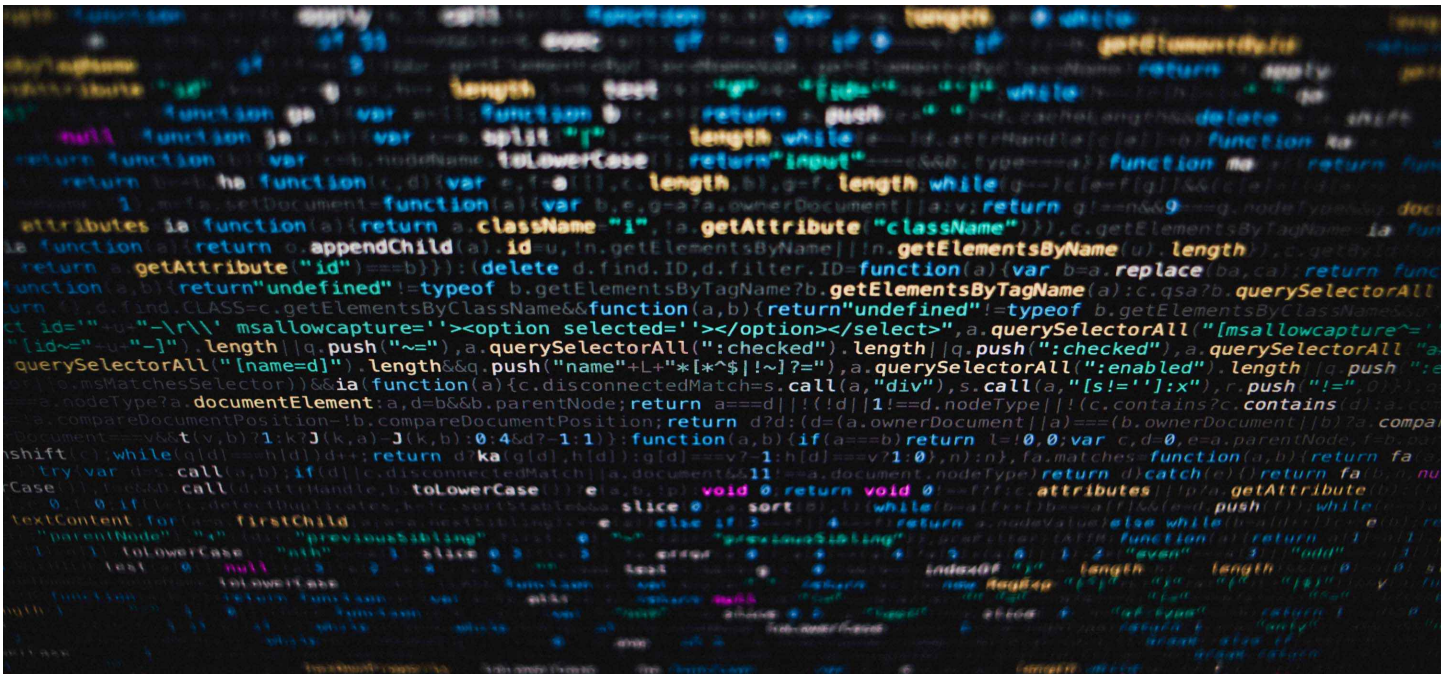

3 电机参数读取，电机数据储存，抽象硬件层，挂载电机，抽象硬件层运行流程

1、电机参数读取

上节课我们提到了四种在电机数据收发的时候会用到的参数。之前我们将他们作为c++文件中的一个变量记录下来，很明显这并不符合文件数据读写的逻辑——参数或系数更应该是作为一个外部加载的常量而不是变量。在c语言中，我们经常会使用 `#define` 命令定义宏变量。但当需要修改参数的时候，我们仍需要打开代码文件修改。我们希望有如下的体系结构：作为代码中参数存在的数据应该使用外挂文件储存，当需要的时候由 `c/c++` 文件读取参数。

巧合的是（bushi），ROS的结构正好就是以上提到的结构。我们使用 `config` 文件夹下的 `.yaml` 文件储存参数，并在c++文件运行的时候加载参数。

`yaml` 文件是一种以json格式编写，以树状形式编写。其形式如 `actuator_coefficient.yaml` 所示。

```yaml
actuator_coefficient:
  rm_3508: # without reducer, only the rotor
    act2pos: 0.0007669903 # 2PI/8192
    act2vel: 0.1047197551 # 2PI/60
    act2effort: 1.90702994e-5 # 20/16384*0.3/19.20320855615
    effort2act: 52437.561519 # tau to current
    max_out: 16384

```

```
  rm_2006:
    act2pos: 2.13078897e-5 # 2PI/8192*(1/36)
    act2vel: 0.0029088820 # 2PI/60*(1/36)
    act2effort: 0.00018 # 10/10000*0.18
    effort2act: 5555.5555555 # tau to current
    max_out: 10000
```

