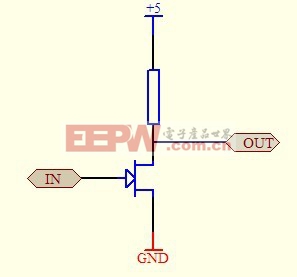
内存中的半导体：内存其实就是密密麻麻的许许多多的半导体的集合，半导体本质还是电容，电容通电之后会留保留电荷，我们将存有电荷的电容计为1，不存有电荷的电容计为0，但是电容中的电荷通常都会自行消失，所以，计算机会周期性地对留有电荷的电容充满电，则，电荷量超过某个阈值计为1，小于该阈值则计为0。

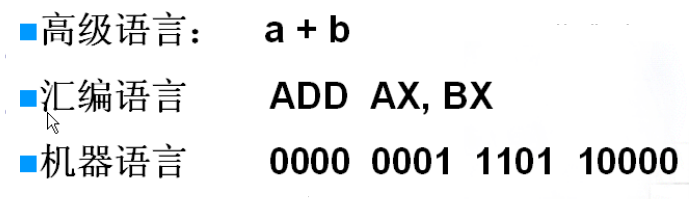
CPU往内存插三根线：CPU通过地址总线、控制总线、数据总线直接作用于内存条，大致过程是，CPU通过地址总线找到内存中的数据单元地址，控制总线控制这一时刻该数据单元的状态是只读还是只写（电平状态的开关），控制状态完成之后通过数据总线对数据进行读或者写。CPU要能连通内存条中全部的内存地址编号（16位）就需要能和内存条有足够多的信号开关连线，比如32个地址总线就能表示2的32次方个内存地址，也就能在不冲突的情况下定位和寻找2的32次方个内存地址，而这个数字相当于4G，这也是为什么32位操作系统最大支持4G内存的原因。

上拉电阻：如下为上拉电阻示意图，IN箭头指示的连接线其实连接着内存中的两个针脚，这两个内存针脚对应于内存中的某个二极晶体管，该晶体管为通路状态则代表数据1，也相当于IN处为通路，晶体管为绝缘状态则代表数据0，也相当于IN处为断路，IN处的通路与断路，相当于+5V到地面的电势差之间是否接入了另外一个晶体管的等效电阻，也因此该等效电阻会与上拉电阻分压，会导致OUT处输出不同的电压，该电压信号超过2.4V认为是高电平，低于则认为是低电平，OUT其实就是CPU针脚，通过限压电容，该处的高电压会激发电磁继电器形成另外一个通路，低电压则不会，这样的通路与断路最终都要通过CPU内部的电路逻辑门，到达终点构成另外的通路与断路，形成结论，逻辑门能组成半加法运算器和全加法运算器，也就能得出二进制的累加运算，多次的二进制运算能表示十进制运算，如此，晶体管的通路与断路逐层递进，最终对应于CPU输出口的通路与断路，也形成了结论逻辑。



从高级语言到寄存器到逻辑电平：计算机上插入了一个全新的硬件设备，这个设备有很多很多的金属针脚，我们将它插入计算机的电路板，最终是将它的针脚与计算机芯片cpu的许多密密麻麻的针脚间接地连接了起来，这就形成了一条条通路，也就相当于构成了许许多多的电磁继电器开关——cpu针脚间的微弱电流变化能像电磁开关一样触发该设备内部的电路连通与否，与此同时还不会使得两方电流有任何相互感染。以上，得出结论：cpu芯片针脚间的电流变化能控制硬件内部的电流开关，从而控制硬件。每一个硬件在插入电脑之后都要安装驱动，驱动所做的就是测量硬件，告诉计算机硬件有多少个针脚可以接受多少种信号，需要什么样的数据。计算机根据该驱动提供的信息在自己的内存条中取出一大片连续的内存，这一片内存包含了很多的内存单元，计算机这样做的目的是想把该片内存地址映射成某硬件的金属针脚，这片内存区域被称为硬件在计算机中的内存寄存器。如果我们的C程序获得了计算机提供的该寄存器内存的操作权限就可以利用指针操作该内存的数据，数据一旦变化，CPU便开始批量消费内存中的数据，当然这些数据都是二进制的，我们人类理解的1和0，就是计算机理解的电平的高与低，就是电路的开与闭。以上，说明了C语言改变的其实是硬件在内存中的映射，CPU消费了这些映射变动而将其转变为电压信号的变动。

