ArduCopter-3.2.1 源码解读

本文针对多轴飞行器最简单常见的 stabilize_run 模式（即自稳模式）的代码进行解读。

首先，在进行代码解读之前，首先我们要先想好，一个飞行器要工作在 stabilize_run 模式，需要什么输入，最后输出了什么东西。很容易地，我们知道在 stabilize_run 模式下，飞行器要周期性地检测陀螺仪信号来更新计算目前的角度，检测加速度计和电子罗盘来修正分别修正 pitch、roll 和 Yaw；同时要检测遥控器的输入，转化为相应的油门值、pitch 目标值、roll 目标值、Yaw 转动速度目标值。然后把上面的输入进行 PID 运算，最后输出各个电机的 PWM 值。

因此，我们需要从 ArduCopter-3.2.1 源码中找到下面的程序：

a. 程序是如何周期性运行的；
b. 程序在哪里更新各个传感器的值（加速度计、陀螺仪、电子罗盘）；
c. 飞行器姿态解算；
d. 遥控器的输入是在哪些程序检测，并转换成 pitch、roll、yaw 目标值的；
e. PID 的计算；
f. 电机 PWM 值输出；

下面，我们根据上面的思路找到相应的程序并进行解读。

程序的周期性运行

首先，我们先了解程序是如何周期性运行的。在 ArduPilot 官网的“Scheduling Code to Run Intermittently (Code Overview)”做出了介绍（网址：http://dev.ardupilot.com/wiki/apmcopter-code-overview/code-overview-scheduling-your-new-code-to-run-intermittently/）。要让程序周期性运行，有两种办法，一种是在 AP_Scheduler::Task scheduler_tasks[] PROGMEM 列表里面添加函数，并定义多少时间运行一次以及设置超时，另一种是在 fast_loop() 函数里面新增代码。

scheduler_tasks[] 可以在 ArduCopter.cpp 里找到：

```
/*
 scheduler table for fast CPUs - all regular tasks apart from the fast_loop()
 should be listed here, along with how often they should be called
 (in 2.5ms units) and the maximum time they are expected to take (in
 microseconds)
 1   = 400hz
 2   = 200hz
 4   = 100hz
 8   = 50hz
 20   = 20hz
 40   = 10hz
 133  = 3hz
 400  = 1hz
 4000 = 0.1hz

 */
const AP_Scheduler::Task Copter::scheduler_tasks[] PROGMEM = {
    { SCHED_TASK(rc_loop),              4,   130 },
    { SCHED_TASK(throttle_loop),        8,    75 },
    { SCHED_TASK(update_GPS),           8,   200 },
#if OPTFLOW == ENABLED
    { SCHED_TASK(update_optical_flow),  2,   160 },
#endif
    { SCHED_TASK(update_batt_compass), 40,   120 },
    { SCHED_TASK(read_aux_switches),   40,    50 },
    { SCHED_TASK(arm_motors_check),    40,    50 },
    { SCHED_TASK(auto_trim),           40,    75 },
    { SCHED_TASK(update_altitude),     40,   140 },
    { SCHED_TASK(run_nav_updates),      8,   100 },
    { SCHED_TASK(update_thr_average),   4,    90 },
    { SCHED_TASK(three_hz_loop),      133,    75 },
    { SCHED_TASK(compass_accumulate),   8,   100 },
    { SCHED_TASK(barometer_accumulate), 8,    90 },
#if FRAME_CONFIG == HELI_FRAME
```

从里面我们可以看到，scheduler_tasks 的程序，最快执行周期是 2.5ms（硬件对应 pixhawk）。并且里面的程序包括 rc_loop 和 throttle_loop ，这两个程序就是从遥控器接收油门、yaw、pitch、roll 控制量的程序（不过我没细看这两个程序，也没想明白为什么没把 throttle 当做遥控器的一部分，高那么复杂）。其中 rc_loop 的执行周期是 4*2.5=10ms。

