ZLToolKit笔记4：域名解析

2024年12月20日　20:20

DNS域名详细解析过程(最全面，看这一篇就够) dns解析-CSDN博客

前驱知识：域名解析

第一步：

本地电脑检查浏览器缓存中国有没有这个域名对应的解析过的IP地址；
但浏览器缓存域名受限制，大小、时间。

第二步：

浏览器缓存中没有数据，则浏览器会查找操作系统缓存中是否有这个域名对应的DNS解析结果。操作系统也有一个域名解析的过程，在Linux中可以通过/etc/hosts文件来设置，而在windows中可以通过配置C:\Windows\System32\drivers\etc\hosts文件来设置，用户可以将任何域名解析到任何能够访问的IP地址。

第三步：

前两个过程无法解析时，就需要用到我们网络配置中的“DNS服务器地址”了。操作系统会把这个域名发送给这个本地DNS服务器。大约80%的域名解析到这里就结束了，后续的DNS迭代和递归也是由本地DNS服务器负责。

第四步：

如果本地DNS服务器仍没有命中，就直接到根DNS服务器请求解析。

第五步：

根DNS服务器返回给本地域名服务器一个顶级DNS服务器地址，它是国际顶级域名服务器，
如.com、.cn、.org等，全球只有13台左右。

第六步：

本地DNS服务器再向上一步获得的顶级DNS服务器发送解析请求。

第七步：

接收请求的顶级DNS服务器查找并返回此域名对应的Name Server域名服务器的地址，这个Name Server服务器就是我要访问的网站域名提供商的服务器，起始该域名的解析任务就是由域名提供商的服务器来完成的。

第八步：

Name Server服务器会查询存储的域名和IP的映射关系表，再把查询出来的域名和IP地址等等信息，连同一个TTL值返回给本地DNS服务器。

第九步：

返回该域名对应的IP和TTL值，本地DNS服务器会缓存这个域名和IP的对应关系，缓存时间由TTL值控制。

第十步：

把解析的结果返回给本地电脑，本地电脑根据TTL值缓存在本地系统缓存中，域名解析过程结束在实际的DNS解析过程中，可能还不止这10步，如Name Server可能有很多级，或者有一个GTM来负载均衡控制，这都有可能会影响域名解析过程。

ZLToolKit中域名解析

```
    if (isIP(url.data())) {
        auto fd = SockUtil::connect(url.data(), port, true, local_ip.data(), local_port);
        (*async_con_cb)(fd == -1 ? nullptr : std::make_shared<SockNum>(fd, SockNum::Sock_TCP));
    } else {
        auto poller = _poller;
        weak_ptr<function<void(const SockNum::Ptr &)>> weak_task = async_con_cb;
        WorkThreadPool::Instance().getExecutor()->async([url, port, local_ip, local_port, weak_task,
poller]() {
            // 阻塞式dns解析放在后台线程执行  [AUTO-TRANSLATED:e54694ea]
            //Blocking DNS resolution is executed in the background thread
            int fd = SockUtil::connect(url.data(), port, true, local_ip.data(), local_port);
            auto sock = fd == -1 ? nullptr : std::make_shared<SockNum>(fd, SockNum::Sock_TCP);
            poller->async([sock, weak_task]() {
                if (auto strong_task = weak_task.lock()) {
                    (*strong_task)(sock);
                }
            });
        });
        _async_con_cb = async_con_cb;
    }
```

当url不是IP地址时，需要进行DNS解析，为了不阻塞当前线程，这里**将DNS解析的任务放到一个后台线程池中异步执行。**

解析的过程仍然在SockUtil::connect函数中进行（进行DNS解析并建立连接）。

一定要注意这个域名解析+连接建立的过程是重新在workThreadPool中获取的一个新的后台线程中执行的，即在这个后台线程中会执行上面加粗的匿名函数体中的内容的。

连接建立后，需要执行回调。但此时的线程是一个为了不阻塞原线程而申请的一个新的后台线程（为执行DNS解析），所以像if(isIP(url.data)) {...}中那样直接执行回调函数。因为当前线程并不是原先的线程。

