你会Rust就可轻松写出可抗4000万网络请求的Http服务器-Pingora开源了

你会Rust就可轻松写出可抗4000万网络请求的Http服务器-Pingora开源了

大家好，我是梦兽编程。欢迎回来与梦兽编程一起刷Rust的系列。公众号【梦兽编程】即可加入梦兽编程交流群与我交流。

语言只是我们程序员实现途径中的工具，语言的生态丰富会让我们做起事来事半功倍。从今天开始我们开始进入Rust的生态学习。

Http服务开发中`网关`是我们必不缺失的组件之一，nginx的易用性让它成为多个服务代理中间件。虽然nginx的性能已经非常高，但是nginx还是存在一些问题。

1. Nginx使用基于事件的模型和异步非阻塞I/O，这在很大程度上提高了其处理并发请求的能力。但是它的worker进程架构可能导致CPU核心之间的负载不平衡，因为每个请求只能由单个worker处理。一旦worker全都负载一样变得不高效。
2. Nginx在连接重用方面存在局限性，这可能会影响请求的处理效率。当需要维护大量的长连接时，这可能会成为一个性能瓶颈。
3. 虽然Nginx支持模块化，但开发新的模块需要具备C语言编程能力，这对于一些开发者来说可能是一个较高的门槛。C语言本身也存在一些安全问题，如内存泄漏和指针错误。
4. 安全性仍然依赖于第三方库比如OpenSSL，OpenSSL自身安全性漏洞可能会影响到Nginx。

>
>
>
> 一般的开发者，都是配置一下Nginx Config，一旦需要对Nginx进行网关服务的开发头都会大，模块不好写，使用lua脚步加载脚步又影响本身的性能。
>
>
>

随着 `Cloudflare` 业务的快速发展和规模的不断扩大，Nginx 以上的问题已经

满足不了Cloudflare的业务发展定制化功能开发。Cloudflare 需要一个能够提供更高处理速度和更低延迟的解决方案。`Pingora` 采用 Rust 编程语言开发，利用其高性能特性和并发模型，实现了更高的性能要求

Pingora

Pingora提供了多协议支持，包括HTTP/1、HTTP/2、TLS、TCP/UDP等，确保广泛的应用场景兼容性。同时它还支持HTTP/1、HTTP/2、gRPC和WebSocket等端到端代理够满足各种不同的代理需求。

Pingora还提供了可定制的负载均衡功能，允许用户自定义负载均衡策略，包括权重分配、故障转移等。这使得用户可以根据自己的需求来优化负载均衡策略，提高系统的可用性和性能。

安全性方面，Pingora集成了OpenSSL和BoringSSL库，提供FIPS标准的安全保障和后量子加密技术。这使得它能够为用户提供高度的安全保障，特别适合于对安全性要求极高的金融服务、电子商务等应用。

最重要的一点：只要你会Rust你就可以轻松在Pingora进行二次开发和日志监控

Pingora提供了强大的监控与日志等监控工具，让开发者可以实时了解系统的运行状态，快速发现和解决问题。

快速入门

我们需要先添加pingora依赖，注意需要指定 `async-trait` 的版本号，Cargo会默认使用最新的兼容版本。pingora作为中间件的一种建议直接开新项目使用它，而不是在原有的服务添加pingora功能。

```
async-trait="0.1"
pingora = { version = "0.1", features = [ "lb" ] }
```

创建 `pingora` 服务器这将编译并运行，但它不会做任何有趣的事情。

```
use async_trait::async_trait;
use pingora::prelude::*;
use std::sync::Arc;

fn main() {
    let mut my_server = Server::new(None).unwrap();
    my_server.bootstrap();
    my_server.run_forever();
}
```

「创建负载均衡器代理」

接下来，让我们来创建一个负载均衡器。该负载均衡器持有一个静态的上游IP列表。框架提供了常见选择算法，如轮询和哈希。我们可以直接使用。我们还可以在函数pingora-load-balancingLoadBalancer中自定义复杂的选择逻辑。

```
pub struct LB(Arc<LoadBalancer<RoundRobin>>);
```