另外一个周期性函数，就是 fast_loop() 了。在 ArduCopter.cpp 我们可以找到它，它的执行时间是 2.5ms。

```
// Main loop - 400hz
void Copter::fast_loop()
{

    // IMU DCM Algorithm
    // --------------------
    read_AHRS();

    // run low level rate controllers that only require IMU data
    attitude_control.rate_controller_run();

#if FRAME_CONFIG == HELI_FRAME
    update_heli_control_dynamics();
#endif //HELI_FRAME

    // send outputs to the motors library
    motors_output();

    // Inertial Nav
    // --------------------
    read_inertia();

    // check if ekf has reset target heading
    check_ekf_yaw_reset();

    // run the attitude controllers
    update_flight_mode();

    // update home from EKF if necessary
    update_home_from_EKF();

    // check if we've landed or crashed
    update_land_and_crash_detectors();

    // log sensor health
    if (should_log(MASK_LOG_ANY)) {
        Log_Sensor_Health();
    }
} ? end fast_loop ?
```

更新传感器和更新姿态

程序周期性运行的问题解决了，接下来我们找程序在哪里更新各个传感器的值，在哪里更新姿态、进行 PID 运算以及输出 pwm。其实在程序 fast_loop() 里面，就找到了。read_AHRS() 就是更新传感器并更新姿态的函数；attitude_control.rate_controller_run() 是进行角速度 PID 运算的函数；motors_output() 是输出电机 PWM 值的函数。下面分别对这些函数作一一分解并进行分析。

read_AHRS()里面调用的是 ahrs.update()，ahrs 是类 AP_AHRS_DCM的一个实例，因此我们在 AP_AHRS_DCM.cpp找到了 AP_AHRS_DCM::update(void)的定义：

```
AP_AHRS_DCM::update(void)
{
    float delta_t;

    if (_last_startup_ms == 0) {
        _last_startup_ms = hal.scheduler->millis();
    }

    // tell the IMU to grab some data
    _ins.update();

    // ask the IMU how much time this sensor reading represents
    delta_t = _ins.get_delta_time();

    // if the update call took more than 0.2 seconds then discard it,
    // otherwise we may move too far. This happens when arming motors
    // in ArduCopter
    if (delta_t > 0.2f) {
        memset(&_ra_sum[0], 0, sizeof(_ra_sum));
        _ra_deltat = 0;
        return;
    }

    // Integrate the DCM matrix
    matrix_update(delta_t);

    // Normalize the DCM matrix
    normalize();

    // Perform drift correction
    drift_correction(delta_t);

    // paranoid check for bad values
    check_matrix();

    // Calculate pitch, roll, yaw for stabilization and navigation
    euler_angles();

    // update trig values including _cos_roll, _cos_yaw
    update_trig();
} // end update
```

