key: P) -> Principal：强制使用Twisted Edwards Curve 25519 point作为公钥。
* anonymous() -> Principal：产生一个匿名Princial。
* from\_slice(bytes: &[u8]) -> Principal：将slice解析为Principal。如果无法解析，则触发陷阱。
* try\_from\_slice(bytes: &[u8]) -> Result<Principal, PrincipalError>：将slice解析为Principal。不会触发陷阱。
* from\_text<S: AsRef<str>>(text: S) -> Result<Principal, PrincipalError>：将数据解析为Principal。不会触发陷阱。

Principal有以下方法：

* to\_text(&self) -> String：返回此 Principal 的文本表示。
* as\_slice(&self) -> &[u8]：返回此 Principal 的切片表示。

Principal同时实现了一些trait。具体见[这里](https://docs.rs/ic-cdk/0.3.1/ic_cdk/export/struct.Principal.html#method.anonymous)。

### candid库

Candid 是一种接口描述语言 (IDL)，用于与运行在 Internet 计算机上的canister进行交互。

在 Rust 中使用 Candid 的三种常见方式。

* 作为类型化的 Rust 值：当在 Rust 中编写canister或前端时，可以在 Rust 和 Candid 之间无缝转换数据。
* 作为无类型的 Candid 值：当只知道Candid 数据值，而没有对应Rust类型的情况下为 Internet 计算机编写通用工具时使用这种方式。
* 作为文本数据流：当从 CLI 获取数据或从文件中读取数据时，可以使用提供的解析器来发送/接收消息。

Candid 提供了在这些表示之间转换数据的高效、灵活和安全的方法。

#### 类型化的Rust值处理

可以使用以下三种方式将类型化的rust数据序列化为candid消息以及将candid消息反序列化为类型化的rust数据。

* IDLBuilder/IDLDeserialize模块
* encode\_args/decode\_args函数
* Encode/Decode宏

##### IDLBuilder/IDLDeserialize模块

例子：

|  |
| --- |
| // Serialize 10 numbers to Candid binary format  let mut ser = candid::ser::IDLBuilder::new();  for i in 0..10 {  ser.arg(&i)?;  }  let bytes: Vec<u8> = ser.serialize\_to\_vec()?;  // Deserialize Candid message and verify the values match  let mut de = candid::de::IDLDeserialize::new(&bytes)?;  let mut i = 0;  while !de.is\_done() {  let x = de.get\_value::<i32>()?;  assert\_eq!(x, i);  i += 1;  }  de.done()?; |

##### encode\_args/decode\_args函数

例子：

|  |
| --- |
| use candid::{encode\_args, decode\_args};  // Serialize two values [(42, "text")] and (42u32, "text")  let bytes: Vec<u8> = encode\_args((&[(42, "text")], &(42u32, "text")))?;  // Deserialize the first value as type Vec<(i32, &str)>,  // and the second value as type (u32, String)  let (a, b): (Vec<(i32, &str)>, (u32, String)) = decode\_args(&bytes)?;  assert\_eq!(a, [(42, "text")]);  assert\_eq!(b, (42u32, "text".to\_string())); |

##### Encode/Decode宏

例子：

|  |
| --- |
| use candid::{Encode, Decode};  // Serialize two values [(42, "text")] and (42u32, "text")  let bytes: Vec<u8> = Encode!(&[(42, "text")], &(42u32, "text"))?;  // Deserialize the first value as type Vec<(i32, &str)>,  // and the second value as type (u32, String)  let (a, b) = Decode!(&bytes, Vec<(i32, &str)>, (u32, String))?;  assert\_eq!(a, [(42, "text")]);  assert\_eq!(b, (42u32, "text".to\_string())); |

#### 无类型的Candid值处理

任何有效的 Candid 值都可以由递归枚举candid::parser::value::IDLValue 中进行表示。

数据结构 candid::IDLArgs 可以来表示一个 IDLValues 序列。

##### value\_arg/get\_value

可以使用 ser.value\_arg(v) 将Candid值编码为Candid消息以及使用 de.get\_value::<IDLValue>() 将Candid消息解码为Candid值。通过这种方式可以将Rust 值和 IDLValue值混合使用。

