Rust 运行时指南

A Guide to the Rust Runtime, by Alex Crichton and Brian Anderson
翻译：庄晓立（Liigo），com.liigo@gmail.com, G+,Weibo,CSDN,Rust 中文圈
日期：2014 年 2 月。

Rust 编程语言的标准发行版包含两个运行时库（libgreen 和 libnative），提供 I/O 等基础设施的统一接口。但对 Rust 语言本身而言，运行时（runtime）并不是必需的；Rust 编译器可以生成在所有环境中运行的代码，包括内核（kernel）环境。Rust 语言也不需要运行时提供内存安全，因为它的类型系统本身已经足够安全——通过编译时静态验证给予保证。运行时只是利用语言的安全特性提供一系列便利的、安全的、高层的抽象。

如果 Rust 没有运行时（runtime），我们编程能做的事情非常有限，所以 Rust 需要提供运行时。这份指导手册将探讨 Rust 用户空间（user-space）的运行时、如何使用它、它能做什么。

1、什么是运行时？

Rust 运行时可以被视为提供以下功能代码的组合：输入/输出（I/O）、任务孵化（task spawning）、任务本地存储（TLS）等等。本质上，它提供一少部分对象，为实现常见功能提供便利支持。Rust 运行时自身的实现是自包含的（self-contained），避免干扰其他库。

运行时目前提供以下功能特性（不完整列表）：

- 输入/输出（I/O）
- 任务孵化（task spawning）
- 消息传递（message passing）
- 任务同步（task synchronization）
- 任务本地存储（TLS, task-local storage）
- 日志（logging）
- 本地堆（local heaps）（GC heaps）
- 任务展开（task unwinding）

1.1、运行时的目标是什么？

运行时的设计初衷是达成以下目标：

- Rust 库能够在多种环境中运行，而不需要关心各种环境的具体细节。经常被提及的两种环境是 M:N 和 1:1。Rust 运行时最初首先支持 M:N，现在也同时支持了 1:1。
- Rust 运行时在达成在多种环境中运行的目标时，不需要强制区分不同的编译模式。一个库被编译一次就能在多种不同的环境中永久运行，是我们明确的设计目标。
- Rust 运行时应当高效运行。在其架构设计上，不应该有阻止程序高效运行的障碍。但也不是说非得在任何情况下都必须以最快的速度运行。
- Rust 运行时应当尽可能对用户透明。不鼓励用户直接与运行时交互。

2、运行时的体系结构

本节将介绍目前的 Rust 运行时的体系结构。Rust 运行时曾经被重写了几遍，本节仅涉及当前最新版本。

2.1、本地任务（a local task）

Rust 运行时的核心抽象概念是任务（Task）。任务代表了运行 Rust 代码的“线程”，但此“线程”并不一定直接对应于操作系统里的线程。运行时里的大多数服务都是通过 Task 提供给用户，因而可以做到单个任务内部决策。

采用这种策略的结果是，要求所有使用标准库的 Rust 代码，都有一个本地的 Task 结构体（local Task structure）。该结构体被存储在操作系统的线程局部存储（TLS, Thread Local Storage）内部，以便高效访问。

一定有这么一个 Task 结构体存在，是 Rust 运行时本质上唯一的假定。这是一个核心假定，令所有使用标准库的代码受益，因此 Task 被定义在标准库内。几乎所有运行时服务都是通过 Task 提供的。

2.2、输入/输出（I/O）

当处理 I/O 时，通常有一些约定俗成的方法，但这些方法未必在任何情况下都正确。I/O 的处理非常复杂，几乎不可能在多种环境中使用一致的处理方案。不能保证 Rust 任务（Task）有权限处理 I/O，也不能保证它以何种方式处理 I/O。

这意味着，标准库中无法定义处理 I/O 功能的具体实现代码，只能定义一批 I/O 操作接口，由各环境下的 Task 各自实现具体功能。这些 I/O 接口被设计为以同步 I/O 调用为核心。此架

构不会根本性的阻碍以其他形式处理 I/O，但目前还没有别的处理方式。

运行时（runtime）必须实现的这些 I/O 接口被定义在 std::rt::rtio 模块内。注意这些接口是不稳定的，是不对用户公开的（仅作为标准库内部实现细节）。

（译者 Liigo 注：任务的接口被定义在标准库 libstd 中，任务的具体实现被定义在运行时库 libgreen/libnative 中。）

2.3、任务孵化（Task spawning）

任务（Task）的一项常见操作是，孵化（spawning）一个子任务（child task），在其中执行某些工作。这意味着并行执行被启用。如何孵化子任务，没有一个统一方法（未在标准库中定义），由各（运行时内的）任务自行决定。

任务孵化被解释为“孵化一个子任务（原文为 sibling，疑为 child 之笔误）”，其高层操作接口定义在 std::task 模块中。孵化子任务前，可以事先设定子任务的参数，运行时的实现必须依据这些参数，执行具体的孵化行为。

任务的另一个操作是处理自身的运行状态，如阻塞（block）和唤醒（wake up）任务。具体操作细节由任务自己决定，标准库未做规定。

2.4、运行时接口（trait Runtime）和任务结构体（struct Task）

运行时的所有特性都被定义在接口`Runtime`和结构体`Task`。在不同运行时库（libgreen、libnative）中，结构体 Task 都是相同的，而接口 Runtime 的实现各不相同。Task 内部存储了 Runtime 接口的实现对象，因而可以调用其接口函数。

