* core components：
* User authority management
* Learning assessment
* Data Entry
* Data analysis
* architecture：

Cargo.lock & Cargo.toml

Cargo is rust's package manager, equivalent to nodejs' npm or yarn, but cargo has more functions and also acts as a rust code organization management tool. Cargo provides a series of projects from project establishment, construction to testing, operation to deployment Tools to provide as complete a means as possible for the management of the rust project.

Cargo.lock contains the exact information of dependencies, which is automatically generated by Cargo without manual editing, while Cargo.toml requires manual configuration of dependencies.

Cargo.toml stores project information [package] and dependent libraries [dependencies], etc., which is equivalent to a guide for cargo building projects.

The contents of Cargo.toml in the root directory are as follows：

[profile.release]  
panic = 'unwind'  
[workspace]  
members = ['node',‘pallets/template', 'runtime',]

substrate-node-template is a Rust workspace project that can clearly manage component libraries (library) and executable programs (binary).。

The members of this [workspace] have：

* node：Executable program, there is an executable main function entry in node/src/main.rs
* pallets/template：Module code, functions and data structures that can be called externally are defined in pallets/template/src/lib.rs
* runtime：Component library, the runtime logic is defined in runtime/src/lib.rs

[profile.release] configured panic='unwind' means to use catch\_unwind together to capture exceptions thrown by panic in a certain thread.

scripts directory

The scripts directory contains two shell scripts:

* docker\_run.sh：Use Docker to start the script of substrate-node-template;
* init.sh：Script to initialize the WASM build environment;
* The content of the init.sh script includes upgrading the Rust version：

rustup update nightly  
  
rustup update stable

And add to build the WebAssembly toolchain：

rustup target add wasm32-unknown-unknown *--toolchain nightly*

node directory

The node directory contains the following files：

* build.rs：Custom build script
* Cargo.toml：Node package building guide
* src/chain\_spec.rs：Construct ChainSpec (chain specification file)
* src/cli.rs：Declare client structure and subcommands
* src/command.rs：Provide implementation functions for client-side related commands
* src/lib.rs：Import library modules
* src/main.rs：Substrate-node-template is compiled into the entry file of the executable program
* src/rpc.rs：The collection of RPC methods specified by the node；
* src/service.rs：Provide tools and methods for constructing Substrate services

1、Cargo.toml is the node package construction guide, use [[bin]] to indicate that this package is executable：

[[bin]]  
  
name = 'node-template'

Introduce compile-time dependencies through [build-dependencies] and use them in build.rs：

[build-dependencies]  
  
substrate-build-script-utils = '2.0.0'  
  
Other information includes [package], [package.metadata.docs.rs], [dependencies], [features], etc., which are the same as general Cargo.toml.

2、build.rs is a custom build script, the content is as follows：

use substrate\_build\_script\_utils::{generate\_cargo\_keys, rerun\_if\_git\_head\_changed};  
  
fn main() {  
  
generate\_cargo\_keys();  
  
rerun\_if\_git\_head\_changed();  
  
}

The function is to let Cargo compile and execute the script.

3、src/main.rs is the entry file that substrate-node-template is compiled into the correct program. The content is as follows：

#![warn(missing\_docs)]  
  
mod chain\_spec;  
  
#[macro\_use]  
  
mod service;  
  
mod cli;

mod command;  
  
mod rpc;  
  
fn main() -> sc\_cli::Result<()> {  
  
command::run()  
  
}

The #![warn(missing\_docs)] annotation means that when compiling, if the module lacks documentation, a warning message will be printed.

mod chain\_spec, mod service, mod cli, mod command, mod rpc introduce other modules in the current directory.

#[macro\_use]Load all the macros under the imported module.

The main() function is the entry point of the application, and the returned sc\_cli::Result<()> is a custom Result type:

pub type Result<T> = std::result::Result<T, Error>;

The main() function internally executes the run() function provided by the command module.

4. src/command.rs provides the implementation functions of client-related commands, creates the implementation of Cli, SubstrateCli, and defines the run() function.

