电子科技大学
University of Electronic Science and Technology of China

第六章 通用输入输出接口（一）

主讲人：漆强

电子科技大学 ytqiqiang@163.com

本章内容

- GPIO功能概述
- 基于寄存器方式控制GPIO
- 基于库函数方式控制GPIO
- BSP的设计及移植

教学目标

- 了解GPIO的功能和电路结构
- 了解GPIO模块的寄存器含义
- 掌握基于寄存器的程序开发方式

6.1 GPIO功能概述

用途

GPIO是通用输入/输出（General Purpose I/O）的简称，主要用于工业现场需要用到数字量输入/输出的场合，例如：

- **输出功能：**继电器、 LED、蜂鸣器等的控制
- **输入功能：**传感器状态、高低电平等信息的读取
- **复用功能：**片内外设的对外接口
- **时序模拟：**模拟SPI、I2C和UART等常用接口的时序




基本特性

STM32的GPIO特性

01 多种工作模式：
输出/输入/复用/模拟

02 灵活的复用模式

03 5V电压容限

04 外部中断功能

两个概念

端口和引脚

端口（PORT）

独立的外设子模块，包括多个引脚，通过多个硬件寄存器控制引脚

GPIO模块由端口GPIOA、GPIOB、GPIOC等多个独立的子模块构成
例如：端口GPIOA包括PA0～PA15这16个引脚，通过10个硬件寄存器控制引脚工作

引脚（PIN）

对应微控制器的一个管脚，归属于端口，由端口寄存器的对应位控制

PA0对应STM32F411RET6芯片的14脚，属于端口GPIOA，输出电平由端口GPIOA的输出数据寄存器GPIOA_ODR的第0位决定

目标芯片

STM32F411RET6的端口和引脚

5组端口，50个I/O引脚

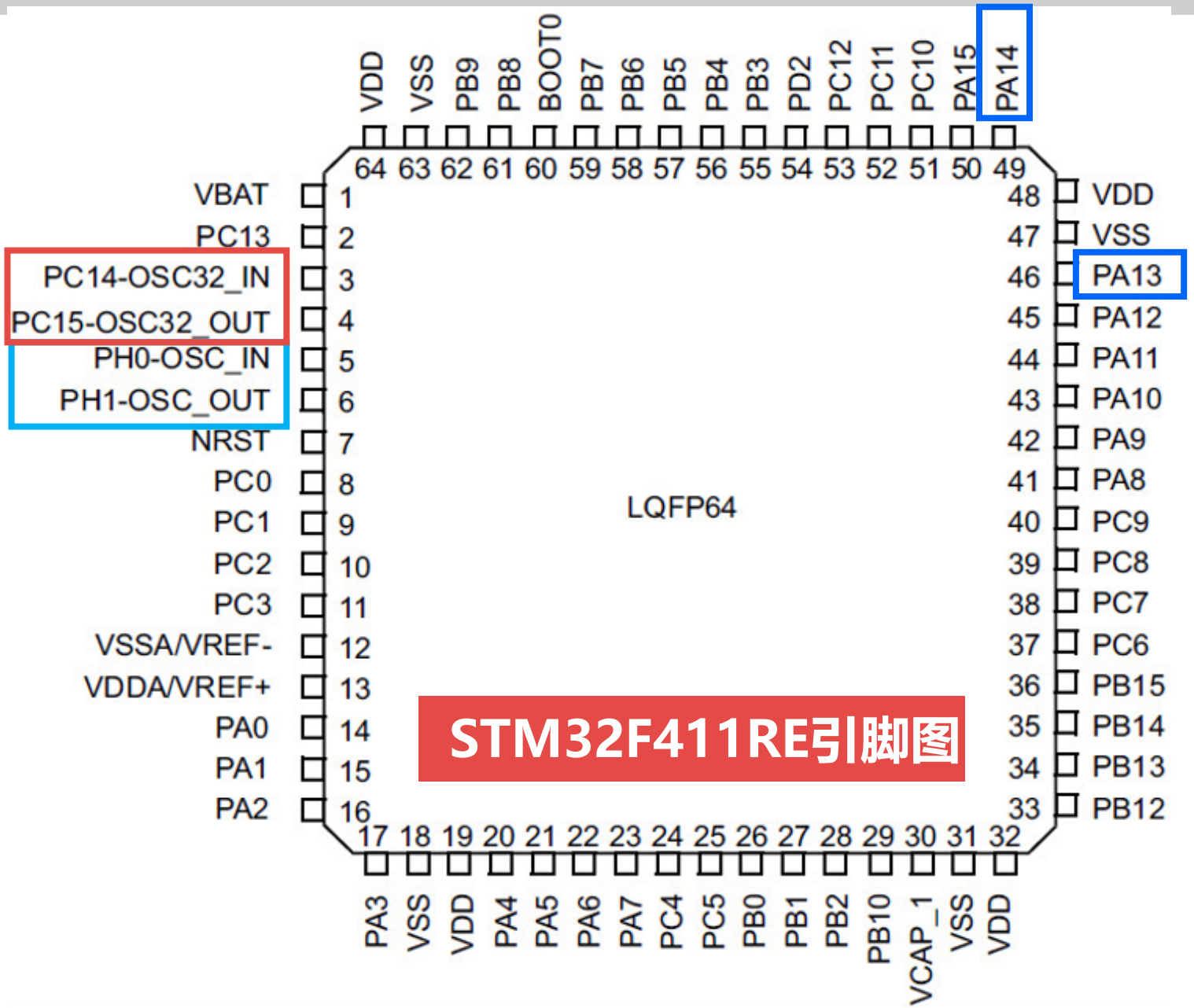
端口	引脚	数量	备注
GPIOA	PA0～PA15	16	—
GPIOB	PB0～PB15	15	PB11引脚被VCAP_1引脚取代
GPIOC	PC0～PC15	16	—
GPIOD	PD2	1	—
GPIOH	PH0、PH1	2	—

