

第一篇文章就贡献给投票了

如果是讲技术的话，大家都想听什么？如果不讲技术的话，大家都想听什么（八卦除外！）？

啐， 左边的那个就是我！

如何成为牛逼的程序员

我有一个想法，不一定对。

第一篇文章（[第一篇文章就贡献给投票了 - vczh的日常 - 知乎专栏](#)）果然给了我灵感耶，标题的图片就是从评论里截出来的。为了以后回答那些层出不穷的月经问题，我决定写下这么一篇文章，讲一下我对牛逼的程序员的理解。为什么我要讲这个呢，当然首先我还是觉得自己是很牛逼的，不然我就不会讲这个了（误

一个牛逼的程序员和一个不牛逼的程序员的区别到底是什么呢？懂的算法多就牛逼吗？懂的API多就牛逼吗？或者懂的工具多机会牛逼吗？其实牛逼不能用这些简单的指标来定义。我们觉得一个人牛逼，通常指的是那个人懂的东西非常多。不过懂的东西多而写出来的程序很蠢的话，或者半天搞不定一个问题，我们就会开始怀疑我们的判断了。那到底什么是牛逼呢？

其实这就跟聪明区别于傻逼一样——直觉准！

直觉一半来自于举一反三，举一反三一半来自于那个一，而当你对很多问题都有那个一的时候显然是因为你这些问题都碰过，碰过问题跟只学会知识还不一样，你还要解决他们。那如何才能碰过并解决大量的问题呢？唯一的方法就是熬过那一大段时间，通常是十几二十年。你光是聪明，可以学会很多东西，但是由于实践的时间不够，仍然不足以成为一个牛逼的程序员。所以牛逼本身不是一个可以速成的东西，它是知识和经验的积累，然后是运用你的知识和经验的熟练程度。

我一直都跟别人推荐这篇文章（[成年人的思想还能进步么？《学而时嘻之》](#)）讲的就是如何变成一个牛逼的人的。上面说到要变牛逼首先一定要花费这么多的时间。那这么多的时间难道光刷那些傻逼ACM题目就可以变牛逼吗？你刷半年可以，刷十年呢？显然刷ACM题目只能让你从傻逼变成菜鸟，后面还是有很长的路要走的。文章给我们的一个重要的结论就是，你自己在利用这一大段时间练习的时候，每次都要给自己足够难但是又刚好可以做出来的题目来做。等到你把这个题目做出来，你就会觉得很多问题便容易了，这个时候你重复的解决他们只能得到很小的锻炼，于是你要自己寻找更难的题目，一个足够难但是又刚好可以做出来的题目。当然找到一个好的题目也不是那么容易的，反正年轻的时候时间都是不值钱的，等你试图做几个题目发现自己根本做不出来的时候，你就知道什么叫做刚好可以做出来了。

于是逼自己过了这么多年，就算跟我一样整天搞windows，遇到需要用linux的时候也只是问问人看看说明书（不过linux好像没有说明书）的事情了。因为本质困难的东西你都会了，剩下的这些操作问题，只是熟练不熟练的区别而已。

不过到底什么是足够难又刚好可以做出来呢？其实我觉得我小时候编程的学习过程就是很好的一个例子。刚开始自学的时候的确难度是很大，学会了while循环半年后我还总是控制不住自己通过复制代码来做循环的事情，这种感觉就跟哑巴英语一样，你知道那个东西，但是用的时候就是想不起来。当然随着训练的逐步加深，这是可以克服的。

当时我是沿着这么一个路线来走的。首先会用函数画几个图做做模糊啊锐化之类的简单滤镜，其实有算法抄那就是几行代码的实情。然后就开始学习如何写出高性能的程序。自己觉得性能差不多了就开始折腾怎么实现一个RPG。每一步大概都花费了几个月，而且步与步之间的跨越是很大的。当然具体到我当时的情况，难免最后会弄失败，这主要是由于思路的问题，因为没人告诉我要怎么做。我记得很清楚初三的时候做一个RPG，结果在VB6里面试图用Picture控件去搞，不死就怪了。过了两年我终于知道什么是靠谱的方法了，于是就做出了这个（[作品：《天地传》](#)）。这个链接还能下载到我当初的Delphi写的代码，小时候的代码就是烂，尽管看起来也有点复杂了，啊哈哈哈。

游戏做完了不能满足于做完，就要开始想模块化的问题了。怎么做一个游戏用的GUI库？怎么做一个游戏用的脚本引擎？怎么给他们制作工具？怎么写一个游戏引擎？怎么写一个RPG Maker？每一个问题想做简单也可以做简单，想做复杂也可以做的很复杂。于是当我问题一个一个的解决之后，都已经来了MSRA了，这也是我为什么后来会做[GacUI](#)和各种奇怪的编译器研究各种奇怪的类型系统的原因了。当然现在做到这地步都不是仅仅为了游戏而做，当时当你做出一个游戏可以用得东西之后，你要开始想怎么把它做成通用的，使得开发软件也可以用。每一个问题都最终上升了一个台阶，而你觉得容易的问题就不要浪费时间去解决了。

我觉得这应该给大家指出了一个道路，这也是为什么我觉得那些花费那么大精力去研究工具的人很浪费时间的原因了。工作要用的事情就应该占用工作时间去研究，课余的时间当然是花在提高自己的元编程水平上：大概就是算法啊、架构啊、设计模式啊、单元测试啊各种任何语言都用得上的东西了。不过为了训练这些能力，你总得通过真刀真枪的解决什么困难的问题来得到。于是最好的选择就是big clean problem了。这些问题都是定义很清晰但是非常复杂的问题，譬如说我大四尝试并最终成功的一个问题——怎么把C语言编译成x86。当然这只是一个问题，如果你想把它做得实用，要么你要知道怎么写PE文件，要么你要知道怎么跟C++的数据结构和函数指针无缝的结合起来，这就不那么clean了。解决这些边角问题纯粹靠资料，而跟你的水平是没有关系的，如果没有兴趣的话完全没有需要去解决它。当然解决它也不是没有好处的，因为解决了你就弄明白了，你就可以用这些知识来解决你未来的工作里可能会遇到的牛角尖问题了。不过这永远不应该成为你课余学习编程的动力，这就像附加的好处一样。

说到这里我们可以明白，牛逼的程序员，在于它的元编程水平很高的同时，还知道很多奇怪的知识，以便于你在遇到一个真正的工程问题的时候，能正确地在已经掌握的浩如烟海的知识里面联想到正确的那一个小片段，从而经过简短的研究从而立刻解决它。这也是为什么我们觉得牛逼的人知识很多，因为这是一个必要条件。这也是为什么我们觉得牛逼的人写程序很快，因为这是牛逼的结果。

当然对于刚入门的菜鸟来讲，他们还处于连一个工具都没用好的状况，那自然应该花点时间去熟悉工具。不过当你已经掌握了C++、C#、Haskell、Erlang之后再花相当多的精力研究什么Go和Swift，就很划不来了。因为Go和Swift所需要研究的问题其实已经包含在前面的C++、C#、Haskell和Erlang的时间里面了，因此在你使用Go的时候，就应该直接用，万一踩到了坑你跳出来也

是相当容易的事情，根本无需花时间去研究Go的细节。当然对于只会写几行php和python的人，花时间研究Go也是好的，因为他们仍然处于连一个工具都没用好的状况。那自然应该花点时间去熟悉工具。

当你至少掌握了一个general purpose的编程语言和一个系统上的API之后，你就不需要花大量的时间去研究另一个类似的general purpose的编程语言和另一个系统的API了，因为这属于举一反三的内容，只要看文档就可以立刻精通了。倘若是完全不同的两门语言，譬如说C#和Haskell，你学会了一门之后还是可以去花时间研究另一门的。

这些事情都不是绝对的。你需要花时间做什么，取决于这个问题是不是够难，是不是刚刚好你可以做出来，再难一点点你就做不出来了。只要你保持这种训练方法长达十年，想不牛逼都难。

想到这里，不禁回忆起小时候的一些傻逼想法。初三的时候曾经试图用Visual Basic 6.0去做一个Basic的解释器，当然最后做出来了，只是性能巨低，因为我把语法树都保存在表格控件里了……不过在当时的知识下面，能用傻逼方法解决这么个问题，也算是进步了不少。

靠谱的代码和DRY_（图片是GacUI）

做广告就是爽！ www.gaclib.net

上次有人来要求我写一篇文章谈谈什么代码才是好代码，是谁我已经忘记了，好像是AutoHotkey还是啥的专栏的作者。撇开那些奇怪的条款不谈，靠谱的代码有一个共同的特点，就是DRY。DRY就是Don't Repeat Yourself，其实已经被人谈了好多年了，但是几乎所有人都会忘记。

什么是DRY (Don't Repeat Yourself)

DRY并不是指你不能复制代码这么简单的。不能repeat的其实是信息，不是代码。要分析一段代码里面的什么东西时信息，就跟给物理题做受力分析一样，想每次都做对其实不太容易。但是一份代码总是要不断的修补的，所以在这之前大家要先做好TDD，也就是Test Driven Development。这里我对自己的要求是覆盖率要高达95%，不管用什么手段，总之95%的代码的输出都要受到检验。当有了足够多的测试做后盾的时候，不管你以后发生了什么，譬如说你发现你Repeat了什么东西要改，你才能放心大胆的去改。而且从长远的角度来看，做好TDD可以将开发出相同质量的代码的时间缩短到30%左右（这是我自己的经验值）。

什么是信息

信息这个词不太好用语言下定义，不过我可以举个例子。譬如说你要把一个配置文件里面的字符串按照分隔符分解成几个字符串，你大概就会写出这样的代码：

```
// name;parent;description
void ReadConfig(const wchar_t* config)
{
    auto p = wcschr(config, L';'); // 1
    if(!p) throw ArgumentException(L"Illegal config string"); // 2
    DoName(wstring(config, p)); // 3
    auto q = wcschr(p + 1, L';'); // 4
    if(!q) throw ArgumentException(L"Illegal config string"); // 5
    DoParent(wstring(p + 1, q)); // 6
    auto r = wcschr(q + 1, L';'); // 7
    if(r) throw ArgumentException(L"Illegal config string"); // 8
    DoDescription(q + 1); // 9
}
```

这段短短的代码重复了多少信息？

- 分隔符用的是分号（1、4、7）
- 第二/三个片段的第一个字符位于第一/二个分号的后面（4、6、7、9）
- 格式检查（2、5、8）
- 异常内容（2、5、8）

除了DRY以外还有一个问题，就是处理description的方法跟name和parent不一样，因为他后面再也没有分号了。

那这段代码要怎么改呢？有些人可能会想到，那把重复的代码抽取出一个函数就好了：

```
wstring Parse(const wchar_t& config, bool end)
{
    auto next = wcschr(config, L';');
    ArgumentException up(L"Illegal config string");
    if (next)
    {
        if (end) throw up;
        wstring result(config, next);
        config = next + 1;
        return result;
    }
    else
    {
        if (!end) throw up;
        wstring result(config);
        config += result.size();
        return result;
    }
}

// name;parent;description
void ReadConfig(const wchar_t* config)
{
    DoName(Parse(config, false));
    DoParent(Parse(config, false));
}
```

```
    DoDescription(Parse(config, true));
}
```

是不是看起来还很别扭，好像把代码修改了之后只把事情搞得更乱了，而且就算config对了我们也会创建那个up变量，就仅仅是为了不重复代码。而且这份代码还散发出了一些不好的味道，因为对于Name、Parent和Description的处理方法还是不能统一，Parse里面针对end变量的处理看起来也是很重复，但实际上这是无法在这样设计的前提下消除的。所以这个代码也是不好的，充其量只是比第一份代码强一点点。

实际上，代码之所以要写的好，之所以不能repeat东西，是因为产品狗总是要改需求，不改代码你就要死，改代码你就要加班，所以为了减少修改代码的痛苦，我们不能repeat任何信息。举个例子，有一天产品狗说，要把分隔符从分号改成空格！一下子就要改两个地方了。description后面要加tag！这样你处理description的方法又要改了因为他是以空格结尾不是0结尾。

因此针对这个片段，我们需要把它改成这样：

```
vector<wstring> SplitString(const wchar_t* config, wchar_t delimiter)
{
    vector<wstring> fragments;
    while(auto next = wcschr(config, delimiter))
    {
        fragments.push_back(wstring(config, next));
        config = next + 1;
    }
    fragments.push_back(wstring(config));
    return fragments; // C++11就是好!
}

void ReadConfig(const wchar_t* config)
{
    auto fragments = SplitString(config, L';');
    if(fragments.size() != 3)
    {
        throw ArgumentException(L"Illegal config string");
    }
    DoName(fragments[0]);
    DoParent(fragments[1]);
    DoDescription(fragments[2]);
}
```

我们可以发现，分号（L';）在这里只出现了一次，异常内容也只出现了一次，而且处理name、parent和description的代码也没有什么区别了，检查错误也更简单了。你在这里还给你的Library增加了一个SplitString函数，说不定在以后什么地方就用上了，比Parse这种专门的函数要强很多倍。

大家可以发现，在这里重复的东西并不仅仅是复制了代码，而是由于你把同一个信息散播在了代码的各个部分导致了很多相近的代码也散播在各个地方，而且还不是那么好通过抽成函数的方法来解决。因为在这种情况下，就算你把重复的代码抽成了Parse函数，你把函数调用了几次实际上也等于重复了信息。因此正确的方法就是把做事情的方法变一下，写成SplitString。这个SplitString函数并不是通过把重复的代码简单的抽取成函数而做出来的。**去掉重复的信息会让你的代码的结构发生本质的变化。**

这个问题其实也有很多变体：

- 不能有Magic Number。L';'出现了很多遍，其实就是一个Magic Number。所以我们要给他个名字，譬如说delimiter。
- 不要复制代码。这个应该不用我讲了。
- 解耦要做成正交的。SplitString虽然不是直接冲着读config来的，但是它反映了一个在其它地方也会遇到的常见的问题。如果用Parse的那个版本，显然只是看起来解决了问题而已，并没有给你带来任何额外的效益。

信息一旦被你repeat了，你的代码就会不同程度的出现各种腐烂或者破窗，上面那三条其实只是我能想到的比较常见的表现形式。这件事情也告诉我们，当高手告诉你什么什么不能做的时候，得想一想背后的原因，不然跟封建迷信有什么区别。

回文C++_--休闲--

```
/***
#include <iostream>
using namespace std;
int main()
{
    cout<<"Hello, world!"<<endl;
    return 0;
}
/**/
/* */
}
;0 nruter
;ldne<<"!dlrow ,olleH"<<touc
{
) (niam tni
;dts ecapseman gnisu
>maertsoi< edulcni#
*///
```

转眼间GacUI已经写了三年

今天跟高富帅的在CMU读phd的前途无量的正在找妹纸的高中师弟兼大学师弟兼前MSRA同事[@Yong He](#)聊起编程的激情，于是我便开始找起GacUI的第一份checkin到底在哪，翻了好久终于让我翻到了。

想当年，因为要实现类型推导做IDE，但是搞来搞去发现C++的GUI都不顺手，C#写起compiler又不顺手，觉得WPF有太复杂不好速成，于是想通过在C++山寨一个WPF地方法直接解决这三个问题。我很早就试图在弄GUI库了，但是无奈一直不得要领，后来发现只要跟WPF设计的不一样就会直接作出渣。因此在失败了7次之后。我终于在没有看过WPF的代码的情况下，拿出了第八份设计，也就是今天GacUI的雏形。不过要注意的是，第一次是2003年给我的RPG写的GUI，当然那会儿才不做什么IDE。不过要是没有那一次的经验，我估计以后根本就不会试图去搞GUI库。

[Ex] [Vczh Library++](#)

commit 81017

Committed by [vczh](#) on 十月 05, 2011.

```
delete Candidate/GUI/GUI/GuiApplication.h
delete Candidate/GUI/GUI/GuiButton.cpp
delete Candidate/GUI/GUI/GuiButton.h
delete Candidate/GUI/GUI/GuiContainer.cpp
delete Candidate/GUI/GUI/GuiContainer.h
delete Candidate/GUI/GUI/GuiControl.cpp
delete Candidate/GUI/GUI/GuiControl.h
delete Candidate/GUI/GUI/GuiGrid.cpp
delete Candidate/GUI/GUI/GuiGrid.h
delete Candidate/GUI/GUI/GuiWindow.cpp
delete Candidate/GUI/GUI/GuiWindow.h
edit Candidate/GUI/GUI/NativeWindow/Windows/GDI/WinGDIApplication.cpp (View full diff)
edit Candidate/GUI/GUI/NativeWindow/Windows/GDI/WinGDIApplication.h (View full diff)
delete Candidate/GUI/GUI/NativeWindow/Windows/GDI/WinGDISimpleElement.cpp
delete Candidate/GUI/GUI/NativeWindow/Windows/GDI/WinGDISimpleElement.h
add Candidate/GUI/GUI/Previous.rar
edit Candidate/GUI/GuiDemo/GuiDemo.vcxproj (View full diff)
edit Candidate/GUI/GuiDemo/GuiDemo.vcxproj.filters (View full diff)
edit Candidate/GUI/GuiDemo/GuiDemo/Main.cpp (View full diff)
delete Candidate/GUI/GuiDemo/GuiDemo/Main_GuiApplication_Window.cpp
delete Candidate/GUI/GuiDemo/GuiDemo/Main_NativeWindow_GuiSimple.cpp
```

当时还没有什么<http://gac.codeplex.com>。<http://vlpp.codeplex.com>一直被我用来做各种我正在学习的编译原理和编程语言的各种概念的实验室。Candidate目录下还有很多奇怪的子项目，当然都跟编译是无关的，因为我的爱好实在是太广泛了。

从这个checkin可以看到，我含着泪光把第七次的成果给rar了之后，留下仅有的几个看起来还能用的文件，开始了GacUI的新设计，时间是2011年10月5日。不过这次终于让我找到了正确的做法，于是过了差不多半年，GacUI的雏形终于出来了。于是有了现在的codeplex账号。

[Gac Library -- C++ Utilities for GPU Accelerated GUI and Script](#)

截图我就不贴了。从这里可以看到那会儿的GacUI的文件还很少，文件的安排也很随意，跟今天的规模肯定完全不同。

维护一个开源项目是有意义的，因为这会锻炼你很多事情。因为首当其冲的要求就是，你要让大家都知道如何方便的编译这个东西。我觉得这是体现程序员节操的一个具体的细节。如果一份开源的代码搞到编译起来都那么苦逼，那这简直就是一个噱头，没打算让你用的。当然这并不是说所有人都不管一看目录就知道怎么编译，还是得有instruction的。但是这个instruction收到的环境的影响应该很少，譬如说GacUI，只要你装了VS2013，不管是哪个版本，都可以按照下面的方法简单粗暴的建立起Release Folder：

- 用 `DebugNoReflection` 的配置（就跟Debug和Release一样，那是我自己创建的一个配置）来编译 `Libraries\GacUI\GacUISrc\GacUISrc\GacUISrc.vcxproj`
- 打开 `Visual Studio Command Prompt`，在 `Libraries\GacUI\Public` 目录下运行 `Release.bat`

你会发现在这份bat里面，已经帮你做完了所有事情，譬如说把几百个GacUI的源码打包成10个（就是你们在<https://github.com/vcjh/gac>看到的那几个巨大的文件），然后编译工具链，整理到一起，读pdb抽元数据，跟XML注释混合成html的文档，然后全部打包起来很漂亮的放在`Library\GacUI\Public\Temp\ReleaseFolder`里面。我每次出了新版本，就跑一下这个bat，把这个文件夹里面的东西跟github里面的替换掉，然后就这么push上去了。

如果大家对自己的项目也有足够高的要求的话，你们会发现给你们的库制造这些周边的东西，可以让你们学到很多具体的编程知识。这跟光刷题目光做几个库的感觉是完全不同的。你把自己放在库的用户的角度，然后怎么让他们能很快的上手，然后来

做这些工具链，本身就是一项不小的工程。

当然，一个好的库，首先要满足 **对修改封闭，对扩展开放** 这个原则。这句话大家在设计模式的课本里面都经常听见，这也是我下一篇文章要讲的主题。

现在的驾校真是用心良苦啊

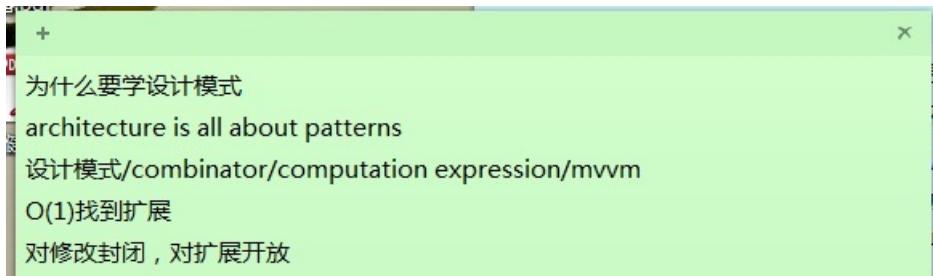
早上去驾校练科目三，刚把车开出来，就目睹了一辆小车在我的旁边强行变道，然后被公共汽车给撞了，漆就像雪花一样飘散开来。

这是不是教练安排来吓我们的。

再水一篇，下一篇一定会讲编程的事情。

在知乎回答了那么多干货，结果这些东西赞最多，当初把知乎定位为娱乐网站果然是没有错的.....