例子：

|  |
| --- |
| use candid::parser::value::IDLValue;  // Serialize Rust value Some(42u8) and IDLValue "hello"  let bytes = candid::ser::IDLBuilder::new()  .arg(&Some(42u8))?  .value\_arg(&IDLValue::Text("hello".to\_string()))?  .serialize\_to\_vec()?;  // Deserialize the first Rust value into IDLValue,  // and the second IDLValue into Rust value  let mut de = candid::de::IDLDeserialize::new(&bytes)?;  let x = de.get\_value::<IDLValue>()?;  let y = de.get\_value::<&str>()?;  de.done()?;  assert\_eq!(x, IDLValue::Opt(Box::new(IDLValue::Nat8(42))));  assert\_eq!(y, "hello"); |

##### to\_bytes/from\_bytes

对于candid::IDLArgs表示的IDLValues 序列，可以使用其 to\_bytes() 和 from\_bytes() 方法来进行编码和解码 Candid 消息。

例子：

|  |
| --- |
| use candid::{IDLArgs, TypeEnv};  // Candid values represented in text format  let text\_value = r#"  (42, opt true, vec {1;2;3},  opt record {label="text"; 42="haha"})  "#;  // Parse text format into IDLArgs for serialization  let args: IDLArgs = text\_value.parse()?;  let encoded: Vec<u8> = args.to\_bytes()?;  // Deserialize into IDLArgs  let decoded: IDLArgs = IDLArgs::from\_bytes(&encoded)?;  assert\_eq!(encoded, decoded.to\_bytes()?);  // Convert IDLArgs to text format  let output: String = decoded.to\_string();  let parsed\_args: IDLArgs = output.parse()?;  let annotated\_args = args.annotate\_types(true, &TypeEnv::new(), &parsed\_args.get\_types())?;  assert\_eq!(annotated\_args, parsed\_args); |

#### 文本数据流处理

candid::parser提供了一个解析器来解析文本格式的 Candid 值。candid为IDLArgs实现了std::str::FromStr trait。代码如下：

|  |
| --- |
| impl std::str::FromStr for IDLArgs {  type Err = Error;  fn from\_str(str: &str) -> std::result::Result<Self, Self::Err> {  let lexer = super::token::Tokenizer::new(str);  Ok(super::grammar::ArgsParser::new().parse(lexer)?)  }  } |

例子：

|  |
| --- |
| use candid::{IDLArgs, TypeEnv};  // Candid values represented in text format  let text\_value = r#"  (42, opt true, vec {1;2;3},  opt record {label="text"; 42="haha"})  "#;  // Parse text format into IDLArgs for serialization  let args: IDLArgs = text\_value.parse()?;  let encoded: Vec<u8> = args.to\_bytes()?; |

注意：在从文件数据流解析 Candid 值时，会假设数字值始终是 Int 类型，因此得到的都是IDLValue::Int的Candid值。

##### did文件解析

在解析文本数据流时，很多时候需要配合did文件进行处理。candid提供了IDLProg和TypeEnv来展示did文件内容。

candid提供了一个解析器来解析did文件。candid为IDLProg实现了std::str::FromStr trait。代码如下：

|  |
| --- |
| impl std::str::FromStr for IDLProg {  type Err = crate::Error;  fn from\_str(str: &str) -> Result<Self> {  let lexer = super::token::Tokenizer::new(str);  Ok(super::grammar::IDLProgParser::new().parse(lexer)?)  }  } |

例子：

|  |
| --- |
| use candid::{IDLProg, TypeEnv, check\_prog, types::Type};  let did\_file = r#"  type List = opt record { head: int; tail: List };  type byte = nat8;  service : {  f : (byte, int, nat, int8) -> (List);  g : (List) -> (int) query;  }  "#;  // Parse did file into an AST  let ast: IDLProg = did\_file.parse()?;  // Pretty-print AST  let pretty: String = candid::parser::types::to\_pretty(&ast, 80);  // Type checking  let mut env = TypeEnv::new();  let actor: Type = check\_prog(&mut env, &ast)?.unwrap();  let method = env.get\_method(&actor, "g").unwrap();  assert\_eq!(method.is\_query(), true);  assert\_eq!(method.args, vec![Type::Var("List".to\_string())]); |

##### 依据did文件序列化数据

candid::IDLArgs在使用 to\_bytes\_with\_types 函数进行序列化时，可以使用来自 Candid 文件的类型签名，来序列化Candid消息。通过这种方式序列化的Candid消息符合Candid 文件的类型签名，而不是原来candid::IDLArgs中的Candid值类型签名。

