一小时教你学会 ARM 架构

架构的演变历史

我们首先介绍ARM Ltd，这里先说的是公司而不是架构。ARM的发展历史非常久远，超乎许多人的想象。首先，我们提供一些背景信息，ARM成立于20世纪90年代末，从另一家位于剑桥的公司分拆而来，那家公司叫做Acorn Computers，曾经是英国教育市场的著名个人台式计算机供应商，现已不复存在，80年代中期时，Acorn一个小团队接受了一个挑战，为他们的下一代计算机挑选合适的处理器，他们起草了一个技术需求说明书，经过相当长的摸索后得出一个结论：无法找到与之相符的产品，于是Acorn决定自己设计处理器，一个小团队只用了18个月就完成了设计并实现了这款处理器。1985年4月26日，第一台原型机在Acorn的剑桥办公室中开始运行代码，那时它被称为”Acorn RISC Machine”，随着Acorn公司转向衰落，处理器设计部门被分了出来，组成了一家新公司，最初叫做Advanced RSIC Machines Ltd。现在公司和处理器都简称为ARM。

ARM以其各种RISC处理器内核而著称，但也出品大量的支持技术满足芯片设计师和软件开发者的需要，这包括物理IP，软件模型和开发工具，图形处理器，以及外围设备，注意，但是ARM并不生产芯片，ARM是半导体知识产权业务中的开拓先锋，目前市面上大量ARM设备都是由ARM分布于世界各地的授权商制造的。下面说下ARM产品在哪些领域通过通用数字产品发挥用武之地，ARM提供：在系统芯片(SoC)上的系统级IP，以及物理IP，确保其可制造性开发工具，帮助设计和制造系统架构和软件，当然生产出成品还需要许多其他投入，如工业设计，封装，环境调查，操作系统，外围IP等，这些都不是ARM的产品，但ARM有很多合作伙伴，制造出了成千上万的设备，从下图中你会发现许许多多部署了基于ARM解决方案的应用。

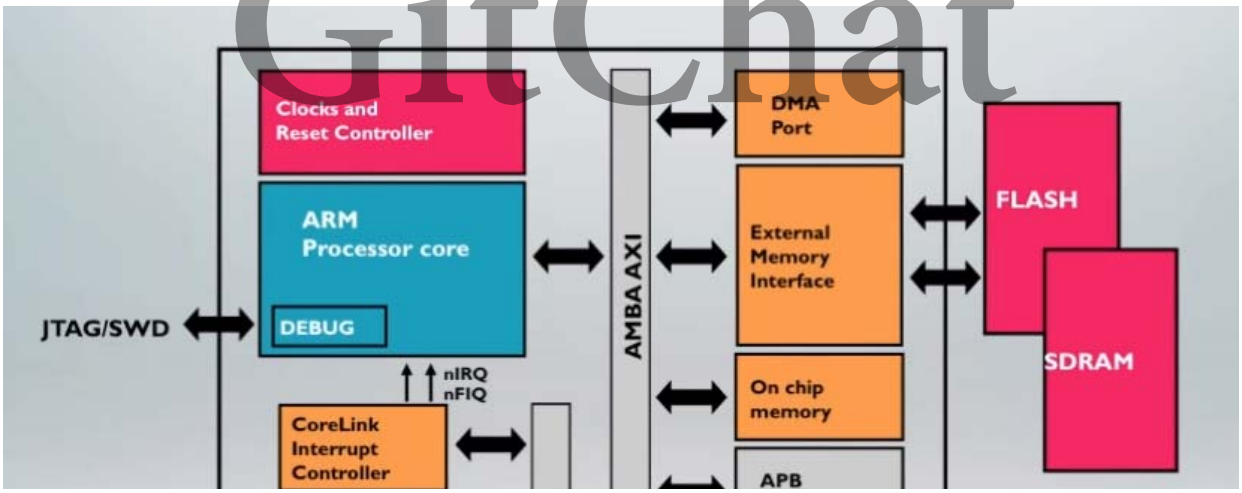
到500亿片，每年还以大约80亿的数字在增长，到2020年，总量有望达到1000亿。