其实提纲都列好了，就是不好下手，啊哈哈哈哈。



为什么我们需要学习（设计）模式

先说点题外话。最近想做一个Computational Graph Database (https://github.com/vcjh/vcjh_to_ye/blob/master/PlayWithLinux/Database/draft)，顺便练习一下Linux下写C++程序的技巧，深刻的体会到了一个道理。Linux下有这么多烂工具，每个人做了一个工具，都会有另外一个人觉得这个工具很烂，然后就做了一个更烂的来恶心他。我本着这种开源的精神，写了一个GayMake（误，于是就有了这次的贴图。倘若最后这个项目没有坑，以后开源狗跟我吵架，我就可以糊他一脸，说【在linux瞎搞编程你也搞不过我】，啊哈哈哈哈。

不管是设计模式也好，别的模式也要，他都是为了解决问题而发明的有效的方法。除了我们已经熟悉的23种设计模式以外，还有**MVVM**、**Combinator**等其它的东西，都已经是前辈们经过多年的摸爬滚打总结出来的，其有效性不容置疑。我这篇文章也不会用来证明设计模式是有用的，因为在我看来，这就跟 $1+1=2$ 一样明显（在黑板上写下 $1+1=2$ ）。

架构的设计也是一样。你要做一个分布式系统，在哪里放gate way，在哪里放database，在哪里放cache，在哪里放计算节点，这些东西都已经是早就总结好的了。类似的东西就叫pattern。一个architecture就是由很多个pattern组合起来的。除此之外，做游戏也好，做编译器也好，设计数据库也好，每一大类的问题都有他们自己的pattern。他们的档次跟设计模式不一样，但是要解决的问题都是一样的，就是让你高效地解决问题。

那为什么我们需要学习设计模式呢？这很明显，这就跟我们看别人的代码来学习一样，是为了学习里面的精髓。每一本设计模式的书都会告诉你，这些都是在讲究如何对修改封闭，对扩展开放的事情。前几天我在看几年前一个数学老师的公开课，叫数学大观（[数学大观 - 专辑](#)）。我觉得里面有句话就讲得很好。我们学东西，重要的是学idea，次要的是学technique。翻译成编程的语言就是，我们学设计模式，是为了学习如何合理的组织我们的代码，如何解耦，如何真正的达到对修改封闭对扩展开放的效果，而不是去背诵那些类的继承模式，然后自己记不住，回过头来就骂设计模式把你的代码搞复杂了，要反设计模式。不要见什么都反，有空应该好好读一读毛泽东的辩证法，知道设计模式有好的一面也有不好的一面，当然总的来说设计模式好的一面比较大。

设计模式要是真的学会了，你们会发现在写代码的时候，脑子里根本没有什么设计模式，你都已经融会贯通了。代码写完了一看，这里有模式，那里也有模式。这就如同我们讲话不会去考虑语法，但是说出来大部分的话都是符合语法要求的。这也如同我们写程序的时候不会总是去想程序的语法的问题，我们自然而然写出来的东西就是可以编译的。道理都是一样的。

不过为了合理的利用设计模式，我们应该明白一个概念，叫做**扩展点**。扩展点不是天生就有的，而是设计出来的。我们设计一个软件的架构的时候，我们也要同时设计一下哪些地方以后可以改，哪些地方以后不能改。倘若你的设计不能满足现实世界的需要，那你就要重构，把有用的扩展点加进去，把没用的扩展点去除掉。这跟你用不用设计模式没关系，跟你对具体的行业的理解有关系。

倘若你设计好了每一个扩展点的位置，那你就可以在每一个扩展点上应用设计模式，你就不需要去想到底这个扩展点要怎么实现他才会真正成为一个扩展点，你只需要按照套路写出来就好了。

如果你发现你最后的代码长得跟设计模式不一样，这不一定代表你没有用到设计模式，也不一定代表这个设计模式没有用。**设计模式归根结底就是因为你使用的程序语言的抽象能力不足才发明出来的**。譬如说那个Listener模式，在C#里面就是一个event关键字搞定，你不需要去写一大堆框架代码来增加这个扩展点。相反，你在Java里面就需要这么做。因此你可能觉得Listener模式在Java有用，在C#没用，其实不是这样的。

真正的情况是Anders Hejlsberg很牛逼，他帮你把这个设计模式做进了语法，你不需要痛苦地写一大堆框架代码就可以用了。这种东西就叫语法糖。有语法糖就不需要写框架代码抄设计模式，没有语法糖你就需要写框架代码抄设计模式。为了解决这一个问题，你所需要放进的扩展点，无论你用什么语言，他都是这么多。区别只有你到底是怎么写出来的。所以语法糖好不好，当然好。你想学语法糖就老老实实抄设计模式，人家也没有把语言设计成你学不会语法糖就不能用。这很公平。

我为什么喜欢Haskell、C++、F#这样的语言，因为它们的代码是可以被计算出来的，因此我可以根据需要随时添加我自己的语法糖，而且还不需要改变语言的语法。当然根据这条标准，我本应该喜欢Lisp的，无奈Lisp的括号太多，噪音太大，我不喜欢这样的东西。同样的理由也见GayMake。那么简单的东西要生成一大坨参杂了各种符号的字符的makefile，那个makefile根本看不懂，虽然已经很接近我手写出来的样子了。其实我就是看了一遍makefile的说明，然后手写了几个makefile，然后照着我写出来的makefile把makefile生成器给写出来的。

设计模式的好处还有一点，就是他做出来的样子性能很高。虽然扩展点的意思就是我在编译的时候不知道到底会命中哪个扩展，但是大多数扩展都是O(1)命中的（除了责任链模式）。与之相关的还有IoC，也就是**Inverse of Control**，这也是一个好东西。Inverse of Control讲的是，类与类之间的依赖是可扩展的，而且是强类型的，并且你还不需要在类的内部指定（因此跟组合不一样）。当一个类需要用到他的依赖的时候，他不需要主动去获取他，而是可以等别人把依赖塞给他，然后再做事情。类似的事情在GacUI（www.gaclib.net）里面大量的使用。

讲了这么多好处，那到底我们要怎样才能学会设计模式呢？答案只有一个，就是创造条件去使用设计模式。很多人总是觉得，要通过简单的程序和例子来学设计模式。这是不对的。设计模式就是因为情况复杂了所以才会出现的，所以我们只能通过复杂的程序来学习设计模式。你不管看别人的程序也好，自己写程序练习也好，那必须要复杂，复杂到你不用设计模式就做不下去，这才能起到学习设计模式的作用。

为什么我对设计模式那么熟悉，这跟我长期以来造的轮子都很复杂是有关系的。不过我有很多设计模式的知识和体会是在搞各种各样的奇怪的语言，譬如说Haskell啊Prolog这些东西才学到的。为什么呢？有些模式就是从那个语言出来的。你倘若不去搞

一搞那个语言，你就不会去看到那个语言的材料，你就不知道那个模式可以被用到你自己喜欢的语言上面。你可能会想，那总有人会这么做啊，譬如说我是吧。但是我是不会把应用到正常语言的过程详细的搞出来给你们听的。原因在于，那个模式，譬如说Combinator，在Haskell下面搞出来很容易，但是在C++里面搞出来就会有大量的框架代码。所以你去看Haskell如何用Combinator，肯定比看C++如何用Combinator难度要低得多。所以倘若你们问我怎么搞定Combinator，我只会告诉你，好好弄Haskell，然后去看Combinator。因为用C++解释太麻烦了，我不会这么做的。

当然，你们有些人肯定不会因为这样就去看Haskell，这只能说明学习Combinator对你来说不是一个迫切的事情。你没了Combinator还能活下去，只是你永远也体会不到你的程序用Combinator可以简化的同时还容易扩展到什么程度。

最后讲一点，Dog fooding是很重要的。微软的东西为什么这么复杂还能继续开发下去，除了代码组织得好以外，质量也好。这跟我们长期自己使用自己的软件给他们积极地爆bug是有关系的。你们自己学东西也一样。譬如说你学了如何自己实现正则表达式，你就把它实现出来，然后以后就用自己的这个实现来干活。东西用的多，爆出来的bug也就多，你对他的理解也就越深刻。倘若你们只是看一眼文章，自己做个小玩具，甚至连小玩具都不做，就这么过去了，肯定什么也学不会。

现在学习编程的积极性果然下降了

编程

在大学时你对编程付出了多大努力？

[添加评论](#) [分享](#) [邀请回答](#)

25 **vczh**, 专业造轮子 [gaclib.net](#) [修改话题经验](#)

大脸猫、shvee、iLeenHow 等人赞同

平均每天至少八个小时编程，不包含看书时间。 [修改](#)

发布于 20:23 [17条评论](#) [分享](#) [收藏](#) [设置](#)

[查看全部 1 个回答](#)

回答了[这个问题](#)之后，想了想现在的状况，看东西比读书的时候容易分神了，不过这跟我想写的程序越来越多估计也是有关系的。读书的时候能做的东西就那几样，现在能做的东西越来越多，想做的实验都做不完啊。

标题插图是我在差不多2011年的时候，有一段时间不知道要干什么，从而写的一个[软件渲染器](#)，用来消磨时间。

现在知乎上的诡辩术真是越来越成熟了

譬如说，“肯定存在很多牛逼的程序员他的效率是普通程序员的100倍。”

很多掌握了诡辩术的网友不同意，他就会来喷你，说：“你要先定义什么是牛逼。”

当然了，这只是诡辩术，这些网友的修为还是比较不行的，真正牛逼的人会说：“你要先定义什么是存在。”

Linux下面的编译器消耗内存就是大啊，烂爆了。

GacUI一共有10个文件，两个给Windows，另外八个是独立与操作系统的。今天把这所有的文件都在g++和clang++下过了（见[GacLib - Download](#)），结果在编译GacUIReflection.cpp的时候，VC++只要300M内存，clang++要2.6G，g++则高达4g，结果把内存都榨干了。

为了看内存，我分别用了top和htop。top是自带的，htop实在比top好太多了但是要自己装。不是说开源就是让大家都来改进项目让软件越来越好用嘛，但是真实的情况永远是，用A不爽的人，做了一个B，而且大多数情况下B也不好用，双倍恶心了用户。

这还让人怎么写代码。

知乎换了新算法，现在再也不能随便点赞了。

现在点赞多的都是按照赞同和反对的比例来排序的，这个就不说了。这次新的更改是，如果一个人在一个话题下面被点了大量的赞，那这个人的权重就会越高。于是我今天发现了，一个问题的我的答案被点了几百个赞，但大部分都是其实并没有怎么回答过问题的热心粉丝。于是我给一个个位数答案赞了一下，蹭一下跳到我前面了，充分体现了权重的巨大区别……

既然我答题点赞都是为了传播价值观，所以只能减少非热门答案的点赞数量了，不然就适得其反了。当然了，对于我只看不答的题，我还是会热心点赞的（逃

我的强大的台式机终于从中国坐轮船抵达西雅图了

上网本就是上网本，连刷知乎都会导致系统假死，现在终于可以恢复到连续打开300个IE tab，系统仍然丝般顺滑的日子了，啊哈哈。

不过每当我的台式机使用wifi的时候，Ubuntu就上不了网，herodb又要搁浅几天了。刚刚注册了个Quora的账号，以后要把知乎的一部分时间分给Quora了。

GacUI迎来了跨越时代的更新

<http://www.gaclib.net>



The image shows the GacLib website. At the top is a large blue 'G' logo followed by the text 'GacLib' and 'GPU Accelerated C++ User Interface'. Below the logo is a horizontal navigation bar with tabs: 'HOME' (highlighted in blue), 'GETTING', 'DEMONS', 'DOWNLOAD', and 'API'. To the right of the navigation bar is a search icon.

GacUI is a GPU accelerated user interface library for C++ programming language. It's similar to WPF, but some features are limited by C++, such as dependency properties. Here we introduce how to use GacUI.

• Cross Platform Supporting

Currently GacUI can run on **Windows, OSX and some Linux distributions**. If you download source code from the [DOWNLOAD](#) page, you will only get the GacUI version that runs on Windows. In order to run on other operating systems, you should navigate to the following urls. In the future, codes for all platforms will be merged together.

For **OSX**: <https://github.com/darkfall/Gaclib OSX>

For **Linux**: <https://github.com/milizhang/xgac>

GacUI for Windows, OSX and Linux shared most of the code, and in most of the cases, you don't need to rely on the non-shared part of the GacUI, except you want to get benefit from a specific platform.

GacUI for OSX and Linux are under development, but most of the importing parts has already been tested by the authors. If you look into the OSX version, you can see there is an "ios" branch, which enable GacUI to run on iOS. But this is only a concept implementation.



GacUI的DataBinding自己用着都觉得开心

在GacUI正确的抽象了类型和提供了DataBinding之后，写代码就跟WPF一样简单。首先准备好一个ObservableList<T>，然后绑定到DataGrid上，最后Linq随便扔一下搞搞逻辑，简直爽。

1、搞定ViewModel

```
void StudioAddExistingFilesModel::AddFiles(LazyList<WString> fileNames)
{
    FOREACH(WString, fileName, fileNames)
    {
        auto fileRef = MakePtr<StudioFileReference>();
        fileRef->fileName = fileName;
        selectedFiles.Add(fileRef);
    }
}

void StudioAddExistingFilesModel::RemoveFiles(LazyList<int32_t> indices)
{
    FOREACH(int, index, From(indices).OrderBy([](int a, int b){return b - a;}))
    {
        selectedFiles.RemoveAt(index);
    }
}
```

2、写几行简单的事件处理

```
void AddExistingFilesWindow::buttonAdd_Clicked(GuiGraphicsComposition* sender, v1::presentation::compositions::GuiEventArgs& arguments)
{
    if (dialogOpen->ShowDialog())
    {
        GetOperationModel()->AddFiles(dialogOpen->GetFileNames());
    }
}

void AddExistingFilesWindow::buttonCancel_Clicked(GuiGraphicsComposition* sender, v1::presentation::compositions::GuiEventArgs& arguments)
{
}

void AddExistingFilesWindow::buttonOK_Clicked(GuiGraphicsComposition* sender, v1::presentation::compositions::GuiEventArgs& arguments)
{
}

void AddExistingFilesWindow::buttonRemove_Clicked(GuiGraphicsComposition* sender, v1::presentation::compositions::GuiEventArgs& arguments)
{
    GetOperationModel()->RemoveFiles(dataGridFiles->GetSelectedItems());
}
```

3、剩下的都交给XML

```
<xaml version="1.0" encoding="utf-8">
    <!Element ref.Class="ui::AddExistingFileDialog">
        <ref.Parameter Name="ViewModel" Class="vm::IStudioModel1"/>
        <ref.Parameter Name="OperationModel" Class="vm::IStudioAddExistingFilesModel"/>
        <ref.Parameter Name="Action" Class="vm::IAddFileDialogAction"/>
        <Window Text="Add Existing Files" ClientSize="x:480 y:320">
            <att.BoundComposition-set PreferredMinSize="x:480 y:320"/>

            <OpenFileDialog ref.Name="dialogOpen" Title="Add Existing Files">
                <att.Options>FileDialogAllowMultipleSelection|FileDialogFileMustExist|FileDialogPreferenceLinks|FileDialogDirectoryMustExist</att.Options>
                </ OpenFileDialog>

            <Table AlignmentToParent="left:0 top:0 right:0 bottom:0" CellPadding="5">
                <att.Rows>
                    <Cell>
                        <CellOptions>composeType:MinSize</CellOptions>
                        <CellOptions>composeType:Percentage percentage:1.0</CellOptions>
                        <CellOptions>composeType:MinSize</CellOptions>
                    </att.Rows>
                    <att.Columns>
                        <CellOptions>composeType:MinSize</CellOptions>
                        <CellOptions>composeType:MinSize</CellOptions>
                        <CellOptions>composeType:Percentage percentage:1.0</CellOptions>
                        <CellOptions>composeType:MinSize</CellOptions>
                        <CellOptions>composeType:MinSize</CellOptions>
                    </att.Columns>
                <Cell Site="row:0 column:0" Margin="left:4 top:4 right:4 bottom:4">
                    <Label Text="Select Files to Add:"/>
                </Cell>
                <Cell Site="row:1 column:0 columnSpan:5">
                    <BindableDataGrid ref.Name="dataGridFiles" Alt="F" HorizontalAlwaysVisible="false" MultiSelect="true" VerticalAlwaysVisible="false" ItemSource-eval="OperationModel.SelectedFiles">
                        <att.BoundComposition-set AlignmentToParent="left:0 top:0 right:0 bottom:0"/>

                        <att.Columns>
                            <BindableDataColumn Text="File Name" Size="300" ValueProperty="fileName"/>
                            <BindableDataColumn Text="Type" Size="120" ValueProperty="fileFactory"/>
                        </att.Columns>
                    </BindableDataGrid>
                </Cell>
                <Cell Site="row:2 column:0">
                    <Button ref.Name="buttonAdd" Text="Add Files ..." Alt="A" ev.Clicked="buttonAdd_Clicked">
                        <att.BoundComposition-set AlignmentToParent="left:0 top:0 right:0 bottom:0" PreferredMinSize="x:100 y:24"/>
                    </Button>
                </Cell>
                <Cell Site="row:2 column:1">
                    <Button ref.Name="buttonRemove" Text="Remove Selected Files" Alt="R" ev.Clicked="buttonRemove_Clicked">
                        <att.BindedBind-1>[ CData[ dataGridFiles.SelectedItems.Count > 0 ] ]</att.BindedBind>
                        <att.BoundComposition-set AlignmentToParent="left:0 top:0 right:0 bottom:0" PreferredMinSize="x:100 y:24"/>
                    </Button>
                </Cell>
            <Cell Site="row:2 column:3">
```

4、就可以运行了，效果见图题。

这个XML的意思是，首先定义一个打开对话框，然后把ViewModel的SelectedFiles属性绑定进DataGrid，最后在XML说好按按钮的时候要运行的函数。后面ViewModel只要自己去改列表对象，UI会跟着变，而且由于LazyList（我给C++准备的Linq的一部分）的存在，列表对象可以无损耗在参数和返回值到处扔，写起来真是太方便了，完爆所有C++的UI库。

GacUI换皮肤就是方便！

开发到一半，把之前给demo写的xml拖进来一编译，皮肤就全上了！就是这图标要重新换了，适合白色背景的图标不适合黑色背景的。

在开发GacStudio的过程之中，我发现原本GacUI还有很多不完善的地方，本着发现一个就干掉一个的精神，目前GacUI主要的开发工具GacGen.exe已经增加了大量的功能，譬如说帮你给一个窗口生成出框架代码，然后你就可以在事件回调里面写你自己的代码，完了XML你给改了重新生成代码的时候，还会帮你把你的修改merge回去。除了要双击一下bat以外，写事件处理这一条就觉得跟当年用Delphi的感觉没什么区别了。

论为什么知乎是娱乐网站

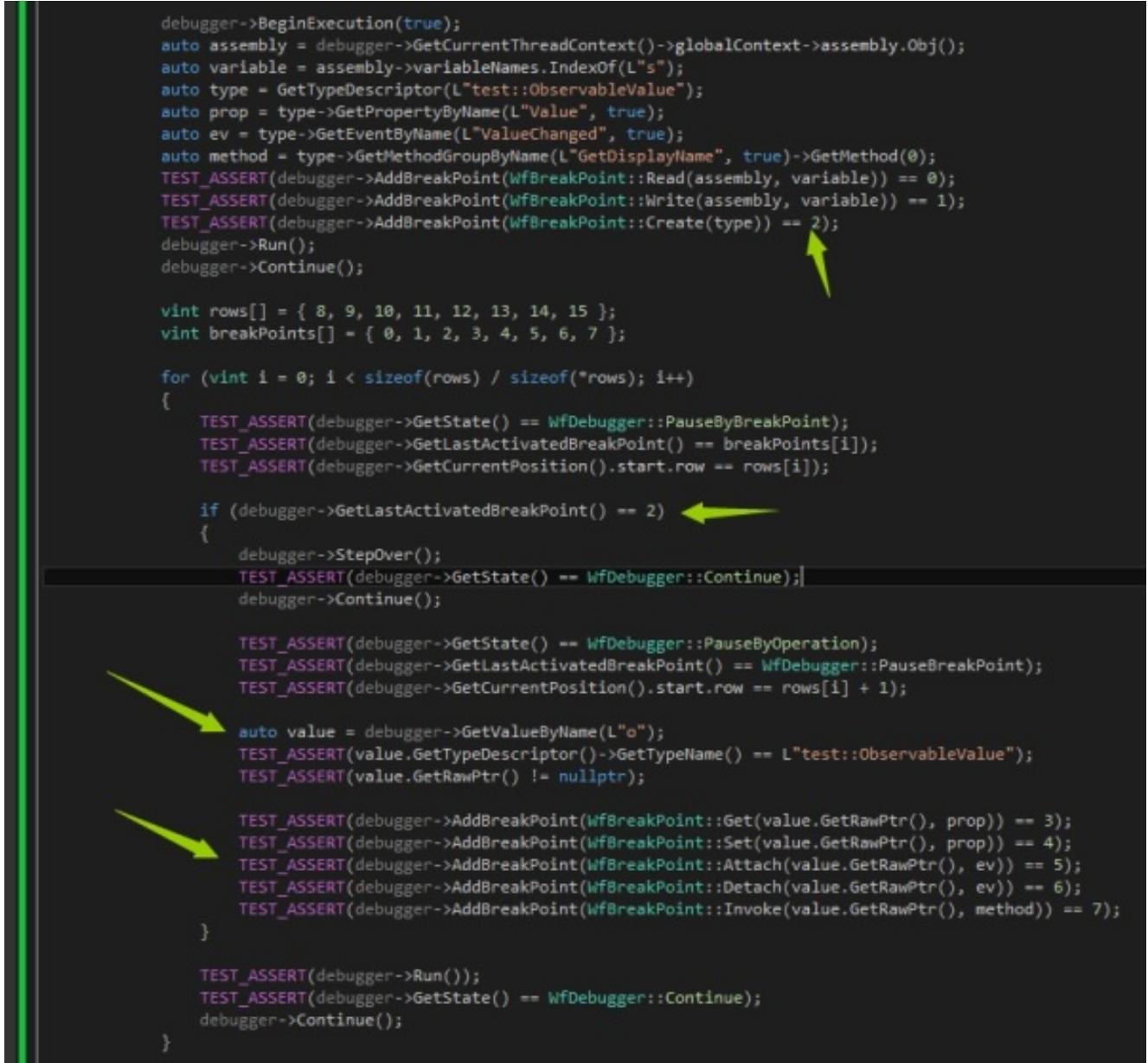
今天我在“看知乎”按照总赞同数排序，我已经排到了14，然后点开我自己的top 10回答，一共占有了10%的赞。我那么多具有无限的智慧的编程回答都没人看，大家还是喜欢看这个。

GacUI使用的脚本语言已经具有调试功能了！

测试用例可以在这里 (<http://gac.codeplex.com/SourceControl/latest#Libraries/Workflow/WorkflowSrc/WorkflowSrc/TestDebugger.cpp>) 找到。现在贴刚刚完成的一个测试用例的代码。功能是：

- 1、下断点监控test::ObservableValue的创建
- 2、StepOver（单步）之后获得创建后的对象的值（通过读取o变量）
- 3、添加一系列断点监控ObservableValue的状态变化，譬如读写属性啊，调用函数啊，操作事件等。
- 4、剩下的TEST_ASSERT确保断点在接下来的运行中挨个命中，并且命中完了会执行完并退出。

GacStudio的完善指日可待。



```
debugger->BeginExecution(true);
auto assembly = debugger->GetCurrentThreadContext()->globalContext->assembly.Obj();
auto variable = assembly->variableNames.IndexOf(L"s");
auto type = GetTypeDescriptor(L"test::ObservableValue");
auto prop = type->GetPropertyByName(L"Value", true);
auto ev = type->GetEventByName(L"ValueChanged", true);
auto method = type->GetMethodGroupByName(L"GetDisplayName", true)->GetMethod(0);
TEST_ASSERT(debugger->AddBreakPoint(WfBreakPoint::Read(assembly, variable)) == 0);
TEST_ASSERT(debugger->AddBreakPoint(WfBreakPoint::Write(assembly, variable)) == 1);
TEST_ASSERT(debugger->AddBreakPoint(WfBreakPoint::Create(type)) == 2);
debugger->Run();
debugger->Continue();

vint rows[] = { 8, 9, 10, 11, 12, 13, 14, 15 };
vint breakPoints[] = { 0, 1, 2, 3, 4, 5, 6, 7 };

for (vint i = 0; i < sizeof(rows) / sizeof(*rows); i++)
{
    TEST_ASSERT(debugger->GetState() == WfDebugger::PauseByBreakPoint);
    TEST_ASSERT(debugger->getLastActivatedBreakPoint() == breakPoints[i]);
    TEST_ASSERT(debugger->GetCurrentPosition().start.row == rows[i]);

    if (debugger->getLastActivatedBreakPoint() == 2) ←
    {
        debugger->StepOver();
        TEST_ASSERT(debugger->GetState() == WfDebugger::Continue);
        debugger->Continue();

        TEST_ASSERT(debugger->GetState() == WfDebugger::PauseByOperation);
        TEST_ASSERT(debugger->getLastActivatedBreakPoint() == WfDebugger::PauseBreakPoint);
        TEST_ASSERT(debugger->GetCurrentPosition().start.row == rows[1] + 1);

        auto value = debugger->GetValueByName(L"o");
        TEST_ASSERT(value.GetTypeDescriptor()->GetTypeName() == L"test::ObservableValue");
        TEST_ASSERT(value.GetRawPtr() != nullptr);

        TEST_ASSERT(debugger->AddBreakPoint(WfBreakPoint::Get(value.GetRawPtr(), prop)) == 3);
        TEST_ASSERT(debugger->AddBreakPoint(WfBreakPoint::Set(value.GetRawPtr(), prop)) == 4);
        TEST_ASSERT(debugger->AddBreakPoint(WfBreakPoint::Attach(value.GetRawPtr(), ev)) == 5);
        TEST_ASSERT(debugger->AddBreakPoint(WfBreakPoint::Detach(value.GetRawPtr(), ev)) == 6);
        TEST_ASSERT(debugger->AddBreakPoint(WfBreakPoint::Invoke(value.GetRawPtr(), method)) == 7);
    }

    TEST_ASSERT(debugger->Run());
    TEST_ASSERT(debugger->GetState() == WfDebugger::Continue);
    debugger->Continue();
}
```

搞了一个下午的CMake

终于可以build [@Mili](#) 移植的GacUI for X11Cairo了， 虽然X11有各种bug，但是总的来说，还是可以看的.....

