在机器人控制程序中使用配置文件 (1)

田雅夫

2016 年 12 月 25 日

1　为什么要在程序中使用配置文件

在调试机器人控制程序 (和所有计算机程序) 的过程中, 常常会遇到一系列运行时错误. 假使我们仅仅将程序看做处理数据的过程, 则这些错误可能来源于以下四方面:

1. 程序运行逻辑错误. 例如边界条件错误, 内存泄露, 野指针等.
2. 数学模型错误. 即在程序中建立的数学模型与我们真正想要的模型有差距. 例如机器人本体应当被近似为一个二阶系统, 却被当成是一阶系统进行处理.
3. 程序内参数错误. 即建立的数学模型在形式上式正确的, 但在参数上有一些误差. 例如机器人的参数 (轴距, 摩擦力等) 在程序与实际中存在着误差. 或者是 PID 控制效果与期望的有差距需要调整.
4. 输入数据失效. 即程序的输入是错误的, 并且这种错误没有得到处理. 例如在机器人运行过程中突然有元件失效, 但控制程序中并没有对这种失效给出解决方案. 这样就会造成机器人的一系列非预期行为.

在这四种错误中, 第一种与第四种是可以通过常规程序设计手段来避免的. 代码审查, 单元测试等手段可以有效的避免这些错误, 而让程序员更少的摄取咖啡并增加他们的睡眠时间也许是一种更有效的方法. 第二种与第三种错误来源于我们对自然环境认知的缺乏. 来源于我们没有 (或者未掌握) 足够强大的数学工具去有效的模拟整个自然环境. 所以只能依靠测试的手段去勉强达成一个可行的结果. 这就涉及对程序内参数的修改 (甚至是对数学模型的修改). 一些未经训练的程序员 (他们可能是很好地设计人员与控制系统设计者) 会采用以下的方式进行调试:

出现错误 -> 在程序中找到可能出问题的参数 -> 修改 -> 编译连接运行

在小型的工程中, 这种调试方式是简单的. 编译连接的过程也要不了几分钟. 但假使代码数量超过了几千行, 单单在程序中找到出问题的那个参数 (前提是知道出错的参数是哪一个) 就要花很大的力气. 通常会变成" 系统的上升时间是在哪里定义的来着"-> 翻看程序源代码 -> 还是全局搜索吧 -> 搜索"riseTime"-> 在搜索"system_tr" 这样. 有一些开发者习惯于把一个模块中用到的参数以常量或宏的形式写在该模块的开头. 这是一种相对较好的习惯, 但他们修改程序的时候可能会在程序的对应语句处忘记使用前面定义好的常量, 就出现了以下情况.

```
#define Damp_ratio 0.707
......
//calculate_something(Damp_ratio); 本来应该是这样的()
calculate_something(1.05); //可能在调试中为了方便就修改成了这样
```

即使程序的开发者有着相当不错的记忆力, 对于接手这个项目的其他人来说, 这样的代码也可能会造成灾难. 并且在多人协作开发的过程中, 有些参数莫名其妙的就被修改了. 假如后人要复用这样的一份代码, 他需要遍历一遍所有的文件, 找出系统内的所有参数与数学模型, 并用他自己的模型加以替换. 这个过程是极其痛苦的, 所以一部分程序员干脆将已有的, 经过测试的代码抛在一边, 重新实现一个有瑕疵但自己可以理解的版本, 下下个程序员再一次的做同样的事情. 项目进度延误就是这样产生的.

一个相对较好的方法是将逻辑 -数据相分离. 即将程序中所有可能改动的地方抽象出来放到一个文件中, 让程序去加载这个文件. 这样一是省却了编译连接的时间. 二是在配置文件中找一个参数比在代码中找这个参数要容易得多. 有很多主流程序都使用了这一策略, 例如 Linux, ROS 等.

