



Computer Culture Advanced

打开计算机硬件的盲盒

揭开数字电子计算机的神秘面纱

二进制加法器实验小结

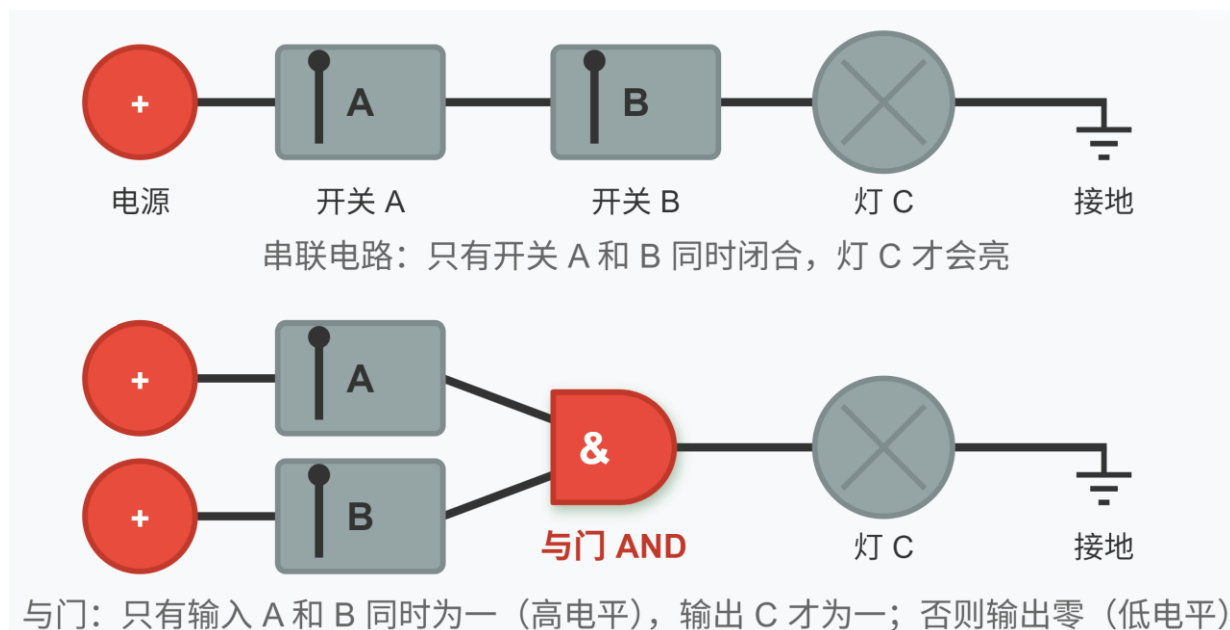
开关→开关电路→逻辑门→逻辑门组合→加法运算

通过基础逻辑门完成知识迁移：从中学串联电路，抽象到数字逻辑【与门】
把电路实物现象，抽象成**符号变换规则**

本质：**硬件连线拓扑**定义了**固定的变换规则 f**

$$y = f(x)$$

把物理状态（高低电平）映射成 0/1 符号，输出体现了按既定规则的符号变换



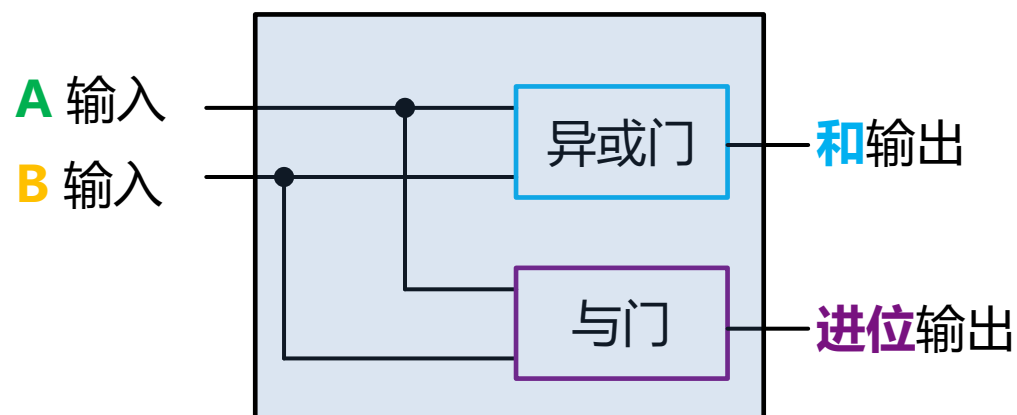
电路连线拓扑 = 符号变换规则



逻辑门组合：把二进制加法规则转化为硬件变换电路

进位计数制二进制形式的运算

$$\begin{array}{r} 0 \\ + 0 \\ \hline 00 \end{array} \quad \begin{array}{r} 0 \\ + 1 \\ \hline 01 \end{array} \quad \begin{array}{r} 1 \\ + 0 \\ \hline 01 \end{array} \quad \begin{array}{r} 1 \\ + 1 \\ \hline 10 \end{array}$$