[milizhang/XGac · GitHub](#)

下个台式机一定不能有机械硬盘了

记得以前SSD装满了游戏，内存又因为主板傻逼的问题不能插16G，于是我就给虚拟机开了个2G的内存，放在机械硬盘上跑Linux，于是就有了这样的故事：[Linux下面的编译器消耗内存就是大啊，烂爆了。 - vczh的日常 - 知乎专栏](#)。本来我看linuxer们把clang和gcc吹的那么响亮，我以为他们也跟VC++那种从4M内存时代走过来的编译器一样是很节约内存的，结果我错了，一个GacUIReflection.cpp编译了一个多小时没出来。

后来我由于一些游戏不完了，就清出了100G的空间，为了编译https://github.com/milizhang/xg_ac的代码，就想想把虚拟机挪到SSD看看怎么样。结果10分钟就编译出来了！

SSD真是clang的好搭档啊。

最近开始恢复开发<https://github.com/vcjh/herodb>了，估计内存再大也不够用了。等到我下次换台式机的时候，1T的SSD也差不多靠谱了吧，一次买三个，机械硬盘再见！

clang就跟gcc一样烂

论编译器如何让我的代码变的更丑：

[Fuck clang++ 3.4-1ubuntu3 who will crash if I write 'volatile vint inten... · vczh/herodb@1496b58 · GitHub](#)

[FUCK G++ http://gcc.gnu.org/bugzilla/show_bug.cgi?id=52852 · vczh/tinymoe@19f8e2b · GitHub](#)

[Move codegen cases from raw strings to files due to the buggy g++ · vczh/tinymoe@463f0f9 · GitHub](#)

把GacUI搬上github了

[Vczh Libraries · GitHub](#)

GacUI里面其实有若干子项目，但是由于项目之间有偏序依赖关系，但是git submodule又没办法轻松应对这种做法，所以一直都放在codeplex上面。不过我最近想到了一个办法可以解决这个问题。

想法源于GacUI的发布系统。GacUI有大约三百个C++文件，而且由于是大量使用模板所以没有dll和lib。这样的话势必会给使用GacUI的人带来麻烦，所以我之前就写了Codepack.exe可以把一大堆C++文件打包成几个大的C++文件，就有了<http://www.gaclib.net>说的那个10个文件的事情。

昨天我突然发现，如果每一个子项目都自己的所有文件打包成一对（复杂的话多点）C++文件，然后别的子项目只引用打包好的文件的话，那依赖就很容易处理了。因此我修改了一下Codepack.exe，让他直接从文件夹里面获取文件，而不是像以前一样从vcxproj里面获取文件。然后开发方法就变成了这样：

1、每当一个子项目到了一个新的阶段之后，就去Release文件夹里面执行Release.bat，生成了打包好的C++文件之后push上去。

2、去引用了这个子项目的其他项目的Import文件夹里面执行Import.bat（还没写），根据配置把打包好的文件从github上下载下来。

这样做好处，如果我的依赖关系是这样B->A, C->A, D->BC的话，原本用git submodule会搞得很麻烦，但是在我这里，只需要在Import文件夹里面下载好ABC的打包好的C++文件就可以了。因为B和C都知道，他们依赖的A是同一个A，所以生成出来的打包文件是兼容的。

所以我昨天就在github上面开了一个Organization，设计好File Naming Convention，然后把GacUI拆散之后上传到了Organization下面的每一个子项目里面。做了一天时间，还有一些东西没上传完，不过亲测这个做法对我的使用习惯来说无比方便。

如果大家要下载下来玩的话，得把你想要下载的项目和Tools项目都放在同一个文件夹下面，然后去Tools/Tools/目录下，要么sudo BuildToolsLinux.sh，要么打开Visual Studio Command Prompt执行BuildTools.bat，我会把需要的工具编译完然后扔在合适的地方，其他项目才能正常编译。

后面我会把vczh/herodb删掉，文件重新组织之后也扔到Vczh Libraries里面。等GacUI的文件都整理好之后，vczh/gac也删掉，然后把GacUI Release专门做成一个项目也扔到Vczh Libraries里面。

第一次“编程格调秀”的结果评论

@高博

比赛内容（看那个url我觉得可能是具有时效性的）：[「编程格调秀」](#)

第一个问题的所有答案：

- 1) [UnDownDing/PStyle_UnDownding](#)
- 2) [michaelliao/PStyle_michaelliao](#)
- 3) [Junzki/PStyle_Junzki](#)
- 4) [aaronzhou/PStyle_aaronzhou](#)
- 5) [8cbx/PStyle_8cbx](#)

第二个问题的所有答案：

- 6) [michaelliao/PStyle_michaelliao](#)
- 7) [8cbx/PStyle_8cbx](#)
- 8) [8cbx/PStyle_8cbx](#)

鉴于我没怎么用过java和python，所以要是有什么说的不对，你们可以很客气的评论。

主办方说，我们只考虑符合题目要求的程序。第二道题目的要求是：“但所选算法要拥有能够无限逼近 π 的能力（仅受限于计算资源）”，但是(6)使用的是java内置的数字类型所以肯定不正确。(8)用的是math.acos所以肯定不正确。(7)的话，我查了一下，python的有些interpreter好像可以自动把数字promote成bignum，但是怎么查也查不到对浮点数也有类似的支持。所以我假设他不行，因此(7)也是不正确的。当然了，如果不考虑这一点的话，三个程序都是给出了一个PI的值。

说到第一道题。有格调的代码是什么意思？首先要正确，其次面对错误的输入的时候要始终有一致的反应。这里我说的并不是你一定要抛出同一个类型的异常然后还加上精心设计的异常消息。只要你能够总是crash，那也算是有一致的反映了。

那么我们来看着(1)到(5)。其中(1)完全没有异常处理。(2)算是比较好的，但是考虑到我们的历法并不是从公元1年开始就是这么固定下来的，所以第10行的做法其实是不对的。当然了，考虑这个问题太蛋疼了，所以我只是随便说说而已。(3)表面上看起来好像是有异常处理，但是实际上并不能保证输入的day一定是在1-365(366)之间的，所以也不行。(4)的异常处理跟(2)是一样的。(5)完全没有异常处理。(5)有一点还特别好玩，如果输入的day ≤ 1 ，那程序还会访问到一些不该访问的东西。漏洞就这么暴露了。

那现在就剩下(2)和(4)了。(4)还有一个比较隐含的问题，就是如果cin>>失败了会怎么样。程序完全没有理就这么跑下去了。所以目前看起来(2)比较强一点。当然了，这跟他到处都用java的标准库也是有关系的。

目前看起来(2)是唯一一个满足正确性的答案了。

那你说(2)有没有格调，我个人认为这算是一个很平常的程序，要上升到格调的程度大概是没有的。当然最后的结果就看主办方的意思了。如果主办方觉得这种情况下应该给奖励，那就给奖励。

我昨晚还想了一下，如果我要来写这个东西那该怎么写。其实我觉得运算的过程其实关系不大，干脆就打表吧。但是表要怎么打，才能使得你写出来的代码最简单的同时，凡是错误的输入一定会稳定的崩溃呢？我来写一段C#的抖机灵代码。但是我懒得上Visual Studio调试，所以不排除会有错误。

```
static class Program
{
    // 下面的先写个程序直接生成结果打表，哈哈哈
    static DateTime[] leapYearData = new DateTime[/*366*/] { ... } ;
    static DateTime[] normalYearData = new DateTime[/*365*/] { ... };

    static void Main(string[] args)
    {
        // 不是数字挂
        var year = int.Parse(args[0]);
        // 不是数字挂
        var day = int.Parse(args[1]);
        bool leapYear = year % 400 == 0 || (year % 100 != 0 && year % 4 == 0);
        // day < 1 挂，day闰年超过366挂，day平年超过365挂
        var date = (leapYear ? leapYearData : normalYearData)[day - 1];
        // year < 1 挂
        date.Year = year;
        // 能走到这一步，输入肯定是对的。
```

```
        Console.WriteLine(date);
    }
}
```

在这里我的评论就结束了。我发表在这里主要是友情推广一下这个比赛，因为要是参赛选手的平均格调没有提高，题目的难度还是这么容易（太容易的题目是很难把格调写出来的）的话，那搞这个活动就没意思了。

下一周是谁来评选？反正我得休息一下了，啊哈哈哈哈。

GacUI_for_Browsers!

vczh-libraries/GacJS · GitHub

GacUI for Browsers! 项目已经正式启动，计划是只要GacGen.exe可以读的xml，都可以添加一些配置让GacGen.exe生成一个js文件，然后结合我事先写好的js，调用一个函数拿到一个HTMLElement，直接appendChild，GUI就在浏览器里面运行了！

GacUI分为Element（图元），Composition（排版），Template（皮肤），Control（控件）以及Workflow Script（脚本）五个部分。

目前我打算先实现Element和Composition，然后结合之前花费了很大的力气实现的C++头文件的XML注释分析器（[GacUI/Document at master · vczh-libraries/GacUI · GitHub](https://GacUI/Document)），先用它们重写一个自带前端导航的像MSDN一样的GacDN（雾），用github.io先发布出去。

接下来修改Workflow脚本的编译器，让他支持把脚本编译成JavaScript，再让我写的js库自己产生出类的元数据的信息，实现Workflow Script无缝编译。

最后把剩下的做完。然后GacUI就可以在浏览器里面运行了！

目前支持的浏览器有：IE最新版，Firefox最新版以及Chrome最新版。所有旧版本都不予考虑，反正你们总是可以更新的。

这个项目最终完成之后，开发GacUI程序的流程就变成

1. 使用GacStudio拖控件搞出XML
2. 使用GacStudio定义好MVVM里面的ViewModel的接口，用Workflow Script做好data binding
3. 使用Workflow Script实现测试用的ViewModel，先在GacStudio里面debug好
4. 调用GacGen.exe生成编译好的二进制资源以及一些C++代码，然后使用C++实现ViewModel，就可以完成同时可以运行在Windows、Linux（[@Mili](#)）以及Mac（[@dark fall](#)）下面的程序了。
5. 调用GacGen.exe生成一个人类看不懂的JavaScript文件，使用JavaScript实现ViewModel，直接运行在浏览器上！

想想都觉得有点小激动。

后面GacUI的任务还有GacStudio、Canvas接口、Ribbon、Dock Panel、图表控件、触摸屏支持等一系列功能，新功能将会同步更新到GacUI上。在GacUI还没支持原生移动设备开发的时候，GacJS还可以用来暂时搞定搞移动设备（虽然我个人的态度是原生的更好）。

我觉得我往One Man Army的星辰大海的道路上越走越远了，技能树枝繁叶茂，就要可以赶上[@Skogkatt](#)了。

一天就写了一个_Razor.js_模板引擎

代码: [GacJS/Razor.js at master · vczh-libraries/GacJS · GitHub](#)

测试页面: [GacJS/Razor.html at master · vczh-libraries/GacJS · GitHub](#)

我只跑了三个test case, 里面估计还有一些bug, 不过总的来说已经做出来了。目前已知还没有实现的就只有一个功能, 譬如说在一个语句的大括号里面是代码模式, 进了<p>转html模式, 出了<p>转回代码模式, HTML tag之间要匹配。我专门去搜集了HTML5的void elements的列表, 明天就把它写出来。

Expressions	Statements	Functions	Codes	
Razor Script	JavaScript	Result		
<pre>Package RazorTest Model User <p> Hi, my name is @model.name.
 I am @model.age years old.
 @if (@model.gender != Gender.Description.Unknown) { @:I am a @(model.gender === Gender.Description.Male ? "man" : "woman").
 } </p></pre>	<pre>(function () { eval(Packages.Inject(["Html.RazorHelper", "RazorTest"])); return function (model) { User.RequireType(model); var Sprinter = new RazorPrinter(); Sprinter.PrintHtml("<p>"); Sprinter.PrintHtml("Hi, my name is " + model.name); Sprinter.PrintValue(model.name); Spriter.PrintHtml("
"); Spriter.PrintHtml("I am "); Spriter.PrintValue(model.age); Spriter.PrintHtml("
"); if (model.gender != Gender.Description.Unknown) { Spriter.PrintHtml("I am a "); Spriter.PrintValue(model.gender === Gender.Description.Male ? "man" : "woman"); Spriter.PrintHtml("
"); } Spriter.PrintHtml("</p>"); return new RazorHtml(Spriter.Text); } })()</pre>	Hi, my name is John I am 16 years old. I am a man.		

Expressions	Statements	Functions	Codes	
Razor Script	JavaScript	Result		
<pre>Package RazorTest Model User @function PrintName() { Hi, my name is @model.name.
 } @function PrintAge(user) { I am @user.age years old.
 } @function PrintGender(user, useless) { @if (@user.gender != Gender.Description.Unknown) { @:I am a @(user.gender === Gender.Description.Male ? "man" : "woman").
 } } <p> @PrintName(model) @PrintAge(model) @PrintGender(model) </p></pre>	<pre>(function () { eval(Packages.Inject(["Html.RazorHelper", "RazorTest"])); return function (model) { User.RequireType(model); function PrintName() { var Spriter = new RazorPrinter(); Spriter.PrintHtml("Hi, my name is "); Spriter.PrintValue(model.name); Spriter.PrintHtml("
"); return new RazorHtml(Spriter.Text); } function PrintAge(user) { var Spriter = new RazorPrinter(); Spriter.PrintHtml("I am "); Spriter.PrintValue(user.age); Spriter.PrintHtml("
"); return new RazorHtml(Spriter.Text); } function PrintGender(user, useless) { var Spriter = new RazorPrinter(); if (user.gender != Gender.Description.Unknown) { Spriter.PrintHtml("I am a "); Spriter.PrintValue(user.gender === Gender.Description.Male ? "man" : "woman"); Spriter.PrintHtml("
"); } return new RazorHtml(Spriter.Text); } var Spriter = new RazorPrinter(); Spriter.PrintHtml("<p>"); Spriter.PrintValue(PrintName(model)); Spriter.PrintValue(PrintAge(model)); Spriter.PrintValue(PrintGender(model)); Spriter.PrintHtml("</p>"); return new RazorHtml(Spriter.Text); } })()</pre>	Hi, my name is John I am 16 years old. I am a man.		

Razor Script	JavaScript
<pre> @package RazerTest @model User @{ function PrintName(user) { return "Hi, my name is " + user.name + ","; } function PrintAge(user) { return "I am " + user.age + " years old."; } function PrintGender(user) { if (user.gender === Gender.Description.Unknown) { return "I am a " + (user.gender === Gender.Description.Male ? "man" : "woman") + ","; } else { return ""; } } } @PrintName(model) @> @PrintAge(model) @> @PrintGender(model) @> </p> </pre>	<pre> (function () { eval(Packages.Inject("Html.RazerHelper", "RazerTest")); return function (model) { User.RequireType(model); function PrintName(user) { return "Hi, my name is " + user.name + ","; } function PrintAge(user) { return "I am " + user.age + " years old."; } function PrintGender(user) { if (user.gender === Gender.Description.Unknown) { return "I am a " + (user.gender === Gender.Description.Male ? "man" : "woman") + ","; } else { return ""; } } var Iprinter = new RazerPrinter(); Iprinter.PrintInitial("<p>"); Iprinter.PrintValue(PrintName(model)); Iprinter.PrintInitial("
"); Iprinter.PrintValue(PrintAge(model)); Iprinter.PrintInitial("
"); Iprinter.PrintValue(PrintGender(model)); Iprinter.PrintInitial("
"); Iprinter.PrintValue("</p>"); return new RazerInitial(Iprinter.Text); } }) </pre>

GacUI的新文档网页看起来就要做完了！

最近星际打得如火如荼啊，但是在一个星期前猛然发现自己已经没有更新github长达三个星期之后，现在又陆陆续续回来写。

GacUI的新文档网页使用了Visual Studio规定的XML文档的格式，不过由于它对C++的模板完全没有支持，所以最后我还是放弃了VC++提供的工具链，自己来parse。

开发流程如下：

1、写C++代码：

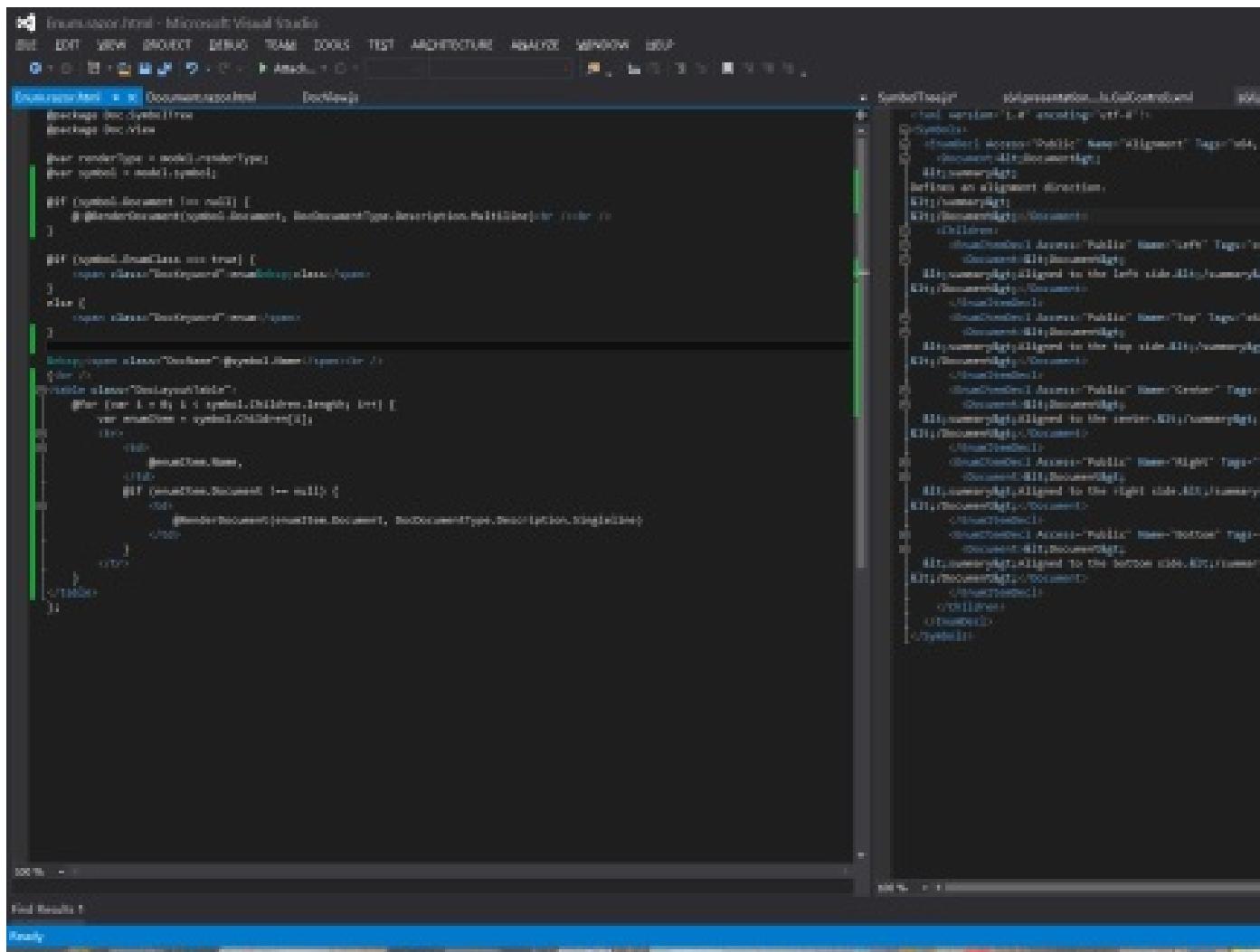
```
/// <summary>
/// Defines an alignment direction.
/// </summary>
enum class Alignment
{
    /// <summary>Aligned to the left side.</summary>
    Left=0,
    /// <summary>Aligned to the top side.</summary>
    Top=0,
    /// <summary>Aligned to the center.</summary>
    Center=1,
    /// <summary>Aligned to the right side.</summary>
    Right=2,
    /// <summary>Aligned to the bottom side.</summary>
    Bottom=2,
};
```

2、创建一个可以直接或间接看到你所有的C++头文件的头文件：

[GacUI/Headers.h at master · vczh-libraries/GacUI · GitHub](#)

3、跑我写的脚本，使用cl.exe做preprocessing之后，使用我写的C#程序来最终生成一堆文件。我使用了受过良好的教育的人才能想出来的猥琐的办法来分析一个C++的头文件。

[GacUI/BuildDocument.bat at master · vczh-libraries/GacUI · GitHub](#)



4、然后只要放进网站的一个目录里面就可以了：

[GacJS/Doc/Data at master · vczh-libraries/GacJS · GitHub](#)

5、最后一步就是目前正在做的，使用我自己写的RazorJS template（[GacJS/Razor.js at master · vczh-libraries/GacJS · GitHub](#)），然后把整个文档网站给开发出来（[GacJS/Doc/View at master · vczh-libraries/GacJS · GitHub](#)）。

6、就可以看到截图里面的内容啦！

自己写的RazorJS真是越用越爽啊，灵感100%来源于<http://ASP.NET> MVC所提供的Razor模板，这个语法设计的真是太牛逼了，简直无法更短更可读。

下面贴一段第二个截图里面的文件（[GacJS/Enum.razor.html at master · vczh-libraries/GacJS · GitHub](#)），也就是最后用来渲染出题图右边网页的razor template，感受一下。

```
@package Doc.SymbolTree
@package Doc.View

@var renderType = model.renderType;
@var symbol = model.symbol;

@if (symbol.Document !== null) {
    @:RenderDocument(symbol.Document, DocDocumentType.Description.Multiline)<br /><br />
}
```

```
@if (symbol.EnumClass === true) {
    <span class="DocKeyword">enum&nbsp;class</span>
}
else {
    <span class="DocKeyword">enum</span>
}

&nbsp;<span class="DocName">@symbol.Name</span><br />
<br />
<table class="DocLayoutTable">
    @for (var i = 0; i < symbol.Children.length; i++) {
        var enumItem = symbol.Children[i];
        <tr>
            <td>
                @enumItem.Name,
            </td>
            @if (enumItem.Document !== null) {
                <td>
                    @RenderDocument(enumItem.Document, DocDocumentType.Description.Singleline)
                </td>
            }
        </tr>
    }
</table>
};
```

GacUI新文档页面前端导航完成！

其实就把带有文档内容的tree node加上一个href="#~/Fuck"，然后就靠很久以前自己写的一个前端导航的小框架（[GacJS/Navigation.js at master · vczh-libraries/GacJS · GitHub](#)）来实现这个效果。

不过这次写起来比较麻烦，因为左边的这颗tree是惰性加载的，也就是说里面分了若干层，打开加号之后才会加载里面的内容。所以这次靠hash直接定位到了最终的文档节点之后，还要反过来确定这个文档在tree中的位置，然后一层一层加载好，然后给选中的tree node加粗，最后把文档显示出来。

尽管如此，其实不费吹灰之力就弄出来了，哈哈哈哈。

现在就剩下最后一步，就是给文档里面的类型和注释里面的超链接做成真正的hash超链接，然后稍微美化一下，就可以上传到[http://github.io](#)预览了。不过据说[http://github.io](#)需要翻墙才能看到。

其实这种这么MVC的页面，为什么不用[http:// ASP.NET](#) MVC和Razor，而要自己重做一次呢？因为[http://github.io](#)连后端的js程序都不能执行（烂爆了），所以只好全部写在了前端……做完之后，给GacUI翻新新的Demo，然后就可以重写[http://gaclib.net](#)了。

gaclib.net_的新文档网站完成！

<http://vczh-libraries.github.io/Document.html>

500个星星的github_repo就这么扔掉了

旧的Demo也不要了，都过时了。我打算使用GacUI的XML资源来重写所有Demo，然后用GacJS重写 <http://gaclib.net>。

GacUI的控件layout系统已经超神！

继Stack, Flow和Table之后，最后一个控件嵌入RichTextDocument的排版功能已经完成了。GacUI已然成为了世界上排版功能最强大的GUI之二（WPF 10年前就实现了）。目前只实现了Direct2D的版本，等GDI的版本也做完了就发release。

