为什么高手都要用非阻塞IO?

萤火架构 2024-06-19 745 阅读9分钟 专栏：编程思想

智能总结 复制 重新生成

本文主要探讨了非阻塞 I/O 相关内容。介绍了传统阻塞 I/O 的问题，如高并发场景下内存占用大、上下文切换频繁影响性能。阐述了非阻塞 I/O 的原理，包括 Java NIO、Python asyncio 等多种语言的实现模型，并指出其设计共性，如多路复用、协程、事件驱动等。还讨论了非阻塞 I/O 对系统整体性能的影响，强调其在高并发场景中的优势及重要性。

关联问题：非阻塞IO适用场景？ 如何优化阻塞IO？ 非阻塞IO有何局限？

基于该文章内容继续向AI提问

IO全称为输入/输出(Input/Output，正规简称I/O)，指的是计算机系统与外部设备之间的数据交换。外部设备包括输入设备（如鼠标、键盘等）、输出设备（如显示器）、存储设备（如硬盘）和网络设备等。

传统的I/O操作是阻塞的，这意味着当一个I/O操作进行时，当前的执行体会被挂起，进入等待状态，直到I/O结果返回，执行体才会继续处理后续的逻辑。这就像你去图书馆前台借一本非常热门的书，但书已经被借出。为了得到这本书，你选择站在前台等待，直到这本书被归还。等待的过程中，你不能去干其它任何事情。

非阻塞I/O更像是这样：你去借那本热门书籍，但被告知现在没有。这时，你留下联系方式并告诉图书管理员，一旦书归还，请通知你。然后你可以自由地去参加其他活动。

在借书的这个例子中，你不用浪费大量的时间在等待上，同样的时间你可以做更多的事，可以说，非阻塞I/O极大的提高了系统运行效率。另外还有很多同学说非阻塞IO快，阻塞IO慢，真的是这样吗?

本文，我们将深入探讨阻塞I/O遇到的问题，非阻塞I/O的原理、优势及其实现方法，帮助大家更好地理解和应用这一技术。

阻塞IO的真正问题

阻塞IO为什么被诟病?

在高并发场景下，如果使用阻塞I/O模型，每个请求都需要创建一个新的线程来处理。当这些请求中有大量操作处于I/O等待状态时，虽然CPU能够切换到其他任务继续执行，但创建和管理大量线程本身也会消耗系统资源，包括内存和用于线程上下文切换的CPU时间，从而影响系统的整体性能和可扩展性。

- **内存占用**：在Windows系统中默认每个线程分配1M的内存，在Linux系统中默认分配8M，假设我们的计算机有8G的内存，那么系统最多也只能创建几千个线程，这个数量级显然无法满足高并发场景下处理数十万甚至上百万并发连接的需求。具体来说，如果按照Linux系统默认每个线程分配8M内存来计算，8G内存理论上最多能支持约1000个线程（8GB / 8MB = 1024），但实际上，系统还需要保留内存给其他进程使用，因此可用线程数会更少。
- **上下文切换**：当操作系统从一个线程切换到另一个线程时，需要保存当前线程的状态（如寄存器内容）并加载新线程的状态，这个过程涉及多次内存读写操作，会占用CPU周期且可能导致延迟。在高并发场景下，阻塞IO会导致大量的线程产生，从而导致频繁的线程切换，而频繁的线程上下文切换会显著降低系统效率。

非阻塞IO的基本原理

萤火架构 LV.6
程序员、AI探索者
榜上有名 优秀作者
238 文章 1.1m 阅读 2.3k 粉丝
关注 私信

相关推荐
日活3kw下，如何应对实际业务场景中S...
1.6k阅读 · 26点赞
四大集合20连问，抗住！
565阅读 · 19点赞
"策略模式：改变你的思维，赋予你的代...
339阅读 · 7点赞
记一次由于Nacos频繁GC导致的Java内...
1.7k阅读 · 15点赞
Java生产环境下问题排查
853阅读 · 11点赞

精选内容
【设计模式】【结构型模式】装饰者模式...
flzjkl · 11阅读 · 1点赞
【设计模式】【结构型模式】适配器模式...
flzjkl · 27阅读 · 1点赞
Linux nftables 命令使用详解
唐青枫 · 14阅读 · 1点赞
spring.factories使用
考虑考虑 · 35阅读 · 0点赞
【设计模式】【行为型模式】解释器模式...
flzjkl · 12阅读 · 1点赞

找对属于你的技术圈子
回复「进群」加入官方微信群

什么是非阻塞IO?

