Pachi 代码阅读 — By Hug

目的：通过阅读 *pachi* 源代码，清楚 *pachi* 整体框架逻辑，以及能够通过对其分布式版本的性能做出评估，具备修改的能力。

方法：通过阅读雨亮的文档以及 Pachi 中的源代码，完成以上目的

时间估计：

$$\frac{15000}{3000} = 5\text{天（一周工作量）}$$

实际用时：3.5天

一、重点讨论内容

1. uct 中的树的表示
2. distributed 中的分布式及 tree_node 节点信息共享策略
3. 基于 engine 抽象接口的引擎实现部分

二、stone.(ch) 代码阅读

模块说明

用来表示棋盘棋子的数据结构，枚举类型，具体说明如下

```
enum stone {
    S_NONE, // 未落子点，空点，未知点
    S_BLACK, // 黑棋
    S_WHITE, // 白棋
    S_OFFBOARD, // 棋盘边界
    S_MAX, // 棋子类型总数
};
```

模块方法说明

其中提供的方法，属于类型值之间的转换功能，如下所示

```
//棋子类型和棋盘 char 的互相转换转换，四类棋子类型分别对应 .XO#
static char stone2char(enum stone s);
static enum stone char2stone(char s);

//棋子类型和 str 的互相转换，其中只提供黑白棋和未知位置的转换，对应：black/white/none
char *stone2str(enum stone s);
enum stone str2stone(char *str);

//获得对家棋子的颜色
static enum stone stone_other(enum stone s);
```

三、move.(ch) 代码阅读

模块说明

用来表示每一步棋的信息，包括棋子落下的位置和类型，具体说明如下

```
struct move {
    // int类型，代表棋子落下的位置，利用类似于如下公式完成整形与坐标值之间的转换
    // x = coord % board_size
    // y = coord / board_size
    coord_t coord;

    // 棋子类型，方法和说明参考 stone.(ch)代码阅读
    enum stone color;
};
```

宏定义说明：

围绕着 move 类型展开的宏定义如下

```
// 完成整型 coord 与棋盘(b)坐标点 (x,y) 之间的的互相转换
define coord_xy(board, x, y)
define coord_x(c, b)
define coord_y(c, b)

// 计算棋盘(b)上两个坐标点(c1, c2)之间 x 坐标和 y 坐标之间的差值
define coord_dx(c1, c2, b)
define coord_dy(c1, c2, b)

// pass 代表放弃本轮下子，resign 代表认输
define is_pass(c)
define is_resign(c)

// 判断棋盘(b)上两个坐标点是否是相邻，相邻的概念有两种，直线相邻4个方向和周边相邻8个方向
define coord_is_adjecent(c1, c2, b)
define coord_is_8adjecent(c1, c2, b)

// 计算坐标点(c)在棋盘上的哪个区域，将棋盘横纵划分成4个区域
define coord_quadrant(c, b)
```

模块inline函数定义说明：

围绕着 move 类型展开的inline函数定义如下

```c
// new一个coord_t，并用 x，y 将其初始化，棋盘大小 size
// 第二个方式是用已有的一个coord_t 类型的数据，进行初始化
static coord_t *coord_init(int x, int y, int size);
static coord_t *coord_copy(coord_t c);

// 分别拷贝放弃和认输的落子方案
static coord_t *coord_pass(void);
static coord_t *coord_resign(void);

// 释放 coord 所占内存
static void coord_done(coord_t *c);
```

模块方法说明

围绕着 move 类型展开的inline函数定义如下

```c
// 将 coord 转换为一个棋盘位置，例如：E2 或者 A9
char *coord2bstr(char *buf, coord_t c, struct board *board);

// 类似于 coord2bstr，只不过这个方法会 dup 内存空间
char *coord2str(coord_t c, struct board *b);

// 缓存 10 步棋的走法，方便后续 debug
char *coord2sstr(coord_t c, struct board *b);

// 字符串（A1/E9） 转换为棋盘上的坐标点等价的整型值
coord_t *str2coord(char *str, int board_size);
```