The run() function first parses the command line parameters through from\_args() and returns a Cli structure, which is defined in cli.rs, and then matches the subcommand in the parameter, and executes it if there is a subcommand.

When executing a subcommand, first call cli.create\_runner(cmd)? to create a runner, and then call async\_run() to execute the subcommand asynchronously.

If there is no subcommand in the command line parameters, call run\_node\_until\_exit() to start the node.

5. src/cli.rs declares the client structure and subcommands.

The Cli structure declaration is as follows:

#[derive(Debug, StructOpt)]  
  
pub struct Cli {  
  
 #[structopt(subcommand)]  
  
 pub subcommand: Option<Subcommand>,  
  
 #[structopt(flatten)]  
  
 pub run: RunCmd,  
  
}  
  
Contains optional sub-commands and command-line options, the compiled substrate-node-template can be passed

./target/release/node-template -h

Get instructions for the available subcommands and options.

Specific subcommands use enumeration declarations:

#[derive(Debug, StructOpt)]  
  
pub enum Subcommand {  
  
BuildSpec(sc\_cli::BuildSpecCmd),  
  
CheckBlock(sc\_cli::CheckBlockCmd),  
  
ExportBlocks(sc\_cli::ExportBlocksCmd),  
  
ExportState(sc\_cli::ExportStateCmd),  
  
ImportBlocks(sc\_cli::ImportBlocksCmd),  
  
PurgeChain(sc\_cli::PurgeChainCmd),  
  
Revert(sc\_cli::RevertCmd),  
  
#[structopt(name = "benchmark", about = "Benchmark runtime pallets.")]  
  
Benchmark(frame\_benchmarking\_cli::BenchmarkCmd),  
  
}

6、src/chain\_spec.rs constructs ChainSpec (chain specification file), ChainSpec defines the available configuration of the chain, which is used to construct the initial block.

Two functions are defined in src/chain\_spec.rs:

* pub fn development\_config() -> Result<ChainSpec, String>
* pub fn local\_testnet\_config() -> Result<ChainSpec, String>

Represents the two modes provided by substrate-node-template:

* The development network (development) specified by the command line option --dev has only one validator Alice;
* Local test network (local\_testnet), there are two validators Alice and Bob;Next, call ChainSpec::from\_genesis to create a chain specification file.

7、src/service.rs provides tools and methods for constructing Substrate services.

src/service.rs first uses the native\_executor\_instance! macro to define a structure Executor, and implements the NativeExecutionDispatch interface, that is, the function can be called by the function name。

The tools and methods of src/service.rs include：

* new\_partial：Build a local node service；
* new\_full：Build a full node service；
* new\_light：Build a light node service；

8、src/rpc.rs provides a collection of RPC methods specified by the node.

rpc.rs provides the create\_full() method to instantiate all complete RPC extensions.

9、src/lib.rs is used to import library modules, the content is as follows:

pub mod chain\_spec;  
  
pub mod service;  
  
pub mod rpc;

runtime directory

The runtime directory contains the following files:

* build.rs：Custom build script;
* Cargo.toml：runtime package construction guide;
* src/lib.rs：Runtime entry file on the chain;

1、Cargo.toml is a build guide for the runtime package. In addition to common configuration items, there are[build-dependencies]：

[build-dependencies]  
  
wasm-builder-runner = { package = 'substrate-wasm-builder-runner', version = '2.0.0' }

Added wasm-builder-runner, which the build script build.rs depends on.

2、build.rs is a custom build script, the content is as follows:

use wasm\_builder\_runner::WasmBuilder;  
  
fn main() {  
  
 WasmBuilder::new()  
  
 .with\_current\_project()  
  
 .with\_wasm\_builder\_from\_crates("2.0.0")  
  
 .export\_heap\_base()  
  
 .import\_memory()  
  
 .build()  
 }

Use wasm-builder-runner to compile the current runtime project into a Wasm binary file located at target/release/wbuild/node-template-runtime/node\_template\_runtime.compact.wasm.

3、src/lib.rs is the runtime entry file on the chain.