机器语言汇编语言和高级语言：程序设计语言的发展历程——计算机本身只能识别0和1的二进制数据串所代表的晶体管的开与闭，我们的特定输入能使它做出特定的动作，这就是机器语言，但是无意义的二进制串对人类来说不能理解和记忆，后来人们将二进制数据串简单直白地翻译成有意义的单词，这样大大增强了编程的可行性，这类粗暴有效的直接对应机器语言的语言就是汇编语言。再后来，人们对汇编语言进行了一些改造，增添了其艺术性，使得语言具备了一定的结构和语法以完成更多的功能，这就是结构化语言，早期的结构化语言主要包括Fortran、Basic、C和Pascal等，其中以C最为经典(C的起源是1960年的Algol60语言和1970年的BCPL70语言，BCPL简称B语言，就是它写出了世界上第一个操作系统unix，C在B语言的基础上继续改进，发布的时候取名为C，C语言写出了目前所有操作系统的基础内核，包括linux、windows、macOS，C的生命力也最为强大)。时间走到1980年，人们发现建立在汇编语言基础之上的基础结构化语言来完成稍微大型项目的时候总是会出现各种各样的麻烦问题，因为它是纯粹面向过程的，于是在基础结构化语言的基础上进行了更高层次的艺术化和结构化加工，开发了面向对象的语言，包括早期的algo以及smalltalk，以及后来影响力最大的C++，但是C++结构异常复杂（C++完全兼容C），于是出现了简化版的C++，即java和C#（两者几乎是雷同的）。直到目前，最流行的编程语言依然都是C、java和C#。



指针：地址是16位的内存单元的编号，地址就是指针，指针也就是地址。C语言能与底层互动，强大而又危险的根本原因就是指针。指针是能存放其他变量地址的变量，它本身是变量，而且特定类型的变量只能被该同类型指针变量指定。int i = 3; int \* p = &i; i==\*p; 含义:第一步，定义一个整型变量i其值为3，操作系统会在内存中寻找空闲区域并分配一个8位的空闲内存单元，并初始化其值，使用二进制00000011表示3；第二步，寻找另一片空闲内存单元，设置其类型为int \*，即为整型数据的指针类型，将该内存单元中的二进制数值设为i内存单元的地址，注意，&的意义为取地址；第三步，\*除了用于修饰变量类型为指针类型外，作为操作符的\*号有另外一重含义，即，根据内存单元地址取值，所以能得出结果p=&i; 那么，i==\*p。从上可以看出，&与\*是动作完全相反的两个操作符，一个变量在内存中就是一个内存单元，该单元结构包含两部分内容:内存核心值和内存地址，而&是根据内存单元得出内存地址，\*是根据内存地址找到内存单元。这也就意味着，只要知道内存地址的16位标识符，利用\*号，我们就可以找到任意一个内存单元。试想，p值设为0x11234321，就可以读到地址为该值的内存单元的数据，而如果该内存单元并不属于当前程序，就会造成不确定性，所以:指针可以使得两个完全不相干的内存单元形成互为引用关系，也因此增加了程序的不确定性。指针就是病毒破坏、内存泄漏等严重问题的基本根源。试想，p、q两个指针都指向同一个内存单元，如果使用完成并未向系统告知释放内存(因为系统并不知道哪些程序什么时候不需要哪些内存所以无法自行free)就会导致内存被不断地占用和吞噬而逐渐地全部消失，这就是内存泄漏，而如果我们释放了p指向的内存，再释放q指向的内存，因为此时该内存已经不属于程序的合法操作范围，因而会导致程序错误。指针是强大高效而又难以驾驭的武器。所以高级编程语言，比如java干脆在底层自己完成了指针的管理，但这也同时降低了功能性。

