Rust Tasks and Communication Tutorial

1 Introduction

Rust provides safe concurrency through a combination of lightweight, memory-isolated tasks and message passing. This tutorial will describe the concurrency model in Rust, how it relates to the Rust type system, and introduce the fundamental library abstractions for constructing concurrent programs.

Rust tasks are not the same as traditional threads: rather, they are considered *green threads*, lightweight units of execution that the Rust runtime schedules cooperatively onto a small number of operating system threads. On a multi-core system Rust tasks will be scheduled in parallel by default. Because tasks are significantly cheaper to create than traditional threads, Rust can create hundreds of thousands of concurrent tasks on a typical 32-bit system. In general, all Rust code executes inside a task, including the `main` function.

In order to make efficient use of memory Rust tasks have dynamically sized stacks. A task begins its life with a small amount of stack space (currently in the low thousands of bytes, depending on platform), and acquires more stack as needed. Unlike in languages such as C, a Rust task cannot accidentally write to memory beyond the end of the stack, causing crashes or worse.

Tasks provide failure isolation and recovery. When a fatal error occurs in Rust code as a result of an explicit call to `fail!()`, an assertion failure, or another invalid operation, the runtime system destroys the entire task. Unlike in languages such as Java and C++, there is no way to `catch` an exception. Instead, tasks may monitor each other for failure.

Tasks use Rust's type system to provide strong memory safety guarantees. In particular, the type system guarantees that tasks cannot share mutable state with each other. Tasks communicate with each other by transferring *owned* data through the global *exchange heap*.

1.1 A note about the libraries

While Rust's type system provides the building blocks needed for safe and efficient tasks, all of the task functionality itself is implemented in the standard and extra libraries, which are still under development and do not always present a consistent or complete interface.

For your reference, these are the standard modules involved in Rust concurrency at this writing:

- `std::task` - All code relating to tasks and task scheduling,
- `std::comm` - The message passing interface,
- `std::pipes` - The underlying messaging infrastructure,

Rust 0.8-pre

fbeeeebf

- `extra::comm` - Additional messaging types based on `std::pipes`,
- `extra::sync` - More exotic synchronization tools, including locks,
- `extra::arc` - The Arc (atomically reference counted) type, for safely sharing immutable data,
- `extra::future` - A type representing values that may be computed concurrently and retrieved at a later time.

2 Basics

The programming interface for creating and managing tasks lives in the `task` module of the `std` library, and is thus available to all Rust code by default. At its simplest, creating a task is a matter of calling the `spawn` function with a closure argument. `spawn` executes the closure in the new task.

```
// Print something profound in a different task using a named function
fn print_message() { println("I am running in a different task!"); }
spawn(print_message);

// Print something more profound in a different task using a lambda expression
spawn( || println("I am also running in a different task!") );

// The canonical way to spawn is using `do` notation
do spawn {
    println("I too am running in a different task!");
}
```

In Rust, there is nothing special about creating tasks: a task is not a concept that appears in the language semantics. Instead, Rust's type system provides all the tools necessary to implement safe concurrency: particularly, *owned types*. The language leaves the implementation details to the standard library.

The `spawn` function has a very simple type signature: `fn spawn(f: ~fn())`. Because it accepts only owned closures, and owned closures contain only owned data, `spawn` can safely move the entire closure and all its associated state into an entirely different task for execution. Like any closure, the function passed to `spawn` may capture an environment that it carries across tasks.

```
// Generate some state locally
let child_task_number = generate_task_number();

do spawn {
   // Capture it in the remote task
   println(fmt!("I am child number %d", child_task_number));
}
```

By default, the scheduler multiplexes tasks across the available cores, running in parallel. Thus, on a multicore machine, running the following code should interleave the output in vaguely random order.

Rust 0.8-pre
fbeeeebf

```
for child_task_number in range(0, 20) {
```

```
    do spawn {
        print(fmt!("I am child number %d\n", child_task_number));
    }
}
```

2.1 Communication

Now that we have spawned a new task, it would be nice if we could communicate with it. Recall that Rust does not have shared mutable state, so one task may not manipulate variables owned by another task. Instead we use *pipes*.

A pipe is simply a pair of endpoints: one for sending messages and another for receiving messages. Pipes are low-level communication building-blocks and so come in a variety of forms, each one appropriate for a different use case. In what follows, we cover the most commonly used varieties.