这是窗口的源代码：

```
<Resource>
<Folder name="EmbeddedDocument">
    <Image content="File" name="Gaclib">Gaclib.png</Image>
    <Doc name="Document">
        <Doc>
            <Content>
                <p align="Center">
                    <div style="Title">
                        <b>Controls Embedded in Document</b>
                    </div>
                </p>
                <p>
                    <div style="Body">
                        <nop>Hi, our valued GacUI users:</nop>
                    </div>
                </p>
                <p>
                    <div style="Body">
                        <nop>This is a rich text document. There are many features in a document. For example, a paragraph can align to the left,</nop>
                    </div>
                </p>
                <p align="Center">
                    <div style="Body">
                        <nop>to the center,</nop>
                    </div>
                </p>
                <p align="Right">
                    <div style="Body">
                        <nop>or to the right,</nop>
                    </div>
                </p>
                <p>
                    <div style="Body">
                        <nop>Even an </nop>
                        <img source="res://EmbeddedDocument/Gaclib" width="250" height="100"/>
                        <nop> can be embedded inside a document and adjust its size.</nop>
                    </div>
                </p>
                <p>
                    <div style="Body">
                        <nop>Besides of this, controls can be embedded into a document too, like a </nop>
                        <object name="Button"/>
                        <nop>, a </nop>
                        <object name="TreeView"/>
                        <nop>, or even a </nop>
                        <object name="Menu"/>
                        <nop>.</nop>
                    </div>
                </p>
                <p>
                    <div style="Body">
                        <nop>If you can embed a control, why not embed another layout composition like this: </nop>
                        <object name="Table"/>
                        <nop>?</nop>
                    </div>
                </p>
                <p>
                    <div style="Body">
                        <nop>Obviously, embedding a </nop>
                        <a href="http://www.gaclib.net">http://www.gaclib.net</a>
                        <nop> is also a feature. You can try to click on it and see what happens.</nop>
                    </div>
                </p>
                <p>
                    <div style="Body">
                        <nop>Hope you like this powerful tools. Thank you.</nop>
                    </div>
                </p>
            </Content>
            <Styles>
                <Style name="Content">
                    <face>Segoe UI</face>
                </Style>
                <Style name="Title" parent="Content">
                    <size>24</size>
                    <b>true</b>
                </Style>
                <Style name="Body" parent="Content">
                    <size>14</size>
                </Style>
            </Styles>
        </Doc>
    </Doc>
</Folder>
<Instance name="MainWindowResource">
    <Instance ref.Class="test::MainWindow">
        <Window Text="Window" ClientSize="x:480 y:640">
            <att.BoundsComposition-set PreferredMinSize="x:480 y:640"/>
            <att.ContainerComposition-set MinSizeLimitation="LimitToElementAndChildren"/>
        <DocumentViewer EditMode="Selectable" Document-uri="res://EmbeddedDocument/Document">
            <att.BoundsComposition-set AlignmentToParent="left:5 top:5 right:5 bottom:5"/>
            <DocumentItem Name="Button">
                <Button Text="Button"/>
        </DocumentViewer>
    </Instance>
</Instance>
```

```

</DocumentItem>
<DocumentItem Name="TreeView">
<TreeView HorizontalAlwaysVisible="false" VerticalAlwaysVisible="false">
<att.BoundsComposition-set PreferredMinSize="x:200 y:100"/>
<att.Nodes>
<TreeNode Text="GacUI Renderers" Expanding="true">
<TreeNode Text="Direct2D"/>
<TreeNode Text="GDI"/>
</TreeNode>
</att.Nodes>
</TreeView>
</DocumentItem>
<DocumentItem Name="Menu">
<ToolStripMenuBar>
<MenuBarButton Text="File" Alt="F">
<att.SubMenu-set>
<MenuItemButton Text="New" Alt="N"/>
<MenuItemButton Text="Open" Alt="O"/>
<MenuItemButton Text="Save" Alt="S"/>
<MenuItemButton Text="Save As ..." Alt="A"/>
<MenuSplitter/>
<MenuItemButton Text="Exit" Alt="X"/>
</att.SubMenu-set>
</MenuBarButton>
<MenuBarButton Text="Edit" Alt="E">
<att.SubMenu-set>
<MenuItemButton Text="Undo" Alt="U"/>
<MenuItemButton Text="Redo" Alt="R"/>
<MenuSplitter/>
<MenuItemButton Text="Cut" Alt="T"/>
<MenuItemButton Text="Copy" Alt="C"/>
<MenuItemButton Text="Paste" Alt="P"/>
<MenuSplitter/>
<MenuItemButton Text="Select All" Alt="A"/>
</att.SubMenu-set>
</MenuBarButton>
</ToolStripMenuBar>
</DocumentItem>
<DocumentItem Name="Table">
<Table AlignmentToParent="left:0 top:0 right:0 bottom:0" CellPadding="5">
<SolidBorder Color="#000000"/>
<att.Rows>
<CellOption>composeType:MinSize</CellOption>
<CellOption>composeType:MinSize</CellOption>
<CellOption>composeType:MinSize</CellOption>
</att.Rows>
<att.Columns>
<CellOption>composeType:MinSize</CellOption>
<CellOption>composeType:MinSize</CellOption>
<CellOption>composeType:MinSize</CellOption>
</att.Columns>
<Cell Site="row:0 column:0">
<Label Text="XOR"/>
</Cell>
<Cell Site="row:0 column:1">
<Label Text="false"/>
</Cell>
<Cell Site="row:0 column:2">
<Label Text="true"/>
</Cell>
<Cell Site="row:1 column:0">
<Label Text="false"/>
</Cell>
<Cell Site="row:1 column:1">
<Label Text="false"/>
</Cell>
<Cell Site="row:1 column:2">
<Label Text="true"/>
</Cell>
<Cell Site="row:2 column:0">
<Label Text="true"/>
</Cell>
<Cell Site="row:2 column:1">
<Label Text="true"/>
</Cell>
<Cell Site="row:2 column:2">
<Label Text="false"/>
</Cell>
</Table>
</DocumentItem>
</DocumentViewer>
</Window>
</Instance>
</Instance>
</Resource>

```

GacUI开QQ群了

群号：231200072

github org: <https://github.com/vczh-libraries>

网站（部分完成）：<http://www.gaclib.net>

GacUI用户和爱好者讨论专用，闲聊请去粉丝群（自己找）

最近GacUI正在进入一个关键的时期。由于脚本功能的不断加强，再过几天XML的窗口、控件、皮肤和其他对象的描述就可以在GacGen.exe的预编译后完全转换成脚本写进二进制的资源文件里面了。当然了这对接口是完全透明的，接口并没有任何改变，该怎么用还怎么用。

但是完全编译成脚本的重大意义是，这个Workflow脚本跟C++的关系，就犹如Swift和Objective C的关系一样，对象模型都是完全一致的，所以未来可以编译成C++。在这之后，不仅可以提高运行效率，而且编译出来的程序再也不需要带上GacUIReflection.cpp了，exe将从20M直线下降到2M。不过话虽如此，其实GacUIReflection编译后的二进制可以zip到超级小，所以比较两个zip以后的exe的话，差别不大。

近期的计划：

- 完善Workflow脚本，使其支持interface、enum、struct和class的定义（目前仅能使用，不能定义）
- Workflow脚本编译到C++
- XML描述加上Visual State, Story Board和Animation
- 抽象绘图API（目前自绘仅能直接使用系统提供的接口，如GDI、Direct2D、CoreGraphics和Cairo等）
- 控件加上Chart、Ribbon、Dock Container
- 完善TextBox的Intellisense接口，实现一个Workflow脚本的带Intellisense的编辑器
- GacStudio

然后GacUI就可以推出1.0了！

那1.0后面还有什么呢？当然是把XML窗口描述资源编译成Javascript的GacUI on Browsers! 了！

GacUI中文教程就写SegmentFault专栏了

<http://segmentfault.com/blog/gacui>

随着GacUI（[Vczh Libraries · GitHub](#)）的进一步完善，我觉得是时候来写写教程了。最近开了一个群（[GacUI开QQ群了 - vczh的日常 - 知乎专栏](#)），有那么一些用户觉得Tutorial的几个vcxproj和GacUI网站的类型参考（[Document](#)）不够用，所以经常有这样的呼声。之前之所以一直没有写，是因为内部的设计还在不断的变动。现在看来应该完全定下来了，可以写了。

SegmentFault的风气实在比知乎好太多了，所以以后我渐渐把回答编程问题的重心都转移到SF和[论道](#)上面去，知乎留下带逛跟娱乐。我觉得SF的机制就很好，有点声望才能赞，有一些声望才能踩，有很多声望才能改问题和折叠答案，不友善内容5次（期望值）永久封禁。这样才能实现内行人评价内行人答案的愿望。不过其实也只是理想，也是远远不完美的。

不过SF只允许人们提精确的问题，不满足要求的可以去论道上。我在那两个网站上会认真一点，知乎我是在也不认真了。想关注我学习技术的知友们，可以跑到那边去。

不使用QQ的、或者活动时间刚好是我在睡觉的时候的GacUI爱好者和用户，如果有问题的时候群里没人答，干脆就来SF开题打[gacui](#)标签邀请我就好了。GacUI已经差不多可以投入使用了，这个时候帮助用户解决使用GacUI的时候遇到的困难，是最好的时候。

趁着现在用户量还处于我一个人可以应付的时候，成为GacUI高手，日后迎娶白富美，走上人生巅峰（逃

陈萌萌的新台灯

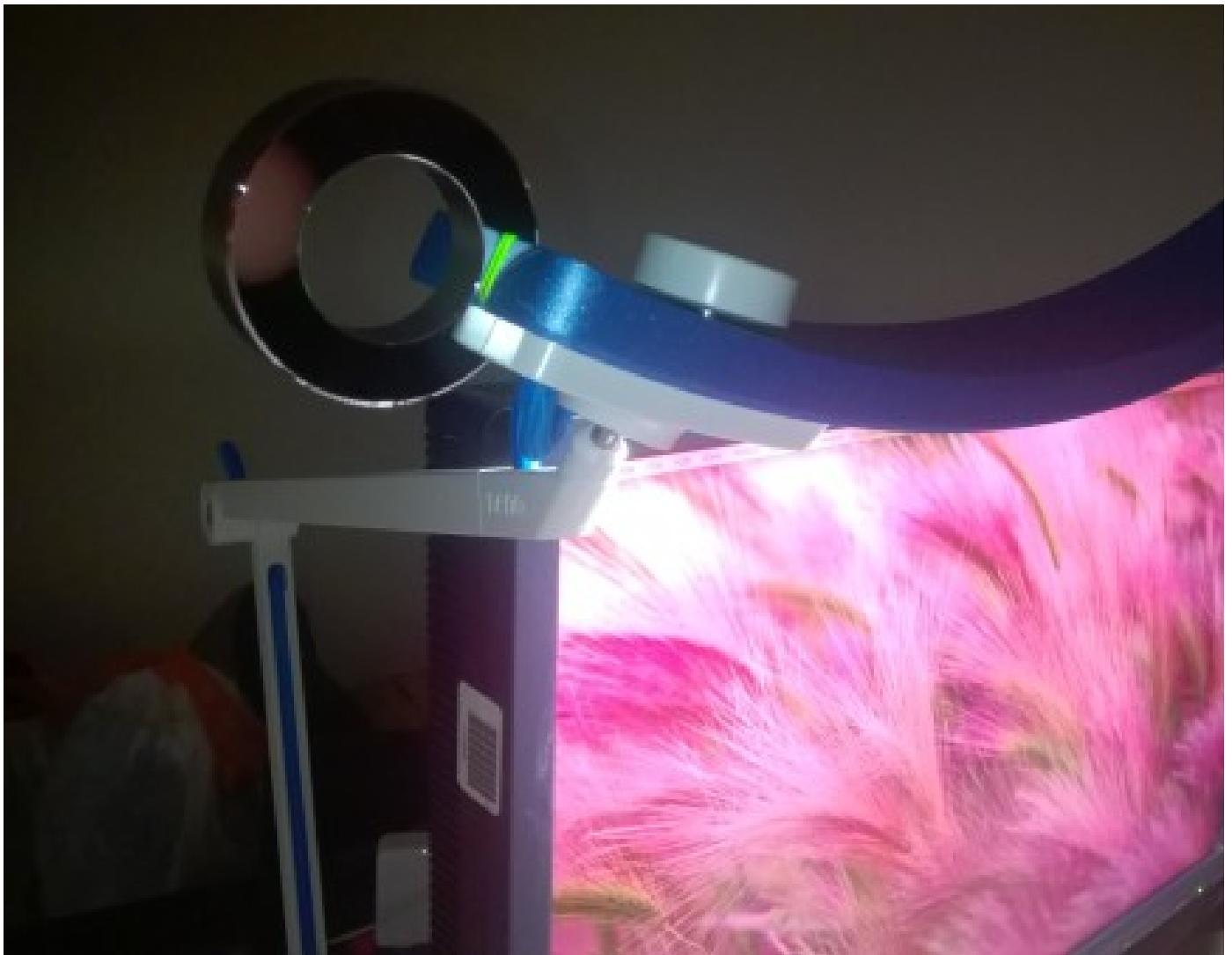
美帝用电量高的一逼，前个月给我收了360刀，我说就算开了暖气也没这么厉害吧，于是就开始换家里的灯泡。虽然这边很多东西都很标准化，但是剩下[@陈萌萌](#)在家里工作用的地方没法换成LED，于是在碰上BENQ搞活动，就给我弄了一台（滚到文章最下面立减300-500）。

自从开了这个灯之后，工作就不需要头顶那一串可以用来煮人的吊灯了，节约一大笔金钱，每个月可以多吃一顿牛排。灯面是弧形的，整张桌子都给照亮了，而且还不会影响显示器。





这个灯有橙色和蓝色过渡的光线，不过由于老婆的iPhone渣画质，我觉得照片上基本没看出效果（逃。而且通过不同的姿势爱抚台灯的迷之突起，还可以实现开关、明暗调节、颜色调节和切换进护眼模式等操作。





更有用的是，不需要移动灯座，在半径一米半球内的任何点都可以放置光源，以任何方向照亮东西。从书上抄代码什么的再也不需要害怕加深近视了。

你们给我弄的这个灯啊，Excited!

其实还可以discount高达500软妹币的：

PC： [GeniusVcZn-明基旗舰店](#)

手机： https://h5.m.taobao.com/weapp/view_page.htm?page=shop/activity&userId=672165860&pageId=25697947



GacUI新进展，exe疯狂减少10M体积。

[GacUI - Home Page](#)

GacUI的工作原理其实很简单，主要就是你写好XML之后，运行GacGen.exe，帮你把一个这样的XML:

```
<?xml version="1.0" encoding="utf-8"?>
<instance ref.CodeBehind="true" ref.Class="helloworld::MainWindow">
  <ref.Parameter Name="ViewModel" Class="vm::IViewModel"/>
  <ref.Property Name="HasLoggedIn" Type="bool" Value="false"/>
  <Window ref.Name="self" Text="Let's Sign Up!" ClientSize="x:320 y:280">
    <att.BoundsComposition-set PreferredMinSize="x:320 y:280"/>
    <att.ViewModel-set UserName-bind="textBoxUserName.Text" Password-bind="textBoxPassword.Text"/>
  <Table CellPadding="5" AlignmentToParent="left:0 top:0 right:0 bottom:0">
    <att.Rows>
      <CellOption>composeType:Absolute absolute:90</CellOption>
      <CellOption>composeType:MinSize</CellOption>
      <CellOption>composeType:MinSize</CellOption>
      <CellOption>composeType:MinSize</CellOption>
      <CellOption>composeType:MinSize</CellOption>
      <CellOption>composeType:Absolute absolute:12</CellOption>
      <CellOption>composeType:Percentage percentage:1.0</CellOption>
      <CellOption>composeType:MinSize</CellOption>
    </att.Rows>
    <att.Columns>
      <CellOption>composeType:MinSize</CellOption>
      <CellOption>composeType:Percentage percentage:1.0</CellOption>
    </att.Columns>
    <Cell Site="row:0 column:0 columnSpan:2">
      <SolidLabel Text="www.gaclib.net" HorizontalAlignment="Center" VerticalAlignment="Center">
        <att.Font>fontFamily:{Segoe UI} size:40 antialias:true</att.Font>
      </SolidLabel>
    </Cell>
    <Cell Site="row:1 column:0">
      <SolidLabel Text="Username: " VerticalAlignment="Center">
        <att.Font>fontFamily:{Segoe UI} size:12 antialias:true</att.Font>
      </SolidLabel>
    </Cell>
    <Cell Site="row:1 column:1">
      <SinglelineTextBox ref.Name="textBoxUserName">
        <att.BoundsComposition-set AlignmentToParent="left:0 top:0 right:0 bottom:0" PreferredMinSize="x:0 y:24"/>
      </SinglelineTextBox>
    </Cell>
    <Cell Site="row:2 column:1">
      <SolidLabel Color="#FF0000" WrapLine="true" WrapLineHeightCalculation="true" Text-bind="ViewModel.UserNameError">
        <att.Font>fontFamily:{Segoe UI} size:12 antialias:true</att.Font>
      </SolidLabel>
    </Cell>
    <Cell Site="row:3 column:0">
      <SolidLabel Text="Password: " VerticalAlignment="Center">
        <att.Font>fontFamily:{Segoe UI} size:12 antialias:true</att.Font>
      </SolidLabel>
    </Cell>
    <Cell Site="row:3 column:1">
      <SinglelineTextBox ref.Name="textBoxPassword" PasswordChar="*">
        <att.BoundsComposition-set AlignmentToParent="left:0 top:0 right:0 bottom:0" PreferredMinSize="x:0 y:24"/>
      </SinglelineTextBox>
    </Cell>
    <Cell Site="row:4 column:1">
      <SolidLabel Color="#FF0000" WrapLine="true" WrapLineHeightCalculation="true" Text-bind="ViewModel.PasswordError">
        <att.Font>fontFamily:{Segoe UI} size:12 antialias:true</att.Font>
      </SolidLabel>
    </Cell>
    <Cell Site="row:7 column:0 columnSpan:2">
      <Button ref.Name="buttonSignUp" Text="Sign Up!" ev.Clicked="buttonSignUp_Clicked">
        <att.BoundsComposition-set AlignmentToParent="left:0 top:0 right:-1 bottom:0" PreferredMinSize="x:100 y:24"/>
        <att.Enabled-bind>
          (not self.HasLoggedIn) and ViewModel.UserNameError == "" and ViewModel.PasswordError == ""
        </att.Enabled-bind>
      </Button>
      <Button ref.Name="buttonCancel" Text="Close">
        <att.BoundsComposition-set AlignmentToParent="left:-1 top:0 right:0 bottom:0" PreferredMinSize="x:100 y:24"/>
        <ev.Clicked-eval>
          self.Close();
        </ev.Clicked-eval>
      </Button>
    </Cell>
  </Table>
</Window>
</Instance>
```

给翻译成脚本语言：

```
Precompiled/Workflow/InstanceCtor/MainWindow/MainWindowResource
=====
module ;
using presentation::controls::Gui*;
using presentation::elements::Gui*Element;
using presentation::compositions::Gui*Composition;
using presentation::compositions::Gui*;
using presentation::templates::Gui*;
using system::;
using system::reflection::*;
using presentation::*;
using presentation::Gui*;
using presentation::controls::*;
using presentation::controls::list::*;
using presentation::controls::tree::*;
using presentation::elements::*;
using presentation::elements::Gui*;
using presentation::elements::text*;
using presentation::compositions::*;
using presentation::templates::*;

var ViewModel : ::vm::IViewModel^ = null;

var self : ::helloworld::MainWindow* = null;

var textBoxUserName : ::presentation::controls::GuiSinglelineTextBox* = null;
var textBoxPassword : ::presentation::controls::GuiSinglelineTextBox* = null;

var buttonSignUp : ::presentation::controls::GuiButton* = null;
var buttonCancel : ::presentation::controls::GuiButton* = null;

var <precompile>0 : ::presentation::compositions::GuiTableComposition* = null;
var <precompile>1 : ::presentation::compositions::GuiCellComposition* = null;
var <precompile>2 : ::presentation::elements::GuiSolidLabelElement^ = null;
var <precompile>3 : ::presentation::compositions::GuiCellComposition* = null;
var <precompile>4 : ::presentation::elements::GuiSolidLabelElement^ = null;
var <precompile>5 : ::presentation::compositions::GuiCellComposition* = null;
```

```

var <precompile>6 : ::presentation::compositions::GuiBoundsComposition* = null;
var <precompile>7 : ::presentation::compositions::GuicellComposition* = null;
var <precompile>8 : ::presentation::elements::GuisolidLabelElement^ = null;
var <precompile>9 : ::presentation::compositions::GuicellComposition* = null;
var <precompile>10 : ::presentation::elements::GuisolidLabelElement^ = null;
var <precompile>11 : ::presentation::compositions::GuicellComposition* = null;
var <precompile>12 : ::presentation::compositions::GuiBoundsComposition* = null;
var <precompile>13 : ::presentation::compositions::GuicellComposition* = null;
var <precompile>14 : ::presentation::elements::GuisolidLabelElement^ = null;
var <precompile>15 : ::presentation::compositions::GuicellComposition* = null;
var <precompile>16 : ::presentation::compositions::GuiBoundsComposition* = null;
var <precompile>17 : ::presentation::compositions::GuiBoundsComposition* = null;
var <precompile>18 : ::vm::IViewModel^ = null;
var <precompile>19 : ::presentation::compositions::GuiBoundsComposition* = null;
func <initialize-instance>(<this> : ::helloworld::MainWindow*, <resolver> : ::presentation::GuiroutePathResolver) : (::system::Void)
{
    (self = <this>);
    (ViewModel = <this>.ViewModel);
    (<precompile>19 = self.BoundsComposition);
    {
        (<precompile>19.PreferredMinSize = (cast (::presentation::Size) "x:320 y:280"));
    }
    {
        (self.ClientSize = (cast (::presentation::Size) "x:320 y:280"));
    }
    (<precompile>18 = self.ViewModel);
    {
        (self.Text = "Let's Sign Up!");
    }
    (<precompile>0 = new (::presentation::compositions::GuitableComposition*));
    {
        (<precompile>0.AlignmentToParent = (cast (::presentation::Margin) "left:0 top:0 right:0 bottom:0"));
    }
    {
        (<precompile>0.CellPadding = (cast (::system::Int32) "5"));
    }
    {
        <precompile>0.SetRowsAndColumns(8, 2);
        <precompile>0.SetRowOption(0, (cast (::presentation::compositions::Guicelloption) "composeType:Absolute absolute:90"));
        <precompile>0.SetRowOption(1, (cast (::presentation::compositions::Guicelloption) "composeType:MinSize"));
        <precompile>0.SetRowOption(2, (cast (::presentation::compositions::Guicelloption) "composeType:MinSize"));
        <precompile>0.SetRowOption(3, (cast (::presentation::compositions::Guicelloption) "composeType:MinSize"));
        <precompile>0.SetRowOption(4, (cast (::presentation::compositions::Guicelloption) "composeType:MinSize"));
        <precompile>0.SetRowOption(5, (cast (::presentation::compositions::Guicelloption) "composeType:Absolute absolute:12"));
        <precompile>0.SetRowOption(6, (cast (::presentation::compositions::Guicelloption) "composeType:Percentage percentage:1.0"));
        <precompile>0.SetRowOption(7, (cast (::presentation::compositions::Guicelloption) "composeType:MinSize"));
        <precompile>0.SetColumnOption(0, (cast (::presentation::compositions::Guicelloption) "composeType:MinSize"));
        <precompile>0.SetColumnOption(1, (cast (::presentation::compositions::Guicelloption) "composeType:Percentage percentage:1.0"));
    }
    (<precompile>1 = new (::presentation::compositions::GuicellComposition*));
    {
        <precompile>1.SetSite(0, 0, 1, 2);
    }
    (<precompile>2 = new (::presentation::elements::GuisolidLabelElement^));
    {
        (<precompile>2.Font = (cast (::presentation::FontProperties) "fontFamily:(Segoe UI) size:40 antialias:true"));
    }
    {
        (<precompile>2.VerticalAlignment = (cast (::presentation::Alignment) "Center"));
    }
    {
        (<precompile>2.HorizontalAlignment = (cast (::presentation::Alignment) "Center"));
    }
    {
        (<precompile>2.Text = "www.gaplib.net");
    }
    {
        (<precompile>1.OwnedElement = <precompile>2);
    }
    {
        <precompile>0.AddChild(<precompile>1);
    }
    (<precompile>3 = new (::presentation::compositions::GuicellComposition*));
    {
        <precompile>3.SetSite(1, 0, 1, 1);
    }
    (<precompile>4 = new (::presentation::elements::GuisolidLabelElement^));
    {
        (<precompile>4.Font = (cast (::presentation::FontProperties) "fontFamily:(Segoe UI) size:12 antialias:true"));
    }
    {
        (<precompile>4.VerticalAlignment = (cast (::presentation::Alignment) "Center"));
    }
    {
        (<precompile>4.Text = "Username: ");
    }
    {
        (<precompile>3.OwnedElement = <precompile>4);
    }
    {
        <precompile>0.AddChild(<precompile>3);
    }
    (<precompile>5 = new (::presentation::compositions::GuicellComposition*));
    {
        <precompile>5.SetSite(1, 1, 1, 1);
    }
    {
        var <controlStyle> = ::presentation::theme::ITheme::GetCurrentTheme().CreateTextBoxStyle();
        (textBoxUserName = new (::presentation::controls::GuisinglelineTextBox*)(<controlStyle>));
    }
    (<precompile>6 = textBoxUserName.BoundsComposition);
    {
        (<precompile>6.PreferredMinSize = (cast (::presentation::Size) "x:0 y:24"));
    }
    {
        (<precompile>6.AlignmentToParent = (cast (::presentation::Margin) "left:0 top:0 right:0 bottom:0"));
    }
    {
        <precompile>5.AddChild(textBoxUserName.BoundsComposition);
    }
    {
        <precompile>0.AddChild(<precompile>5);
    }
    (<precompile>7 = new (::presentation::compositions::GuicellComposition*));
    {
        <precompile>7.SetSite(2, 1, 1, 1);
    }