我们可以看到，上一节提到的几个参数都储存在这个文件中，等到我们后续需要的时候，就可以到这里调取所需的参数。

接下来我们进行电机参数读取。我们使用 `getParam("a", b)` 函数读取参数。其中 `a` 是参数的名字，其由 `yaml` 文件：前的部分决定，而 `b` 是要储存该参数的变量名，该变量类型需要和参数的类型相同。

以 `rm_3508` 电机的 `act2pos` 参数为例。该参数的值为double值，则需要一个double类型的变量来储存。需要注意的是该参数的名字并不是 `act2pos` ，而是 `actuator_coefficient/rm_3508/actpos`，`yaml` 文件的树结构也构成了像文件夹一样的命名规则，每一个参数都是在若干个“文件夹”（命名空间）下储存的，所以调用参数的时候必须加上命名空间的名字。

完整的写法如下。

```
ros::NodeHandle nh;
double act2pos;
nh.getParam("actuator_coefficient/rm_3508/actpos", act2pos);
```

由此理论，所有基本类型的变量（如int/double/char/string等）都可以如此方式被读取。那么那些更加复杂的变量应该怎么办呢。

事实上，`yaml` 文件的格式的树状结构是有对应数据结构的。该数据结构可由 `key: value` 的方式表示，称之为键值对（类似于python中的字典）。可以想像得到，在 `rm_3508:` 后的部分都以这种形式记录，那就可以以所谓键值对格式的变量储存。

我们调用了 `XmlRpc::XmlRpcValue` 来储存这种类型的数据。这是一种可以接受键值对类型数据的变量。

```
ros::NodeHandle nh;
XmlRpc::XmlRpcValue xml_rpc_value;
nh.getParam("actuator_coefficient/rm_3508", xml_rpc_value);
```

再深入想一想，键值对的值部分可以是普通类型的变量，可不可以是键值对类型的变量呢？当然可以。 `actuator_coefficient:` 后方的部分便是两个键值对变量 `rm_3508` 和 `rm_2006` ，这

两部分也可以成为键值对数据储存。

```
ros::NodeHandle nh;
XmlRpc::XmlRpcValue xml_rpc_value;
nh.getParam("actuator_coefficient", xml_rpc_value);
```

这部分实验请在 `catkin_ws` 下编译后先后运行 `roslaunch lesson3 load_coefficient.launch` (用以加载参数到参数服务器)， `rosrun lesson3 load_coefficient` 进行。

2、电机数据储存

接下来我们学习储存电机数据。之前我们读取的数据只是打印到终端中，而实际运行中我们肯定需要储存电机数据以备更高层控制程序使用。我们这里只需要关注4个最核心的参数，即电机当前状态的位置、速度、力矩和想要控制电机运行的力矩。

我们使用 `data_types.hpp` 文件中如下的结构体来储存电机数据：

```
struct ActDataSimple
{
    double pos, vel, effort;
    double cmd;
};
```

只要使用这个结构体生成一个实例，并每次在循环中更新该实例数据，就可以实现电机数据储存的效果。同时，我们可以把该指向该结构体的指针发给上层控制程序，以便其使用或改变数据。

我们对原有 `can_bus` 的 `void CanBus::read(ros::Time time)` 和 `void CanBus::write()` 函数稍作修改，这部分代码储存于 `can_bus_act_data.cpp` 中。

```
/**
 * @brief 按照rm电机的can通讯协议写入电机命令
 * @param
 */
void CanBus::write()
    {
        bool has_write_frame0 = false, has_write_frame1 = false;
        std::fill(std::begin(rm_can_frame0_.data), std::end(rm_can_frame0_.data)
        std::fill(std::begin(rm_can_frame1_.data), std::end(rm_can_frame1_.data)

```

```
11          for (auto &act_data : *id2act_data_)
12          {
13              int id = act_data.first - 0x200;
14              double cmd = limitAmplitude(eff2act  * act_data.second.cmd, 10000.0)
15              if (1 <= id && id <= 4)
16              {
17                  rm_can_frame0_.data[2 * (id - 1)] = static_cast<uint8_t>(static_
18                  rm_can_frame0_.data[2 * (id - 1) + 1] = static_cast<uint8_t>(cmd
19                  has_write_frame0 = true;
20              }
21              else if (5 <= id && id <= 8)
22              {
23                  rm_can_frame1_.data[2 * (id - 5)] = static_cast<uint8_t>(static_
24                  rm_can_frame1_.data[2 * (id - 5) + 1] = static_cast<uint8_t>(cmd
25                  has_write_frame1 = true;
26              }
27          }
28
29          if (has_write_frame0)
30              socket_can_.write(&rm_can_frame0_);
31          if (has_write_frame1)
32              socket_can_.write(&rm_can_frame1_);
33
34      }
```