更新加速度计陀螺仪

使用陀螺仪更新四元素矩阵

使用加速度计和罗盘与该次计算的四元素矩阵做差，修正下一次的陀螺仪输出。

转换成欧拉角

AP_AHRS_DCM::update(void)里面，使用 _ins.update() 来更新陀螺仪加速度计。在这个函数里面，调用 _backends[i]->update(); 这里才是更新陀螺仪加速度计动作的函数。但是很多人在这里就无法进一步往下看了，因为 AP_InertialSensor_Backend 的 update() 是没有进一步定义的。我们去找 _backends 的定义：AP_InertialSensor_Backend *_backends[INS_MAX_BACKENDS]可以看到 _backends[] 是个指针数组，我们找到这些指针指向哪些变量，就可以找到该变量对应的类里面的 update() 函数了（这个不得不表达一下对 ArduCopter 源码的怨恨，太多的嵌套，太多的子类，找个函数像找迷宫一样。我无法想象结构这么不清晰的程序是怎么维护的）。通过搜索 _backends，在 AP_InertialSensor.cpp 找到了 void AP_InertialSensor::_add_backend(AP_InertialSensor_Backend *(detect)(AP_InertialSensor &)) 函数的定义，再找到了 AP_InertialSensor::_detect_backends(void) 函数里进行了 _add_backend(AP_InertialSensor_PX4::detect); 的操作。因此 _backends 指向的是 AP_InertialSensor_PX4，最后，我们在 AP_InertialSensor_PX4.cpp 里面找到了 bool AP_InertialSensor_PX4::update(void) 的定义：（使用 _publish_accel 等函数更新传感器的值）

```
bool AP_InertialSensor_PX4::update(void)
{
    // get the latest sample from the sensor drivers
    _get_sample();

    for (uint8_t k=0; k<_num_accel_instances; k++) {
        Vector3f accel = _accel_in[k];
        // calling _publish_accel sets the sensor healthy,
        // so we only want to do this if we have new data from it
        if (_last_accel_timestamp[k] != _last_accel_update_timestamp[k]) {
            _publish_accel(_accel_instance[k], accel, false);
            _publish_delta_velocity(_accel_instance[k], _delta_velocity_accumulator[k], _delta_velocity_dt[k]);
            _last_accel_update_timestamp[k] = _last_accel_timestamp[k];
        }
    }

    for (uint8_t k=0; k<_num_gyro_instances; k++) {
        Vector3f gyro = _gyro_in[k];
        // calling _publish_accel sets the sensor healthy,
        // so we only want to do this if we have new data from it
        if (_last_gyro_timestamp[k] != _last_gyro_update_timestamp[k]) {
            _publish_gyro(_gyro_instance[k], gyro, false);
            _publish_delta_angle(_gyro_instance[k], _delta_angle_accumulator[k]);
            _last_gyro_update_timestamp[k] = _last_gyro_timestamp[k];
        }
    }

    for (uint8_t i=0; i<INS_MAX_INSTANCES; i++) {
        _delta_angle_accumulator[i].zero();
        _delta_velocity_accumulator[i].zero();
        _delta_velocity_dt[i] = 0.0f;
    }

    if (_last_accel_filter_hz != _accel_filter_cutoff()) {
        _set_accel_filter_frequency(_accel_filter_cutoff());
        _last_accel_filter_hz = _accel_filter_cutoff();
    }

    if (_last_gyro_filter_hz != _gyro_filter_cutoff()) {
        _set_gyro_filter_frequency(_gyro_filter_cutoff());
        _last_gyro_filter_hz = _gyro_filter_cutoff();
    }

    return true;
} ? end update ?
```

绕迷宫绕了大半圈，终于把加速度计陀螺仪的检测程序稍微读懂了。下面回到函数 AP_AHRS_DCM::update(void)，继续往下看程序是怎样更新姿态的。 _ins.update() 获取完传感器的值后， matrix_update(delta_t) 就是使用刚刚测量出来的陀螺仪值、以及上个周期对陀螺仪的补偿值进行角度更新的函数， normalize() 用来做余弦矩阵的归一化， drift_correction() 则是使用电子罗盘、加速度计、和 GPS来计算角度误差，并更新 _omega（_omega 会在下一个周期的 matrix_update(delta_t) 中被使用），其详细的算法介绍请查看，本文不再详述：

http://wenku.baidu.com/link?url=ikm5-xBTm_-0wxQj2EMp7qZwtPeAwWFsQbZrwGBDrLH_c3g69ljtSGNFUd3JSR5AlRcTKzf_yQElVkXIIdduQT6HOrjT4OAQl50BpRZcQZK上面工作完成后，在把方向余弦矩阵转换为欧拉角（函数 euler_angles(); ），生成我们想要的 pitch、roll、yaw。至此，姿态换算完毕。