```

```

}
(<precompile>8 = new (::presentation::elements::GuiSolidLabelElement^) ());
{
    (<precompile>8.WrapLineHeightCalculation = (cast (::system::Boolean) "true"));
}
{
    (<precompile>8.WrapLine = (cast (::system::Boolean) "true"));
}
{
    (<precompile>8.Color = (cast (::presentation::Color) "#FF0000"));
}
{
    (<precompile>8.Font = (cast (::presentation::FontProperties) "fontFamily:{Segoe UI} size:12 antialias:true"));
}
{
    (<precompile>7.OwnedElement = <precompile>8);
}
{
    <precompile>0.AddChild(<precompile>7);
}
(<precompile>9 = new (::presentation::compositions::GuiCellComposition*) ());
{
    <precompile>9.SetSite(3, 0, 1, 1);
}
(<precompile>10 = new (::presentation::elements::GuiSolidLabelElement^) ());
{
    (<precompile>10.Font = (cast (::presentation::FontProperties) "fontFamily:{Segoe UI} size:12 antialias:true"));
}
{
    (<precompile>10.VerticalAlignment = (cast (::presentation::Alignment) "Center"));
}
{
    (<precompile>10.Text = "Password: ");
}
{
    (<precompile>9.OwnedElement = <precompile>10);
}
{
    <precompile>0.AddChild(<precompile>9);
}
(<precompile>11 = new (::presentation::compositions::GuiCellComposition*) ());
{
    <precompile>11.SetSite(3, 1, 1, 1);
}
{
    var <controlStyle> = ::presentation::theme::ITheme::GetCurrentTheme().CreateTextBoxStyle();
    (textBoxPassword = new (::presentation::controls::GuiSinglelineTextBox*)(<controlStyle>));
}
{
    (textBoxPassword.PasswordChar = (cast (::system::Char) "*"));
}
(<precompile>12 = textBoxPassword.BoundsComposition);
{
    (<precompile>12.PreferredMinSize = (cast (::presentation::Size) "x:0 y:24"));
}
{
    (<precompile>12.AlignmentToParent = (cast (::presentation::Margin) "left:0 top:0 right:0 bottom:0"));
}
{
    <precompile>11.AddChild(textBoxPassword.BoundsComposition);
}
{
    <precompile>0.AddChild(<precompile>11);
}
(<precompile>13 = new (::presentation::compositions::GuiCellComposition*) ());
{
    <precompile>13.SetSite(4, 1, 1, 1);
}
(<precompile>14 = new (::presentation::elements::GuiSolidLabelElement^) ());
{
    (<precompile>14.WrapLineHeightCalculation = (cast (::system::Boolean) "true"));
}
{
    (<precompile>14.WrapLine = (cast (::system::Boolean) "true"));
}
{
    (<precompile>14.Color = (cast (::presentation::Color) "#FF0000"));
}
{
    (<precompile>14.Font = (cast (::presentation::FontProperties) "fontFamily:{Segoe UI} size:12 antialias:true"));
}
{
    (<precompile>13.OwnedElement = <precompile>14);
}
{
    <precompile>0.AddChild(<precompile>13);
}
(<precompile>15 = new (::presentation::compositions::GuiCellComposition*) ());
{
    <precompile>15.SetSite(7, 0, 1, 2);
}
{
    var <controlStyle> = ::presentation::theme::ITheme::GetCurrentTheme().CreateButtonStyle();
    (buttonSignUp = new (::presentation::controls::GuiButton*)(<controlStyle>));
}
(<precompile>16 = buttonSignUp.BoundsComposition);
{
    (<precompile>16.PreferredMinSize = (cast (::presentation::Size) "x:100 y:24"));
}
{
    (<precompile>16.AlignmentToParent = (cast (::presentation::Margin) "left:0 top:0 right:-1 bottom:0"));
}
{
    (buttonSignUp.Text = "Sign Up!");
}
{
    <precompile>15.AddChild(buttonSignUp.BoundsComposition);
}
{
    var <controlStyle> = ::presentation::theme::ITheme::GetCurrentTheme().CreateButtonStyle();
    (buttonCancel = new (::presentation::controls::GuiButton*)(<controlStyle>));
}
(<precompile>17 = buttonCancel.BoundsComposition);
{
    (<precompile>17.PreferredMinSize = (cast (::presentation::Size) "x:100 y:24"));
}
{
    (<precompile>17.AlignmentToParent = (cast (::presentation::Margin) "left:-1 top:0 right:0 bottom:0"));
}
{
    (buttonCancel.Text = "Close");
}
{
    <precompile>15.AddChild(buttonCancel.BoundsComposition);
}
{
    <precompile>0.AddChild(<precompile>15);
}
{
    self.ContainerComposition.AddChild(<precompile>0);
}

```

```

}

var <created-subscription> = <this>.AddSubscription(new (::system::Subscription^)
{
    var <bind-cache>1 : ::vm::IViewModel^ = null of (::vm::IViewModel^);
    var <bind-handler>1_0 : ::system::reflection::EventHandler^ = null of (::system::reflection::EventHandler^);
    var <bind-opened> : ::system::Boolean = (cast (::system::Boolean) "false");
    var <bind-closed> : ::system::Boolean = (cast (::system::Boolean) "false");
    var <bind-listeners> : func (::system::Object) : (::system::Void)[::system::Listener^] = {};
    func <bind-activator>() : (::system::Void)
    {
        var <bind-activator-result> = <bind-cache>1.PasswordError;
        for (<bind-callback> in <bind-listeners>.Values)
        {
            (cast (func (::system::Object) : (::system::Void)) <bind-callback>(<bind-activator-result>));
        }
    }
    func <bind-callback>1_0() : (::system::Void)
    {
        <bind-activator>();
    }
    func <bind-initialize>() : (::system::Void)
    {
        (<bind-cache>1 = ViewModel);
        (<bind-handler>1_0 = attach(<bind-cache>1.PasswordChanged, <bind-callback>1_0));
    }
    override func Subscribe(<bind-callback> : func (::system::Object) : (::system::Void) : (::system::Listener^)
    {
        if (!!<bind-opened>)
        {
            (<bind-opened> = true);
            <bind-initialize>();
        }
        var <subscription> : ::system::Subscription* = this;
        var <listener-shared> = new (::system::Listener^)
        {
            override func GetSubscription() : (::system::Subscription*)
            {
                return <subscription>;
            }
            override func GetStopped() : (::system::Boolean)
            {
                return (!<bind-listeners>.Keys.Contains(this));
            }
            override func StopListening() : (::system::Boolean)
            {
                if (<bind-listeners>.Keys.Contains(this))
                {
                    <bind-listeners>.Remove(this);
                    return true;
                }
                return false;
            }
        };
        <bind-listeners>.Set(<listener-shared>, <bind-callback>);
        return <listener-shared>;
    }
    override func Update() : (::system::Boolean)
    {
        if (!!<bind-closed>)
        {
            <bind-activator>();
            return true;
        }
        return false;
    }
    override func Close() : (::system::Boolean)
    {
        if (!!<bind-closed>)
        {
            (<bind-closed> = true);
            detach(<bind-handler>1_0);
            (<bind-cache>1 = null of (::vm::IViewModel^));
            (<bind-handler>1_0 = null of (::system::reflection::EventHandler^));
            <bind-listeners>.Clear();
            return true;
        }
        return false;
    }
});
<created-subscription>.Subscribe(func (<value> : ::system::Object) : (::system::Void)
{
    var <old> = <precompile>8.Text;
    var <new> = (cast (::system::String) <value>);
    if ((<old> == <new>))
    {
        return;
    }
    (<precompile>8.Text = <new>);
});
<created-subscription>.Update();
}
var <created-subscription> = <this>.AddSubscription(new (::system::Subscription^)
{
    var <bind-cache>1 : ::vm::IViewModel^ = null of (::vm::IViewModel^);
    var <bind-handler>1_0 : ::system::reflection::EventHandler^ = null of (::system::reflection::EventHandler^);
    var <bind-opened> : ::system::Boolean = (cast (::system::Boolean) "false");
    var <bind-closed> : ::system::Boolean = (cast (::system::Boolean) "false");
    var <bind-listeners> : func (::system::Object) : (::system::Void)[::system::Listener^] = {};
    func <bind-activator>() : (::system::Void)
    {
        var <bind-activator-result> = <bind-cache>1.PasswordError;
        for (<bind-callback> in <bind-listeners>.Values)
        {
            (cast (func (::system::Object) : (::system::Void)) <bind-callback>(<bind-activator-result>));
        }
    }
    func <bind-callback>1_0() : (::system::Void)
    {
        <bind-activator>();
    }
}

```

```

func <bind-initialize>() : (::system::Void)
{
    (<bind-cache>1 = ViewModel;
    (<bind-handler>1_0 = attach(<bind-cache>1.PasswordErrorChanged, <bind-callback>1_0));
}

override func Subscribe(<bind-callback> : func (::system::Object) : (::system::Void) : (::system::Listener^)
{
    if (!<bind-opened>)
    {
        (<bind-opened> = true);
        <bind-initialize>();
    }
    var <subscription> : ::system::Subscription* = this;
    var <listener-shared> = new (::system::Listener^)
    {
        override func GetSubscription() : (::system::Subscription*)
        {
            return <subscription>;
        }

        override func GetStopped() : (::system::Boolean)
        {
            return (!<bind-listeners>.Keys.Contains(this));
        }

        override func StopListening() : (::system::Boolean)
        {
            if (<bind-listeners>.Keys.Contains(this))
            {
                <bind-listeners>.Remove(this);
                return true;
            }
            return false;
        }
    };
    <bind-listeners>.Set(<listener-shared>, <bind-callback>);
    return <listener-shared>;
}

override func Update() : (::system::Boolean)
{
    if (!<bind-closed>)
    {
        <bind-activator>();
        return true;
    }
    return false;
}

override func Close() : (::system::Boolean)
{
    if (!<bind-closed>)
    {
        (<bind-closed> = true);
        detach(<bind-handler>1_0);
        (<bind-cache>1 = null of (::vm::IVViewModel^));
        (<bind-handler>1_0 = null of (::system::reflection::EventHandler^));
        <bind-listeners>.Clear();
        return true;
    }
    return false;
}
);
<created-subscription>.Subscribe(func (<value> : ::system::Object) : (::system::Void)
{
    var <old> = <precompile>14.Text;
    var <new> = (cast (::system::String) <value>);
    if ((<old> == <new>))
    {
        return;
    }
    <precompile>14.Text = <new>;
});
<created-subscription>.Update();
}

var <created-subscription> = <this>.AddSubscription(new (::system::Subscription^)
{
    var <bind-cache>1 : ::vm::IVViewModel^ = null of (::vm::IVViewModel^);

    var <bind-handler>1_0 : ::system::reflection::EventHandler^ = null of (::system::reflection::EventHandler^);

    var <bind-cache>2 : ::vm::IVViewModel^ = null of (::vm::IVViewModel^);

    var <bind-handler>2_0 : ::system::reflection::EventHandler^ = null of (::system::reflection::EventHandler^);

    var <bind-cache>3 : ::helloworld::MainWindow* = null of (::helloworld::MainWindow*);

    var <bind-handler>3_0 : ::system::reflection::EventHandler^ = null of (::system::reflection::EventHandler^);

    var <bind-opened> : ::system::Boolean = (cast (::system::Boolean) "false");

    var <bind-closed> : ::system::Boolean = (cast (::system::Boolean) "false");

    var <bind-listeners> : func (::system::Object) : (::system::Void)[::system::Listener^] = {};

    func <bind-activator>() : (::system::Void)
    {
        var <bind-activator-result> = (((!<bind-cache>3.HasLoggedIn) and (<bind-cache>2.UserNameError == "") and (<bind-cache>1.PasswordError == "")));
        for (<bind-callback> in <bind-listeners>.Values)
        {
            (cast (func (::system::Object) : (::system::Void)) <bind-callback>)(<bind-activator-result>);
        }
    }

    func <bind-callback>1_0() : (::system::Void)
    {
        <bind-activator>();
    }

    func <bind-callback>2_0() : (::system::Void)
    {
        <bind-activator>();
    }

    func <bind-callback>3_0() : (::system::Void)
    {
        <bind-activator>();
    }

    func <bind-initialize>() : (::system::Void)
    {
        (<bind-cache>3 = self);
        (<bind-cache>2 = ViewModel);
        (<bind-cache>1 = ViewModel);
        (<bind-handler>1_0 = attach(<bind-cache>1.PasswordErrorChanged, <bind-callback>1_0));
        (<bind-handler>2_0 = attach(<bind-cache>2.UserNameErrorChanged, <bind-callback>2_0));
        (<bind-handler>3_0 = attach(<bind-cache>3.HasLoggedInChanged, <bind-callback>3_0));
    }
}

```

```

}

override func Subscribe(<bind-callback> : func (::system::Object) : (::system::Void)) : (::system::Listener^)
{
    if (!<bind-opened>)
    {
        <bind-opened> = true;
        <bind-initialize>();
    }
    var <subscription> : ::system::Subscription* = this;
    var <listener-shared> = new (::system::Listener^)
    {
        override func GetSubscription() : (::system::Subscription*)
        {
            return <subscription>;
        }

        override func GetStopped() : (::system::Boolean)
        {
            return (!<bind-listeners>.Keys.Contains(this));
        }

        override func StopListening() : (::system::Boolean)
        {
            if (<bind-listeners>.Keys.Contains(this))
            {
                <bind-listeners>.Remove(this);
                return true;
            }
            return false;
        }
    };
    <bind-listeners>.Set(<listener-shared>, <bind-callback>);
    return <listener-shared>;
}

override func Update() : (::system::Boolean)
{
    if (!<bind-closed>)
    {
        <bind-activator>();
        return true;
    }
    return false;
}

override func Close() : (::system::Boolean)
{
    if (!<bind-closed>)
    {
        (<bind-closed> = true);
        detach(<bind-handler>1_0);
        detach(<bind-handler>2_0);
        detach(<bind-handler>3_0);
        (<bind-cache>1 = null of (::vm::IViewModel^));
        (<bind-cache>2 = null of (::vm::IViewModel^));
        (<bind-cache>3 = null of (:helloworld::MainWindow*));
        (<bind-handler>1_0 = null of (::system::reflection::EventHandler^));
        (<bind-handler>2_0 = null of (::system::reflection::EventHandler^));
        (<bind-handler>3_0 = null of (::system::reflection::EventHandler^));
        <bind-listeners>.Clear();
        return true;
    }
    return false;
}
});
<created-subscription>.Subscribe(func <value> : ::system::Object) : (::system::Void)
{
    var <old> = buttonSignUp.Enabled;
    var <new> = (cast (::system::Boolean) <value>);
    if ((<old> == <new>))
    {
        return;
    }
    (buttonSignUp.Enabled = <new>);
});
<created-subscription>.Update();
}
{
    attach(buttonSignUp.Clicked, [<this>.buttonSignUp_Clicked($1, $2)]);
}
{
    var <event-handler> = func (sender : ::presentation::compositions::GuiGraphicsComposition*, arguments : ::presentation::compositions::GuiEventArgs*) : (::system::Void)
    {
        self.Close();
    };
    attach(buttonCancel.Clicked, <event-handler>);
}
{
    var <created-subscription> = <this>.AddSubscription(new (::system::Subscription^)
    {
        var <bind-cache>1 : ::presentation::controls::GuiSinglelineTextBox* = null of (::presentation::controls::GuiSinglelineTextBox*);

        var <bind-handler>1_0 : ::system::reflection::EventHandler^ = null of (::system::reflection::EventHandler^);

        var <bind-opened> : ::system::Boolean = (cast (::system::Boolean) "false");

        var <bind-closed> : ::system::Boolean = (cast (::system::Boolean) "false");

        var <bind-listeners> : func (::system::Object) : (::system::Void)[::system::Listener^] = {};

        func <bind-activator>() : (::system::Void)
        {
            var <bind-activator-result> = <bind-cache>1.Text;
            for (<bind-callback> in <bind-listeners>.Values)
            {
                (cast (func (::system::Object) : (::system::Void)) <bind-callback>)(<bind-activator-result>);
            }
        }

        func <bind-callback>1_0(<bind-callback-argument>0 : ::presentation::compositions::GuiGraphicsComposition*, <bind-callback-argument>1 : ::presentation::compositions::G
        {
            <bind-activator>();
        }

        func <bind-initialize>() : (::system::Void)
        {
            (<bind-cache>1 = textBoxUserName);
            (<bind-handler>1_0 = attach(<bind-cache>1.TextChanged, <bind-callback>1_0));
        }
    });
    override func Subscribe(<bind-callback> : func (::system::Object) : (::system::Void)) : (::system::Listener^)
    {
        if (!<bind-opened>)
        {
            <bind-opened> = true;
            <bind-initialize>();
        }
        var <subscription> : ::system::Subscription* = this;
        var <listener-shared> = new (::system::Listener^)
    }
}

```

```

{
    override func GetSubscription() : (::system::Subscription*)
    {
        return <subscription>;
    }

    override func GetStopped() : (::system::Boolean)
    {
        return (!<bind-listeners>.Keys.Contains(this));
    }

    override func StopListening() : (::system::Boolean)
    {
        if (<bind-listeners>.Keys.Contains(this))
        {
            <bind-listeners>.Remove(this);
            return true;
        }
        return false;
    }
};

<bind-listeners>.Set(<listener-shared>, <bind-callback>);

return <listener-shared>;
}

override func Update() : (::system::Boolean)
{
    if (!!<bind-closed>)
    {
        <bind-activator>();
        return true;
    }
    return false;
}

override func Close() : (::system::Boolean)
{
    if (!!<bind-closed>)
    {
        (<bind-closed> = true);
        detach(<bind-handler1_0>);
        (<bind-cache>1 = null of (::presentation::controls::GuiSinglelineTextBox*));
        (<bind-handler>1_0 = null of (::system::reflection::EventHandler^));
        <bind-listeners>.Clear();
        return true;
    }
    return false;
}
};

<created-subscription>.Subscribe(func (<value> : ::system::Object) : (::system::Void)
{
    var <old> = <precompile>18.UserName;
    var <new> = (cast (::system::String) <value>);
    if ((<old> == <new>))
    {
        return;
    }
    <precompile>18.UserName = <new>;
});
<created-subscription>.Update();
}

var <created-subscription> = <this>.AddSubscription(new (::system::Subscription^)
{
    var <bind-cache>1 : ::presentation::controls::GuiSinglelineTextBox* = null of (::presentation::controls::GuiSinglelineTextBox*);
    var <bind-handler>1_0 : ::system::reflection::EventHandler^ = null of (::system::reflection::EventHandler^);
    var <bind-opened> : ::system::Boolean = (cast (::system::Boolean) "false");
    var <bind-closed> : ::system::Boolean = (cast (::system::Boolean) "false");
    var <bind-listeners> : func (::system::Object) : (::system::Void)[::system::Listener^] = {};

    func <bind-activator>() : (::system::Void)
    {
        var <bind-activator-result> = <bind-cache>1.Text;
        for (<bind-callback> in <bind-listeners>.Values)
        {
            (cast (func (::system::Object) : (::system::Void)) <bind-callback>)(<bind-activator-result>);
        }
    }

    func <bind-callback>1_0(<bind-callback-argument>0 : ::presentation::compositions::GuiGraphicsComposition*, <bind-callback-argument>1 : ::presentation::compositions::Gu
    {
        <bind-activator>();
    }

    func <bind-initialize>() : (::system::Void)
    {
        (<bind-cache>1 = textBoxPassword);
        (<bind-handler>1_0 = attach(<bind-cache>1.TextChanged, <bind-callback>1_0));
    }

    override func Subscribe(<bind-callback> : func (::system::Object) : (::system::Void)) : (::system::Listener^)
    {
        if (!!<bind-opened>)
        {
            (<bind-opened> = true);
            <bind-initialize>();
        }
        var <subscription> : ::system::Subscription* = this;
        var <listener-shared> = new (::system::Listener^)
        {
            override func GetSubscription() : (::system::Subscription*)
            {
                return <subscription>;
            }

            override func GetStopped() : (::system::Boolean)
            {
                return (!<bind-listeners>.Keys.Contains(this));
            }

            override func StopListening() : (::system::Boolean)
            {
                if (<bind-listeners>.Keys.Contains(this))
                {
                    <bind-listeners>.Remove(this);
                    return true;
                }
                return false;
            }
        };
        <bind-listeners>.Set(<listener-shared>, <bind-callback>);
        return <listener-shared>;
    }

    override func Update() : (::system::Boolean)
    {

```

```

    if ((!<bind-closed>))
    {
        <bind-activator>();
        return true;
    }
    return false;
}

override func Close() : (::system::Boolean)
{
    if ((!<bind-closed>))
    {
        <bind-closed> = true;
        detach(<bind-handler>_0);
        (<bind-cache>1 = null of (::presentation::controls::GuiSinglelineTextBox*));
        (<bind-handler>1_0 = null of (::system::reflection::EventHandler^));
        <bind-listeners>.Clear();
        return true;
    }
    return false;
}
});

<created-subscription>.Subscribe(func (<value> : ::system::Object) : (::system::Void)
{
    var <old> = <precompile>18.Password;
    var <new> = (cast (::system::String) <value>);
    if ((<old> == <new>))
    {
        return;
    }
    (<precompile>18.Password = <new>);
    <created-subscription>.Update();
}
}
}

```

于是最后就可以出这么个东西了：



这个脚本语言我是编译成字节码（当然不是JVM的那个），保存在了GacGen.exe生成的二进制资源文件里面。于是你的程序启动就要加载它。这个脚本语言调用C++的类，因此GacUI的类就得有反射。GacUIReflection.h/cpp里面的代码正是用来支持反射的，所以需要那么多的空间。

今天为止我做的工作去掉了一部分内容，主要是把Workflow.h/cpp和GacUIReflection.h/cpp里面编译器的部分拿了出来，变成了WorkflowCompiler.h/cpp和GacUICompiler.h/cpp。这些代码是GacGen.exe需要的，但是最终的GacUI应用程序不需要。这样就分别在Debug和Release都减少了大约10M的体积。

以后把脚本编译成了C++代码，那么GacUIReflection.h/cpp和Workflow.h/cpp也不需要了，Release的GacUI体积将进一步减少到1.6M左右。已经不能再减下去了。本来说这个功能要清明节上线，不过现在看起来，多半要到儿童节了……

偶然找到了大二的时候做的OpenGL_GUI

时光飞逝啊，都快10年了。

IT人才全球化大势所趋，我们需要成为先行者！

作者：刘老师

链接：[我为什么创办【直通硅谷】培训机构？ - 刘老师的回答](#)

来源：知乎

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

2014年年中的时候，有国内的学弟问我：“学长，我家的经济条件不允许我出国留学，但是我很想去像Google这样世界最顶尖的科技公司工作，哪怕第一年不赚钱，只是学习，也很想去。我这样的条件有机会么？”

而20多年前，一家当时来看，无比伟大的教育公司，新东方，在俞敏洪的地下教室里创立。那时，很少有人知道外面的世界是什么样子的。更很少有人敢横下一条心，将出国留学作为自己人生的一个重要阶段，以此为目标去奋斗。

随着中国经济的发展，大概从2006, 07年起，出国留学开始平民化。很多学生，已经将出国留学作为人生规划的重要部分。也正是这十年，中国的互联网企业的发展，突飞猛进。科技上逐渐与世界强国接轨，尽管还落后一段距离，但世界信息的通畅，已是历史之最。换句话说，二十年前，如果Apple发布了一款新电脑，在遥远的中国，没有人也没有足够的信息渠道去了解这件事情。十年前，Apple发布一款手机，我们最多只能到门户网站，去了解信息的详情。而很有可能，这个信息还要延迟两三天才能抵达世界的这一端。现在，如果我们喜欢科技信息，只需要简单的关注相关微信平台，每天，就有源源不断的信息流入我们的手机中，流入我们的意识里。

这20年，是让人咋舌的20年。从中国的经济发展，科技发展到中国人的意识形态，已经发生了翻天覆地的变化。

那么未来的20年呢？

说回2014年年中，我和我的Google和Linkedin的小伙伴闲聊时，聊起了科技人才的流动趋势的话题。一个很明显的事实是，硅谷，这里除了制度还是美国的制度外，从人口构成来看，可以说是一个大杂烩城市。20年前，仅有的几十万分之一的亚洲人中，如果做统计的话，可能多半来自于台湾香港，来自于日韩。而现在，中国大陆人几乎占到了这里科技人才的五或者六分之一。而美国政府，不断的试图在控制人口流入与科技产业发展之中寻找平衡点。一个现实的问题是，停止高科技人才的引入，美国的科技将必受重创。因此，过去20年，在经历了2000年IT泡沫危机H1名额回归正常之后，自2004年开始，美国政府先是增加了20000个名额给advanced degree所有者，又将STEM的OPT时长从08年前的一年变为之后的29个月，又在最近延期到了三年。也就是说，美国政府正在用时间换取概率，帮助高科技人才增加拿到工作签证的机会。除了美国之外，澳洲，加拿大，欧洲各国，都对科技界人才求贤若渴。纷纷开放工作签证，试图吸引世界各地科技人才。