注意 `Line11` 和 `Line14` 的部分。

```
1 for (auto &act_data : *id2act_data_)
```

这里的 `id2act_data_` 是 `CanBus` 类的私有变量，其类型为 `std::unordered_map<int, ActDataSimple> *`，这句话表示遍历 `id2act_data_` 指向的变量（该变量为 `std::unordered_map` 类型，可以粗略理解为一张哈希表，由多个键值对变量组成），并每次遍历时使用 `act_data` 来表示当前指向的键值对。

```
1 double cmd = limitAmplitude(eff2act  * act_data.second.cmd, 10000.0);
```

`act_data.second` 表示当前键值对的值部分。可以看出，这个值的类型为 `ActDataSimple`，所以可以使用 `.cmd` 调用该结构实例的 `cmd` 变量。

```
1 /**
```

```
 * @brief 对read_buffer_中的数据进行处理
 * @param time
 */
void CanBus::read(ros::Time time)
{
    std::lock_guard<std::mutex> guard(mutex_);
    for (const CanFrameStamp &can_frame_stamp : read_buffer_)
    {
        can_frame frame = can_frame_stamp.frame;

        //根据rm电机协议读取回传数据
        int id = frame.can_id;
        double q_raw = (frame.data[0] << 8u) | frame.data[1];
        double qd_raw = (frame.data[2] << 8u) | frame.data[3];
        int16_t mapped_current = (frame.data[4] << 8u) | frame.data[5];
        double temptrue = frame.data[6];

        //将数据转化为公制单位
        double position = act2pos * q_raw;
        double velocity = act2vel * qd_raw;
        double effort = act2eff * mapped_current;

        ActDataSimple &act_data = id2act_data_->find(id)->second;
        act_data.pos = position;
        act_data.vel = velocity;
        act_data.eff = effort;
    }
    read_buffer_.clear();
}
```

注意 `Line24:Line27` 的部分。

```
ActDataSimple &act_data = id2act_data_->find(id)->second;
act_data.pos = position;
act_data.vel = velocity;
act_data.eff = effort;
```

我们生成了一个 `ActDataSimple` 引用类型的变量 `act_data` ，并寻找 `id2act_data_` 各键值对中键为 `id` 到的键值对，并将该键值对的值赋给刚刚定义的变量。引用的特性告诉我们，这时该键值对的值和 `act_data` 指向同样的变量，当我们改变 `act_data` 的值时，刚才键值对中的值也会随之改变。

这里我们将转换完成的电机数据 `position` `velocity` `effort` 赋给 `act_data` 中的对应变量，同时也意味着改变了 `id2act_data_` 中对应键值对的值的数值。

由此，我们将电机的回传数据储存在`id2act_data_`中，也从`id2act_data_`中拿取电机控制数据。

还有一个问题，电机的控制数据在什么时候放到`id2act_data_`中的呢？请看`main.cpp`中的`int main(int argc, char **argv)`。

```
int main(int argc, char **argv)
{
    ros::init(argc, argv, "sp_hw");
    ros::NodeHandle nh;
    std::string bus_name = "can0";
    ros::Rate loop_rate(5);
    std::unordered_map<int, sp_hw::ActDataSimple> id2act_data_;

    for (int id = 1; id < 9; id++)
    {
        id2act_data_.emplace(std::make_pair(id + 0x200, sp_hw::ActDataSimple{
                                                        .pos = 0,
                                                        .vel = 0,
                                                        .eff = 0,
                                                        .cmd = 0}));
    }
    sp_hw::CanBus canbus(bus_name, &id2act_data_, 0);

    while(ros::ok())
    {
        canbus.read(ros::Time::now());
        for (auto &act_data : id2act_data_)
        {
            if (act_data.first == 0x201)
            {
                ROS_INFO_STREAM("----------------------------------");
                ROS_INFO_STREAM("id: " << std::hex << act_data.first);
                ROS_INFO_STREAM("pos: " << act_data.second.pos << "rad");
                ROS_INFO_STREAM("vel: " << act_data.second.vel << "rad/s");
                ROS_INFO_STREAM("eff: " << act_data.second.eff << "N.m");
                ROS_INFO_STREAM("cmd: " << act_data.second.cmd << "N.m");
                act_data.second.cmd = 0.015;
            }
        }
        canbus.write();
        loop_rate.sleep();
    }
    return 0;
}
```

`Line7:Line15` 初始化了一个 `std::unordered_map<int, sp_hw::ActDataSimple>` 类型的变量 `id2act_data_`，`Line17` 初始化了 `sp_hw::CanBus` 类型的实例 `canbus`，还将 `id2act_data_` 的地址传入该实例中。之后 `Line26:Line31` 打印了 `id2act_data_` 的数据，并在 `Line32` 为 `cmd` 变量赋值。数据由 `main` 函数传入 `canbus` 实例的 `id2act_data_` 变量。