The simplest way to create a pipe is to use the `pipes::stream` function to create a `(Port, Chan)` pair. In Rust parlance, a *channel* is a sending endpoint of a pipe, and a *port* is the receiving endpoint. Consider the following example of calculating two results concurrently:

```
let (port, chan): (Port<int>, Chan<int>) = stream();

do spawn || {
    let result = some_expensive_computation();
    chan.send(result);
}

some_other_expensive_computation();
let result = port.recv();
```

Let's examine this example in detail. First, the `let` statement creates a stream for sending and receiving integers (the left-hand side of the `let`, `(chan, port)`, is an example of a *destructuring let*: the pattern separates a tuple into its component parts).

```
let (port, chan): (Port<int>, Chan<int>) = stream();
```

The child task will use the channel to send data to the parent task, which will wait to receive the data on the port. The next statement spawns the child task.

```
do spawn || {
    let result = some_expensive_computation();
    chan.send(result);
}
```

Notice that the creation of the task closure transfers `chan` to the child task implicitly: the closure captures `chan` in its environment. Both `Chan` and `Port` are sendable types and may be captured into tasks or otherwise transferred between them. In the example, the child task runs an expensive computation, then sends the result over the captured channel.

Rust 0.8-pre
fbeeeebf

Finally, the parent continues with some other expensive computation, then waits for the child's result to arrive on the port:

```
some_other_expensive_computation();
let result = port.recv();
```

The `Port` and `Chan` pair created by `stream` enables efficient communication between a single sender and a single receiver, but multiple senders cannot use a single `Chan`, and multiple receivers cannot use a single `Port`. What if our example needed to compute multiple results across a number of tasks? The following program is ill-typed:

```
let (port, chan) = stream();

do spawn {
    chan.send(some_expensive_computation());
}

// ERROR! The previous spawn statement already owns the channel,
// so the compiler will not allow it to be captured again
do spawn {
    chan.send(some_expensive_computation());
}
```

Instead we can use a `SharedChan`, a type that allows a single `Chan` to be shared by multiple senders.

```
let (port, chan) = stream();
let chan = SharedChan::new(chan);

for init_val in range(0u, 3) {
    // Create a new channel handle to distribute to the child task
    let child_chan = chan.clone();
    do spawn {
        child_chan.send(some_expensive_computation(init_val));
    }
}

let result = port.recv() + port.recv() + port.recv();
```

Here we transfer ownership of the channel into a new `SharedChan` value. Like `Chan`, `SharedChan` is a non-copyable, owned type (sometimes also referred to as an *affine* or *linear* type). Unlike with `Chan`, though, the programmer may duplicate a `SharedChan`, with the `clone()` method. A cloned `SharedChan` produces a new handle to the same channel, allowing multiple tasks to send data to a single port. Between `spawn`, `stream` and `SharedChan`, we have enough tools to implement many useful concurrency patterns.

Note that the above `SharedChan` example is somewhat contrived since you could also simply use three `stream` pairs, but it serves to illustrate the point. For reference, written with multiple streams, it might look like the example below.

Rust 0.8-pre
fbeeeebf

```
// Create a vector of ports, one for each child task
let ports = do vec::from_fn(3) |init_val| {
    let (port, chan) = stream();
    do spawn {
        chan.send(some_expensive_computation(init_val));
    }
    port
};

// Wait on each port, accumulating the results
let result = ports.iter().fold(0, |accum, port| accum + port.recv() );
```

2.2 Backgrounding computations: Futures

With `extra::future`, rust has a mechanism for requesting a computation and getting the result later.

The basic example below illustrates this.

```
fn fib(n: uint) -> uint {
    // lengthy computation returning an uint
    12586269025
}

let mut delayed_fib = extra::future::spawn (|| fib(50) );
make_a_sandwich();
println(fmt!("fib(50) = %?", delayed_fib.get()))
```

The call to `future::spawn` returns immediately a `future` object regardless of how long it takes to run `fib(50)`. You can then make yourself a sandwich while the computation of `fib` is running. The result of the execution of the method is obtained by calling `get` on the future. This call will block until the value is available (*i.e.* the computation is complete). Note that the future needs to be mutable so that it can save the result for next time `get` is called.