计算电机输出

看完姿态换算，我们回到 fast_loop() 函数。 read_AHRS() 接着的是函数 attitude_control.rate_controller_run() 。该函数就是进入角速度 PID 运算的入口。打开函数 AC_AttitudeControl::rate_controller_run() ，我们进入下一轮不断嵌套来嵌套去的迷宫。

```
void AC_AttitudeControl::rate_controller_run()
{
    // call rate controllers and send output to motors object
    // To-Do: should the outputs from get_rate_roll, pitch, yaw be int
    // To-Do: skip this step if the throttle out is zero?
    _motors.set_roll(rate_bf_to_motor_roll(_rate_bf_target.x));
    _motors.set_pitch(rate_bf_to_motor_pitch(_rate_bf_target.y));
    _motors.set_yaw(rate_bf_to_motor_yaw(_rate_bf_target.z));
}
```

以_motors.set_roll(rate_bf_to_motor_roll(_rate_bf_target.x)) 为例，rate_bf_to_motor_roll() 是使用当前的 _ahrs.get_gyro().x 与_rate_bf_target.x 进行 PID 运算，最终把运算结果赋值给 _roll_control_input 。 而_rate_bf_target.x 是怎么来的呢，它是在 void Copter::stabilize_run() 里面通过下面的语句而获得的（简单的说， 就是对目标角度和当前做 PID 运算的出的值，即双环结果的外环输出） ：

```
get_pilot_desired_lean_angles(channel_roll->control_in, channel_pitch->control_in, target

// get pilot's desired yaw rate
target_yaw_rate = get_pilot_desired_yaw_rate(channe

// get pilot's desired throttle
pilot_throttle_scaled = get_pilot_desired_throttle(channel_t

// call attitude controller
attitude_control.angle_ef_roll_pitch_rate_ef_yaw_smooth(target_roll, target_pitch, target_y
```

读遥控器pitch/roll信号

读遥控器yaw信号

读遥控器油门信号

角度误差PD运算

void Copter::stabilize_run() 因为是在后面会介绍到的， 所以这里先不在详述。 只是要感慨一下，为什么 ardupilot 的程序不按照一般人的思维，先算完外环，再算内环，程序搞得人绕来绕去。

回到 Copter::fast_loop() ， attitude_control.rate_controller_run() 运行完后，我们获得了 _roll_control_input 、_pitch_control_input 、_yaw_control_input ，接下来， 程序 motors_output() 就是使用上面三个值进行换算，得出各个电机的 PWM 值。继续绕迷宫：Copter::motors_output() 引用 motors.output() ， motors 属于 AP_MotorsQuad 类， AP_MotorsQuad 又继承于 AP_MotorsMatrix ， AP_MotorsMatrix 又继承于 AP_MotorsMulticopter ，然后终于找到 AP_MotorsMulticopter::output() 了；迷宫还没走完， 还要接着绕： AP_MotorsMulticopter::output() 最终运行的是 output_armed_stabilizing() ，这个函数在 AP_MotorsMatrix::output_armed_stabilizing() 找到了定义， 该函数的重点是下面， 就是每个电机根据飞行机型的设置， 对 roll_pwm 、pitch_pwm 、yaw_pwm 进行一定系数的加减乘：

```
for (i=0; i<AP_MOTORS_MAX_NUM_MOTORS; i++) {
    if (motor_enabled[i]) {
        rpy_out[i] = roll_pwm * _roll_factor[i] * get_compensation_gain() +
                     pitch_pwm * _pitch_factor[i] * get_compensation_gain();

        // record lowest roll pitch command
        if (rpy_out[i] < rpy_low) {
            rpy_low = rpy_out[i];
        }
        // record highest roll pich command
        if (rpy_out[i] > rpy_high) {
            rpy_high = rpy_out[i];
        }
    }
}
```

```
for (i=0; i<AP_MOTORS_MAX_NUM_MOTORS; i++) {
    if (motor_enabled[i]) {
        rpy_out[i] =    rpy_out[i] +
                    yaw_allowed * _yaw_factor[i];

        // record lowest roll+pitch+yaw command
        if( rpy_out[i] < rpy_low ) {
            rpy_low = rpy_out[i];
        }
        // record highest roll+pitch+yaw command
        if( rpy_out[i] > rpy_high) {
            rpy_high = rpy_out[i];
        }
    }
}
```