这部分实验请在 `catkin_ws` 下编译后运行 `rosrun lesson3 can_bus_act_data` 进行。

3、抽象硬件层

为了解释什么是硬件抽象层，我们需要了解为什么要有抽象硬件层。首先从一个简单的例子入手：假如超级天才美少女陆竹翠有朝一日成为某跨国机器人公司"巨疆"（Jji）的CTO，作为她的工作，她需要向不同的研究团队发出研发指令。近日，她正频繁地与机械团队确认技术细节。但这只位于总部的机械臂团队最近因年龄原因喜提炒鱿鱼，因此只能由位于德国的团队接手工作。现在问题来了，作为CTO，陆竹翠的工作流程应该发生变化吗，她需要去学习德语吗，需要从线下通知变成邮件通知或者视频会议吗？

当然是不应该的，因为CTO有秘书，秘书小朴会帮他处理好这些事情，CTO仍然只需要像以前一样工作就行。将她的工作流程作图如下：

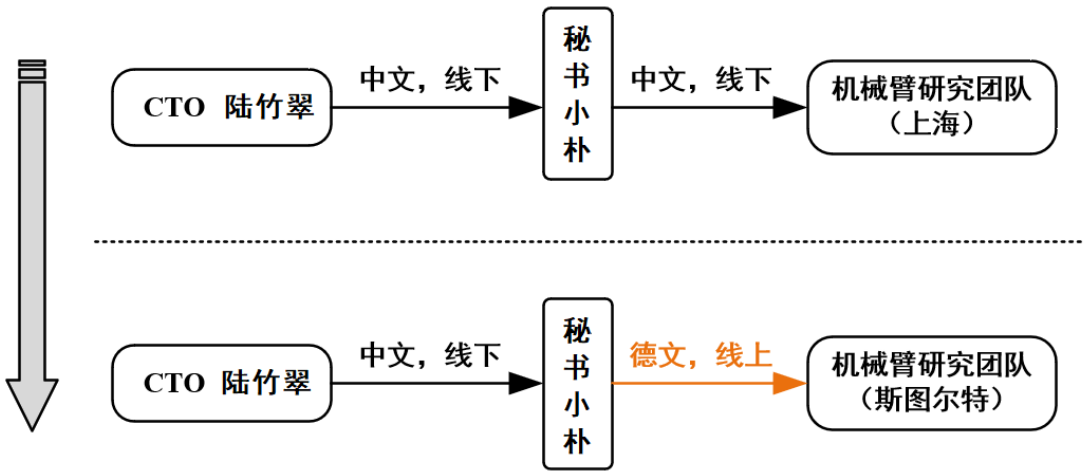

图1 能干的秘书小朴

可以看到秘书小朴实际上起到一个“转译”的作用，无论机械臂团队是在上海、德国还是日本，也不管是用邮件、线下、视频，小朴都会负责将CTO的指令传达给研究团队，CTO的工作流程并不会因为研究团队的变更而发生变化。讲完了这个，现在回顾一下第一次培训的内容：

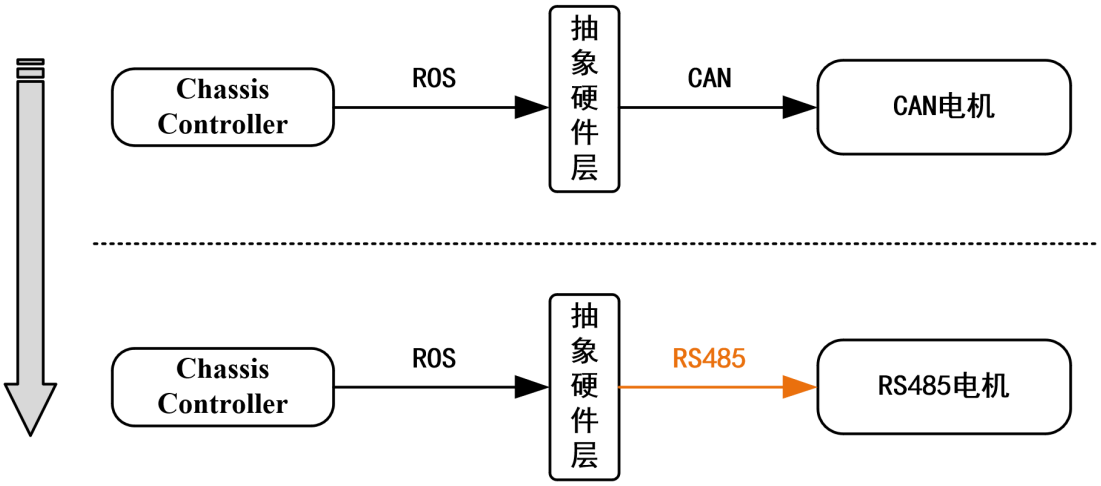

图2. 抽象硬件层示例

在图2中呈现了一种常见开发中遇到的问题，也即是需要更换驱动器，比如在该例中需要将 `CAN` 协议电机更换为 `RS485` 协议电机。显然，协议的更换不应该影响上层算法，就好像无论你是使用电动汽车还是燃油汽车，其自动导航的算法仍然是一致的。在开发中，我们也希望能保证算法和执行器部分没有直接的联系，因此引入了抽象硬件层。类比上文的例子，Chassis Controller就是CTO，而硬件抽象层就是秘书，Chassis Controller不会因为电机的变化而改变运行内容，就如同CTO的工作流程并不会因为研究团队的变更而发生变化。