一位半加器

二进制算术规则可以
固化为数字逻辑电路

和

进位

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

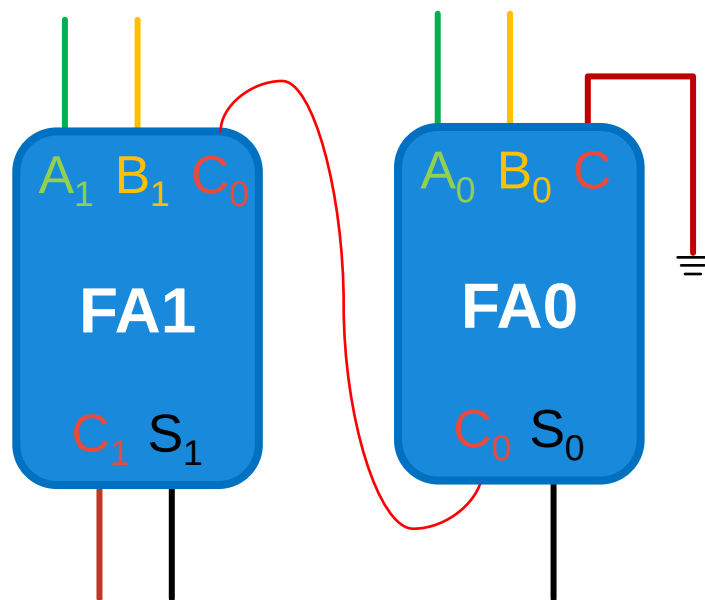
A	B	CO
0	0	0
0	1	0
1	0	0
1	1	1

二进制加法拆解到每一位，本质上是一连串的真假条件判断

多位全加器级联：二进制编码数值送入硬件完成运算

$$A = (3)_{10} = (11)_2 \quad B = (2)_{10} = (10)_2$$

求 $A + B = ?$

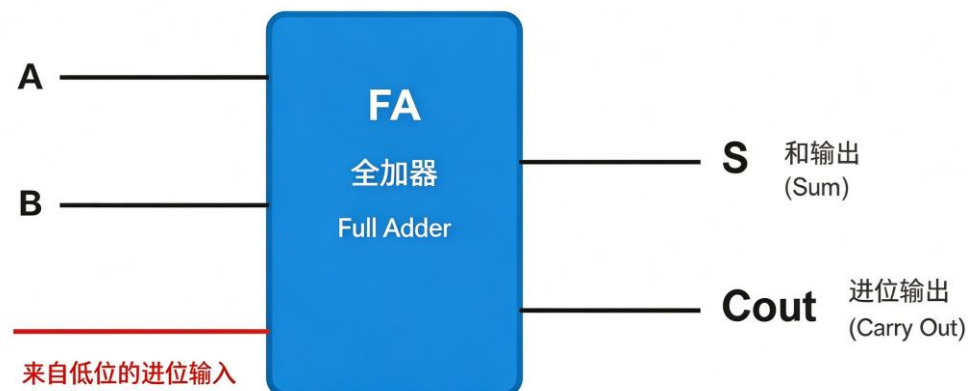


两位全加器级联实现两位二进制数加法

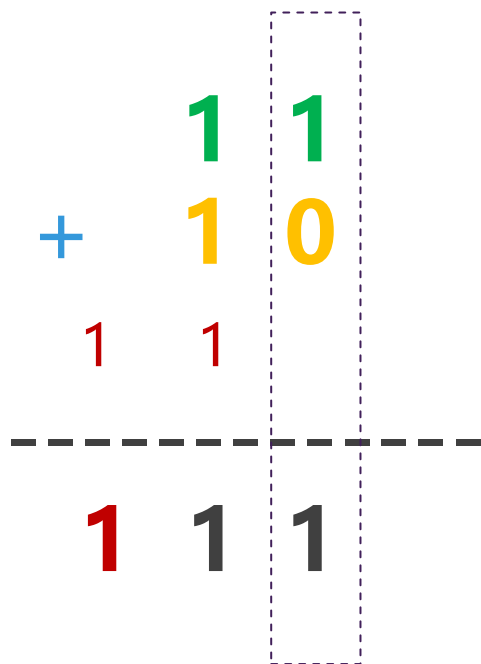
$$\begin{array}{r} 11 \\ + 10 \\ \hline 111 \end{array}$$

• 硬件只有高低电平，电平本身没有数值含义

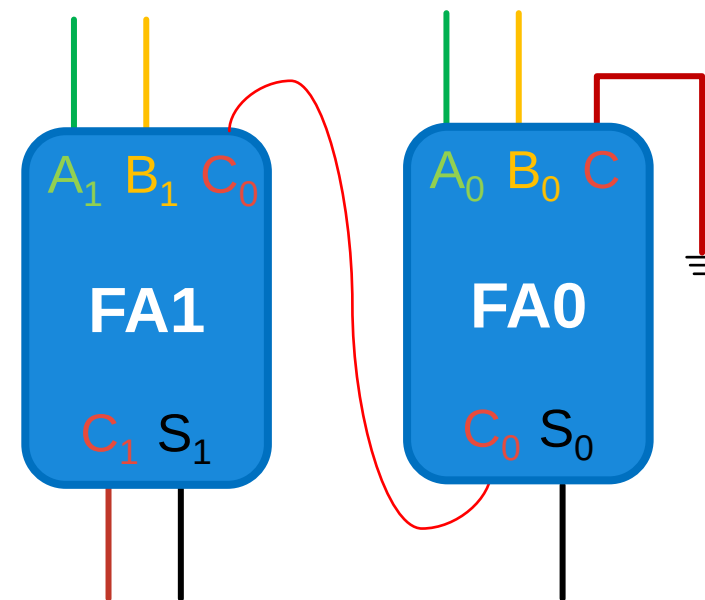
• 编码：人为约定电平组合，用来表达带有大小含义的数值



比特：二进制数位在硬件上的物理载体



- **比特 (bit)**：二进制数位在计算机硬件上对应的物理载体，仅有 **0/1** 两种状态
- **一组比特构成位宽**，代表 **ALU** 一次可并行处理多少个比特。**64位**处理器，即 **ALU** 单次并行处理 **64 bit**