总结：一个端口默认包含16个引脚，但是不同型号的STM32微控制器所包含的端口数量及各端口包含的引脚数量各不相同，具体信息可以查询芯片的数据手册

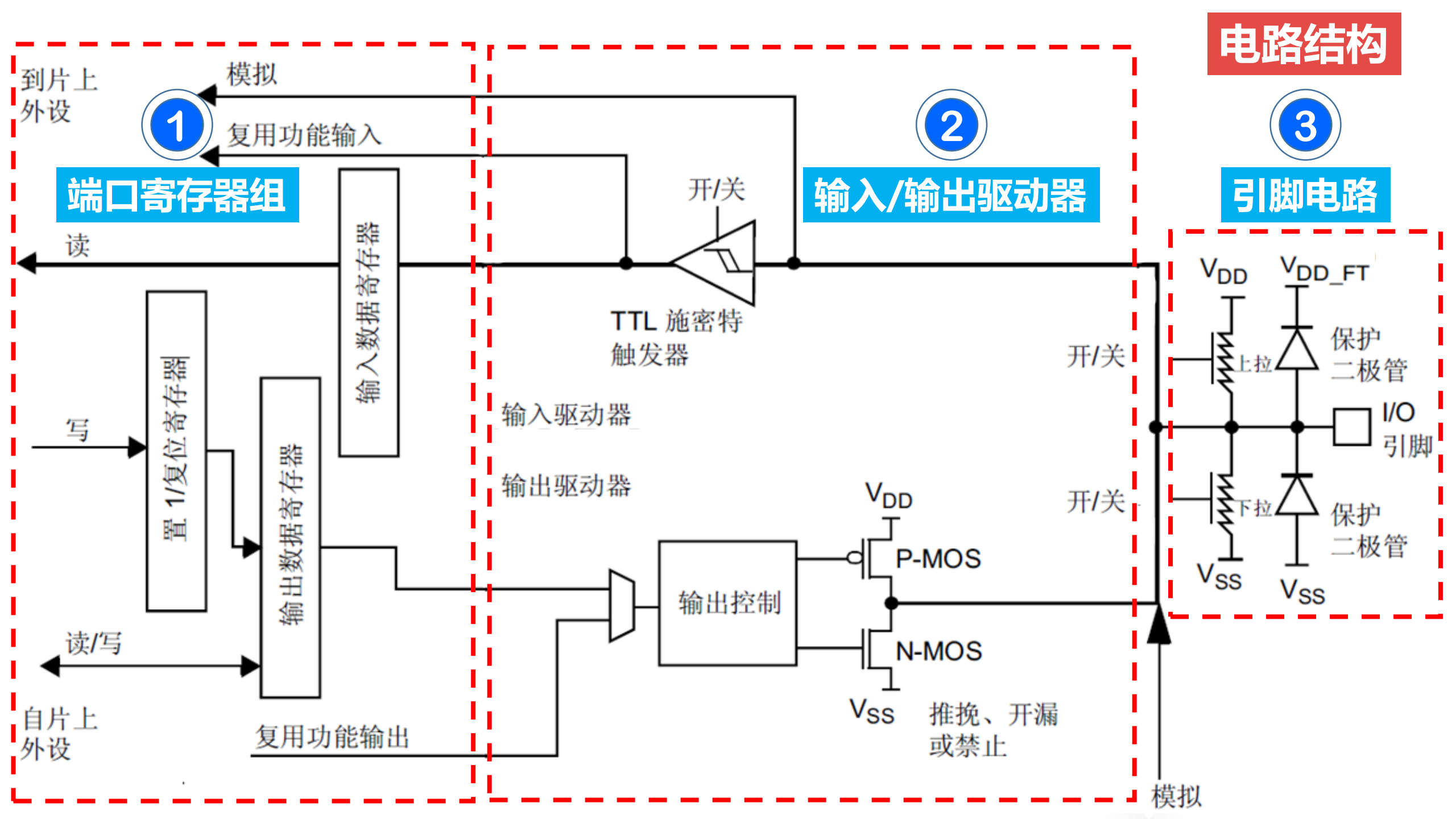
引脚占用情况

- PH0和PH1接入HSE
- PC14和PC15接入LSE
- PA13和PA14作为SWD接口

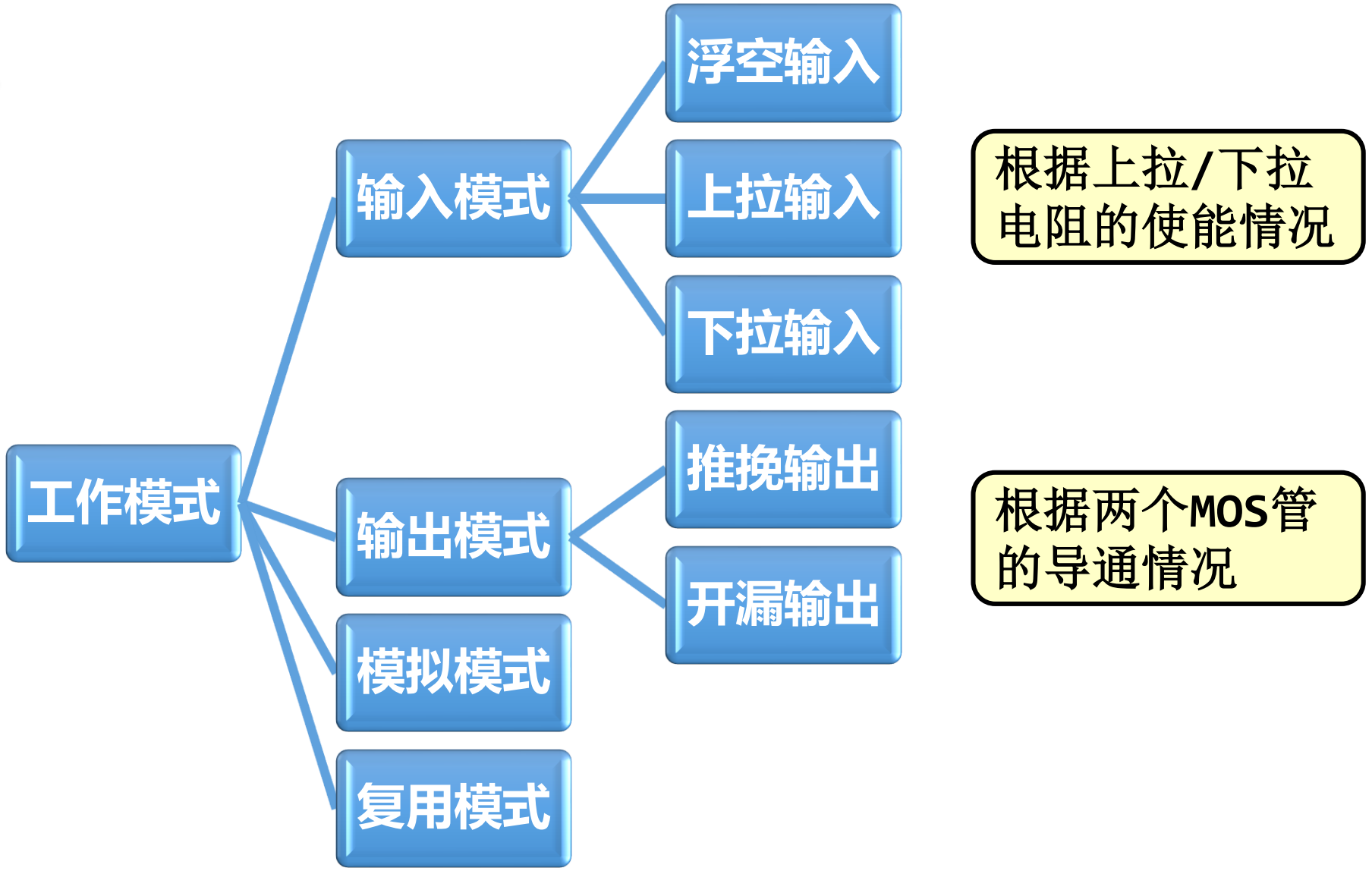
44



6.2 GPIO模块的电路结构


电路结构
到片上外设
模拟
1
复用功能输入
端口寄存器组
2
输入/输出驱动器
3
引脚电路
读
输入数据寄存器
开/关
TTL 施密特触发器
写
置 1/复位寄存器
输出数据寄存器
输入驱动器
输出驱动器
开/关
开/关
VDD
P-MOS
输出控制
N-MOS
VSS
推挽、开漏或禁止
读/写
自片上外设
复用功能输出
模拟
VDD
VDD_FT
上拉
保护二极管
I/O 引脚
下拉
保护二极管
VSS
VSS

