Vectorization vs. Compilation in Query Execution 论文阅读笔记

阅读论文之前，我们首先来了解下cpu几个影响执行引擎执行效率的特性：如超标量流水线与乱序执行，分支预测，多级存储与数据prefetch，simd

超标量流水线与乱序执行

CPU指令的执行可以分为多个阶段：fetch instruction,decode instruction,execute instuction,memory access,write back result

流水线的含义：一套控制单元可以同时执行多条指令，不需要等到上一条指令执行完就可以执行下一条指令。程序分支越少或者是分支预测成功率越高，对流水线的执行就越有利，因为如果预测失败了，是要丢弃当前pipeline的所有指令重新flush，这个过程往往会消耗掉10几个cpu周期

超标量的含义：一个cpu核有多套控制单元，因此可以有多条pipeline并发执行。cpu还会维护一个乱序执行的指令窗口，窗口中的无数据依赖的指令就可以被取来并发执行。并发指令越多越好，因为这样指令之间没有依赖，并发流水线的执行会更加的流畅。

分支预测

程序分支越少，流水线的效率就会越高。程序分支可以分为两种：

- 条件跳转

来自于if或者switch之类的语句

- 无条件跳转

根据指令的操作数为跳转地址还是跳转地址指针，分为直接跳转和间接跳转。直接跳转一般为静态绑定的函数调用，间接跳转来自函数返回或者虚函数之类的动态绑定函数调用。

分支预测的含义：执行跳转指令的时候，在得到跳转目的地址之前，不知道从哪里取下一条指令，pipeline就只能stall，那么为了提高这种情况下的流水线效率，cpu会引入一组寄存器，用于专门记录最近几个跳转指令的目的地址。这样，当再一次执行到这个跳转指令的时候，就可以直接从上次保存的目的地址中取出指令，放入流水线。等到真正取得目的地址的时候，再看如果取错了，就flush掉当前pipeline的流水线，取正确的指令去执行。

多级存储与数据预取

当数据在寄存器，cache或者内存中，cpu取数的速度并不是在一个数量级上的。

cpu取指令/数据的时候并不是直接从内存中取的，通常cpu和内存中会有多级缓存。分别为l1，l2, l3 cache, 其中l1cache又可以分为l1-data cache, l1-instruction cache。先从cache中取数据，若不存在，才访问内存。访问内存的时候会同时把访问数据所在的一个内存块一起加载入cache中。

prefetch指的是若数据存在线性访问的模式，CPU会主动把后续的内存块预先加载入cache中。

SIMD

单指令多数据流，对于data-intensive的程序来说，可能会需要对大量不同的数据进行相同的运算，SIMD引入前，执行流程为同样的指令重复执行，每次取一条数据进行运算。例如有8个32位整形数据都需要进行移位运行，则由一条对32位整形数据进行移位的指令重复执行8次完成。SIMD引入了一组大容量的寄存器，一个寄存器包含8*32位，可以将这8个数据按次序同时放到一个寄存器。同时，CPU新增了处理这种*832位寄存器的指令，可以在一个指令周期内完成8个数据的位移运算。

