1. 起因

**这些年我偷的那些懒~**

**想总结一下，但是没有大块的时间，一直在构思懒的动笔，或者动笔了但是一次没写完就丢下了，后面再也捡不起来了。**

**想看某个部分的代码，但是没坚持多久就因为各种各样的事情偷懒了一段时间，再次想继续的时候接不上了。**

**怎么系统的建设自己的知识体系？**

**工程的最终目的—解决问题。**

汉光善于总结、去维护组时不要荒废了C语言，由于太过零散边总结边丢弃，并没有形成系统的体系。

得益于github,工作流引擎（JIRA）,goole的浏览器书签，libreoffice draw的画图。

目标：帮助自己更好的成长，规格严格、训练有素的职场人，以及有更多的空闲时间过更有趣的生活；

帮助刚入行的程序员同行们更快的成长。

============================================================================================

从2013年毕业转眼工作接近5年，在小整年的时间节点做个小结，为日复一日的工作加点仪式感。恭喜自己选了这一行，相对有意思的一个行业，相比其他行业有更多按照自己的想法干活的自由，以及社会平均水平以上的收入。

## 1.怎样看代码

代码并不是像小说一样从头到尾顺序写下来的，小说也有倒叙、插叙，但是你从第一个页看到最后一页总能看懂；但是多数代码并不是从第一行看到最后一行就一定能啃下来的，以现在的软件规模也不现实——工作之前多数同学都会对自己的精力有些盲目自信，觉得我可以加班、可以比别人付出更多的时间——但是现实很骨感：大多数人都很努力，而且真的有忙不完的事情，你怎么会有比比人更多的时间；再说，如果把全部空闲时间全部砸在这上面，搞得自己非常疲惫，换来的成长有什么意义。

现在很多网文推崇一万小时定律 -- -- 任何领域投入一万小时就可以成为专家，但是一万小时太恐怖，若每天拿出一小时的空闲时间全年365天不间断，一万小时大约需要三十年，你可能会铆足劲头奔着30年后称为某领域专家一直努力下去，抱歉我是做不到。但是想一下大学的课程，一门课多数集中在30 -- 60学时，也就是说用兴趣和习惯去积累一个月就可以了解一个专业的一门专业课，同样地，在统一个领域的持续投入会有持续的提高。所以重在方法，以及积累 -- -- "坚持"是一个听起来悲催的词。

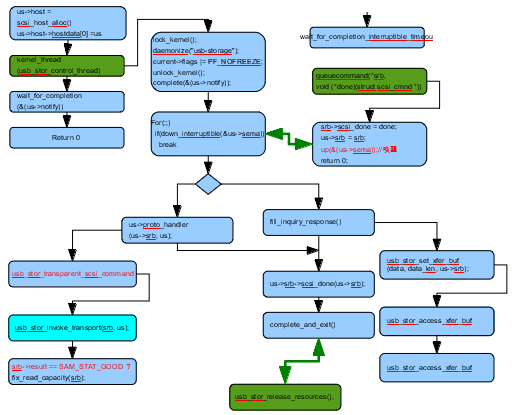
比如需要跟上下游模块交互的代码，以网卡驱动软件为例，上层软件调用驱动软件的接口向远端设备发送数据，数据发送完了这个阶段就完成了，如果埋头看代码那么看到这就断片了——数据发出后是整个过程完成了，还是只是一个阶段完成了？如果只是阶段完成，远端设备会给我回复什么样的数据，处理这些数据的入口在哪？没有对整个通信过程的了解很难理解驱动的代码逻辑。

此外，上下文的衔接也是一个令人头疼的问题：工作中你所要负责的代码通常要耗费一周或者几周的时间才能初步有所理解。如果没有一种很好的方法记录下分析代码的进展，每次分析都从头开始就会重复阅读开始的一部分代码而进展缓慢；在代码分析过程中还涉及到大量相互关联、错综复杂的数据结构：如一些变量的初始化和真正使用并不在一个阶段里面，真正使用的时候早已忘了这个变量被初始化成什么值，这时候再去翻初始化的代码非常耗时，还有可能翻到了初始化的部分又忘记了当前进展。

