电脑开机的时候系统做了什么？

1）加载BIOS

因为ROM的发明，开机程序会被刷入ROM中。当计算机通电的时候，首先读取ROM。

ROM里面的程序叫做**基本输入输出系统（Basic Input Output System BIOS）**

BIOS首先“硬件自检”，查看硬件是否能够工作。完成后BIOS把权限交给启动程序，用来对启动设备进行排序，依次启动。基本就是对主板上的键盘、鼠标、外部接口、频率、电源、磁盘驱动器等方面进行

2）读取MBR

BIOS首先把控制权交给存储设备。系统会读取该设备的最前面512字节，如果最后两个字节分别是0x55和0xAA则表示可以启动，然后把控制权交给下一个设备

存储设备最前面的512个叫做**主引导记录（MBR）**，由三部分组成：

①1-446字节，调用操作系统的机器码

②447-510字节，分区表（选择在那个分区启动，以前是把不同操作系统放在两个硬盘）

③511-512字节，主引导记录签名（0x55和0xAA）

3）Bootloader

Boot Loader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核做好一切准备。Boot Loader有若干种，其中Grub、Lilo和spfdisk是常见的Loader。Linux环境中，目前最流行的启动管理器是 Grub。

4）加载内核

内核加载后，接开始操作系统初始化，根据进程的优先级启动进程，这时候linux操作系统已经可以运行了

5）Loading Kernel image 和 initial RAM disk

6）用户层init依据inittab文件来设定运行等级

```
0：关机

1：单用户模式

2：无网络支持的多用户模式

3：有网络支持的多用户模式

4：保留，未使用

5：有网络支持有X-Window支持的多用户模式

6：重新引导系统，即重启
```


- BIOS和UEFI（Unified Extensible Firmware Interface）则是取代传统BIOS的，相比传统BIOS来说，它更易实现，容错和纠错特性也更强。
- **MBR与GPT：**MBR是传统的分区表类型，当一台电脑启动时，它会先启动主板上的BIOS系统，BIOS再从硬盘上读取MBR主引导记录，硬盘上的MBR运行后，就会启动操作系统，但最大的缺点则是不支持容量大于2T的硬盘。而GPT是另一种更先进的磁盘系统分区方式，它的出现弥补了MBR这个缺点，最大支持 18EB 的硬盘，是基于 UEFI 使用的磁盘分区架构。

都有那些编程范式？

- 面向过程（Process Oriented Programming，POP）
  最原始，也是我们最熟悉的一种编程语言。他的编程思维源自于计算机指令的顺序排列。
  步骤：首先将待解决的问题抽象为一系列概念化的步骤。然后一步一步的按照顺序实现所有步骤。
  优点：流程化使得编程任务明确，在开发之前基本考虑了实现方式和最终结果，具体步骤清楚，便于节点分析。效率高，面向过程强调代码的短小精悍，善于结合数据结构来开发高效率的程序。
  缺点：需要深入的思考，耗费精力，代码重用性低，扩展能力差，后期维护难度比

较大。

- 面向对象（Object Oriented Pr
- ogramming，OOP）
  所有事物都是对象。易于维护，扩展和复用
  优点:结构清晰，程序是模块化和结构化，更加符合人类的思维方式；易扩展，代码重用率高，可继承，可覆盖，可以设计出低耦合的系统；易维护，系统低耦合的特点有利于减少程序的后期维护工作量。
  缺点：开销大，当要修改对象内部时，对象的属性不允许外部直接存取，所以要增加许多没有其他意义、只负责读或写的行为。这会为编程工作增加负担，增加运行开销，并且使程序显得臃肿。
  性能低，由于面向更高的逻辑抽象层，使得面向对象在实现的时候，不得不做出性能上面的牺牲，计算时间和空间存储大小都开销很大。

  > 举个例子：下五子棋
  >
  > 面向过程：开始游戏（）；
  > 黑子先走（）；
  > 绘制画面（）；
  > 判断输赢（）；
  > 轮到白子（）；
  > 绘制画面（）；
  > 判断输赢（）；
  > 返回到 黑子先走（）；
  > 输出最后结果；
  >
  > 面向对象：黑白双方，这两方的行为是一样的。棋盘系统，负责绘制画面。规则系统，负责判定犯规、输赢等。

- 事件驱动编程
  主要是用在图形用户界面，比如C#这种
  功能都是提前写好的，就等着触发
- 面向接口（Interface Oriented Programming， IOP）
- 面向切面（Aspect Oriented Programming，AOP）
- 函数式（Funtional Programming，FP）
- 响应式（Reactive Programming，RP）
- 数响应式（Functional Reactive Programming，FRP）

软件开发模型

瀑布模型

瀑布模型（Waterfall Model） 是一个软件生命周期模型，开发过程是通过设计一系列阶段顺序展开的，从系统需求分析开始直到产品发布和维护，项目开发进程从一个阶段“流动”到下一个阶段，这也是瀑布模型名称的由来。

瀑布模型核心思想是按工序将问题化简，将功能的实现与设计分开，便于分工协作，即采用结构化的分析与设计方法将逻辑实现与物理实现分开。将软件生命周期划分为制定计划、需求分析、软件设计、程序编写、软件测试和运行维护等六个基本活动，并且规定了它们自上而下、相互衔接的固定次序，如同瀑布流水，逐级下落。

现在的互联网项目已经不再像传统的瀑布模型的项目，有明确的需求。现在项目迭代的速度和需求的变更都非常的迅速。在软件开发的编码之前我们不可能事先了解所有的需求，软件设计肯定会有考虑不周到不全面的地方；而且随着项目需求的不断变更，很有可能原来的代码设计结构已经不能满足当前的需求。

优点：

每个阶段交出的所有产品都必须经过质量保证小组的仔细验证。

缺点：

瀑布模型是由文档驱动，在可运行的软件产品交付给用户之前，用户只能通过文档来了解产品是什么样的。瀑布模型几乎完全依赖于书面的规格说明，很可能导致最终开发出的软件产品不能真正满足用户的需要。也不适合需求模糊的系统。

迭代开发

迭代增量式开发，也越来越接近现代的开发流程。

在迭代式开发中，整个开发工作被组织 为一系列短小的、固定长度的小项目，每次迭代都包括需求分析、设计、实现与测试。采用迭代式开发时， 工作可以在需求被完整地确定之前启动，并在一次迭代中完成系统的一部分功能 或业务，再通过客户的反馈来细化需求，并开始新一轮的迭代。