Here is another example showing how futures allow you to background computations. The workload will be distributed on the available cores.

```
fn partial_sum(start: uint) -> f64 {
    let mut local_sum = 0f64;
    for num in range(start*100000, (start+1)*100000) {
        local_sum += (num as f64 + 1.0).pow(&-2.0);
    }
    local_sum
}

fn main() {
    let mut futures = vec::from_fn(1000, |ind| do extra::future::spawn { partial_sum(i
```

Rust 0.8-pre

fbeeeebf

```
    let mut final_res = 0f64;
    for ft in futures.mut_iter()  {
        final_res += ft.get();
    }
    println(fmt!("π^2/6 is not far from : %?", final_res));
}
```

2.3 Sharing immutable data without copy: Arc

To share immutable data between tasks, a first approach would be to only use pipes as we have seen previously. A copy of the data to share would then be made for each task. In some cases, this would add up to a significant amount of wasted memory and would require copying the same data more than necessary.

To tackle this issue, one can use an Atomically Reference Counted wrapper (`Arc`) as implemented in the `extra` library of Rust. With an Arc, the data will no longer be copied for each task. The Arc acts as a reference to the shared data and only this reference is shared and cloned.

Here is a small example showing how to use Arcs. We wish to run concurrently several computations on a single large vector of floats. Each task needs the full vector to perform its duty.

```
use extra::arc::Arc;

fn pnorm(nums: &~[float], p: uint) -> float {
    nums.iter().fold(0.0, |a,b| a+(*b).pow(&(p as float)) ).pow(&(1f / (p as float)))
}

fn main() {
    let numbers = vec::from_fn(1000000, |_| rand::random::<float>());
    println(fmt!("Inf-norm = %?",  *numbers.iter().max().unwrap()));

    let numbers_arc = Arc::new(numbers);

    for num in range(1u, 10) {
        let (port, chan)  = stream();
        chan.send(numbers_arc.clone());

        do spawn {
            let local_arc : Arc<~[float]> = port.recv();
            let task_numbers = local_arc.get();
            println(fmt!("%u-norm = %?", num, pnorm(task_numbers, num)));
        }
    }
}
```

The function `pnorm` performs a simple computation on the vector (it computes the sum of its items at the power given as argument and takes the inverse power of this value). The Arc on the vector is created by the line

Rust 0.8-pre

fbeeeebf

```
let numbers_arc=Arc::new(numbers);
```

and a clone of it is sent to each task

```
chan.send(numbers_arc.clone());
```

copying only the wrapper and not its contents.

Each task recovers the underlying data by

```
let task_numbers = local_arc.get();
```

and can use it as if it were local.

The `arc` module also implements Arcs around mutable data that are not covered here.

3 Handling task failure

Rust has a built-in mechanism for raising exceptions. The `fail!()` macro (which can also be written with an error string as an argument: `fail!( ~reason)`) and the `assert!` construct (which effectively calls `fail!()` if a boolean expression is false) are both ways to raise exceptions. When a task raises an exception the task unwinds its stack---running destructors and freeing memory along the way---and then exits. Unlike exceptions in C++, exceptions in Rust are unrecoverable within a single task: once a task fails, there is no way to "catch" the exception.

All tasks are, by default, *linked* to each other. That means that the fates of all tasks are intertwined: if one fails, so do all the others.

```
// Create a child task that fails
do spawn { fail!() }

// This will also fail because the task we spawned failed
do_some_work();
```

While it isn't possible for a task to recover from failure, tasks may notify each other of failure. The simplest way of handling task failure is with the `try` function, which is similar to `spawn`, but immediately blocks waiting for the child task to finish. `try` returns a value of type `Result<int, ()>`. `Result` is an `enum` type with two variants: `Ok` and `Err`. In this case, because the type arguments to `Result` are `int` and `()`, callers can pattern-match on a result to check whether it's an `Ok` result with an `int` field (representing a successful result) or an `Err` result (representing termination with an error).

```
let result: Result<int, ()> = do task::try {
    if some_condition() {
        calculate_result()
    } else {
        fail!("oops!");
    }
```

Rust 0.8-pre

fbeeeebf

```
};
assert!(result.is_err());
```

Unlike `spawn`, the function spawned using `try` may return a value, which `try` will dutifully propagate back to the caller in a `Result` enum. If the child task terminates successfully, `try` will return an `Ok` result; if the child task fails, `try` will return an `Error` result.