自己实现一个配置文件的加载 -读取过程是相对容易的. 但处理所有的边界条件以保证配置文件的加载程序正确工作就相对困难一些. 这种复杂程度与开发者想在配置文件中传入数据类型的数量成平方正比关系. 另一种方案是使用现成的可扩展标记语言, 例如 XML, YAML, JSON 都有着相当的可读性与灵活性. 并且在主流编程语言中都有相对良好的库函数来处理这些文件的加载过程.

2 使用 JSON 文件保存程序中的设置

JSON 文件是一种很方便的可扩展标记语言. 熟悉 JavaScript 的开发者对 JSON 是再熟悉不过的了. 我采用 JSON 文件作为机器人标记文件的格式, 是因为相对于 XML 而言, 在未安装插件的编辑器上使用 JSON 格式不会造成额外的痛苦. 这里我给出一段 JSON 格式的参考范例, 来自一个喷雾机器人的配置文件的一部分.

```
{
    "_comment" : "␣JSON␣语言里面不能加注释␣,␣␣不过可以通过增加一个属性的方式在里面写入注
        释␣\nJSON␣语言中所有的字符串都会被解析成␣Unicode␣格式",
    "__docString__" : "喷雾机器人的基本配置文件␣\n␣␣␣␣config->␣基本配置模块␣",
    "config" :{
        "robotName" : "Robot_White",
        "firm_version" : "0.1",
        "author" : "Tian␣Yafu,␣*********@qq.com",
        "robotConnectionMethod" : "Ethernet",
        "connectionCharacter" : "Client",
        "IPAddress" : "DHCP",
        "port" : "None",
        "defaultTimeOut" : 5,
        "autoReConnection" : true,
        "functionList" : ["setDefaultSpeed", "setBackwardCoef", "goForward", "
            goBackward",
         "turnLeft", "turnRight","makeFog_start","makeFog_end","light_on","light_off","
             laser_enable","laser_disable"],
        "robotModuleDict" : ["CanBusController", "HokuyoLaserScanner", "
            UltraSoundSensor", "Adam6017", "Adam6060", "Camera"],
        "tempDict" : {"p1":10, "p2" : 15}
    },

    "robotDefaultKinematic" : {
```

```
        "defaultSpeed" : 1000,
        "backwardCoef" : 0.3

    },

    "robotModulesConfig" : {
        "CanBusController" : {
            "__docString__" : "␣以太网转␣Can␣总线服务器\n␣␣connectMode␣:␣连接时主机的连接
                方式␣\n␣␣IP␣␣地址当␣Server␣模式时时工控机的地址␣\n␣␣Port:␣当␣Server␣模式是
                本地绑定的端口␣,␣否则是远程端口",
            "connectMode" : "Client",
            "Can1IPAddress" : "192.168.1.253",
            "can1Port" : 7001,
            "Can2IPAddress" : "192.168.1.253",
            "can2Port" : 7001,
            "EnableConnectionTimeout" : false,
            "EthernetConnectionTimeout" : 3
        },

        "HokuyoLaserScanner" : {
            "__docString__" : "␣北阳的激光雷达␣",
            "IPAddress" : "192.168.1.10",
            "port" : 10940,
            "ScanRange" : 270,
            "DataLength" : 1081,
            "rangeThreshold_low" : 100,
            "rangeThreshold_high" : 10000,
            "recvDataBufferSize" : 9999
        },

        "UltraSoundSensor" : {
            "__docString__" : "超声传感器,␣串口连接的\n␣timeDelay:␣每次工作指令发送完毕到
                接收信号的延时␣",
            "addressPool" : ["01", "02", "03", "04", "05", "06"],
            "timeDelay" : 25
        },

        "Adam6017" : {
            "__docString__" : "␣多路模拟量采集卡␣Adam␣6017␣",
            "IPAddress" : "192.168.1.12",
            "port" : 502
        },

        "Adam6060" : {
            "__docString__" : "␣多路模拟量采集卡␣Adam6017␣",
            "IPAddress" : "192.168.1.13",
            "port" : 502
        },

        "Camera" : {
        }
    },

    "robotFunction" : {
    }
}
```