根据上面两段提出的问题也就知道了高效阅读代码的两个关键点：**一是理清逻辑框架；二是做好代码记录。**

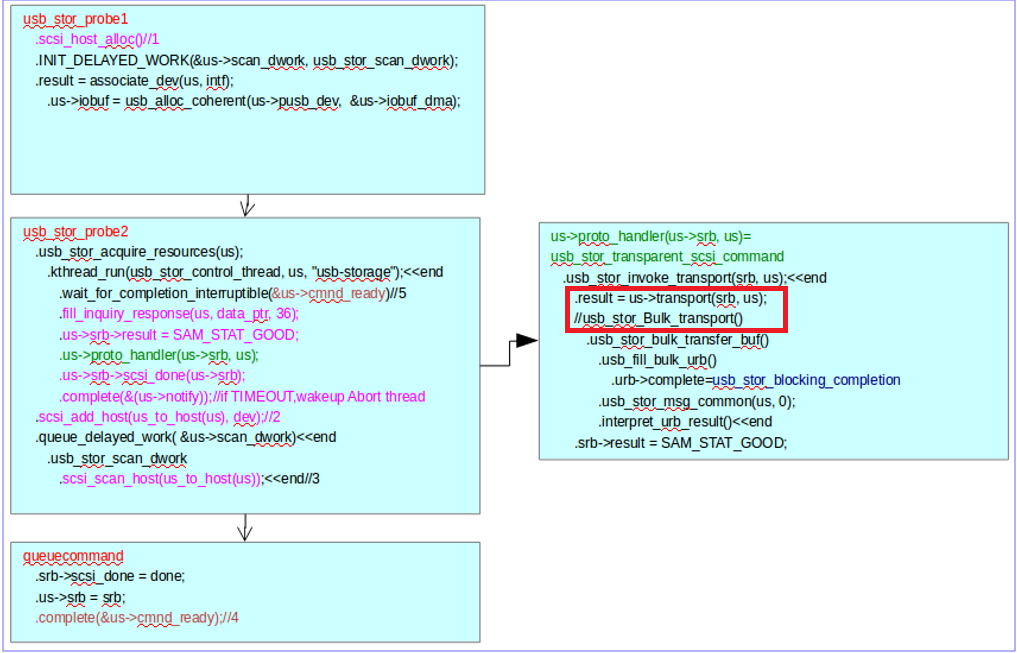
**理清逻辑框架**首先要了解代码和处理逻辑分为几个阶段，再把每个阶段的入口拎出来。完成这两步对于负责的模块就了解了一半；剩下的无非就是了解每个阶段的实现细节，理解有快有慢但总有一天会熟悉。像考古一样阅读代码——先用铁锹挖去周边的泥土露出文物的轮廓，再用铲子清理露出文物的表面，最后用毛刷去除尘土露出文物的纹理。对于逻辑框架的理解可以通过老员工讲解、模块文档、芯片手册、概要设计等方式获取，对于驱动软件开发一般的芯片手册会有对芯片使用场景的介绍或者demo代码，可以了解大致的流程。

**做好代码记录**建议建立两个图，一个是代码执行顺序/调用关系图，一个是数据结构组织图。相信不少人对记录代码执行顺序/调用关系做过尝试，如下图（仅用作示意，看不清内容不必纠结）是以前采用的一种方式，优点是能够清晰的展现出一些流程分支等逻辑细节，缺点是这种方式画图非常耗时，信息密度（一个作图页面能记录的内容）也很低。

