初赛 Code Review 辅助材料 - Blink` 团队

1. 算法设计思路

题目分析

题目要求实现 RPC agents(consumer and provider agents)， 并且通过 docker 限制了资源, 线上环境 consumer CPU 处理能力在 `512` 并发请求时候不足， provider small 由于内存小 gc 很多，并且 cpu 时间片少，相对 latency 会很大。

我们大体想法是 consumer agent 只负责解析 http 请求，转发表单的内容到 provider agent； 然后通过一定基于 time-window 自适应的消息响应时间 load balance 策略分发请求， provider agent 解析表单可以利用向量化的指令进行 tokenize 和产生 dubbo package 发送到 provider。 所有的异步请求都通过 libuv 的 epoll 进行实现。为了减小 latency，我们使用多个 eventloop 绑定到 consumer 的 tcp 端口分流 htttp 请求。

最通用代码版本 (`7046+`, `2018-06-19 02:00:48`)

content	info
link	https://code.aliyun.com/191836400/agent-submit.git
git version	`27d591d55135abeda39aa824d747daf09ee1c460`

最高分代码版本 (`7144+`, `2018-06-17 18:55:46`)

content	info
link	https://code.aliyun.com/191836400/agent-submit.git
git version	`6af4f23f60b2c43c70eca9d7265c05617e68ced7`

LibUV Event Loops

Consumer <-> Consumer Agent:

1. Consumer agent 使用 2 个 eventloop 通过 tcp 端口复用同时监听服务端口，接受从 consumer 来的请求，并且分配请求 ID，
2. 使用基于 SSE 加速的 http parser picohttpparser 解析 http 请求，
3. 使用基于历史返回时间的 load balancer 选择 provider agent，
4. 接受从 provider agent 返回的响应数据，根据请求 ID 返回给对应的 http 连接线路。

Consumer Agent <-> Provider Agent

1. 每个 consumer agent eveltloop 建立一条长连接到每个 provider agent。即 agent 之间连接数=(consumer agent eventloop 个数)×(provider 个数)，
2. consumer agent 与 provider agent 之间使用自定义协议格式通信，转发请求的 formBody，
3. provider agent 接受到来自 dubbo 的响应之后进行拆包并且构造 http response 响应包，通过自定义协议发送给 consumer agent。

Provider Agent <-> Provider

1. 每个 provider agent 使用 1 条 长连接与 dubbo provider 通信，
2. provider agent 接受 consumer agent 发来的 formBody 进行协议转换，构造符合 dubbo 要求的二进制请求包，发送给 provider，
3. provider agent 使用向量化指令进行协议转换，
4. provider agent 接受 dubbo provider 返回的响应包，进行拆包并且构造最终的 http response 响应包，发送给 consumer agent。

基于历史返回时间的 Load Balancer

数据结构

```C++
typedef struct {
    int64_t diff_time_sum;
    int head;
    int tail;
    int remain;
    int64_t diff_time_history[HISTORY_WINDOW_SIZE];
} history_diff_time_t;

typedef struct {
    int num_of_ends_;
    history_diff_time_t *history_diff_time_lst[16];
} load_balancer_t;

load_balancer_t *init_load_balancer(int num_of_ends);

history_diff_time_t *init_history_time();

int select_endpoint_time_based(load_balancer_t *this, http_req_t *http_req);

void finish_endpoint_task_time_based(load_balancer_t *this, int end_point_idx, http_req_t *http_req);
```

数据结构含义

`history_diff_time_t` 对象用于记录一个 provider 端消息的处理情况(通过记录一个 `HISTORY_WINDOW_SIZE` 大小的时间窗口对应的响应时间), 整个应用有一个全局的 `load_balancer_t` 对象(受到一把全局锁保护)，用来存储多个 provider 的历史处理 RPC 能力。`select_endpoint_time_based` 返回当前评估最优的 provider，`finish_endpoint_task_time_based` 在 consumer 接收到来自 provider(编号 `end_point_idx`) 时候，更新历史处理能力窗口内数据。

限制流量和防止饥饿

我们在 Load Balancer 中进行了**限制流量**和**防止饥饿**的操作。`history_diff_time->remain` 表示了一个 provider(编号 `index`)当前未返回的请求个数， `history_diff_time->remain >= penalty_send_limit[index]`是为了限制流量，防止 exhausted 情况出现; `history_diff_time->remain < direct_send_limit[index]` 是为了防止饥饿出现，让历史处理时间窗口中的信息反映最近某 provider 真实响应能力。

```C++
inline int64_t estimate_cost(load_balancer_t *this, int index) {
    history_diff_time_t *history_diff_time = this->history_diff_time_lst[index];
    if (history_diff_time->remain >= penalty_send_limit[index]) {
        return INT64_MAX;
    }
    if (history_diff_time->remain < direct_send_limit[index]) {
        return  0;
    }
    return history_diff_time->diff_time_sum;
}
```

向量化指令使用 (解析表单，构造 dubbo 二进制请求包)

向量化指令指南: (intrinsic guide: see https://software.intel.com/sites/landingpage/IntrinsicsGuide/)