那么，这一切又与中国有什么关系呢？作为泱泱大国，年青一代的精英，所具备的智慧与勤奋这两点品质，已经给全世界各大科技公司留下了非常深刻的印象。因此，对于人才引进，各个国家的主流观点基本都是持积极态度。如果说，过去的二十年，是全球制造业，地产业，金融业告诉发展的二十年。

那么未来二十年，一定是科技突飞猛进的二十年。换言之，一定是科技人才全球化的二十年。

【直通硅谷】在14年应运而生，15年初正式成立。但现在的中国高科技人才，对于直接出国，去顶尖科技公司的看法，就好像十几年前对留学的态度。认为这是只有极其个别的人才能办到的事情。而事实上，每年有成千上万的科技人才，正是以这种方式，远赴美国，欧洲，澳洲，在各个高科技公司中崭露头角。当然，比起动辄每年50万人的留学大军，这点显得微不足道。而五年之后，随着各国对科技人才工作以及移民态度的放宽，这将是一个必然的发展趋势。

那么，【直通硅谷】在做什么？

一句话，引领中国科技人才全球化，就像新东方培训托福一样，我们要让这个过程更简单。除了培训外，信息的搜集和从头到尾的一体化辅助服务，让科技人才有一个心里依靠，无论到了什么时候，遇到什么困难，我们都在。

为什么要这么做呢？

出国留学成井喷式发展的那几年间，很多经济学家质疑，认为留学为发达国家带来了巨大的经济利益。人们一度也因为财富分配不均造成孩子上学机会的不同而对这个现象大为不满。从06年至今，10年过去了。我所能看到的是，尽管在脚踏实地与服务精神方面，中国的科技人才仍与美国，德国等世界强国具有一定距离。但是，必须承认的是，科技意识和整体的意识导向上，中国的进步是巨大的。试问，如果没有这一批批出国留学的留学党，有的在“海龟”后把学到的知识奉献给了祖国，有的在美国海外就拉着国内的校友做企业做事情，有的不断的给国内的朋友分享自己的见闻。潜移默化之中，我们确实交了学费，但是更换回了，用金钱本身换不来的意识与素质进步。

那么科技人才全球化呢？【直通硅谷】为什么认为这是利国利民的？

留学尚且如此，一个科技人才到了顶级的科技公司工作，所见所闻所感，都是在本土无法相比的。长远来看，无论从世界角度的话语主导权还是中国角度的知识获取方面，都是有百益而无一害。甚至只从经济角度来说，很多我认识的朋友，到最后都是，赚的是高薪美金，花的是人民币。

我们的想法很简单，为“中国的科技人才引领世界科技潮流”这个目标，尽自己的一份力。不久的未来，等我们的能力达到了的时候，我们会让这一过程，逐渐亲民化平民化。帮助您完成梦想，也是我们的梦想。

最后，是一位我们的前学员的进阶经历，希望能给看到这篇文章的，有梦想的朋友，一点点鼓励与自信。

http://mp.weixin.qq.com/s?__biz=MzA4MDIyNDExNg==&mid=2650916251&idx=1&sn=a5d7728fd3b8968d752bdb1387ae8044&scene=0#wechat_redirect

【直通硅谷】详细信息：

http://mp.weixin.qq.com/s?__biz=MzA4OTA0MjE1NA==&mid=506819712&idx=1&sn=cb2b02c4574411826e57117b9482b89b&scene=25#wechat_redirect

计划赶不上变化

由于最近公司年底（6月份）review 涨工资，然后要去旅游，然后父母要过来，眼看着我给自己设定的 GacUI 激动人心的 feature 的 deadline 又要跳票了。一个人做项目就是慢啊……

于是下个星期要带老婆一起去匹兹堡，估计能吃饭的时间也不多，有没有人想来吃饭？萌妹纸的话就更好了！

终于拿下了知乎回答最多的成就！

喜闻乐见
大快人心
普天同庆
奔走相告

应邀开通了值乎3.0

开通的过程中我对这个破东西的感觉就很不好，在Windows Phone上面“我的”面板死活点不出来，还得去PC版里面点才行。我对此感到很愤怒，所以把定价设置在了¥100，反正也不会有人来问我什么靠谱的问题的。

当然如果你们觉得就算给¥100也一定要跟我谈人生谈理想的话，我也不会阻止你们的，15刀也就只能吃 John Howie 牛排上面的一根菜（逃

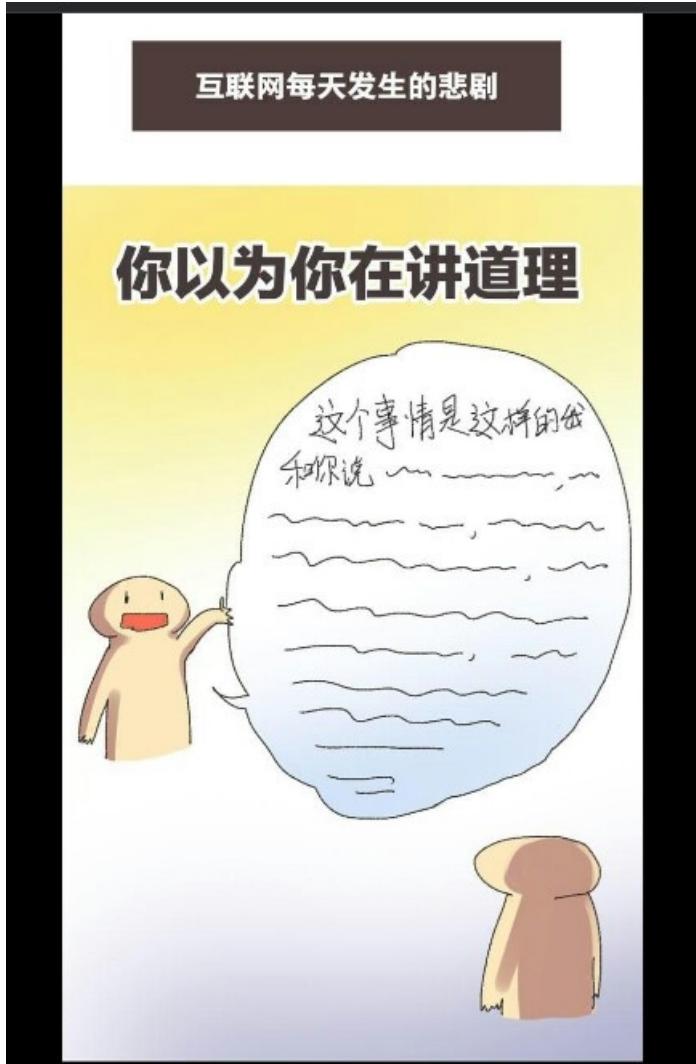
=====更新=====

卧槽，我就随便这么一说，真的有人来提问了，我来研究一下PC版微信能不能录音……

=====更新=====

结论是不能，好吧等老婆明天睡醒了我拿她的iPhone来讲……

第一次遇到，adj跟n连着用的时候，会觉得adj∈n的人。



在别人看来



哦！这里有
喷点！



我要
骂死这
个傻逼
！！

© 搞漫画

7 vczh , 专业造轮子 , 前排已拉黑。... 修改话题经验

蒙奇 D路飞 等 7 人赞同

procedure放在haskell里面大概就是返回某个()的monad类型 (譬如说IO ()) 的函数 修改

发布于 2016-06-13 收起评论 分享 收藏 设置 · 作者保留权利



匿名用户 (提问者)

轮子哥秒答，原来其实是等价的么。非常感谢！

6 天前



问题少年

轮子哥，这是不是专业的答案啊？感觉怎么没人呢？

6 天前



[REDACTED] monad不是类型。应该说是返回某个F ()，其中F符合monad定义

5小时前



vczh (作者) 回复 [REDACTED]

查看对话

不要抠字眼，我指的是所有的instance

32分钟前



[REDACTED] 回复 vczh (作者)

查看对话

根本不是抠字眼，而是你定义错误，就算所有的instance，monad也不是类型

31分钟前



vczh (作者) 回复 [REDACTED]

查看对话

我没有说monad是类型啊，这是一个定语从句，所以才说你是抠字眼

7分钟前



[REDACTED] 回复 vczh (作者)

查看对话

就算你找到奇怪的抠字眼方法把类型赋给不知道什么鬼，你上一回复也暴露了

5分钟前



vczh (作者) 回复 [REDACTED]

查看对话

我的“上一回复”就是在直到你没有把他看成定语从句的前提下，试图跟你说一下，

结果你竟然没有听明白。

3分钟前



vczh (作者) 回复 [REDACTED]

查看对话

我来简单的跟你解释一下，monad类型的语法结构就像“红球”，你既然说我的意思

是monad都是类型，那你也应该接受红色就是球。

第一次遇到，adj跟n连着用的时候，会觉得adj \in n的人。

我也来开启专栏的乞讨功能

我的专栏大部分都是自吹自擂，所以就先在[为什么我们需要学习（设计）模式](#)、[靠谱的代码和DRY](#)和[如何成为牛逼的程序员](#)等具有教育意义的文章的尾部开始乞讨。要是真的有用的话，以后我再把私信里面老是有人问的重复问题写在这里。

当然你们沉不住气的话，还是可以来知乎里面问，但是不要老是问“发什么样的照片我才会点赞”的问题，这是商业机密，我怎能随随便便告诉你们！

之前一直有人问这个照片是不是我，这当然是我，刚刚参加工作在机场的时候拍的，哈哈哈哈。

也谈谈装配脑袋

我跟脑袋认识已经有好几年了。最近听闻他去世的消息，深感震惊和惋惜。震惊是因为脑袋最近一次的消息还是半个月前，说他老爸的干细胞已经几乎完全代替他自己的了，最后却发生了排异。惋惜是说，我以前在的微软亚洲研究院的 IEG 组又损失了一名大将。使用微软的说法，这种有 unique skill set 的人得来不易，主要都要看缘分。

如果要我给脑袋贴标签的话，那肯定不会包含编程的东西。脑袋是一个热爱化学的人，从小就凭借着竞赛一路保送到了南京大学（听说是[@刘未鹏](#)的师兄），Nirputer 也是他高中跟几个朋友在一起搞的计算机爱好者组织的名字。后来脑袋跟我说，之所以做了码农，是因为当时英语成绩不好，无法念博士，也就搞不了化学了。毕业后加入了一个后来很快就倒闭了的创业公司，被坑了一票大的，于是就在2006年愤而离职，加入了微软。

我跟脑袋很久以前就在博客园有过交流，只不过他早期在那上面也就写写一些.net的教程，所以并没有太留意。后来是因为我去北京工作了，听闻他就在我楼下，所以交往才频繁了起来。那个时候脑袋还在给Office的中国分舵工作，结果被老板坑了一票大的。我就问他要不要来MSRA啊，他说还想再试试。于是就在同一个部门下面换了一个组，又被老板坑了一票大的，最后只好来了MSRA。来了之后，脑袋觉得这个部门实在是太好了，于是拼命工作，没事就加班，觉得很开心。于是我知道原来微软里面也有这么多不堪入目的事情，庆幸我在微软一路走来遇到的几个老板，甚至是现在加入的Office总舵的老板，还都算是好老板。

说起跟脑袋到底都交谈些什么，其实除了他的VBF和我的一些开源项目以外，主要还是跟编程无关的东西。譬如说他的元素单质收藏，一些物理化学的知识，还有发现了已经看不懂的当年的大学作业，或者参加了同学会之后，发现同学都做到了厦大的教授，搞研究的搞研究出国的出国，只有他还是个码农的事情什么的。

不过你们别看脑袋过得这么惨，他可有钱了。就不说读书的时候可以买正版VB6.0、床下三大箱正版游戏、Steam几页游戏买了不玩的事情了，脑袋从年轻的时候开始，就剁摄像器材，器材和技术的专业程度一点都不比私房圈的摄像要差，公司活动要录像也是他做的后期。人们都说，摄影穷三代，单反毁一生。脑袋不仅没穷，钱却好像越来越多，继续剁电脑。在我刚工作后不久买i7-920，觉得自己简直有钱的一逼的时候，脑袋电脑的声卡就是我的电脑的两倍价钱，而且过后不久就把i7-920退给了老爸上网，自己组了台新的电脑，还有水冷。不过每次跟他吃饭的时候，都说这个贵不吃那个贵不吃……

而且脑袋还有一个收集元素单质的爱好。他就算是气体单质，也自己烧玻璃瓶，装了进去，还通电发光拍照给我们看，拍得比教科书上的还漂亮，据说他后来所有可以收集的元素全都有了，而且每一个元素的各种形态都差不多全了，什么碳还分石墨和钻石啊，不过C60好像没到手。这些东西也是要剁手的，脑袋就经常跟我说，他又买了一块小铁块，比金还贵好几倍，还打磨的特别漂亮什么的。后来有一次吃饭的时候他提到，他的愿望就是，以后买一座大房子，客厅竖一道玻璃墙，上面雕刻一个元素周期表，每一个格子都把他收集的单质放进去。

当然这个喜好也是很危险的。记得几年前有一次他跟我说，他在房间里发现了一大瓶盖子已经破了的水银，赶紧就剁了一个容器重新装起来。后来脑袋胆子越来越大，而且对自己的技术很有信心，开始作大死，还玩出了翔。当然具体干了什么，你们问我我也不会告诉你们的……

两年前我离开了MSRA，人肉翻墙来到了西雅图，过了一年就听闻他得了白血病的事。一开始各种指标都很有希望，没想到突然就去世了。元素周期墙的愿望，可惜再也不能实现了。

谨此纪念天国的装配脑袋。

SEP 19, 2016

最新评论拉黑标准（随时不及时更改）

这些知友们真是不长眼睛啊，都说了那么多遍了.....

1. 前排后排、一楼二楼、沙发板凳等
2. 骂我的、骂别人语气不够有教养的
3. 苟
4. 跟答主距离这么近、怎么还不睡、起床好早啊、你怎么这么有空、老是逃、怎么不逃
5. 跟以上所有内容在语义上等价、相关、类似的东西
6. 跟评论里面已经存在的无意义评论类似的评论
7. 我说的是别人，但是评论询问我是否也发生在我身上的
 1. 说出你的故事
 2. 你是不是也xxx
 3. 这就是你xxx的原因吗
 4. 等
8. 不仅语气没有讨好我，而且同时还
 1. 说我答案里中英混杂/喜欢用“(逃”
 2. **(最新) 叫我不要回答不擅长的问题 (AUG 24, 2017)**
9. 其它我看不顺眼但是一下子没有归类出来的

C++奇技淫巧：通过无脑字符串替换的方法，来把一个递归函数改写成非递归函数

首先声明，学会了这个对你们的C++编程知识没有任何帮助，还有可能走火入魔，初学者自行关闭浏览器（逃

1、准备好一个递归的函数。这个函数不能有返回值和嵌套的局部变量，不能再循环里面递归。如果有的话，做如下改动：

- 如果函数里面有一些定义在大括号里面的变量的话，先改写成所有变量都放在最前面的形式，就像C预言所要求的一样。如果变量类型刚好没有缺省构造函数的话，自己想办法（逃
- 如果有返回值，改写成通过最后一个指针参数返回，反正递归函数必须返回void类型。
- 如果你有一个循环体递归调用了自己的话，把循环改成goto。

这里的例子是一个中序遍历函数

```
struct Tree
{
    int data;
    unique_ptr<Tree> left;
    unique_ptr<Tree> right;
};

void PrintTree(const unique_ptr<Tree>& tree)
{
    if (tree->left)
    {
        PrintTree(tree->left);
    }
    cout << tree->data << " ";
    if (tree->right)
    {
        PrintTree(tree->right);
    }
}
```

2、把所有的变量都放在一个struct里面，额外加上_var_state。当然在这里，只有tree一个变量。

```
struct PrintTreeVariables
{
    const unique_ptr<Tree>& tree;
    int _var_state;
};
```

3、把固定的代码插入到函数的最前面和最后面，并在所有递归调用的地方标记上数字，下标从1开始。

```
void PrintTree(const unique_ptr<Tree>& tree)
{
/* 最前面的代码，要在正确的位置写上函数的参数 */
stack<PrintTreeVariables> _var_stack;
{
    PrintTreeVariables _var{ tree, -1 };
    _var_stack.push(_var);
}
int _var_state = 0;

_var_func_begin:
switch (_var_state)
{
case -1:
    return;
case 0:
/* 刚才的函数体，所有的递归已经被标记上了数字 */
    if (tree->left)
    {
        PrintTree(tree->left); /* 1 */
    }
    cout << tree->data << " ";
    if (tree->right)
    {
        PrintTree(tree->right); /* 2 */
    }
/* 最后面的代码 */
}
_var_state = _var_stack.top()._var_state;
_var_stack.pop();
```

```
goto _var_func_begin;
}
```

4、在所有的数字标记的地方插入固定的代码：“case <数字>;”

```
void PrintTree(const unique_ptr<Tree>& tree)
{
/* 最前面的代码 */
stack<PrintTreeVariables> _var_stack;
{
PrintTreeVariables _var{ tree, -1 };
_var_stack.push(_var);
}
int _var_state = 0;

_var_func_begin:
switch (_var_state)
{
case -1:;
return;
case 0:;
/* 刚才的函数体，所有的递归已经被标记上了数字 */
if (tree->left)
{
PrintTree(tree->left); /* 1 */
case 1:;
}
cout << tree->data << " ";
if (tree->right)
{
PrintTree(tree->right); /* 2 */
case 2:;
}
/* 最后面的代码 */
}
_var_state = _var_stack.top()._var_state;
_var_stack.pop();
goto _var_func_begin;
}
```

5、这就是最有意思的一步了：

a) 把所有的变量的使用都在前面加上“_var_stack.top().”

b) 把所有的return语句都改成函数的最后三行代码。当然这里有一个缺陷，如果原本函数的循环刚好跨了case，那么使用了break语句就会在这里傻逼。这就是为什么有文章开头的条件。

c) 把所有的递归调用

```
PrintTree(x)
```

都替换成这样的代码（要在正确的位置写上函数的参数）：

```
{
PrintTreeVariables _var{ x, <数字> };
_var_stack.push(_var);
_var_state = 0;
goto _var_func_begin;
}
```

这个数字就是我们给出来的标记，然后函数会变成这样：

```
void PrintTree(const unique_ptr<Tree>& tree)
{
stack<PrintTreeVariables> _var_stack;
{
PrintTreeVariables _var{ tree, -1 };
_var_stack.push(_var);
}
int _var_state = 0;

_var_func_begin:
switch (_var_state)
{
case -1:;
return;
case 0;;
if (_var_stack.top().tree->left)
{
```

```
{  
    PrintTreeVariables _var{ _var_stack.top().tree->left, 1 };  
    _var_stack.push(_var);  
    _var_state = 0;  
    goto _var_func_begin;  
}  
case 1::  
{  
    cout << _var_stack.top().tree->data << " ";  
    if (_var_stack.top().tree->right)  
    {  
        PrintTreeVariables _var{ _var_stack.top().tree->right, 2 };  
        _var_stack.push(_var);  
        _var_state = 0;  
        goto _var_func_begin;  
    }  
case 2::  
{  
    _var_state = _var_stack.top()._var_state;  
    _var_stack.pop();  
    goto _var_func_begin;  
}
```

6、F5运行！

基于现代C++的四则运算语法分析器（上）

大约八年前，轮子哥曾经写过一篇文章，论述了如何用当时的C++来实现一个四则运算语法分析器——它会把四则运算表达式创建为一颗语法树（AST），然后将其转换为LISP语言风格的表达式。文章地址如下：[如何手写语法分析器](#)

最近一段时间，因为工作上某些暂时不能说的原因，我在这篇文章的基础上又做了一点工作——主要是用C++ 11的写法重写了这篇文章中的代码，并给出了AST的声明。这里把目前不需要遮盖的部分放出来，也算是我在续写的第一篇技术类文章了……

本文将分为上下两个部分——上半部分会以Expression类的形式，给出AST在C++中的声明，而下半部分将使用C++ 11，重写原文中的语法分析器，使其可以用上半部分给出的声明，建立AST的模型。

首先，对于一个四则运算表达式（形如 $1*(2+3)$ ），我们有若干条语法规约。以下引用原文：

```
Operator="+"  
Operator="-"  
Operator="*"  
Operator="/"  
Expression=<数字>  
Expression=( Operator Expression Expression )  
Expression=( Expression )
```

这样写的话觉得很烦，我们可以追加多两种定义语法的语法：

- 1、A | B代表A或者B都可以，并且如果字符串被A匹配成功的话将不会考虑B
- 2、[A]代表A是可以存在或者不存在的，但是尽量使其存在

于是我们可以把上面的语法改写成如下形式：

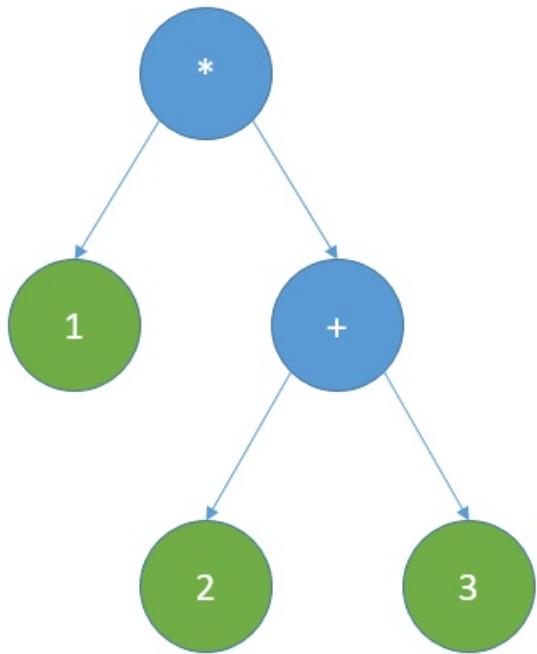
1. Operator="+" | "-" | "*" | "/"
2. Expression=<数字> | (" Expression ")" | "(" Operator Expression Expression ")"

第一条语法规则说的是Operator，也就是操作符，可以是加号、减号、乘号或者除号。第二条语法规则说的是一条表达式可以只由数字构成、一个加了括号的表达式或者一个加上了括号的操作符和两个参数。

需要注意的是，对于本文来说，我们直接使用四则运算表达式原本的格式（而非原文的LISP格式），所以对于语法规则2，应该改为这样的形式：

```
Expression=<数字> | (" Expression ")" | (" Expression Operator Expression ")"
```

基于这样的形式，我们可以把表达式简单地分成两种类型：**数字表达式**，和**二元运算表达式**。它们恰好对应语法分析器所得到的语法树上的两种不同的节点——以上文提到的表达式 $1*(2+3)$ 为例，如图所示：

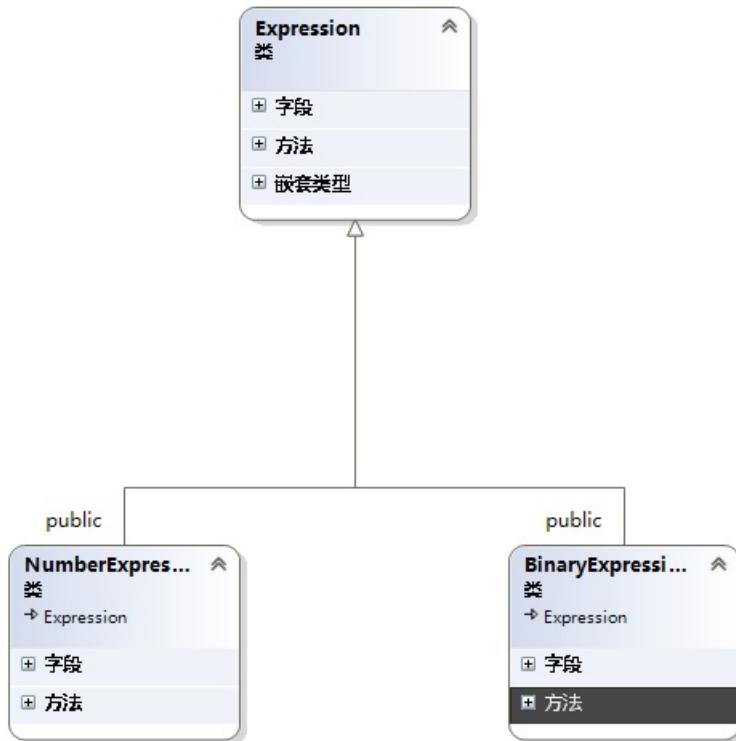


图中蓝色结点对应二元运算表达式，而绿色节点对应数字表达式。从语法树上，我们可以看到，对于四则运算表达式，其语法树有这么几个特点：

- 语法树恰好是一颗二叉树。
- 一个数字表达式的节点一定是一个叶子节点，没有左右孩子节点。
- 而一个二元运算表达式，则恰好相反——它有，且一定有两个孩子节点（因为二元运算符号左右两边必然都要有表达式）。
- 这两个叶子节点的类型，是笼统的“表达式”，既可以是二元运算表达式，也可以是数字表达式。

