多线程 web 服务器开发过程文档

软件学院 2019141460404 王泽远

开发过程

为什么使用 C++?

本项目使用 C++而不是 Python 开发，主要出于一下几点原因：

1. 本人最近在学习 C++，想借这个项目练手
2. 相较于 Python 的高封装，C++暴露了更多的底层细节，更加考验编程者对底层本质的了解把握程度
3. C++的性能比 Python 高

综上，除了性能原因外，使用 C++也是本人对自己提出的挑战。

对 C sytle socket 进行封装

C++标准中没有提供网络相关的库函数，因此项目开发使用的 socket 是 linux 下的 C 语言实现。不过为了方便后续使用以及利用 C++的 RAII 特性，可以先对 C sytle 的 socket 进行 C++ sytle 风格的包装。源码中的 makalo_socket.cpp 和 makalo_socket.h 实现了对 socket 的封装。其中最重要的一点就是利用了 RAII 和引用计数技术。

RAII 即在 Socket 类的析构函数中调用 close 函数关闭套接字，使得套接字可以在 Socket 对象生命周期函数后自动 close 释放。

引用计数允许 Socket 在按值传递的同时传递 socket 的所有权，以防止旧的 Socket 实例被析构时过早关闭 socket。

具体请参考 src/core/makalo_socket.cpp,include/core/makalo_socket.h

另外对于 C sytle 的介绍本人已整理成文档，其中内容大多来自《UNIX 网络编程卷一》一书。

线程池

为什么使用线程池?

对于多线程服务器来说，每个客户的连接都是由一个单独的线程处理的。一种简单的想法是为每一个新客户连接创建一个新线程。然而这种朴素的做法至少有两个问题：

1. 操作系统创建一个新线程是相当耗时的操作。当有大量新客户到来时现场逐个分配线程可能导致系统并发度降低。
2. 线程的运行，调度是要占据系统资源的，而系统的资源总量是有限的。因此，盲目的开启大量线程可能会导致系统将大量性能浪费在线程间调度上，使性能不升反降。更不用说防范恶意程序开启大量访问对服务器进行攻击，引发服务器崩溃。

使用线程池就可以解决上述问题。线程池中的线程都是事先分配好的，不需要新连接到来时现场分配。另外，线程池也可以限制开启的最大线程数量，使系统能够维持在合理的线程调度状态。

线程池实现

线程池的重要组成部分有两个：

1. 一个线程数组：存放所有的线程
2. 一个任务队列：存放待执行的任务

所谓任务就是一个函数和其参数打包形成的结构体。

在线程池初始化时，所有的线程都被构造出来，之后每个线程就进入等待状态直到有新的任务被放进任务队列。

当调用线程池的 execute 方法是时，一个新任务就被传入线程池，空闲的线程开始争抢新来的任务，抢到任务的线程开始执行任务，没有抢到任务的线程则继续进入等待。如果当前没有空闲线程，则任务暂存在队列中。当然这一切对线程池调用者来说都是透明的，调用者只知道传入的任务最终一定会被执行。

具体代码请参考：include/core/threadpool.h

HTTP 处理实现

HTTP 是应用层协议，而 socket 只提供了到传输层的封装。因此程序员必须自行实现对 HTTP 的处理。

对于 HTTP 服务器来说，重要的任务有两个：

1. 读取并解析客户端发送的 HTTP 请求
2. 封装并发送 HTTP 响应

相应的处理逻辑代码在 webserver/src/http/http.cpp 中

HTTP 请求

一个典型的 HTTP 请求报文格式如下：

由于本项目实现的 web 服务器考虑比较简单，着重实现了 GET 方法请求服务器资源的功能，因此着重关心第一行的请求方法和 URL 字段。

在调用了 socket 的 read 方法后可以获得客户端发送的请求报文。之后解析请求行，如果是以 GET 开头，则按照 URL 查找资源文件，如果找到即返回文件内容，如果找不到则返回 404 not found。如果请求方法不是 GET 则返回 400 Bad Request。

HTTP 响应

一个典型的 HTTP GET 方法响应报文格式如下：

GET 方法的 HTTP 响应报文格式

HTTP/1.1	空格	200	空格	OK	\r	\n
Content-Type	:	text/html		\r		\n
...						
Content-Encoding	:	gzip		\r		\n
\r		\n				
省略						