这种解释可能优点模糊。但这里使用poller->async的主要目的就是确保回调在正确的线程中执行。

我理解的是**确保回调在正确的上下文中执行，避免未定义的行为发生。**

下面是chatgpt生成的分析：

这段代码的作用是将回调函数 `strong_task` 的执行推迟到 `poller` 所管理的线程池或事件循环中执行，而不是在当前线程直接执行回调函数。

为什么使用 `poller->async(...)` 而不是直接执行回调？

1. **线程安全和事件循环**
   - 事件循环的设计原则之一是将某些操作推迟到特定的线程中执行，通常这个线程是主线程或一个指定的工作线程。
   - 如果直接在当前线程执行回调，可能会导致线程不安全的情况或阻塞其他重要操作，尤其是在多线程环境下。
   - 使用 `poller->async(...)` 可以确保回调在正确的线程中执行，并且通常是在事件循环的上下文中，这有助于保持应用程序的结构清晰和线程安全。
2. **避免直接在当前线程中执行阻塞操作**

- 如果回调函数中包含一些可能阻塞的操作，直接执行会导致当前线程阻塞，进而影响整个应用的响应能力。
- 通过 `poller->async(...)`，回调会被放入事件循环队列中，等待执行，这样就可以避免当前线程的阻塞。特别是在网络通信或事件驱动的系统中，保持事件循环的高效和非阻塞是非常重要的。

3. **确保回调在正确的上下文中执行**
   - 许多网络库或事件驱动框架在多个线程中工作，回调函数可能会涉及对特定资源的访问（例如套接字、文件描述符等）。这些资源可能只在某个特定线程中有效（例如，在事件循环线程中），而直接调用回调可能会导致资源访问冲突。
   - 使用 `poller->async(...)` 可以确保回调函数在正确的线程或上下文中执行，从而避免潜在的资源竞争或死锁问题。
4. **避免回调函数被提前销毁**
   - 在这段代码中，`weak_task` 是一个 `weak_ptr`，它的目的是避免回调函数被提前销毁。如果在回调执行之前 `weak_ptr` 被销毁，那么 `strong_task.lock()` 会返回 `nullptr`，从而避免访问已销毁的对象。
   - `poller->async(...)` 确保了回调的执行被延迟，直到当前的事件循环或者工作线程空闲。这样可以避免 `weak_ptr` 被销毁导致回调无法执行。

直接执行回调的潜在问题

1. **线程问题**:
   - 如果你在当前线程直接执行回调，而该回调访问了某些只能在主线程或事件循环线程中使用的资源，可能会导致线程安全问题或资源冲突。
2. **阻塞问题**:
   - 如果回调函数执行了阻塞的操作（例如同步的网络请求、文件操作等），那么直接执行回调会导致当前线程被阻塞，影响程序的其他部分。
3. **事件驱动框架要求**:
   - 很多事件驱动框架要求所有与 I/O 相关的操作必须在指定的线程中进行。如果直接在非事件循环线程中执行回调，可能会违反这些框架的设计规则，导致未定义的行为。

结论

使用 `poller->async(...)` 是为了确保回调在适当的上下文中执行，并且避免潜在的线程安全、资源访问冲突和阻塞问题。直接执行回调函数可能会导致程序结构混乱，尤其是在多线程或事件驱动的环境中。
因此，**这段代码是有必要的**，它确保了回调在合适的线程中执行，并且保持了代码的清晰性、可维护性和线程安全。

上面阐述了进行关于异步域名解析的问题。
域名解析的过程包含在了SockUtil::connect的函数中：

```
sockaddr_storage addr;
    //优先使用ipv4地址 [AUTO-TRANSLATED:b7857afe]
    //Prefer IPv4 address
    if (!getDomainIP(host, port, addr, AF_INET, SOCK_STREAM, IPPROTO_TCP)) {
        //dns解析失败 [AUTO-TRANSLATED:1d0cd32d]
        //DNS resolution failed
```

```
        return -1;
    }


bool SockUtil::getDomainIP(const char *host, uint16_t port, struct sockaddr_storage &addr,
                           int ai_family, int ai_socktype, int ai_protocol, int expire_sec) {
    bool flag = DnsCache::Instance().getDomainIP(host, addr, ai_family, ai_socktype, ai_protocol,
expire_sec);
    if (flag) {
        switch (addr.ss_family ) {
            case AF_INET : ((sockaddr_in *) &addr)->sin_port = htons(port); break;
            case AF_INET6 : ((sockaddr_in6 *) &addr)->sin6_port = htons(port); break;
            default: break;
        }
    }
    return flag;
}
```