Abstract

query的编译执行与解释执行相比，好处有：

1.减少了解释执行的性能消耗，编译执行是远远比解释执行要快的（python是解释执行的，所以python的运行效率十分的低）

2.指令编码的局部性更好

3.可以使用SIMD指令来加速

而这些好处都是由**向量化执行模型**带来的

重点关注的是在Project,select,hash join三种情况下的向量化执行和编译执行策略。

论文指出两种策略：query 的编译执行要求以block为单位组织数据，另外就是向量化的执行模型在性能上是比火山模型要更好的。

这两种策略的具体应用：

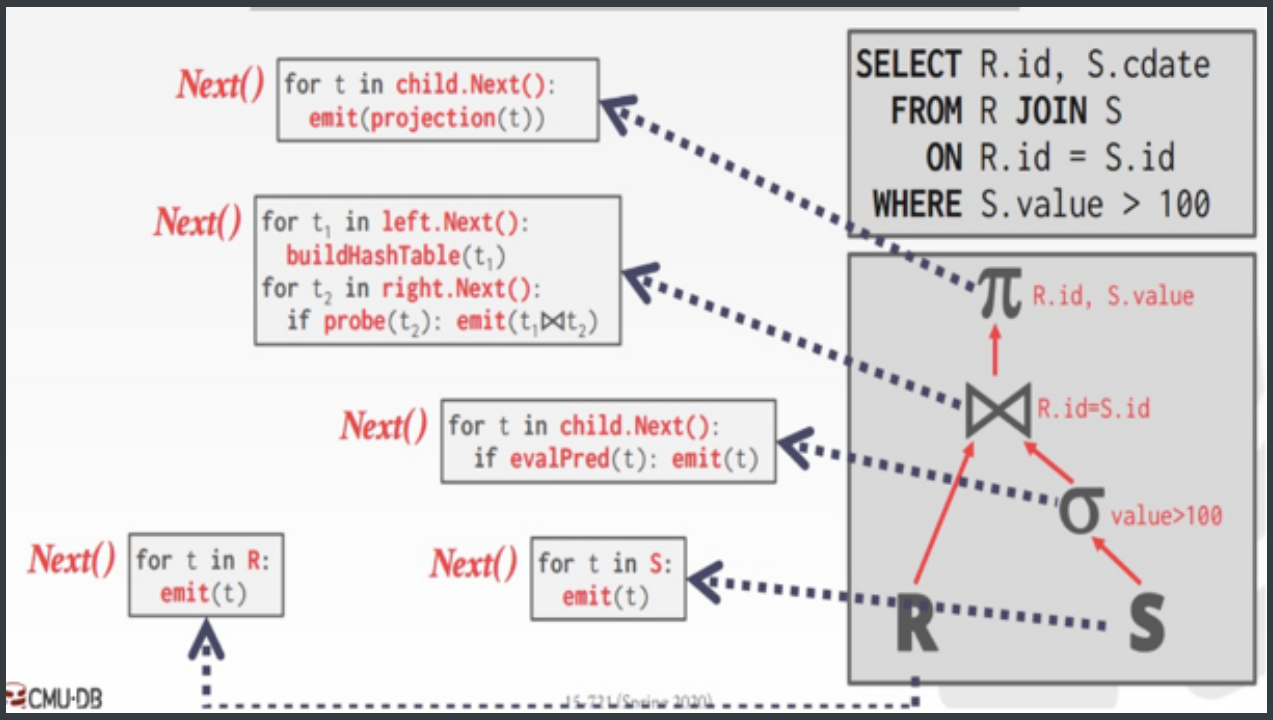
- 在编译执行的query计划中集成向量化执行策略
- 利用query的编译执行去生成用于向量化执行的block