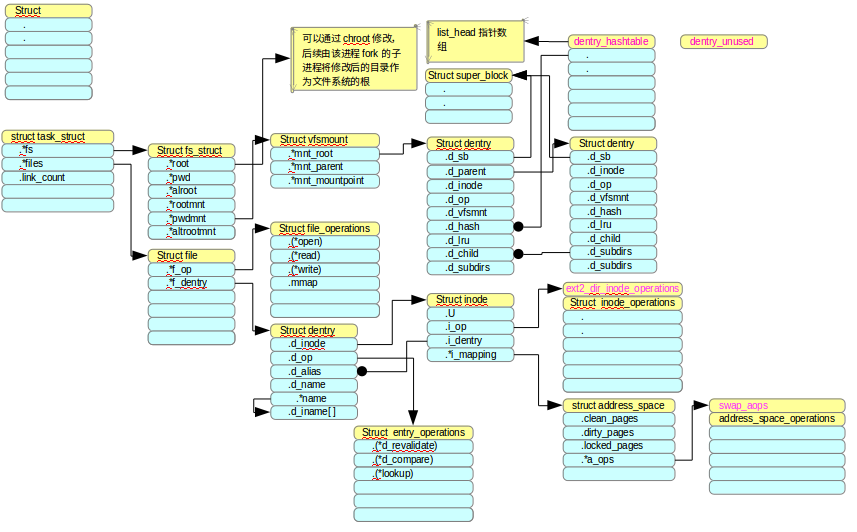


个人习惯用libreoffice draw画图工具记录代码执行顺序/调用关系：自上而下表示顺序执行，通过缩进表示调用关系，缩进层次过深时在右边另起一个区域。遇到函数指针时，在同一级缩进下用注释符号”//”记录实际执行的函数，如下图红色矩形框标注的部分。

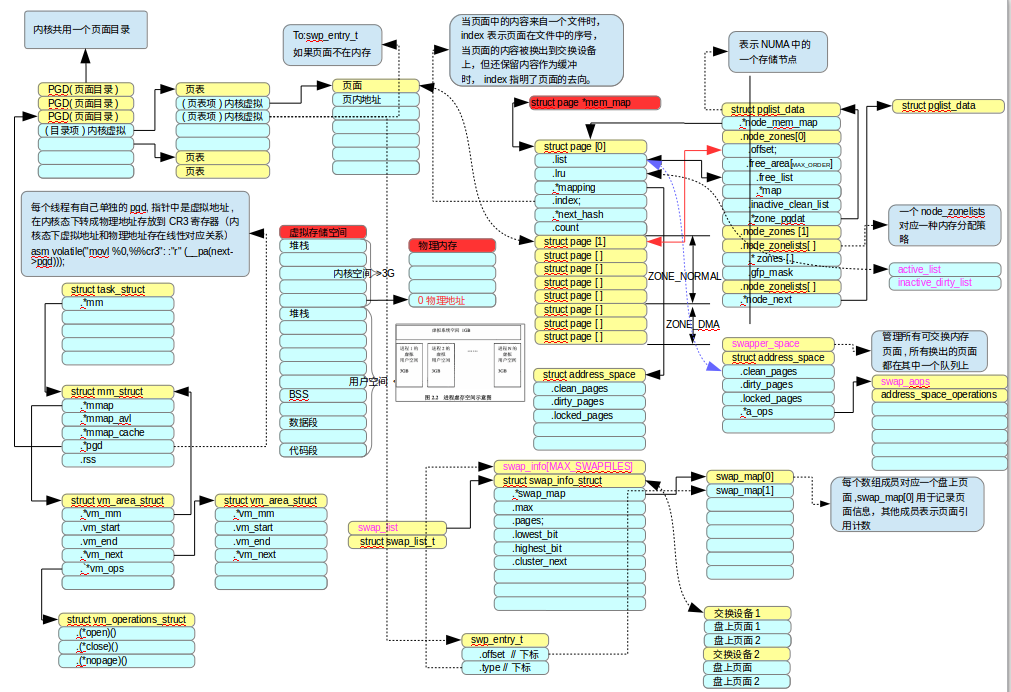
利用这种方式代码的**阶段划分**、**调用关系**以及**变量（如函数指针等）取值**都能清晰的表现出来；其次信息密度较高，通常一个模块仅包含几个主要流程，每个流程用一个框图就能将关键信息展示出来，且可以随时添加关键信息，显著提升阅读代码效率。此外，在团队分享中，可以利用这种方法向其他团队成员介绍你所了解的模块的处理逻辑，同事既能够了解大致流程、又可以根据图中等关键入口函数自己深入分析，可以极大提高团队内知识传递的效率。