最后送到电机上：

```
for (i=0; i<AP_MOTORS_MAX_NUM_MOTORS; i++) {
    if (motor_enabled[i]) {
        motor_out[i] = out_best_thr_pwm+thr_adj +
                    rpy_scale*rpy_out[i];
    }
}
```

飞行模式判断， **pitch、roll** 等目标设定

回到 Copter::fast_loop()， motors_output() 后面是 read_inertia()、check_ekf_yaw_reset()，这两个函数没来得及分析，本文跳过。

接下来是 update_flight_mode()。前面都比较简单我们直接跳进 Copter::stabilize_run() 去分析。 stabilize_run() 这个函数大部分工作其实都是在更新和计算目标值 _rate_bf_target。其中，update_simple_mode() 是读取遥控器的信号， 该函数里面的 channel_roll->control_in 是通过如下的迷宫获取的： Copter::rc_loop()（10ms 执行一次） 里面的 Copter::read_radio() 里面的 RC_Channel::set_pwm_all 的 rc_ch[i]->set_pwm(rc_ch[i]->read()) , RC_Channel::set_pwm(int16_t pwm) 里面的 control_in = pwm_to_range() 或 control_in = pwm_to_angle()， pwm_to_range() 里面的 pwm_to_range_dz(_dead_zone)，最终返回 return (_low + ((int32_t)(_high - _low) * (int32_t)(r_in - radio_trim_low)) / (int32_t)(radio_max - radio_trim_low)) 。（例如对于油门，往上拨动到最多，返回 1000，往下拨到最小，返回 _low）

接着，在 update_simple_mode() 中根据 simple_mode 对 control_in 进行进一步换算：

a. 在默认情况下， simple_mode 为 0，不作任何处理，一直以飞机的朝向来定 roll 和 pitch 方向；
b. 当 simple_mode 为 1 时，则以刚起飞时飞行器面向的方向为参考计算 roll 和 pitch 期望值。例如刚起飞的时候飞行器朝向正北， 那么起飞以后， 无论飞行器跑到哪里，朝向改到哪里 （朝向改到东西南都一样） ，我们调整遥控器的 roll 摇杆就是往东或者西方向走，调整遥控器的 pitch 摇杆就是往南或者北方向走。这种模式就是方便操控者一直面向一个方向，不需要考虑飞行器本身的方向。
c. 当 simple_mode 大于 1 时，则以飞行器目前的坐标位置与起飞坐标位置的连线来计算 roll 和 pitch 期望值。例如某一时刻飞行器飞到起飞坐标的北边，那么我们调整遥控器的 roll 摇杆就是往东或者西方向走，调整遥控器的 pitch 摇杆就是往南或者北方向走；过一会飞行器飞到起飞坐标的西边，那么我们调整遥控器的 roll 摇杆就是往南或者北方向走，调整遥控器的 pitch 摇杆就是往东或者西方向走。这种模式方便操控者不断地面向飞机来操作。

具体实现就是作正弦余弦换算，弄明白上面的目的，推导起来就比较简单：

```
if (ap.simple_mode == 1) {
    // rotate roll, pitch input by -initial simple heading (i.e. north facing)
    rollx = channel_roll->control_in*simple_cos_yaw - channel_pitch->control_in*simple_sin_yaw;
    pitchx = channel_roll->control_in*simple_sin_yaw + channel_pitch->control_in*simple_cos_yaw;
}else{
    // rotate roll, pitch input by -super simple heading (reverse of heading to home)
    rollx = channel_roll->control_in*super_simple_cos_yaw - channel_pitch->control_in*super_simple_sin_yaw;
    pitchx = channel_roll->control_in*super_simple_sin_yaw + channel_pitch->control_in*super_simple_cos_yaw;
}

// rotate roll, pitch input from north facing to vehicle's perspective
channel_roll->control_in = rollx*ahrs.cos_yaw() + pitchx*ahrs.sin_yaw();
channel_pitch->control_in = -rollx*ahrs.sin_yaw() + pitchx*ahrs.cos_yaw();
```

