概述

本系统的设备驱动是基于模板的，通过外部的配置文件可实现在不改变源码的前提下对设备驱动进行配置，配置成功后即可通过配置好的设备驱动对设备进行相应的操作。

系统模块简介

本系统由以下四个模块组成：

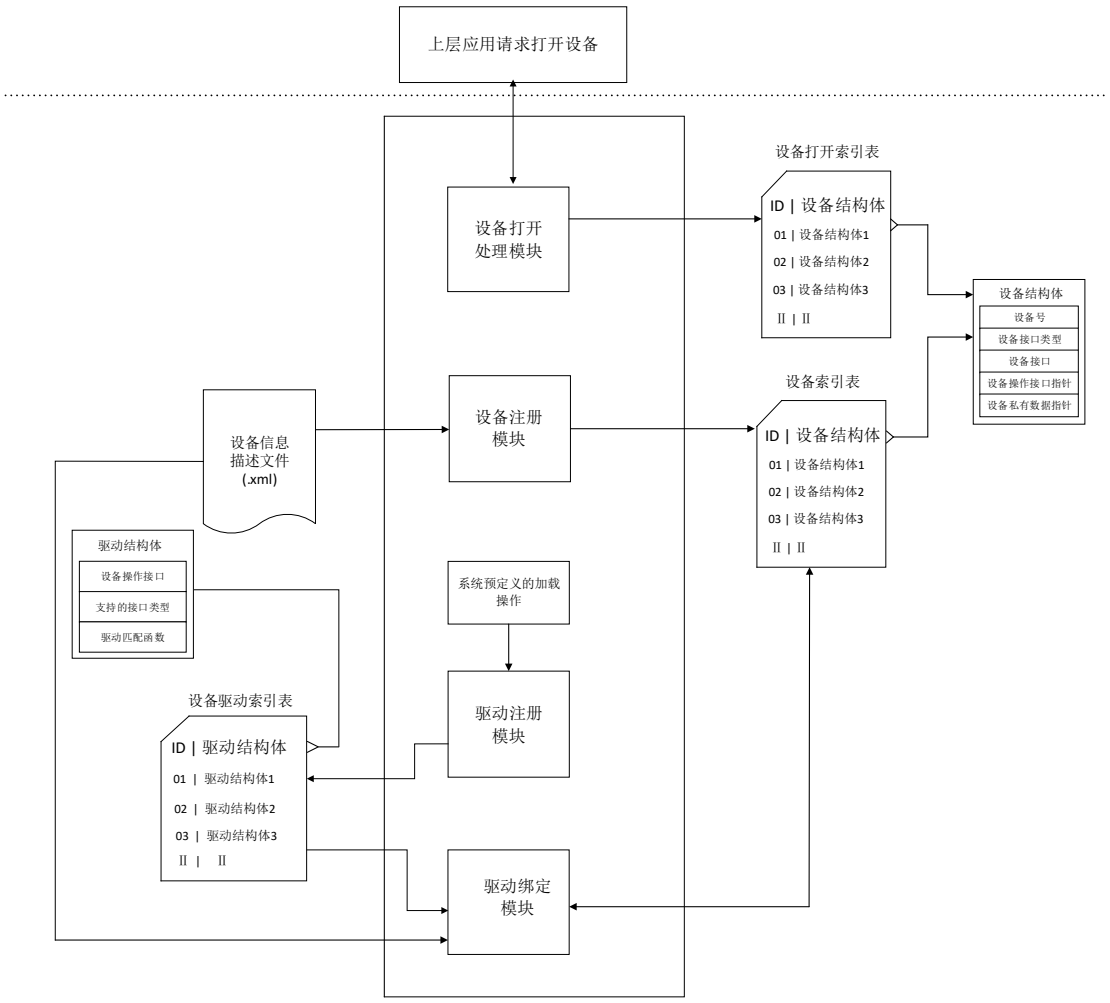
1 设备注册模块；

2 驱动注册模块；

3 驱动绑定模块；

4 设备打开模块。

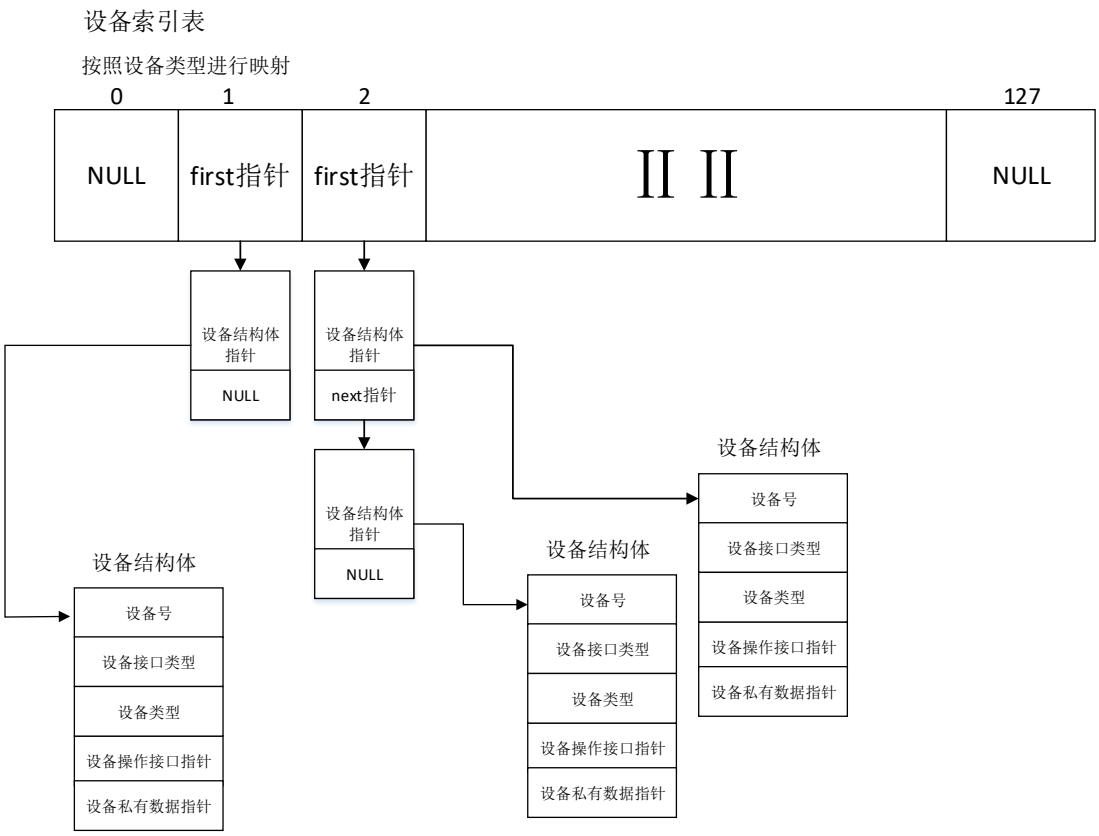
其中前面三个模块主要完成设备与设备驱动相关初始化工作，下面是这四个模块之间的关系图：



设备注册模块

接入系统的设备信息通过配置文件的形式给出，系统初始化时该模块会从该配置文件中获取所有的设备以及设备相关信息，然后在内部分配一个表示设备的结构体来存放这些信息，并将表示设备的结构体添加到系统维护的一个设备索引表中。