阅读代码时通常遇到函数一层层调用，参数一次次传递；在分析最内层函数时早已忘记函数参数的指代对象是什么。数据组织结构图可以快速帮助找到函数参数的指代对象，并形成整个软件结构的整体认识。一个数据组织结构图示例如下图所示，一些重要的数据成员可以添加注释，一些全局变量可以重点标注，帮助理解。



数据组织结构图示例2----内存管理：



## 2.效率提升

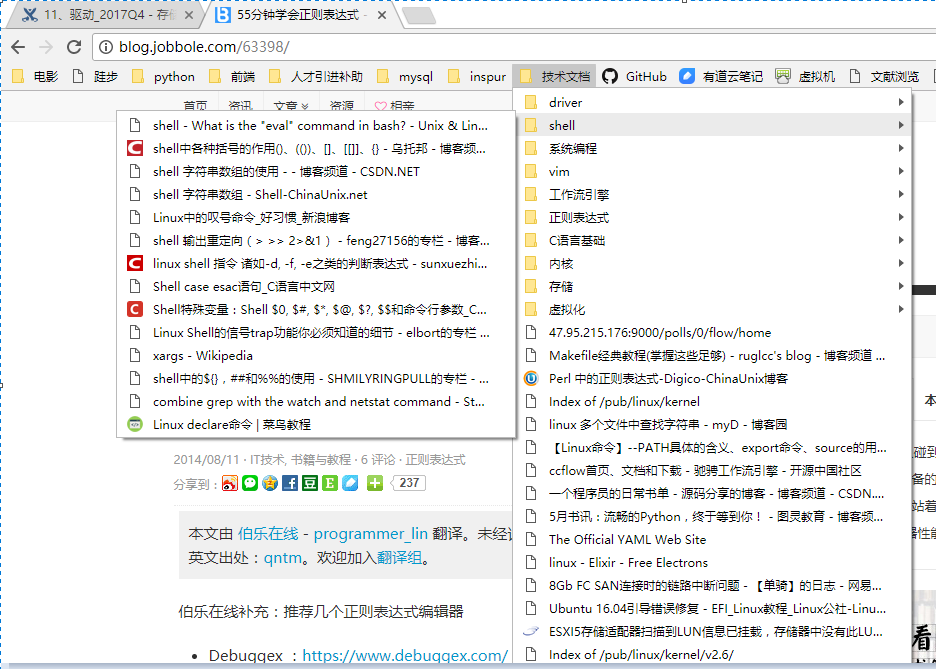
**个人:** 工欲善其事必先利其器，好的工具可以起到事半功倍的效果，个人常用的能够显著提高效率的工具主要有git、libreoffice draw、tasksteper、notepad++、正则表达式、有道云、快速验证方法（7-ZIP、环境变量控制分支等）等，以下是个人的使用心得：

### 2.1利用git构建自己的知识库

git是一个开源的分布式版本控制系统，设计之初是为了管理linux的源代码。除源代码外日常的一些文档资料、技术书籍等也可以使用git来保存管理，逐步完善自己的知识库。

比如可以用git来管理随手记。可能很多人每隔一段时间都想好好总结一下，但是通常要么一直在构思，没有落到纸面上；要么开始写了但是一鼓作气没完成，后面就会不知道丢到哪去再也找不回来了。Git的优点在于记录当前进展和多台设备同步，这样有灵感的时候记录的一些琐碎的事务、杂乱无章的随手记，心血来潮的时候整理一点，随手push一下；到另外一台设备上pull一下就是上次终结的地方，修改完后再随手push一下。慢慢会形成一篇像模像样的东西。比如这篇个人总结就是用了很长的时间，利用git靠一些零零散散的东西堆出来的。