静态内存与动态内存：C语言中，程序的默认执行都是将变量放入栈中，而这样的变量都是占用了静态内存，一旦程序结束栈中的变量，即静态变量全部都会被自动释放，这是毫无质疑的，但是，我们也可以通过malloc方法告知系统在某个特殊的地方为我们分配一个内存，该内存暂存，且不由系统自动释放，而由我们自己通过free方法手动释放，利用指针的特性，我们就可以一直保持对该特殊变量的引用，哪怕函数终止，也不会丢失指针与它之间的钩子关系，我们称此为动态内存，这样的动态内存可以长时间保持对一个甚至是基本类型变量的引用，静态内存和动态内存也就是java中所说的栈与堆中存放的数据，只不过java默认对象都是静态的，基本类型都是静态的。

从硬盘到内存区域划分：系统将程序数据从硬盘读取装载到内存中(因为内存的IO处理速度很快)并执行入口方法，开启程序，每一个程序启动后都会向系统申请从机器内存中划分一块内存自己独占，该应用程序还会将独占的内存进行科学合理地规划，但主要是按照处理方式的不同来进行功能性的划分。例如，java主要划分了5种区域的内存，寄存器区（cpu接受机器码并运算的地方）、本地方法区（java语言映射为机器码的地方）、方法区（类中方法区，它们全都作用于this，以及类中静态方法和静态属性，它们处于方法区中的静态方法区，属于类自身，并由所有实例共享，比如作为程序入口的main就是静态方法，处于方法区的静态方法区，它接受入口配置，public static void main(String[] arguments)，其中的arguments就是我们给程序传入的配置，比如java demo.class test1 test2，这样就将test1和test2作为字符串数组传入main方法，这就像是url中的hash或者bat中的调用参数一样）、栈区和堆区。几乎所有的汇编语言都有栈内存和堆内存的概念。栈内存通常处于高速RAM ，处理速度非常非常快，它用于处理局部变量：为了服务于嵌套的作用域间的关系，栈内存的特点是由低处往高处存放，即先进后出，后进先出，这样以来，程序非常方便快速地处理完最低级作用域（高栈）再一层层向高级作用域（低栈）执行，直到最高级作用域执行完毕。因为要保证快速执行，栈内存要求执行完后立即释放，这也就客观决定了栈内存存放的只能是基本类型的数据，因为基本类型长度固定，由程序体系自行分配和释放空间。相对而言，堆内存中存储的是结构复杂的对象，由程序的书写者分配和回收(或者是垃圾回收机制)，优先到达的对象将会被优先处理，类似于队列，先进先出，当我们将一个对象赋值给一个变量，变量持有的其实是该对象的引用，所以，可以这么认为：栈内存中存放的就是当前时刻全部的局部变量，变量如果是基本类型则直接随身携带，如果是引用类型，则持有引用对象的内存地址，那么，读取对象变量就是根据内存地址找到该对象在堆内存中的位置以读取某些属性并将属性值拿回到栈内存中作为基本类型进行运算。整个过程就是：处理器接受到机器码执行指令去方法区调用方法，并压入栈中，在栈中处理所有的变量作用，如果是基本类型则直接处理，如果是引用类型还要根据引用地址去堆中找到对象。调用方法如果是非静态则其中this指向当前对象，如果是静态则作用于类，执行完成之后由JVM翻译为机器码，并交给寄存器区等待处理。