> ***Note:*** A failed task does not currently produce a useful error value (`try` always returns `Err(())`). In the future, it may be possible for tasks to intercept the value passed to `fail!()`.

TODO: Need discussion of `future_result` in order to make failure modes useful.

But not all failures are created equal. In some cases you might need to abort the entire program (perhaps you're writing an assert which, if it trips, indicates an unrecoverable logic error); in other cases you might want to contain the failure at a certain boundary (perhaps a small piece of input from the outside world, which you happen to be processing in parallel, is malformed and its processing task can't proceed). Hence, you will need different *linked failure modes*.

3.1 Failure modes

By default, task failure is *bidirectionally linked*, which means that if either task fails, it kills the other one.

```
do spawn {
    do spawn {
        fail!();  // All three tasks will fail.
    }
    sleep_forever();  // Will get woken up by force, then fail
}
sleep_forever();  // Will get woken up by force, then fail
```

If you want parent tasks to be able to kill their children, but do not want a parent to fail automatically if one of its child task fails, you can call `task::spawn_supervised` for *unidirectionally linked* failure. The function `task::try`, which we saw previously, uses `spawn_supervised` internally, with additional logic to wait for the child task to finish before returning. Hence:

```
let (receiver, sender): (Port<int>, Chan<int>) = stream();
do spawn {  // Bidirectionally linked
    // Wait for the supervised child task to exist.
    let message = receiver.recv();
    // Kill both it and the parent task.
    assert!(message != 42);
}
do try {  // Unidirectionally linked
    sender.send(42);
```

Rust 0.8-pre
fbeeeebf

```
    sleep_forever();  // Will get woken up by force
}
// Flow never reaches here -- parent task was killed too.
```

Supervised failure is useful in any situation where one task manages multiple fallible child tasks, and the parent task can recover if any child fails. On the other hand, if the *parent* (supervisor) fails, then there is nothing the children can do to recover, so they should also fail.

Supervised task failure propagates across multiple generations even if an intermediate generation has already exited:

```
do task::spawn_supervised {
    do task::spawn_supervised {
        sleep_forever();  // Will get woken up by force, then fail
    }
    // Intermediate task immediately exits
}
wait_for_a_while();
fail!();  // Will kill grandchild even if child has already exited
```

Finally, tasks can be configured to not propagate failure to each other at all, using `task::spawn_unlinked` for *isolated failure*.

```
let (time1, time2) = (random(), random());
do task::spawn_unlinked {
    sleep_for(time2);  // Won't get forced awake
    fail!();
}
sleep_for(time1);  // Won't get forced awake
fail!();
// It will take MAX(time1,time2) for the program to finish.
```

3.2 Creating a task with a bi-directional communication path

A very common thing to do is to spawn a child task where the parent and child both need to exchange messages with each other. The function `extra::comm::DuplexStream()` supports this pattern. We'll look briefly at how to use it.

To see how `DuplexStream()` works, we will create a child task that repeatedly receives a `uint` message, converts it to a string, and sends the string in response. The child terminates when it receives `0`. Here is the function that implements the child task:

```
fn stringifier(channel: &DuplexStream<~str, uint>) {
    let mut value: uint;
    loop {
        value = channel.recv();
        channel.send(uint::to_str(value));
        if value == 0 { break; }
```

Rust 0.8-pre

fbeeeebf

```
    }
}
```

The implementation of `DuplexStream` supports both sending and receiving. The `stringifier` function takes a `DuplexStream` that can send strings (the first type parameter) and receive `uint` messages (the second type parameter). The body itself simply loops, reading from the channel and then sending its response back. The actual response itself is simply the stringified version of the received value, `uint::to_str(value)`.

Here is the code for the parent task:

```
let (from_child, to_child) = DuplexStream();

do spawn {
    stringifier(&to_child);
};

from_child.send(22);
assert!(from_child.recv() == ~"22");

from_child.send(23);
from_child.send(0);

assert!(from_child.recv() == ~"23");
assert!(from_child.recv() == ~"0");
```

The parent task first calls `DuplexStream` to create a pair of bidirectional endpoints. It then uses `task::spawn` to create the child task, which captures one end of the communication channel. As a result, both parent and child can send and receive data to and from the other.

Rust 0.8-pre

fbeeeebf