设备注册完成后设备索引表示意图如下所示：



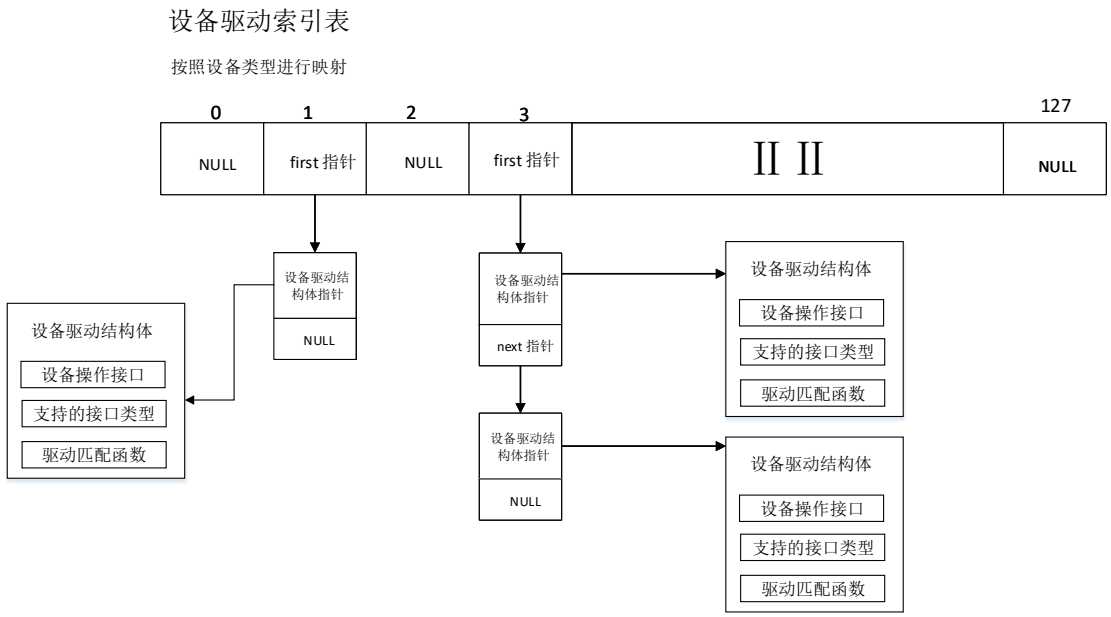
不同类型的设备在设备索引表中的索引号是不一样的，同种类型的设备根据其注册的顺序链接在一个链表中。

设备结构体中有五个成员：设备号、设备接口类型、设备类型、设备操作接口指针以及设备私有数据指针。其中设备号用于唯一标识设备；当设备与驱动成功绑定后，设备操作接口指针就指向设备驱动实现的一组设备操作函数；同样地当设备与驱动成功绑定后，从配置文件中收集的配置信息会存放到一个称为设备模板数据信息表数据结构中，设备私有数据指针就指向这个数据结构，该数据结构会在后面的驱动匹配机制中介绍。

驱动注册模块

系统中的每个设备驱动都对应有一个结构体用于描述设备驱动。系统为每个设备驱动定义了一个驱动加载函数，该函数的功能就是将描述设备驱动的结构体添加到系统维护的一个设备驱动索引表中。驱动注册模块依次执行这些预定义的驱动加载函数，间接地将系统中所有的设备驱动添加到设备驱动索引表中。

设备驱动注册完成后设备驱动索引表如下图所示：



设备驱动结构体按照该驱动支持的设备类型映射到设备驱动索引表中，设备驱动支持该类型设备的接口类型包含在设备驱动结构体中。支持同种类型设备但不同接口类型的驱动根据其注册的顺序链接在一个链表中。

设备驱动结构体中有三个成员：设备操作接口、支持的接口类型

以及驱动匹配函数。其中设备操作接口是设备驱动实现的一组设备操作函数；支持的接口类型为驱动支持的设备接口，该信息会在驱动绑定模块中查找驱动过程中使用；驱动匹配函数判断配置文件中的信息与驱动所需的信息是否相符，每个驱动都会提供一个这样的驱动匹配函数，该函数会用于驱动绑定的时候被调用。

驱动绑定模块

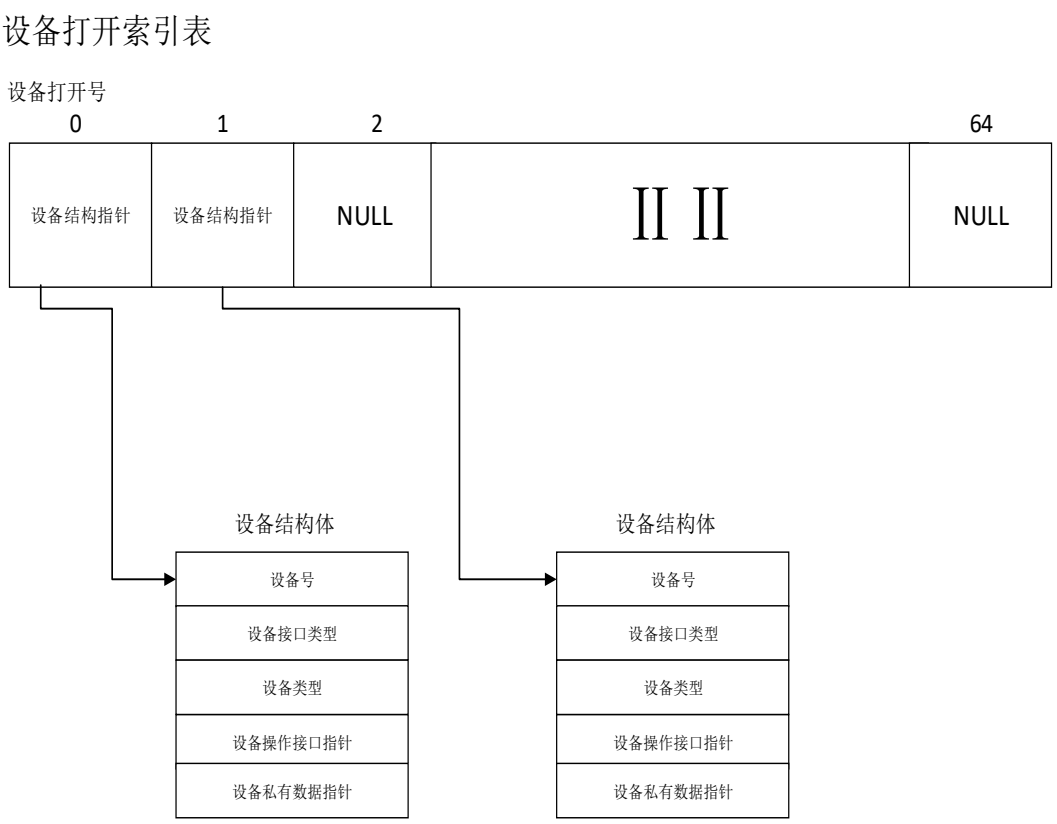
前面两个模块已经将接入系统的设备信息以及系统中的设备驱动准备就绪了，下一步就是将设备与能够驱动它的设备驱动之间建立关系。驱动绑定模块依次遍历设备索引表中的设备，根据设备的类型和接口类型去查询设备驱动索引表中的设备驱动，如果找到支持该接口类型的设备驱动则执行描述该设备驱动的结构体中的匹配函数判断配置文件给出的信息是否与驱动所需的一致，如果是那么就将设备驱动提供的设备操作函数与设备结构体中的设备操作函数指针建立连接关系以及在匹配过程中收集的设备信息与设备结构体中设备私有数据指针建立连接关系。