抽象硬件层中“抽象”二字是相对上层控制器（controller）来说的。就如上文所述，算法无需考虑驱动，controller也不需要关心下层某个关节具体由什么电机执行，用什么协议，需要如何执行。上层controller向抽象硬件层发送的数据是关节的转动数据，而具体的运动指令由抽象硬件层转化给底层驱动器。

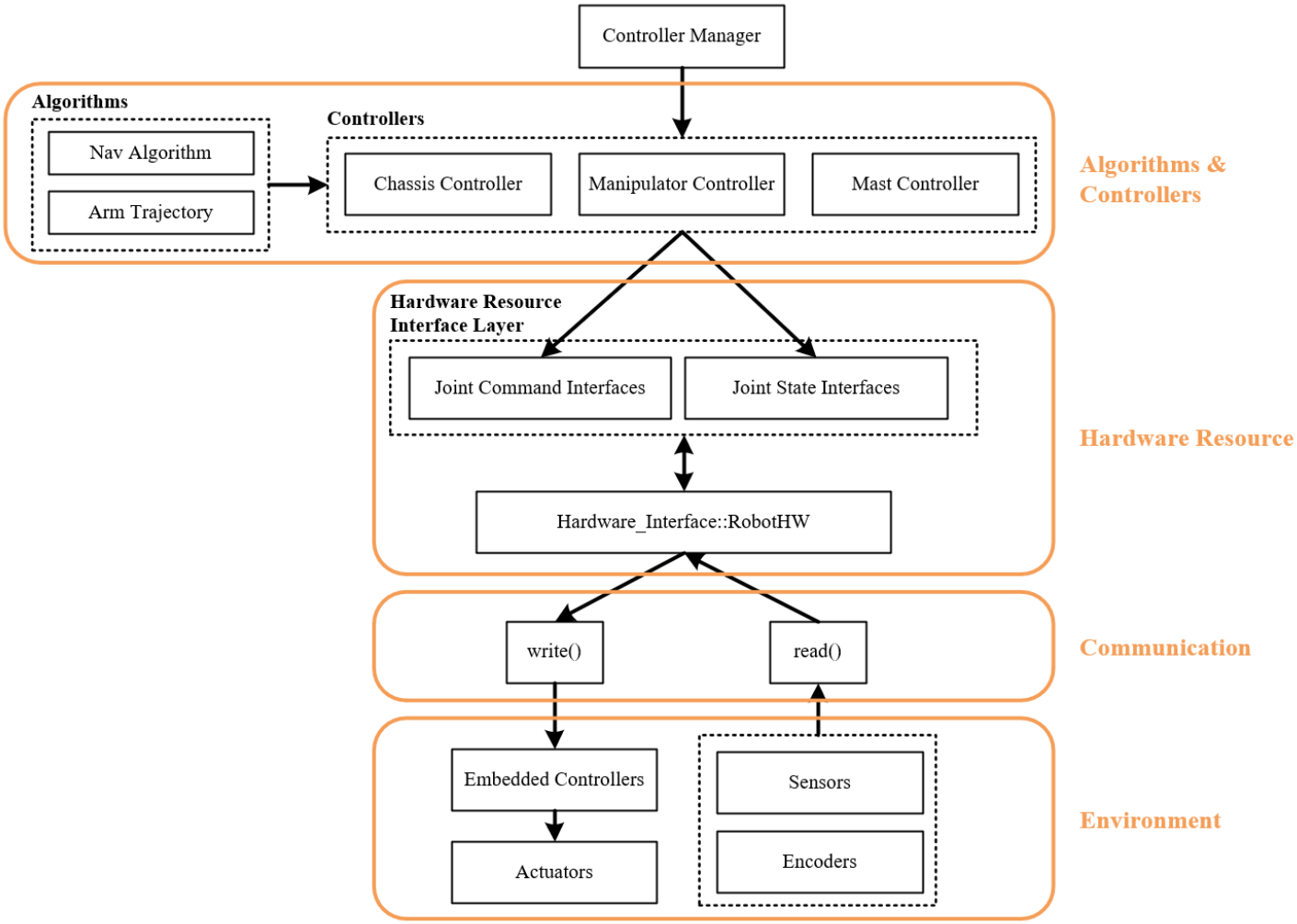

图3. SP_Control抽象框架图

回顾第一节展示的sp_control抽象架构图，Environment部分是实体电机、传感器等，相关数据的读写由Communication部分完成，上节课介绍的can_bus就是负责can类型数据的读写工作的。再上一层的Hardware Resource部分就是今天介绍的抽象硬件层，主要工作是作为桥梁沟通上层Controller和下层Environment（通过Communication），并储存电机过传感器的数据。