工作模式分类


工作模式
输入模式
浮空输入
上拉输入
下拉输入
输出模式
推挽输出
开漏输出
模拟模式
复用模式
根据上拉/下拉电阻的使能情况
根据两个MOS管的导通情况

输入模式

三种输入模式：浮空输入、上拉输入、下拉输入

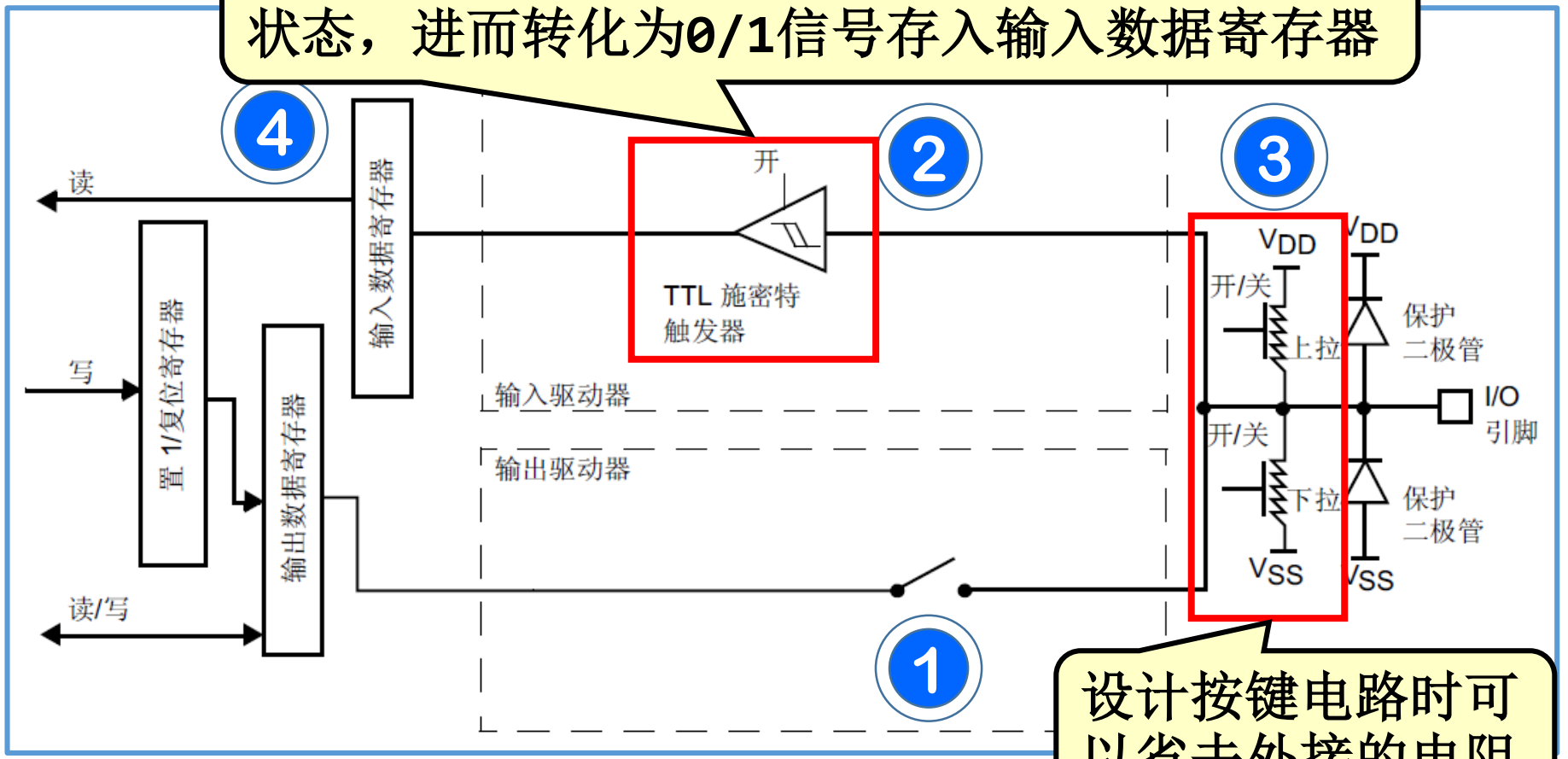
① 输出驱动器关闭

② 施密特触发器打开，可以获取引脚状态

③ 通过寄存器使能上/下拉电阻：浮空/上拉/下拉输入

④ 引脚电平状态将存入输入数据寄存器

利用门阀电压将引脚电压转换为高/低电平两种状态，进而转化为0/1信号存入输入数据寄存器



输出模式

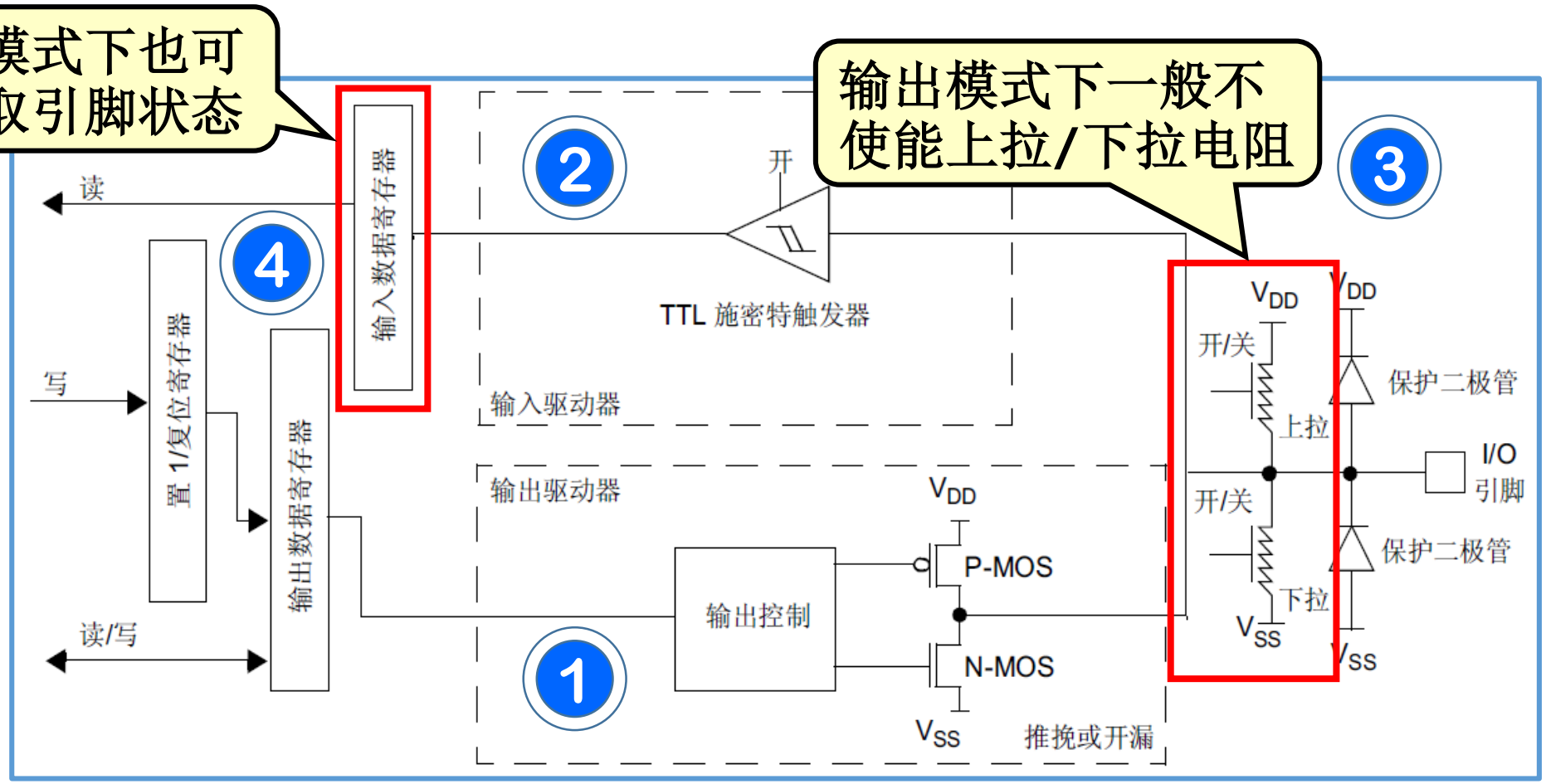
两种输出模式：开漏输出和推挽输出

① 输出驱动器打开

② 施密特触发器打开

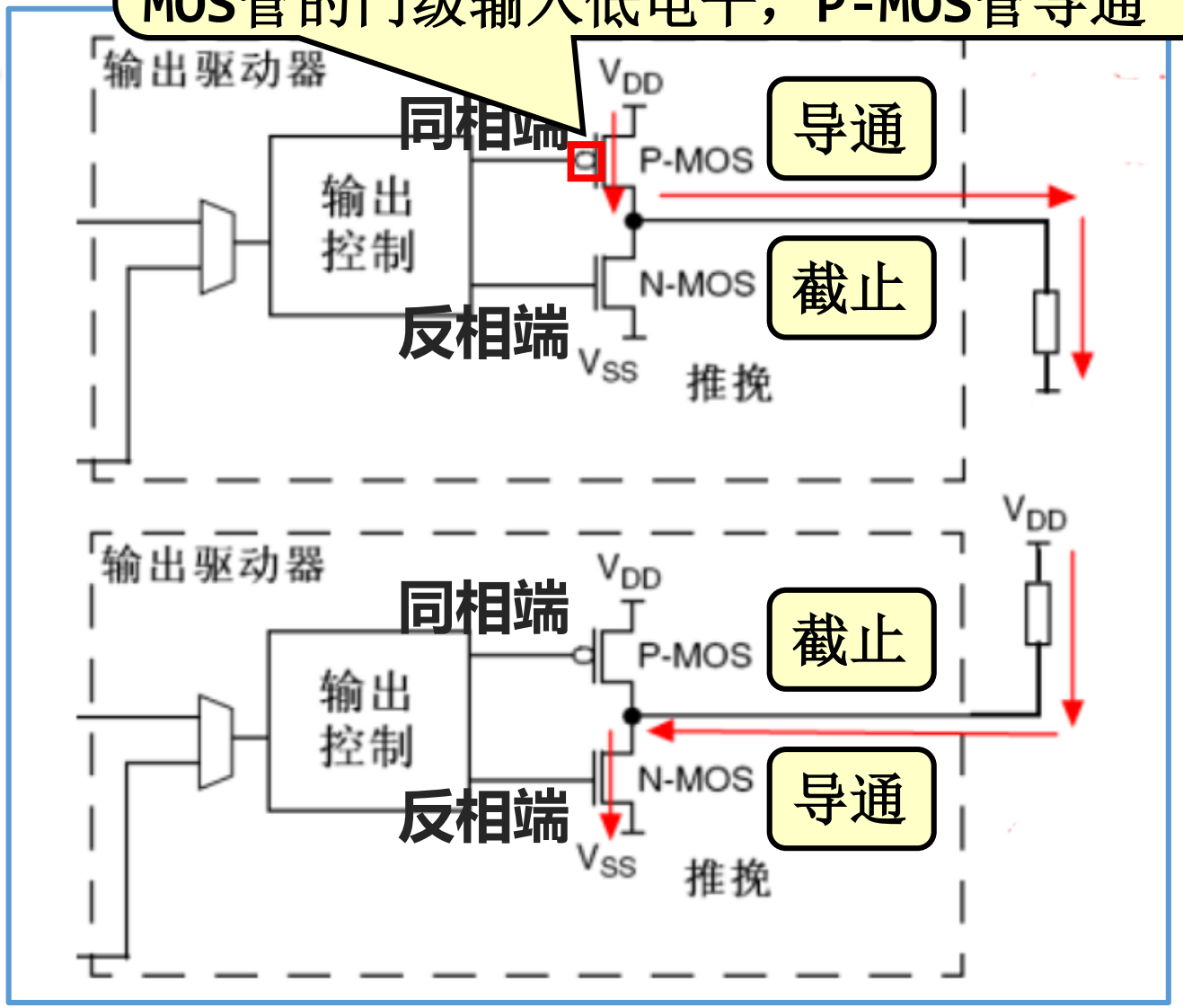
③ 通过寄存器使能上拉/下拉电阻

④ 通过输入数据寄存器获取引脚状态