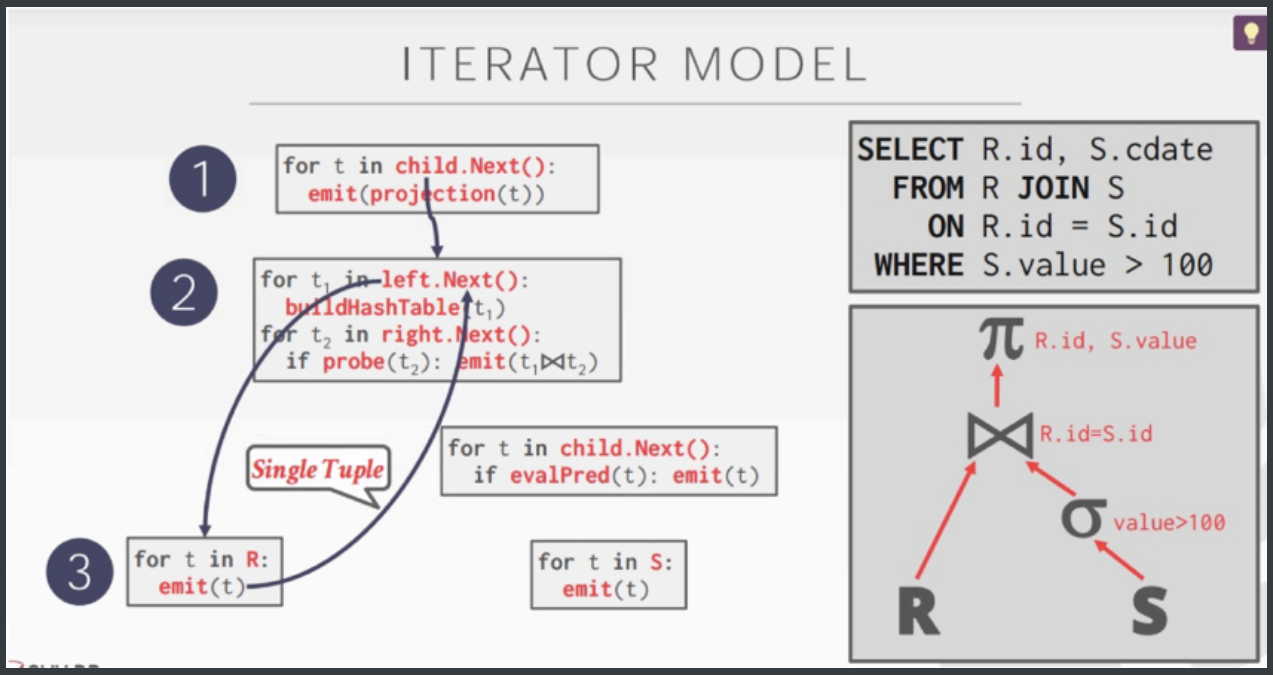
Introduction

解释执行的模型之火山模型

执行计划由多个代数算子组成一颗查询计划树，如Scan,Join,Project,Aggregation和Select，每个算子内部包含了若干表达式，表达式可以是join或者是select所使用的bool表达式，也可以是投影算子所使用的计算表达式，也可以是聚合算子所使用的MIN,MAX,SUM表达式。

大多数的query解释器模型都是使用基于迭代器的火山模型，每个算子看成一个iterator，iterator都会提供一个next()方法，每个next()方法只会产生一个tuple，可以理解为一行数据或者是一行的多列数据。查询执行的时候，查询树自顶向下调用next()接口，数据则自底向上被拉取处理。所以火山模型属于“pull”模型。





而在整个pipeline中，每次只会传递一个tuple的数据。下面来简要分析一下火山模型的缺点，即为何性能如此之差：

- 时间都花在了评估query plan上，而不是计算query 结果上。

next（）函数实现为虚函数，调用虚函数的时候要去查虚函数表，在编译器实现中为非直接跳转。会导致一次错误的cpu分支预测，需要多花费十几个周期的开销。火山模型为了返回一个tuple需要调用多次next方法。

- cpu cache利用率低

next方法一次只返回一个元组，元组通常采用行存储，如果仅需访问其中某些列而每次均将一整行填入CPU Cache，将导致那些不会被访问的列也放在了Cache中，使得cache利用率非常的低。