再比如可以用git来管理浏览器书签。为了方便，工作中一些常用的网站通常会保存到浏览器书签中。每次重装电脑、更换电脑、创建新的虚拟机时怎样找回这些网站是一个令人头疼的问题----很多浏览器都提供了账号登录后的书签保存和同步功能。但chrome等浏览器的账号登录在国内不可用，此时就可以定期导出书签备份到github中，重装、更换电脑后从github下载最新的书签文件导入浏览器即可。从个人使用经验来看几周才需要利用git备份一次，并不会耗费太多时间。



最好创建一个自己的小项目并使用git进行维护，尝试一下git push、git pull、git log、git merge、创建分支、版本回退等常用功能。

Git相关教程可以自行搜索“git 廖雪峰”，这是我目前发现的最好的git零基础入门教程，详细地址为：

<https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000>

git的使用受众已经超出了程序员的范畴，知乎上有很多非程序员使用git的讨论，可见它的价值不菲。

### 2.2利用 Libreoffice draw帮助理解代码流程和数据结构组织

上文展示的两张图——代码执行顺序/调用关系图和是数据结构组织图——都是利用libreoffice draw画的。有不少同事看到我做的代码笔记，问过这两张图是怎么画的。

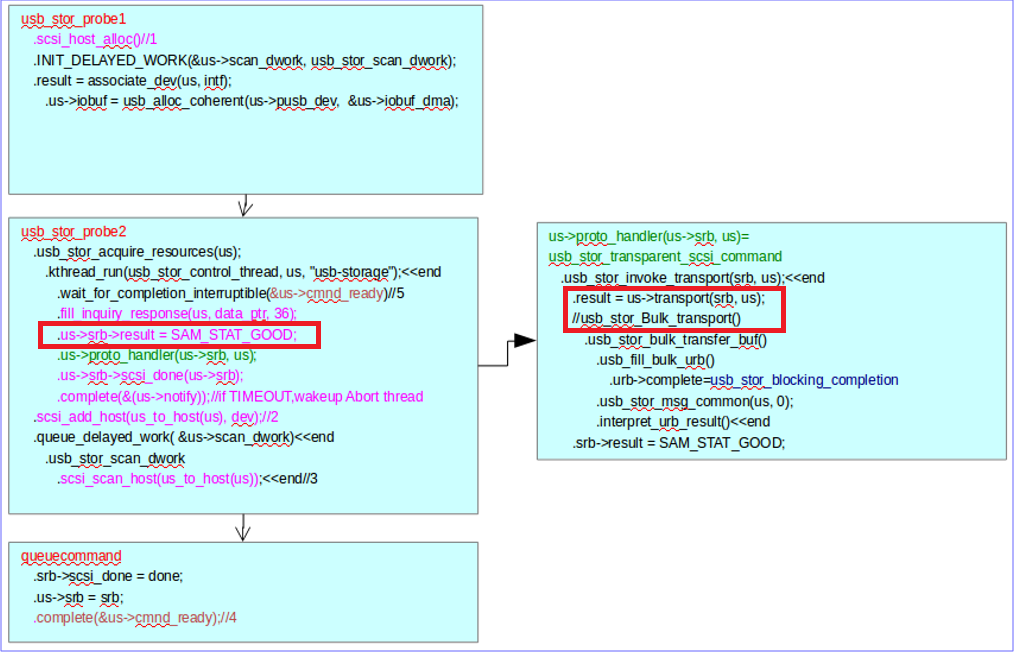
**代码执行顺序/调用关系图**中主要用到的就是一个大矩形，在常用的画图工具（visio，libreoffice draw等）都有现成的组件。在矩形中粘贴关键代码进行记录即可，粘帖的代码遵循一些规则：

1）通过3个空格的缩进表示调用关系，同一级缩进自上而下的代码表示顺序执行。缩进时利用“.”表示正向调用，“-”表示反向调用；缩进层次过深时在右边另起一个矩形区域。反向调用关系在根据日志记录的错误追溯问题源头的时候非常好用，特别是对模块的处理逻辑还不熟悉的时候。