四、board.(ch) 代码阅读

模块说明

用来表示棋盘和棋盘相关状态的数据结构，具体说明如下

```
struct board {
    // 棋盘大小相关数据，包括了 2 个长度的棋盘边框
    // 大小数据包括：变长、面积 和 整型二进制所占位数
    // bits2 的作用，请参看《特殊算法说明》一节
    int size;
    int size2;
    int bits2;
    int captures[S_MAX];
    floating_t komi;
    int handicap;

    // 围棋的不同规则
    enum go_ruleset {
        RULES_CHINESE, /* default value */
        RULES_AGA,
        RULES_NEW_ZEALAND,
        RULES_JAPANESE,
        RULES_STONES_ONLY,
        RULES_SIMING,
    } rules;

    // 棋谱信息，在某些策略中，会以棋谱数据为准
    char *fbookfile;
    struct fbook *fbook;

    // 计算 8 邻接和 4 邻接时用到的 offset 值
    int nei8[8], dnei[4];

    // 记录棋盘中落子的信息，记录从倒数第一个落子到倒数第4次的落子信息
    int moves;
    struct move last_move;
    struct move last_move2;
    struct move last_move3;
    struct move last_move4;

    // "专业"棋手鉴定后，表示不用理会，后面再说
    bool superko_violation;
};
```

五、network.(ch) 代码阅读

```
// 监听一个本机的端口，成功则返回文件描述符
int port_listen(char *port, int max_connections);

// 打开一个服务器
int open_server_connection(int socket, struct in_addr *client);

// 打开一个网络端口，专门接收 log 信息
void open_log_port(char *port);

// 大开一个 GTP 接口，如果成功，则将 stdin 和 stdout 重定向到 socket 对应的文件，三者共享状态
void open_gtp_connection(int *socket, char *port);
```

六、pachi.(ch) 代码阅读

代码说明

pachi 主函数所在位置，主要处理前向信息，其中主要函数及解释如下

```
// 对应了 engine 的不同策略，E_MAX 代表 engine 的总数
enum engine_id {
    E_RANDOM,
    E_REPLAY,
    E_PATTERNSCAN,
    E_PATTERNPLAY,
    E_MONTECARLO,
    E_UCT,
    E_DISTRIBUTED,
    E_JOSEKI,
#ifdef DCNN
    E_DCNN,
#endif
    E_MAX,
};

// 定义 engine_init 的函数指针数组，返回不同策略的 engine 初始化的方法
// 当策略定义是 E_DISTRIBUTED 的时候，会用到 engine_distributed_init 方法
// 此方法会初始化分布式环境的 pachi-engine
static struct engine *(*engine_init[E_MAX])(char *arg, struct board *b)
= {
    engine_random_init,
    engine_replay_init,
    engine_patternscan_init,
    engine_patternplay_init,
    engine_montecarlo_init,
    engine_uct_init,
    engine_distributed_init,
    engine_joseki_init,
```

```
#ifdef DCNN
    engine_dcnn_init,
#endif
};

// 初始化引擎，引擎的主要功能是完成落棋的每一步计算
static struct engine *init_engine(enum engine_id engine, char *e_arg,
struct board *b);

// 销毁引擎用到的相关内存资源
static void done_engine(struct engine *e);

// 打印帮助信息
static void usage(char *name);

// Pachi 程序的主函数，主函数中主要是进行命令行参数解析，以及处理接收到的命令
// 处理完命令后，消息的回传分为两类：单机模式 和 网络模式
// 当指定了 -g 参数后，消息回传属于网络模式，否则属于单机模式(直接输出到本机命令行)
int main(int argc, char *argv[]);
```