设备打开模块

当驱动绑定模块完成对接入系统的设备与系统提供的设备驱动建立连接关系时，意味着应用程序就可以打开该设备，并对设备进行操作了。设备打开模块在应用程序第一次请求打开具体设备时，根据设备逻辑名查找设备结构体，然后将这个结构体指针添加到一个设备打

开索引表中并返回对应的索引号，该索引号称为设备打开号。设备打开成功后，应用程序就可以通过设备驱动提供的设备操作对打开的设备进行操作了。这里需要说明的是设备打开操作不仅仅是过设备结构体指针添加到设备打开索引表，内部还调用了驱动的设备打开操作，设备打开操作一般实现为设备的初始化操作。

下图是应用程序打开两个设备之后的设备打开索引表结构：



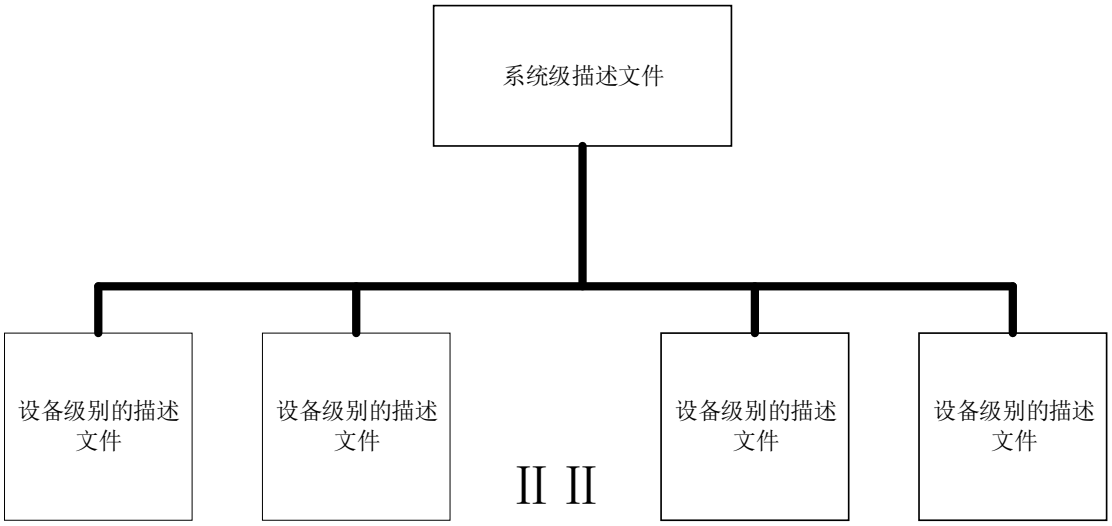
由于系统中的设备驱动是基于模板的或者说是数据驱动的，即设备驱动是抽象的未具体化的，如果没有相应的数据则设备驱动不能正常工作。系统中这些模块的主要目的是为设备驱动中的每个模板找到符合条件的数据，并在应用程序需要通过设备驱动中的某个操作模板操作设备时，通过符合条件的数据来使得操作模板具体化，进而执行

具体的操作。

配置文件简介

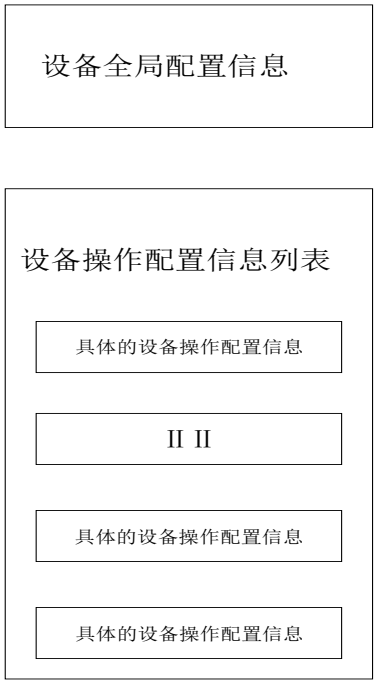
配置文件构成

配置文件采用 xml 格式，根据配置文件描述的信息种类可以分为两种：一种是系统级描述文件，用于描述系统的整体结构，该文件描述包括接入系统的设备以及系统硬件接口信息；另外一种设备级的描述文件，用于配置设备驱动，该文件主要描述驱动该设备所需的模板数据信息。



配置文件格式

设备级的描述文件分为两个部分：一部分用于描述各个设备驱动函数所需的模板数据信息；另一部分描述设备全局的模板数据信息，该部分是可选的。描述文件的大致结构如下所示：



配置文件标签介绍

配置文件中的标签用于组织配置信息，标签是依照前面配置文件格式来进行设置的，下表是用于描述设备配置信息的一些基本标签的介绍。

标签名	标签属性	说明
<global>	name	用于设备的全局配置。
<op_list>	length	length 属性指定要对多少个设备操作（op）进行配置。
<op>	name template_id	设备操作名 name; 选择的模板号 template_id。
<paras>	list_length	list_length 属性用来指定某个设备操作有多少个模板数据组成。系统提供一些基本的模板数据，配置信息时可以根据需要组合这些模板数据
<para_list>	length name	para_list 标签下的参数配置就指定了一个模板数据；length 属性指定要对多少个设备操作（para）进行配置; name 属性用于标识这个模板数据。
<para>	-	参数根据具体的要求其属性也是不一样的。需要根据参数的特点进行属性的设置。

除了上表中描述的基本的标签之外，设备级描述文件中还有一些用于描述代码块的标签，考虑到代码块信息描述灵活性以及收集的方便性，标签命名按照代码块的功能进行命名。

下表是目前使用到的用于描述代码块的标签介绍。

驱动匹配机制

驱动绑定模块在将某设备与系统中设备驱动进行绑定之前需要判断该设备在配置文件中的配置信息是否与设备驱动需要的信息相符，该过程是通过驱动匹配机制完成的。

本系统中的设备驱动是基于模板实现的，因此要使得驱动正常工作的关键是判断配置文件所提供的设备配置信息是否能够符合设备驱动中操作模板以及可选的全局配置的要求。为此本系统提供了一个驱动匹配机制，该机制的工作原理如下：为设备驱动的每一个操作模板定义一个操作模板匹配函数以及一个全局配置匹配函数，如果这些匹配函数都匹配成功的话，那么就可以将该设备与驱动进行绑定。

为实现上述的匹配机制，系统定义了四类匹配函数：设备全局配置匹配函数、设备操作模板匹配函数、设备驱动匹配函数以及执行匹配流程的函数。

设备全局配置匹配函数：此类匹配函数用于匹配配置文件中设备全局配置的信息是否符合要求；

设备操作模板匹配函数：设备驱动的每一个设备操作模板都有一个相应的设备操作模板匹配函数，匹配函数的功能是判断配置文件中对某个设备操作模板的配置信息是否符合要求；

设备驱动匹配函数：该函数是驱动注册时驱动结构的一部分，主要为了在进行驱动绑定时作为一个统一抽象的接口来判断配置信息是否与驱动所需的相符合，而不涉及到具体的匹配过程；

执行匹配流程的函数：根据匹配机制，该函数会调用相应的设备全局配置匹配函数和设备操作模板匹配函数，如果这些匹配函数都匹配成功的话，那么该函数就返回匹配。匹配流程对于所有的驱动都是一样的，但是由于每个特定的设备驱动其操作模板的个数以及设备操作函数的个数不同，需要将这些信息作为参数传递给执行匹配流程的函数。