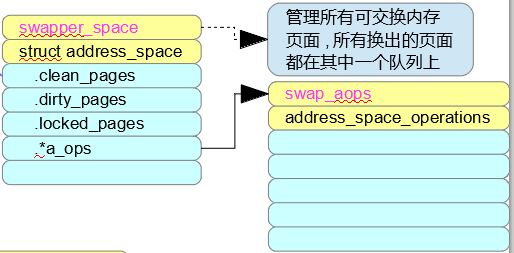
2）一些重要成员的赋值记录下来，比如状态机状态赋值；再比如下图中的usb->srb->result；这样在用到这些变量等时候可以清晰的看出这些关键变量在什么地方被置为什么值，不用再返回去到处翻代码。

3）利用“//”表示对函数指针的解释，比如下图中的us->transport(srb,us)实际调用的函数是usb\_stor\_Bulk\_transport（）

4）重要的部分（下图方框所示）可以用颜色标注、或者添加注释等，长时间之后再次阅读代码时方便理解。



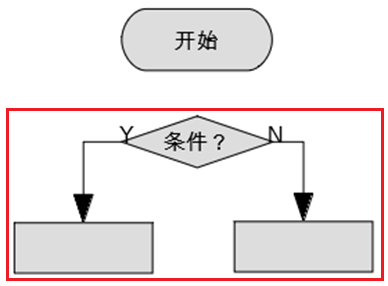
**数据结构组织图**使用多个矩形或者圆角矩形组成结构体，结构体内部通过缩进表示结构体成员的包含关系，多个结构体之间用连接线（也是Visio、libreoffice draw等工具的现有组件）连接，表示关联关系。利用结构体定义的全局变量名可以放在结构体头部，用紫色字体表示。此外还可以添加注释，表明结构体的作用等。



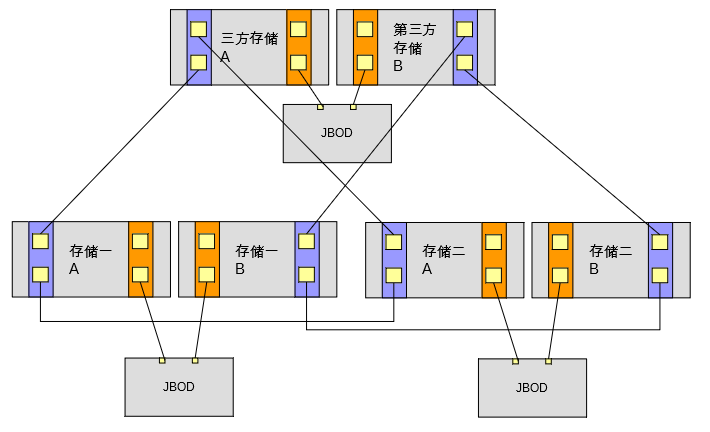
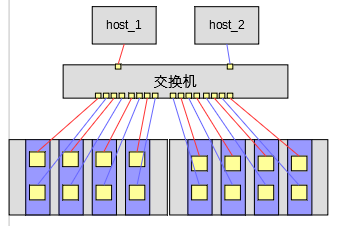
### 2.3利用 Libreoffice draw快速画流程图、组网图

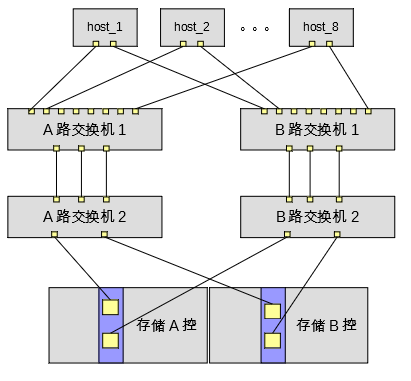
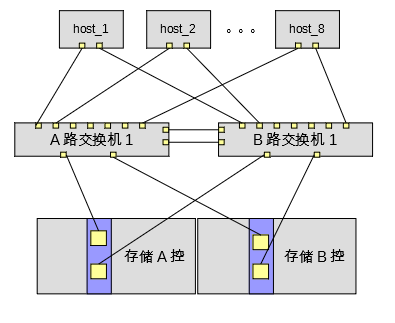
工作中经常需要画各种流程图、组网图等，用于加深理解代码逻辑，帮助分析问题以及向别人表明自己给出的问题处理方案等。以流程图为例，通常利用visio/Libreoffice draw等工具画流程图通常需要寻找适合的组件（流程起始、过程、判断等）、调整背景颜色、字体大小、添加文本并放置到合适的位置（如下图的Y/N）等，通常画完一个稍微简单的流程图需耗费15—30分钟。