从这个函数入手：

整体流程就是两个过程：（1）从DNS缓存中获取域名host对应的IP地址，并将其存储在addr中；（2）如果DNS缓存中没有找到该域名的IP地址，它会尝试从系统进行域名解析。

host如果就是IP地址（比如说IPv4的点分十进制形式），此时无需进行域名解析，只需将点分十进制转换为二进制形式存储在addr中返回即可。（下面代码（1）的过程SockUtil::make_sockaddr）。

host如果是域名，则就需要进行域名解析的过程。（下面代码（2）的过程，这里是（1）抛出异常后才会进行域名解析，也就是说host不能直接转换为合适的二进制形式存储在addr中后，才会执行域名解析）。

```
static DnsCache &Instance() {
        static DnsCache instance;
        return instance;
    }
    bool getDomainIP(const char *host, sockaddr_storage &storage, int ai_family = AF_INET,
                     int ai_socktype = SOCK_STREAM, int ai_protocol = IPPROTO_TCP, int expire_sec =
60) {
        try {
            storage = SockUtil::make_sockaddr(host, 0);//(1)
            return true;
        } catch (...) {
                /*(2)*/
            auto item = getCacheDomainIP(host, expire_sec);
            if (!item) {
                item = getSystemDomainIP(host);
                if (item) {
                    setCacheDomainIP(host, item);
                }
            }
            if (item) {
                auto addr = getPerferredAddress(item.get(), ai_family, ai_socktype, ai_protocol);
                memcpy(&storage, addr->ai_addr, addr->ai_addrlen);
            }
            return (bool)item;
        }
    }
```

// (1) make_sockaddr函数

- **功能**：该函数将 IP 地址（IPv4 或 IPv6）转换为 sockaddr_storage 结构。它首先尝试解析 host 是否为 IPv4 地址，如果是，则填充 sockaddr_in 结构。如果不是 IPv4，则继续尝试解析为 IPv6 地址，并填充 sockaddr_in6 结构。
- **流程**：
  - inet_pton(AF_INET, host, &addr)：将 host 转换为 IPv4 地址。如果 host 是有效的 IPv4 地址，inet_pton 将其转换为二进制格式并存储在 addr 中。
  - reinterpret_cast<struct sockaddr_in &>(storage) 将 storage 转换为 sockaddr_in 结构，并填充 sin_addr、sin_family 和 sin_port 等字段。
  - 如果 host 不是有效的 IPv4 地址，继续使用 inet_pton(AF_INET6, host, &addr6) 来检查它是否为 IPv6 地址。
  - 如果 host 既不是 IPv4 也不是 IPv6 地址，则抛出 std::invalid_argument 异常，表示 host 不是有效的 IP 地址。

```
struct sockaddr_storage SockUtil::make_sockaddr(const char *host, uint16_t port) {
    struct sockaddr_storage storage;
    bzero(&storage, sizeof(storage));
    struct in_addr addr;
    struct in6_addr addr6;
    if (1 == inet_pton(AF_INET, host, &addr)) {
        // host是ipv4  [AUTO-TRANSLATED:ba5c03a7]
        //Host is IPv4
        reinterpret_cast<struct sockaddr_in &>(storage).sin_addr = addr;
        reinterpret_cast<struct sockaddr_in &>(storage).sin_family = AF_INET;
        reinterpret_cast<struct sockaddr_in &>(storage).sin_port = htons(port);
        return storage;
    }
    if (1 == inet_pton(AF_INET6, host, &addr6)) {
        // host是ipv6  [AUTO-TRANSLATED:8048db0f]
        //Host is IPv6
        reinterpret_cast<struct sockaddr_in6 &>(storage).sin6_addr = addr6;
        reinterpret_cast<struct sockaddr_in6 &>(storage).sin6_family = AF_INET6;
        reinterpret_cast<struct sockaddr_in6 &>(storage).sin6_port = htons(port);
        return storage;
    }
    throw std::invalid_argument(string("Not ip address: ") + host);
}
```