敏捷开发模型

敏捷开发（Agile）是一种以人为核心、迭代、循序渐进的开发方法。在敏捷开发中，软件项目的构建被切分成多个子项目，各个子项目的成果都经过测试，具备集成和可运行的特征。简单地来说，敏捷开发并不追求前期完美的设计、完美编码，而是力求在很短的周期内开发出产品的核心功能，尽早发布出可用的版本。然后在后续的生产周期内，按照新需求不断迭代升级，完善产品。

首要任务是尽早地、持续地交付可评价的软件，以使客户满意。

频繁交付可使用的软件，交付的间隔越短越好，可以从几个月缩减到几个星期。

在整个项目开发期间，业务人员和开发人员必须朝夕工作在一起。

围绕那些有推动力的人们来构建项目，给予他们所需的环境和支持，并且相信他们能够把工作做好。

scrum开发模型

scrum的团队不需要那么大，十几个人即可。

> **下面先给出scrum的模型：**

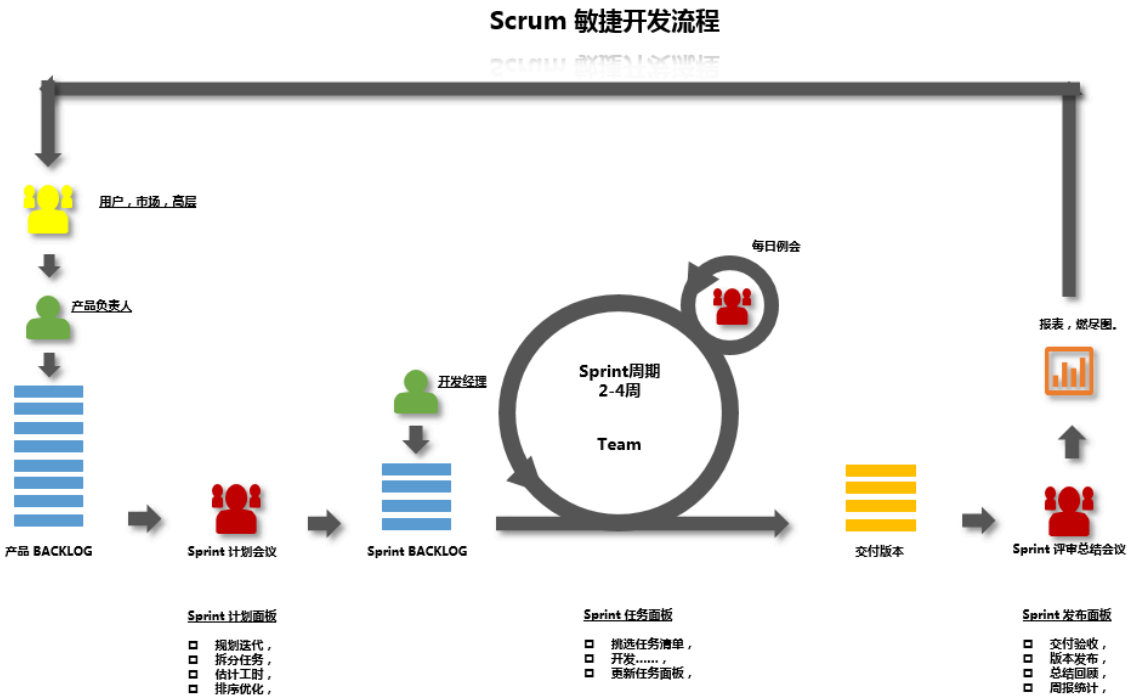
scrum所包含的角色

1. PO：Product Owner，产品负责人，确定「大家要做什么」。互联网公司的 PO 一般由相关的产品经理担任；如果是为客户做项目，PO 就是客户负责人。
2. Scrum Master：Scrum的推动者，掌控大节奏的人。
3. Scrum Team ：Developer，开发的主力。

三种角色有各自的责任，但三者间并没有上司和下属的关系。这正是 Scrum 区别于传统开发流程的精华：

- 传统的开发流程，是由领导拍板的中央集权制；
- Scurm 是人人平等的民主制，每个人的能力都被信任，更加自主，能发挥出更高的效率。

scrum的一些名词



1. Sprint：周期指的是一次迭代，而一次迭代的周期一般是2-4周，也就是我们要把一次迭代的开发内容以最快的速度完成它，这个过程我们称它为Sprint。
2. Backlog ：待办工作事项的集合。
3. Product Backlog ：PO将产品待办事项列表放入，是量化的用户需求，条目化地表达实际需要开发的需求。一般来说这个是以sprint来计算
4. Sprint Backlog：任务列表。是一次迭代中需要完成的任务，也是开发过程用得最多的Backlog，非常细化。一般来说以天来计算。

如何进行Scrum开发？

1. 我们首先需要确定一个Product Backlog（按优先顺序排列的一个产品需求列表），这个是由Product Owner 负责的；
2. Scrum Team根据Product Backlog列表，做工作量的预估和安排；
3. 有了Product Backlog列表，我们需要通过 Sprint Planning Meeting（Sprint计划会议） 来从中挑选出一个Story作为本次迭代完成的目标，这个目标的时间周期是1~4个星期（intel我们组是2周），然后把这个Story进行细化，形成一个Sprint Backlog；
4. Sprint Backlog是由Scrum Team去完成的，每个成员根据Sprint Backlog再细化成更小的任务（细到每个任务的工作量在2天内能完成）；
5. 在Scrum Team完成计划会议上选出的Sprint Backlog过程中，需要进行 Daily Scrum Meeting（每日站立会议），每次会议控制在15分钟左右，每个人都必须发言，并且要向所有成员当面汇报你昨天完成了什么，并且向所有成员承诺你今天要