例子：

|  |
| --- |
| use candid::{IDLArgs, parser::value::IDLValue};  // Get method type f : (byte, int, nat, int8) -> (List)  let method = env.get\_method(&actor, "f").unwrap();  let args = "(42, 42, 42, 42)".parse::<IDLArgs>()?;  // Serialize arguments with candid types  let encoded = args.to\_bytes\_with\_types(&env, &method.args)?;  let decoded = IDLArgs::from\_bytes(&encoded)?;  assert\_eq!(decoded.args,  vec![IDLValue::Nat8(42),  IDLValue::Int(42.into()),  IDLValue::Nat(42.into()),  IDLValue::Int8(42)  ]); |

#### 自定义struct/enum

只有实现了CandidType trait的类型才可以进行序列化，只有实现了CandidType 和 Deserialize trait的类型才可以进行反序列化。内置的 Rust 标准库类型，如 Vec<T> 和 Result<T, E>，以及任何用 #[derive(CandidType, Deserialize)] 注释的结构或枚举都会实现CandidType 和 Deserialize trait。

除了序列化之外，CandidType trait 还定义了将当前Rust类型转换为何种Candid 类型(candid::types::Type)。

可以使用 serde rename属性对每个字段进行重命名，即 #[serde(rename = "foo")] 和 #[serde(rename(serialize = "foo", deserialize = "foo"))]。 这在涉及变体类型的 Rust 和 Motoko 容器之间进行互操作时很有用，因为它们对字段名称使用不同的命名约定。

可以#[serde(with = "serde\_bytes")]以高效处理&[u8]和Vec<u8>。还可以使用包装器类型 serde\_bytes::ByteBuf 和 serde\_bytes::Bytes。

例子：

|  |
| --- |
| use candid::{Encode, Decode, CandidType, Deserialize};  #[derive(CandidType, Deserialize)]  enum List {  #[serde(rename = "nil")]  Nil,  #[serde(with = "serde\_bytes")]  Node(Vec<u8>),  Cons(i32, Box<List>),  }  let list = List::Cons(42, Box::new(List::Nil));  let bytes = Encode!(&list)?;  let res = Decode!(&bytes, List)?;  assert\_eq!(res, list); |

#### Big Integers

为了支持大整数类型 Candid::Int 和 Candid::Nat，candid使用了 num\_bigint crate。 提供了将 i64、u64、&str 和 &[u8] 转换为大整数的接口。 也可以用 i128 和 u128 分别表示 Candid int 和 nat 类型（但是超过 128 位会解码失败）。

例子：

|  |
| --- |
| use candid::{Int, Nat, Encode, Decode};  let x = "-10000000000000000000".parse::<Int>()?;  let bytes = Encode!(&Nat::from(1024), &x)?;  let (a, b) = Decode!(&bytes, Nat, Int)?;  let (c, d) = Decode!(&bytes, u128, i128)?;  assert\_eq!(a + 1, 1025);  assert\_eq!(b, Int::parse(b"-10000000000000000000")?); |

* candid::ser::encode\_args函数可以将rust语言数据根据自断推断的接口描述数据类型，生成可以在网络上传输的数据流。
* candid::de::decode\_args函数可以将在网络传输的数据流转换为rust语言数据。
* candid::Encode宏可以将 Rust 值序列编码为 candid::Result<Vec<u8>> 类型的 Candid 消息。
* candid::Decode宏可以将Candid 消息解码为给定类型的 Rust 值元组。 如果消息无法在任何给定类型解码，则产生 Err。 如果消息只包含一个值，则直接返回该值而不是元组。

## storage模块

storage模块提供了stable\_save和stable\_restore两个函数用于持久化内存。使用持久化内存流程如下：

* 在#[pre\_upgrade]函数中通过stable\_save进行数据持久化。注意：stable\_save只能调用一次，否则会覆盖前一次存储的内容。
* 在#[post\_upgrade]函数中通过stable\_restore将持久化数据反序列化。

注意：通过stable\_save序列化的类型必须实现了candid::utils::ArgumentEncoder。

storage模块另外提供了delete、get和get\_mut三个函数简化用于持久化的静态项创建的过程。使用过程如下：

* 在#[init]函数中通过get\_mut获取值的静态可变引用。
* 在#[pre\_upgrade]函数中通过get或get\_mut获取值的不可变/可变引用，并和其他静态项组成tuple，调用stable\_save进行持久化。
* 在#[post\_upgrade]函数中调用stable\_restore进行反序列化得到一个元组，通过get\_mut获取的静态可变引用再用元组元素进行赋值。