下面以陀螺仪(gyroscope)驱动的匹配为例进一步说明。

陀螺仪的设备驱动匹配函数 gyroscope_match 实现很简单，只是将陀螺仪驱动匹配所需的一些信息传递给执行匹配流程的函数 do_match 函数，下图是 gyroscope_match 函数代码：

```
int gyroscope_match(void)
{
    return  do_match(&gyro_match_info);
}
```

驱动匹配所需的一些信息是组织成一个类型为 match_info 的结构体，它的成员有设备模板数据信息表的大小、匹配函数表以及匹配函数数量。

```
struct match_info gyro_match_info = {
   (GYROSCOPE_OP_NUM+1),
   match_funcs_table,
   MATCH_FUNCS_NUM
};
```

其中设备模板数据信息表是用来保存后面从配置文件中获取的信息，它的大小为设备驱动的操作函数的数量加上一个设备全局配置；匹配函数表的结构如下：

```
struct template_match match_funcs_table[GYROSCOPE_TEMPLATE_NUM] = {
    {"global", global_match},
    {"gyroscope_open_template0", open_template0_match},
    {"gyroscope_getx_template0", getx_template0_match},
    {"gyroscope_gety_template0", gety_template0_match},
    {"gyroscope_getz_template0", getz_template0_match},
    {"gyroscope_getxyz_template0", getxyz_template0_match}
};
```

执行匹配流程的函数接收一个匹配信息结构体指针作为参数，然后通过上述的匹配函数表调用相应的设备全局配置匹配函数和设备操作模板匹配函数。

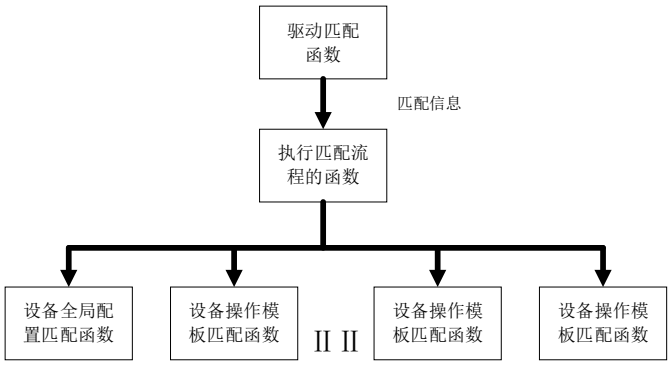
```
int do_match(struct match_info* mip)
{
    ...
}
```

例如对于陀螺仪配置文件中下面的配置信息，执行匹配流程的函数会通过 global 标签中的 name 属性在匹配函数表中找到用于匹配设备全局配置信息的设备全局配置匹配函数 global_match；类似地对于设备操作 gyroscope_open，因为这里是想要对模板号为 0 的操作进行配置，因此通过 name 和 template_id 属性在匹配函数表中找到设备操作模板 gyroscope_open_template0 对应的设备操作模板匹配函数 open_template0_match。

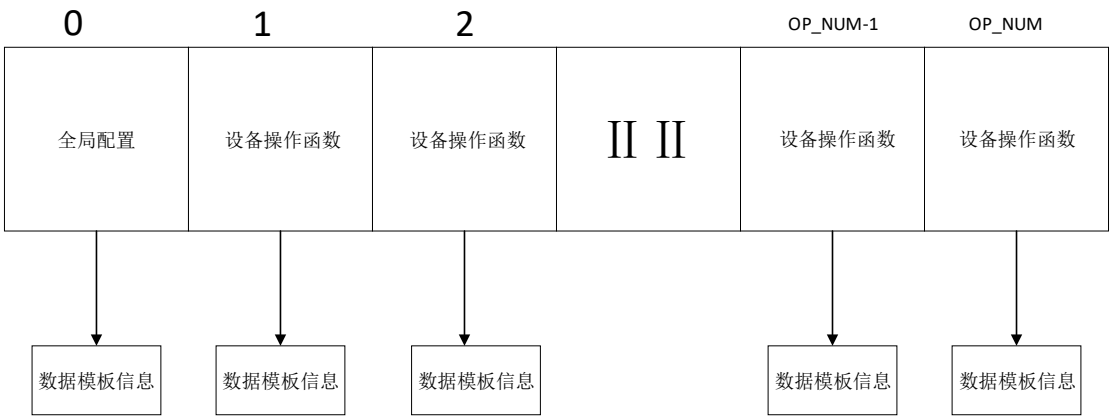
```
<global name="global">
  <para_list length="1" name="global">
     <para name="slave_address" type="char">0xD2</para>
  </para_list>
</global>
```

```
<op name="gyroscope_open" template_id="0">
  <para_list length="5" name="open_template0">
    <para address="0x20">0x00</para>
    <para address="0x21">0x00</para>
    <para address="0x22">0x08</para>
    <para address="0x23">0x03</para>
    <para address="0x24">0x00</para>
  </para_list>
</op>
```

下图是前面四类匹配函数之间的关系：



如果一个操作模板匹配函数返回成功匹配的话，执行匹配流程的函数会将其从配置信息中获取的数据模板信息添加到一个设备模板数据信息表中相应表项，该表的结构如下所示：



其中 OP_NUM 是指设备驱动中设备操作函数的数量，驱动中每一个

设备操作函数在设备模板数据信息表都对应于一个下标，全局配置在0号位置。前面提到过驱动匹配函数实际上通过传递一个包含所有驱动函数模板匹配函数等信息的结构体给通用的匹配函数，OP_NUM 也是包含在这个结构体中的，通用的匹配函数执行具体的匹配流程前会根据 OP_NUM 的大小初始化一个空的设备模板数据信息表，此后每当有一个操作模板匹配函数返回匹配成功，就将其对应的模板数据信息添加到相应的表项中。

前面用于匹配的设备操作模板匹配函数的数量至少等于设备操作函数的数量，因为每一个设备操作可以有多个模板，每一模板都需要有一个对应的模板匹配函数。在设备配置信息中设备操作都要写上模板号信息，当进行匹配的时候系统会根据这个模板号调用相应的操作模板匹配函数去匹配配置信息是否符合这个模板所需的数据信息。

如果设备的所有操作模板匹配函数都返回匹配的话，那么通用的匹配函数才会返回匹配，也就是说驱动绑定的时候调用设备驱动结构体中的匹配函数返回匹配。此时驱动绑定模块就可以执行具体的绑定操作了：将设备结构体中的设备操作函数指针赋值为设备驱动提供的设备操作函数；设备结构体中设备私有数据指针赋值为在匹配过程中建立的设备模板数据信息表。

驱动相关模块

本系统实现的驱动是基于模板的，为了能够对其实现配置以及上层应用程序能够通过一套 API 配置好的设备进行操作，对于一个具体

的设备驱动与其相关的模块主要有：驱动匹配模块、上层 API 模块、驱动函数分派模块、设备操作模板函数模块以及底层总线接口驱动模块。

其中驱动匹配模块实现驱动匹配机制，其它几个模块之间协作完成对设备进行操作，具体地上层 API 模块为应用层提供了一组操作设备的 API；当应用调用这些 API 的时候，将设备的私有数据（设备模板数据信息表）和参数传递给驱动函数分派的模块，该模块从设备模板数据信息表中取出相应的模板数据并根据操作模板号调用驱动操作模板函数模块中相应的操作模板函数；操作模板函数则根据模板数据以及上层应用传递的参数进行处理，然后调用底层总线接口驱动模块中的相应函数进行数据的收发功能，进而完成对设备的操作。