ARM内核现在广受欢迎的一个原因是支持一系列的性能和功能点。我相信许多人听说ARM是从ARM7TDMI处理器内核的成功开始的，从上世纪90年代起这一内核在手机行业得到了广泛采用，也是ARM早期成功的奠基石，虽然现在依然受到广泛使用，也可以购买到包含这一内核的大量部件，但不再提供ARM7TDMI的授权许可，现在已经从这一内核发展出以实时嵌入式空间为目标的整个产品路线图，现在有两大产品系列，Cortex-M系列主要用于注重成本节约的微控制器；Cortex-R系列，提供非常高的性能和吞吐量，同时保持精准的时序属性和可预测的中断延时，通常用于时序关键的应用中，如引擎管理系统和硬盘驱动器控制器。

后来发展了整个系列的应用处理器，从产品线最初的ARM926EJ-S开始，发展到了ARM11MP，现在包含了Cortex-A系列，这些处理器设计为可在要求linux等平台操作系统的应用中提供可缩放的高性能，它们融合了精密的内存管理功能，以及多媒体处理扩展指令集，从ARM11MP开始加入了针对多核系统的支持，Cortex-A系列的最新核心现在以多核配置提供，这使得它们能够真正涵盖广阔的功率和性能点范围。

我们现在已经对公司渊源和架构有了些了解，现在让我们进入ARM芯片的内部吧。

内核的工作原理



设总线，APB通常用于连接所有外设，AXI则用于存储器和其他发高速设备，大多数设备都有一定数量的芯片上存储以及连接外设存储器设备的接口，但是注意，与设备的外部连接并不是AMBA总线，这仅在设备内部使用，并不外露。

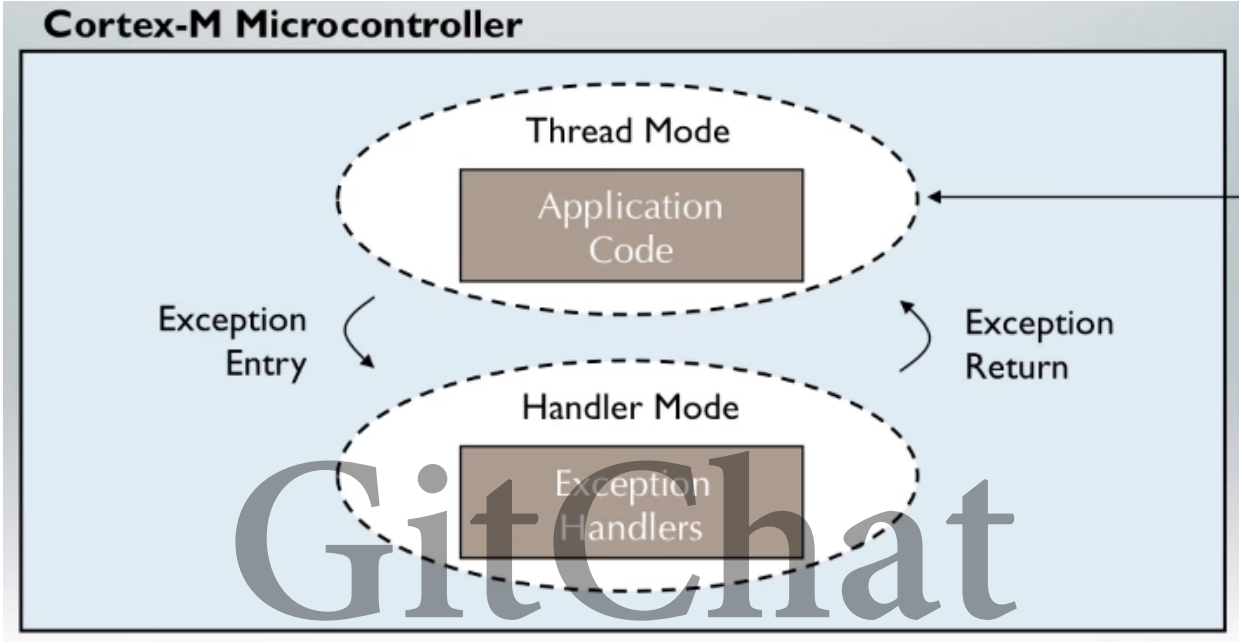
下面看看这个Soc的工作原理——编程器模型（programmer's model）。谨记，A系列和R系列配置在编程器模型上非常相似，但M系列配置在许多非常重要的方面都有很大不同，这在接下来的讲解中会指出这些差别。从根本上说，ARM是RISC架构，你可能会否认现在的ARM内核其实不属于RISC平台，但它们与RISC有很大的渊源，也保留了传统上与RISC架构相关的许多特性，例如大多数指令在一个周期内执行，寄存器集基本上是正交的，而且指令集实施加载存储式架构，也就意味着能够直接处理内存中内容的指令只有加载和存储指令，如果需要对内存中的值执行任何处理，程序必须将这些值加载到寄存器中，执行所需的处理，然后将结果存回到内存中，其他常见架构则能够直接操控或修改内存中的内容。

