1. Introduction

大家好，今天我们来读 tokio 源码

2. syscall

有三个 syscall 非常用要

syscall	简述
clone3	用于创建线程，需要指定线程使用栈空间、指令启动地址等等
futex_wait	判断资源是否被占用，如若占用则休眠否则继续执行
futex_wake	唤醒由于该资源被占用而一直等待的线程

3. Task

1. 运行 tokio hello_world example
2. 观察 tokio hello_world example 的执行流程
3. 观察 tokio 如何分配任务
4. 观察 tokio 是如何实现惰性执行(即等到 await 的时候才执行)

4. Flamegraph

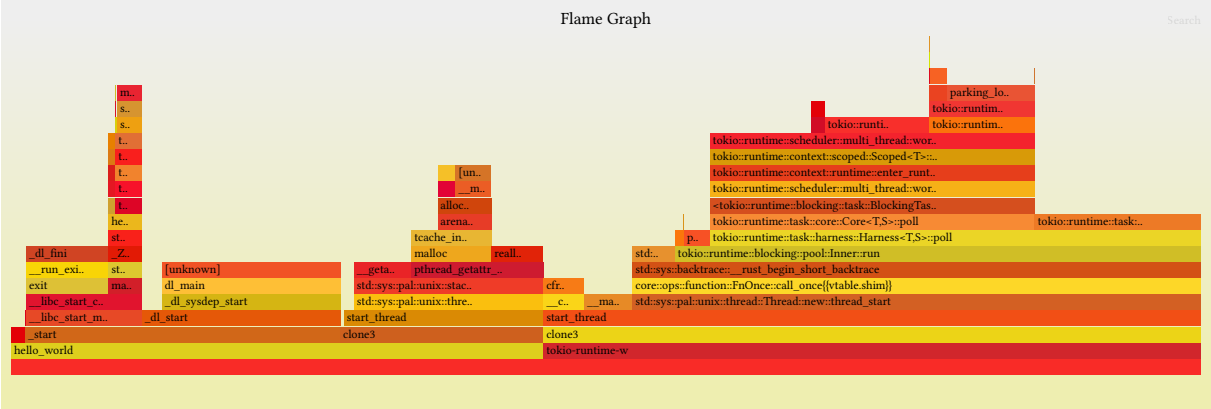

Figure 1: 运行命令 `cargo flamegraph --no-inline --example hello_world` 可得

首先阅读 flamegraph 图了解大概的执行 hello_world 流程，可以观察到 整个程序的运行总时间被分割成了两个部分

1. hello_world
2. tokio-runtime-w

这是因为 hello_world 代表 main 线程，而 tokio-runtime-w 则是独立出去的调度线程，在这个调度线程的基础上再创建线程池，因此可以发现 tokio-runtime-w 中 clone3 的运行次数很多。

1. main 函数首先调用了一次 clone3 生成独立的 tokio-runtime 调度线程
2. tokio-runtime-w 调用多次 clone3 创建线程池
3. 为线程池分配任务

5. Concepts

5.1. Tokio CurrentThread Runtime & MultiThread Runtime

multi_thread @tokio/src/runtime/scheduler/multi_thread/mod.rs @tokio 存在两种 Runtime 模型

Runtime	简述
CurrentThread	单线程调度无并行，但好处是支持!Send 任务，也可以避免线程池创建的开销
MultiThread	多线程调度，支持多核 CPU 的并行性并抢占式调度

5.2. Blocking Pool & Thread Pool

5.3. Builder

@tokio/src/runtime/builder.rs

Builder 设计模式是为了应对当 new 函数中存在大量选项的情况和解耦合的考虑 比如 A 的创建必须通知 B ，那么最好引出一个 Builder 作为中间协调方操作 B，而不是把 B 的引用直接传给 A 的 new 函数。 同时由于 tokio 可以支持不同的调度模型(Section 5.1)，我们也需要考虑 Builder 去静态分发不同的调度模型。

6. Tokio Runtime Init

6.1. Thread Pool Init

1. 初始化首先会生成 cpu 个数的线程(如果没有使用环境变量指定要运行的线程数量)。

而 cpu 个数则通过 sysconf 调用获取。

```rust
fn spawn_thread(
        &self,
        shutdown_tx: shutdown::Sender,
        rt: &Handle,
        id: usize,
    ) -> io::Result<thread::JoinHandle<()>> {
        let mut builder = thread::Builder::new().name((self.inner.thread_name)());

        if let Some(stack_size) = self.inner.stack_size {
            builder = builder.stack_size(stack_size);
        }

        let rt = rt.clone();

        builder.spawn(move || {
            // Only the reference should be moved into the closure
            let _enter = rt.enter();
            rt.inner.blocking_spawner().inner.run(id);
            drop(shutdown_tx);
        })
    }
```

在这里我们可以看