但如果保存一个visio/libreoffice draw文件作为画图模板，在画流程图的分支时只需要复制粘贴下图中红框部分，对判断条件略加修改即可。利用这些拼凑流程图，基本一个简单的流程图可以在3分钟左右完成。



分析客户问题或者给出解决方案时，通常需要利用组网图来帮助描述客户现场的设备连接情况。常见的设备无非整机、控制器、卡、端口、连线等；把这些保存到组网图的画图模板中，画新的组网图的时候直接复制黏贴需要是组件，可以极大提高效率。





### 2.4开发过程中的快速测试

**2.4.1 利用7-ZIP替换设备压缩包中的文件进行快速测试**

开发过程中新功能、修改的问题都需要自测，而一些测试需要的前期准备过多，可能造成修改一行代码花费一天测试。不排除有些修改的确需要大量时间验证，有些则不需要——如果你的修改是嵌入到某个流程中，并且影响只影响流程的一个环节，则完全可以想办法进行局部验证。

以硬件的固件升级为例，业界惯用的方法是软件包中放置版本号记录文件（记录期望版本号）和二进制固件文件，在系统启动初期从硬件中读取实际版本号，并与记录文件中的期望版本号进行对比，若二者不一致则利用二进制固件文件对硬件进行升级。

如果发布了新的固件，通常的验证方法是：

1. 修改源码中版本号记录文件中的期望版本号为新固件版本号；
2. 更新源码中的二进制固件文件；
3. 将源码编译、打包成系统软件包；
4. 将系统软件包安装到设备上，等设备启动后验证固件版本是否已更新、硬件是否工作正常。

作为开发人员，如果了解升级流程，完全可以通过在设备上修改版本号记录文件、更换二进制固件文件，然后重启设备的方式进行验证。如果要替换的一些文件在压缩包中，则可以先将设备上的压缩包取出，利用7-zip压缩/解压软件进行替换，再将替换后的压缩包上传到设备。

**2.4.2 利用环境变量控制代码分支进行快速测试**

当前较复杂的软件系统在设计实现时，除了正常的处理逻辑，还需要有大量的错误异常处理需要考虑；实现完成后，针对所有分支的完备覆盖测试是保证软件质量、保证系统在网稳定运行的重要环节。

一些用于异常处理的分支只有当故障等异常发生时才会运行，才能验证异常处理是否有效；但在测试验证过程中许多错误却难以注入。

常用的故障注入手段主要有以下两种：

1.注入真实故障

如在驱动软件的测试过程中，可能需要对驱动相关的硬件进行焊接飞线等操作注入故障；一方面操作复杂、耗时长效率低，另一方面容易损毁研发物料，导致浪费。

2.通过修改代码进行打桩，模拟故障

每次打桩后需要重新编译再安装系统进行测试，且打桩代码写死后就无法进入正常分支；测试完成后需要修改回原来的代码并重新编译以及安装系统，效率低下且容易出错。

如对于以下条件处理分支：

if (pcie\_error\_condition) //条件判断

{

process\_pcie\_error(); //逻辑处理

}

要验证代码中的处理逻辑，要么构造出验证条件，对硬件注入pcie错误；要么代码打桩，直接将pcie\_error\_condition改为TRUE，验证完成后需要再改回去，开发过程中如果有多个条件需要频繁的修改为TRUE/FALSE这种方法就需要多次代码修改、系统构建、系统安装。