// (2) **真正域名解析过程**

```
 auto item = getCacheDomainIP(host, expire_sec);
            if (!item) {
                item = getSystemDomainIP(host);
                if (item) {
                    setCacheDomainIP(host, item);
                }
            }
            if (item) {
                auto addr = getPerferredAddress(item.get(), ai_family, ai_socktype, ai_protocol);
                memcpy(&storage, addr->ai_addr, addr->ai_addrlen);
            }
            return (bool)item;
```

a. 从DNS缓存中获取域名的解析结果；
b. DNS缓存中没有找到，使用系统级DNS解析（getSystemDomainIP），并将结果存入缓存（setCacheDomainIP)。
c. 解析成功后，使用getPerferredAddress函数获取解析结果中最合适的地址，然后将其复制到storage中

首先DNS缓存用哈希表实现，存储<域名，DnsItem>的键值对。

```
unordered_map<string, DnsItem> _dns_cache;
```

DnsItem是个类，包含封装了IP地址的智能指针以及这个键值对是何时创建的时间。

```
class DnsItem {
public:
    std::shared_ptr<struct addrinfo> addr_info;
    time_t create_time;
};
```

DNS缓存查找域名对应的IP地址的过程也就是从哈希表_dsn_cache中查找相应键值对的过程。（需要注意判断缓存中的IP地址是否过期，如果过期，需要从哈希表中删除表项，并返回nullptr。

```
std::shared_ptr<struct addrinfo> getCacheDomainIP(const char *host, int expireSec) {
    lock_guard<mutex> lck(_mtx);
    auto it = _dns_cache.find(host);
    if (it == _dns_cache.end()) {
        //没有记录 [AUTO-TRANSLATED:e99e45df]
        //No record
        return nullptr;
    }
    if (it->second.create_time + expireSec < time(nullptr)) {
        //已过期 [AUTO-TRANSLATED:5dbe0c9a]
        //Expired
        _dns_cache.erase(it);
        return nullptr;
    }
    return it->second.addr_info;
}
```

将DNS解析后的结果存入到DNS缓存中，以便后续查找更加快速。

```
void setCacheDomainIP(const char *host, std::shared_ptr<struct addrinfo> addr) {
    lock_guard<mutex> lck(_mtx);
    DnsItem item;
    item.addr_info = std::move(addr);
    item.create_time = time(nullptr);
    _dns_cache[host] = std::move(item);
}
```

系统级DNS解析

```
std::shared_ptr<struct addrinfo> getSystemDomainIP(const char *host) {
    struct addrinfo *answer = nullptr;
    //阻塞式dns解析，可能被打断 [AUTO-TRANSLATED:89c8546f]
    //Blocking DNS resolution, may be interrupted
    int ret = -1;
    do {
        ret = getaddrinfo(host, nullptr, nullptr, &answer);
    } while (ret == -1 && get_uv_error(true) == UV_EINTR);
    if (!answer) {
        WarnL << "getaddrinfo failed: " << host;
        return nullptr;
    }
    return std::shared_ptr<struct addrinfo>(answer, freeaddrinfo);
```

```
    }
```

注意：参见《Linux高性能服务器编程》书籍5-13节，getaddrinfo将隐式的分配堆内存，所以调用完毕后，必须使用相应的配对函数freeaddrinfo来释放这块内存。

```
    struct addrinfo *getPerferredAddress(struct addrinfo *answer, int ai_family, int ai_socktype, int ai_protocol) {
        auto ptr = answer;
        while (ptr) {
            if (ptr->ai_family == ai_family && ptr->ai_socktype == ai_socktype && ptr->ai_protocol == ai_protocol) {
                return ptr;
            }
            ptr = ptr->ai_next;
        }
        return answer;
    }
```

至此，域名解析的整个流程基本就清楚了。