正如上面借书的例子，当IO操作发生时，我们无需等待，可以去干别的事，只有IO操作返回时，我们才需要处理IO返回的结果，这就是非阻塞IO的本质。

非阻塞IO可以解决阻塞IO的内存占用过大和上下文切换频繁问题，下边我将介绍几个典型的非阻塞IO模型，方便大家理解其中的原理。

Java NIO

Java NIO（New I/O）引入了非阻塞I/O机制，通过Channel和Buffer来处理数据，使用Selector来管理多个Channel。

- **Channel**：Channel是数据传输的通道，可以进行非阻塞的读写操作。
- **Buffer**：Buffer是数据的容器，用于读写数据。Buffer直接管理一块操作系统内存，减少了数据拷贝，支持多种数据类型，读写更方便、效率更高。
- **Selector**：Selector是多路复用器，允许一个线程同时监控多个Channel的状态（如读、写、连接等），从而实现非阻塞I/O。

java 代码解读 复制代码

```java
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.net.InetSocketAddress;
import java.util.Iterator;

public class NIOServer {
    public static void main(String[] args) throws IOException {
        Selector selector = Selector.open();
        ServerSocketChannel serverSocket = ServerSocketChannel.open();
        serverSocket.bind(new InetSocketAddress("localhost", 8080));
        serverSocket.configureBlocking(false);
        serverSocket.register(selector, SelectionKey.OP_ACCEPT);

        while (true) {
            selector.select();
            Iterator<SelectionKey> keys = selector.selectedKeys().iterator();
            while (keys.hasNext()) {
                SelectionKey key = keys.next();
                keys.remove();
                if (key.isAcceptable()) {
                    SocketChannel client = serverSocket.accept();
                    client.configureBlocking(false);
                    client.register(selector, SelectionKey.OP_READ);
                } else if (key.isReadable()) {
                    SocketChannel client = (SocketChannel) key.channel();
                    ByteBuffer buffer = ByteBuffer.allocate(256);
                    client.read(buffer);
                    System.out.println("Received: " + new String(buffer.array()).trim());
                }
            }
        }
    }
}
```

Python asyncio

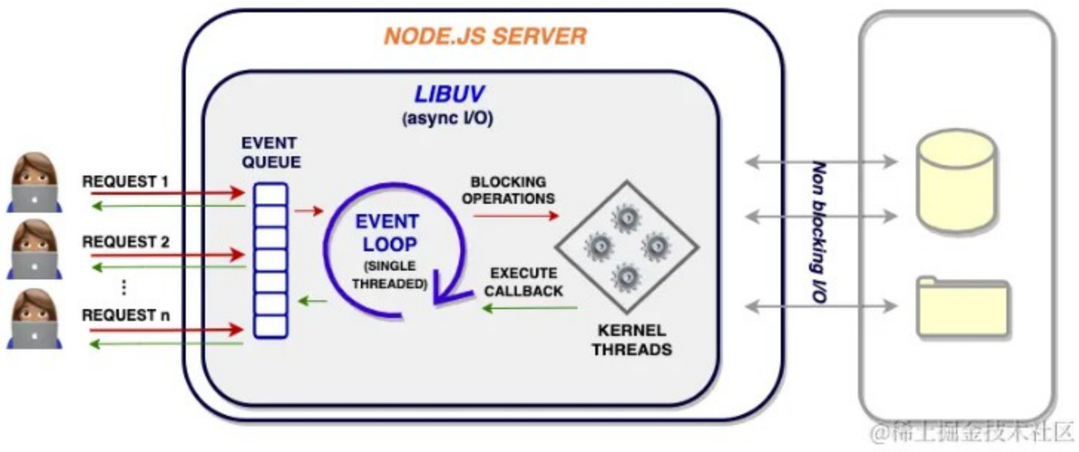
asyncio是Python标准库中的一个库，提供了异步I/O支持。它基于事件循环（event loop），可以调度和执行异步任务（coroutines）。

- **事件循环**：事件循环是asyncio的核心，负责调度和执行异步任务。事件循环有点类似Java NIO中的Select，不过事件循环是更高级的抽象，除了通过操作系统的多路复用机制调度IO，它还调度协程、定时器和信号处理器。
- **协程**：协程是比线程更小的程序执行单位，更小的内存分配，更短的切换时间，是编程语言在用户态维护管理的。协程是使用async/await关键字定义的函数，可以在等待I/O操作时挂起并让出控制权，从而实现并发。

python 代码解读 复制代码

```python
import asyncio

async def handle_client(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    print(f"Received: {message}")

    writer.write(data)
    await writer.drain()
    writer.close()

async def main():
    server = await asyncio.start_server(handle_client, '127.0.0.1', 8888)
    async with server:
        await server.serve_forever()

asyncio.run(main())
```