完成什么，同时遇到不能解决的问题也可以提出，每个人回答完成后，要走到黑板前更新自己的 Sprint burn down（Sprint燃尽图）；

6. 做到每日集成，也就是每天都要有一个可以成功编译、并且可以演示的版本；很多人可能还没有用过自动化的每日集成，其实TFS就有这个功能，它可以支持每次有成员进行签入操作的时候，在服务器上自动获取最新版本，然后在服务器中编译，如果通过则马上再执行单元测试代码，如果也全部通过，则将该版本发布，这时一次正式的签入操作才保存到TFS中，中间有任何失败，都会用邮件通知项目管理人员；
7. 当一个Story完成，也就是Sprint Backlog被完成，也就表示一次Sprint完成，这时，我们要进行 Srpint Review Meeting（演示会议），也称为评审会议，产品负责人和客户都要参加（最好本公司老板也参加），每一个Scrum Team的成员都要向他们演示自己完成的软件产品（这个会议非常重要，一定不能取消）；
8. 最后就是 Sprint Retrospective Meeting（回顾会议），也称为总结会议，以轮流发言方式进行，每个人都要发言，总结并讨论改进的地方，放入下一轮Sprint的产品需求中；

C++和C的区别

面向对象和面向过程语言的区别

首要要知道这两个都是一种编程思想

面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了。

面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。

举一个例子：

例如五子棋，面向过程的设计思路就是首先分析问题的步骤：1、开始游戏，2、黑子先走，3、绘制画面，4、判断输赢，5、轮到白子，6、绘制画面，7、判断输赢，8、返回步骤2，9、输出最后结果。把上面每个步骤用分别的函数来实现，问题就解决了。

而面向对象的设计则是从另外的思路来解决问题。整个五子棋可以分为 1、黑白双方，这两方的行为是一模一样的，2、棋盘系统，负责绘制画面，3、规则系统，负责判定诸如犯规、输赢

等。第一类对象（玩家对象）负责接受用户输入，并告知第二类对象（棋盘对象）棋子布局的变化，棋盘对象接收到了棋子的i变化就要负责在屏幕上面显示出这种变化，同时利用第三类对象（规则系统）来对棋局进行判定。

可以明显地看出，面向对象是以功能来划分问题，而不是步骤。同样是绘制棋局，这样的行为在面向过程的设计中分散在了总多步骤中，很可能出现不同的绘制版本，因为通常设计人员会考虑到实际情况进行各种各样的简化。而面向对象的设计中，绘图只可能在棋盘对象中出现，从而保证了绘图的统一。功能上的统一保证了面向对象设计的可扩展性。

面向过程

优点：性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源;比如单片机、嵌入式开发、 Linux/Unix等一般采用面向过程开发，性能是最重要的因素。

缺点：没有面向对象易维护、易复用、易扩展

面向对象

优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统 更加灵活、更加易于维护

缺点：性能比面向过程低

C++和java的区别

指针：java语言在程序员层面屏蔽了指针，让程序员没办法根据指针找到内存，所以没有指针这一概念。但是java有内存的自动管理功能，从而能够避免c++那种内存泄露的事务。

多重继承：c++支持多重继承，但是java好像不支持，但是java有接口（抽象类，是一系列方法的声明，是一些方法特征的集合，一个接口只有方法的特征没有方法的实现），一个类可以继承多个接口。c++多重继承的问题。c++多重继承有虚继承来解决问题。

数据类型和类：java是一门完全面向对象的语言，因此所有的函数和变量必须是类的一部分，除了基本数据类型之外，其余都是作为类对象而存在的，对象将数据和方法结合起来把其封装在类中。

struct和union：java取消了struct和union，我的理解是struct本来就是类的始祖，用起来不方便，而且java一切皆对象，没必要使用struct。

操作符重载：java不支持操作符重载，这是c++突出特性之一。

预处理机制：c/c++都在编译前有一个预处理阶段，该阶段主要有源文件替换，宏替换，去掉注释等功能。java没有，但是提供了import（*import* 关键字. 为了能够使用某一个包的成员，我们需要在*Java* 程序中明确导入该包。）。

自动内存管理：java在堆上建立内存无需手动释放，java无用内存回收是用现成的方式在后台运行，利用空闲时间删除。

c++为什么不加入垃圾回收机制

参考链接

作为支持指针的编程语言，C++将动态管理存储器资源的便利性交给了程序员。在使用指针形式的对象时(请注意，由于引用在初始化后不能更改引用目标的语言机制的限制，多态性应用大多数情况下依赖于指针进行)，程序员必须自己完成存储器的分配、使用和释放，语言本身在此过程中不能提供任何帮助，也许除了按照你的要求正确的和操作系统亲密合作，完成实际的存储器管理。

C++的设计者Bjarne Stroustrup关于问题给了一段说法：我很害怕那种严重的空间和时间开销，也害怕由于实现和移植垃圾回收系统而带来的复杂性。还有，垃圾回收将使C++不适合做许多底层的工作，而这却正是它的一个设计目标。但我喜欢垃圾回收的思想，它是一种机制，能够简化设计、排除掉许多产生错误的根源。

需要垃圾回收的基本理由是很容易理解的：用户的使用方便以及比用户提供的存储管理模式更可靠。而反对垃圾回收的理由也有很多，但都不是最根本的，而是关于实现和效率方面的。

我的结论是，从原则上和可行性上说，垃圾回收都是需要的。但是对今天的用户以及普遍的使用和硬件而言，我们还无法承受将C++的语义和它的基本库定义在垃圾回收系统之上的负担。”

原码反码和补码

机器数和真值

在学习原码, 反码和补码之前, 需要先了解机器数和真值的概念.

机器数

一个数在计算机中的二进制表示形式, 叫做这个数的机器数。机器数是带符号的，在计算机用一个数的最高位存放符号, 正数为0, 负数为1.

比如，十进制中的数 +3 ，计算机字长为8位，转换成二进制就是00000011。如果是 -3 ，就是 10000011 。

那么，这里的 00000011 和 10000011 就是机器数。

真值

因为第一位是符号位，所以机器数的值就不等于真正的数值。例如上面的有符号数 10000011，其最高位1代表负，其真正数值是 -3 而不是形式值131（10000011转换成十进制等于131）。所以，为区别起见，将带符号位的机器数对应的真正数值称为机器数的真值。

对于一个数, 计算机要使用一定的编码方式进行存储. 原码, 反码, 补码是机器存储一个具体数字的编码方式.

原码

原码是人脑最容易理解和计算的表示方式.