P-MOS管的小圆圈表示还要进行一次反相，因此当同相端输出“1”时，反相后，P-MOS管的门级输入低电平，P-MOS管导通

推挽输出



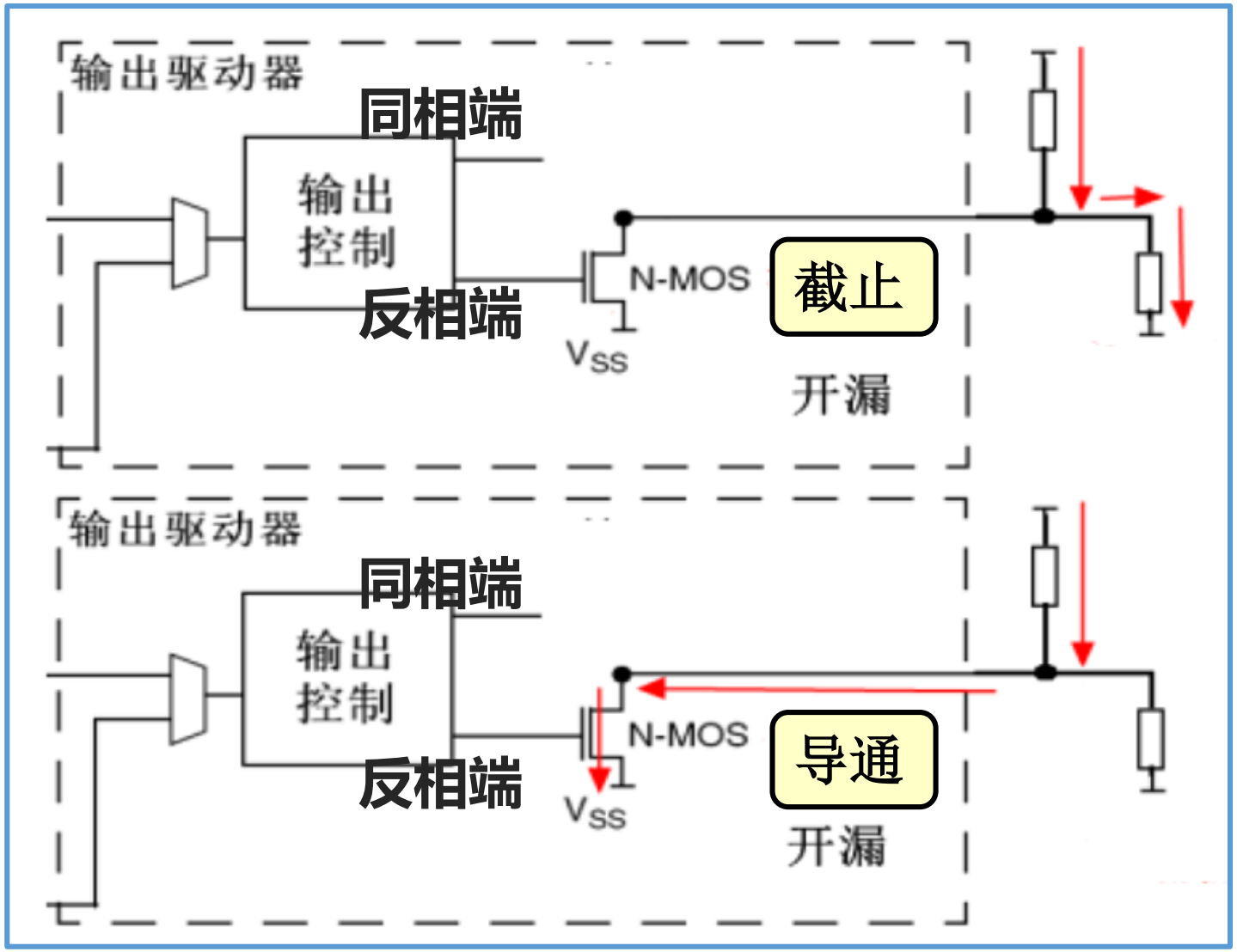
推挽输出时，P-MOS管和N-MOS管轮流工作，可以输出高电平或低电平

当输出数据寄存器对应位为“1”时，同相端输出“1”，反相端输出“0”。此时P-MOS管导通，N-MOS管截止，引脚输出高电平

当输出数据寄存器对应位为“0”时，同相端输出“0”，反相端输出“1”。此时P-MOS管截止，N-MOS管导通，引脚输出低电平

开漏输出

开漏输出时，P-MOS管关闭，只有N-MOS管工作，此时只能输出低电平。要输出高电平必须外加上拉电阻，主要用于I2C总线



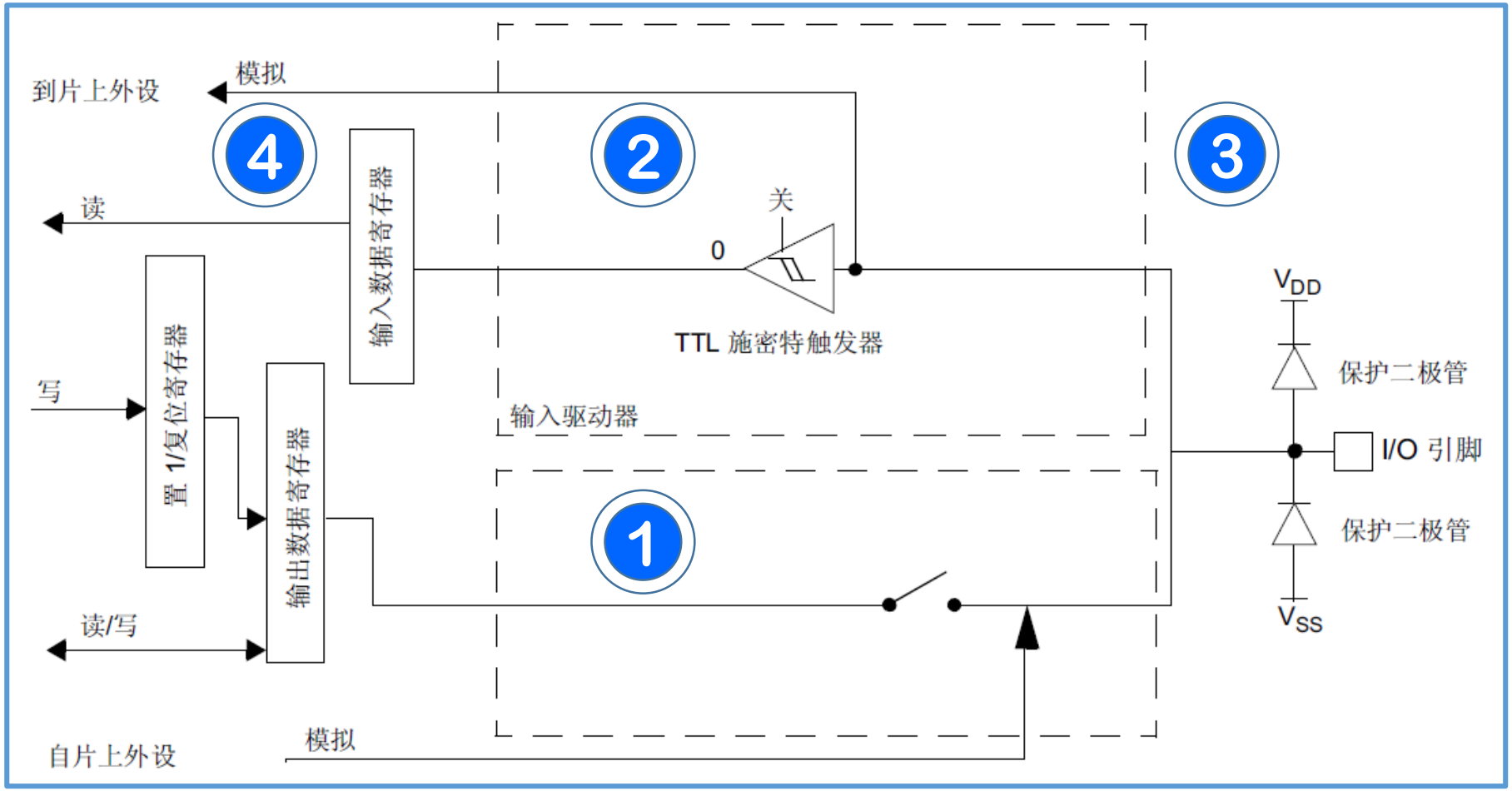
当输出数据寄存器对应位为“1”时，反相端输出“0”。此时N-MOS管截止，由外加的上拉电阻提供高电平