- 不能充分利用**SIMD**和 **CPU流水线**

目前火山模型的数据库对于cpu设计存在的问题：

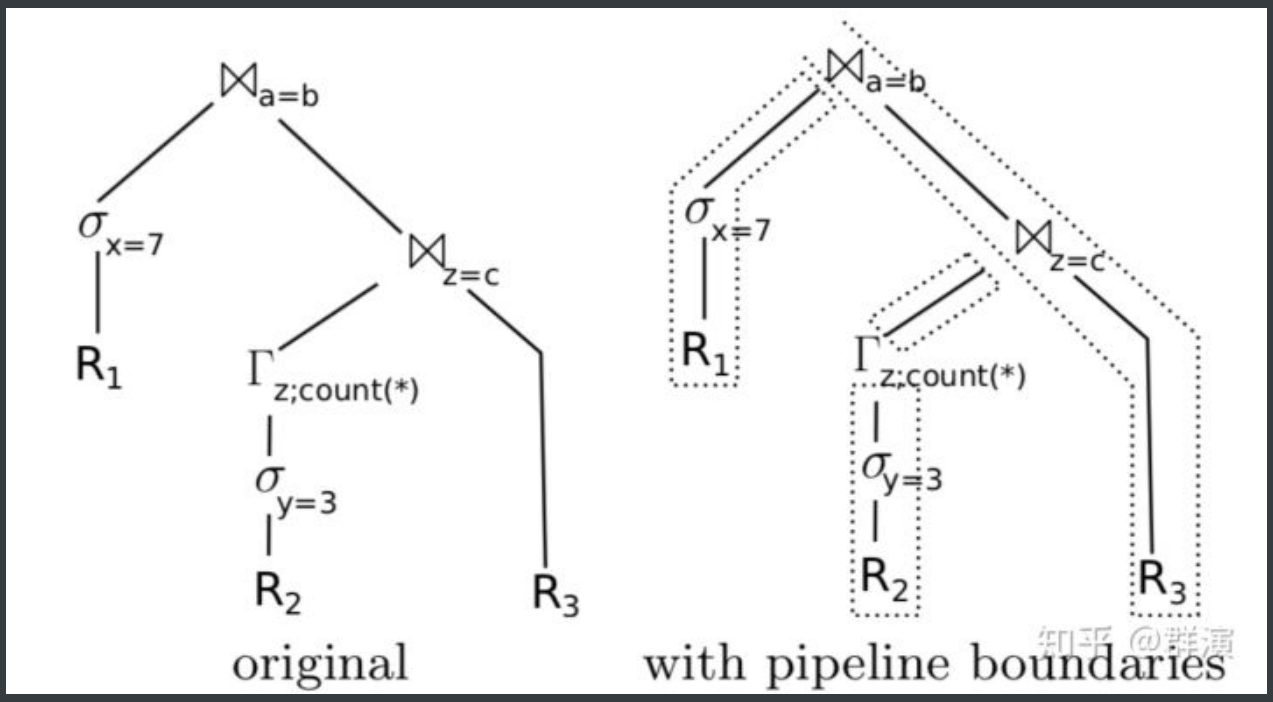
1.依赖性：如果指令之间相互依赖，那么它们是不能够被立刻推进同一条pipeline里面的。

2.分支预测：cpu会尝试预测程序中的某一个代码分支，并且会把指令填充到pipeline中，如果预测失败的话就会把所有预测的工作全部丢弃并且重新flush这个pipeline

编译执行：

火山模型大量虚函数的调用导致的性能损失，编译执行的模型采用push based的执行模型来解决问题。简单来说，模型是自底向上执行的，执行逻辑由底层operator开始，处理完一个tuple后，再把tuple传给上层的operator处理。