原码就是符号位加上真值的绝对值, 即用第一位表示符号, 其余位表示值. 比如如果是8位二进制:

[+1]原 = 0000 0001

[-1]原 = 1000 0001

反码

正数的反码是其本身

负数的反码是在其原码的基础上, 符号位不变，其余各个位取反.

[+1] = [00000001]原 = [00000001]反

[-1] = [10000001]原 = [11111110]反

可见如果一个反码表示的是负数, 人脑无法直观的看出来它的数值. 通常要将其转换成原码再计算.

补码

正数的补码就是其本身

负数的补码是在其原码的基础上, 符号位不变, 其余各位取反, 最后+1. (即在反码的基础上+1)

[+1] = [00000001]原 = [00000001]反 = [00000001]补

[-1] = [10000001]原 = [11111110]反 = [11111111]补

负数, 补码表示方式也是人脑无法直观看出其数值的. 通常也需要转换成原码在计算其数值.

有了原码为什么还要有补码？

现在我们知道了计算机可以有三种编码方式表示一个数. 对于正数因为三种编码方式的结果都相同:

> [+1] = [00000001]原 = [00000001]反 = [00000001]补

所以不需要过多解释. 但是对于负数:

> [-1] = [10000001]原 = [11111110]反 = [11111111]补

可见原码, 反码和补码是完全不同的. 既然原码才是被人脑直接识别并用于计算表示方式, 为何还会有反码和补码呢?

首先, 因为人脑可以知道第一位是符号位, 在计算的时候我们会根据符号位, 选择对真值区域的加减. (真值的概念在本文最开头). 但是对于计算机, 加减乘数已经是最基础的运算, 要设计的尽量简单. 计算机辨别"符号位"显然会让计算机的基础电路设计变得十分复杂! 于是人们想出了将符号位也参与运算的方法. 我们知道, 根据运算法则减去一个正数等于加上一个负数, 即: 1-1 = 1 + (-1) = 0 , 所以机器可以只有加法而没有减法, 这样计算机运算的设计就更简单了。于是人们开始探索**将符号位参与运算, 并且只保留加法的方法**. 首先来看原码:

计算十进制的表达式: 1-1=0

> 1 - 1 = 1 + (-1) = [00000001]原 + [10000001]原 = [10000010]原 = -2

如果用原码表示, 让符号位也参与计算, 显然对于减法来说, 结果是不正确的.这也就是为何计算机内部不使用原码表示一个数.

为了解决原码做减法的问题, 出现了反码:

计算十进制的表达式: 1-1=0

> 1 - 1 = 1 + (-1) = [0000 0001]原 + [1000 0001]原= [0000 0001]反 + [1111 1110]反 = [1111 1111]反 = [1000 0000]原 = -0

发现用反码计算减法, 结果的真值部分是正确的. 而唯一的问题其实就出现在"0"这个特殊的数值上. 虽然人们理解上+0和-0是一样的, 但是0带符号是没有任何意义的. 而且会有[0000 0000]原和[1000 0000]原两个编码表示0.

于是补码的出现, 解决了0的符号以及两个编码的问题:

> 1-1 = 1 + (-1) = [0000 0001]原 + [1000 0001]原 = [0000 0001]补 + [1111 1111]补 = [0000 0000]补=[0000 0000]原

使用补码, 不仅仅修复了0的符号以及存在两个编码的问题, 而且还能够多表示一个最低数. 这就是为什么8位二进制, 使用原码或反码表示的范围为[-127, +127], 而使用补码表示的范围为[-128, 127].

因为机器使用补码, 所以对于编程中常用到的32位int类型, 可以表示范围是: [-231, 231-1] 因为第一位表示的是符号位.而使用补码表示时又可以多保存一个最小值.

Linux系统各个目录的一般作用

d9995847888d6fd086d100078b505090.png

目录	说明
/bin	存放二进制可执行文件(ls,cat,mkdir等)，常用命令一般都在这里。
/home	存放所有用户文件的根目录，是用户主目录的基点，比如用户user的主目录就是/home/us
/usr	用于存放系统应用程序，比较重要的目录 /usr/local 本地系统管理员软件安装目录（安装系统级的应用）。这是最庞大的目录，要用到的应用 众多的应用程序 /usr/sbin 超级用户的一些管理程序 /usr/doc linux文档 /usr/include linux下 常用的动态链接库和软件包的配置文件 /usr/man 帮助文档 /usr/src 源代码，linux内核的源 本地增加的库
/opt	额外安装的可选应用程序包所放置的位置。一般情况下，我们可以把tomcat等都安装到这
/proc	虚拟文件系统目录，是系统内存的映射。可直接访问这个目录来获取系统信息。
/root	超级用户（系统管理员）的主目录（特权阶级o）
/sbin	存放二进制可执行文件，只有root才能访问。这里存放的是系统管理员使用的系统级别的管
/dev	用于存放设备文件。
/mnt	系统管理员安装临时文件系统的安装点，系统提供这个目录是让用户临时挂载其他的文件
/boot	存放用于系统引导时使用的各种文件
/lib	存放跟文件系统中的程序运行所需要的共享库及内核模块。共享库又叫动态链接共享库，
/tmp	用于存放各种临时文件，是公用的临时文件存储点。
/var	用于存放运行时需要改变数据的文件，也是某些大文件的溢出区，比方说各种服务的日志
/lost+found	这个目录平时是空的，系统非正常关机而留下“无家可归”的文件（windows下叫什么.chk）

谷歌C++编程规范

命名约定

通用规则

函数命名，变量命名、文件命名要有描述性，少用缩写

文件命名

文件名要全部小写，用下划线(_)连起来，c++文件要以.cc结尾，头文件以.h结尾，专门插入文本的文件以.inc结尾

类命名

类的每个单词首字母均大写，不包含下划线，比如：MyExcitingClass

变量命名

变量名一律小写，单词之间用下划线连接

类的成员变量以下划线结尾

结构体成员变量和类一样

常量命名

在全局或类里的常量名称前加 k: `kDaysInAWeek`. 且除去开头的 k 之外每个单词开头字母均大写。

所有编译时常量, 无论是局部的, 全局的还是类中的, 和其他变量稍微区别一下. k 后接大写字母开头的单词:

```
const int kDaysInAWeek = 7;
```

函数命名

常规函数使用大小写混合，如MyExcitingFunction()

如果您的某函数出错时就要直接 crash, 那么就在函数名加上 OrDie.