#![cfg\_attr(not(feature = "std"), no\_std)] means that if the feature is not std (Rust standard library) when compiling, then it must be no\_std (compiling to Wasm).

#![recursion\_limit="256"]Set the maximum number of infinite recursive operations that may occur during compilation.

The code below：

#[cfg(feature = "std")]  
  
include!(concat!(env!("OUT\_DIR"), "/wasm\_binary.rs"));

Indicates that when using the Rust standard library to compile, the generated Wasm binary content is introduced into the current runtime code by means of constants.

Introduce dependent modules and template modules:

pub use pallet\_template;

Next are the aliases of some basic types required by runtime, which are consistent with the related type names in the module.

The opaque module encapsulates some types used for CLI initialization.

Define constants related to block time:

pub const MILLISECS\_PER\_BLOCK: u64 = 6000;

That is, the production time of each block is 6 seconds, and the configuration can be modified as needed.

Next, use the parameter\_types! macro to generate data types that meet the Get interface required by some functional modules.

Then implement the interface of each functional module for runtime：

impl frame\_system::Trait for Runtime {...}  
  
impl pallet\_aura::Trait for Runtime {...}  
  
impl pallet\_grandpa::Trait for Runtime {...}  
  
impl pallet\_timestamp::Trait for Runtime {...}  
  
impl pallet\_balances::Trait for Runtime {...}  
  
impl pallet\_transaction\_payment::Trait for Runtime {...}  
  
impl pallet\_sudo::Trait for Runtime {...}  
  
impl pallet\_template::Trait for Runtime {...}  
  
The runtime is constructed by the construct\_runtime! macro：

construct\_runtime!(  
  
 pub enum Runtime where  
  
 Block = Block,  
  
 NodeBlock = opaque::Block,  
  
 UncheckedExtrinsic = UncheckedExtrinsic {

System: frame\_system::{Module, Call, Config, Storage, Event<T>},  
  
 RandomnessCollectiveFlip: pallet\_randomness\_collective\_flip::{Module, Call, Storage},  
  
 Timestamp: pallet\_timestamp::{Module, Call, Storage, Inherent},  
  
 Aura: pallet\_aura::{Module, Config<T>, Inherent},  
  
 Grandpa: pallet\_grandpa::{Module, Call, Storage, Config, Event},  
  
 Balances: pallet\_balances::{Module, Call, Storage, Config<T>, Event<T>},  
  
 TransactionPayment: pallet\_transaction\_payment::{Module, Storage},  
  
 Sudo: pallet\_sudo::{Module, Call, Config<T>, Storage, Event<T>},  
  
 // Include the custom logic from the template pallet in the runtime.  
  
 TemplateModule: pallet\_template::{Module, Call, Storage, Event<T>},

}  
  
);

The construct\_runtime! macro constructs the runtime according to the module name and the components in the used module. The initial storage is loaded in order during construction. Therefore, when module B depends on module A, module A should be placed before module B.

Finally, use the impl\_runtime\_apis! macro to implement the interfaces defined by the runtime api, and these interfaces are defined by the decl\_runtime\_apis! macro.

pallets/template directory

The pallets directory can contain multiple pallets (modules), and template is a pallet.

The pallets/template directory contains the following files:

* Cargo.toml：Template module building guide；
* src/lib.rs：The specific function implementation code of the module；
* src/mock.rs：Test case service code；
* src/tests.rs：Test case；

1. Cargo.toml is the construction guide for the template module, which mainly configures [dependencies] and [features] according to requirements.

[dependencies] is the dependency library of the module. [features] uses std feature by default to ensure that the runtime can be compiled to either the native version (using std feature) or the wasm version (using no\_std feature).

2. src/lib.rs is the specific function implementation code of the module.

Mock and tests are only compiled when running tests.