举个例子：

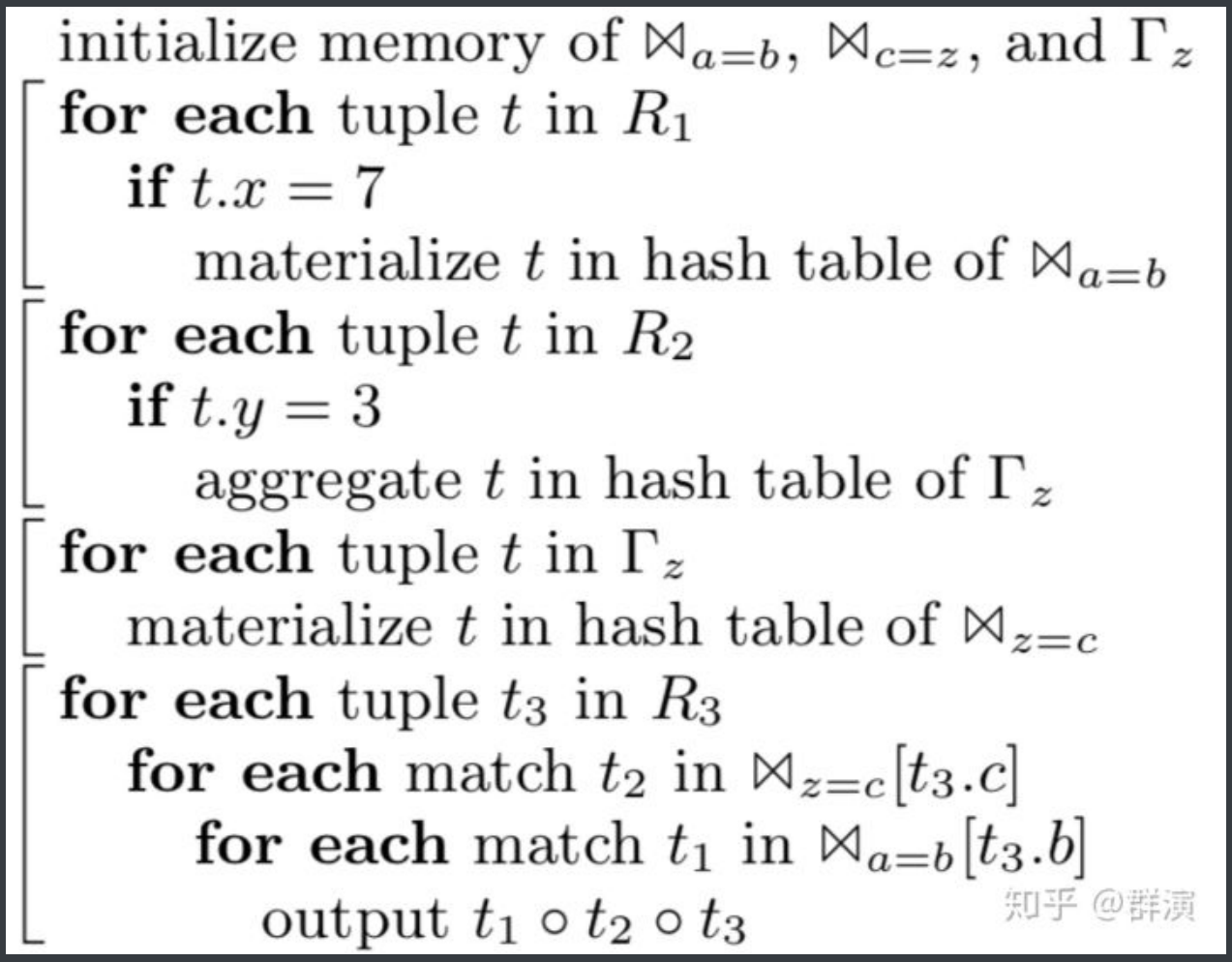


我们知道，数据访问效率最高的就是寄存器。所以执行查询树的时候最理想的情况就是数据一直保留在寄存器中，operator就可以直接处理寄存器的数据。

最理想的情况，就是tuple留在寄存器不动，第一个operator直接在寄存器处理完数据后，交给第二个operator来处理，消灭了next（）函数调用来传递tuple这个过程，执行模式变成了“data-centric”以数据为中心的模型。

但现实是骨感的，并不是所有的operator的运算逻辑都可以处理完寄存器中的tuple之后，把tuple留在寄存器中，由下一个Operator 接着处理。例如Join的时候，需要构建hash表，tuple就必须物化（materialization），从寄存器读出来写入内存。

这种operator叫做Pipeline Breaker，那么正如上图右方，查询树将会以Pipeline Breaker为分割，将查询树划分为多个Pipeline,在一个Pipeline内，数据可以一直留在寄存器中。