Node.js的事件驱动模型



Node.js使用事件驱动模型和非阻塞I/O操作，基于libuv库实现。libuv是一个跨平台的异步I/O库，封装了不同操作系统的I/O多路复用机制（如epoll、kqueue、IOCP等）。

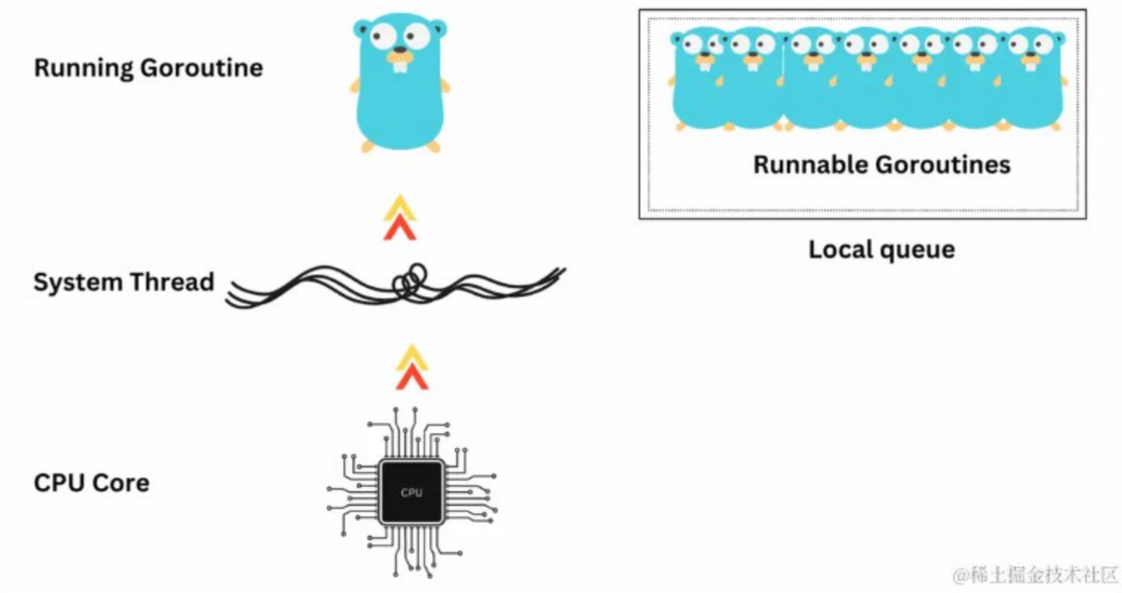
- **事件循环**：与Python asyncio的事件循环类似，不过Node.js底层基于libuv，侧重的是IO操作，通过检查事件队列，调用相应的回调函数来处理事件。Node.js 的事件循环主要针对构建高并发的网络应用，特别是I/O密集型任务，如Web服务器、API服务器等。它利用了非阻塞I/O和事件驱动模型，允许在单线程中处理大量并发连接。
- **回调函数**：回调函数是处理异步操作的主要方式，当I/O操作完成时，调用相应的回调函数。
- **Promise**：Promise是一种异步编程的模式，用于处理异步操作的结果。
- **async/await**：async/await是基于Promise的语法糖，使得异步代码看起来更像同步代码。

```ini
```

代码解读 复制代码

```
const http = require('http');

const server = http.createServer((req, res) => {
    if (req.method === 'GET') {
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end('Hello, World!\n');
    }
});

server.listen(8080, '127.0.0.1', () => {
    console.log('Server running at http://127.0.0.1:8080/');
});
```

Go语言的goroutine



Go语言通过goroutine和channel提供了轻量级的并发支持。goroutine是Go语言中的轻量级线程，或者也叫协程，通过channel进行通信和同步。select语句用于监听多个channel的操作，实现非阻塞I/O。

- goroutine：goroutine是Go语言中的协程，可以并发执行函数。协程的内存占用更小，使用的操作系统线程更少。
- channel：channel用于在goroutine之间传递消息，实现同步和通信。
- select：select语句用于监听多个channel的操作，可以实现非阻塞I/O。

```go
```

代码解读 复制代码

```
package main

import (
    "fmt"
    "net"
    "bufio"
)

func handleConnection(conn net.Conn) {
    defer conn.Close()
    reader := bufio.NewReader(conn)
    for {
        message, _ := reader.ReadString('\n')
        fmt.Printf("Received: %s", message)
        conn.Write([]byte(message))
    }
}

func main() {
    listener, _ := net.Listen("tcp", "localhost:8080")
    defer listener.Close()
    for {
        conn, _ := listener.Accept()
        go handleConnection(conn)
    }
}
```