在 webserver/src/http/http_response.cpp 中定义了如下信息：

status：

200 OK

404 Not Found

301 Moved Permanently

400 Bad Request

403 Forbidden

content_type:

text/plain; charset=UTF-8

text/html; charset=UTF-8

image/gif

image/jpeg

image/png

application/json

application/pdf

connection:

Keep-Alive

Close

服务器填充好响应的 HTTP 信息后将其封装成字节流调用 socket.write 方法写回客户端

服务器实现

在有了上述组件的支持后，服务器的实现就相对容易了。

先新建一个服务器框架类，框架类业务流程是开启一个单独的线程在后台 accept 监听客户的连接请求，一旦建立新连接就将已连接套接字连同处理函数打包成任务传给线程池后台处 1 理，然后立即返回监听过程。

具体代码参考 webserver/src/server/server.cpp

HTTP 就是将 HTTP 处理函数当作 server 框架的已连接套接字处理函数。

测试结果

本人将服务器部署在了同意子网下的一台树梅派 4B 上（子网 ip 地址 192.168.3.17，端口号 8000），然后进行了多种访问测试。

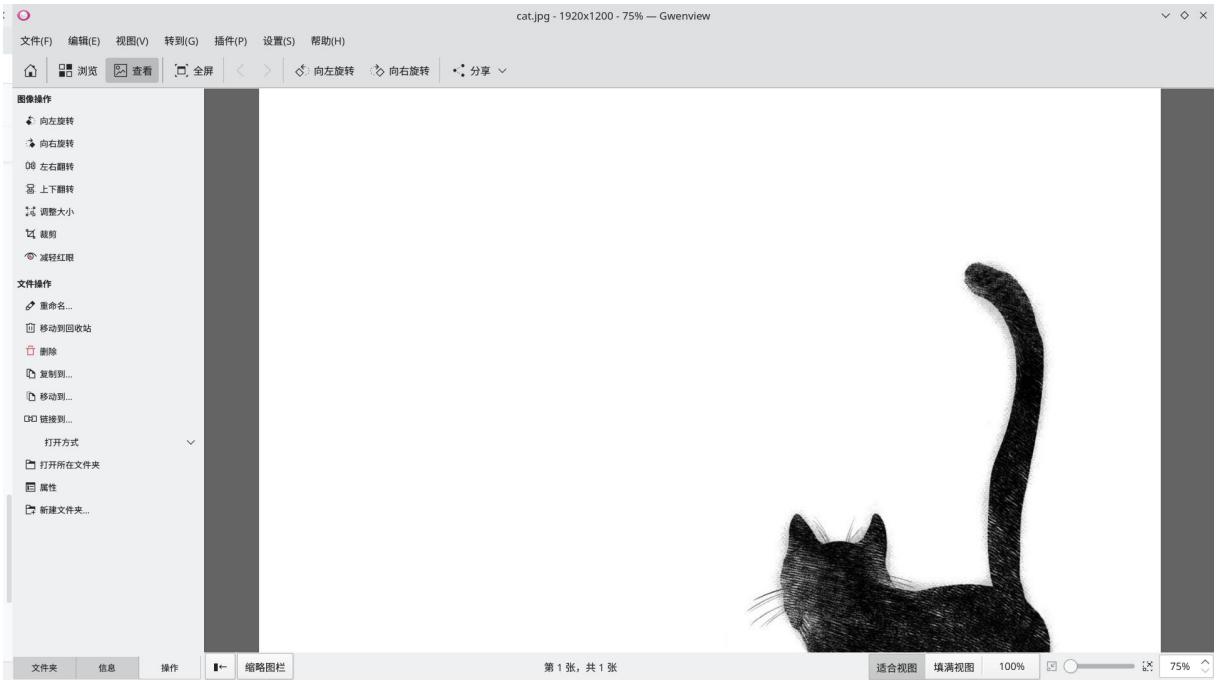
多种客户端多种文件格式访问测试

命令行 curl 工具

```
# makalo @ makalo-15z990vaa52c in ~ [17:19:34]
$ curl 192.168.3.17:8000/hello.txt
Hellooooooooooooooooooooooooo!
Nice to meet yooooooooooooooooooooooou!
```