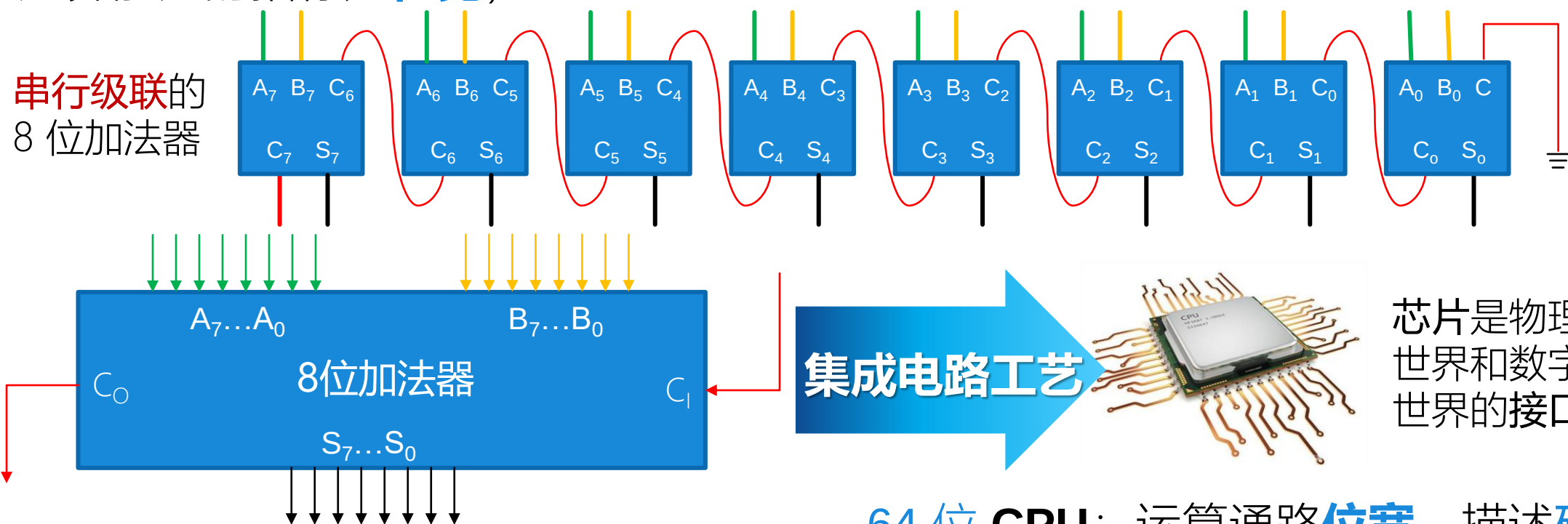
从**数位**到**比特**bit (位)

如何计算 $2 + 4 =$



比特在不同场景的状态和用途（运算硬件）

运算硬件： 比特表现为瞬时高低电平，用于实时完成符号变换（衡量硬件并行运算能力的指标是**位宽**）



芯片把逻辑门电路全部**封装**在一个器件内部

64 位 CPU： 运算通路**位宽**，描述**硬件并行运算能力**，一次能算多大

比特在不同场景的状态和用途（存储硬件）

存储硬件： 比特是可维持的物理状态，用来保存比特序列

只要能稳定维持两种状态的**电路**或**介质**，都可以用来存储比特

SRAM (**Static** Random Access Memory - **静态内存，CPU高速缓存**)

用双稳态触发器电路保存 1 比特。通电时状态稳定，不需要反复刷新。

电路复杂、晶体管多，速度快，成本高。

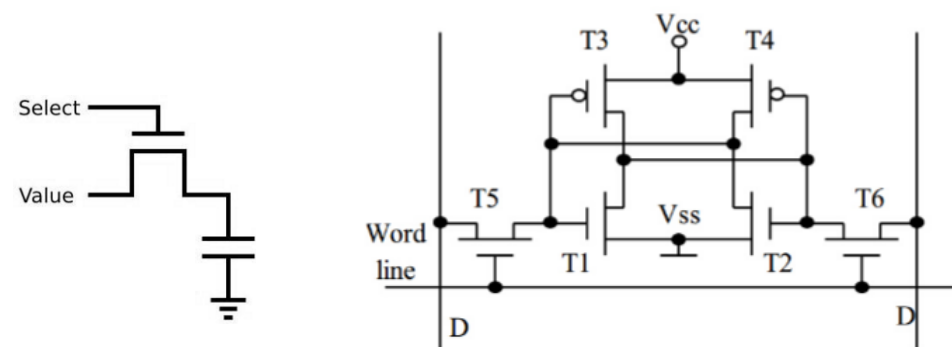
DRAM (**Dynamic** Random Access Memory - **动态内存，电脑主内存**)

用一个电容来记录 1 比特，电容每隔 10~100

毫秒电荷就会流失，需要**周期性地刷新维持**。

单元结构简单，密度高、成本低。

内存是断电丢失数据，属于**易失性存储**

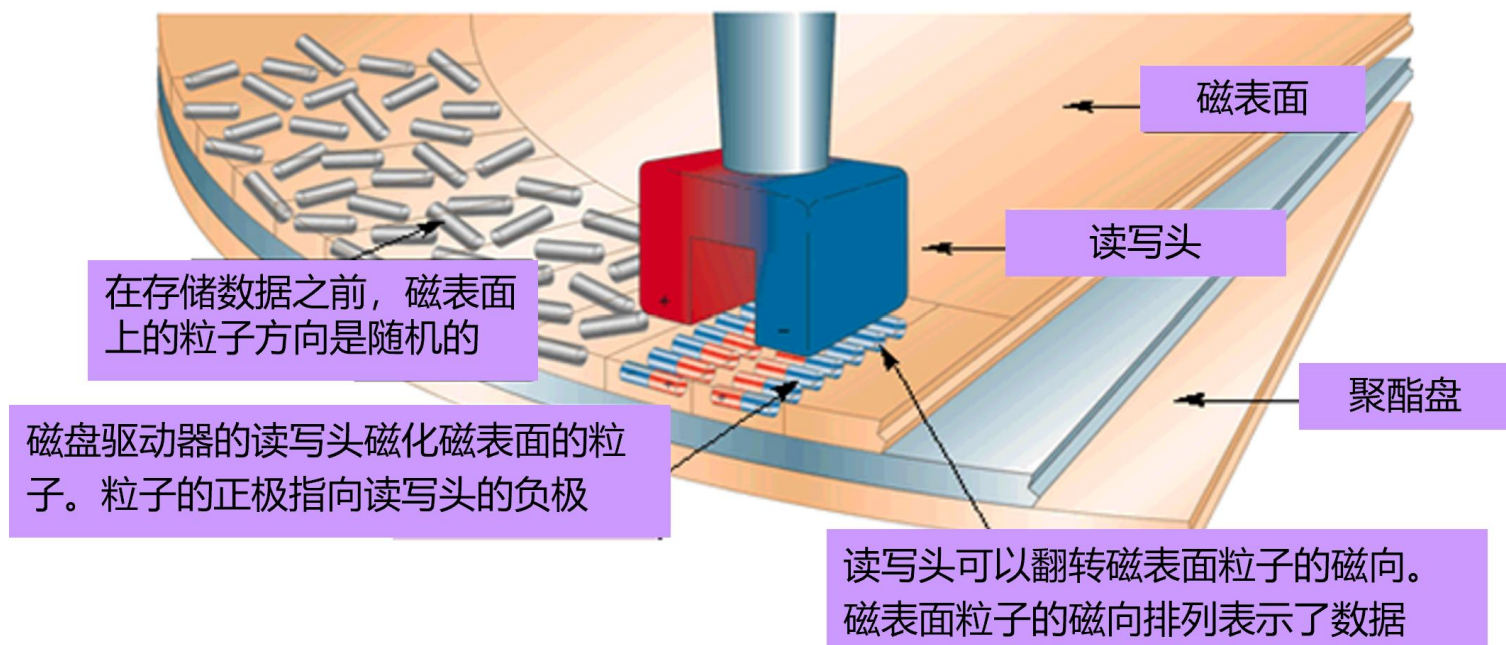


DRAM (1bit)