此外，不难看出对于两个具有父子关系的节点来说，子节点的运算优先级要高于父节点——这个运算优先级，跟数学上的四则运算的运算优先级规则完全一致。

基于此，我们可以很容易地采用这样一种方式，设计两种表达式在C++中的类的结构：



我们首先定义一个**Expression**类作为基类——这是所有表达式类型的公共基类，然后再在基类的基础上，对其进行扩展，得到两种表达式派生类：**NumberExpression**，表示数字表达式，以及**BinaryExpression**，表示二元运算表达式。

这里我们首先给出**Expression**类的基础代码——里面目前还是空的：

```
class Expression
{
public:
};
```

接下来我们给出**NumberExpression**类的定义——为了方便起见，这里把所有成员都设置为公有：

```
class NumberExpression :public Expression
{
public:
    int Value;
    NumberExpression(int number)
    {
        Value = number;
    }
};
```

我们可以看到，**NumberExpression**只有一个整型成员**value**（这里只考虑整数），以及对应的构造函数。

最后我们给出**BinaryExpression**的定义：

```
class BinaryExpression :public Expression
{
public:
    shared_ptr<Expression> First;
    shared_ptr<Expression> Second;
    BinaryOperator Op;
    BinaryExpression(BinaryOperator theOp, shared_ptr<Expression> theLeft, shared_ptr<Expression> theRight) :
        Op(theOp), First(theLeft), Second(theRight) {
    }
};
```

我们可以看到，**BinaryExpression**的成员包括运算符**Op**，以及两个表达式成员**First**和**Second**——《如何》原文使用的是裸指针，而这里使用了C++11提供的智能指针**shared_ptr**。正如我们上文所说，**BinaryExpression**的孩子节点，既可以是数字，也可以是二

元运算，所以这里shared_ptr对应的类型，只能是两种表达式的基类Expression——这样就可以确保孩子节点的类型不受限制。

BinaryExpression的构造函数使用初始化列表定义三个成员变量，其中Op是一个定义好的enum class，代码如下：

```
enum class BinaryOperator
{
    Plus,
    Minus,
    Multiply,
    Divide,
};
```

到这一步，对AST的声明所需的三种Expression类的构造，就基本完成了——注意是“基本”，因为我们还有些工作，不能就这么结束。

毫无疑问，在AST的构造过程中，会在堆上建立对象——《如何》原文手动使用delete来负责回收内存，而这里将全程使用shared_ptr，所以不需要手动回收内存，堆对象什么时候需要回收，由shared_ptr进行管理。但是这样一来，就引出了一个问题——以我们目前的代码，当一个BinaryExpression结点对象需要析构的话，会发生什么呢？

我们知道，当我们需要对一个类的对象进行析构的时候，我们还需要对这个类的成员对象（如果有）进行析构——比如说，对于以下代码：

```
class Base
{
};

class Derived : Base
{
    string s {"This is a very long string that force std::string to allocate a buffer."};
};
```

这里std::string初始化的时候保存了一个非常长的字符串，所以它就必须申请额外的内存来保存它。如果我们现在要销毁Derived类的某个实例对象的话，那么我们就必须首先销毁它的std::string成员对象。

本来，这个过程是由Derived类的析构函数自动完成的——它在调用的时候，首先会调用std::string的析构函数来回收它放在堆内存上的字符串，然后才会销毁自己。整个过程非常安全，不会产生内存泄露——如下代码所示：

```
Derived* obj = new Derived;
delete obj;
```

但是这个安全的过程，有一个非常重要的前提条件：我们调用的是**Derived自己的析构函数**。当这个条件不满足，比如这样的情况，又会怎么样呢？

```
Base* obj = new Derived;
delete obj;
```

毫无疑问，delete语句不可能知道obj到底是不是Derived，所以就只调用了Base的析构函数。这完全是因为我们new Derived后把指针的类型转换成了Base*。

在这种情况下，由于我们只调用了Base函数的析构函数，却没有调用Derived类的析构函数，所以字符串就没有释放。而delete了之后，我们也丢掉了对这个对象的引用——于是就产生了内存泄露，堆上的字符串没人管了。

现在，再回过头来看我们的BinaryExpression类——大家不难看出问题在哪。没错，为了保证First和Second可以指向两种表达式类，我们将其定义成了Expression类型的shared_ptr——于是在目前的情况下，就会出现析构的时候无法调用正确的析构函数，进而产生内存泄露问题的隐患。

而要解决这个问题，方法也比较简单——我们需要给Expression基类，添加一个**虚析构函数**：

```
class Expression
{
public:
    bool IsLeft;
    Expression() {
        IsLeft = false;
    }
    virtual ~Expression() = default;
};
```

当shared_ptr认为某个对象需要回收的时候，它就会首先调用基类的析构函数——而这是一个虚析构函数，已经被派生类的析构函数给覆盖掉了。这样一来，我们就能保证当对象析构时，调用的是正确的析构函数。

这里需要说明的是——我们额外添加了一个bool类型变量IsLeft，它的作用是，标记对象是否为左表达式。这个变量的作用是——如果我们需要从AST还原回正确的四则运算表达式的话，那么这个变量可以帮助我们，对于是否需要给表达式加上括号，做出正确的判断（因为减法和除法不允许交换律）。

上篇的内容就先写这些——改写语法分析器的部分留到下篇再讲。作为一名弱渣，贸然开坑，压力实在很大——如果有什么疏漏，还请各位大神不吝赐教。

下篇地址：

<https://zhuanlan.zhihu.com/p/23139200>

基于现代C++的四则运算语法分析器（下）

书接上文

接下来，就要进入语法分析器的实质部分了——这篇说实话其实有点无聊，毕竟该解释的东西原文都解释了，很多内容我只是复制粘贴而已：

到了这里，我们可以考虑一下如何通过语法组织我们的代码了。上面的语法并没有包含如何去除空格的语法，这个事情语法表达只会徒增烦恼，因此我们自己解决可能会更好一点。在语法分析的时候，我们都是一点一点读入字符串的，因此我们的函数的形式大概如下：

1. 读入字符串，返回结果或者错误信息
2. 如果没有错误的话，则将字符指针偏移到尚未读取的位置
3. 如果有错误的话，保持字符指针不变

好了，现在我们来看第一条语法。我们需要一个方法来检查输入是否由我们需要的字符串开头，当然这里仍然需要考虑空格的问题。我们可以写一个函数，输入字符指针和一个字符串。这个函数先过滤掉空格然后检查剩下的地方是不是由指定的字符串开始的。正确的话返回true并将输入的字符指针往后挪到尚未读取的地方：

```
/*
检查Stream的前缀是否Text
是返回true并将Stream偏移strlen(Text)个字符
否则返回false
此函数会过滤Stream开头的空格
*/
bool Is(char*& Stream, const char* Text)
{
    size_t len=strlen(Text);
    /*保存参数*/
    char* Read=Stream;
    /*过滤空格*/
    while(*Read==' ') Read++;
    if(strncmp(Read, Text, len)==0)
    {
        Stream=Read+len;
        return true;
    }
    else
    {
        return false;
    }
}
```

代码很短我就不解释了。

以上为原文引用——最后实际使用的代码如下所示（为了消除编译器警告，这里我们使用const char* 替代了原来的char*——对此有问题的同学，请自行复习 常量指针和指针常量 这两个类似而又不一样的概念）

```
bool Is(const char*& Stream, const char* Text)
{
    size_t len = strlen(Text);
    const char* Read = Stream;
    while (*Read == ' ') Read++;
    if (strncmp(Read, Text, len) == 0)
    {
        Stream = Read + len;
        return true;
    }
    else
    {
        return false;
    }
}
```

Is函数的作用是，判断Stream是不是以Text开头——如果是的话，就返回true，顺带把Stream偏移strlen(Text)个字符，反之则返回false。这个函数在判断的时候会自动滤掉Stream开头的空格。

有了Is函数之后，我们可以定义下一个函数GetNumber了——它的作用是，按照数字来处理原文返回的是Expression* 裸指针，这里我们用shared_ptr替换。特别需要说明的是，这里我们只实现AST，所以这里其实可以换成unique_ptr。评论来自R大[@RednaxelaFX](#)

[RednaxelaFX](#)

话说作者的意图确实是只打算做树形AST而不是做成DAG么？如果是树的话，每个节点只可能被一个parent节点所引用，感觉不需要shared_ptr而用unique_ptr就够了啊。

代码如下所示——因为Expression的定义方式完全改变了，所以这部分代码也需要重写：

```
shared_ptr<Expression> GetNumber(const char*& Stream)
{
    int Result = 0;
    bool GotNumber = false;
    const char* Read = Stream;
    while (*Read == ' ') Read++;
    while (true)
    {
        char c = *Read;
        if ('0' <= c && c <= '9')
        {
            Result = Result * 10 + (c - '0');
            GotNumber = true;
            Read++;
        }
        else
        {
            break;
        }
    }
    if (GotNumber)
    {
        Stream = Read;
        return make_shared<NumberExpression>(Result);
    }
    else
    {
        throw Exception(Stream, "此处需要表达式");
    }
}
```

这个函数的作用是，按照数字的方式，对Stream进行解析（同样改成了const char*，下同），如果解析成功，就将Stream指针的位置移动到解析完成的最后位置（代码中的Read），然后调用make_shared，构造一个shared_ptr并返回；而如果失败（即一个数字都没找到），则抛出一个异常。跟Is一样，这个函数会过滤掉开头的空格。

Exception类是我们自己定义的，如下所示：

```
struct Exception
{
    const char* Start;
    const char* Error;
    Exception(const char* aStart, const char* aError)
    {
        Start = aStart;
        Error = aError;
    }
};
```

这里需要特别强调的是，对shared_ptr的构造类型，应该注意这样一点：按照**NumberExpression**类型构造shared_ptr，然后按照**Expression**类型返回。为什么要这么做，上篇已经说过了。

必要的准备工作都已经完成了，接下来我们可以进入正题了——对于四则运算，前文已经说过，我们有这样的语法规则：

1. Operator="+" | "-" | "*" | "/"
2. Expression=<数字> | "(" Expression ")" | Expression Operator Expression

但是这里有一个问题——如果直接按照Expression=Expression "+" Expression来处理的话，就会陷入左递归。从Expression出发从左往右分析，结果遇到第一个又是Expression，这样递归导出自己的结果就是陷入无限循环出不来了……

下面继续贴原文：

我们考虑一下乘除先于加减背后的本质是什么。看一下一条比较长的表达式：

1*2*3+4*5*6+7*8*9

我们在计算的时候会把他们分成三个部分：1*2*3、4*5*6、7*8*9，分别计算出结果，然后相加。如果我们可以把仅仅由乘除组成的表达式的语法写出来，那么写出四则运算式子的语法也就有希望了。事实是可以的。于是我们要对之前的结果做一下调整。无论是数字或者是括号包含的表达式都不可能因为在旁边添加其他操作符而对优先级有所影响，因此我们抽象出一个类型叫Term：

Term=<数字> | "(" Expr ")"

然后我们就可以写一条只用乘除构成的表达式的语法了：

```
Factor=Term | Factor "*" Term | Factor "/" Term
```

最后，我们可以写出一条只用加减和Factor构成的表达式的语法：

```
Exp=Factor | Exp "+" Factor | Exp "-" Factor
```

到了这里表达式的语法就大功告成了。上面的三条语法中的Exp就是四则运算的语法了。

我们注意到Exp和Factor都是左递归的。在这里我介绍一种消除左递归的方法。我们考察一下语法Factor=Term | Factor "*" Term这一条。为了形象的表达出什么是Factor，我们反过来可以考察一下Factor究竟可以产生出什么样的东西来。

一个Factor可以产生出一个Term。然后，一个Factor可以变成Factor "*" Term。如果我们把Factor "*" Term中的Factor替换成已知的结果的话，那么我们可以得到一个结论：一个Factor可以产生出Term "*" Term。同理，我们又可以知道一个Factor可以产生出Term "*" Term "*" Term，为Factor可以产生出Term "*" Term。于是我们大概可以猜出解决左递归的方法：

假设存在如下表达式：

```
A=B1  
...  
A=Bn  
A=A C1  
...  
A=A Cn
```

我们可以将这个语法修改为如下形式：

```
A'=C1 | C2 | ... | Cn [A']  
A=(B1 | B2 | ... | Bn) [A']
```

我们可以看到现在的A没有发生变化，但是新的语法已经不存在左递归了。我们为了简化表达，可以引进一种新的语法：我们让X*代表X、X、X等等只由A组成的字符串或者空字符串，那么上面这个语法就可以被修改成A=(B1 | B2 | ... | Bn)(C1 | C2 | ... | Cn)*了。

于是，我们重新写一下四则运算式子的语法：

```
Term=<数字> | "(" Exp ")"  
Factor = Term ( ("*" | "/") Term) *  
Exp = Factor ( ("+" | "-") Factor) *
```

接下来，我们需要完成以下三个函数——这里先把它们的声明写在前面：

```
shared_ptr<Expression> GetTerm(const char*& Stream);  
shared_ptr<Expression> GetFactor(const char*& Stream);  
shared_ptr<Expression> GetExp(const char*& Stream);
```

它们的定义方式跟GetNumber差不多。

首先来看GetTerm的写法：

```
shared_ptr<Expression> GetTerm(const char*& Stream)  
{  
    try  
    {  
        return GetNumber(Stream);  
    }  
    catch (Exception& e)  
    {  
        const char* Read = Stream;  
        if (Is(Read, "("))  
        {  
            auto Result = GetExp(Read);  
            if (Is(Read, ")"))  
            {  
                Stream = Read;  
                return Result;  
            }  
            else  
            {  
                throw Exception(Stream, "此处需要右括号");  
            }  
        }  
    }  
}
```

```

    }
    else
    {
        throw e;
    }
}
}

```

它使用了一个try——catch模块来实现。首先，我们已经知道Term遵守这样的规则：

```
Term=<数字> | "(" Exp ")"
```

首先，它会按照数字来对Stream进行处理——具体来说就是在try模块中调用GetNumber。如果GetNumber正常找到了数字，那就皆大欢喜，直接返回；反之，如果没有找到数字，GetNumber会表示，“这个锅劳资不背”——抛出一个异常。

抓到异常之后，GetTerm会按照规则的右半部分进行处理——首先看看是否找到了左括号，如果没有，那么“这个锅劳资也不背”，再把抓到的异常e扔出去就行了。

如果说有的话，则直接进入规则的右半部分——调用GetExp来寻找Exp，结果保存在result中（这里使用了自动类型推导auto，省得写那一长串），然后检查是否有一个可以匹配的右括号中。同样，如果有，那就皆大欢喜，直接更新Stream指针，返回result。而如果没有，则重新构造一个异常，扔出去——注意这个异常跟之前GetNumber的甩锅可不一样，这里的异常意味着原来的表达式写错了，漏写了右括号。

下一步，让我们跟随第二条规则，进入到GetFactor中：

```

shared_ptr<Expression> GetFactor(const char*& Stream)
{
    const char* Read = Stream;
    shared_ptr<Expression> Result = GetTerm(Read);
    while (true)
    {
        BinaryOperator Operator;
        if (Is(Read, "*"))
        {
            Operator = BinaryOperator::Multiply;
        }
        else if (Is(Read, "/"))
        {
            Operator = BinaryOperator::Divide;
        }
        else
        {
            break;
        }
        Result = make_shared<BinaryExpression>(Operator, Result, GetTerm(Read));
    }
    Stream = Read;
    return Result;
}

```

规则如下：

```
Factor = Term ( ( "*" | "/" ) Term) *
```

首先我们来读取第一个Term，保存到变量Result中，然后进入一个循环——根据正则表达式规则，后面那个*符号表示((“*”|“/”)Term)可以重复任意次。

在循环中，我们首先定义一个BinaryOperator类型的变量Operator，然后判断下一个字符是否为乘法或者除法的运算符。如果是的话，我们就把Operator赋值为对应的运算，然后在结尾构造一个新的BinaryExpression。如果不是，则表示表达式已经结束了，直接break即可。

在构造的过程中，Operator作为运算符参数，之前已经得到的Result作为第二个参数（即左表达式First），而接下来运算符右边的Term，则直接用GetTerm得到，作为最后一个参数（右表达式Second）。这样，在每一轮循环中，上一次构造得到的Result，都会在下一次构造中，作为First参数传递给构造函数。有兴趣的同学，可以据此自己画一下由此构造产生的AST结构。这里我就不画了（懒……）

最后看一下GetExp：

```

shared_ptr<Expression> GetExp(const char*& Stream)
{
    const char* Read = Stream;
    shared_ptr<Expression> Result = GetFactor(Read);
    while (true)
    {
        BinaryOperator Operator;
        if (Is(Read, "+"))

```

```

{
    Operator = BinaryOperator::Plus;
}
else if (Is(Read, "-"))
{
    Operator = BinaryOperator::Minus;
}
else
{
    break;
}
Result = make_shared<BinaryExpression>(Operator, Result, GetFactor(Read));
}
Stream = Read;
return Result;
}

```

跟GetFactor一样，这里就不详细解释了。

这样一来，我们的语法分析器就构造完成了——对一个表达式调用GetExp，我们就可以得到表达式AST的根节点。

```

int main(){
    auto exp=GetExp("1+2+3");
    //Do what you want to do
    return 0;
}

```

接下来，根据需要，如果我们想得到前缀，或者后缀表达式，就非常容易了——我们只要对其进行先序或者后序遍历就可以了。这里就先不实现了，有兴趣的同学可以自行完成。

不过，这篇文章其实还有点内容没有结束——所以后面还会有一个《基于现代C++的四则运算语法分析器（续）》，介绍剩余的内容（主要是关于在C++中实现Visitor模式的部分）。

static_cast-const_char--(-Fuck_GTK+-)

今天 Mili (<https://www.zhihu.com/people/milizhang>) 同学继续在做GacUI (<http://www.gaclib.net/>) 的移植工作，结果因为傻逼 GTK+是C语言写的，Signal都是内部乱cast的，但是GacUI的OS Driver又都是接口，于是就遇到了一个被无数人解决了无数遍的问题：如何把类成员函数指针变成void(*)(void)从而进行回调？

前辈们的代码肯定是不能抄的，毕竟现在已经是C++17了，当然要用现代的方法来解决。于是我就展示了下面的这个解决方案以供参考。

整份代码的需求在最下面，已知Fuck接受回掉函数，已知Callback::Invoke就是那个回掉函数，因此上面的一个片段就是构造了一个CALLBACK宏，把Callback*和&Callback::Invoke变成了void(*)void)和void*。

原理很简单，就是不断的用模板元编程来日clang++，然后祈祷clang++不会被日死。不过中间还是遇到了一些小问题，譬如说CallbackWrapper偏特化的时候，前面的template声明里面用了using后的void(T::*)(TArgs...)，因为不想写两遍。VC++就可以把using的东西展开进去继续推导，clang++直接就罢工了。真是烂爆了。

```
#include <iostream>
#include <type_traits>

using namespace std;

//*****************************************************************************
CALLBACK
***** */

template<typename T>
using ObjectOf =
    typename std::remove_pointer<
        typename std::remove_reference<T>::type
    >::type;

template<typename T, T f>
struct CallbackWrapper
{
    static void Get()
    {
    }
};

template<typename T, typename ...TArgs, void(T::*f)(TArgs...)>
struct CallbackWrapper<void(T::*)(TArgs...), f>
{
    static void Invoke(TArgs ...args, void* argument)
    {
        reinterpret_cast<T*>(argument)->*f)(args...);
    }

    static void(*Get())()
    {
        return reinterpret_cast<void(*)()>(&Invoke);
    }
};

#define CALLBACK(OBJECT, NAME) \
CallbackWrapper<\n\
    decltype(&ObjectOf<decltype(OBJECT)>::NAME), \n\
    &ObjectOf<decltype(OBJECT)>::NAME \n\
    >::Get(), \n\
    static_cast<void*>(OBJECT) \n\
***** */

CLIENT
***** */

void Fuck(void(*shit)(), void* argument)
{
    auto bitch = reinterpret_cast<void(*)(int, double, const char*, void*)>(shit);
    bitch(1, 1.2, "Fuck GTK+", argument);
}

struct Callback
{
    int context;

    void Invoke(int a, double b, const char* c)
    {
        cout << context << "," << a << "," << b << "," << c << endl;
    }
}
```

```
}

};

int main()
{
    Callback callback{ 200 };
    Fuck(CALLBACK(&callback, Invoke));
    return 0;
}
```

C++需要不断地练习。

随手干点什么都能发现开源界可以娘的东西

背景，先上一段代码：

```
int main()
{
    vector<Bird> birds;
    birds.reserve(4);
    birds.emplace_back(1);
    birds.emplace_back(2);
    birds.emplace_back(3);

    try
    {
        birds.emplace(begin(birds) + 2, 4, true);
    }
    catch (Exception)
    {
    }

    for (auto& bird : birds)
    {
        cout << bird.id << " ";
    }
    cout << endl;
    return 0;
}
```

Bird类的构造函数是Bird(int id, bool requiresThrow=false)，第二个参数填true，就会在构造函数里面引爆异常。

现在我们来看这个程序，vector里面一共有三个对象，然后我们调用emplace把第四个对象插入到中间去，然后在构造函数里面引发异常，那么emplace函数将会失败。那这个时候vector应该只有原先的3个对象。那么emplace函数整个就是一个transaction，要么发生了，要么没发生，不会发生一半。

我就在Bird类的每一个函数里面打log。reserve(4)的意义就在这里，保证push_back和emplace都不会触发延长内部buffer的代码，去掉无谓的噪音。之后我发现Visual Studio 2015出现了一堆其他噪音，搜了一下发现有人曾经发bug说VC的STL的emplace写的不好，居然是先把4放在3后面，然后把他们交换过来。这显然浪费了一点CPU资源。看样子一年过去了没有修，不过好歹结果是没有问题的。

那就去尝试clang++了，clang++倒是一点噪音都没有，STL写得很紧凑，但是结尾给我输出了“1 2 3 3”，第一个3其实就是{4, true}的尸体，竟然留在里面了。因此想想VC++的那种奇葩先push_back再rotate的写法，多半就是为了防止这种事情发生的，错怪他了。

其实写这篇文章的时候，我还在想说，有没有可能是STL根本就没担保这种情况，只是VC++做了些多余的好事？但是评论有人贴出了文档（[std::vector::emplace - cppreference.com](#)），证明这个好事一点都不多余（逃

连基础库都可以写成这样，要让大家如何信任上面所有软件的质量？传说中开源代码看的人多所以没问题的事情，已经被OpenSSL打脸了。参考一下之前gcc号称紧跟C++11的步伐，出了一个没有实现的regexp，估计他们压根就没打算好好做编译器工具。

尾声：之前听说g++和clang++是共享同一份STL的，我试了一下，果然是真的。

轮子哥带你学C++——《CS212-面向对象程序设计C++》

上周我曾经在一个问题里说过 [如何评价计蒜客即将推出的“CS 212: 面向对象的程序设计 \(C++\)”?](#) ——一周后的今天，我们要搞一个超级大的大新闻。

今天，你们要的大新闻来了

——《CS212:面向对象程序设计C++》在经过长达四个月的开发和测试之后，终于在今天，亮相了。

The screenshot shows the homepage of Jiguoke. At the top, there's a green header bar with the logo '计蒜客' (Jiguoke) featuring a stylized garlic bulb icon. Below the header, a large blue banner displays the text '轮子哥 带你学' (Wheel Man Leads You to Learn) next to a circular progress icon showing '1%' completion. The main title 'C++¹⁴' is prominently displayed in large black letters. Below it, a subtitle reads '基于现代的 C++ 构建，教你以面向对象思想编写优雅而健壮的代码' (Built based on modern C++, teach you to write elegant and robust code with object-oriented thinking). The page also features three icons with subtitles: a gear icon for '面向真实工程开发' (Faced with real engineering development), a code bracket icon for '伴随式编程，以最高效率学习' (Accompanying programming, learning with the highest efficiency), and a wrench icon for '注重练习，提升“内功”' (Focus on practice, improve "internal power").