所有的内部寄存器除了一些受到NEON架构的矢量处理功能支持外都是32位宽的，它们的内部由32位ALU处理，内存则通常在32位元中予以处理，这就是ARM的字长。谈到指令集时，你会发现ARM核心不只有一个指令集，所有ARMv7-A和ARMv7-R核心都支持32位原生ARM指令集和Thumb指令集，后者中的指令可以是32位或者16位的。ARM指令集释放了内核的完整性能潜力，而Thumb指令集则提供了更出色的代码密度，我们把ARM和Thumb指令间切换这一过程称为“交互工作”，不要担心，编译器和链接器会处理它们。一些较旧的内核支持Thumb指令集的早期版本，其中所有的指令都是16位指令，比如ARMv7-M内核仅就支持Thumb指令集。

如果你之前接触过处理器架构，相信你会熟悉运行模式的概念以及特权的概念。许多架构通常支持两种模式，分别为“Supervisor”和“User”，其中一个模式拥有特权，另一个则没有。在无特权模式下代码可能无法直接执行某些特定的操作，比如，禁用中断，重新配置内存保护，或访问特定的内存区域，这是大多数操作系统的基本要求，允许系统从用户任务中保护自己。ARM内核通常支持七种基本运行模式，每种模式有权访问自己的堆栈空间，以及一组不同的寄存器子集，除一个外其余都是由特权的模式，如下：

Mode	Description
Supervisor (SVC)	Entered on reset and when a Supervisor call instruction (SVC) is executed

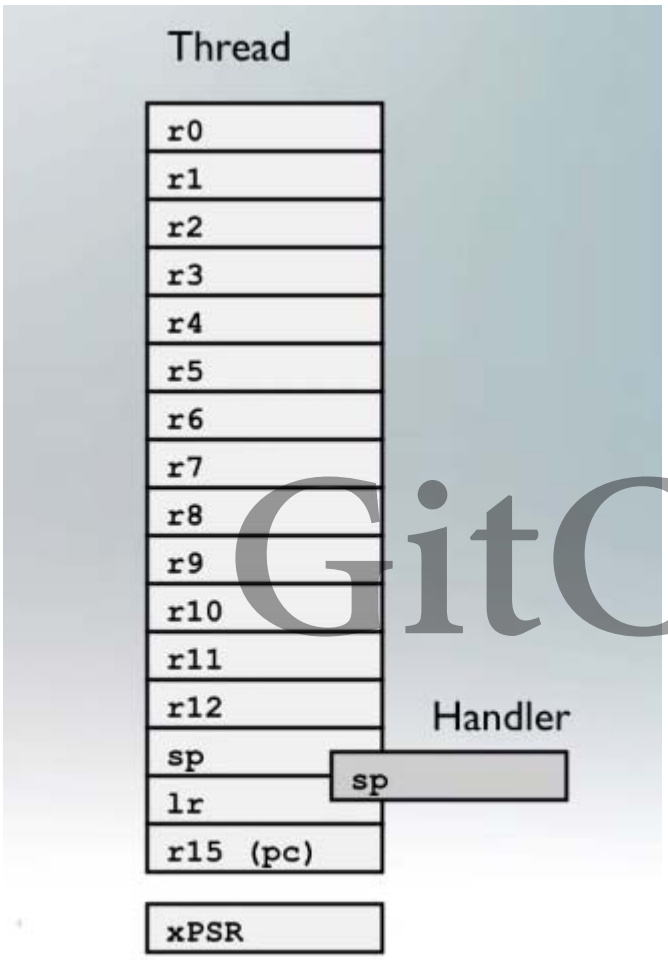
其中6种是特权模式，User模式是没有特权的模式，作为唯一的无特权模式，User模式供操作系统用于用户任务和处理器。此外，有5种模式称为“异常模式”，每一种都与处理特定种类的异常或中断相关，例如，当内核开始处理外部中断时会自动进入IRQ模式，而Supervisor模式则用于处理SVC指令和硬件复位，这些模式分别拥有专属的堆栈空间，以及一小组专用寄存器，我们把这一功能叫做寄存器编组，只是这些异常属于不同的类型。注意，上图仅适用于Cortex-A和Cortex-R处理器，Cortex-M微控制器的模式结构则全然不同。ARMv7-M架构配置仅定义了两种模式，如下图，分别是Thread模式和Handler模式，Thread模式没有特权，用于应用程序代码，Handler模式有特权，用于异常处理程序，当系统复位时在Thread模式中开始执行，遇到异常时自动变为Handler模式，处理程序完成后再回到Thread模式。