4、挂载电机

我们在抽象硬件层涉及的第一个概念就是挂载电机。抽象硬件层对上层控制器屏蔽电机的具体类型，也就是说抽象硬件层可以使用不同类型的电机。不同电机的通信协议，电机参数各不相同，而这些信息在电机数据读写的时候又特别重要。因此，我们需要先告诉抽象硬件层，对于某个关节我们想要使用什么电机，当上层控制数据传下来的时候，抽象硬件层就可以根据储存的该类型电机的通信协议和电机参数对应得出应该发送出去的CAN数据；反之，抽象硬件层也可以因此准确转化电机的状态数据，所有电机的数据都会统一为一样的格式，再传输给上层控制器，这样就实现了上下层解耦的效果。

我们同样使用YAML文件挂载电机（因为电机的类型此时应该被视为参数）。一个范例文件`hw_config.yaml`如下所示：

```
1 bus:
```

```yaml
  - can0
loop_frequency: 800
cycle_time_error_threshold: 1
thread_priority: 30

actuators:
  motor1:
    bus: can0
    id: 0x201
    type: rm_3508

  motor2:
    bus: can0
    id: 0x202
    type: rm_3508
```

大家回想一下第一节提到的电机参数读取，显然上面的YAML文件也呈现一个树状结构。该文件被解释为：对于每个电机我们赋予一个名称，如程序中的motor1、motor2等。每一个电机具有3个重要的信息参数：电机使用的can总线名称bus，电机id(或者说储存电机数据时所使用的id)，以及电机类型type。读取这些信息后，每当遇到上层控制器希望某电机（这里即使用电机名称来需要控制哪个电机）读写时，抽象硬件层就会根据具体的电机类型选择电机通讯协议和电机参数，并视电机通讯协议使用对应的电机id(有的电机的读写数据均使用其id作为can帧id，有些电机只有读数据或写数据中的一个使用其id作为can帧id)。

我们先来看 `hardware_interface.cpp` 中的这段程序。这段程序读取了电机参数并保存起来。

```cpp
/**
 * @brief 挂载电机
 * @param act_types
 */
bool SpRobotHW::parseActTypes(XmlRpc::XmlRpcValue &act_types)
{
    ROS_ASSERT(act_types.getType() == XmlRpc::XmlRpcValue::TypeStruct);

    for (auto it = act_types.begin(); it != act_types.end(); ++it)
    {
        if (!it->second.hasMember("bus"))
        {
            ROS_ERROR_STREAM("Actuator " << it->first << " has no associated bus
            continue;
        }
        else if (!it->second.hasMember("type"))
```

```
        {
            ROS_ERROR_STREAM("Actuator " << it->first << " has no associated typ
            continue;
        }
        else if (!it->second.hasMember("id"))
        {
            ROS_ERROR_STREAM("Actuator " << it->first << " has no associated ID.
            continue;
        }

        std::string bus = it->second["bus"], type = it->second["type"];
        int id = it->second["id"];

        //构建电机表
        //电机ID---->电机数据(ActData)

        if (!(id2act_data_.find(id) == id2act_data_.end()))
        {
            ROS_ERROR_STREAM("Repeat actuator on can0 and ID 0x" << std::hex <<
            return false;
        }
        else
        {
            id2act_data_.emplace(std::make_pair(id, ActData{.name = it->first,
                                                            .type = type,
                                                            .pos = 0.0,
                                                            .vel = 0.0,
                                                            .eff = 0.0,
                                                            .cmd = 0.0}));
        }
    }

    device_tree<ActData>(id2act_data_);
    return true;
}
```

为了方便我们查看到底有哪些电机被挂载到电机树上，我们会调用下面这个函数，把挂载的电机信息输出到终端上。这部分代码参见`hardware_interface.cpp`。

```
/**
 * @brief 已挂载电机信息显示
 * @param id2device_data
 */
template <typename T>
void device_tree(const std::unordered_map<int, T> &id2device_data)
```

```
7  {
8      std::cout << "|-- " << "can0" << std::endl;
9      for (auto &device_data : id2device_data)
10     {
11         std::cout << "|   "
12                   << "|-- "
13                   << "0x" << std::hex << device_data.first << " - " << std::de
14                   << device_data.second.type << " - " << device_data.second.na
15     }
16
17 }
```

💻 电机的挂载实际上是个抽象概念。抽象硬件层中的电机挂载过程不需要实际程序运行时外界有对应的电机相连。我们实际上可以做到在没有连接电机的情况下输出数据。

5、抽象硬件层运行流程

本节之后我们需要使用`ros_control`包，请在任意终端输入`sudo apt-get install ros-noetic-ros-control ros-noetic-ros-controllers`