3、运行时的实现（implementations）

Rust 发行版提供了两个运行时库，分别是 1:1 线程模型的 libnative 和 M:N 线程模型的 libgreen。就像许多计算机科学问题一样，你很难说选择哪个运行时库是正确的，它们各自都有优势和劣势。下面分别介绍两个运行时库提供的功能和没有的功能，供程序员参考并自行决定选择使用哪一个。

3.1、1:1 - 使用 libnative

libnative 运行时库的实现，是基于操作系统本地线程，加上 libc 阻塞 I/O 调用。因其用户空间的线程一一对应于操作系统线程，而被称为 1:1 线程模型。

在这种模型下，每一个 Rust 任务（Task）对应于一个操作系统线程，并且每一个 I/O 对象唯一对应一个文件描述符(fd)（或者其他系统内的对等物）。

使用 libnative 的一些优势：

- 保证与 FFI 绑定（外部函数接口绑定）交互操作。即使你调用的外部 C 库函数（例如数据库驱动）阻塞在线程 I/O，也不会干扰其他 Rust 任务（Task）正常执行（因为它们在不同的系统线程内）。

- 在某些情况下相比 M:N 有更少的 I/O 损耗。并非所有 M:N I/O 都保证尽最大可能的快，而且有些东西（比如文件系统 API）在某些平台下不是真正的异步操作，意味着 M:N 实现可能会比 1:1 实现引发更多损耗。

3.2、M:N - 使用 libgreen

运行时库 libgreen 基于异步 I/O 框架 libuv 实现了“绿色线程”。M:N 线程模型中的 M 是指当前进程内的操作系统本地线程个数，N 是指 Rust 任务个数（即绿色线程个数）。在这种模型中，M 个系统线程调度运行 N 个 Rust 绿色线程，在用户空间中进行线程上下文切换（context switching）。（译者 Liigo 注：通常情况下，N 远大于 M。）

M:N 模型中很重要的一点是，Rust 任务不能使用同步系统调用，阻塞任务自身。一旦被阻塞，任务所属的系统本地线程就整个僵化，无法再运行其他 Rus 任务。这意味着 M 个本地线程被（暂时）废掉了一个（但还有 M-1 个本地线程继续工作，因而能够做到 0 死锁）。通过在底层调用异步 I/O 接口（但从用户使用的角度看仍然像同步接口），系统本地线程永远不会阻塞。

libgreen 库没有任何 I/O 实现，仅实现了 Rust 绿色线程的调度器（Schedulers）。真正的 I/O 实现位于 libuv 的封装库 librustuv 中。这么做的目的是希望将来会有不依赖 libuv 的 I/O 实现版本（当然目前还没有）。

使用 libgreen 的一些优势：

- 任务孵化的速度快。在 M:N 模型中，孵化新任务（绿色线程）时可以完全避免系统调用，效率更高。

- 任务切换的速度快。因为上下文切换是在用户空间进行的，所有任务间竞争操作（互斥、管道等）不用执行系统调用，因而速度更快。更高效的上下文切换也会促成更大的吞吐量。

3.2.1、调度器池（Pool of Schedulers）

M:N 线程模型基于以下思路构建：通过 M 个操作系统本地线程（在 libgreen 中被称为 M 个调度器，它们共同组成一个调度器池），调度运行 N 个 Rust 任务（或称之绿色线程）。通过 green::SchedPool 类型可以细粒度地控制调度器池。SchedPool 还是唯一能够孵化新的 M:N 任务的类型。新孵化的任务，跟当前任务一样，平等的从属于同一个调度器池。新任务必然持有调度器池的句柄，用于内部调用以便孵化其他任务。

3.3、选择哪一个？

既然有两个运行时库的实现，显然要做出决定选择使用哪一个。默认情况下，编译器总是链接其中之一。当前默认的运行时库是 libgreen，但是今后默认的运行时库将会是 libnative。

编译器默认选择链接一个运行时库，满足了用户的简单需求。而且这种默认行为不是强制性的，用户还可以自行选择使用哪个运行时库。

例如，这个程序将链接到默认运行时库：

```
fn main() {}
```

然而下面这个程序就是由用户决定明确的链接到特定的运行时库（libgreen）：

```
extern mod green;

#[start]
fn start(argc: int, argv: **u8) -> int {
    green::start(argc, argv, main);
}

fn main() {}
```

两个运行时库 libgreen 和 libnative 都提供了上层的 start 函数，用于在各自运行时中启动初始的第一个 Rust 任务（Task）。

4、运行时在哪里？

运行时的源代码分散在以下多个地方：

- std::rt
- libnative
- libgreen
- librustuv

---------- 全文完 ----------

译者 Liigo 注：这篇文章英文原文[1]多有重复混乱处，而我限于英文水平和 Rust 技术水平，有时也不能完全理解原意。如有翻译不周，敬请谅解，并恳请指正。我的联系方式在本文开头。
[1]: http://static.rust-lang.org/doc/master/guide-runtime.html

本文地址：

CSDN：http://blog.csdn.net/liigo/article/details/19249145
Github：https://github.com/liigo/learn-rust/blob/master/doc/rust-runtime-zh.md
Gdrive：https://docs.google.com/document/d/1uqW9BfPtop9_Nuj0akGRtotsjO0WfkJj3JZ-5gDLAds
Rust 中文圈：https://plus.google.com/+LiigoZhuang/posts/5wAqgkJxq1m
PDF 可打印版：https://github.com/liigo/learn-rust/raw/master/doc/rust-runtime-zh.pdf