当输出数据寄存器对应位为“0”时，反相端输出“1”。此时N-MOS管导通，引脚输出低电平

模拟模式

单纯的引脚模拟状态或者引脚作为片内模拟外设的复用脚

① 输出驱动器关闭
② 施密特触发器旁路
③ 上/下拉电阻关闭
④ 输入数据寄存器的值为0
⑤ 功耗最小



区别

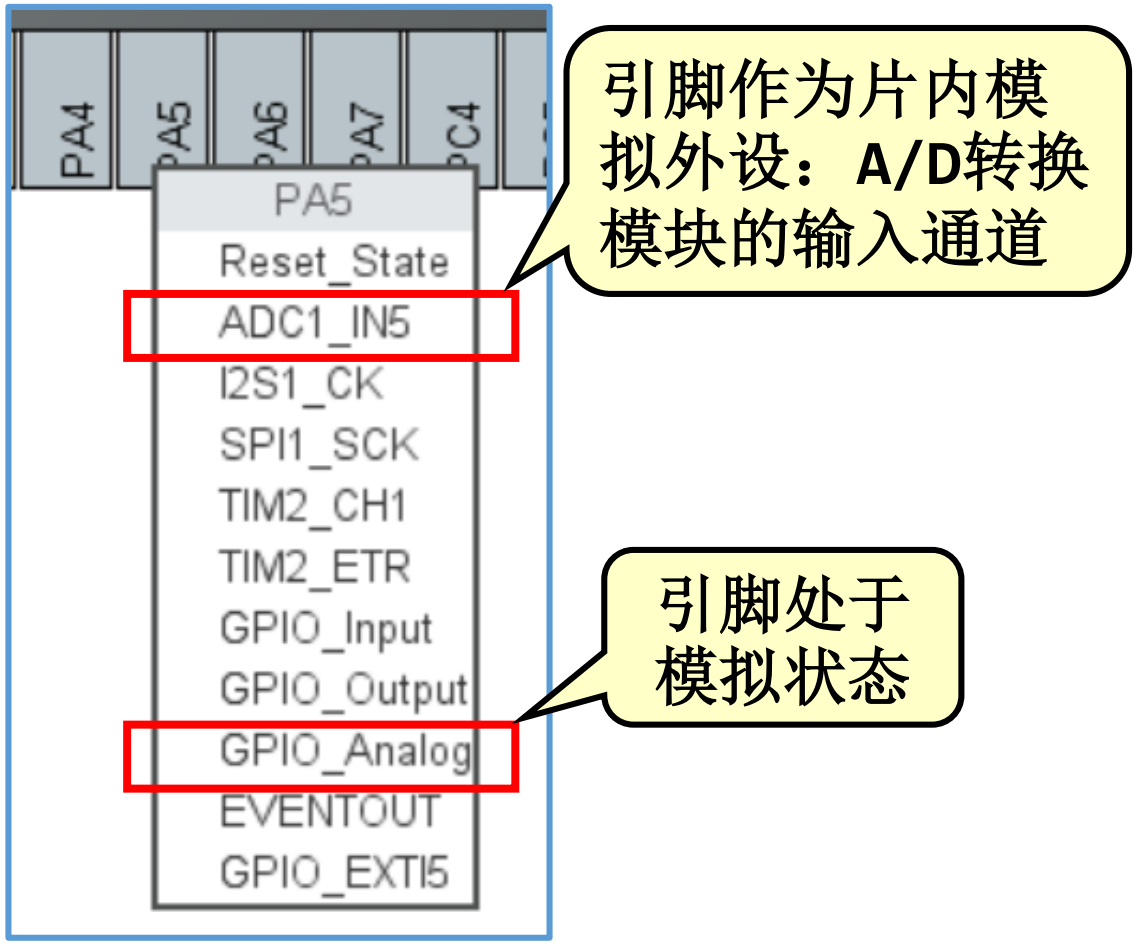
模拟状态

表示引脚功能选择为模拟模式，但不作为任何片内模拟外设的复用脚，只是为了减少系统功耗

模拟外设复用引脚

表示引脚作为片内模拟外设（A/D转换模块、D/A转换模块、模拟比较器等）的复用脚，用于完成相应的功能操作

CubeMX软件中的引脚分配操作



复用模式

引脚电平状态由片内外设控制

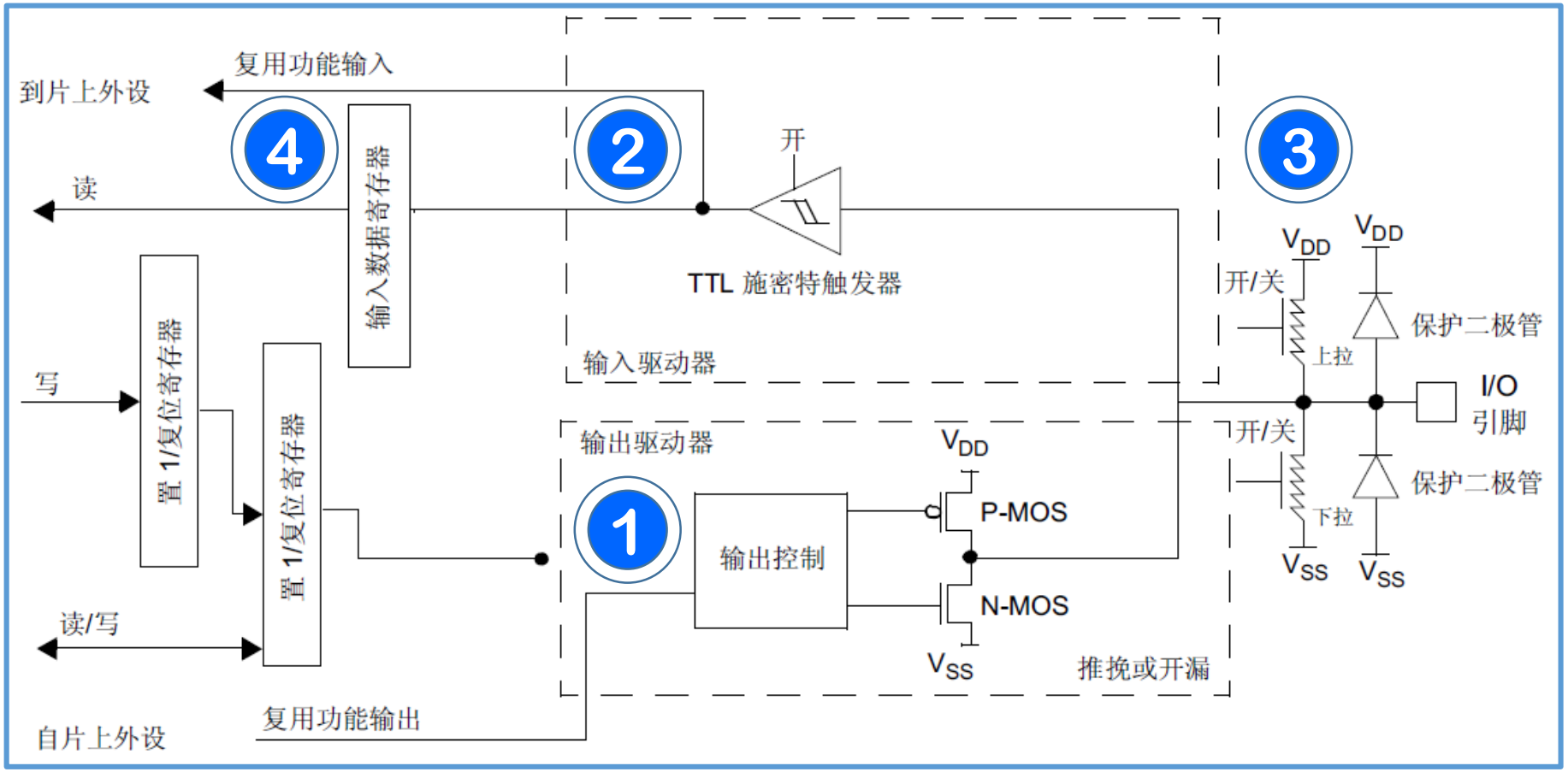
一般情况下，处于复用模式的GPIO引脚的参数可借助CubeMX软件自动配置

① 输出可配置为推挽或者开漏模式

② 施密特触发器打开

③ 通过寄存器使能上拉/下拉电阻

④ 通过输入数据寄存器获取引脚状态



查看复用功能

快速查看引脚复用功能的方法

在CubeMX软件的引脚分配图中点击引脚即可弹出引脚的复用功能

① I2S1_CK ：音频总线I2S1的时钟信号

② SPI1_SCK ：同步串口SPI1的时钟信号

③ TIM2_CH1 ：定时器2的捕获/比较通道1

④ TIM2_ETR ：定时器2的外部触发输入

⑤ EVENTOUT：事件输出

用法总结

GPIO工作模式的用法总结

- **输入模式**
  - 浮空输入：按键识别
  - 上拉输入：IO内部上拉电阻输入
  - 下拉输入：IO内部下拉电阻输入
- **模拟模式**
  - 作为片内模拟外设的对外引脚
  - 单纯作为低功耗使用

用法总结

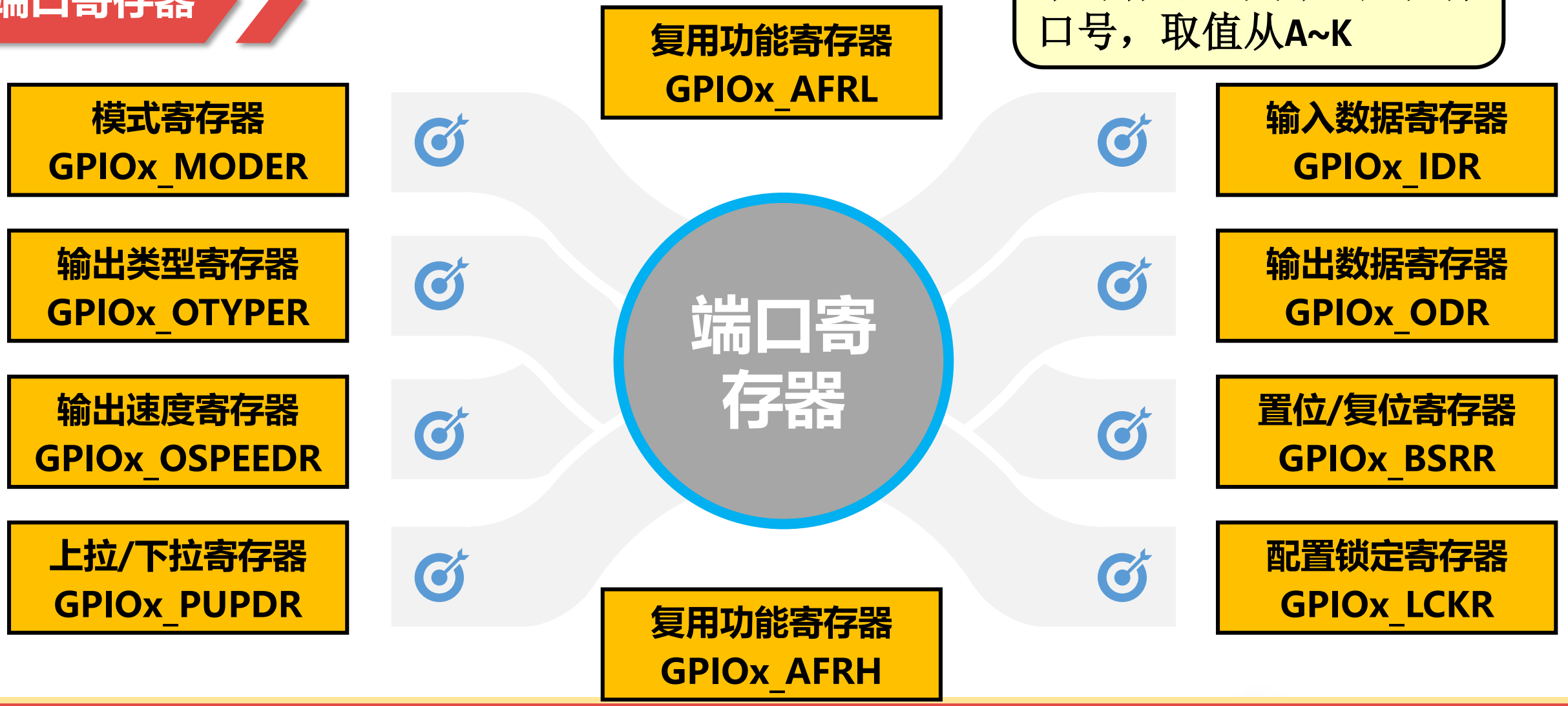
GPIO工作模式的用法总结

- **输出模式**
  - 推挽输出：可以输出高/电平，主要用于连接数字器件，如指示灯和继电器等模块；
  - 开漏输出：只能输出低电平，适合于电流型驱动，也可作为电平转换。
- **复用模式**
  - 复用推挽：片内外设功能（URAT的TX，RX，SPI的MOSI，MISO，SCK，SS）；
  - 复用开漏：片内外设功能（I2C的SCL，SDA）。