七、gtp.(ch) 代码阅读

gtp 中包含了pachi 最主要的处理逻辑，包括对于策略引擎各接口的调用。其中最大的逻辑代码就是gtp_parse 函数，所以下面主要就 gtp_parse 函数做介绍，从而能够理解整个框架的逻辑。

```
// 此函数将 to_ 指向当前 cmd， next 指向下一个命令的开头或者结尾
#define next_tok(to_)

// 以下是第一组命令，基本的查询的命令
// 根据 cmd 的值，进入不同分支，主要对应关系如下
("protocol_version", gtp_reply(id, "2", NULL))
("name", gtp_reply(id, "Pachi ", engine->name, NULL))
("echo", gtp_reply(id, next, NULL))
("version", gtp_reply(id, PACHI_VERSION, ": ", engine->comment, " Have a nice game!", NULL))
("list_commands", gtp_reply(id, known_commands(engine), NULL))
("known_command", gtp_reply(id, gtp_cmd, NULL))

// 第二组逻辑，主要是围绕着engine 的 notify 功能
// 如果引擎设置了 notify 功能，并且引擎的 notify 功能能处理 cmd 命令
// 根据 notify 的返回值进入不同分支，对应关系如下
c = engine->notify(engine, board, id, cmd, next, &reply)
(P_NOREPLY, 修改 id 的值为 NO_REPLY)
(P_DONE_OK, gtp_reply(id, reply, NULL))
(P_DONE_ERROR, gtp_error(id, reply, NULL)) // 引擎发生内部错误
(c != P_OK, return c) // 其余情况

// 第三组逻辑，是整个功能代码最大的逻辑（没有之一），这部分逻辑将在下面的 Table-List 中做解释
```

gtp_parse 中第三组逻辑的解释，每5个一组做介绍，如下 Table-List，第一列是 **Cmd 字符串**，第二列是**动作解释**，第三列是**特殊说明**

quit	**退出程序**	
boardsize	设置棋盘大小	会重新初始化棋盘信息 clear 操作
clearboard	清空棋盘	游戏次数会增加 +1
kgs-game_over	接收的 kgs游戏结束的标志	
komi	设置贴目的数量	

kgs-rules	**为 kgs 预留的接口，设置棋局的规则**	
play	模拟某个玩家下一个棋子，当引擎设置了 notify_play（被通知，棋局上玩了一手）方法时，可以做 pondering，例如 uct	将 debug-level 设置到 4可以看到棋盘实时输出的结果
pachi-predict	人为预测 pachi 的引擎会将某个颜色的棋子落在某个位置	其中会调用 genmove
genmove	由引擎内部策略生成某一颜色的棋子的一步落棋，其中会以『棋谱』优先，其次是引擎 genmove 策略	kgs-genmove_cleanup
pachi-genmoves		pachi-genmoves_cleanup

set_free_handicap		
place_free_handicap		fixed_handicap
final_score		
final_status_list		
undo		

pachi-gentbook		
pachi-dumptbook		
pachi-evaluate		
pachi-result		
kgs-chat	为 kgs 预留的聊天接口，会调用 engine 的 chat 方法	

time-left		
time_settings		kgs-time_settings
gogui-analyze_commands		
gogui-owner_map		
gogui-best_moves		

gogui-winrates		
others	gtp_error(id, "unknown command", NULL)	

gtp中其他函数解释

```
// 输出回复（正常）消息
void gtp_reply(int id, ...);
// 输出错误信息
void gtp_error(int id, ...);
// 输出 gtp 消息到 stdout，先输出前缀信息，然后按照字符串输出 params 中的所有信息
void gtp_output(char prefix, int id, va_list params);
// 输出 gtp 的前缀，gtp_reply 中是 "="，gtp_error 中是 "?"
void gtp_prefix(char prefix, int id);
// flush stdout 中的全部消息
void gtp_flush(void);
// 输出赢家的分数
void gtp_final_score(struct board *board, struct engine *engine, char
*reply, int len);
// 暂时不知道是干什么的
void gogui_set_live_gfx(struct engine *engine, char *arg);
//
char *gogui_best_moves(struct board *b, struct engine *engine, char
*arg, bool winrates);
// 输出当前棋局最可能的计分 owner_map
void gogui_owner_map(struct board *b, struct engine *engine, char
*reply);
// 判断一个命令 cmd 是否是引擎 engine 所支持的
bool gtp_is_valid(struct engine *e, const char *cmd);
//
void gtp_predict_move(struct board *board, struct engine *engine,
struct time_info *ti);
```