回到 Copter::stabilize_run() 再继续往下看， get_pilot_desired_lean_angles() 是把遥控器的刻度转换为角度的刻度； target_yaw_rate = get_pilot_desired_yaw_rate(channel_yaw->control_in) 是对遥控器 yaw 的输入进行换算； pilot_throttle_scaled = get_pilot_desired_throttle(channel_throttle->control_in) 是对遥控器 throttle 的输入进行换算（中间设为死区） 。

接着是 attitude_control.angle_ef_roll_pitch_rate_ef_yaw_smooth(target_roll, target_pitch, target_yaw_rate, get_smoothing_gain()) 这个函数，这个函数做了几个工作：

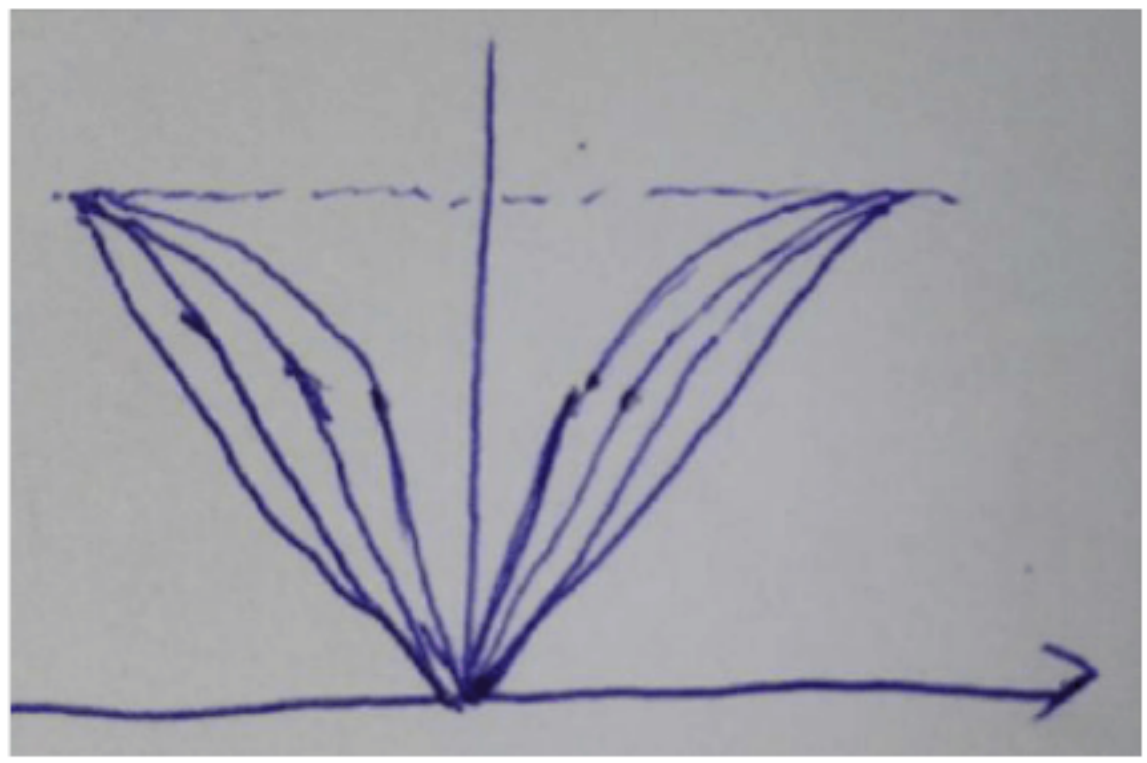
1、 对遥控器 roll/pitch 信号进行软硬处理

2、 对遥控器 Yaw的输入进行处理

3、 坐标系转换

4、 对角度差进行 PD运算，生成内环的参考；

一开始的时候，判断 _accel_roll_max、_rate_bf_ff_enabled 等变量，如果为 1，则对遥控器 roll、pitch 的信号做了处理， 使用 sqrt_controller() 函数改变摇杆输出曲线的 “软硬”程度，并使用 constrain_float() 和 rate_change_limit 限制 roll/pitch 的角加速度。 算法使用的是先比例再反抛物线（开根号）的算法，如下图所示， smoothing_gain 越大，则曲线越靠近里面，即摇杆动很小就有较大的输出。如果判断为否，则直接计算角度差。



对遥控器 Yaw的输入进行处理思路是， 使用 update_ef_yaw_angle_and_error 这个函数，里面这条语句 _angle_ef_target.z += yaw_rate_ef （对应遥控器 yaw 摇杆） * _dt; 设定 yaw 的目标角度，即当 yaw 摇杆偏向一边的时候 _angle_ef_target.z 会不断增大或者减小，从而实现飞行器 yaw 的控制。其他语句，基本上的作用都是为了限幅（ if (_accel_yaw_max > 0.0f) 做角加速度限幅、 update_ef_yaw_angle_and_error 作角度误差限幅） 。

这里对 yaw 的控制多做一些分析。理论上，当遥控器 yaw 摇杆在中间不动的时候，yaw_rate_ef 应该为 0，那么 _angle_ef_target.z 也是一直为 0，即 yaw 摇杆不动的时候飞行