6.3 GPIO模块的寄存器介绍

端口寄存器

每一个端口都包括这10个寄存器，其中x表示端口号，取值从A~K



配置寄存器

模式寄存器GPIOx_MODER

31~30	29~28	27~26	25~24	23~22	21~20	19~18	17~16
MODER 15[1:0]	MODER 14[1:0]	MODER 13[1:0]	MODER 12[1:0]	MODER 11[1:0]	MODER 10[1:0]	MODER 9[1:0]	MODER 8[1:0]
15~14	13~12	11~10	9~8	7~6	5~4	3~2	1~0
MODER 7[1:0]	MODER 6[1:0]	MODER 5[1:0]	MODER 4[1:0]	MODER 3[1:0]	MODER 2[1:0]	MODER 1[1:0]	MODER 0[1:0]

- 32位寄存器，每2位一组，用于设置对应引脚的工作模式，如：bit0和bit1控制Px0
- 00：输入模式（复位值）01：输出模式 10：复用模式 11：模拟模式

配置寄存器

输出类型寄存器GPIOx_OTYPER

31~16															
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0

- 32位寄存器，高16位保留，低16位的每1位用于设置对应引脚的输出类型
- 0：推挽输出（复位值）1：开漏输出

配置寄存器

输出速度寄存器GPIOx_OSPEEDR

31~30	29~28	27~26	25~24	23~22	21~20	19~18	17~16
OSPEEDR 15[1:0]	OSPEEDR 14[1:0]	OSPEEDR 13[1:0]	OSPEEDR 12[1:0]	OSPEEDR 11[1:0]	OSPEEDR 10[1:0]	OSPEEDR 9[1:0]	OSPEEDR 8[1:0]
15~14	13~12	11~10	9~8	7~6	5~4	3~2	1~0
OSPEEDR 7[1:0]	OSPEEDR 6[1:0]	OSPEEDR 5[1:0]	OSPEEDR 4[1:0]	OSPEEDR 3[1:0]	OSPEEDR 2[1:0]	OSPEEDR 1[1:0]	OSPEEDR 0[1:0]

- 32位寄存器，每2位一组，用于设置对应引脚的输出速度
- 00：低速（复位值）01：中速 10：高速 11：超高速

配置寄存器

上拉/下拉寄存器GPIOx_PUPDR

31~30	29~28	27~26	25~24	23~22	21~20	19~18	17~16
PUPDR 15[1:0]	PUPDR 14[1:0]	PUPDR 13[1:0]	PUPDR 12[1:0]	PUPDR 11[1:0]	PUPDR 10[1:0]	PUPDR 9[1:0]	PUPDR 8[1:0]
15~14	13~12	11~10	9~8	7~6	5~4	3~2	1~0
PUPDR 7[1:0]	PUPDR 6[1:0]	PUPDR 5[1:0]	PUPDR 4[1:0]	PUPDR 3[1:0]	PUPDR 2[1:0]	PUPDR 1[1:0]	PUPDR 0[1:0]

- 32位寄存器，每2位一组，用于使能对应引脚的上拉/下拉电阻
- 00：无上拉和下拉电阻（复位值）01：使能上拉电阻 10：使能下拉电阻 11：保留

数据寄存器

输入数据寄存器GPIOx_IDR

31~16															
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0

- 32位寄存器，高16位保留，低16位的每1位用于存放对应引脚的电平状态
- 0：对应引脚输入低电平 1：对应引脚输入高电平

数据寄存器

输出数据寄存器GPIOx_ODR

31~16															
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0

- 32位寄存器，高16位保留，低16位的每1位用于控制对应引脚输出高/低电平
- 0：控制对应引脚输出低电平 1：控制对应引脚输出高电平

数据寄存器

置位/复位寄存器GPIOx_BSRR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR 15	BR 14	BR 13	BR 12	BR 11	BR 10	BR 9	BR 8	BR 7	BR 6	BR 5	BR 4	BR 3	BR 2	BR 1	BR 0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS 15	BS 14	BS 13	BS 12	BS 11	BS 10	BS 9	BS 8	BS 7	BS 6	BS 5	BS 4	BS 3	BS 2	BS 1	BS 0

- 高16位控制对应引脚输出低电平：写入1对应引脚输出低电平；写入0，没有任何作用
- 低16位控制对应引脚输出高电平：写入1对应引脚输出高电平；写入0，没有任何作用

使用举例

要求PA0输出低电平，PA15输出高电平

方法一：使用输出数据寄存器实现

```
unsigned int   value;                  //  定义变量value，存放寄存器的值
value = GPIOA_ODR;                     //  读出输出数据寄存器原有的值
value = value & 0xFFFFFFFE;  //  设置value变量的bit0为0
value = value | 0x00008000;  //   设置value变量的bit15为1
GPIOA_ODR = value;                     //   将新的配置数据写入输出数据寄存器
```

方法二：使用置位/复位寄存器实现

```
GPIOA_BSRR = 0x00018000;    //  设置PA0输出低，PA15输出高
```

结论：设置多个引脚输出高/低电平时，使用置位/复位寄存器更加简单

电子科技大学
University of Electronic Science and Technology of China

6.4 基于寄存器方式控制GPIO

一　利用指针访问单个寄存器

端口地址范围

STM32F411芯片的GPIO模块各端口寄存器的地址范围

端口号	寄存器组地址范围	备注
GPIOA	0x4002 0000 ~ 0x4002 03FF	每组端口占用了0x000 ~ 0x3FF共1K字节的存储空间，而实际每组端口只包括10个32位的寄存器，只需要40个字节的存储空间，多余的存储空间预留给芯片升级使用。
GPIOB	0x4002 0400 ~ 0x4002 07FF	
GPIOC	0x4002 0800 ~ 0x4002 0BFF	
GPIOD	0x4002 0C00 ~ 0x4002 0FFF	
GPIOE	0x4002 1000 ~ 0x4002 13FF	
GPIOH	0x4002 1C00 ~ 0x4002 1FFF	

注意：端口GPIOE和GPIOH之间的地址预留给端口GPIOF和GPIOG使用

寄存器地址

端口GPIOA对应寄存器的起始地址及偏移量

寄存器名称	寄存器起始地址	偏移量
GPIOA_MODER	0x4002 0000	0x00
GPIOA_OTYPER	0x4002 0004	0x04
GPIOA_OSPEEDR	0x4002 0008	0x08
GPIOA_PUPDR	0x4002 000C	0x0C
GPIOA_IDR	0x4002 0010	0x10
GPIOA_ODR	0x4002 0014	0x14
GPIOA_BSRR	0x4002 0018	0x18
GPIOA_LCKR	0x4002 001C	0x1C
GPIOA_AFRL	0x4002 0020	0x20
GPIOA_AFRH	0x4002 0024	0x24

1

0x40020000作为模式寄存器的起始地址，也是端口GPIOA寄存器组的起始地址

2

所有寄存器为32位，占用四个字节的存储空间

地址转换方法

利用指针完成地址转换

1. 对于C语言的编译器而言，寄存器的地址值只代表一个有符号的常数，无法代表地址；
2. 在C语言中，利用指针类型来存放变量的地址，利用指针类型定义的变量称为指针变量；
3. 利用强制类型转换可以将常数转为指针变量；
4. 指针的基类型表示指针所指向变量的类型，它决定了从该地址开始，可以访问的地址范围；
5. 指针的解引用表示从指针所指向的地址中取出存放的数据。

地址转换方法

利用指针完成地址转换

unsigned int *p　　表示定义一个指向无符号整型的指针

p= 0x40020000UL　表示为指针变量p赋值，指向地址单元0x40020000

*p 即指针的解引用，表示访问从地址单元0x40020000开始的4个地址单元的内容

① 寄存器地址值不具备符号特性
② 寄存器为32位，在32位处理器系统中，unsigned long类型占用4个字节，32位

地址转换过程

寄存器定义：#define GPIOA_MODER *(volatile unsigned int *)(0x40020000UL)