问题记录

- known_commands 函数中有严重的内存泄露问题，长时间多次调用，可能导致系统崩溃
- gtp_is_valid 内部会调用 known_commands，基本所有命令都会吊起 gtp_is_valid，结果可想而知

八、chat.(ch) 代码阅读

模块说明

用来处理 server 与 client 交互时的对话信息的，存储对话信息的结构说明如下

```c
static struct chat {
    // 当胜率在 minwin 和 maxwin 之间，并且信息是来自 from 的时候
    // 对话信息命中 regex，回复 reply 中的内容，reply 中允许引用变量
    // 当前版本中，允许的变量只有 winrate 一个，引用方式采用 %lf 的形式
    double minwin;
    double maxwin;
    char from[20];
    char regex[100];
    char reply[300];

    // displayed 表明此条信息是否显示，match 显示当前会话中的信息是否命中此条 chat 配置
    regex_t preg;
    bool displayed;
    bool match;
} *chat_table;
```

模块方法说明

方法主要是对话信息的初始化、数据处理和数据销毁，以下函数均有两个编译版本，当系统不支持正则表达式时，以下函数均为空方法，各函数的声明及说明如下所示

```c
// 从 chat_file 中初始化对话信息
void chat_init(char *chat_file);

// 销毁对话相关数据
void chat_done();

// 根据当前环境，生成对话信息
char *generic_chat(struct board *b, bool opponent, char *from,
                   char *cmd, enum stone color, coord_t move,
                   int playouts, int machines, int threads,
                   double winrate, double extra_komi);
```

九、distributed 模块代码阅读

模块说明

此模块主要提供分布式的 engine 框架，使得下棋策略计算压力可以分散到多台 slave 服务器上，slave服务器的计算结果最后由 master 服务器统一用 merge 策略完成。