下面还是以陀螺仪为例对上层 API 模块、驱动函数分派模块、驱动操作模板函数模块以及底层总线接口驱动模块进一步说明。

为了方便说明规定上层 API 模块中的函数称为 api 函数，驱动函数分派模块中的函数称为分派函数，驱动操作模板函数模块中的函数称为操作模板函数。

陀螺仪的上层 API 模块主要实现了以下几个对设备进行操作的 api 函数，gyroscope_open 函数以设备逻辑号例如“gyroscope1”为参数调用设备打开模块中的相应函数将设备结构体指针添加到设备打开索引表中，并将表的索引号作为设备打开号返回。其它几个 api 函数都有一个设备打开号的参数。在这些函数内部通过设备结构体中的设备操作接口指针调用驱动函数分派模块中相应的分派函数，分派

函数以设备私有数据指针 private_data 和传递给 api 函数的数据参数 data 作为参数。

```
extern int gyroscope_open(char* lid);
extern int gyroscope_getx(int index, void* data);
extern int gyroscope_gety(int index, void* data);
extern int gyroscope_getz(int index, void* data);
extern int gyroscope_getxyz(int index, void* data);
```

驱动函数分派模块有一组分派函数组成，这些分派函数与上面的 api 函数是一一对应的。

```
static int general_gyroscope_open(void* private_data, void* data);
static int general_gyroscope_getx(void* private_data, void* data);
static int general_gyroscope_gety(void* private_data, void* data);
static int general_gyroscope_getz(void* private_data, void* data);
static int general_gyroscope_getxyz(void* private_data, void* data);
```

这些分派函数实际上和陀螺仪驱动结构体 gyroscope_driver 中的设备操作接口 gdo 是同一组函数。这些函数并不执行具体的设备操作，而是从 private_data 即设备模板数据信息表中获取设备操作对应的模板数据，并根据模板号调用设备操作模板函数模块中具体的操作模板函数，因为它的功能就是根据模板号对具体的操作模板函数进行分派，所以称这些函数为分派函数。

```
static struct gyroscope_device_operation gdo = {
    general_gyroscope_open,
    general_gyroscope_getx,
    general_gyroscope_gety,
    general_gyroscope_getz,
    general_gyroscope_getxyz
};
```

```
static interface_t driver_supported_interfaces;
static struct driver gyroscope_driver = {
     &driver_supported_interfaces,
     (void*)&gdo,
     gyroscope_match
};
```

前面说过一个设备操作都至少有一个设备操作模板函数，设备操作模板函数模块由一组设备操作模板函数组成，目前系统只为陀螺仪的每个设备操作提供一个设备操作模板函数，后面有需要的话可以很方便添加其它的设备操作模板函数。

```
extern int gyroscope_open_template0(void* template_data, void* data);
extern int gyroscope_getx_template0(void* template_data, void* data);
extern int gyroscope_gety_template0(void* template_data, void* data);
extern int gyroscope_getz_template0(void* template_data, void* data);
extern int gyroscope_getxzy_template0(void* template_data, void* data);
```

这些设备操作模板函数对传进来的模板数据以及参数进行相关处理后调用底层总线接口驱动模块中的函数进行数据收发，进而实现对设备的操作。

底层总线接口驱动模块是对底层总线驱动的一个总称，这些总线驱动包括 I2C、AD/DA 以及 RS422 等总线接口的驱动。

数据模板类型

系统中的设备驱动由一组操作模板组成，每个操作模板需要特定的数据模板对其进行配置。数据模板信息根据其结构以及内容是否可在配置阶段确定分为两大类：配置阶段可确定的静态信息和运行阶段

可确定的动态信息。

配置阶段可确定的静态信息

该类信息的特点是其结构以及内容都可以在配置阶段确定。静态的数据模板信息由参数列表进行组织，根据参数列表中的参数数量的是否可变以及参数之间的类型是否相同，可以将静态的数据模板信息进一步分为以下 3 种：

普通结构体 plain_struct：

这种类型的数据模板信息其特点在于参数数量固定，参数之间的类型没有限制。例如下面的例子：

```
<op name="gyroscope_getx" template_id="0">
   <para_list length="2" name="getx_template0">
      <para name="reg_address" type="char">0x28</para>
      <para name="size" type="int">2</para>
   </para_list>
</op>
```

上面的数据模板信息用于配置获取陀螺仪 x 轴数据的设备操作 gyroscope_getx，要使得该设备操作能够正常工作需要指定两种信息：一种是 x 轴数据存放的寄存器地址信息，另一种是 x 轴数据大小信息。这两种信息通过<para>标签进行配置。为了防止在缺少标识情况下因参数配置顺序错误而导致参数配置混乱增加了 name 属性对相应的参数进行标识。

普通数组 plain_array：

这种类型的数据模板其特点在于参数之间的类型相同，参数数量无限制。例如下面的例子：

```
<op name="magnetometer_readx" template_id="0">
  <para_list length="1" name="readx_template0">
    <para type="int">1</para>
  </para_list>
</op>
```

上面的数据模板信息用于配置获取 AD 接口的磁强计 x 轴数据的设备操作 magnetometer_readx，要使得该设备操作能够正常工作只需要指定一种信息即该磁强计 x 轴所对应 AD 的通道号。AD/DA 接口的模拟式太阳敏感器、磁力矩器也可以使用 plain_array 来收集相关通道号信息。

寄存器数组 reg_array ：

这种类型的数据模板其特点在于参数之间的类型相同，参数数量无限制，但是需要对参数进行标识。例如下面的例子：

```
<op name="gyroscope_open" template_id="0">
  <para_list length=5 name="open_template0" >
    <para address="0x20" type="char">0x4f</para>
    <para address="0x21" type="char">0x00</para>
    <para address="0x22" type="char">0x08</para>
    <para address="0x23" type="char">0x00</para>
    <para address="0x24" type="char">0x00</para>
  </para_list>
</op>
```

上面的数据模板信息用于配置获取 I2C 接口的陀螺仪打开设备操作 gyroscope_open，这种设备在进行初始化的时候需要设备的相应

的寄存器中写入数据以配置设备工作方式等。对于不同的应用设备的工作方式可能不一样，因此需要配置的参数数量可能也是变化的。对于需要向设备的某个寄存器地址写入数据的配置都可以使用 reg_array 来收集相关信息。