Then type declaration：

pub trait Trait: frame\_system::Trait {  
  
 type Event: From<Event<Self>> + Into<<Self as frame\_system::Trait>::Event>;  
  
}

And four macros related to business logic code：

* decl\_storage!：Define storage；
* decl\_event!：Define event；
* decl\_error!：Define error handling mechanism；
* decl\_module!：Define business logic code；

Cargo.lock & Cargo.toml

Cargo is rust's package manager, equivalent to nodejs' npm or yarn, but cargo has more functions and also acts as a rust code organization management tool. Cargo provides a series of projects from project establishment, construction to testing, operation to deployment Tools to provide as complete a means as possible for the management of the rust project.

Cargo.lock contains the exact information of dependencies, which is automatically generated by Cargo without manual editing, while Cargo.toml requires manual configuration of dependencies.

Cargo.toml stores project information [package] and dependent libraries [dependencies], etc., which is equivalent to a guide for cargo building projects.

The contents of Cargo.toml in the root directory are as follows：

[profile.release]  
  
panic = 'unwind'  
  
[workspace]  
  
members = [  
  
 'node',  
  
 'pallets/template',  
  
 'runtime',  
  
]

substrate-node-template is a Rust workspace project that can clearly manage component libraries (library) and executable programs (binary).

The members of this [workspace] have：

* node：Executable program, there is an executable main function entry in node/src/main.rs;
* pallets/template：Module code, functions and data structures that can be called externally are defined in pallets/template/src/lib.rs;
* runtime：Component library, the runtime logic is defined in runtime/src/lib.rs;

[profile.release] configured panic='unwind' means to use catch\_unwind together to capture exceptions thrown by panic in a certain thread.

scripts directory

The scripts directory contains two shell scripts：

* docker\_run.sh：Use Docker to start the script of substrate-node-template;
* init.sh：Script to initialize the WASM build environment;

The content of the init.sh script includes upgrading the Rust version：

rustup update nightly  
  
rustup update stable

And add to build the WebAssembly toolchain：

rustup target add wasm32-unknown-unknown *--toolchain nightly*

node directory

The node directory contains the following files：

* build.rs：Custom build script；
* Cargo.toml：Node package building guide；
* src/chain\_spec.rs：Construct ChainSpec (chain specification file)；
* src/cli.rs：Declare client structure and subcommands；
* src/command.rs：Provide implementation functions for client-side related commands；
* src/lib.rs：Import library modules；
* src/main.rs：Substrate-node-template is compiled into the entry file of the executable program；
* src/rpc.rs：The collection of RPC methods specified by the node；
* src/service.rs：Provide tools and methods for constructing Substrate services；

1、Cargo.toml is the node package construction guide, use [[bin]] to indicate that this package is executable：

[[bin]]  
  
name = 'node-template'

Introduce compile-time dependencies through [build-dependencies] and use them in build.rs：

[build-dependencies]  
  
substrate-build-script-utils = '2.0.0'

Other information includes [package], [package.metadata.docs.rs], [dependencies], [features], etc., which are the same as general Cargo.toml.

2、 build.rs is a custom build script, the content is as follows：

use substrate\_build\_script\_utils::{generate\_cargo\_keys, rerun\_if\_git\_head\_changed};  
fn main() {

generate\_cargo\_keys();  
  
 rerun\_if\_git\_head\_changed();  
  
}

The function is to make Cargo compile and execute the script.

3、src/main.rs is the entry file compiled into an executable program by substrate-node-template, the content is as follows：

#![warn(missing\_docs)]  
  
mod chain\_spec;  
#[macro\_use]  
  
mod service;  
  
mod cli;  
  
mod command;  
  
mod rpc;  
  
fn main() -> sc\_cli::Result<()> {  
  
 command::run()

}

The #![warn(missing\_docs)] annotation means that when compiling, if the module lacks documentation, a warning message will be printed.

mod chain\_spec, mod service, mod cli, mod command, mod rpc introduce other modules in the current directory.

#[macro\_use]Load all the macros under the imported module.

The main() function is the entry point of the application, and the returned sc\_cli::Result<()> is a custom Result type:

pub type Result<T> = std::result::Result<T, Error>;

The main() function internally executes the run() function provided by the command module.

4、src/command.rs provides implementation functions for client-related commands, creates an implementation of Cli, SubstrateCli, and defines the run() function.