模块组成部分

此模块由三部分组成：节点共享传输协议、分布式框架控制功能、合并策略功能

对应代码文件分别为：protocol, distributed, merge

接下来将对每部分做**详细阅读**

protocol.(ch) 代码阅读

子模块说明

此模块主要维护 master 与 slave 之间的信息传输过程。其中维护信息传输的协议与相关传输方法。

下面就子模块中实现的重要函数做说明

```c
// 初始化传输协议，根据设置的 slave_nums = N，起 N 个线程监听 slave 的连接请求
// 一旦某个 slave 连接上以后，就会保持持续连接，与此 slave 通信的动作都由这个线程完成
void protocol_init(char *slave_port, char *proxy_port, int max_slaves)

// 获得至少 min_replies 个 slave 的回复信息
// 一般在调用了 new_cmd 或者 update_cmd 命令后，上层函数会调用此函数获取 m 个返回结果
void get_replies(double time_limit, int min_replies)

// 新建一个发送给 slave 的 gtp cmd，此函数最后会调用 update_cmd
void new_cmd(struct board *b, char *cmd, char *args)

// 将 cmd 中的命令更新到 gtp_cmd 所在内存空间，并会发送广播信号量
// 所有连接 slave 的线程都会对这个信号量做出反应，并检查 gtp_cmd 中的内容，将内容发送给 slave
void update_cmd(struct board *b, char *cmd, char *args, bool new_id)

// slave 线程的主要控制函数，主要做环境初始化
// 对于 slave 的长链接过程中的功能实现在 slave_loop 函数中
// 此函数的主要功能就是等到 slave 节点的连接
static void * __attribute__((noreturn)) slave_thread(void *arg)

// 维护 master 与 slave 之间长链接的方法，主要就是等到 gtp_cmd 相关信号量
// 检测到相关信号量以后，会将命令发送给 slave 节点，并且接收 slave 节点的返回信息
static void slave_loop(FILE *f, char *reply_buf, struct slave_state *sstate, bool resend)

// 获得共享 tree_node 数据信息的二进制值，以指针形式返回其数据区头指针
void *get_binary_arg(struct slave_state *sstate, char *cmd, int cmd_size, int *bin_size)

// 处理 slave 节点的返回信息的函数，将返回信息整理并且储存起来，供上层函数做进一步解析
static bool process_reply(int reply_id, char *reply, char *reply_buf,
          void *bin_reply, int bin_size, int *last_reply_id,
          int *reply_slot, struct slave_state *sstate)

// 向 slave 节点发送命令的函数，发送命令之前，会先将 bin_buf 中的内容发送到 slave 节点上
// bin_buf 中的数据就是共享 tree_node 数据信息
static int
send_command(char *to_send, void *bin_buf, int *bin_size,
         FILE *f, struct slave_state *sstate, char *buf)
```

merge.(ch) 代码阅读

子模块说明

merge 模块主要做共享 tree_node 信息的维护。其中主要函数就是 get_new_stats，其挂载在 sstate 的 args_hook 的函数指针上，主要功能是完成共享 tree_node 信息的更新，此处策略在整个流程中的作用，请阅读**【特殊算法说明-4】**。

distributed.(ch) 代码阅读

入口功能函数说明

入口功能函数主要有两个：path2sstr 和 engine_distributed_init，对于二者的说明如下

```
// 将一条树上的路径，转换为一个字符串输出
// 对于路径的表示，用到了特殊算法中的基于 board.bits2的知识表示相关内容
char *path2sstr(path_t path, struct board *b);

// 初始化一个 distributed 环境的 master 节点，作为 engine
// slave 节点的相关策略方法的 engine_id 为 E_UCT
struct engine *engine_distributed_init(char *arg, struct board *b);
```

主要功能函数说明

```
// 初始化一个 distributed 的引擎，设置 engine 中各 hook 方法之前
// 会首先调用 distributed_state_init 方法，初始化一个 dist 实例
struct engine *engine_distributed_init(char *arg, struct board *b)

// 初始化一个 dist 对象，其中包括了 distributed 引擎在运行时所需要的必要的方法和数据
// 在初始化时，最后会初始化两个分布式环境中重要的模块信息
// merge 策略模块 和 protocol 模块
static struct distributed *distributed_state_init(char *arg, struct board *b)

// 根据棋局 b 描述的信息，生成下一步走棋的位置，会调用 genmoves_args 函数
// 从分布式的信息中，选择 best_move 的策略会调用 select_best_move 方法
static coord_t *distributed_genmove(struct engine *e, struct board *b, struct time_info *ti, enum stone color, bool pass_all_alive)

// 生成发送给 slave 的 genmove 的命令，并且会调用new_cmd 生成一条新命令
// 在new_cmd 中会调用 update_cmd，update_cmd 方法会发送一个信号量
// 所有 slave_loop 线程接到这个信号量后，会向连接的 slave 发送一条命令消息
// 在发送消息之前，会首先更新 master 节点的共享 tree_node 数据信息
// 在发送命令之前，首先会将共享 tree_node 信息发送给各个 slave 节点
static void genmoves_args(char *args, enum stone color, int played, struct time_info *ti, bool binary_args)

// master 节点选择 best_move 的策略非常简单，选择 playouts 次数最多的节点作为 best_move
static coord_t select_best_move(struct board *b, struct large_stats *stats, int *played, int *total_playouts, int *total_threads, bool *keep_looking)

// 发送合法的命令行 cmd 给 slave
static enum parse_code distributed_notify(struct engine *e, struct board *b, int id, char *cmd, char *args, char **reply)

// 共享 tree_node 信息的更新策略，具体请参考 特殊算法说明-6
static void large_stats_add_result(struct large_stats *s, floating_t result, long playouts)

// 此方法不是重点，不做重点介绍
static char *distributed_chat(struct engine *e, struct board *b, bool opponent, char *from, char *cmd)

// 此方法在主要下棋策略的流程中没看到相关调用，后续再说，记录到 todo 里面
static void distributed_dead_group_list(struct engine *e, struct board *b, struct move_queue *mq)
```