非阻塞I/O的设计共性

- **多路复用**：非阻塞IO都直接或间接使用了多路复用，这是一种高效的I/O处理技术，允许一个线程同时监控多个I/O通道。常见的多路复用机制有epoll（Linux）、kqueue（BSD）、IOCP（Windows）等。
- **协程**：尽管一些语言没有明确的提出这一概念，但都蕴含了协程的思想。协程是一种轻量级的并发处理机制，可以暂停和恢复执行，内存占用更小，切换成本更低，运行在用户态，比传统线程更高效。协程通过非阻塞I/O操作和事件循环实现并发处理。
- **事件驱动**：事件驱动是一种编程模式，通过事件通知机制来处理I/O操作的完成。程序不主动等待I/O操作，而是注册一个事件，当I/O操作完成时，事件触发相应的处理程序。触发形式可能是简单的回调，也可能是复杂的执行体（线程、协程等）调度。

非阻塞IO更快吗?

对于单次IO，从发起到收到响应，其中主要有三段时间：请求数据从客户端到服务端的传输时间、服务端的处理时间、响应数据从服务端到客户端的返回时间。对于这三段时间，非阻塞IO和阻塞IO都没有任何影响力或者说影响甚小，它们都不会因为使用非阻塞IO而变的更快。

但是非阻塞IO因为更优的内存使用效率，服务器可以支撑更大的并发访问，在繁忙的系统中，如果存在因为内存分配或者线程调度而导致请求接入等待的情况，非阻塞IO一定程度上会降低请求接入的平均时间，从而让服务端的处理更快一些。不过这是非阻塞IO结合协程机制的效果，单纯非阻塞IO没有这个能力。

以上就是本文的主要内容。非阻塞I/O通过更高效的资源利用和更低的线程管理开销，显著提升了系统在高并发场景下的性能和扩展性。尽管它不能直接加快单次I/O操作的速度，但其在整体性能优化方面的优势使其成为现代软件系统中不可或缺的重要部分。掌握非阻塞I/O技术，对于开发高性能、高可扩展性的应用至关重要。希望本文能帮助大家更好地理解和应用这一技术。

关注萤火架构，加速技术提升！

标签： 性能优化　　话题： 金石计划征文活动

本文收录于以下专栏

编程思想 专栏目录

编程的一些套路。

41 订阅 · 27 篇文章

订阅

上一篇 Go语言中的context包到底解决了啥问题?

评论 2

登录 / 注册 即可发布评论!

最热 | 最新

小码编匠 后端工程师（DotNet技术匠）

不错

2月前　1　评论　…

女程小v @字节

因为他们是高手，哈哈

8月前　2　评论　…

为你推荐

select、poll、epoll详解
传达室马大爷 | 3年前 | 2.2k | 1 | 评论 | Java

网络IO模型详解
豆豆酱 | 1年前 | 411 | 点赞 | 评论 | 后端

Linux应用编程基础09-高级I/O
VvUppppp | 3月前 | 48 | 点赞 | 评论 | Linux

🔍【攻破技术盲点】— 网络IO模型的分析（上）
洛神灬殇 | 3年前 | 578 | 5 | 评论 | 架构 | 后端

网络编程：什么是阻塞IO
邓磊的技术笔记 | 7月前 | 51 | 1 | 评论 | Java

浅谈面试常考的I/O模型
一瓶小七酱 | 3年前 | 1.9k | 9 | 2 | 操作系统

Nginx（搭建，工作模式与升级）
lc111 | 1年前 | 451 | 点赞 | 评论 | 后端 | Nginx

UNIX 网络编程定义的5种I/O模型
nsiy | 4年前 | 1.1k | 2 | 评论 | 分布式

（BAT必会知识）Linux网络编程中五种IO模型讲解
九八年的尾巴 | 3年前 | 376 | 2 | 评论 | Netty

五大经典IO模型
Singgle | 1年前 | 678 | 1 | 评论 | 后端 | Linux

通信——IO模型
努力的码农 | 3年前 | 349 | 点赞 | 评论 | Java

Datenlord | 重新思考Rust Async - 如何实现高性能IO
Rust_Magazine | 3年前 | 1.2k | 1 | 评论 | Rust

IO 操作时计算机底层发生了什么?
垂慕容 | 9月前 | 640 | 2 | 评论 | 程序员

IO模型（BIO、NIO、AIO）
TimxYo | 3年前 | 1.1k | 3 | 评论 | Java

Go语言中的缓冲区及其在fmt包中的应用
沙蒿同学 | 1年前 | 417 | 1 | 评论 | 后端 | Go