取值（Accessors）和设值（Mutators）函数要与存取的变量名匹配，用小写：int num_entries() const { return num_entries_; }

函数参数

跟变量命名一样

宏命名

全部大写，像这样命名: MY_MACRO_THAT_SCARES_SMALL_CHILDREN

总结

Google 的命名约定很高明，比如写了简单的类QueryResult, 接着又可以直接定义一个变量query_result, 区分度很好；再次，类内变量以下划线结尾，那么就可以直接传入同名的形参，比如 TextQuery::TextQuery(std::string word) : word_(word) {} , 其中 word_ 自然是类内私有成员。

海量数据处理题问题

第一题

问题：海量日志数据，提取出某日访问百度次数最多的IP。

答案：假设内存无穷大，我们可以用常规的HashMap(ip，value)来统计ip出现的频率，统计完后利用排序算法得到次数最多的IP，这里的排序算法一般是堆排序或快速排序。但考虑实际情况，我们的内存是有限的，所以无法将海量日志数据一次性塞进内存里，那应该如何处理呢？很简单，分而治之！即将这些IP数据通过Hash映射算法划分为多个小文件，比如模1000，把整个大文件映射为1000个小文件，再找出每个小文件中出现频率最大的IP，最后在这1000个最大的IP中，找出那个频率最大的IP，即为所求（是不是很像Map Reduce的思想？）。

这里再多说一句：Hash取模是一种等价映射算法，不会存在同一个元素分散到不同小文件中的情况，这保证了我们分别在小文件统计IP出现频率的正确性。我们对IP进行模1000的时候，相同的IP在Hash取模后，只可能落在同一个小文件中，不可能被分散的。因为如果两个IP相等，那么经过Hash(IP)之后的哈希值是相同的，将此哈希值取模（如模1000），必定仍然相等。

总结一下，该类题型的解决方法分三步走：

1. 分而治之、hash映射；
2. HashMap（或前缀树）统计频率；
3. 应用排序算法（堆排序或快速排序）。

第二题

问：搜索引擎会通过日志文件把用户每次检索使用的所有查询串都记录下来，每个查询长度不超过 255 字节。假设目前有一千万个记录（这些查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门），请你统计最热门的10个查询串，要求使用的内存不能超过1G。

答：我们首先分析题意：一千万个记录，除去重复后，实际上只有300万个不同的记录，每个记录假定为最大长度255Byte，则最多占用内存为：3M*1K/4=0.75G<1G，完全可以将所以查询记录存放在内存中进行处理。相较于第一道题目，这题还更简单了，直接HashMap（或前缀树）+堆排序即可。

具体做法如下：

1. 遍历一遍左右的Query串，利用HashMap（或前缀树）统计频率，时间复杂度为O(N)，N=1000万；
2. 建立并维护一个大小为10的最小堆，然后遍历300万Query的频率，分别和根元素（最小值）进行对比，最后找到Top K，时间复杂度为N‘logK，N‘=300万，K=10。

第三题

问：有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小是1M。返回频数最高的100个词。

答：经过前两道题的训练，第三道题相信大家已经游刃有余了，这类题型都有相同的特点：文件size很大，内存有限，解决方法还是经典三步走：分而治之 + hash统计 + 堆/快速排序。

具体做法如下：

1. 分而治之、hash映射：遍历一遍文件，对于每个词x，取hash(x)并模5000，这样可以将文件里的所有词分别存到5000个小文件中，如果哈希函数设计得合理的话，每个文件大概是200k左右。就算其中有些文件超过了1M大小，还可以按照同样的方法继续往下分，直到分解得到的小文件的大小都不超过1M；

2. HashMap（或前缀树）统计频率：对于每个小文件，利用HashMap（或前缀树）统计词频；
3. 堆排序：构建最小堆，堆的大小为100，找到频率最高的100个词。

第四题

问：给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url？

答：每个url是64字节，50亿*64=5G×64=320G，内存限制为4G，所以不能直接放入内存中。怎么办？分而治之！

具体做法如下：

1. 遍历文件a中的url，对url进行hash(url)%1000，将50亿的url分到1000个文件中存储（a0，a1，a2.....），每个文件大约300多M，对文件b进行同样的操作，因为hash函数相同，所以相同的url必然会落到对应的文件中，比如文件a中的url1与文件b中的url2相同，那么它们经过hash(url)%1000也是相同的。即url1落入第n个文件中，url2也会落入到第n个文件中。
2. 遍历a0中的url，存入HashSet中，同时遍历b0中的url，查看是否在HashSet中存在，如果存在则保存到单独的文件中。然后以此遍历剩余的小文件即可。

总结

这几道题都有一个共性，**那就是要求在海量数据中找出重复次数最多的一个/前N个数据**，我们的解决方法也很朴实：分而治之/Hash映射 + HashMap/前缀树统计频率 + 堆/快速/归并排序，具体来说就是先做hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数，最后利用堆这个数据结构高效地取出前N个出现次数最多的数据。

HelloWorld程序开始到打印到屏幕上的全过程

1. 用户告诉操作系统执行 HelloWorld 程序（通过键盘输入等）；
2. 操作系统找到 HelloWorld 程序，检查其类型是否是可执行文件；并通过程序首部信息，确定代码和数据在可执行文件中的位置并计算出对应的磁盘块地址；
3. 操作系统创建一个新进程，将 HelloWorld 可执行文件映射到该进程结构，表示由该

进程执行HelloWorld 程序；

4. 操作系统为 HelloWorld 程序设置 cpu 上下文环境，并跳到程序开始处；
5. 执行 HelloWorld 程序的第一条指令，发生缺页异常；然后分配一页物理内存，并将代码从磁盘读入内存，然后继续执行 HelloWorld 程序;
6. HelloWorld 程序执行 puts 函数（系统调用），在显示器上写一字符串;
7. 操作系统找到要将字符串送往的显示设备，通常设备是由一个进程控制的，所以操作系统将要写的字符串 送给该进程;
8. 操作系统控制设备的进程告诉设备的窗口系统，它要显示该字符串，窗口系统确定这是一个合法的操作，然后将字符串转换成像素，将像素写入设备的存储映像区;
9. 视频硬件将像素转换成显示器可接收和一组控制数据信号;
10. 显示器解释信号，激发液晶屏; OK，我们在屏幕上看到了 HelloWorld;