下面我们重点讲下这些模式是如何与寄存器组交互工作的:

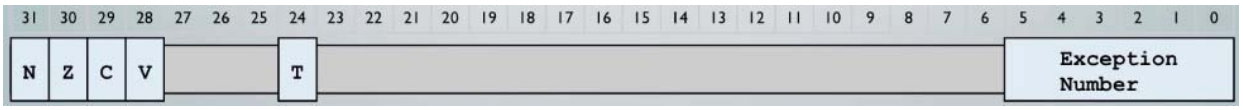
快照，才能使得在处理中断事件后返回到User模式并恢复程序变得非常容易，当中断处理结束后，就回到User模式，重新获取原先的寄存器。

在以上描述的寄存器集合和组织适用于Cotex-M之外的所有ARM内核，Cotex-M内核具有不同的寄存器集合和组织，见下图。之前一直强调Cotex-M寄存器是不同的，差别就在这里，只有18个寄存器没有我们在其他内核上看到的编组方案。首先，有13个通用寄存器，其中r0到r7是低位寄存器，r8到r12是高位寄存器，还有3个特殊寄存器：Stack Pointer，Link Register和Program Counter，最后一个寄存器是程序状态寄存器xPSR。注意，Contex-M内核有两种处理器模式：Thread模式和Handler模式，只有一个寄存器在这两种模式之间编组，它就是Stack Pointer。



数据中止。E位允许在程序控制下动态更改数据接口的字节序(Little或Big字节序)，简化了混合字节序数据的处理。剩余的位被“保留”或者用于和特定指令的内部系统状态，不可由程序修改。

下面来讲一下Cortex-M内核中可用的状态寄存器：



你会发现它比前面讲的状态寄存器简单的多，这也说明了Cortex-M内核的简洁性。有一个T位，因为Cortex-M内核仅支持Thumb指令集，所以此位始终是1。最后又一个字段，它在核心执行异常处理程序时包含当前活动的异常编号。初学者可能会问异常时会发生什么，在ARM架构中，异常是某种类型的事件，导致任何内容正常的程序流中出现中断，异常可以是内部的，如内存转译错误；也可以是外部的，如来自外设的中断；也可以是同步的，如SVC指令；或者是异步的，如计时器中断。无论原因如何，核心对所有异常的处理方式基本上相同。

当一个应用程序在逐一执行各个指令时，异常来时内核要做的第一件事就是确保它能够在异常之后回到这一点上，为此我们必须对当前状态抓取一个快照，所以内核复制CPSR并保存在SPSR中，再复制PC并保存在LR中，然后内核切换到相应的异常模式禁用进一步的中断，确保它处于正确的状态，接着使用矢量表确定可以找到异常处理程序的位置，每一个异常类型分别有一个条目，每一条目是一个指令，分出相关的处理程序代码，所以核心就是从正确的矢量表条目加载Program Counter执行异常处理程序。当处理程序完成时，要返回到中断的程序就简单了，只要从SPSR中保留的副本还原CPSR，再从链接寄存器还原Program Counter。当然Cortex-M在处理异常时完全是另一回事，这里就不详讲了。

现在相信你已经了解了寄存器，模式和状态的所有信息，现在我们来谈谈ARM内核提供的指令集。目前市场上的大多数ARM内核至少支持两种指令集：原生的32位ARM指令集，以及混合了16位和32位的Thumb指令集，我们先看看ARM指令集。虽然这次chat不是ARM汇编语言的课程，但也能让你有足够的了解。ARM指令集中的所有指令都是32位长，乍一看ARM指令的语法似乎非常复杂，不过一旦你了解运算符和可能的运算对象的基本结构，其实还是非常简单的，毕竟它是RISC架构。下面举例说明，第一个真的很简单：