运行阶段可确定的动态信息

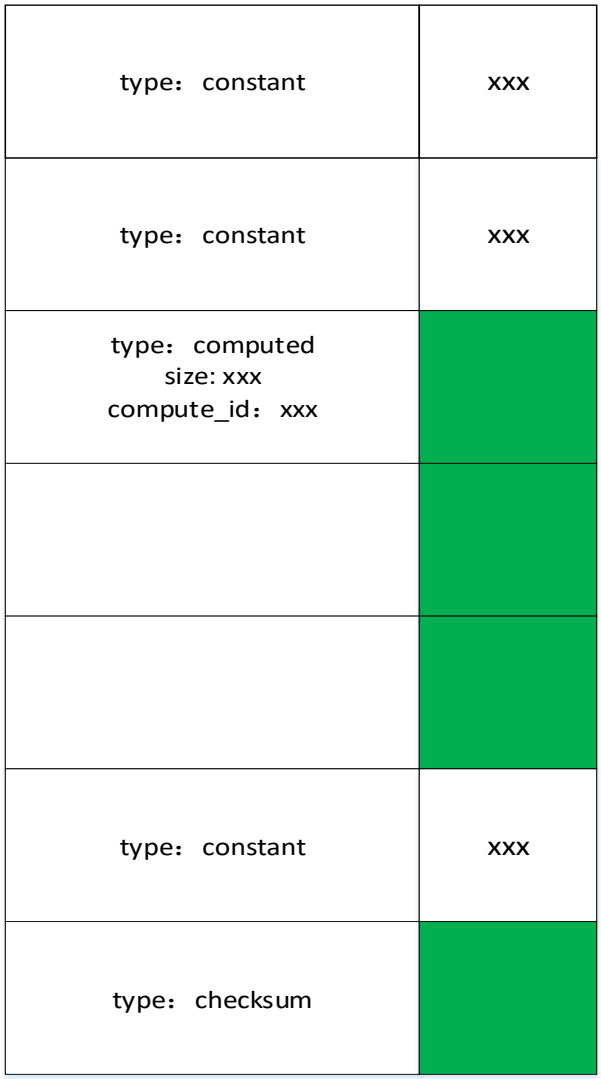
信息的动态性主要体现在其可变性以及可计算性，例如需要对程序中变化的变量做一些计算以及计算数据的校验和。这里涉及到了怎样去描述程序中某个量的计算流程，此类信息很难通过类似前面 3 中静态的配置信息一样描述，为此需要一种新的描述格式。考虑到描述某个量的计算流程的较大的可变性，在配置文件中引入了代码块以及解析并执行代码块解析器，对于解析器而言变化的代码块是一种可是确定的数据，只要代码块按照一定的语法规则去书写即可。代码块及解析器是一种通用的解决配置文件中可变性大的配置信息部分，可以单独使用也可以配合其它静态的数据模板完成运行阶段的动态信息配置，目前系统通过后面这种方法实现了两种数据模板格式，这两种数据模板格式主要用于配置 RS422 接口的姿态传感器：

命令序列 command_sequence

RS422 接口的姿态传感器发送的数据有以下四种形式：常量、变量、校验和以及对变量进行处理后的结果。为此引入了一种处理这种配置信息的数据模板结构，该数据模板结构需要搭配代码块来完成相

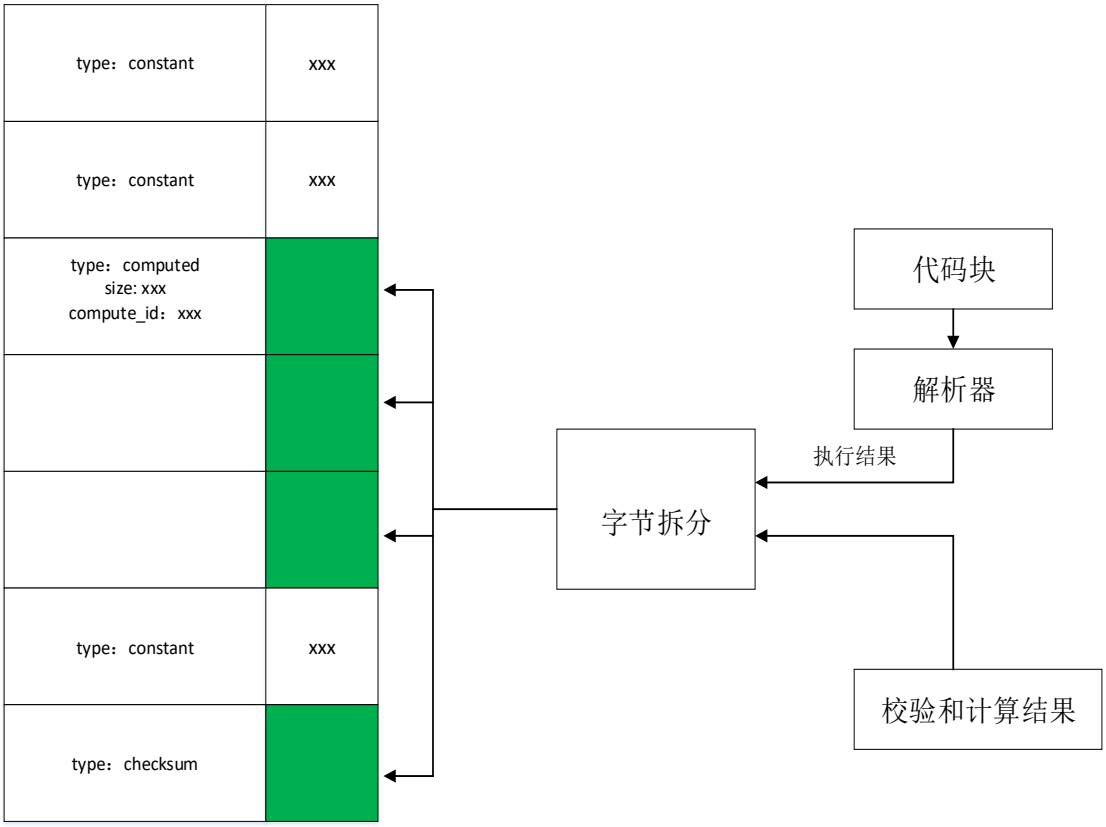
应的工作，其静态结构如下所示：

更改 type 为 occupied_by



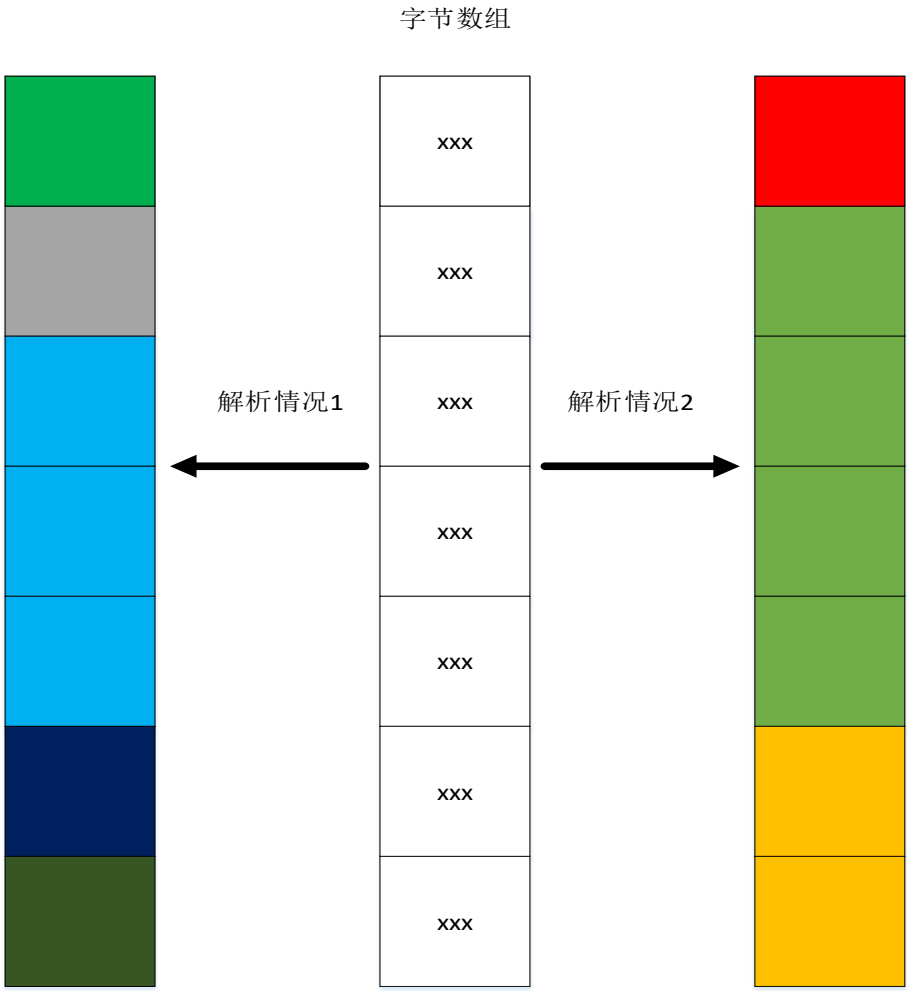
其中，绿色方块表示运行时才能确定的信息。数据类型有三种：常量、检验和以及变量，其中如果变量不需要计算的话就可以指定一个无效的 compute_id，否则指定一个有效的 compute_id 并在相应的操作模板配置信息中添加对应的代码块，该代码块用于描述程序某个量的计算流程，计算的结果用于填充上面的结构。