例子：

|  |
| --- |
| use ic\_cdk::{storage, export::Principal};  use ic\_cdk\_macros::\*;  use std::collections::{BTreeMap, BTreeSet};  type Users = BTreeSet<Principal>;  type Store = BTreeMap<String, Vec<u8>>;  #[init]  fn init() {  let users = storage::get\_mut::<Users>();  users.insert(ic\_cdk::api::caller());  }  fn is\_user() -> Result<(), String> {  let users = storage::get::<Users>();  if users.contains(&ic\_cdk::api::caller()) {  Ok(())  } else {  Err("Store can only be set by the owner of the asset canister.".to\_string())  }  }  #[update(guard = "is\_user")]  fn store(path: String, contents: Vec<u8>) {  let store = storage::get\_mut::<Store>();  store.insert(path, contents);  }  #[query]  fn retrieve(path: String) -> &'static Vec<u8> {  let store = storage::get::<Store>();  match store.get(&path) {  Some(content) => content,  None => panic!("Path {} not found.", path),  }  }  #[update(guard = "is\_user")]  fn add\_user(principal: Principal) {  let users = storage::get\_mut::<Users>();  users.insert(principal);  }  #[pre\_upgrade]  fn pre\_upgrade() {  let mut vec = Vec::new();  for p in storage::get\_mut::<Users>().iter() {  vec.push(p);  }  storage::stable\_save((vec,)).unwrap();  }  #[post\_upgrade]  fn post\_upgrade() {  let (old\_users,): (Vec<Principal>,) = storage::stable\_restore().unwrap();  for u in old\_users {  storage::get\_mut::<Users>().insert(u);  }  } |

### get\_mut函数

用于获取一个静态项的可变引用，该静态项在全局静态B树中，该B树以类型IP为键，因此不同的静态项，他们的类型一定要不同。

核心代码如下：

|  |
| --- |
| type StorageTree = BTreeMap<TypeId, Box<dyn Any>>;  static mut STORAGE: Option<StorageTree> = None;  fn storage() -> &'static mut StorageTree {  unsafe {  if let Some(s) = &mut STORAGE {  s  } else {  STORAGE = Some(BTreeMap::new());  storage()  }  }  }  pub fn get\_mut<T: Sized + Default + 'static>() -> &'static mut T {  let type\_id = std::any::TypeId::of::<T>();  let store = storage();  store  .entry(type\_id)  .or\_insert\_with(|| Box::new(T::default()))  .downcast\_mut()  .expect("Unexpected value of invalid type.")  } |

### get函数

获取一个静态项的不可变引用。

其代码如下：

|  |
| --- |
| pub fn get<T: Sized + Default + 'static>() -> &'static T {  get\_mut::<T>()  } |

### delete函数

删除静态项内容。

其代码如下：

|  |
| --- |
| pub fn delete<T: Sized + Default + 'static>() -> bool {  let type\_id = std::any::TypeId::of::<T>();  storage().remove(&type\_id).is\_some()  } |

### stable\_save函数

将指定数据进行序列化持久保持。

注意：

* 数据类型必须实现了candid::utils::ArgumentEncoder trait。
* 如果进行了多次持久化，那么只会保留最后一次结果。

代码如下：

|  |
| --- |
| pub fn stable\_save<T>(t: T) -> Result<(), candid::Error>  where  T: candid::utils::ArgumentEncoder,  {  candid::write\_args(&mut stable::StableWriter::default(), t)  } |

### stable\_restore函数

将序列化的持久数据进行反序列化到内存中。

代码如下：

|  |
| --- |
| pub fn stable\_restore<T>() -> Result<T, String>  where  T: for<'de> candid::utils::ArgumentDecoder<'de>,  {  let bytes = stable::stable\_bytes();  let mut de =  candid::de::IDLDeserialize::new(bytes.as\_slice()).map\_err(|e| format!("{:?}", e))?;  let res = candid::utils::ArgumentDecoder::decode(&mut de).map\_err(|e| format!("{:?}", e))?;  // The idea here is to ignore an error that comes from Candid, because we have trailing  // bytes.  let \_ = de.done();  Ok(res)  } |

