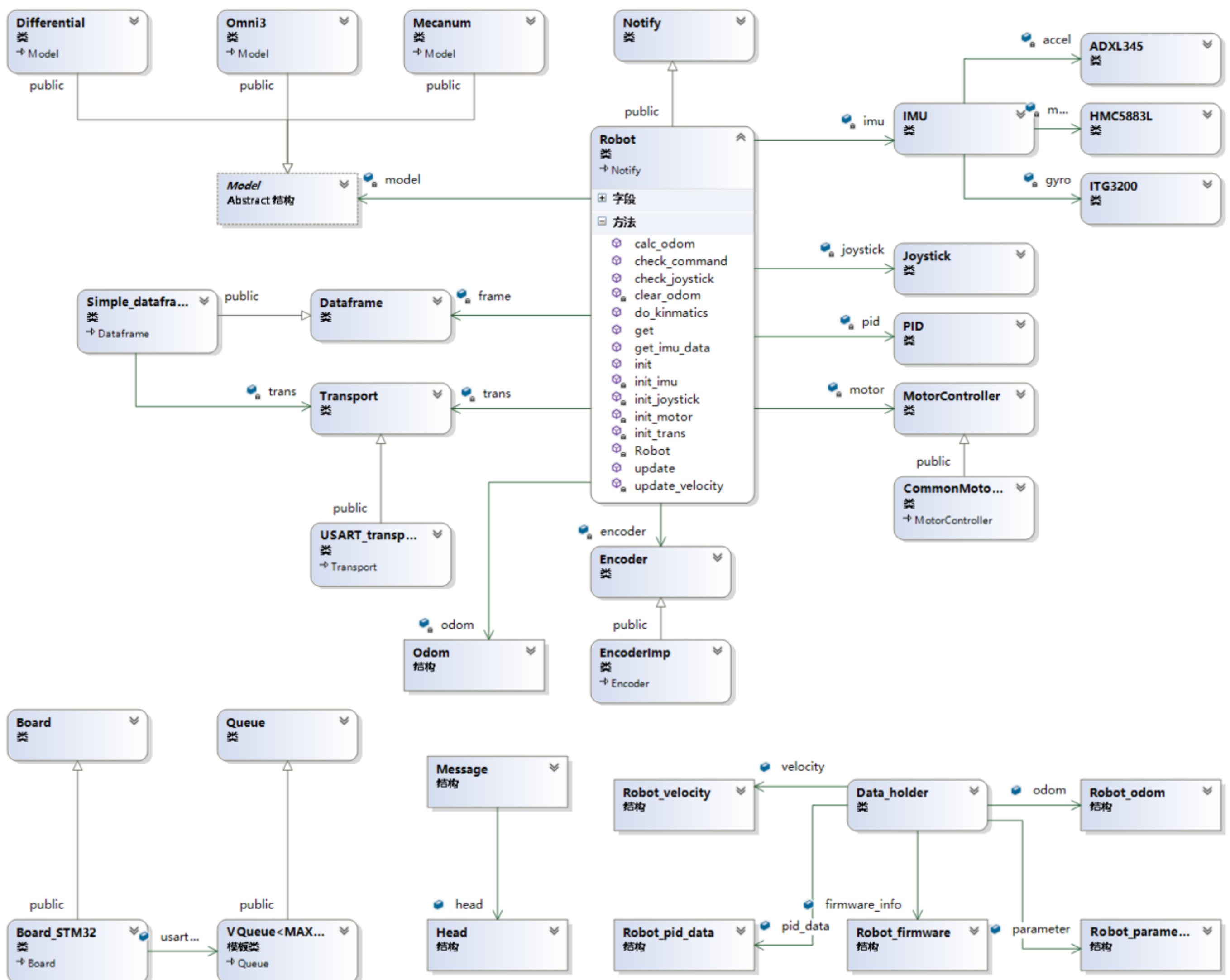
- 1.Robot类
- 2.Robot::init
    - Robot中的接口实例化
- 3.运动解算
- 4.PID
- 5.wheel odom