十、特殊算法说明

1、基于 board.bits2 的知识表示

pachi 中用一个整型（int）表示一个棋盘的点，在某些策略中，涉及到树型结构，树型结构的表示和存储成为一个新问题。由于 board 最大是 $19 * 19 = 361 < 2^8$（board.bits2 = 8）, 所以一个8位二进制数字，可以表示任意一个棋盘点，而一个64位2进制数字，可以切分成8个8位2进制数字，假设树型结构的深度不会超过8，那么一个 Int64 可以表示一条长度不超过8的路径，这正是树型结构需要的。

所以在 pachi 中的某些engine 中，会用一个 Int64，表示长度不超过 $\frac{64}{m}$ 的一条树上的路径。其中 m = board.bits2。

2、connection 后续逻辑中的特殊处理

如果程序启动时，设置了-g 参数，内部就会调用 open_gtp_connection 操作，此操作会监听一个端口，并且返回对应的 socket 文件描述符。这个函数内部还做了一个非常重要的操作，就是将 socket, stdin, stdout 三者的文件描述符，都对应到 socket 所对应的文件描述符上。这么做以后，会产生一个奇怪的问题，奇怪的问题出现在如下流程中：

从 stdin 读入命令 —> 处理逻辑（中间可能有输出 stdout） —> 返回结果 —> 从 stdin 读入命令

在处理逻辑过程中，产生的输出中，可能会包含 pachi 中的命令（这一点无法避免），就会使得下一轮读入命令的操作会获得这一输出部分数据，这部分数据会被错误的解析成命令，从而使得整个流程混乱掉。所以，在系统中，作者加了对于这一特殊情况的处理代码，这一部分代码显得冗余，实则有用，如下：

```
for (;;) {
        char buf[4096];
        while (fgets(buf, 4096, stdin)) {
            if (DEBUGL(1))
                fprintf(stderr, "IN: %s", buf);

            enum parse_code c = gtp_parse(b, e, ti, buf);
            if (c == P_ENGINE_RESET) {
                ti[S_BLACK] = ti_default;
                ti[S_WHITE] = ti_default;
                if (!e->keep_on_clear) {
                    b->es = NULL;
                    done_engine(e);
                    e = init_engine(engine, e_arg, b);
                }
             // 增加判断逻辑，条件满足时，需要关闭旧的 socket，打开一个新的。
            } else if (c == P_UNKNOWN_COMMAND && gtp_port) {
                break;
            }
        }
        if (!gtp_port) break;
        // 略显冗余的逻辑，就是处理上述情况的
        open_gtp_connection(&gtp_sock, gtp_port);
    }
```

3、unlikely 和 likely 的作用

pachi工程内部，在棋局相关的程序内部的条件判断中，大量用到了 unlikely 和 likely 将条件包括起来，其作用与 linux 内核中的 likely 与 unlikely 的作用相同，增加 CPU 做分支预测的成功率。

4、UCT 引擎中的分布式

基本说明

pachi 中的分布式版本主要依靠 distributed(master) 与 UCT(slave) 两个引擎共同完成。整体结构属于经典的星型结构。

master 节点中可以设置各 slave 之间共享树节点的数量（这个值在 slave 中必须相同），初始化时，在 protocol init 的时候，会起 N（max_slaves） 个线程，每个线程用来处理与一个 slave 节点的信息传输。利用 slave_loop 保持与一个 slave 的长链接属性。master 节点通过 update_cmd 以及 cmd_cond 信号量向所有 slave（没在与 master 进行交互） 节点发送命令。