大整数运算和构造

参考连接

构造

可以用一个数组来存储数字的每一位来表示一个大整数。

使用一个类来包装，大整数类中有保存数据的数组，数位长度

表示

在构造一个大整数的时候，我们应该有两个步骤。

1. 把整个数组填充为 0
2. 大部分情况下我们需要构造的整数的各个数位逆序填入数组中

第 1 点比较好理解，因为我们要做加减乘除的时候，肯定需要进位借位，这时候把暂时没有用到的位数设置为 0 是非常合理的。

> 我们所做的大部分运算都是从低位往高位进行的，而且往往会涉及到进位。这时采用逆序保存的方法就会很方便进位操作，反之如果我们按照原始顺序进行保存，想要进位的话，还需要把整个数组往后移动。当然并不是说，任何情况下都需要用这种顺序来保存。但是在做题的时候，往往只会涉及到加法和乘法，偶尔还有减法，几乎不会涉及到除法。这种情况下采用逆序保存就很合适了。

字符编码笔记：ASCII，Unicode 和 UTF-8

ASCII 码

计算机内部，所有信息最终都是一个二进制值。每一个二进制位（bit）有`0`和`1`两种状态，因此八个二进制位就可以组合出256种状态，这被称为一个字节（byte）。也就是说，一个字节一共可以用来表示256种不同的状态。60年代，美国制定了一套字符编码，对英语字符与二进制位之间的关系，做了统一规定。这被称为 ASCII 码，一直沿用至今。ASCII 码一共规定了128个字符的编码

英语用128个符号编码就够了，但是用来表示其他语言，128个符号是不够的。

Unicode

是否有一种编码，将世界上所有的符号都纳入其中。每一个符号都给予一个独一无二的编码，那么乱码问题就会消失。这就是 Unicode，就像它的名字都表示的，这是一种所有符号的编码。Unicode 当然是一个很大的集合，现在的规模可以容纳100多万个符号。

但是Unicode 只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储。

UTF-8

互联网的普及，强烈要求出现一种统一的编码方式。UTF-8 就是在互联网上使用最广的一种 Unicode 的实现方式。

UTF-8 最大的一个特点，就是它是一种变长的编码方式。它可以使用1~4个字节表示一个符号，根据不同的符号而变化字节长度。

编译器优化

O0：编译器默认就是O0，该选项下不会开启优化，方便开发者调试。

O1：致力于在不需要过多的编译时间情况下，尽量减少代码大小和尽量提高程序运行速度，它开启了下面的优化标志：

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions-called-once
-fipa-modref
-fipa-profile
-fipa-pure-const
-fipa-reference
-fipa-reference-addressable
-fmerge-constants
-fmove-loop-invariants
-fomit-frame-pointer
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
```

```
-ftree-pta
-ftree-scev-cprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-ter
-funit-at-a-time
```

O2：常见的Release级别，该选项下几乎执行了所有支持的优化选项，它增加了编译时间，提高了程序的运行速度，又额外打开了以下优化标志：

```
-falign-functions  -falign-jumps
-falign-labels  -falign-loops
-fcaller-saves
-fcode-hoisting
-fcrossjumping
-fcse-follow-jumps  -fcse-skip-blocks
-fdelete-null-pointer-checks
-fdevirtualize  -fdevirtualize-speculatively
-fexpensive-optimizations
-ffinite-loops
-fgcse  -fgcse-lm
-fhoist-adjacent-loads
-finline-functions
-finline-small-functions
-findirect-inlining
-fipa-bit-cp  -fipa-cp  -fipa-icf
-fipa-ra  -fipa-sra  -fipa-vrp
-fisolate-erroneous-paths-dereference
-flra-remat
-foptimize-sibling-calls
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-freorder-blocks-algorithm=stc
-freorder-blocks-and-partition  -freorder-functions
-frerun-cse-after-loop
-fschedule-insns  -fschedule-insns2
-fsched-interblock  -fsched-spec
-fstore-merging
-fstrict-aliasing
-fthread-jumps
-ftree-builtin-call-dce
-ftree-pre
-ftree-switch-conversion  -ftree-tail-merge
-ftree-vrp
```

Os：打开了几乎所有的O2优化标志，除了那些经常会增加代码大小的优化标志：使编译器根据代码大小而不是程序运行速度进行优化，为了减少代码大小。

O3：较为激进的优化选项（对错误编码容忍度最低），在O2的基础上额外打开了十多个优化选项,在O2的基础上又打开了以下优化标志：