3. ANDS r4, r4, #0x20
   这是一个逻辑AND指令，注意这个AND有个后缀'S'，这指定将CPSR中的ALU条件代码设为反映该结果，ARM数据处理运算默认情况下不影响条件代码，所以使用这个'S'后缀来指定需要这么做的运算。

4. ADDEQ r5, r5, r6
   这又是一个ADD，它是有条件指令，该助记符带有"EQ"后缀，表明只有在达到EQ条件为真时才会执行这一指令，如果该条件不为真，指令将表现为NOP。

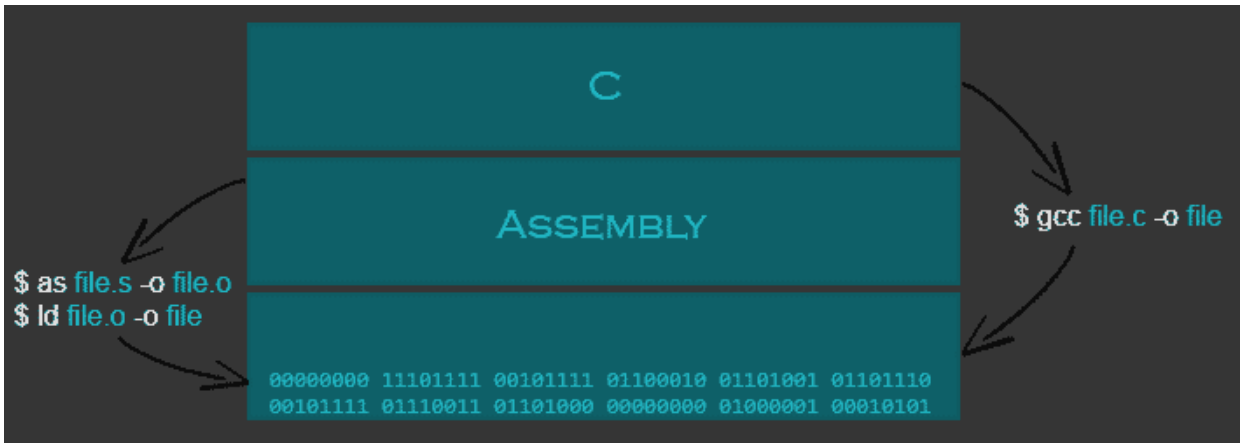
5. LDR r0, [r1]
   这是一个加载指令，将r1中指定地址的值加载到r0中。在指定内存访问指令的地址时，我们使用方括号来表达。

6. STRNEB r2, [r3, r4]
   这是存储指令，只有在NE条件有效时才会执行操作，其次它是一个字节层面的存储，它将r2中最不重要的字节存储到r3加r4得到的内存位置上。

目前为止，我们只是谈了ARM指令集，众所周知所有ARM指令都是32位的，为了提供更好的代码密度，ARM在很久之前推出了第二指令编码，叫做Thumb，Thumb所有指令都是16位的。Thumb代码通常在代码密度上可以改善大约35%，大多数C和C++代码都针对具备Thumb功能的核心上的Thumb进行编译。既然Thumb这么好，我们为何要把真么多精力放在ARM指令集上呢？这是因为Thumb是编译代码的最佳目标，如果你直接在汇编程序中编写代码，ARM相对是更好的选择。下面让我们进一步地剖析ARM的实现原理。

ARM的技术实现

要想深入理解ARM的实现原理是个很大的学习工程，这里一样希望读者读后能对ARM起到一个总体的认识，后续可以进一步的深入学习。我们先以ARM汇编基础来展开这一章的chat。



我们从最底层来看下，在最底层，电路上有电信号，信号是将电压切换为两个电平来形成的，例如0伏(关)或5伏(开)。因为只是我们不能轻易的告诉电路电压，只能选择使用1/0来写入开/关的模式，然后我们对0和1的顺序进行分组，以形成机器码指令，该指令是计算机处理器的最小工作单元，以下是机器语言的示例：

```
1110 0001 1010 0000 0010 0000 0000 0001
```