最后一小节我们需要综合之前所提到的全部流程，详细阐述抽象硬件层的运行流程。下面这张图表示了抽象硬件层的运行流程。

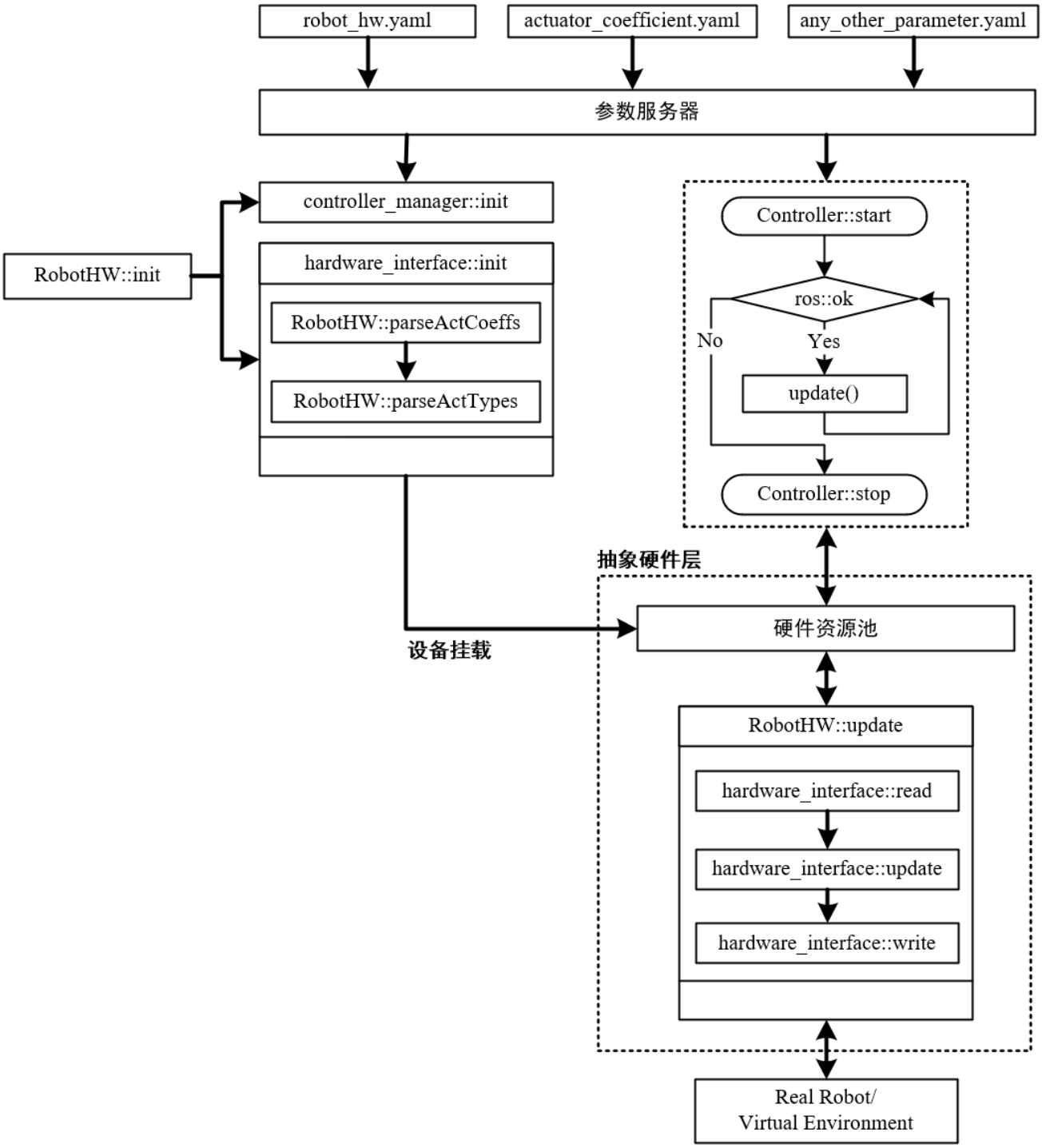

图4. 抽象硬件层运行流程

相比之前提到的`can_bus`的流程，抽象硬件层需要加载并调用电机参数，挂载电机以及储存电机数据。前两个流程分别由`bool SpRobotHW::parseActCoeffs(XmlRpc::XmlRpcValue &act_coeffs)`和`bool SpRobotHW::parseActTypes(XmlRpc::XmlRpcValue &act_types)`两个函数完成，而最后一个流程通过使用`std::unordered_map<int, ActData>`类型变量`id2act_data_`完成。