1.Robot类

先来张类关系图



在贴个`main`的代码

```
#include "robot.h"

int main(void)
{
        Robot::get()->init();

        while (1)
        {
```

```
            Robot::get()->check_command();
            Robot::get()->do_kinmatics();
            Robot::get()->calc_odom();
            Robot::get()->get_imu_data();
            Robot::get()->check_joystick();
        }
}
```

`Robot`为一个单例模式类，通过他完成大部分工作, `main`函数中可以看到其功能:

- 初始化
- 检测命令
- 运动解算
- 计算里程
- 获取imu数据
- 检查遥控

与之关联的类有:

- `Model` 这是个运动模型接口类，可以看到该类有几个派生类`Differential` `Omni3` `Mecanum` 分别代表对不同机器人模型的解算
- `Dataframe` 协议接口类，通过继承该类可以使用不同的协议进行通讯，这里实现了一个`Simple_dataframe`的派生类
- `Transport` 通讯接口类，通过继承该类可以使用不同的通讯口接口通讯这里实现了一个`USART_transport`的派生类，显然这个`USART_transport`是使用串口通讯
- `MotorController` 电机控制接口，派生出一个名为`CommonMotorController`类控制
- `PID` 显然是`PID`控制类
- `Encoder` 是编码器读取类
- `Joystick` ps2遥控器
- `IMU` 惯性测量单元相关 同时`Robot`从`notify`继承，实现了一个观察者模式实现对某一个或者多个事件的观察，这里是对消息的关注，即收到消息会通知到`Robot`，通过`Dataframe::register_notify`注册

另外有另外几个单例类

- `Board` 该类维护bsp相关的接口
- `Queue` 实现了一个简单的队列
- `Data_holder`该类维护这多个与通讯业务相关的类对象

2.Robot::init

```
void Robot::init(){
    Board::get()->init();

#if DEBUG_ENABLE
    Board::get()->usart_debug_init();
#endif

    Data_holder::get()->load_parameter();

#if DEBUG_ENABLE
    printf("RobotParameters: %d %d %d %d %d %d %d %d %d %d %d %d\r\n",
        Data_holder::get()->parameter.wheel_diameter, Data_holder::get()->parameter.wheel_track,  Data_holder::get()->parameter.encoder_resolution,
        Data_holder::get()->parameter.do_pid_interval, Data_holder::get()->parameter.kp, Data_holder::get()->parameter.ki, Data_holder::get()->parameter.kd, Data_holder::get()->parameter.ko,
        Data_holder::get()->parameter.cmd_last_time, Data_holder::get()->parameter.max_v_liner_x, Data_holder::get()->parameter.max_v_liner_y, Data_holder::get()->parameter.max_v_angular_z);
#endif

#if IMU_ENABLE
    init_imu();
#endif

#if DEBUG_ENABLE
    printf("init_motor\r\n");
#endif

    init_motor();

#if DEBUG_ENABLE
    printf("init_trans\r\n");
#endif

    init_trans();

    init_joystick();

#if DEBUG_ENABLE
    printf("pibot startup\r\n");
#endif
}
```

这里主要完成对各个部件的初始化

Robot中的接口实例化

```
private:
    MotorController* motor[MOTOR_COUNT];
    Encoder*         encoder[MOTOR_COUNT];
    PID*             pid[MOTOR_COUNT];
    float            input[MOTOR_COUNT];
    float            feedback[MOTOR_COUNT];

    Transport*       trans;
    Dataframe*       frame;
    Model*           model;
```

Robot中维护的多为接口类的指针, 可以看到在下面的代码中完成对象的实例化

```
#if ROBOT_MODEL == ROBOT_MODEL_DIFF
    static Differential diff(Data_holder::get()->parameter.wheel_diameter*0.0005, Data_holder::get()->parameter.wheel_track*0.0005);
    model = &diff;
#endif

#if ROBOT_MODEL == ROBOT_OMNI_3
    static Omni3 omni3(Data_holder::get()->parameter.wheel_diameter*0.0005, Data_holder::get()->parameter.wheel_track*0.0005);
    model = &omni3;
#endif
```

根据配置的宏使用不同的运动模型， 其他接口的实例化也类似

3.运动解算

这里主要根据发送的速度得到各个轮子的转速，这部分工作是在`Robot::update_velocity` 而该函数为作为

```
frame->register_notify(ID_SET_VELOCITY, this);
```