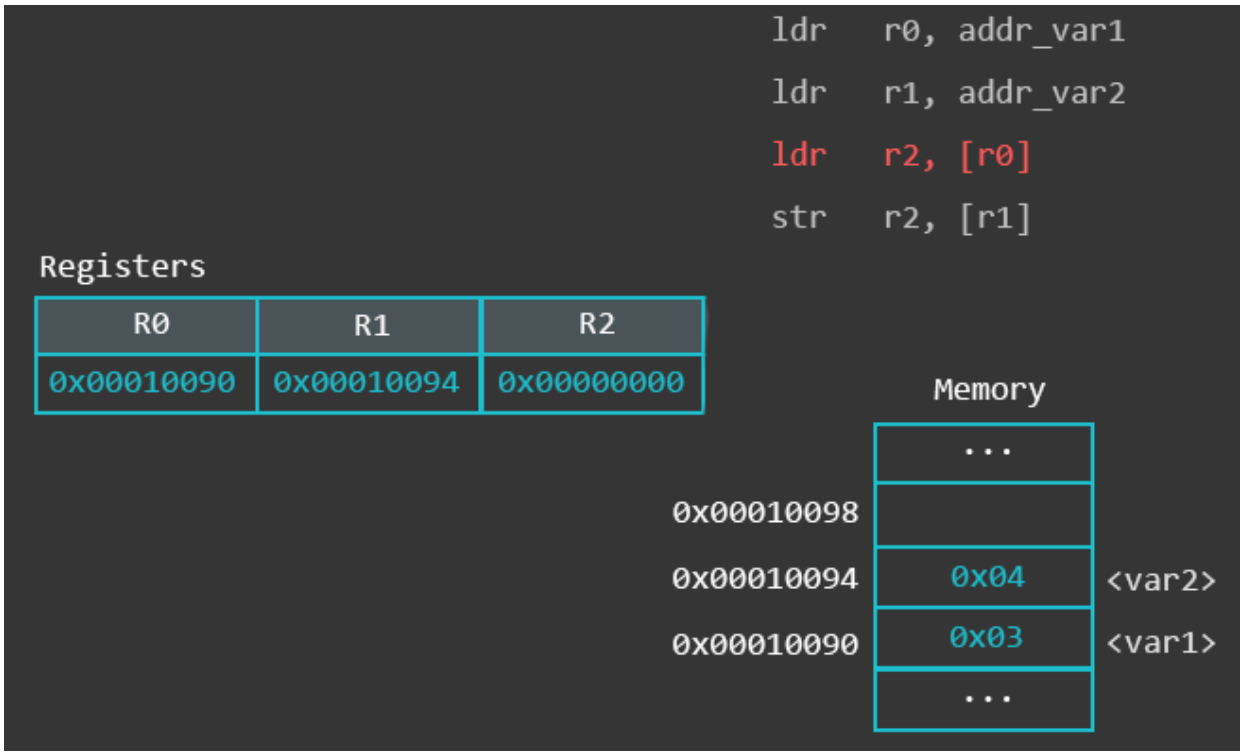
我们知道ARM处理器只能对寄存器执行数据处理，所以与存储器的交互有两种：从存储器加载到寄存器，并将值从寄存器存储到存储器，即ARM使用加载/存储(LDR和STR)模型进行内存访问。通常LDR用于将内存中的内容加载到寄存器中，STR用于存储寄存器中的内容到存储器地址。我们来举一个基本例子：


```
.data / * .data部分是动态创建的，其地址不能轻易预测* /
var1: .word 3 / *内存中的变量1 * /
var2: .word 4 / *内存中的变量2 * /

.text / *开始的文本（代码）部分* /
.global _start

_start:
```

第一看的小伙伴或许会一头雾水，下面以一张动态图来解释下ARM是如何和存储器交互的：



参考

正如刚开始所说的，本次chat不是所有ARM架构和技术的详尽概览，而是通向ARM世界的一扇大门，ARM网站上有丰富的文档等你去查阅，探索。比如（http://infocenter.arm.com）可以找到架构参考手册，知识库文章，常见问题解答，处理器文档，以及开发者指南等。ARM还有一个不断壮大的全球大学计划，为你提供大量的教学和培训资源，软件工具，以及硬件开发板。