SRAM (1bit)

比特在不同场景的状态和用途（存储硬件）

存储硬件：比特是可维持的物理状态，用来保存比特序列
只要能稳定表示两种状态的**电路**或**介质**，都可以作为存储硬件



- **存储容量常用单位：**
KB、MB、GB、TB、PB（以字节 Byte 为基础）
- 硬盘 **1TB**表示**存储介质的总容量**，可以存放多少字节的比特序列（**保存信息能力**）

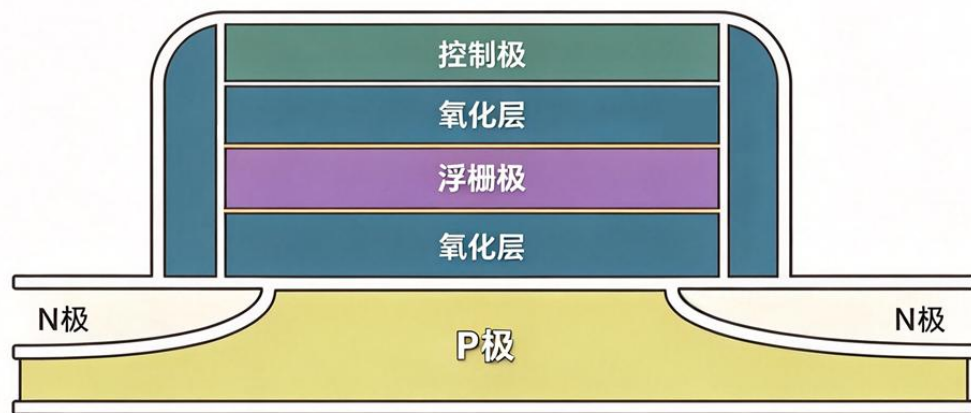
存储介质示例（磁盘：靠磁粒子磁化方向记录比特，属于**非易失存储**）



比特在不同场景的状态和用途（存储硬件）

存储硬件： 比特是可维持的物理状态，用来保存比特序列
只要能稳定表示两种状态的**电路**或**介质**，都可以作为存储硬件

闪存（NAND-Flash）：一种半导体存储芯片，依靠浮栅晶体管俘获电子记录比特，属于**非易失存储**，闪存的产品形态很丰富。



一个浮栅单元，利用有无电子区分两种状态，记录 1 比特

固态硬盘 SSD是以闪存芯片为存储颗粒，配上读写主控电路，组装而成的完整存储产品。没有机械结构，存取速度比机械式硬盘快。



机械硬盘

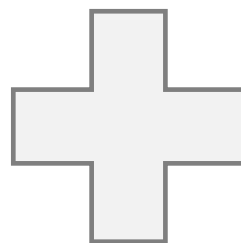


$$y = f(x)$$

计算模型的两类基础动作：

- ❶ 条件判断（布尔逻辑）
- ❷ 符号变换（可拆解为布尔逻辑的组合）

物理基座（数字电子）



信息翻译层
编码与解码



信息加工处理机器



Computer Culture Advanced

信息表示与处理

数字化：把各类信息转化为计算机能够识别和处理的形式

从硬件实验到上层编码

信息与数字化

什么是**信息**？

香农：**信息用于消除不确定性。**

直观理解：信息建立在**可感知、可区分的状态差异**之上，本源来自物理世界。

两类信息：

- ① **感知得来**的物理信号（声音、图像、压力等连续物理量）
- ② 人类**抽象创造**的符号系统（数字、文字、几何图形等）

计算模型

信息翻译层

$$y = f(x) + \text{编码与解码}$$



信息加工处理机器

所有信息进入计算机前，都必须**数字化**：

- ① **感知型数字化**（对连续物理信号，经过**采样、量化**，再**编码**成 0/1 比特串）
- ② **逻辑型数字化**（对人类抽象符号，直接通过**编码**映射为 0/1 比特串）



本次课的主题：逻辑型数字化（编码）

逻辑型数字化，核心动作就是 **编码**：把人类抽象符号映射成 0/1 比特串

人为定义编码规则 → **编码** 将符号映射成 0/1 比特串

→ 硬件 f 完成符号变换 → **解码** 还原信息语义

$$y = f(x)$$

主要介绍三种类型的**信息**编码

- **数值编码**（补码 / 浮点数）→ **数量**信息的人为映射
- **文本编码**（ASCII / GB2312 / UTF-8）→ **文字符号**信息的人为映射
- **矢量图形编码**（SVG）→ **几何图形**信息的人为映射

带着问题听课：为什么不同类型的信息要采用不同的编码方案



数值数据编码：基于进位规则的唯一映射

二进制进位计数制

- 基数固定为 **2**（二进制），每位只有 0 或 1 两个符号，符合**逢二进一**原则
- 位置有权值，从右向左，第 i 位权重 = 2^i ，最低位 $i = 0$

$$(1101.11)_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$$

$$\begin{array}{cccc} (1 & 3 & . & 75)_{10} \\ | & | & | & | \\ 10^1 & 10^0 & 10^{-1} & 10^{-2} \end{array}$$

$$(13.75)_{10}$$

=

$$(1101.11)_2$$

$$\begin{array}{ccccccc} & & & & 2^{-1} & 2^{-2} & \\ & & & & | & | & \\ (1 & 1 & 0 & 1 & . & 11) &_2 \\ | & | & | & | & | & | & \\ 2^3 & 2^2 & 2^1 & 2^0 & & & \end{array}$$





减法运算 $8-4 = ?$

能否复用加法电路?