`bool SpRobotHW::parseActCoeffs(XmlRpc::XmlRpcValue &act_coeffs)`完成电机参数读取，并储存电机参数于`act_coeffs_`中。这部分代码参见`hardware_interface.cpp`。

```
1 /**
```

```
 * @brief 加载电机参数
 * @param act_coeffs
 */
bool SpRobotHW::parseActCoeffs(XmlRpc::XmlRpcValue &act_coeffs)
{
    ROS_ASSERT(act_coeffs.getType() == XmlRpc::XmlRpcValue::TypeStruct);
    for (auto act_coeff : act_coeffs)
    {
        if (act_coeff.first == "rm_3508")
        {
            if (act_coeff.second.hasMember("act2pos"))
                act_coeffs_.act2pos = xmlRpcGetDouble(act_coeff.second, "act2pos
            else
                ROS_WARN_STREAM("Actuator Type " << act_coeff.first << " has no

            if (act_coeff.second.hasMember("act2vel"))
                act_coeffs_.act2vel = xmlRpcGetDouble(act_coeff.second, "act2vel
            else
                ROS_WARN_STREAM("Actuator Type " << act_coeff.first << " has no

            if (act_coeff.second.hasMember("act2effort"))
                act_coeffs_.act2effort = xmlRpcGetDouble(act_coeff.second, "act2
            else
                ROS_WARN_STREAM("Actuator Type " << act_coeff.first << " has no

            if (act_coeff.second.hasMember("effort2act"))
                act_coeffs_.effort2act = xmlRpcGetDouble(act_coeff.second, "effo
            else
                ROS_WARN_STREAM("Actuator Type " << act_coeff.first << " has no
        }
    }
    return true;
}
```

`bool SpRobotHW::parseActTypes(XmlRpc::XmlRpcValue &act_types)` 挂载电机并初始化电机参数结构体，具体流程请参见上一小节。

最后是储存电机数据。之前我们发现 `parseActTypes` 使用了 `id2act_data_`，并将其初始化以备储存电机数据，那么这个 `id2act_data_` 为何能储存电机数据呢，还有之前提到的储存电机参数 `act_coeffs_` 又是如何传递电机参数的呢。注意，我们这里所做的事情都在 `hardware_interface` 中，而电机数据的读写和转换都在 `can_bus` 中。因此，我们需要建立 `hardware_interface` 与 `can_bus` 间的数据传递关系。

我们使用 `CanDataPtr` 来传输 `hardware_interface` 与 `canbus` 之间的数据，`CanDataPtr` 包含储存电机参数的结构 `ActCoeff` 和储存电机数据的结构 `ActData`。为方便起

见，我们只会储存m3508电机的参数，而电机数据部分由键值对格式表示，键为电机id，值为电机的关键数据。这部分代码参见 `data_type.hpp`

```
struct ActCoeff
{
    double act2pos, act2vel, act2effort;
    double effort2act;
};

struct ActData
{
    std::string name;
    std::string type;
    double pos, vel, eff;
    double cmd;
};

// only used in can_bus and store the pointers
struct CanDataPtr
{
    ActCoeff *act_coeffs_;
    std::unordered_map<int, ActData> *id2act_data_;
};
```

我们在 `can_bus` 初始化的时候需要将 `CanDataPtr` 的实例的地址传给 `can_bus` ，在 `can_bus` 中进行电机数据转换。我们在函数 `void SpRobotHW::initCanBus()` 中实现这一点。这部分代码参见 `hardware_interface.cpp` 。

```
/**
 * @brief 初始化canbus
 * @param
 */
void SpRobotHW::initCanBus()
{
    can_bus_ = std::make_unique<CanBus>("can0", CanDataPtr{.act_coeffs_ = &act_c
                                        .id2act_data_ = &id2act_data_}, thread_p
}
```

这样，在 `can_bus` 中读到的数据可以上传到 `hardware_interface` 中，也可以在 `hardware_interface` 中写数据到 `can_bus` 中发出去。此外，电机参数也是由 `hardware_interface` 根据挂载电机的类型选择之后传给 `can_bus` ，以供 `can_bus` 完成电机数据转换。

最后，我们在`void SpRobotHW::show()`函数中显示读取的电机数据，并写入控制电机的命令。这部分代码参见`hardware_interface.cpp`。

```
void SpRobotHW::show()
{
    for (auto &act_data : id2act_data_)
    {
        if (act_data.first == 0x201)
        {
            ROS_INFO_STREAM("----------------------------------");
            ROS_INFO_STREAM("id: " << std::hex << act_data.first);
            ROS_INFO_STREAM("pos: " << act_data.second.pos << "rad");
            ROS_INFO_STREAM("vel: " << act_data.second.vel << "rad/s");
            ROS_INFO_STREAM("eff: " << act_data.second.eff << "N.m");
            ROS_INFO_STREAM("cmd: " << act_data.second.cmd << "N.m");
            act_data.second.cmd = 0.015;
        }
    }
}
```

将来，我们可以把`hardware_interface`中储存的数据发送给上层控制器，以供控制器进行控制乃至更高级的算法运行。

这部分实验请在`catkin_ws`下编译后运行`rosrun lesson3 hardware`进行。