解析表单

查找下一个`=` 和 `&`可以使用 `SSE4` 指令集来减少指令条数。

C++

```cpp
inline int sse4_split_efficient(char *str, short len, short *off) {
#ifdef DEBUG
    assert(len > 0 && len + 16 < 32767);
#endif
    char restore_buf[16];
    memcpy(restore_buf, str + len, sizeof(char) * 16);
    memset(str + len, 0, 16);
    off[0] = -1;
    for (short i = 0, next = 1;;) {
        // 1st: advance fo find first '='
        while (true) {
            __m128i pivot_u = _mm_set1_epi8('=');
            __m128i inspected_ele = _mm_loadu_si128((__m128i *) (str + i));
            __m128i cmp_res = _mm_cmpeq_epi8(pivot_u, inspected_ele);
            int mask = _mm_movemask_epi8(cmp_res); // 16 bits

            int advance = (mask == 0 ? 16 : __tzcnt_u32_using_popcnt_cmpgt(mask));
            i += advance;
            if (advance < 16) { break; }
        }
        off[next] = i;
        next++;

        // 2nd: advance to find first '&'
        while (true) {
            __m128i pivot_u = _mm_set1_epi8('&');
            __m128i inspected_ele = _mm_loadu_si128((__m128i *) (str + i));
            __m128i cmp_res = _mm_cmpeq_epi8(pivot_u, inspected_ele);

            int mask = _mm_movemask_epi8(cmp_res); // 16 bits

            int advance = (mask == 0 ? 16 : __tzcnt_u32_using_popcnt_cmpgt(mask));
            i += advance;
            if (advance < 16 || i >= len) { break; }
        }
        off[next] = i < len ? i : len;
        next++;

        if (i >= len) {
            memcpy(str + len, restore_buf, sizeof(char) * 16);
            return next;
        }
    }
}

#endif
```

构造 dubbo 二进制请求包

构造 dubbo 包需要解析解析表单中`%xx`的字段， 寻找`%`可以使用`SSE4`指令集来减少指令条数。

C++

```cpp
int serialize_str(char *str, const short *off, short len_off, int offset, char *res, int j) {
    for (int i = 0; i < len_off / 2; i++) {
        int real_i = i * 2;
        int key_off_beg = off[real_i] + 1;
        int key_off_end = off[real_i + 1];
        // add post_keys[j][0] == str[key_off_beg] to reduce strncmp invocations
        if (post_keys[j][0] == str[key_off_beg] && key_off_end - key_off_beg == post_keys_size[j]) {
            if (strncmp(post_keys[j], str + key_off_beg, key_off_end - key_off_beg) == 0) {
                res[offset] = '"';
                offset++;

                // copy and decoding url e.g. '%2F' to '/', wrap as a inline function later
                int val_off_beg = off[real_i + 1] + 1;
                int val_off_end = off[real_i + 2];
                for (int last_seek, seek = val_off_beg; seek < val_off_end;) {
                    last_seek = seek;

                    // find first % or end
#if defined(__BMI__)
                    // printf("bmi...");
                    while (true) {
                        __m128i pivot_u = _mm_set1_epi8('%');
                        __m128i inspected_ele = _mm_loadu_si128((__m128i *) (str + seek));
                        __m128i cmp_res = _mm_cmpeq_epi8(pivot_u, inspected_ele);

                        int mask = _mm_movemask_epi8(cmp_res); // 16 bits

                        int advance = (mask == 0 ? 16 : __tzcnt_u32(mask));
                        seek += advance;
                        if (advance < 16 || seek >= val_off_end) { break; }
                    }
                    if (seek > val_off_end) { seek = val_off_end; }

```

```
#elif defined(__SSE4_2__)
                    // printf("sse...");
                    while (true) {
                        __m128i pivot_u = _mm_set1_epi8('%');
                        __m128i inspected_ele = _mm_loadu_si128((__m128i *) (str + seek));
                        __m128i cmp_res = _mm_cmpeq_epi8(pivot_u, inspected_ele);

                        int mask = _mm_movemask_epi8(cmp_res); // 16 bits

                        int advance = (mask == 0 ? 16 : __tzcnt_u32_using_popcnt_cmpgt(mask));
                        seek += advance;
                        if (advance < 16 || seek >= val_off_end) { break; }
                    }
                    if (seek > val_off_end) { seek = val_off_end; }
#else
                    while (seek < val_off_end && str[seek] != '%') { seek++; }
#endif
                    int advance = seek - last_seek;
                    memcpy(res + offset, str + last_seek, advance);
                    offset += advance;
                    // decode %xx
                    if (seek != val_off_end) {
                        res[offset] = (char) (hex2int(str[seek + 1]) * 16 + hex2int(str[seek + 2]));
                        offset++;
                        seek += 3;  //skip "%xx"
                    }
                }

                memcpy(res + offset, "\"\n", 2);
                offset += 2;
                break;
            }
        }
    }
    return offset;
}
```

2. 创新点

1) 使用了向量化指令进行 tokenize，解析 http 表单的内容以及生成 dubbo 二进制请求。

3) 使用 linux 提供的 tcp 端口复用 (SO_REUSEPORT)技术，将一个 tcp 端口上的请求分流到不同的 eventloop，减小 latency。

2) 使用了 libuv event loop，来处理 http 请求和转发的 http 表单 tcp 包的解析, 使用纯 C 编写回调函数； consumer agent 仅负责解析 http 请求，不做协议转换, 来减少 consumer 镜像 CPU 的占用。

3) 使用了自适应的基于消息响应时间的 load balacer 分发表单到 provider。