```
fgcse-after-reload
-fipa-cp-clone
-floop-interchange
-floop-unroll-and-jam
-fpeel-loops
-fpredictive-commoning
-fsplit-loops
-fsplit-paths
-ftree-loop-distribution
-ftree-loop-vectorize
-ftree-partial-pre
-ftree-slp-vectorize
-funswitch-loops
-fvect-cost-model
-fvect-cost-model=dynamic
-fversion-loops-for-strides
```

Debug和Release版本的区别

Debug通常称为调试版本，通过一系列编译选项的配合，编译的结果通常包含调试信息，而且不做任何优化，以为开发 人员提供强大的应用程序调试能力。

Release通常称为发布版本，是为用户使用的，一般客户不允许在发布版本上进行调试。所以不保存调试信 息，同时，它往往进行了各种优化，以期达到代码最小和速度最优。为用户的使用提供便利。

有以下几个不同：

1. Debug模式下在内存分配上有所区别，在我们申请内存时，Debug模式会多申请一部分空间，分布在内存块的前后，用于存放调试信息。
2. 对于未初始化的变量，Debug模式下会默认对其进行初始化，而Release模式则不会，所以就有个常见的问题，局部变量未初始化时，Debug模式下可能运行正常，但Release模式下可能会返回错误结果

3. Debug模式下可以使用assert，运行过程中有异常现象会及时crash，Release模式下模式下不会编译assert，遇到不期望的情况不会及时crash，稀里糊涂继续运行，到后期可能会产生奇奇怪怪的错误，不易调试，殊不知其实在很早之前就出现了问题。编译器在Debug模式下定义_DEBUG宏，Release模式下定义NDEBUG宏，预处理器就是根据对应宏来判断是否开启assert的。
4. 数据溢出问题，在一个函数中，存在某些从未被使用的变量，且函数内存在数据溢出问题，在Debug模式下可能不会产生问题，因为不会对该变量进行优化，它在栈空间中还是占有几个字节，但是Release模式下可能会出问题，Release模式下可能会优化掉此变量，栈空间相应变小，数据溢出就会导致栈内存损坏，有可能会产生奇奇怪怪的错误。

> **问：有时候程序在Debug模式下运行的好好的，Release模式下就crash了，怎么办？**
>
> 答：看一下代码中是否有未初始化的变量，是否有数组越界问题，从这个思路入手。
>
> **问：有些时候程序在Debug模式下会崩溃，Release模式下却正常运行，怎么办？**
>
> 答：可以尝试着找一找代码中的assert，看一下是否是assert导致的两种模式下的差异，从这个思路入手。

Intel CPU型号解读

Intel生产的CPU分为高中低端，最低端的G系列，然后是低端i3系列，中端i5系列，高端i7系列和至尊i9系列。

U：代表超低电压以15W和28为主

M：代表标准电压cpu

U：代表低电压节能的

H：是高电压的

X：代表高性能

Q：代表至高性能级别

Y：代表超低电压的

K：代表不锁倍频的处理器

“MX”：代表旗舰级，

“HQ”：封装方式FCBGA1364，并且部分支持Trusted Execution Technology和博锐技术，

“MQ”：版本封装方式FCBGA946。

在 4GB 物理内存的机器上，申请 8G 内存会怎么样？

这个问题要考虑三个前置条件：

- 操作系统是 32 位的，还是 64 位的？
- 申请完 8G 内存后会不会被使用？
- 操作系统有没有使用 Swap 机制？

首先，应用程序通过 malloc 函数申请内存的时候，实际上申请的是虚拟内存，此时并不会分配物理内存。当应用程序读写了这块虚拟内存，CPU 就会去访问这个虚拟内存，这时会发现这个虚拟内存没有映射到物理内存，CPU 就会产生缺页中断，进程会从用户态切换到内核态，并将缺页中断交给内核的 Page Fault Handler （缺页中断函数）处理。缺页中断处理函数会看是否有空闲的物理内存：

1. 如果有，就直接分配物理内存，并建立虚拟内存与物理内存之间的映射关系。
2. 如果没有空闲的物理内存，那么内核就会开始进行回收内存 (opens new window)的工作，如果回收内存工作结束后，空闲的物理内存仍然无法满足此次物理内存的申请，那么内核就会放最后的大招了触发 OOM （Out of Memory）机制。

32还是64

另外，32 位操作系统和 64 位操作系统的虚拟地址空间大小是不同的，在 Linux 操作系统中，虚拟地址空间的内部又被分为内核空间和用户空间两部分：

1. `32` 位系统的内核空间占用 `1G`，位于最高处，剩下的 `3G` 是用户空间；
2. `64` 位系统的内核空间和用户空间都是 `128T`，分别占据整个内存空间的最高和最低处，剩下的中间部分是未定义的。

32 位操作系统

因为 32 位操作系统，进程最多只能申请 3 GB 大小的虚拟内存空间，所以进程申请 8GB 内存的话，在申请虚拟内存阶段就会失败

64位操作系统

64 位操作系统，进程可以使用 128 TB 大小的虚拟内存空间，所以进程申请 8GB 内存是没问题的，因为进程申请内存是申请虚拟内存，只要不读写这个虚拟内存，操作系统就不会分配物理内存。

有没有swap

> **什么是 Swap 机制？**
>
> 当系统的物理内存不够用的时候，就需要将物理内存中的一部分空间释放出来，以供当前运行的程序使用。那些被释放的空间可能来自一些很长时间没有什么操作的程序，这些被释放的空间会被临时保存到磁盘，等到那些程序要运行时，再从磁盘中恢复保存的数据到内存中。另外，当内存使用存在压力的时候，会开始触发内存回收行为，会把这些不常访问的内存先写到磁盘中，然后释放这些内存，给其他更需要的进程使用。再次访问这些内存时，重新从磁盘读入内存就可以了。
>
> 将内存数据换出磁盘，又从磁盘中恢复数据到内存的过程，就是 Swap 机制负责的。
>
> Swap 就是把一块磁盘空间或者本地文件，当成内存来使用，它包含换出和换入两个过程：
>
> 1. 换出（Swap Out），是把进程暂时不用的内存数据存储到磁盘中，并释放这些数据占用的内存；
> 2. 换入（Swap In），是在进程再次访问这些内存的时候，把它们从磁盘读到内存中来；
>
> **Linux 中的 Swap 机制会在内存不足和内存闲置的场景下触发**：
>
> 1. 内存不足：当系统需要的内存超过了可用的物理内存时，内核会将内存中不常使用的内存页交换到磁盘上为当前进程让出内存，保证正在执行的进程的可用性，这个内存回收的过程是强制的直接内存回收（Direct Page Reclaim）。直接内存回收是同步的过程，会阻塞当前申请内存的进程。
> 2. 内存闲置：应用程序在启动阶段使用的大量内存在启动后往往都不会使用，通

> 过后台运行的守护进程（kSwapd），我们可以将这部分只使用一次的内存交换到磁盘上为其他内存的申请预留空间。kSwapd 是 Linux 负责页面置换（Page replacement）的守护进程，它也是负责交换闲置内存的主要进程，它会在空闲内存低于一定水位 (opens new window)时，回收内存页中的空闲内存保证系统中的其他进程可以尽快获得申请的内存。kSwapd 是后台进程，所以回收内存的过程是异步的，不会阻塞当前申请内存的进程。
>
> **Swap 换入换出的是什么类型的内存？**
>
> 内核缓存的文件数据，因为都有对应的磁盘文件，所以在回收文件数据的时候， 直接写回到对应的文件就可以了。但是像进程的堆、栈数据等，它们是没有实际载体，这部分内存被称为匿名页。而且这部分内存很可能还要再次被访问，所以不能直接释放内存，于是就需要有一个能保存匿名页的磁盘载体，这个载体就是 Swap 分区。
>
> **swap的优缺点**
>
> 使用 Swap 机制优点是，应用程序实际可以使用的内存空间将远远超过系统的物理内存。由于硬盘空间的价格远比内存要低，因此这种方式无疑是经济实惠的。当然，频繁地读写硬盘，会显著降低操作系统的运行速率，这也是 Swap 的弊端。

使用`free -m`命令查看有没有swap分区

没有开启 Swap 机制

当申请完，使用们memset函数访问的时候，超过了机器的物理内存（2GB），进程（test）被操作系统杀掉了。

通过`var/log/message`可以看到报错了 Out of memory，也就是发生 OOM（内存溢出错误）。

> 什么是 OOM?
>
> 内存溢出(Out Of Memory，简称OOM)是指应用系统中存在无法回收的内存或使用的内存过多，最终使得程序运行要用到的内存大于能提供的最大内存。此时程序就运行不了，系统会提示内存溢出。

开启 Swap 机制

在有 Swap 分区的情况下，即使笔记本物理内存是 8 GB，申请并使用 32 GB 内存是没问题，程序正常运行了，并没有发生 OOM。

但是磁盘 I/O 达到了一个峰值，非常高

> 有了 Swap 分区，是不是意味着进程可以使用的内存是无上限的？
>
> 当然不是，我把上面的代码改成了申请 64GB 内存后，当进程申请完 64GB 虚拟内存后，使用到 56 GB （这个不要理解为占用的物理内存，理解为已被访问的虚拟内存大小，也就是在物理内存呆过的内存大小）的时候，进程就被系统 kill 掉了，当系统多次尝试回收内存，还是无法满足所需使用的内存大小，进程就会被系统 kill 掉了，意味着发生了 OOM

CPU和GPU的区别

https://mp.weixin.qq.com/s/jPh5o5LXDWi7WogyN6AHvQ

C++性能调优

https://www.cnblogs.com/wujianlundao/archive/2012/11/18/2776372.html

冗余的变量拷贝

参数

相对C而言，写C++代码经常一不小心就会引入一些临时变量，比如函数实参、函数返回值。在临时变量之外，也会有其他一些情况会带来一些冗余的变量拷贝。

这个要看要不要修改参数，可能会修改，则需要用值传递无可避免

如果参数不会被函数给修改，那么引用传递可以

返回值

RVO(return value optimization)，这时候只能在函数返回一个未命名变量的时候进行优化。

字符数组的初始化

写代码时，很多人为了省事或者说安全起见，每次申请一段内存之后都先全部初始化为0。

用了一些API，不了解底层实现，把申请的内存全部初始化为0了，比如char buf[1024]=""的方式，

上面提到两种内存初始化为0的情况，其实有些时候并不是必须的。比如把char型数组作为string使用的时候只需要初始化第一个元素为0即可，或者把char型数组作为一个buffer使用的大部分时候根本不需要初始化。

频繁的内存申请、释放操作

https://bbs.csdn.net/topics/330179712

提前计算

这里需要提到的有两类问题：

1. 局部的冗余计算：循环体内的计算提到循环体之前
2. 全局的冗余计算

问题1很简单，大部分人应该都接触到过。有人会问编译器不是对此有对应的优化措施么？对，公共子表达式优化是可以解决一些这个问题。不过实测发现**如果循环体内是调用的某个函数**，即使这个函数是没有side effect的，编译器也无法针对这种情况进行优化。（我是用gcc 3.4.5测试的，不排除更高版本的gcc或者其他编译器可以针对这种情况进行优化）

对于问题2，我遇到的情况是：服务代码中定义了一个const变量，假设叫做MAX_X，处理请求是，会计算一个pow(MAX_X)用作判断（y的x次方），而性能分析发现，这个pow操作占了整体系统CPU占用的10%左右。对于这个问题，我的优化方式很简单，直接计算定义一个MAX_X_POW变量用作过滤即可。代码修改2行，性能提升10%。

空间换时间

哈希表的思想

内联频繁调用的短小函数

小函数尽量内联，频繁调用会提高效率

位运算代替乘除法

%2的次方可以用位运算代替，a%8=a&7（两倍多效率提升）

/2的次方可以用移位运算代替，a/8=a>>3（两倍多效率提升）

`*2`的次方可以用移位运算代替，a*8=a<<3（小数值测试效率不明显，大数值1.5倍效率）

整数次方不要用pow，`i*i`比pow(i,2)快8倍，`i*i*i`比pow快40倍

C++为什么提供move函数？

我想说一下一个我的个人经历

有一段代码，作用是把数据库表保存到XML文件。这个转换的过程，有个中间容器，大概是这样：