这种情况下可以利用环境变量控制代码分支提高验证效率。环境变量是计算机系统中能够影响系统运行方式的一组命名变量。环境变量可以在软件模块外部通过系统的命令行进行修改，软件实现时可以根据环境变量取值进入相应的分支。以下是使用环境变量控制是否进入分支的一个实例：

将上述代码修改为：

if ((pcie\_error\_condition || getenv(“env\_var1”)) &&

getenv(“env\_var2”)) //条件判断

{

process\_pcie\_error(); //逻辑处理

}

则可以通过命令行修改env\_var1，env\_var2这两个环境变量的值来控制是否进入这个分支，变量取值和控制关系如下表所示：

|  |  |  |
| --- | --- | --- |
| env\_var2  env\_var1 | FASE | TRUE |
| FASLE | 不进入 | 维持原逻辑，即  根据pcie\_error\_condition判断 |
| TRUE | 不进入 | 进入 |

如要观察调试process\_pcie\_error()的处理逻辑，只需要执行将env\_var1、env\_var2两个环境变量都改为”TRUE”即可。调试完成后将将env\_var1、env\_var2两个环境变量分别改为”FALSE”、”TRUE”即可恢复原来的处理逻辑，不影响代码的正常运行。

**注：**由于getenv获取到的只是一个字符串，上述代码仅作为示例使用并不能直接运行。要运行需要进行简单处理——如判断通过环境变量的获取的字符串与给定的字符串”TRUE” 是否相等，将环境变量字符串转换为布尔真假。

### 2.5材料归档

如果涉及的工作内容较多，经常会查询一些历史问题、材料等，会花费大量时间。问题归档（文件和问题单，【20180203】XX局点XX设备发生XX问题），往来邮件的保存，结论的存档等

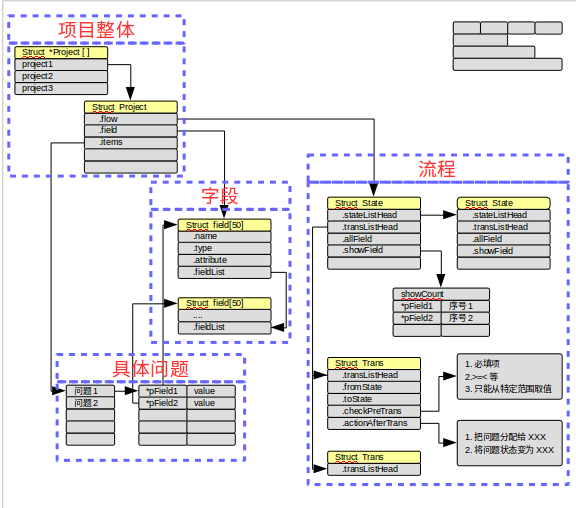
有道云笔记，draw画图记录代码，中断后可以连续，使用git的案例（版本记录）。

Notepad++, 文件搜索，正则表达式，同时过滤多个关键词分析日志

怎样快速查找某次修改的commit\_id?

1.git log [xxxfile];2.过滤出commit xxxxx信息；3.替换，将commit XXXX替换为git show xxxx；4.再次替换将(.\*)替换为echo “\1” >>logfile\n\1 >>logfile 5.执行命令 6.打开logfile用”(commit)|(关键词)”搜索，可以快速找到修改的内容以及对应的commit。

**团队:**版本跟踪记录，标准化运作，像管理bug一样管理每一项日常事务，反对邮件，团队。



## 3.碎片时间利用

2.2聚少成多：零散时间的利用和知识体系建设，网页书签的保存

地铁上用图片《内核设计与实现》

兴趣，每日半小时，python，django，mysql，nginx部署，阿里云，虚拟化

3.娱乐

单车、耳机

4.英语

5.不足

业务场景，存储怎么使用，块、文件、对象存储，虚拟化场景下有哪些可以提升，如文件系统，VAAI特性应该进一步了解。

系统的学习内核；

调试手段欠缺，makefile编写等，慢慢来，比较快

版本记录、保存home/debug用于搭建调试环境等