下图是在程序运行时填充绿色方块的示意图：



字节数组组装 bytes_array_assembly

与数据发送过程相反，接收数据时数据的格式是字节数组，需要从接收到的字节数组中提取相应的一个或多个字节来组成可用的数据。下图是从接收到的字节数组提取相应的数据示意图：

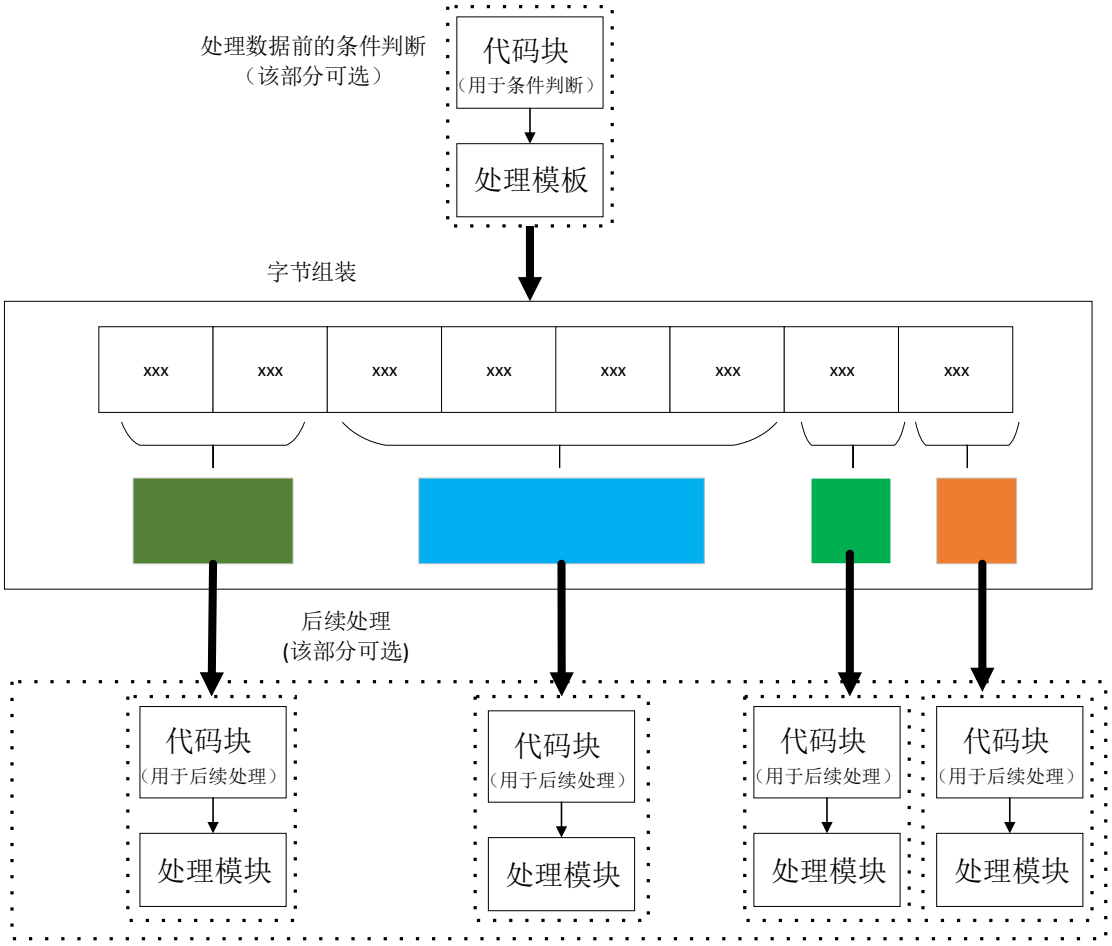


从图中可以看出对于相同的字节数组，如果解析的方式不同得到的数据也是不一样的。因此为了能够从字节数组中提取正确的数据，需要指定用于组成数据的起始字节在字节数组中的位置以及字节数，组合后的数据可能需要进行一些后续的处理，此外在正式接收数据时还要进行一些条件判断，例如接收字节数是否正确以及校验和是否正确等，这些条件可能依赖具体设备。为此引入了一种处理这种配置信息的数据模板结构，同样地该数据模板结构需要搭配代码块来完成相应接收前的条件判断以及后续的处理工作。

类似于前面指定 compute_id，如果组合后的数据需要进行一些

后续处理的话那么可以指定一个有效的 postprocess_id，同时需要给出相应的代码块，该代码块的 id 和 postprocess_id 是对应的；如果不需要处理的话，可以指定一个无效的 postprocess_id，程序处理的时候会忽略这个数据的后续处理操作。如果需要在接收数据前进行一些条件判断的话，可以在该操作模板的配置信息添加相应的代码块。