其实 JSON 的语法就用两句话就可以说明. 在 JSON 中一切数据要么是对象要么是对象的属性. 并且对象之间可以嵌套, 一个属性的值可以是对象. 属性的取值具体支持一下形式

1. 数字 (整数, 浮点数)
2. 列表 (用方括号括起来)
3. 对象 (用花括号括起来)
4. 字符串 (用双引号括起来), 所有属性的" 键"(Key) 必须是字符串
5. 布尔类型 (true/false), 空值 (Null)

另外需要注意的一点是, Json 文件在标准中是不包含注释的, 也就是说, 任何包括注释的 Json 文件都不能被正常的解析. 作为替代, 可以在每个类中加入 { "_comment":"What you want to say" } 这样的方法.(方法名称可以任取, 但最好让人一眼就能明白这是注释比较好. 如果每次取同样的名称, 可以一次性的批量干掉所有注释, 在发布代码时很方便.)

3 使用 Python 处理 JSON 格式的文件

Python 内置了 json 的解析库, 使用方式如下 (以 Python2.7 版本为例):

```
import json

_fileHandler = None

try:
    _fileHandler = open("config.json", "r")
except IOError, reason:
    print "Error␣on␣opening␣JSON␣file!"
    exit()

try:
    config = json.loads(_fileHandler.read())
except ValueError, reason:
    print "Error␣on␣loading␣␣JSON␣File!\n", reason, "\nExiting␣Program!"
    exit()

print json.dumps(config, sort_keys=True, indent=4, separators=(',', ':␣'))
```

将读取的 Json 文件以字符串方式传入 json.loads() 函数进行解析. 如果 Json 文件存在顶层对象, 则会产生一个字典类型的对象.(否则产生一个字符串). 加入传入的 Json 文件不合法, 可能会产生一系列的错误. 详见最后一节所示.

json.loads() 将配置文件变为一个由字典类型组成的树, 可以按照 Python 操作字典的语法, 操作配置文件. 或者遍历整个配置文件. 特别要注意的是, 除非你很清楚你在做什么, 否则请不要对这个配置文件中的变量进行赋值操作.(只使用读取操作). 并且不对其中的列表类型 (由 Json) 的 array 类型解析而来. 尽管这时候所做的一切操作都不会影响实际的, 文本形式的配置文件. 但在接下来使用配置文件的参数时可能会报错.

json.dumps() (dict/str[1] -> str) 函数将读取的 Json 文件转化为字符串的形式进行输出. 该函数支持以下可选参数 (等号右边是默认值):

[1] Json 中不存在顶层对象时

- skipkeys=False Json 解析器在遇到一个 Json 对象时, 若该对象的方法 (解析后字典的" 键") 不是字符串形式的, 会报错 (TypeError), 通过启用这个选项 (True) 可让解析器忽略这些错误 (直接跳过去). 是个危险的选项呢.
- ensure_ascii=True 默认情况下, 解析器会把字符串中所有的非 AscII 码字符转换为"\ Uxxxx" 的形式. 通过这样的形式来使得每个字符串对象中只包含 AscII 码字符. 但这样做的话, 在字符串中包含汉字的情况下输出不是很直观
- allow_nan=True : 在数字超过 float 类型可表示的最大范围的情况下是否报错 (ValueError).
- indent=None, separators=None 在以字符串输出 JSON 文件时, 配置输出格式. 具体用法参照代码样例即可.
- encoding="utf-8" : 输出字符串的编码
- sort_keys=False 在输出字符串的时候是否对键的顺序进行排序 (Python 的字典是无序的, 也就意味着不同的 dump 可能得到不同的结果)

需要注意的是 config 的类型是 Dict, 即 JSON 的顶层对象会被转化为字典类型. 其他的类型转换遵循以下规则[2]:

- object->dict
- array list
- string->unicode
- number (int)->int, long
- number (real)->float
- true->True
- false->False
- null->None

在一些特别的情况下, 需要对配置文件进行更改, 这种更改是要固化在文件里, 以便每次运行程序时使用的. 在这种情况下, 可以简单地重新赋值解析出的字典对象中的值, 并用 json.dumps() 的输出结果覆盖回配置文件即可. 第一次使用这种方法可能会造成配置文件中对象的顺序发生变化. 这是因为人手写的配置文件通常不会按照对象 (和属相) 的键名称, 按照字典降序进行排序. 但每次 json.dumps() 输出的结果在语义上都是相等的, 不会影响解析出的结果.