1. 0x40020000UL 将有符号常数转换为一个无符号32位常数

2. volatile unsigned int * (0x40020000UL)表示
将常数强制转换为指向无符号整型的指针，即寄存器的地址

避免编译器优化

指针的基类型

3. *(volatile unsigned int *)(0x40020000UL)
表示指针的“解引用”，访问(读/写)指针所指向地址的数据

代表从地址0x40020000开始的4字节存储空间

4. 利用define取一个别名，用GPIOA_MODER代表该存储空间
通过GPIOA_MODER来访问对应的存储空间

演示例程

演示例程：驱动指示灯

1 例程目标

演示基于寄存器的程序开发方式

2 例程内容

控制Nucleo开发板上的用户指示灯LD2连续闪烁

操作步骤

1. 利用MDK新建工程

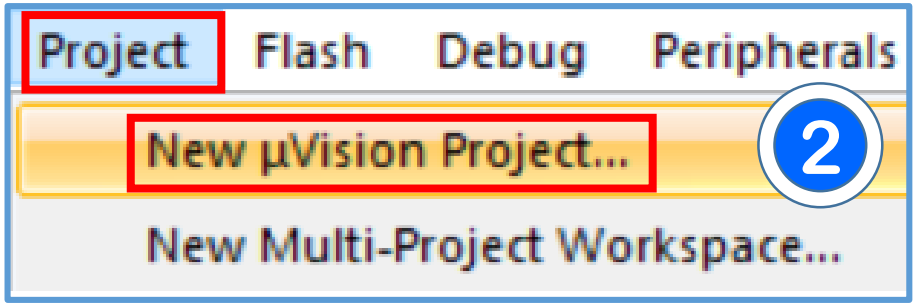
① 桌面建立Register文件夹；

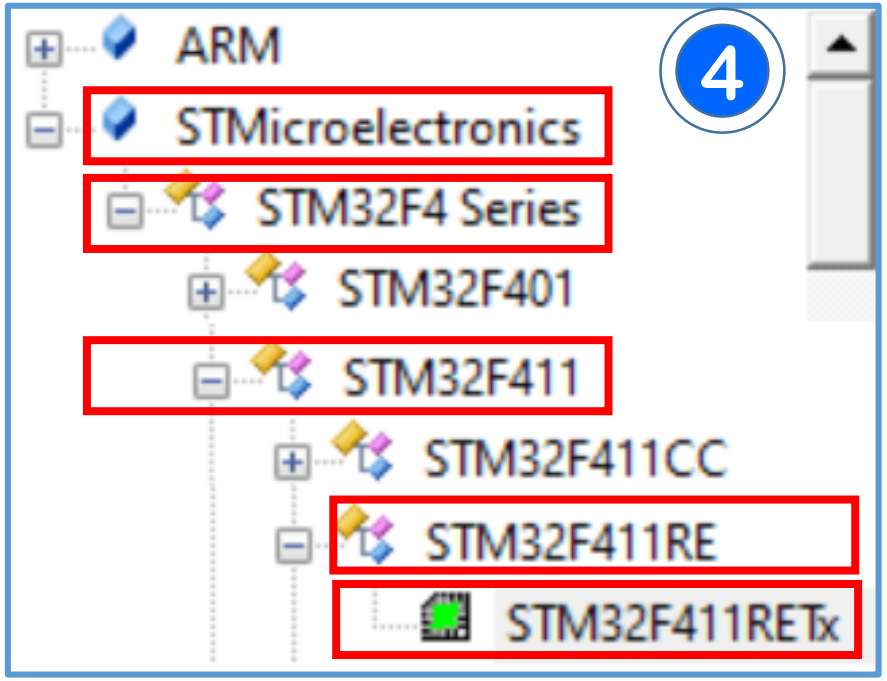
② 启动MDK软件，选择Project菜单；

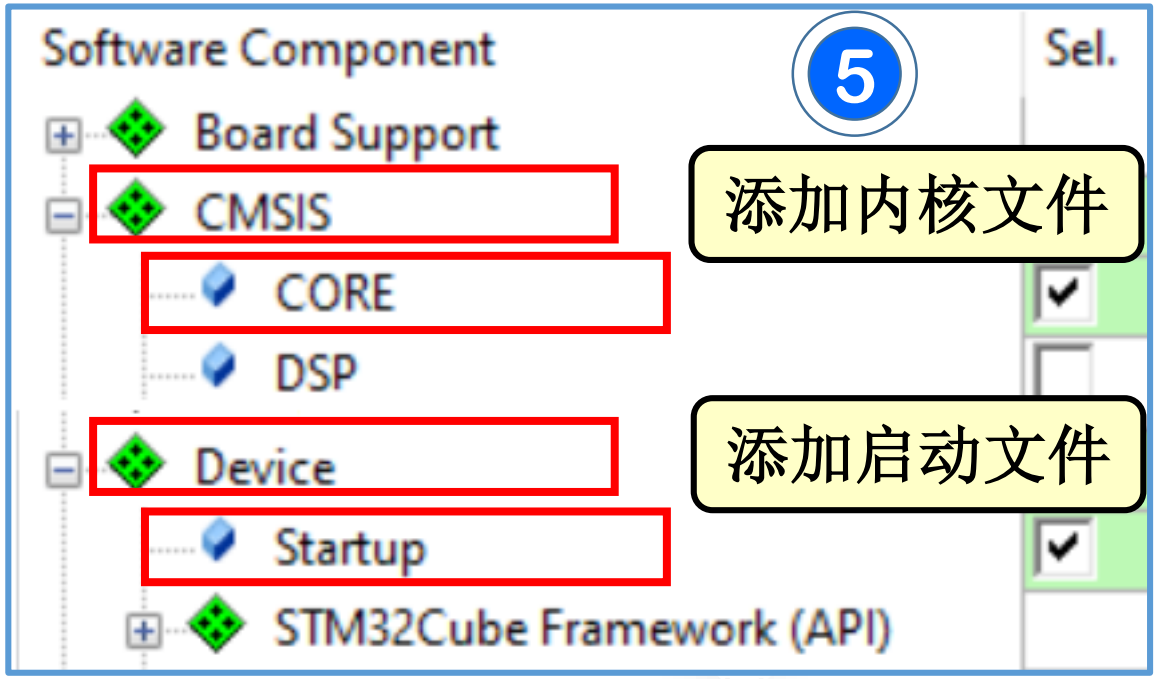
③ 在弹出的新建工程窗口中，找到Register文件夹，输入工程名LED；

④ 选择目标MCU；

⑤ 在RTE环境中勾选CORE和Startup组件。







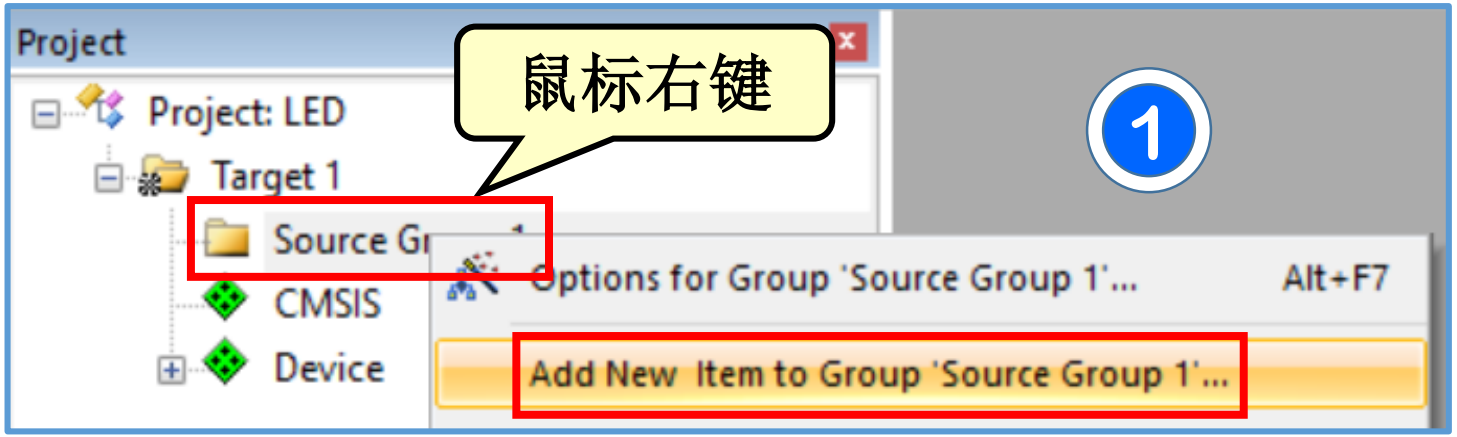
操作步骤

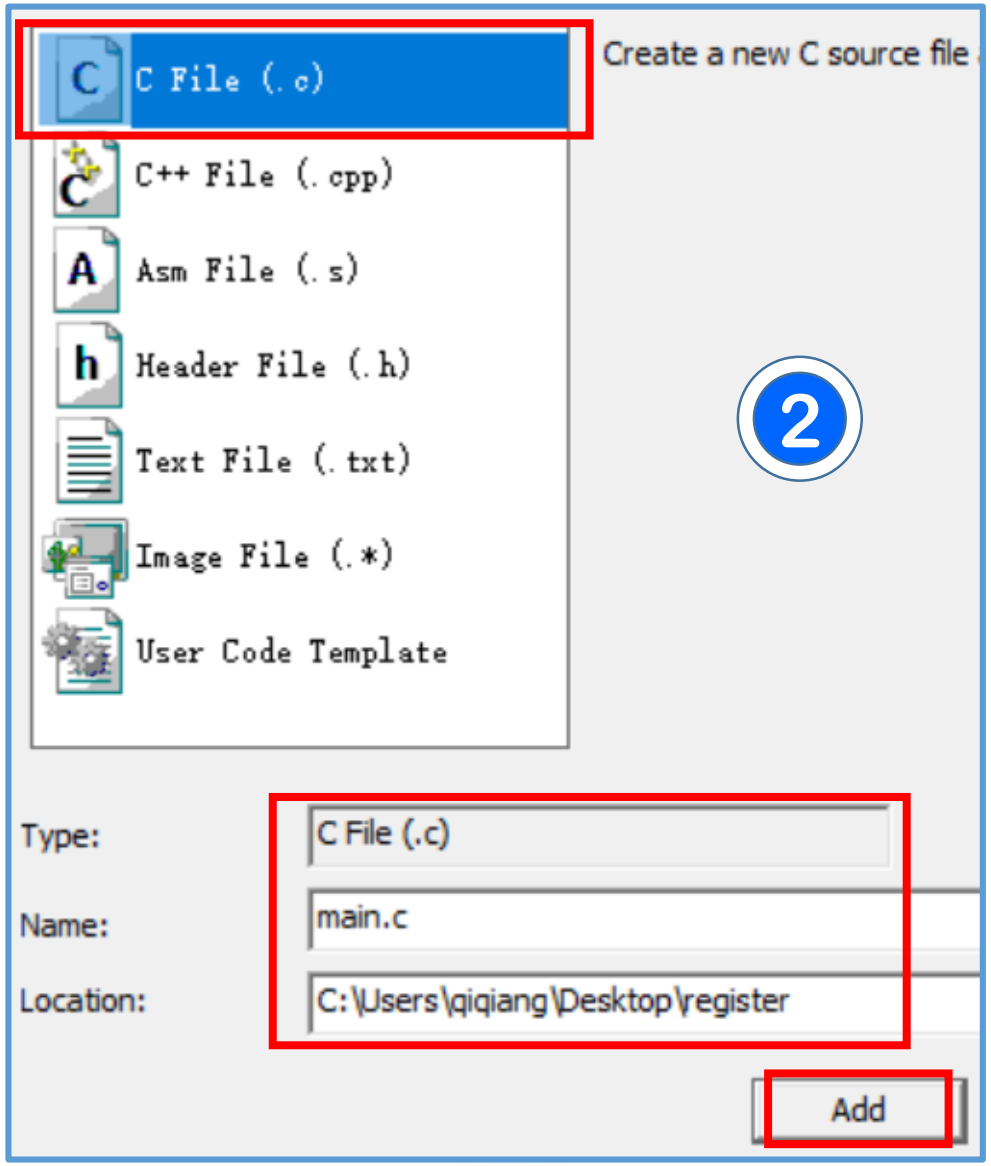
2. 新建源文件并编写用户代码

① 工程窗口 -> Source Group 1

-> Add New Item to Group Source Group 1

② 选择C File类型文件，文件名为main.c，利用Add按钮加入到工程中





```c
#define GPIOA_MODER   *(unsigned  int *)(0x40020000UL) // 模式寄存器
#define GPIOA_ODR     *(unsigned  int *)(0x40020014UL) // 输出数据寄存器
#define RCC_AHB1ENR   *(unsigned  int *)(0x40023830UL) // 外设时钟使能寄存器
int main()
{
  unsigned  int delay = 1000000; // 延时变量
  RCC_AHB1ENR |= 1<<0;           // 开启端口 GPIOA 的时钟
  GPIOA_MODER &= ~(3<<(5*2);     // 清除模式寄存器的 bit11:bit10 为 00
  GPIOA_MODER |= 1<<(5*2);       // 设置 bit11:bit10 = 01，PA5 为推挽输出类型
  while(1)
  {
    GPIOA_ODR |= 1<<5;           // 设置 bit5 为 1，PA5 输出高电平，开启 LD2
    delay = 1000000;
    while(delay--);              // 延时
    GPIOA_ODR &= ~(1<<5);        // 设置 bit5 为 0，PA5 输出低电平，关闭 LD2
    delay = 1000000;
    while(delay--);              // 延时
  }
}
```

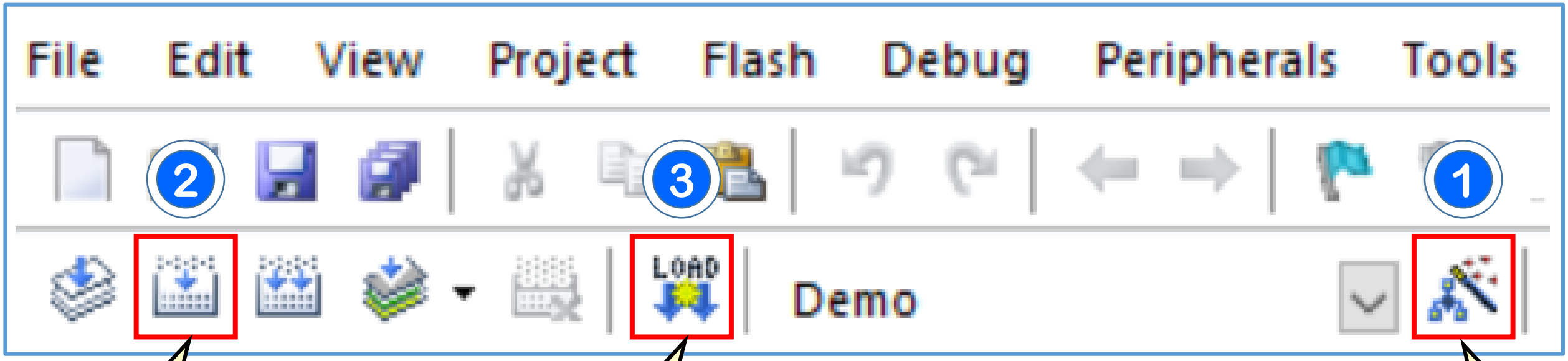
扩展任务
使用置位/复位寄存器
BSRR控制指示灯LD2

芯片复位后，引脚默认的输出类型为推挽输出

在main.c文件中添加用户代码

工程设置

利用工具栏的三个图标完成工程设置、程序编译和下载



程序编译

程序下载

在Debug标签页中的选择所使用的仿真器，并在仿真器设置窗口中勾选“Reset and Run”

工程设置

二 利用结构体指针访问寄存器组

寄存器组 多个寄存器的定义 ？ 地址连续 ➡ 利用结构体实现

寄存器名称	寄存器起始地址	偏移量
GPIOA_MODER	0x4002 0000	0x00
GPIOA_OTYPER	0x4002 0004	0x04
GPIOA_OSPEEDR	0x4002 0008	0x08
GPIOA_PUPDR	0x4002 000C	0x0C
GPIOA_IDR	0x4002 0010	0x10
GPIOA_ODR	0x4002 0014	0x14
GPIOA_BSRR	0x4002 0018	0x18
GPIOA_LCKR	0x4002 001C	0x1C
GPIOA_AFRL	0x4002 0020	0x20
GPIOA_AFRH	0x4002 0024	0x24

➡

```
struct GPIO
{
  volatile unsigned int MODER;
  volatile unsigned int OTYPER;
  volatile unsigned int OSPEEDR;
  volatile unsigned int PUPDR;
  volatile unsigned int IDR;
  volatile unsigned int ODR;
  volatile unsigned int BSRR;
  volatile unsigned int LCKR;