The run() function first parses the command line parameters through from\_args() and returns a Cli structure, which is defined in cli.rs, and then matches the subcommand in the parameter, and executes it if there is a subcommand.

When executing a subcommand, first call cli.create\_runner(cmd)? to create a runner, and then call async\_run() to execute the subcommand asynchronously.

If there is no subcommand in the command line parameters, call run\_node\_until\_exit() to start the node.

5、src/cli.rs declares the client structure and subcommands.

The Cli structure declaration is as follows:

#[derive(Debug, StructOpt)]  
  
pub struct Cli {  
  
 #[structopt(subcommand)]  
  
 pub subcommand: Option<Subcommand>,  
  
 #[structopt(flatten)]  
  
 pub run: RunCmd,  
  
}

Contains optional sub-commands and command-line options, the compiled substrate-node-template can be passed

./target/release/node-template -h

Get instructions for the available subcommands and options.

Specific subcommands use enumeration declarations:

#[derive(Debug, StructOpt)]  
  
pub enum Subcommand {

BuildSpec(sc\_cli::BuildSpecCmd),  
  
 CheckBlock(sc\_cli::CheckBlockCmd),  
  
 ExportBlocks(sc\_cli::ExportBlocksCmd),  
  
 ExportState(sc\_cli::ExportStateCmd),  
  
 ImportBlocks(sc\_cli::ImportBlocksCmd),  
  
 PurgeChain(sc\_cli::PurgeChainCmd),  
  
 Revert(sc\_cli::RevertCmd),  
  
 #[structopt(name = "benchmark", about = "Benchmark runtime pallets.")]  
 Benchmark(frame\_benchmarking\_cli::BenchmarkCmd),  
  
}

6. src/chain\_spec.rs constructs ChainSpec (chain specification file), ChainSpec defines the available configuration of the chain, which is used to construct the initial block.

Two functions are defined in src/chain\_spec.rs：

* pub fn development\_config() -> Result<ChainSpec, String>
* pub fn local\_testnet\_config() -> Result<ChainSpec, String>

Represents the two modes provided by substrate-node-template：

* The development network (development) specified by the command line option --dev has only one validator Alice;
* Local test network (local\_testnet), there are two validators Alice and Bob;Next, call ChainSpec::from\_genesis to create a chain specification file.

7、src/service.rs provides tools and methods for constructing Substrate services.

src/service.rs first uses the native\_executor\_instance! macro to define a structure Executor, and implements the NativeExecutionDispatch interface, that is, the function can be called by the function name.

The tools and methods of src/service.rs include：

* new\_partial：Build a local node service;
* new\_full：Build a full node service;
* new\_light：Build a light node service;

8. src/rpc.rs provides a collection of RPC methods specified by the node。

rpc.rs provides the create\_full() method to instantiate all complete RPC extensions.

9. src/lib.rs is used to import library modules, the content is as follows:

pub mod chain\_spec;  
  
pub mod service;  
  
pub mod rpc;

runtime directory

The runtime directory contains the following files:

* build.rs：Custom build script;
* Cargo.toml：runtime package construction guide;
* src/lib.rs：Runtime entry file on the chain;

1、Cargo.toml is a build guide for the runtime package. In addition to common configuration items, there are[build-dependencies]：

[build-dependencies]  
  
wasm-builder-runner = { package = 'substrate-wasm-builder-runner', version = '2.0.0' }

Added wasm-builder-runner, which the build script build.rs depends on.

2. build.rs is a custom build script, the content is as follows:use wasm\_builder\_runner::WasmBuilder;  
  
fn main() {  
  
 WasmBuilder::new()  
  
 .with\_current\_project()  
  
 .with\_wasm\_builder\_from\_crates("2.0.0")  
 .export\_heap\_base()

.import\_memory()  
  
 .build()  
  
}

Use wasm-builder-runner to compile the current runtime project into a Wasm binary file located at target/release/wbuild/node-template-runtime/node\_template\_runtime.compact.wasm.

3. src/lib.rs is the runtime entry file on the chain.