器是不应该旋转 yaw 角的。但是实测发现飞行器不平衡或者起飞的时候有东西弄转了偏航角的话，飞行器是没有调整回去的。我猜测是 update_ef_yaw_angle_and_error 里面的限幅语句导致的。下图的语句运行完，按理来说 angle_ef_target.z 一直为 0（yaw 摇杆不动的情况下），但是第二行语句的存在，会使得当飞行器被外力导致 yaw 旋转超过 overshoot_max 的话， angle_ef_error.z 就不等于 - _ahrs.yaw_sensor，第三行语句运行完， _angle_ef_target.z 也无法恢复到 0 了。

```
angle_ef_error.z = wrap_180_cd( angle_ef_target.z - _ahrs.yaw_sensor);
angle_ef_error.z = constrain_float(angle_ef_error.z, -overshoot_max, overshoot_max);

// update yaw angle target to be within max angle overshoot of our current heading
_angle_ef_target.z = angle_ef_error.z + _ahrs.yaw_sensor;

// increment the yaw angle target
_angle_ef_target.z += yaw_rate_ef * _dt;
```

Roll 、 Pitch 、 Yaw 的角度误差都计算完毕后，进行了坐标系转换：frame_conversion_ef_to_bf(angle_ef_error, _angle_bf_error) 。一开始我没有想明白，但是后来再仔细想想， 飞行器测出来的 Roll、Pitch、Yaw 的角速度， 是以飞行器本身为参考系的，例如，当飞行器水平的时候， yaw 角速度跟地理坐标轴定义的 yaw 是一样的，但是当飞行器 pitch 已经前倾了 45 度的情况下， 飞行器 yaw 角速度跟地理坐标轴定义的 yaw 方向已经相差了 45 度。因此两者之间需要做坐标轴换算。

接下来，程序使用 update_rate_bf_targets(); 函数对角度误差进行 PID 的 P 运算，生成前面所提到的 _rate_bf_target.x/y/z 。里面的判断用于选择是纯比例计算还是比例 +反抛物线计算（函数 sqrt_controller() ）。

最后，程序根据 _rate_bf_ff_enabled 判断是否对角度误差添加前馈（类似于 PID 的 D 运算）。