  volatile unsigned int AFR[2];
};
```

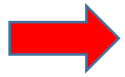
unsigned int
数据类型占用4个字节

注意：目前只定义结构体模板，还没有定义结构体变量

typedef

为系统固有的或用户自定义的数据类型定义一个别名

```
struct GPIO
{
  volatile unsigned int MODER;
  volatile unsigned int OTYPER;
  volatile unsigned int OSPEEDR;
  volatile unsigned int PUPDR;
  volatile unsigned int IDR;
  volatile unsigned int ODR;
  volatile unsigned int BSRR;
  volatile unsigned int LCKR;
  volatile unsigned int AFR[2];
};
```



别名

```
typedef struct
{
  __IO uint32_t MODER;
  __IO uint32_t OTYPER;
  __IO uint32_t OSPEEDR;
  __IO uint32_t PUPDR;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t LCKR;
  __IO uint32_t AFR[2];
} GPIO_TypeDef;
```

区别

两种方式在定义结构体变量时的区别

利用结构体模板定义结构体变量

```
struct  GPIO      GPIOA;      //  定义结构体变量
struct  GPIO   *  pGPIOA;     //  定义指向结构体变量的指针
```

利用typedef定义结构体变量

```
GPIO_TypeDef     GPIOA;      //  定义结构体变量
GPIO_TypeDef  *  pGPIOA;     //  定义指向结构体变量的指针
```

寄存器组定义

```
#define GPIOA    ((GPIO_TypeDef *) 0x40020000UL)
```

1. 0x40020000UL 将有符号常数转换为一个无符号32位常数
2. GPIO _TypeDef *(0x40020000UL)表示
   将常数强制转换为指向GPIO_TypeDef类型的结构体指针
3. 利用define取别名，用GPIOA作为结构体指针的别名

结构体指针加成员变量的形式访问硬件寄存器

```
GPIOA -> MODER;    GPIOA -> OTYPER
```

端口定义

GPIO端口的定义

```
1.  #define GPIOA    ((GPIO_TypeDef *) GPIOA_BASE)
2.  #define GPIOB    ((GPIO_TypeDef *) GPIOB_BASE)
3.  #define GPIOC    ((GPIO_TypeDef *) GPIOC_BASE)
4.  #define GPIOD    ((GPIO_TypeDef *) GPIOD_BASE)
5.  #define GPIOE    ((GPIO_TypeDef *) GPIOE_BASE)
6.  #define GPIOH    ((GPIO_TypeDef *) GPIOH_BASE)
```

1、GPIOx_BASE表示该端口寄存器组的起始地址
2、利用结构体指针+成员变量的形式访问寄存器，如 GPIOC -> MODER

芯片头文件

芯片生产厂商提供的芯片头文件

① 为了便于用户使用，ST公司将STM32微控制器片内所有外设的寄存器都采用上述的方法进行了定义，以 .h 文件的形式提供给用户；

② 用户在使用时，只需包含该头文件，就可以通过外设的结构体指针访问外设的相关寄存器；

③ 不同型号微控制器的头文件各不相同。以STM32F411系列微控制器为例，对应的.h文件为stm32f411xe.h；

```c
#include "stm32f411xe.h"          // 头文件中包含所有外设寄存器的定义
int main()
{
  unsigned  int delay = 1000000; // 延时变量
  RCC->AHB1ENR |= 1<<0;          // 开启端口 GPIOA 的时钟
  GPIOA->MODER &= ~(3<<(5*2);    // 清除模式寄存器的 bit11:bit10 为 00
  GPIOA->MODER |= 1<<(5*2);      // 设置 bit11:bit10 = 01，PA5 为推挽输出类型
  while(1)
  {
    GPIOA->BSRR |= 1<<5;         // 设置 bit5 为 1，PA5 输出高电平，开启 LD2
    delay = 1000000;
    while(delay--);              // 延时
    GPIOA->BSRR |= 1<<(5+16);    // 设置 bit21 为 1，PA5 输出低电平，关闭 LD2
    delay = 1000000;
    while(delay--);              // 延时
  }
}
```

利用结构体指针的形式修改演示例程

本章结束