加法电路运算实现 $8+(-4) = ?$

符号位怎么编码进二进制数中?



朴素符号位编码方案



采用**符号-真值**法的**编码**

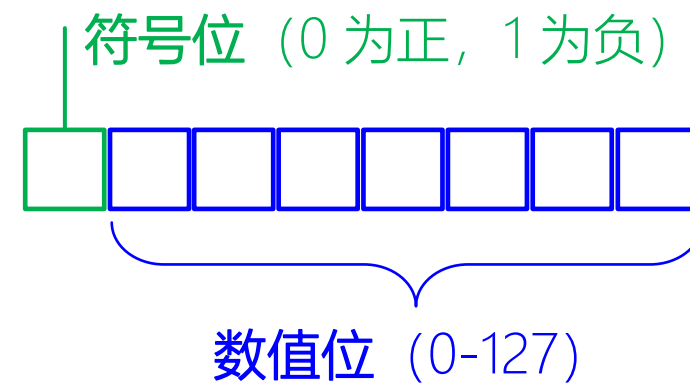
比如，用 8 位二进制表示一个整数 ($-127 \sim +127$)

计算一下： $+5 + -5$ 

$+5$ 表示为 00000101
(最高位 0 为**正**，后 7 位是 5 的**二进制真值**)

-5 表示为 10000101
(最高位 1 为**负**，后 7 位是 5 的**二进制真值**)

- 加法逻辑混乱
- 存在两个 0
- 减法需要单独设计电路



$$\begin{array}{r} 00000101 \\ + 10000101 \\ \hline 10001010 \end{array}$$



朴素符号位编码符合人类直觉，但加法器直接计算会错



加法硬件复用引出补码规则



补码规则的底层逻辑（模运算）

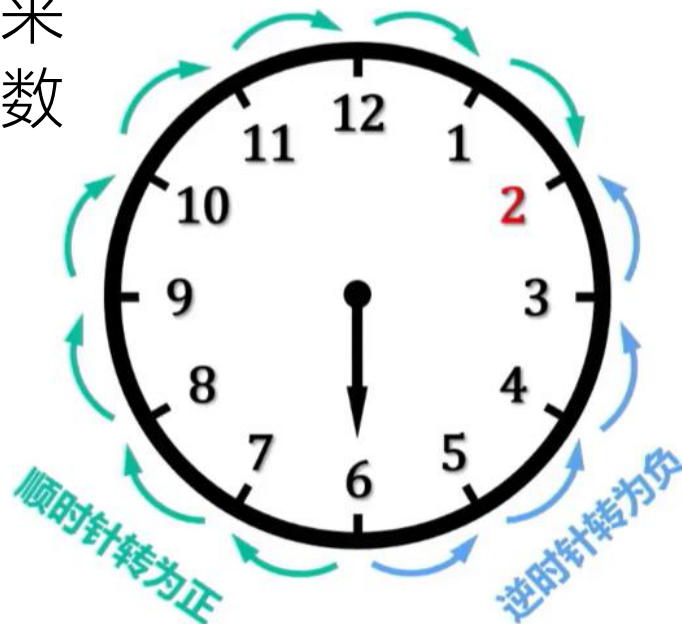
模是一个数值计量系统的计量范围 (mod)。只要确定了**模**，就可找到一个**与负数等价的正数**来代替此负数，该正数就是负数的**补数**，超出计量范围的数都应该自动舍去模数。

指针从**6**拨到**2**的两种方式：

- 逆时针拨动**4**下（**减法**）
- 顺时针拨动**8**下（**加法**）

-4和**8**是一对**补数**

$$6 - 4 = 6 + (12 - 4) = 6 + 8$$



钟表模运算

举例

以 **8** 位为例，**模** = **256**

$$5 - 5 = 5 + (256 - 5) = 256$$

自动丢掉最高位（**溢出**），
超出范围舍去，剩下 **0**

00000101 （**+5**的补码）

+ 11111011 （**-5**的补码）

100000000

8位模空间运算时，**超出范围的1被丢弃（溢出）**，
结果为 00000000



已知：清华1911年建校，今年2026年

要求：不能直接做减法 2026-1911

约定：年份有效表示0~9999，用补码规则，
减法转换为加法

如何用补码规则，只用加法运算，
计算出今年清华建校多少周年？





$$N = m \times 10^e$$

类比十进制的科学计数法

二进制的**浮点数**编码方案

$$N = (-1)^s \times m \times 2^e$$

以最常用的 32 位 **单精度浮点数** (IEEE754) 为例，这套人为定义的编码，把 32 位划分成三段含义：

符号位(s)	指数位(e)	尾数位(m)
1 位	8 位	23 位

如果让你设计一个能表示**宇宙年龄 (138亿年)**
和**普朗克时间 10^{-44} 的数**，你会**怎么做**？



从补码（定点整数）方案理解编码



补码是固定位宽模空间下的工程编码方案：在固定位宽的模运算体系中，将负数映射成一个等效正数。使计算机只用加法电路，就能完成带符号整数的加减运算。

- **硬件视角**：加法器只认高低电平（0 / 1），只会做加法
- **编码的本质**：人为定义的映射规则
- **编码的价值**：把符号信息嵌入比特序列，改造数字的语义
- **编码的选择**：编码方案不是唯一的，要适配硬件工程目标