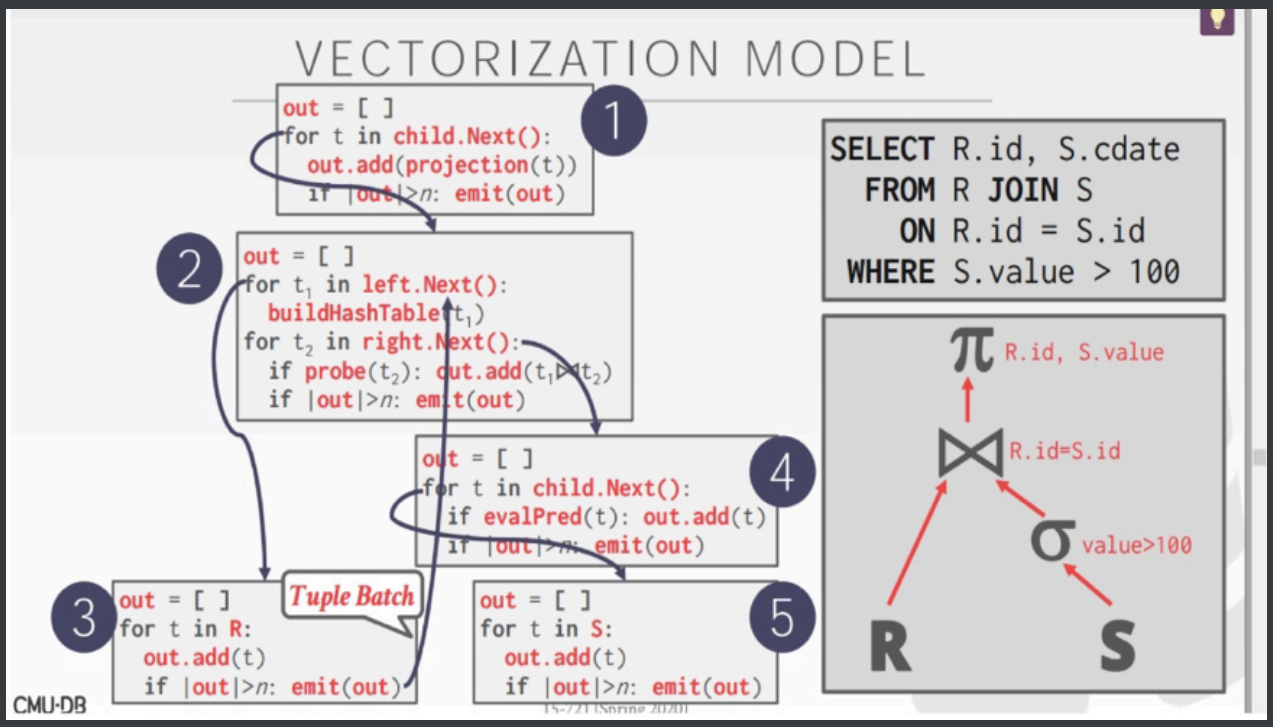


一句话总结就是：编译执行以数据为中心，消灭了火山模型中的大量虚函数调用开销。甚至使大部分指令执行，可以直接从寄存器取数，极大的提高了执行效率

在java中通过JIT来实现，在c++中通过llvm来实现codegen，对于olap这种运行时间较长的query来说，通常编译的时间是可以忽略的。

向量化执行：

向量可以理解为按列组织的一组数据，连续存储的一列数据，在内存中可以表示为一个向量。



而向量模型和火山模型的本质区别就在于，数据处理的基本单元不再是按行组织的tuple，而是按列组织的多个vector，我们常说的一个block其实就是多个vector的集合，就是多个列的意思。

好处是：

- 由于每次next（）都是处理一个batch的数据，那么大大减少了虚函数调用的次数，分支预测的成功概率会提高，免去了分支预测失败多消耗的那十几个cpu周期，解释执行耗费的时间大大减少。
- 每个operator处理的是batch数据，那么每个operator内部都是一个tight-loop循环处理，这就可以用到不少性能优化的trick了：如循环展开，让编译器自动生成SIMD指令来一次处理多组数据等。
- 以block形式组织数据，提高了cache利用率

  行式：tuple cache中存储的是一行的很多列，许多列数据不会被利用，占用了不必要的cache空间

  列式：vector cache中存储的是一列中的很多行，很有可能所有数据都会被利用
- 多行并发处理，很好低利用了cpu乱序执行与并发执行的特性。
- 同时也有利于cpu流水线的执行，因为并发指令的执行，指令之间没有依赖加上分支预测的成功率提高了，流水线的并发执行会更加地流畅。

编译执行和向量化执行比较

向量化执行的主要访存开销在于像join这种算子的物化开销，物化就是从寄存器把数据读到内存中。

而编译执行的话，tuple数据可以一直留存在寄存器中，一个operator处理完后，给另外一个operator继续处理。除非遇到不得不物化的情况。

向量化执行模型的循环较短，并发度高，可以同时有更多的指令等待取数.

编译执行循环内部会包含多个Operator的运算，这些有依赖关系的指令占据了大部分的乱序执行窗口，并发度低

编译执行模型Pipeline执行无Materialization,没有访存那些乱七八糟的开销

编译执行与向量化执行的融合

基本思想是把查询树分解一下，部分用向量化方式，部分用编译执行方式。

编译执行的主要目标是减少Materialization，在编译执行的基础上，主动在其中的Pipeline中插入Materialization，将Pipeline分割为Stage，在Stage内依然是tuple-at-a-time data-centric的推送模型，保留了编译执行数据停留在寄存器中的优点。而跨Stage或Pipeline时，则以Block(一组tuple)为单位传递数据，这个时候就可以利用上SIMD，获得向量化执行的优点。

ref

https://zhuanlan.zhihu.com/p/63996040