A Simple Rust Coroutines

Pang Anqi
SIST
Shanghaitech University
49954521
Email: pangaq@shanghaitech.edu.cn

Qin Xin
SIST
Shanghaitech University
50822444
Email: qinxin@shanghaitech.edu.cn

Abstract—**This report introduces the revival of coroutines as a general control abstration. We introduce the concept of coroutines and the classification of coroutines. We show how to build a coroutine, the API of it and what we need to improve. Finally , we provide a collection of examples that illustrates the use of full asymmetric coroutines.**

1. Introduction

The concept of coroutines was introduced in the early 1960s and was one of the oldest proposals for general control abstraction. Its attributer Conway described coroutines as "subroutines who act as the master program". Coroutines were widely explored during the next 20 years in several contexts, including simulation, artificial intelligence, concurrent programming, text processing, and various kinds of data-structure manipulation [1] [2] [3].

1.1. Definition of Coroutins

The rarity of coroutine in mainstream languages can be partly attributed to that the concept was never procisely defined. Marlin's doctoral thesis [2], resumes the fundamental characteristics of a coroutine as follows:

- the values of data local to a coroutine persist between successive calls;
- the values of data local to a coroutine persist between successive calls;

1.2. Different construct of implementing coroutines

This is the common perception of the coroutine. We can identify three main issues that distinguish coroutine facilities:

- the control-transfer mechanism, which can provide *symmetric* or *asymmetric* coroutines;
- whether coroutines are provided in the language as *first-class* objects, which can be freely manipulated by the programmer, or as constrained constructs;
- whether a coroutine is a *stackful* construct, i.e., whether it is able to suspend its execution from within nested calls.

As a consequence of the distinguish coroutine facilities, some different implementations of coroutines were developed, such as Python generators(stackless,non-first-class), Goroutine, and coroutine in Lua. In our project, we implement the stackful asymmetric coroutines.

Another significant reason for absence of coroutines in modern mianstream languages is the *multithreading*. In the last few years, mainstream languages like Java , C and C# provide *thread* as their primary concurrency construct, Rust the same.

The old version of Rust implement coroutine but delete it in consideration of the aim of focusing more on security. Therefore we implement an alternative choice of coroutine in Rust through reviewing the old source code.

The next section will introduce the coroutine classification, Rust coroutine operators and framework, applications of coroutines.

2. Coroutines Classification

In this section we identify the three issues that distinguish coroutine mechanisms and influence their usefulness despite of their common ability of keeping state between successive calls.

2.1. Control Transfer Mechanism

In the sight of control transfer mechanism, there are two kinds of coroutines – *symmetric* and *asymmetric* coroutines. *Symmetric* coroutine provide a single control-transfer operation that allows coroutines to pass control between themselves. *Asymmetric* coroutine provide two control-transfer operations, one for invoking a coroutine another for suspending(in our project, there will be suspending and blocked, which are also one kind of suspending). The suspending will return control to the coroutine invoker.

Coroutine implemented for producing sequences of values typically provide *asymmetric* coroutines, such as iterators and generators.

A general purpose coroutine should provide either symmetric or asymmetric coroutine. But in this project we only implement asymmetric coroutines for no expressive power is lost if only asymmetric coroutines are provided in a language, which will be discussed in **Section 4.1**

2.2. Firts-class

Whether coroutine is provided as first-class objects influences its expressive power. In some cases that coroutines are implemented for particular uses, the coroutine is constrained coroutines, like Python iterator. In Lua, Go and Rust the coroutine is powerful enough to provide programmer-defined control structures, and this facility is only provided when coroutine is implemented as first-class objects.

2.3. Stackful or Stackless

Stackful coroutine allow their execution to suspend with nested functions, the next time the coroutine is resumed, its execution will continue from where it suspend. Stackless coroutines can only suspend its execution when control stack is at the same level of when it was created, so only main body of the stackless coroutine can suspend. Most scripting language, notable Python and Perl provide an `yield` statement and *generator function*, they are stackless, first-class, asymmetric coroutines. This type of generator is not powerful enough to implement user-level multitasking.