编码（真值→补码）：

- **正数**：补码与原码（真值）相同
- **负数**：符号位固定为 1，数值位按位取反后加 1

解码（补码→真值）：

- **正数**：直接读取数值位，即为真值
- **负数**：对负数的补码整体按位取反后加 1，得到对应的原码，再根据符号位判断正负，读出真值

编码↔解码：信息处理的一对配套动作



Computer Culture Advanced

编码 是把信息翻译成机器能处理的符号（信息在机器内部的表示）

解码 是把机器内部表示的符号还原成人能理解的信息

示例

- **加法器硬件实验：**

十进制数字（人看懂）→**编码**成二进制 0/1 输入加法器

加法器输出 0/1 →**数码管解码**，变回我们可读的十进制数字

- **补码运算：**

带正负的整数真值 →**编码**转为补码送入加法器

运算得到补码结果 →**解码**还原成带正负的真值

- **浮点数编码：**

极大 / 极小实数 →**编码**为符号位 + 指数位 + 尾数位的比特序列

读取比特序列 →**解码**还原为实数

- **英文字符编码（ASCII）：**

英文字符 →**编码**成 0/1 表示

读取 0/1 →**解码**，配合字体文件，渲染出屏幕上的文字

- **SVG 矢量图形**

绘图指令（文字描述画圆）→**编码**为文本文件

应用程序读取 0/1 →**解码**，配合渲染器绘出图形

编码规则由人定义。
编码和解码必须成对配套，使用同一套规则，否则出现乱码、解析错误

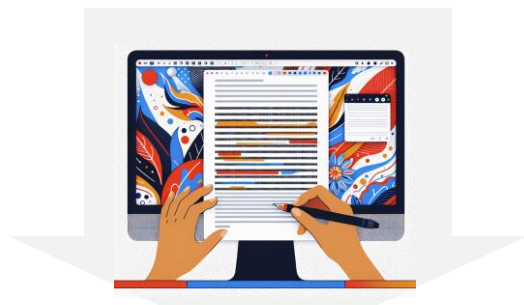




计算模型

信息翻译层

$$y = f(x) + \text{编码与解码}$$



信息加工处理机器

文字、图形这类
非数值信息，**如**
何利用这套模型
进行处理？



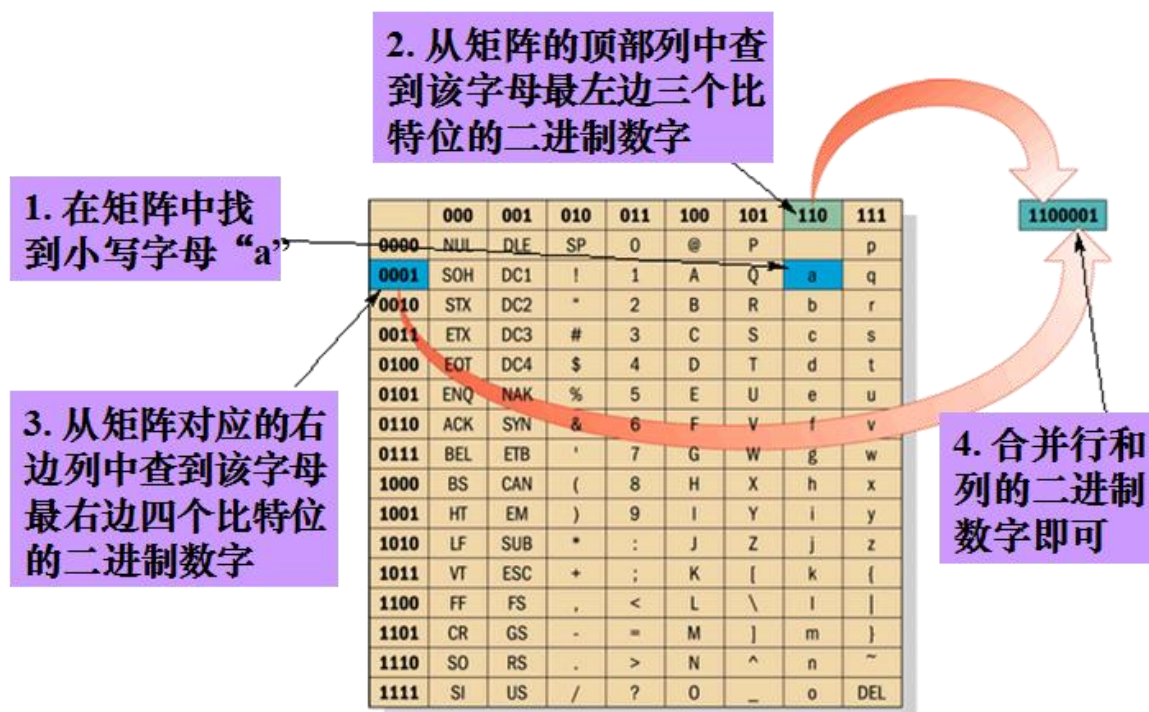
非数值数据编码：从数字到文本

编码的本质：建立唯一无歧义的映射关系

字符集：ASCII 一共收录 128 个符号

编码规则：人为约定，把表格排列成 16 行 × 8 列，用 7 比特来记录每个字符在表格里的行列位置。这个位置编号，直接作为字符对应的**唯一的二进制数串**

存储约定：7 比特的编码，在计算机里占用 1 字节存放，最高位补 0



ASCII编码1963年发布



编码的本质：建立唯一无歧义的映射关系



Computer Culture Advanced

逻辑型数字化的本质是符号映射

映射步骤：字符集 → 编码空间 → 编码

① 确定字符集

GB2312国标码 (1980年)

汉字：6763个（一级汉字3755个按拼音排序，二级汉字3008个按部首排序）。

非汉字：682个（拉丁字母、日文假名、俄文字母、标点符号等）。

② 在编码空间里给每个一个唯一编号

区位码 (94**区** × 94**位**)

帧 症 郑 证 芝 枝 支 吱 蚰 知 肢 脂 汁 之 织
职 直 植 殖 执 值 侄 址 指 止 趾 只 旨 纸 志 摯
擲 至 致 置 帜 峙 制 智 秩 稚 质 炙 痔 滯 治 室
中 盅 忠 钟 衷 终 种 肿 重 仲 众 舟 周 州 洲 迺
粥 轴 肘 帚 咒 皱 宙 昼 骤 珠 株 蛛 朱 猪 诸 诛
逐 竹 烛 煮 拄 瞩 嘱 主 著 柱 助 蛀 贮 铸 筑

中 $\xrightarrow[\text{GB2312编码}]{\text{区位码}}$ **5448** (3630_{10})
00110110 00110000

③ 按照规则把这个位置编号转换成二进制

编码的本质：建立唯一无歧义的映射关系

映射步骤：选定**字符集** → 分配字符编号（**编码空间**） → 二进制**编码**

①确定**字符集**

小字符集 (ASCII)

大字符集扩展 (GB18030)

跨语言统一集合 (Unicode)