在每次 master 想要向 slave 节点发送命令时，都会调用 args_hook，此函数会去获取每一个 slave 上的节点信息，并且依据获得信息更新 master 中存储的共享节点的信息，并且在发送命令之前，会先将共享节点信息发送给 slave 节点。

所以依据以上策略，各 slave 节点之间部分信息得到了共享，共享的信息是否越多越好，需要进一步验证。如果不设置共享节点，在此分布式环境中，每个机器维护一棵树的信息，并且自行做 pondering。

重点总结

1. master 节点中不存储树的信息，只做共享tree_node数据管理
2. master 在每一次向 slave 发送消息之前，首先会收集整理共享 tree_node 数据，然后发送共享 tree_node数据给各个 slave
3. 每个 slave 各自存储一棵树，并且在没有命令的时候，每个 slave 均在持续地做 pondering
4. master 与 slave 之间属于长链接

5、Pachi中树的表示

在 pachi 中，Pachi 将一棵 N 叉树，表示成为一棵等价的二叉树，由此可以得知其每个节点记录三条边的信息，parent、sibling 和 children。每个节点有一个记录深度的变量 d，系统内对 d 做了限制，TREE_NODE_D_MAX + 1，系统内默认最大值是 4，此树的深度代表对应的 N 叉树的深度。

利用 tree 结构做棋局展开的时候，除了记录 tree 的 root 节点地址以外，还需要记录当前棋局状态(board)，每次将 tree 展开时，都会调用 tree_expand_node 方法。

6、分布式环境下的选择 best move 的策略

在 distributed 中，genmove 时，需要得到每个 slave 返回的 playout 的结果，将各结果进行合并。合并的公式为：

$$r1 = r1 + \frac{t2 \times (r1 - r2)}{total}$$

上述公式是胜率更新的公式，由于胜率与 playouts 的次数正相关，所以选择一个 playouts 次数最多的节点作为 best_move，返回。在此处需要注意一点，由于权值合并了后，playouts的 total 总数发生了变化，对 total 做后处理，total = total / reply_num。

十一、问题杂记

1. UCT 中有两类策略，**配置策略**(policy)和**随机策略**(random_policy)，其中配置策略包含两种：ucb1 和 ucb1amaf。如果没有设置**配置策略**，则默认的配置策略是 ucb1amaf。
2. UCT 中 playout 策略默认是 playout_moggy
3. 如果 UCT 引擎处于 slave 位置，则设置共享节点数量为 1408M，并且其中会设置一个 shared_levels，只会共享 level <= shared_levels 的节点，默认值为1。
4. UCT 引擎在初始化的时候，不会立马初始化棋盘状态树，因为不知道根节点的颜色，当第一步棋下完以后，才会初始化棋盘状态树以及相关的信息。
5. tree 中使用 nodes 作为节点的内存池，对于内存池的使用，主要是为了加快速度，并且提供 fast init 的方式。
6. 如果上一手棋是『过』，则只扩展当前一层节点，这个策略为什么成立— 经过『专业』棋手分析认为合理。
7. tree 中提供了调整树的方法，tree_promote_at，根据确定的一步落棋，调整树的存储结构，将代表相应局面的子节点调整成根节点，在 UCT 引擎作为 slave（主要功能）时，接到下棋信息后，调整自己的树信息，在调整之前，需要停止 pondering，调整完后，再打开 pondering 线程。
8. 在 distributed 模块内 merge 用来管理作为 slave 的 UCT 之间的共享节点问题。

十二、TODO

1. tree_node 中 depth 和 d 的区别
2. dead_group_list 在引擎框架中的作用