2.4. Full Coroutines

Whether first class and stackful determines the expressive power of the coroutine. Our implementation is asymmetric, first-class, stackful(with stackless option) full coroutine supporting user-level multitasking.

3. Rust Coroutine Operators and the Framework

3.1. Operators

Our coroutine of full asymmetric coroutine has three basic operators:`spawn`, `resume`, and `yield`.

The operator `spawn` creates a new coroutine. It is implement by `spawn_opts(f: F, opts: Options)` which the Option is `Default::default()`–2Mb sized stack. Also we can use `Options` with zero sized stack to implement a stackless coroutine.

And the coroutine also receives nested functions as one of its argument, after `spawn` or `spawn_opts` is called, it will return a coroutine it created. Creating a coroutine does not start its execution; a new coroutine begins in suspended state with its *context* set to the first statement of its main body.

The operator `resume` activate or reactivate a coroutine. And there are two types of resume, one is the default `resume` and another is `resume_with`, their name fit their functions, the `resume_with` method will resume with a data T. Once resumed, a coroutine starts execution form the saved *context* and runs until it suspends or its main function terminates, control is then transferred back to the coroutine's invocation point. After main function terminates, coroutine will switch its state to `Finished` and cannot be resumed any more.

The operator `yield` suspends a coroutine execution. The coroutine *context* is saved so that when it's resumed its execution will continue from the point it suspended. We also have several kinds of `yield`, `yield_back`,`yield_with` and `sched`, `block`. These methods are based on `yield_to`, and `try_switch`, and these two were implemented by `Context::swap` in extern module *context* of Rust.

Mostly we use `yield_back` ,`sched` , `block`, when implementing a generator, it should use `yield_with`.

There are interesting things for `yield` and `resume`. The first time a coroutine is activated, a value can be given to the operator `resume` and it's passed to the coroutine function. In subsequent reactivations of a coroutine, that argument becomes the value of the operator `yield`. On the other hand, when a coroutine suspends, the argument passed to the operator `yield` becomes the result value of the operator `resume` that activated the coroutine. When a coroutine terminates, the value returned by its main function becomes the result value of its last reactivation, which is same as Lua.

3.2. Framework

The coroutine in Rust is running in one thread, so in the thread it is non-preemptive scheduling. In the thread, we set a environment stack to maintain the relations between coroutines, such as *resume* and *yield*.

The top of the environment is the current running coroutine, when invoke `yield`, pop out the current running coroutine and run the new top coroutine; when invoke `resume`, push the coroutine into the stack.

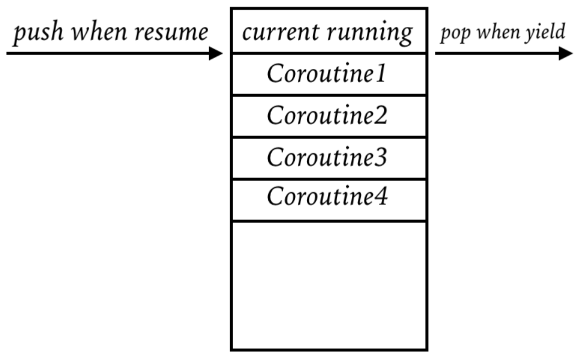

Figure 1. environment stack

For every coroutine, when spawned, it will return a handle, the handle contains all the information of coroutine:

name, stack, state and context. And there are two types of handles, one is clonable, another is unique. The clonable coroutine handle use `Mutex` to avoid race conditions.

There are six states for coroutine: Normal, means the coroutine is waiting for another coroutine to give back control; Suspended, means the coroutine is suspended and waited to be waked up by `resume`; Blocked, similar to Suspended, can be waked up by `resume`; Running , means the coroutine is running; Finished, means the main function of coroutine is finished and it cannot be resumed any more; Panicked, the coroutine panic inside it.