②根据应用场景 配套**字符编码规则**

同一套 Unicode 字符集，
可以选用 UTF-8、UTF-16
等不同编码。

Unicode 国际字符集把世界上所有语言的文字、符号全部收录进同一本总表，给每个字符分配一个全球唯一编号，彻底解决各国各搞一套编码表的问题

UTF(Unicode Transformation Format)是 Unicode 的存储实现规则，后缀数字代表编码的基础单元大小。
例如：**UTF-8** 是**变长编码**，根据字符频率用1~4个字节表示一个Unicode字符（英文字符兼容 ASCII 占1字节，常用中文字符占 3 字节），兼顾兼容性与存储效率



非数值数据编码：从文本到图形

编码对象： **图形**的基础形状种类是有限集合（圆、矩形...），**但**形状摆放位置、大小、颜色，可以有无穷多种组合

编码方案： **图元 + 参数**（绘图说明的结构化描述）

图元指定是什么基础形状，参数指定这个形状的位置、大小、颜色等属性

矢量图形拥有一套几何生成规则，但图元本身不能单独确定画面，必须配套具体参数，才能唯一确定图形长什么样

图形由坐标、几何方程描述，不是预先固定像素点阵，因此称为矢量图形

```
<svg width="200" height="200" xmlns="http://www.w3.org/2000/svg">
  <!-- 画一个红色填充、黑色轮廓的圆 -->
  <circle cx="100" cy="100" r="80" fill="red" stroke="black" stroke-width="2"/>
</svg>
```

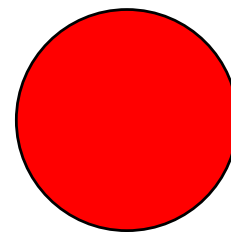
SVG矢量图形编码示例

矢量逻辑编码（业务语义）+ UTF8编码（文件存储的字符编码）



Computer Culture Advanced

图形如何编码



SVG文件：绘图描述文本（图元 + 参数）→ UTF-8文本编码 → 二进制0/1

```
<svg width="200" height="200" xmlns="http://www.w3.org/2000/svg">  
  <!-- 画一个红色填充、黑色轮廓的圆 -->  
  <circle cx="100" cy="100" r="80" fill="red" stroke="black" stroke-width="2"/>  
</svg>
```

记事本打开
的SVG文件

• 指令: `circle` (画圆) 生成画面（圆）的结构化描述

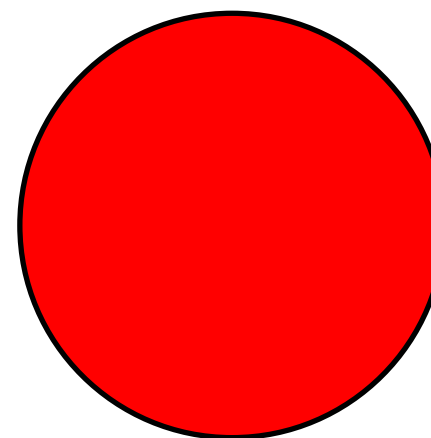
• 参数:

- `cx="100"`: 圆心 X 坐标在 100
- `cy="100"`: 圆心 Y 坐标在 100
- `r="80"`: 半径为 80

图元基本参数

- `fill="red"`: 内部填充红色
- `stroke="black"`: 轮廓为黑色
- `stroke-width="2"`: 轮廓宽度为 2

样式属性参数



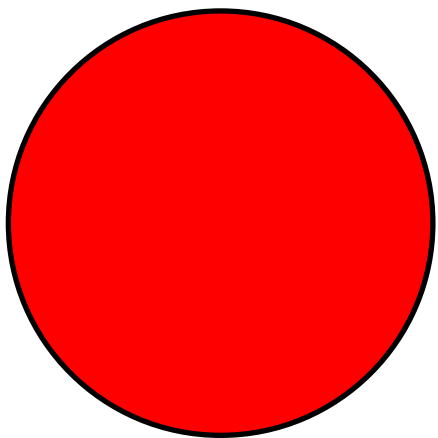
PowerPoint幻灯片中插入该SVG文件
内置的渲染器根据绘制说明画出图形



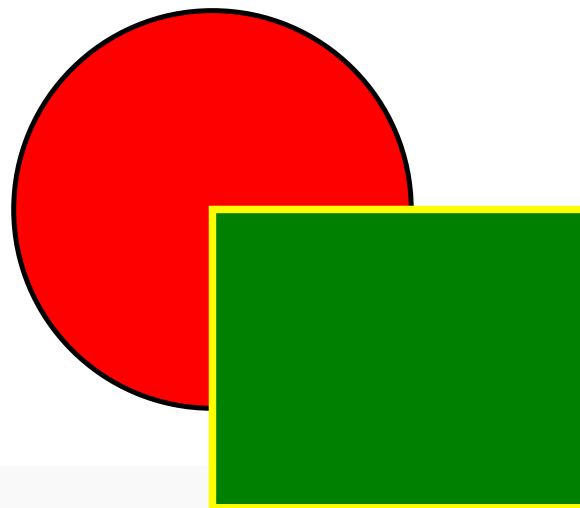
矢量图形编辑练习

矢量图形： 编码包含几何生成规则和参数，渲染器实时计算坐标

矢量图形编辑： 不修改像素，只增加几何运算规则、变换参数，修改绘图描述文本



任务： 在原有绘图描述文本里新增一个矩形的几何规则，渲染器按先后顺序，叠加绘制圆与矩形，得到叠加后的画面



```
<svg width="300" height="300" xmlns="http://www.w3.org/2000/svg">
```

```
<!-- 画一个红色填充、黑色轮廓的圆 -->
```

```
<circle cx="100" cy="100" r="80" fill="red" stroke="black" stroke-width="2"/>
```

```
<!-- 再画一个绿色填充、黄色轮廓的矩形 -->
```

```
<rect x="100" y="100" width="150" height="120" fill="green" stroke="#FFFF00" stroke-width="3"/>
```

```
</svg>
```