```
std::map<string, std::vector<int>> mapTable;
```

可以理解为map的key是数据表的列名，std::vector是那列数据（一行一行的）。

我之前是这么填充的：

```
std::vector<std::variant> vecRow;
for(){
        vecRow.push_back(...);
}
mapTable["列名1"] = vecRow;
```

codereview的时候我的mentor就和我讲了这个事情，本质上上述代码，把vecRow中的所有元素都复制了以便然后放到mapTable中，白白的重新创建了一遍所有行数据，又把不再需要的vecRow释放掉了。这样就很蠢。

改进：当我们知道vecRow生命（作用域后），我们可以利用这个vecRow，在std::move之前，还是有办法的，创建vecRow 的时候就让它是mapTable里某列的引用，如下：

```
std::vector<std::variant> &vecRow = mapTable["列名1"];
for(){
        vecRow.push_back(...);
}
```

但是考虑到这样的话会改动别的代码，所哟用谁提的std::move是最好的

`mapTable["列名1"]= std::move(vecRow);`

就这么一点点改动，就能让vecRow里的东西放进mapTable里，又没避免大规模创建、析构对象。执行完上面的函数，应该会发现vecRow空了。

总结

其实编译器已经在力所能及的优化他能够优化的东西了，但是编译器的优化不是万能的。有时候某个变量的生命周期编译器不可预见，但是我们自己是可以知道的，因此对于这些生命周期很短的变量我们为了节省效率就可以使用move函数。举个例子：比如黄金交易，张三买了李四的黄金，就应该把黄金从李四家移动到张三家里。但如果黄金量很大，移动的成本就会非常高。另一种方式就是大家的黄金都存在银行里，张三买李四的黄金，无非就是账户里的黄金数发生个变化，实体黄金不移动，这样效率就高很多。至于"为什么管理机构（编译器）不优化全世界的黄金交易为纸上黄金交易？"，**那是因为真的有人需要搬黄金回家用啊**