java数据类型：java的基本数据类型有四类，关于数字的整数和浮点数，关于字符的字符型，关于真假的布尔型，而引用类型的数据就是各种各样的Object。数据或者数据的引用总是由变量携带，而变量应该被清晰地定义其携带或者持有引用的数据的类型。一个变量携带一个数据，但有很多时候我们需要一组数据，反复地创造变量是非常麻烦的，但是我们可以在堆内存的实体中存放多个数据，而数据就是解决这一问题的特定实体。在java中，数组就是创建在堆内存中的一种实体，是用来存储数量固定、类型相同的一种基本数据容器。新建一个固定长度为5的数组并将其赋值给一个变量，本质上是在堆内存中寻找一个空闲的内存，以该起始位置为起点开辟连续的大小相同的5片内存单元，并将起始位置的内存地址（引用）赋值给变量且将其推入栈内存中，当我们访问该数组的第3个元素的时候，即是以变量引用的内存地址为起点的大小固定的第3个内存单元中的数据，这就是数组的最大特点：内容自动索引。程序总是需要分块的，在符号学中，我们使用括号进行区块划分，可用的有小括号，大括号，方括号和尖括号，小括号是运算划分，大括号是作用域块划分，方括号是连续的数据块划分，java中就使用方括号来标识一段长度固定的数据集合，例:int[] s = new int[5];。由以上可以看出，java包含出现于栈内存中的4种基本数据类型以及堆内存中的实体，数组和通常意义上说的对象都是实体。数组和对象，连续的以及不连续的，它们的组合，能构造出各种各样的数据结构。

集合框架：java构建了更加强大灵活的批量存储容器——集合框架。集合框架的顶级类是包括单列集合的Collection以及双列集合的map。collection拥有基础的添加、下一个、便利器等方法。其一级子类包括List、Set。List是有序的可重复的可变长度的对象集合，它下辖的主要二级子类是ArrayList和LinkedList，ArrayList（替代了原有的Vector）是基于数组的，增加元素的原理是将原数组复制到新建数组上，所以增删慢，但是查询快。LinkedList是基于对象的，每一个对象都保存了下一个对象的内存地址，就像很多小朋友手拉手排列，增加对象也就是改变毗邻对象记录的引用关系，所以增删快，但是查询速度不如数组。Set是“无序”的不可重复的可变长度的对象集合，其下辖二级子类常用的有hashSet和treeSet，所谓hash码其实就是根据对象的特征计算出来的用于区分该对象与其他同类对象的标识。hashSet认为通过便利来查询数组速度很快但依然可以继续优化，通过计算元素的hash值来将元素存放到固定位置，下一次取出该元素则重新计算该元素hash码直接去指定位置取即可，因为相同元素总能计算得出相同的hash码，所以hashSet不能有重复的值（事实上，重复的元素是由对象类的hashCode和equals方法共同决定的，也因此hashSet的元素顺序是程序的自然排序法则决定的）。treeSet在hashSet的基础之上更近一步，它通过二叉树算法和compare方法对元素进行指定排序。ArrayList和LinkedList都可以通过数字索引查找元素，HashSet和TreeSet内部利用了hashCode更快速地查询元素，但它们都是单列集合，好像数据都是作为“单身汉”的形式聚集的。相对而言，map是双列集合，它其中存储的数据似乎都是以“一夫一妻”成对的形式存在的。它的底层结构也是hash表，不允许重复，键可以为一个对象，值也可以为一个对象，只不过键对象同时存储了指向值对象的引用。综上：集合框架的底层实现一共有三种：数组（查询快增删慢可重复）、链表（查询慢增删快可重复）、哈希表（查询快增删快不能重复）。

泛型：泛型是额外地对对象进行类型修饰和限定，用以将类型除错提前到编译之前及早发现。编译器按照泛型的指定类型对元素进行检查，编译之后的待运行class文件是没有泛型的，因为JVM是不识别泛型的，而只有编译器能识别。这就是泛型擦除，意思就是泛型用来声明的集合只在编译阶段被定义，在运行阶段被擦除。一个对象要操作另外一种类型总是无法避免类型强转以及潜在冲突的问题，这个时候就需要使用泛型来指定即将被操作的类型，这样就能在程序编译之前具备约束检查性及早发现错误，这是使用泛型的根本原因。某些类或者方法，比如ArrayList，将来要操作某些类型不确定的对象，为了避免发生意外，通常要对该类或方法进行编译前的限制，可以限定其为某一类型，<E>，也可以限定其为某一类型的子类<? extends E>，还可以限定为通配类型<?>。在最终使用的时候一定要将形式泛型参数中传入实际的泛型参数，比如ArrayList<String> Set< Map.Entry<K, V> >