# ic-types

ic-types提供了与 Internet 计算机协议相关的类型集合。

其提供了以下三个项：

* Principal结构体：与ic\_cd::Principal一样。
* PrincipalError枚举：principal相关错误。
* HashTree结构体：表示完整树的 HashTree。

# ic-agent

ic-agent 是一个简单易用的库，能够在 Rust 中构建应用程序并与 Internet 计算机交互。它是DFINITY Canister 软件开发套件(SDK) 和 Canister SDK 命令行执行环境dfx的底层后端。

ic-agent旨在与副本 API 的多个版本兼容，并暴露出用于与互联网计算机协议组件（如副本）通信的底层 API，并提供用于与canister通信的高级 API。

其提供了以下几个项：

* agent模块：主要agent模块。包含agent类型和所有关联的结构。
* identity模块：处理跨 Internet 计算机身份的类型和trait。
* request\_id模块：该模块处理根据消息内容计算请求 ID。
* export模块：里面只有一个Principal。

# ic-utils

ic-utils 是一组实用程序，可帮助构建在 Internet 计算机上运行的客户端和容器。 它是一种更高级别的工具。

ic-utils提供了如下项：

* call模块：封装了对canister的调用的实用工具。
* canister模块：封装了canister操作的使用工具。
* interfaces模块：封装了三种内置canister类型的接口。

# Canister管理

IC提供了一个虚拟的canister进行所有的canister管理，称它为canister manager，这个canister的ID为"aaaaa-aa"。IC canister manager实际上并不作为容器存在（具有隔离状态、Wasm 代码等）。

canister manager的did文件如下描述：

|  |
| --- |
| type canister\_id = principal;  type user\_id = principal;  type wasm\_module = blob;  type canister\_settings = record {  controllers : opt vec principal;  compute\_allocation : opt nat;  memory\_allocation : opt nat;  freezing\_threshold : opt nat;  };  type definite\_canister\_settings = record {  controllers : vec principal;  compute\_allocation : nat;  memory\_allocation : nat;  freezing\_threshold : nat;  };  service ic : {  create\_canister : (record {  settings : opt canister\_settings  }) -> (record {canister\_id : canister\_id});  update\_settings : (record {  canister\_id : principal;  settings : canister\_settings  }) -> ();  install\_code : (record {  mode : variant {install; reinstall; upgrade};  canister\_id : canister\_id;  wasm\_module : wasm\_module;  arg : blob;  }) -> ();  uninstall\_code : (record {canister\_id : canister\_id}) -> ();  start\_canister : (record {canister\_id : canister\_id}) -> ();  stop\_canister : (record {canister\_id : canister\_id}) -> ();  canister\_status : (record {canister\_id : canister\_id}) -> (record {  status : variant { running; stopping; stopped };  settings: definite\_canister\_settings;  module\_hash: opt blob;  memory\_size: nat;  cycles: nat;  });  delete\_canister : (record {canister\_id : canister\_id}) -> ();  deposit\_cycles : (record {canister\_id : canister\_id}) -> ();  raw\_rand : () -> (blob);  // provisional interfaces for the pre-ledger world  provisional\_create\_canister\_with\_cycles : (record {  amount: opt nat;  settings : opt canister\_settings  }) -> (record {canister\_id : canister\_id});  provisional\_top\_up\_canister :  (record { canister\_id: canister\_id; amount: nat }) -> ();  } |

## create\_canister

create\_canister : (record {settings : opt canister\_settings}) -> (record {canister\_id : canister\_id});

在部署一个容器之前，容器的管理员首先要在系统中注册它，得到一个容器ID（是一个空容器），然后单独安装代码。

可选settings参数可用于进行以下设置：

* controllers (vec principal)：principal列表。大小必须在 0 到 10 之间。默认值：仅包含调用create\_canister的调用者。该值分配给容器的控制器属性。
* compute\_allocation (nat)：必须是 0 到 100 之间的数字，包括 0 和 100，默认值为0。它指示应该为该容器保证多少计算能力，表示为单个容器可以分配的最大计算能力的百分比。如果系统无法提供请求的分配，例如因为它被超额预订，调用将被拒绝。
* memory\_allocation (nat)：必须是介于 0 和 2^48（即 256TB）之间的数字，包括在内，默认值为0。它指示容器总共允许使用多少内存。 任何超出此分配增加内存使用量的尝试都将失败。 如果系统无法提供请求的分配，例如因为它被超额预订，呼叫将被拒绝。 如果设置为 0，则容器的内存增长将尽最大努力并受网络上可用内存的限制。
* freezing\_threshold (nat)：必须是 0 到 2^64-1（含）之间的数字，并表示时间长度（以秒为单位），默认值：2592000（大约 30 天）。考虑容器的当前大小和系统的当前存储成本，当系统估计容器将在经过 freeze\_threshold 秒之后耗尽cycle，就会将容器视为已冻结。