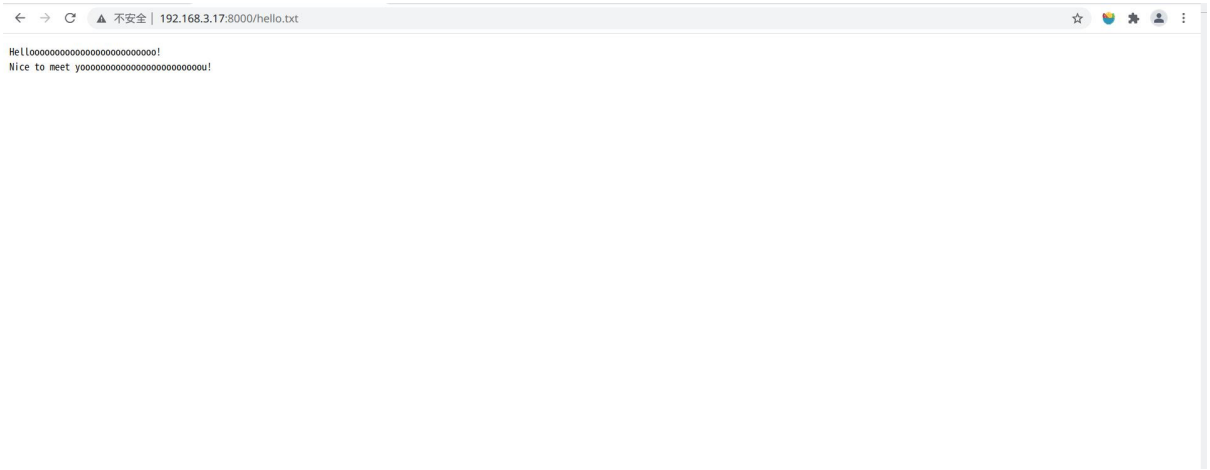
curl 请求文本文件

```
# makalo @ makalo-15z990vaa52c in ~ [17:19:34]
$ curl 192.168.3.17:8000/hello.txt
Hellooooooooooooooooooooooooo!
Nice to meet yooooooooooooooooooooooou!
```



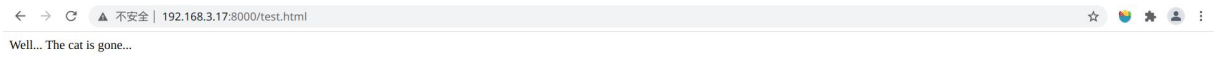
curl 命令下载图片

linux 平台下 chrome 浏览器



请求文本文件

请求 html 文件

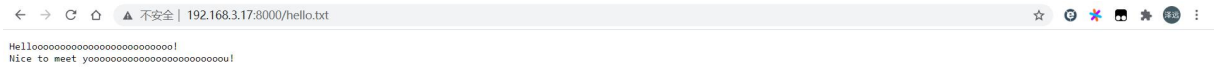


超链接测试

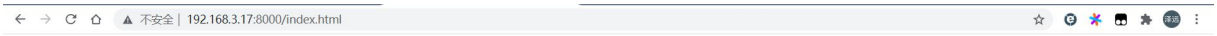
404 Not Found

请求不存在对象

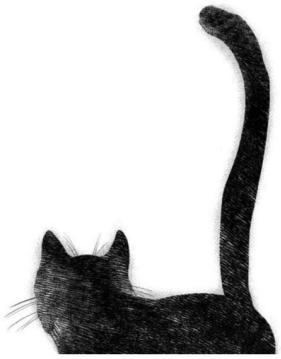
windows 平台下 chrome 浏览器



请求文本文件



look there is a cute cat!



click there to try hyperlink

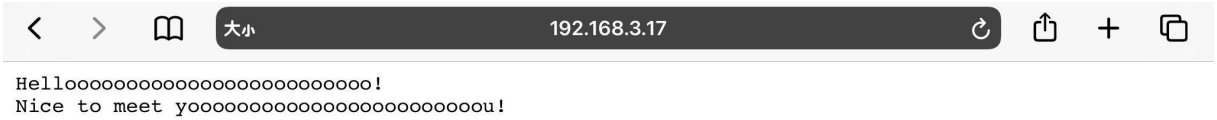
请求 html 文件

Well... The cat is gone...