[2]来自 Py2.7 官方文档

4 使用 C++ 处理 JSON 格式的文件

4.1 用 C++ 读取 JSON 文件

4.2 用 C++ 修改 JSON 文件

5 JSON 与 XML 的相互转换

JSON 与 XML 都是被广泛应用的结构标记语言. 在很多情况下, 我们所要求的格式和实际拿到的格式有所不同. 这样就要进行转换.

在 Python 中, 可以将这两种格式的文件统一转换成 dict(嵌套的树结构), 用 dict 作为一种过渡的数据结构来进行这两种格式的转换. 前面已经对 dict 与 JSON 相互转换的方法说清楚了. 接下来介绍使用 xmltodict 这个 Python 的 xml 解析库进行 dict 与 XML 互转的过程.

首先安装 xmltodict 库, 假设读者已经安装了 pip 工具, 则通过以下命令进行安装即可.

```
pip install xmltodict
```

在前文的代码中, 已经将 JSON 形式的配置文件转换为一个 dict 形式的树结构, 接下来就将这个树结构转换为 xml 格式的文本文件. 例如:

```
## 注意这个代码会报错
convertedXXml = xmltodict.unparse(config)
print convertedXml
```

运行这段代码, 报了一个错ValueError: Document must have exactly one root , 这是因为在 xmlToDict 中要求一个树只能有一个顶层节点. 而这棵树里面有 6 个顶层节点 _comment,robotModulesConfig,robotDefaultKinematic,__docString__,config,robotFunction. 所以报了这样一个错误.

当高斯在森林里面迷路了, 它随手添加了几条边, 把森林变成了一棵树. 在这里使用同样的方法. 添加一个顶层节点"root", 让它的子节点是原来 dict 里面的顶层节点, 代码如下:

```
## 正确的代码
## 这个模块要求索要解析的字典中只有一个顶层节点, 所以添加一个新的顶层节点"root", 把原有的
    顶层节点都作为它的子节点
newTopRoot = {"root":{}}
for i in config:
    print i
    newTopRoot["root"][i] = config[i]

convertedXml = xmltodict.unparse(config)
print convertedXml
```

输出结果太长了, 这里就省略了. 读者可以自行输出, 并验证一下正确性.

xml 转换为 dict 的过程与上面的过程类似, 以文本形式读取 xml 以后, 使用 xmltodict.parse() 进行解析就行.

6 在程序中使用配置文件

在以后的更新中, 将会介绍在何种场合下使用配置文件对程序进行配置. 本文将集中于以下几个要点:

- 用配置文件来避免代码中的 magic number
- 利用配置文件进行机器人的远程配置
- 利用 JSON 进行机器人的远程控制
- B/S 架构下机器人的配置 (这节单拿出来还能写不少)

7 一小部分常见错误的解决方案

在解析文件时产生 ValueError, 首先要检查传入 (loads) 的字符串是否合法:

- json 模块要求所有属性的名字 (即解析出的字典的键) 必须为字符串类型. 即 { property:100 } 这样的代码是不合法的. 需要转换为 { "property":100 } 这样的形式.
- 该模块是不能识别 Json 文件内的注释的 (有一些不严格的解析器是允许传入注释的)
- 包含类似 nan, inf 的数字是会报错的, 可以通过"allow_nan=True" 来避免
- 请确保 JSON 文件格式正确, 有时一些隐含的格式问题并不容易被发现. 但不管怎样, 程序都会报错的.