注意：执行create\_canister时需要额外添加cycle用于注入到新canister中。

## update\_settings

update\_settings : (record {canister\_id : principal; settings : canister\_settings}) -> ();

只有canister的controllers可以更新设置。

* canister\_id指定需要更新设置的canister的id。
* settings同create\_canister中的settings，如果在settings中不包括某个字段，则意味着不更改该字段。

## install\_code

install\_code : (record {mode : variant {install; reinstall; upgrade}; canister\_id : canister\_id;

wasm\_module : wasm\_module; arg : blob;}) -> ();

此方法将代码安装到容器中。只有容器的 controllers 才能安装代码。

对于不同的mode，情况所有不同：

* 如果 mode = install，则容器之前必须是空的。这将实例化容器模块并调用其 canister\_init 系统方法（如果存在），并将 arg 传递给该方法。
* 如果mode = reinstall，如果容器不为空，则在进行mode = install之前删除其现有代码和状态。请注意，这与后跟 install\_code 的uninstall\_code 不同，因为这将强制拒绝所有未响应的调用。
* 如果 mode = upgrade，这将执行非空容器的升级，会将 arg 传递给新实例的 canister\_post\_upgrade 系统方法。

注意：如果对此请求的响应是reject，则此调用无效。

## uninstall\_code

uninstall\_code : (record {canister\_id : canister\_id}) -> ();

此方法删除容器的代码和状态，使容器再次清空。只有容器的 controllers 才能卸载代码。

卸载将reject容器尚未响应的所有调用，并删除容器的代码和状态。不会处理对容器的未完成响应，即使它们在再次安装代码后到达。

canister现在是空的。 特别是，任何传入或排队的调用都将被拒绝。

卸载后的容器保留其cycle数量、controllers,、status和allocations。

## canister\_status

canister\_status : (record {canister\_id : canister\_id}) -> (record {

status : variant { running; stopping; stopped }; settings: definite\_canister\_settings;

module\_hash: opt blob; memory\_size: nat; cycles: nat;});

指示有关canister的各种信息。只有容器的controller可以请求其状态。它包含了：

* status。 它可以是运行、停止或停止之一。
* SHA256 哈希值。安装在容器上的模块的 SHA256 哈希值。 如果容器为空，则为 null。
* controller。控制器列表
* allocations。占用的内存大小。
* cycle数量。

## stop\_canister

stop\_canister : (record {canister\_id : canister\_id}) -> ();

容器的controller可以停止容器（例如，为容器升级做准备）。

停止canister不是原子操作。 直接效果是容器的状态更改为正在停止（除非容器已停止）。 系统将reject对正在停止的容器的所有调用，表明容器正在停止。 对停止canister的响应照常处理。 处理完所有未完成的响应后（因此没有打开的调用上下文），容器状态更改为停止，并且管理容器响应 stop\_canister 请求的调用者。

## start\_canister

start\_canister : (record {canister\_id : canister\_id}) -> ();

容器可以由其controller启动。

如果容器状态已停止或正在停止，则容器状态仅设置为运行。 在后一种情况下，所有正在处理的 stop\_canister 调用都失败（并被拒绝）。

如果容器已在运行，则状态保持不变。

## delete\_canister

delete\_canister : (record {canister\_id : canister\_id}) -> ();

此方法从 IC 中删除一个canister。只有容器的 controllers 可以删除它，并且容器必须已经停止。

删除容器无法撤消，存储在容器上的任何状态都将被永久删除并丢弃其cycle。容器一旦被删除，其 ID 就不能再使用。

## deposit\_cycles

deposit\_cycles : (record {canister\_id : canister\_id}) -> ();

此方法将包含在此调用中的cycle存放到指定的容器中。

对谁可以调用此方法，没有controller限制。

## raw\_rand

raw\_rand : () -> (blob);

此方法不接受任何输入并向调用者返回 32 个伪随机字节。 在提交此调用时，IC 的任何部分都不知道返回值。 每次调用此方法都会生成一个新的返回值。