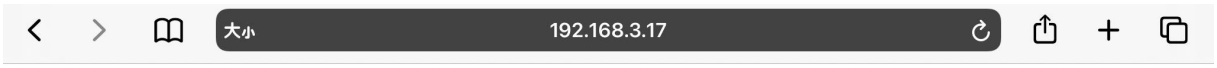
超链接测试

请求不存在对象

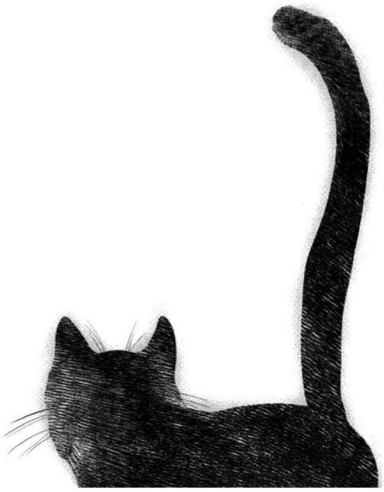
ipadOS 下 safari 浏览器



请求文本文件



look there is a cute cat!

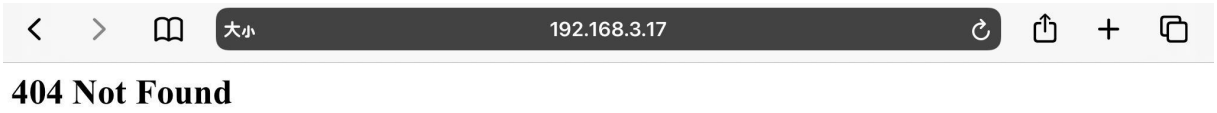


click there to try hyperlink

请求 html 文件

Well... The cat is gone...

超链接测试



404 Not Found

请求不存在对象

性能测试

测试方法

使用 wrk 工具进行压力测试。

wrk 是一个比较先进的 HTTP 压力测试工具，当在单个多核 CPU 上运行时，能够产生大量负载。

在测试设备命令行中运行：

```
# makalo @ makalo-15z990vaa52c in ~ [17:44:33]
$ wrk -t8 -c200 -d10s http://192.168.3.17:8000/cat.jpg
```

参数解释：

-c200：启动 200 个连接

-t8: 开启 8 个线程做压力测试

-d10: 压测持续 10s

请求对象为 cat.jpg

压力测试设备硬件信息：

Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz x 8
8GB Memory

服务器设备硬件信息：

树梅派 4B

1.5GHz quad-core 64-bit ARM Cortex-A72 CPU x 4

4GB Memory

性能测试结果

当线程池大小为 1：

```
# makalo @ makalo-15z990vaa52c in ~ [17:56:42]
$ wrk -t8 -c200 -d10s http://192.168.3.17:8000/cat.jpg
Running 10s test @ http://192.168.3.17:8000/cat.jpg
  8 threads and 200 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    34.96ms   24.59ms 122.47ms   80.16%
    Req/Sec   115.31     18.09   140.00     89.00%
  1154 requests in 10.07s, 108.24MB read
Requests/sec:    114.59
Transfer/sec:     10.75MB
```

当线程池大小设置为 4：

```
# makalo @ makalo-15z990vaa52c in ~ [17:44:33]
$ wrk -t8 -c200 -d10s http://192.168.3.17:8000/cat.jpg
Running 10s test @ http://192.168.3.17:8000/cat.jpg
  8 threads and 200 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    35.19ms   40.83ms 344.51ms   87.25%
    Req/Sec   114.64     18.60   141.00     92.00%
  1145 requests in 10.10s, 107.47MB read
Requests/sec:    113.38
Transfer/sec:     10.64MB
```

当线程池大小设置为 8：

```
# makalo @ makalo-15z990vaa52c in ~ [17:46:38]
$ wrk -t8 -c200 -d10s http://192.168.3.17:8000/cat.jpg
Running 10s test @ http://192.168.3.17:8000/cat.jpg
  8 threads and 200 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    34.76ms   37.59ms 337.00ms   90.14%
    Req/Sec   115.37     14.23   141.00     66.00%
  1156 requests in 10.06s, 108.56MB read
Requests/sec:    114.96
Transfer/sec:     10.80MB
```