为了让服务器成为一个代理，需要实现 `ProxyHttp` trait，`ProxyHttp` trait 中只有一个必需的方法是 `upstream_peer()`。这个方法返回请求应该被代理到的目标地址。在 `upstream_peer()` 方法的实现中，使用 `LoadBalancer` 的 `select()` 方法来实现轮询算法，在上游 IP 列表中选择一个目标。

```
#[async_trait]
impl ProxyHttp for LB {

    /// 对于这个小型的例子来说,我们不需要使用上下文存储
(context storage)。
    type CTX = ();
    fn new_ctx(&self) -> () {
        ()
```

```
    }

    async fn upstream_peer(&self, _session: &mut Session, _ctx: &mut ())
-> Result<Box<HttpPeer>> {
        let upstream = self.0
            .select(b"", 256) // 对于轮询(round-robin)这种负载均衡算法来说
,哈希(hash)是不重要的。
            .unwrap();

        println!("upstream peer is: {upstream:?}");

        // Set SNI to one.one.one.one
        let peer = Box::new(HttpPeer::new(upstream, true, "one.one.one.on
e".to_string()));
        Ok(peer)
    }
}

```

如果你想在代理的服务器中进行一些预处理，比如修改`Host` ，我们可以实现`upstream_request_filter` 方法。

```
impl ProxyHttp for LB {
    // ...
    async fn upstream_request_filter(
        &self,
        _session: &mut Session,
        upstream_request: &mut RequestHeader,
        _ctx: &mut Self::CTX,
    ) -> Result<()> {
        upstream_request.insert_header("Host", "one.one.one.one").unwra
p();
        Ok(())
    }
}

```

创建Http代理

`pingora-proxy` 代理服务会将收到的 HTTP 请求代理到之前配置好的后端服务器。在示例中，我们创建了一个 LB 负载均衡器实例，其中配置了两个后端服务器：`1.1.1.1:443` 和 `1.0.0.1:443`。

```
fn main() {
    let mut my_server = Server::new(None).unwrap();
    my_server.bootstrap();

    let upstreams =
        LoadBalancer::try_from_iter(["1.1.1.1:443", "1.0.0.1:443"]).unwrap()
;
  /// 将这个 LB 负载均衡器实例传递给 http_proxy_service() 函数,创建了一
个代理服务。
    let mut lb = http_proxy_service(&my_server.configuration, LB(Arc::new
(upstreams)));
        lb.add_tcp("0.0.0.0:6188");
    my_server.add_service(lb);

    my_server.run_forever();
}

```

不可用服务处理
---

1. 为每个后端服务器添加一个健康检查逻辑，定期检查它们是否可用。
2. 在选择后端服务器时，只选择那些通过健康检查的服务器。
3. 如果某个服务器失败了健康检查，就将它从可用列表中移除，不再将请求路由到它。

这样做可以确保我们的负载均衡器只将请求转发到可用的后端服务器，从而避免出现 `502: Bad Gateway` 的错误。

```
fn main() {
    let mut my_server = Server::new(None).unwrap();
    my_server.bootstrap();

    // Note that upstreams needs to be declared as `mut` now
    let mut upstreams =
        LoadBalancer::try_from_iter(["1.1.1.1:443", "1.0.0.1:443", "127.0.0.
1:343"]).unwrap();
```

```
  /// 心跳监测
    let hc = TcpHealthCheck::new();
    upstreams.set_health_check(hc);
    /// 每秒监测一次服务存活状态
    upstreams.health_check_frequency = Some(std::time::Duration::from_secs(1));

    let background = background_service("health check", upstreams);
    let upstreams = background.task();

    // `upstreams` no longer need to be wrapped in an arc
    // 不需要使用 Arc 来实现 upstreams 的共享和线程安全。主要是为了提高性能
    let mut lb = http_proxy_service(&my_server.configuration, LB(upstreams));
    lb.add_tcp("0.0.0.0:6188");

    my_server.add_service(background);

    my_server.add_service(lb);
    my_server.run_forever();
}

```

启动服务

一般这种服务器，我们都需要让它运行在后台中。

```
cargo run -- -d //即可实现类似linux nohup的功能
```

结果

通过使用 Pingora，该视频平台能够为全球用户提供快速、安全、稳定的视频流服务。Pingora 的高性能和可扩展性确保了即使在高峰时段，用户也能享受到高质量的视频观看体验。

如果您对Rust异步编程感兴趣，欢迎我的公众号【梦兽编程】，一起探索Rust的奥秘。如果您觉得这篇文章对您有帮助，请分享给更多需要的朋友。您

的转发是我最大的动力！

原文链接: https://juejin.cn/post/7353536741615714315