There are many auxiliary functions and struct when implement the main coroutine.

4. Expressing and Examples

4.1. Symmetric Coroutines

The basic characteristic of symmetric coroutine facilities is the provision of a single control-transfer operation that allows coroutines to pass control to each other. It is easy to demonstrate that we can provide any of *symmetric* or *asymmetric* mechanisms using the other; therefore, no expressive power is lost if only asymmetric coroutines are provided in a language.

The implementation of symmetric coroutines on top of asymmetric facilities is straightforward. Symmetrical transfers of control between asymmetric coroutines can be easily simulated with pairs of yieldresume operations and an auxiliary dispatching loop that acts as an intermediary in the switch of control between the two coroutines. When a coroutine wishes to transfer control, it yields to the dispatching loop, which in turn resumes the coroutine that must be reactivated.

This is why in **Section 2.1** this project only implement asymmetric coroutines.

4.2. Applications for coroutine

4.2.1. The Producer-Consumer Problem. The producer-consumer problem is the best example of the use of coroutines. This problem involves the interaction between two parts: one produces a sequence of items and one consumes them. Asymmetric coroutines provide a simple and structured solution to this problem: we can implement the consumer as a conventional function that *resumes* the producer(an asymmetric coroutine) when the next item is required.

pipeline is the extension of the producer-consumer problem, asymmetric coroutine provide a easy implementation of *pipeline* too. A filter behaves both as consumer and a producer, and can be implemented by a coroutine that resumes its antecessor to get a new value and yield the transformed value to its invoker(the next consumer in the chain).

4.2.2. Generators. A *generator* is a control abstraction that produces a sequence of values, returning a new value to its caller for each invocation. This can also be regard as a producer-consumer problem, the generator act as the producer.

A typical use of generators is to implement *iterators*. Besides the capability of keeping state, the possibility of exchanging data when transforming control makes asymmetric coroutines a very good facility for implementing iterators.

4.2.3. Cooperative Multitasking. One of the most obvious uses of coroutines is to implement multitasking.

multithread mechanism use preemptive scheduling, this scheduling and the consequent need for complex synchronization mechanisms make developing a correct multithreading application a difficult task. Preemptive is necessary for operating systems and real-time applications as timely responses are essential sometimes. However, most thread implementations do not need and real timing guarantees. In this scenario, a cooperative multitasking environment, which eliminates conflicts due to race conditions and minimizes the need for synchronization seems much more acceptable. Application 'Wechat' also reconstruct their system with coroutine.

The eliminates of race conditions make the coroutine also called *green thread*

The only drawback of the cooperative multitasking is when using blocking operations; if a coroutine calls an I/O operation and blocks, the entire program blocks until the operation completes. However, we can providing auxiliary functions that initiate an I/O operation and suspend the active coroutine when operation cannot be immediately completed. Ierusalimschy [4] shows an example of a application that use Lua to implement this.

5. Conclusions and Source Code

For the renewal of interest in coroutines, we implement coroutine in Rust by reviewing the old source code once in the Rust lib and extend it to support more. And this is a full asymmetric, stackful or stackless, first-class coroutine.

Source code:
https://github.com/panganqi/OS2_final_project/
tree/master/Final_project

References

[1] Knuth, D. E. , *Art of Computer Programming, Volume 1, Fundamental Algorithms*, . Addison-Wesley, Reading, MA. 1968.
2

[2] Marlin, C. D.*Coroutines: A Programming Methodology, a Language Design and an Implementation*, . LNCS 95, Springer-Verlag. 1980.
3

[3] Pauli, W. and Soffa, M. L.*Coroutine behaviour and implementation. Software: Practice & Experience 10*, . 1980.
4

[4] Ierusalimschy, R.*Programming in Lua*, . Lua.org, ISBN 85-903798-1-7, Rio de Janeiro, Brazil. 2003.