对设置速度消息的关注的回调，该回调函数中

```
void Robot::update_velocity(){
    short vx = min(max(Data_holder::get()->velocity.v_liner_x, -
```

```
(short(Data_holder::get()->parameter.max_v_liner_x))), short(Data_holder::get()->parameter.max_v_liner_x));
    short vy = min(max(Data_holder::get()->velocity.v_liner_y, -(short(Data_holder::get()->parameter.max_v_liner_y))), short(Data_holder::get()->parameter.max_v_liner_y));
    short vz = min(max(Data_holder::get()->velocity.v_angular_z, -(short(Data_holder::get()->parameter.max_v_angular_z))), short(Data_holder::get()->parameter.max_v_angular_z));

    float vel[3]={vx/100.0, vy/100.0, vz/100.0};
    float motor_speed[MOTOR_COUNT]={0};
    model->motion_solver(vel, motor_speed);


    for(int i=0;i<MOTOR_COUNT;i++){
        input[i] = motor_speed[i]*short(Data_holder::get()->parameter.encoder_resolution)/(2*__PI)*short(Data_holder::get()->parameter.do_pid_interval)*0.001;
    }

#if DEBUG_ENABLE
    printf("vx=%d %d motor_speed=%ld %ld\r\n", vx, vz, long(motor_speed[0]*1000), long(motor_speed[1]*1000));
#endif

    last_velocity_command_time = Board::get()->get_tick_count();
    do_kinmatics_flag = true;
}
```

- 首先对速度限制做判断

  ```
  short vx = min(max(Data_holder::get()->velocity.v_liner_x, -(short(Data_holder::get()->parameter.max_v_liner_x))), short(Da
  short vy = min(max(Data_holder::get()->velocity.v_liner_y, -(short(Data_holder::get()->parameter.max_v_liner_y))), short(Da
  short vz = min(max(Data_holder::get()->velocity.v_angular_z, -(short(Data_holder::get()->parameter.max_v_angular_z))), shor
  ```

- 然后通过运动模型类解算得到轮子的速度`motor_speed`

  ```
  float vel[3]={vx/100.0, vy/100.0, vz/100.0};
  float motor_speed[MOTOR_COUNT]={0};
  model->motion_solver(vel, motor_speed);
  ```

- 在转换到`pid`间隔时间（`Data_holder::get()->parameter.do_pid_interval`）内各个电机行径的编码器值，得到一组`input`(多个电机)

  ```
  for(int i=0;i<MOTOR_COUNT;i++){
      input[i] = motor_speed[i]*short(Data_holder::get()->parameter.encoder_resolution)/(2*__PI)*short(Data_holder::get()->parameter.do_pid_interval)*0.001;
  }
  ```

4.PID

- 采集编码器在`pid`间隔时间（`Data_holder::get()->parameter.do_pid_interval`）走了多少，即一组 `feedback`(多个电机)

```
for(int i=0;i<MOTOR_COUNT;i++){
    feedback[i] = encoder[i]->get_increment_count_for_dopid();
}
```

- 有了输入`input`，反馈`feedback`我们既可以做`pid`运算了，得到的一组`output`即为控制电机的`pwm`（负为反向）

```
for(int i=0;i<MOTOR_COUNT;i++){
    output[i] = pid[i]->compute(Data_holder::get()->parameter.do_pid_interval*0.001);
}
```

- 控制电机输出

```
for(int i=0;i<MOTOR_COUNT;i++){
    motor[i]->control(output[i]);
}
```

5.wheel odom

- 计算一定间隔时间内(`CALC_ODOM_INTERVAL`这里固定为100ms)编码器改变值，得到一组`dis`

```
for(int i=0;i<MOTOR_COUNT;i++){
    dis[i] = encoder[i]->get_increment_count_for_odom()*__PI*Data_holder::get()->parameter.wheel_diameter*0.001/Data_holder::get()->parameter.encoder_resolution
_DEBUG_OUTPUT
    printf(" %ld ", long(dis[i]*1000000));

}
```

- 反解， 通过运动模型接口转换轮子的位移到机器人的里程

```
model->get_odom(&odom, dis, CALC_ODOM_INTERVAL);
```