下图是运行时处理该数据模板是的流程：



代码块与其处理模块

引入背景

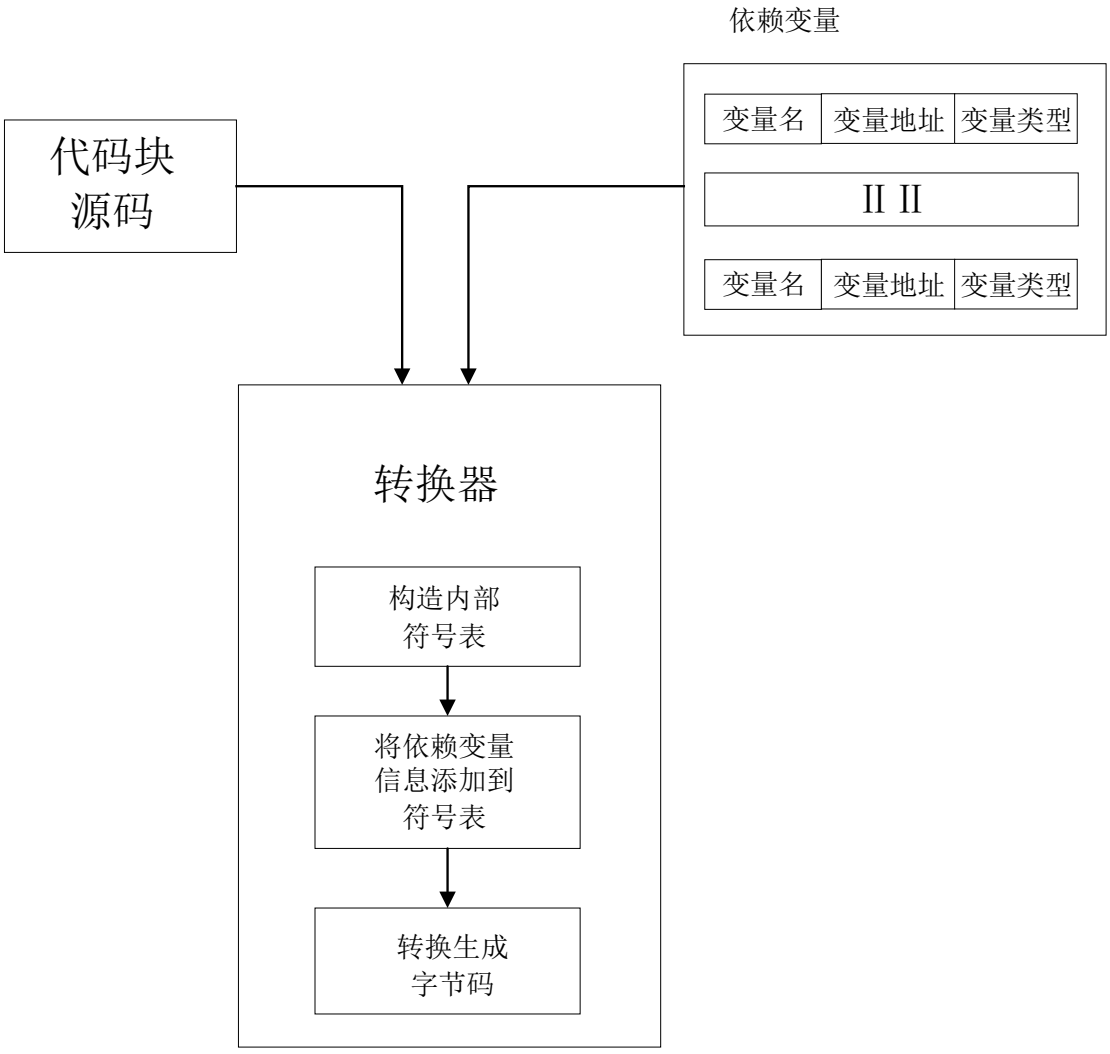
对于一些结构确定的数据格式我们可以很方便地在配置文件中取描述并在程序内部定义一些数据结构存放这些配置信息以便程序运行的时候可以使用，这些数据模板在配置的阶段是可确定的而且在程序运行的时候也不会发生变化。但是有时需要描述的数据其数值在程序运行的时候才能确定而且还是动态变化的，甚至有的时候需要描述程序中的某个量的一个计算过程，此时显然不可能去定义一些固定格式的数据结构去描述。为了解决这个问题，在配置文件中引入代码块，用代码块（实际上是一些 C 语言的表达式和语句）去描述这些动态变化的和需要计算的数据信息。

工作原理

配置文件中的代码块可以像普通的数据一样收集，因为在配置信息收集阶段代码块只不过是一串字符。这里关键在于处理方式，很显然这里需要有一套机制解析代码块并能够执行解析的结果。如果将这些代码直接编译成机器码的话，虽然执行代码块本身的逻辑是没有什么问题的，但是这样的代码块对于系统没有任何用处，它只是一个独立的存在，与系统没有任何交互。引入代码块的目的是让代码块能够引用系统中的某些变量并进行一些逻辑处理，然后将结果反馈给系统。

所以直接编译成机器码的方式是不可取的，另外一种方法是将代码转换成内部的字节码并由相应的解释器去解析执行，在代码进行转换的时候将代码块中用到的与系统进行交互的变量信息给注入进去，这些变量称为依赖变量。依赖变量的信息包括变量名、变量类型以及变量地址。

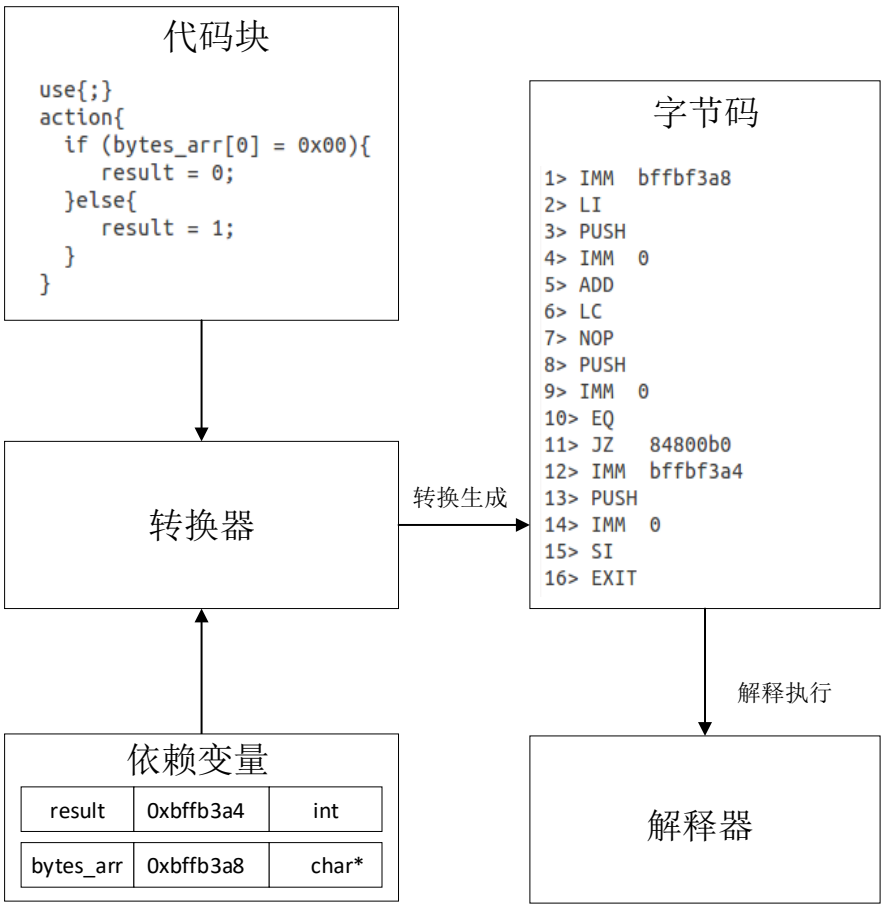
下图是代码块转换成内部的字节码流程示意图：



转换代码块的时候将其依赖变量信息传递给转换器，转换器在开始转换的前将依赖变量信息加入到内部的符号表中，在转换过程中所有对依赖变量的引用都会生成对变量地址操作的字节码。转换生成的

字节码将作为操作数据模板的一部分保存在设备模板数据信息表中。这里代码块只会在第一次使用的时候转换并执行，后面使用的话可以直接从设备模板数据信息表中取出字节码并执行。系统中代码块的处理模块主要有由词法分析和语法分析组成的转换器以及解释器等子模块组成，并向外提供转换代码块以及解释执行字节码的 API。

下图是系统某时第一次执行配置文件中的某个具体的代码块的流程示意图：



其中 result 和 bytes_arr 是依赖变量，它们和代码块一起经由转换器转换生成内部的字节码，并由解释器解释执行。如果系统后面还需要执行代码块的时候，不需要重新进行转换只需要将字节码从设备模板数据信息表取出给解释器就可以了。

配置文件中的代码块

系统选择的配置文件格式为 xml，由于代码块中的用到的空格、大于号、小于号以及取地址符号等与 xml 中特殊符号有冲突导致配置文件解析错误，因此在系统解析配置文件前需要对代码块进行相应的处理。具体地将代码块中出现的这些符号用相应的实体符号代替，例如将<替换成&#0062。