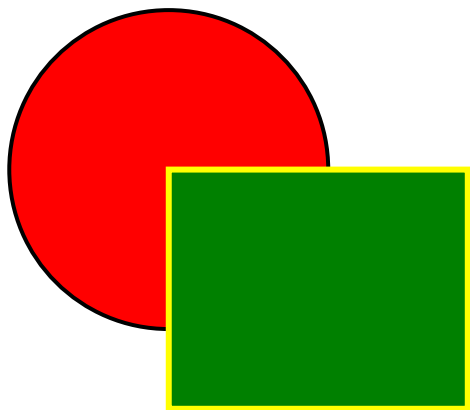




矢量图形编辑练习

矢量图形： 编码包含几何生成规则和参数，渲染器实时计算坐标

矢量图形编辑： 不修改像素，只增加几何运算规则、变换参数，修改绘图描述文本



任务： 修改SVG绘图描述，对圆和矩形执行布尔差集运算（圆减去矩形），再将运算得到的新图形整体逆时针旋转 45°，得到右侧缺口扇形图形



```
<svg width="300" height="300" xmlns="http://www.w3.org/2000/svg">  
  <!-- 画一个红色填充、黑色轮廓的圆 -->  
  <circle cx="100" cy="100" r="80" fill="red" stroke="black" stroke-width="2"/>  
  <!-- 再画一个绿色填充、黄色轮廓的矩形 -->  
  <rect x="100" y="100" width="150" height="120" fill="green" stroke="#FFFF00" stroke-width="3"/>  
</svg>
```





矢量图形编辑练习

```
<svg width="300" height="300" xmlns="http://www.w3.org/2000/svg">
```

```
<!-- defs: 新增 <defs> 定义遮罩规则, 实现圆形减去矩形 -->
```

```
<defs>
```

```
<!-- 定义遮罩mask -->
```

```
<mask id="cutMask">
```

```
<!-- 白色矩形: 全部保留区域 -->
```

```
<rect x="0" y="0" width="300" height="300" fill="white"/>
```

```
<!-- 黑色矩形: 这个区域透明, 用来挖掉圆形的一部分 -->
```

```
<rect x="100" y="100" width="150" height="120" fill="black"/>
```

```
</mask>
```

```
</defs>
```

```
<!-- 红色圆形, 应用遮罩: 黑色矩形区域被挖去, 实现圆减去矩形 -->
```

```
<circle cx="100" cy="100" r="80" fill="red" stroke="black" stroke-width="2" mask="url(#cutMask)"/>
```

```
</svg>
```





矢量图形编辑练习

```
<svg width="300" height="300" xmlns="http://www.w3.org/2000/svg">
```

```
<!-- defs: 新增 <defs> 定义遮罩规则, 实现圆形减去矩形 -->
```

```
<defs>
```

```
<!-- 定义遮罩mask -->
```

```
<mask id="cutMask">
```

```
<!-- 白色矩形: 全部保留区域 -->
```

```
<rect x="0" y="0" width="300" height="300" fill="white"/>
```

```
<!-- 黑色矩形: 这个区域透明, 用来挖掉圆形的一部分 -->
```

```
<rect x="100" y="100" width="150" height="120" fill="black"/>
```

```
</mask>
```

```
</defs>
```

```
<!-- g分组: 把图形打包成一个整体, 添加变换规则: 整体逆时针旋转45°, 旋转中心(100,100) -->
```

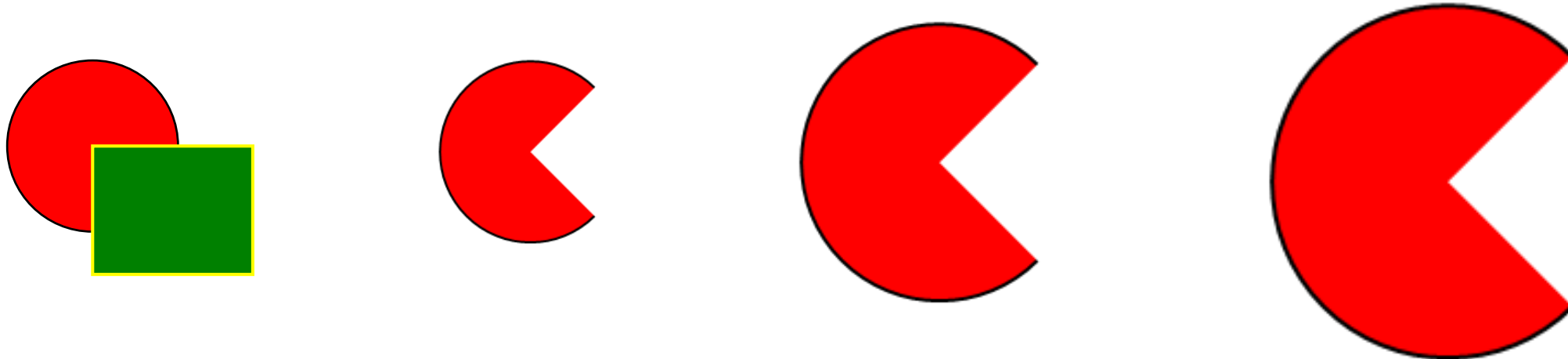
```
<g transform="rotate(-45,100,100)">
```

```
<circle cx="100" cy="100" r="80" fill="red" stroke="black" stroke-width="2" mask="url(#cutMask)"/>
```

```
</g>
```

```
</svg>
```





矢量图形的编辑：

- 全程修改文本描述（指令 + 参数）

矢量图形的显示：

- 渲染器读取全部几何规则，实时计算图形坐标并重新绘制
- 放大/缩小只是在不同比例上重新绘制，图形本身不会失真



逻辑型数字化（编码）小结

逻辑型数字化，核心动作就是 **编码**：把人类抽象符号映射成 0/1 比特串

编码 抽象符号 → 0/1 比特串（交给硬件处理）

解码 0/1 比特串 → 还原成人可理解的符号语义

$$y = f(x)$$

编码 是工程权衡，需要适配：硬件运算特性、表达范围、存储开销、传输效率、解码复杂度，不存在万能编码。

- **数值**编码（**补码**方案）：为复用加法硬件
- **数值**编码（**浮点数**方案）：用来表达极大 / 极小数值，需要专门设计运算电路
- **文本**编码（**Unicode/UTF-8**）：解决多国文字统一表示，兼顾存储大小、兼容性
- **矢量**图形（**SVG**方案）：用图元（几何规则）+ 参数（尺寸位置等）描述几何概念，缩放不失真

为什么不同信息要设计不同编码方案