#![cfg\_attr(not(feature = "std"), no\_std)] means that if the feature is not std (Rust standard library) when compiling, then it must be no\_std (compiling to Wasm).

#![recursion\_limit="256"]Set the maximum number of infinite recursive operations that may occur during compilation.

The following code:

#[cfg(feature = "std")]  
  
include!(concat!(env!("OUT\_DIR"), "/wasm\_binary.rs"));

Indicates that when using the Rust standard library to compile, the generated Wasm binary content will be introduced into the current runtime code by means of constants.

Introduce dependent modules and template modules:

pub use pallet\_template;

Next are the aliases of some basic types required by runtime, which are consistent with the related type names in the module.

The opaque module encapsulates some types for CLI initialization.

Define constants related to block time:

pub const MILLISECS\_PER\_BLOCK: u64 = 6000;

That is, the production time of each block is 6 seconds, and the configuration can be modified as needed.

Next, use the parameter\_types! macro to generate data types that meet the Get interface required by some functional modules.

Then implement the interface of each functional module for runtime:

impl frame\_system::Trait for Runtime {...}  
  
impl pallet\_aura::Trait for Runtime {...}  
  
impl pallet\_grandpa::Trait for Runtime {...}  
  
impl pallet\_timestamp::Trait for Runtime {...}  
  
impl pallet\_balances::Trait for Runtime {...}  
  
impl pallet\_transaction\_payment::Trait for Runtime {...}  
  
impl pallet\_sudo::Trait for Runtime {…}

impl pallet\_template::Trait for Runtime {...}

The runtime is constructed by the construct\_runtime! macro:

construct\_runtime!(  
  
 pub enum Runtime where  
  
 Block = Block,  
  
 NodeBlock = opaque::Block,  
  
 UncheckedExtrinsic = UncheckedExtrinsic  
  
 {  
  
 System: frame\_system::{Module, Call, Config, Storage, Event<T>},  
  
 RandomnessCollectiveFlip: pallet\_randomness\_collective\_flip::{Module, Call, Storage},  
  
 Timestamp: pallet\_timestamp::{Module, Call, Storage, Inherent},  
  
 Aura: pallet\_aura::{Module, Config<T>, Inherent},  
 Grandpa: pallet\_grandpa::{Module, Call, Storage, Config, Event},  
  
 Balances: pallet\_balances::{Module, Call, Storage, Config<T>, Event<T>},  
  
pallet\_transaction\_payment::{Module, Storage},  
  
 Sudo: pallet\_sudo::{Module, Call, Config<T>, Storage, Event<T>},  
  
 // Include the custom logic from the template pallet in the runtime.  
  
 TemplateModule: pallet\_template::{Module, Call, Storage, Event<T>},  
  
 }  
  
);

The construct\_runtime! macro constructs the runtime according to the module name and the components in the used module. The initial storage is loaded in order during construction. Therefore, when module B depends on module A, module A should be placed before module B.

Finally, use the impl\_runtime\_apis! macro to implement the interfaces defined by the runtime api, and these interfaces are defined by the decl\_runtime\_apis! macro. pallets/template directory

The pallets directory can contain multiple pallets (modules), and template is a pallet.

The pallets/template directory contains the following files：

* Cargo.toml：Template module building guide；
* src/lib.rs：The specific function implementation code of the module;
* src/mock.rs：Test case service code;
* src/tests.rs：Test case

1. Cargo.toml is the construction guide for the template module, which is mainly configured according to requirements[dependencies] and [features].

[dependencies] is the dependency library of the module. [features] uses std feature by default to ensure that the runtime can be compiled to either the native version (using std feature) or the wasm version (using no\_std feature).

2. src/lib.rs is the specific function implementation code of the module.

Mock and tests are only compiled when running tests.

Then the type declaration:

pub trait Trait: frame\_system::Trait {  
  
 type Event: From<Event<Self>> + Into<<Self as frame\_system::Trait>::Event>;  
  
}

And four macros related to business logic code:

* decl\_storage!：Define storage；
* decl\_event!：Define event；
* decl\_error!：Define error handling mechanism；
* decl\_module!：Define business logic code；