当线程池大小设置为 16：

```
# makalo @ makalo-15z990vaa52c in ~ [17:49:33]
$ wrk -t8 -c200 -d10s http://192.168.3.17:8000/cat.jpg
Running 10s test @ http://192.168.3.17:8000/cat.jpg
  8 threads and 200 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    37.24ms   32.62ms 345.99ms   93.63%
    Req/Sec   108.33     25.00   141.00     77.00%
  1083 requests in 10.10s, 101.63MB read
Requests/sec:    107.25
Transfer/sec:     10.06MB
```

当线程池大小设置为 32：

```
# makalo @ makalo-15z990vaa52c in ~ [17:49:44]
$ wrk -t8 -c200 -d10s http://192.168.3.17:8000/cat.jpg
Running 10s test @ http://192.168.3.17:8000/cat.jpg
  8 threads and 200 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    35.19ms   37.40ms 243.13ms   86.36%
    Req/Sec   114.24     17.32   141.00     90.00%
  1144 requests in 10.07s, 107.39MB read
Requests/sec:    113.60
Transfer/sec:     10.66MB
```

分析

当线程池数量为 1 时，服务器中存在两个线程，一个负责监听，一个负责处理连接。可以看出服务器的性能并不是随着开启的线程数量增大而增多，其实多线程的服务器性能分析是个相当复杂的课题，其受线程数量，IO 速度，网速等诸多因素的影响。

发现的问题及解决办法

请求方法响应问题

目前服务器已经支持了不带参数的 GET 方法，即普通的资源获取请求。实际上，HTTP 请求方法至少还有 POST, DELETE, PUT 三种，以及携带参数的对应方法。对于这些方法的支持不但要考虑请求方法和 URL，还要解析请求体以获取相应的参数。

KEEP-ALIVE 问题

在一开始的版本中，我对请求的处理方式很简单——用 read 函数读请求->解析请求->相应->关闭连接。但这种处理方式显然是不对的。一般情况下浏览器发出的请求的 connection 参数都是 keep-alive，即连接应该保持，服务器不能私自关闭连接。具体表现是浏览器在接收到服务器的响应后没有立即渲染，直到服务器关闭后才开始渲染结果。

所以在之后我将请求处理过程改为一个循环，直到对方明确发送 FIN 才跳出循环关闭连接。并且我填充了响应体的 connection 字段，使得上述问题得以解决。

文件读取格式问题

服务器解析 GET 请求中的 URL 并读取对应文件的 URL。在一开始我没有按二进制读取文件内容，导致文件测试没有问题，但图片测试发生错误。我立即想到这一原因可能是文件打开方式造成的，于是我立即改正了文件为二进制打开。

光是这样还不够。客户端收到一堆二进制数据，如何知道其解读格式呢？所以要在响应体的 content-type 字段填充相应的格式，使接收方能够正确解读。

其他细节问题

编码规范问题：

404 400 等错误页面的 html 被写死在代码里，实际上应该独立写进一个文件中以便自定义样式。

心得体会

使用 C++语言确实给编程工作带来了不少的麻烦，许多在 python 当中轻而易举甚至是理所当然的功能在 C++当中都可能需要耗费一天的时间造轮子。如果当初用 python 语言的话，相比现在服务器的功能还会更加丰富完善吧。

这一过程是痛苦并快乐的。网络知识不但有用而且有趣。正是计算机网络的诞生使得数据可以在多台设备中轻松共享，而不再只是限制在用一台机器的内存和硬盘当中。这使得分布式计算成为可能，即一台设备将需要计算的参数通过网络发送给另一台性能更高的设备，待运算结果完成后再通过网络返回。实际上 web server 也属于上述模型的一种特例，而我对 socket 的封装也是我之前研究 RPC 实现时完成的。

当然得益于前辈们的工作如今传输层一下的复杂细节都已经被屏蔽掉了，我们要做的也只不过是调用相对简单的传输层接口完成应用层实现。将来有机会的话传输层一下的具体代码实现也是一块值得深度挖掘的宝藏。