大家肯定会问：这为什么算是大新闻？

原因很简单——以前我曾经有意无意地说过，这门课是“某位知名大神”跟我们合作的。现在到了今天，我终于可以说了：

正如那个抽象的头图所展示的那样，这门课的作者是轮子哥。

教学团队



陈梓瀚
ID vczh
Microsoft Office 开发者
《C++ Primer》中文第五版审校

为什么选择这门课

蛤？你确定你真的要问这个问题吗——“轮子哥带你学C++”这件事情难道不就是最大的理由吗？

嘛，先不开玩笑——严肃地说，这门课程，拥有以下的几个显著的优点：

- 真正意义上地以《C++ Primer》为蓝本，基于现代 C++ 构建，教你学会如何书写优雅、高效而健壮的C++代码

许多类似的课程都会在参考书目里写上《C++ Primer》，但是实际上课程的内容，基本上只有C++最为“简单”的部分，几乎全都是C++98/03的传统内容，对于现代C++基本上没有提及。大家在“学会”了C++之后往往会觉得“其实C++也没那么难啊”，然而实际使用的时候写出的代码往往漏洞百出，到处都是bug。

```
1 int a[] = {1, 2, 3, 4, 5};
2 int b[] = {3, 4, 5, 6, 7};
3 int* it = nullptr;
4 for (int i = 0; i < sizeof(a) / sizeof(*a); i++)
5 {
6     for (int j = 0; j < sizeof(b) / sizeof(*b); j++)
7     {
8         if (a[i] == b[j])
9         {
10             it = &a[i];
11             break;
12         }
13     }
14 }
15 bool found = (it != nullptr);
```

典型C风格代码

究其原因，一方面是因为C++98/03还是太“C style”了，很多地方都没有填坑——所以你必须要充分地了解语言的方方面面才能确保不出问题；而另一方面则是因为，尽管这些课程在参考书目里写上了《C++ Primer》，但是在课程安排上却并没有像《C++ Primer》那样与时俱进，仍然从旧标准中的C feature开始介绍——这就导致了很多初学者遇到各种莫名其妙的问题，学习热情遭到打击甚至裹足不前。

合理的学习顺序，应该是先从现代C++的部分开始介绍，然后再来由易到难地介绍那些传统的C风格特性。这，也正是这门课程的特点：它真的是基于《C++ Primer》而构造，致力于介绍一个现代C++（C++11和C++14）的子集——可以替代绝大多数常用的C语言feature。对于经验不足的程序员来说，使用现代C++特性写出的代码，相比使用C-feature写出的代码，通常可以变得更加优雅、高效而健壮。

vector 的做法：

```
1 vector<int> a = {1, 2, 3, 4, 5};
2 vector<int> b = {3, 4, 5, 6, 7};
3 auto it = find_first_of(begin(a), end(a), begin(b), end(b));
4 bool found = (it != end(a));
```

在这个代码里，**a** 中第一个在**b** 里出现的数字是 3，所以：

- ***it** 会返回 3。
- 这个时候，如果执行 ***it = 100;** 就会把第一个值为 3 的元素重新赋值成 100。

相同功能的现代C++代码

- 形象讲解C++的特性，克服“从入门到放弃”

尽管这门课程不能完全取代《C++ Primer》，但是对于大部分学生来说，这门课程可以显著降低学习《C++ Primer》的难度，帮助大家彻底克服“从入门到放弃”的窘境。



- 伴随式编程，将学习效率提高到最高

大家注意到了吗？我之前说的是，轮子哥是这门课的“作者”而不是“讲师”——没错，我们的课程不是MOOC，没有视频。因为看视频太耗费时间了，而且学习效果极其可疑。我们的教学形式，除了基本的阅读和选择题之外，最为主要的就是伴随式编程课了：你可以想象，有一个虚拟的“轮子哥”在手把手地教你一句一句地写代码。

我们现在来写 `fib` 的内容。相信大家都明白什么是斐波那契数列，这个函数就是用来返回数列第*i*项的数字。*i* 从 0 开始。所以当 *i* 小于 2 的时候返回 1，否则返回前两项相加的结果。虽然这可以通过一个循环来解决，但是我仍然希望大家使用递归来写。

在 lambda 表达式里面，我们先假装它可以看成 `fib`，所以前两项的值可以通过调用 `fib` 来解决。在使用 `if` 语句的时候，不要省略大括号和 `else` 分支。

点击查看提示

```
about_lambda_code.cpp
1 #include <iostream>
2 #include <functional>
3
4 using namespace std;
5
6 int main()
7 {
8     auto fib = [] (int i) -> int {
9
10    };
11    for (int i = 0; i < 10; i++)
12    {
13        cout << fib(i) << endl;
14    }
15    return 0;
16 }
```

俗话说“好记性不如烂笔头”——计算机科学的理论属于知识，而编程则属于技能。既然是技能，那就需要用动手的方式来学习——在学习编程的所有方法中，没有什么比自己亲自敲一遍代码更快捷更高效了。计蒜客为你提供了一个随时随地都可以直接开始写代码的，方便快捷的平台——如果你在学习中遇到问题，可以直接去问答区提问，所有问题都会在24小时之内得到友善而实用的回答。

- 从工程实践出发，真正学会如何使用面向对象的编程思想和设计模式构建程序

“面向对象”不光是语言特性，同样也是一种极其重要的编程方法论——然而过去的大量课程中，都是指浅尝辄止地教同学如何使用相关语法，装模作样地定义几个类就完事了。这样一来，学生往往是经过一段时间的学习之后，还是不知道该怎么设计类。

而这门课程，将利用大量的示例代码，用程序实际执行的过程，教会大家如何正确地设计类。除此之外，这门课程还会对面向对象设计模式进行初步的介绍，并向大家演示如何应用设计模式。为了巩固这些内容，这门课程还提供了多个精心设计的，规模庞大的编程题和工程题，让大家在实际的项目中，亲身体会到为什么要这么设计类，为什么要应用设计模式。

计蒜客

课程 题库 技能 比赛

实验：使用 Visitor 模式实现四则运算表达式去除多余括号

本节内容计 50 分，提交后系统自动判分

点击下载项目文件

实验：使用 Visitor 模式实现四则运算表达式去除多余括号

- Visitor 模式
- 抽象语法树 (AST)
- 中序遍历

分数 50

这个题目对于各位同学来说会比较有挑战性，并且需要一些本课程以外的知识——所需要的知识在之前的阅读中已经介绍给大家了。仍然不懂的同学可以对前面的阅读课内容进行复习，或者自行学习《CS 261 数据结构》的知识。

这里我们需要做的，是读入一个四则运算表达式

```
1 #pragma once
2 #include <memory>
3 using std::shared_ptr;
4 enum class BinaryOperator
5 {
6     Plus,
7     Minus,
8     Multiply,
9     Divide,
10 };
11 class Expression;
12 class NumberExpression;
13 class BinaryExpression;
```

```
Test 16 passed.
Test 17 passed.
Test 18 passed.
Test 19 passed.
Test 20 passed.
Test 21 passed.
You have passed all tests!
→ ~/project
```

重做 提交

PS：我之前曾经写过的两篇文章：

[基于现代C++的四则运算语法分析器（上）](#)、[基于现代C++的四则运算语法分析器（下）](#)，讲了怎么用C++11的语言特性，实现一个四则运算语法分析器——这两篇文章，其实讲的就是给这个课程的工程题造轮子的过程。

课程的所有正文内容（包括阅读、选择和伴随编程课）都由轮子哥完成，我的任务是按照轮子哥给的思路，出每一章的编程题（包括造轮子、写标程、写题面、准备Testcase和单元测试等）……

- 通过对C++的学习和训练，提升自己的“内功”，成为一名真正的工程师

C++是一个语言联邦，它博大精深，收放自如，包含多个编程范式却又完全不会阻止你使用其中的任何一个部分。作为一个想保持自我发展的程序员，在工作的过程中保持不断的学习是相当重要的——事实上随着项目和雇主的变更，大部分程序员在其一生中都需要学习所有流行的语言，所以学习语言更多的只是哪个先哪个后的区别。

The screenshot shows a web-based programming course interface. At the top, there's a navigation bar with icons for user profile, search, course selection, and other features. Below the navigation is a header bar with sections for '前言' (Introduction), '本节内容计 1 分。完成后得满分' (This section counts 1 point. Complete to get full marks), and a progress bar.

当然，这些话都是针对初学者说的。作为一个想保持自我发展的程序员，在工作的过程中保持不断的学习是相当重要的——实际上随着项目和雇主的变更，大部分程序员在其一生中都需要学习所有流行的语言，所以学习语言更多的只是哪个先哪个后区别。

在此，我推荐大家把 C++ 的学习放在一个早期（不一定需要是第一个，但是最好是在学生时代）的时间点，通过接触多个领域的知识。使用多种方法来学习解决一个问题。在知道什么方法对什么问题有什么影响的时候，为这个问题挑选语言并且学习它（当然你也可以选择 C++），比起你被命令去学习一个语言并解决这个问题，要更加地事半功倍。

上一页

下一

在这门课程中，你所学习到的知识，训练到的技能，都是普适性的——哪怕你今后不从事C++开发相关工作，你在这门课程中的收获，仍然会伴随你的职业生涯，让你终生受用。

在刚刚接到轮子哥发来的大纲的时候，我实在是懵逼了一会儿——因为课程内容的安排已经完全超过了我的认知范围，一时间我竟意识不到这门课程究竟应该处在一个什么地位。

现在，回顾整个成品，我终于可以给这门课程一个最终的定性：

大家都知道，现实中的美国高校，并没有纯粹的编程课——在我看来，如果美国顶尖高校的课程安排里，有《ObjectOrientedProgramming》这样一门课的话，那么这门课，大概应该就是长这个样子吧……

这门课程适合哪些学生

总而言之，这是一门面向初学者的C++高级编程课程——它的定位跟《C++ Primer》类似：如果你已经会用其他语言编程的话，那么它可以教会你用C++，但如果你是初学者，或者编程基础不够强的话，那么这门课的学习过程中你会非常难受。

因此，我们推荐大家至少学过一种编程语言（如果是JAVA、C或者“旧式”的C++的话那就更好了），并且至少完整地学过一遍数据结构。大家可以直接受计蒜客学习 [《CS 112 C++ 程序设计》](#) 和 [《CS 261 数据结构》](#)。

课程定价

这门课程以按时间方式进行计费，有多种标准可以选择：

- 按周计费每周需要140元人民币
- 包月一个月190元
- 包季度（三个月）480元
- 终身可用共600元

根据我们的评估，在投入足够时间的情况下，扎实地学完整门课程需要大约三个月——另外，当你完成这门课程（获得足够的总分）之后，计费将会终止，课程内容将会对你永久开放。大家可以量力而行，自行选择对应套餐……

PS：这门课程还有一些内容（例如模板）没有讲到——我们后续还会继续推出一系列课程，继续补完这些内容，敬请大家期待！

圣诞节我也要去旅游了

但是我跟[@叛逆者](#)不一样，我是不会写游记的。[@空明流转](#)安排的行程真他妈紧凑，所以应该会停止使用知乎一个星期，你们不要想念我（逃

如何跟老代码友好相处

旅游了一天，睡觉前刚好有点时间，就上来写点东西。后面还有将近一个星期的路程。

在Office工作也已经超过了两年了。尽管Office是微软C++新标准最有力的推手之一，我在这里面学到的东西其实跟语言倒没什么关系，主要还是跟老代码(legacy code)相关的事情。[@Hush](#) 曾经安利过我一本《Working Efficiency with Legacy Code》，不过我还没看，你们有兴趣可以去读一读。

面对legacy code的情况有很多，不仅仅在工作中会遇到一些198x年写的代码跟2016年写的代码混在一起的情况，哪怕是开发自己的GacUI也会有。尽管GacUI公开立项是在C++11发布之前不久，但是实际上整份代码是在我读大学的时候造各种轮子的时候，慢慢组合起来的。现在在注释里面还能看到类似Vczh Library ++ 3.0的字样，那1.0是什么呢？

Borland是在我上大一的时候把Delphi卖掉的，我也是差不多在那个时候第一次感觉到了自己学到的东西好像突然就没有价值了，于是趁这个机会全方面转向C++。第一个任务自然就是要把我以前写Delphi的时候积累下来的轮子用C++重写一遍，那就是Vczh Library++ 1.0。++的意思就是这是C++写的，怀念Delphi。后来慢慢的一边修改一边重构一边删除各种东西，因此我自己所有的C++个人项目都是围绕着这个库来开发的。从这个角度来看，GacUI的一部分代码算起来也有十年之久了，这个年龄其实已经超过了大家工作的时候能遇到的大部分项目的年龄了。所以在这里介绍的经验，对大部分的人都应该是合适的。

Legacy code造成最大的问题是什么？其实就是最新的best practice和标准，与过去的开发经验的矛盾。这是面对legacy code开发的时候，遇到的主要矛盾。这个问题在Office尤为明显。GacUI嘛，也只有十几万行。只要我哪天中了彩票，我可以辞职在家里从头优化，花个一年还是能够把所有的东西都改成最好的。至于Office，哪怕你让全球的办公软件开发商停下来等你，好让你把所有的代码都翻新一遍改成最好的，也是一件不可能的事情。

Office客户端的一个版本的代码（不包括分支也不包括历史），拉下来所有的文件就有300多G。这里面有差不多20-30G其实是所有平台的编译器和SDK，还有一些全球语言的字符串和配置，还有一些图标和测试，剩下的占了大部分内容的都是代码。Office现在有很多人在做，30多年通过不断的收购以及打字，最终创造了这么多代码。平均每个人要负责的代码就有超过30M那么多（是GacUI的十几倍）。要全部翻新一遍，量子计算机应该也普及了。所以首先我们要明白的事情就是，**用最新的标准来要求程序员产出的代码是不可能的。哪怕是新的代码，只要这些东西跟古老的部分有一点关系，你做起来就会更加困难**。那落实到具体应该怎么做呢？实际上最合适的方法就是，当你在修改哪一个年代的代码的时候，就按照那个时候的要求，也就是整份代码的风格来写。

其次就是重构。前辈们的经验告诉我们，**重构最大的好处就是，通过现在多花一点时间，来节省未来无穷多的时间。那节省的时间到底是什么呢？其实就是新的需求跟就的架构的矛盾带来的开发效率的降低**。你为了现在的需要做了一个设计，很好的满足了需要，架构弄好了之后业务逻辑写出来特别的快。但是需求总是会变更的，总有一天你的架构就会成为落后的架构，在上面实现新的需求就会变得很困难，开发效率就降低了。在我们总是希望软件的生命无限延续的前提下，我们需要适当地做一些重构，来满足现在或者短期的快速开发业务逻辑的需要。

举个很简单的例子，如果我们在命令行里面打印一个菜单，按下数字键就可以做一些不同的事情，那当软件刚刚诞生，里面的东西还不多的时候，我们会直接的使用if(input == 1) { ... } else ... 的方法来写。后来你加进去的东西越来越多，你会发现if的那些代码就总是重复，所以有一天你改成了switch(input) { case 1: ... break; ... }。再后来，你发现由于业务逻辑的变化，这个菜单开始有增删改的要求，那你总不能每次拿掉一个东西就把所有的case重新修正一遍吧？这个时候就会开始使用函数指针数组，在main函数里面初始化之后，input就可以直接当下标。后来这个软件中遇到了更加复杂的需求，菜单开始不是线性的了，你可能也就即将开始感觉到一个UI库的重要性，慢慢的就引进了各种设计模式。软件的迭代从宏观上来看，道理也是一样的。

但是面对legacy code的重构有其独特的难点。一个持续进化软件的legacy code很legacy，通常也就意味着这个软件也不小，那你重构的时候需要处理的地方就非常多。你这项工作可能要持续半年，在这半年里面你又不能push，因为重构了一半的代码多半是跑不起来的。别人也不可能停下来等你重构，所以会在旧的架构的基础上不断地添加新东西，那么你需要处理的事情就会越来越多，直到爆炸。这也是很多古老的软件无法进行任何重构的重要原因之一。

但是这个问题并不是无法解决的。在Office里面有三种风格的重构。

第一种就是靠一个牛逼的人，就是可以迅速结束战斗，同时一个change上去感染了几个C++代码文件，上去还能跑。遇到这样的人只能每天路过办公室门口的时候进行膜拜。我就有幸目睹了一个principal的毛子干了这样的事情，因为之前一直都有合作，觉得真是太伟大了。

第二种就是让大家一起来。你开一个branch，把基础的东西弄好，然后让每一个人都在自己的工作之余加入到你的重构工作来，其实也就是把他们自己的组件的代码改成兼容你的新架构的。等到所有的组件都翻新过后，最后让大家一起再解决一遍pull request里面的conflict。

第三种就是，在旧的库的旁边写一个新的库，然后只要你库的对象不是在整个系统里到处传播的，那么你总是可以一个一个文件慢慢地把#include换掉，把代码改成兼容新库的形式。这样在后面修改这个文件的人自然也就被迫使用你的新库了。一直到所有对老库的#include都消失了，把老的删掉，重构工作就完成了。这样做的好处是你的新代码是不断地push给大家的，不会有merge的噩梦出现。

但是重构也不是万能的。因为毕竟一个架构如果没有影响到你的开发效率的话，为什么要重构他呢？做这个很容易就变成过度设计了。这在GacUI的身上就很明显。大家可能会发现，我在知乎上说C++新标准下面应该如何如何做的时候，GacUI出现的却总是那些过时的方法。其实一个很重要的原因就是，事情还没发展到我非翻新旧代码不可的时候。

举个例子，C++11说你们可以用shared_ptr、weak_ptr和unique_ptr来表达不同对象的生命周期，从而最大程度的避免对裸指针的使用（避免粗心用错）。但是GacUI仍然大量使用Ptr<T>和裸指针。其实原因有三个。

第一个就是，代码是旧的，在这套东西还没反映在标准里面的时候，Ptr<T>已经被广泛使用了。那现在我要不要再添加新功能的时候，使用shared_ptr，让系统里面同时出现两套智能指针呢？这当然是不行的，因为智能指针有自己管理引用计数的方法，不同的智能指针几乎是不可能混用的。

第二个就是，既然如此，为什么我不把Ptr<T>删掉直接全部换成shared_ptr？这就反映了上面粗体的内容。Ptr<T>换成shared_ptr真的就能提高开发效率嘛？如果GacUI是由很多个人写的，那这个很难说，毕竟不是所有人都知道Ptr<T>的各方面细节。但是我作为GacUI的“几乎”唯一的作者，我对代码的所有方面都有无限的了解，我用裸指针也很少犯错误，所以shared_ptr的好处仍然不值得我做一次全文替换。

第三个就是，其实Ptr<T>还有shared_ptr所没有的功能。GacUI的脚本引擎支持脚本创建新的类，这个新类可以继承自若干个C++里面的类，新类还能被反射出来使用。这就在实际上造成了，在内存布局上面，新类其实就是N个对象组成的，N-1个C++里面的类，还有一个脚本创建的模拟的类。那么这几个类的实例其实应该共享同一个引用计数的指针。不然父类实例用了一半，子类实例被早早析构了，这就尴尬了。

大家可以发现，用新的智能指针实现这个东西的方法就是，脚本创建的类去unique_ptr所有的父类，然后dynamic_cast其实就是去获得一个内部用shared_ptr装着子类的、父类接口的shared_ptr或unique_ptr，实现就是把虚函数redirect到父类那里去。当然这样你的父类就要求全部都写成接口（COM就是这么干的）。看起来很别扭是不是？因为C++这一套东西对这个场景其实不能很好的表达。

但是反过来，我可以很轻易地通过修改Ptr<T>、可以被脚本继承的类的基类DescriptableObject，加上通过SFINAE来让Ptr<T>面对普通类型的时候使用普通的实现，来轻松的做到这一点。要换成shared_ptr就要变成另一套做法了，重构难度还是挺大，超过了对以后开发效率的改善。COM的Aggregation是对相同的问题的另一套做法，我不小心重新发明了一次。

类似的内容还有很多，C++11出了for(auto x : xs){ ... }，然而我所有的地方还是用FOREACH(X, x, xs){ ... }宏。C++17即将就要有range了，而我却有老早就为了弥补range不存在，#include <algorithm>里面的东西组合起来又太难而山寨的Linq。C++11有了T&&之后，容器就变得非常好用。但是我自己需要的容器不多，加上这个T&&的支持也不难，所以我最终也没有把自己的类换成STL。给定新标准下一个支持X&&够早的X类，一个返回X类型的属性的setter最好的写法是SetX(X x)，不是X&&x，也不是const X& x，也不是两个都有。但是其实很多地方我也没换。因为这些东西的替换实际上都无法带来什么显著的好处，所以干脆就留着了。

剩下的还有很多琐碎的地方我也就不一一提到了，10年前的一些次要组件现在看起来可能会发现代码里有各种问题，但是反正已经写好了，改得更好用起来也不会更方便，干脆就放着不管了。

GacUI就几乎只有我一个人在写，添加新feature赶紧做完，好腾出手来造新的轮子，才是第一目标。代码是否符合最新的C++规范，那是其次。这放在很多商业软件上也是成立的。我的理由是由于我想造新轮子所以要赶紧把GacUI做完，商业软件为了生存下去给自己员工涨工资要迅速发布和迭代产品，这两个理由其实是等价的。

回到Office里面，其实也会经常遇到类似的问题。三个不同的古老的组件，使用了收购回来的时候内部就已经有的三个字符串类。有一天我要把他们整合到一起，怎么办？其实最经济的方法就是，把他们都严严实实地封装进自己的接口里面，别人不要去碰他们，只有我来碰。那我内部最多也就是多写几个恶心的字符串转换的代码而已。隔绝的好处就是，落后的组件的实现，通过我改头换面之后，对别人的伤害降到了最低。因此我也没有必要去重构他们了。这样就使用最短的时间，在保证质量的前提下，写出了不会降低别人开发效率的代码。

重构是要看成本的。当然反过来，哪怕是一个重构很难，规模很大，但是他创造的效益更大的话，就值得你去说服大老板让你去做。

在讲完了重构之后，剩下的一个大问题就是，那添加新的代码怎么办？其实这也是要按照相同的标准去做的。你添加新的代码，是否会因为旧的架构的影响，让使用你的新代码进行二次开发的人的开发效率被降低？如果你认为答案是“会”，这可能就是一个重构的信号，你要先重构，然后再加新代码。否则，你就按照跟原有的部分兼容的方法去写就好了。当然了，如果情况允许的话，你也可以通过上面讲到的第三种重构的方法，先从你的新代码开始，推广新的架构。等整个系统的每一个角落都被你翻新了之后，旧的架构就删掉了，你就在完成了一次重构的同时，要加的新feature早就上线了，不会影响到release的日期。

总的来说，为什么会有这些准则？其实根本的原因是，修改代码会造成regression，如果你测试的覆盖不够，重构也会引发大量的问题（这就是为什么重构跟测试是相辅相成的，少了一个都不行）。老的代码没有重构的必要就不要重构，你的整体工作量也就大大降低了，同时保证了软件可以被release。

一个刚刚加入工作的程序员，或者是一个学生，可能在遇到类似的问题的时候都比较激进，然后就会被事实教做人。其实这都是因为年轻人眼界不够高，没办法在全局观上看到很多事情背后的cost。所以如果你恰好加入了一个古老的软件的项目组，不要对旧的东西产生抵触，就是一个良好的开始。

圣诞期间的公路旅行

2016圣诞节终于去了人生的第一次公路旅行。主线任务主要是给[@空明流转](#)的房东买二锅头和五粮液，支线任务才是访问俄勒冈的各种灯塔、灯塔、灯塔、灯塔和在1号公路北段的山路中各种晕、各种晕、种晕、晕……

你们的轮子哥[@vczh](#)也因此消失了数日，所以没给知乎广大乡亲父老好好带逛。

对于公路旅行，一开始我以为是这样的：



跟基友开着车，唱着歌，吃着火锅（不对），在开阔的高速公路上风驰电掣，享受一把当王子的赶脚~！



偶尔也要停下来，在风和日丽的山谷间吃点烤物，思考一下人生。



不用开车的我，就坐在后座安安静静装个逼，欣赏下风景。我真的做了这个姿势，并尝试保持，然并卵。[@空明流转](#)菊苣随便拐个弯就把我给磕了，西部1号公路北段翻山越岭的部分真的不是盖的，不是盖的，不是盖的！九曲十三弯已经不足以形容它的恐怖。上山下山折腾了一个下午，我差点都吐了。看来前庭过分发达的我，还是不适合当王子吗？残念，岂可修！我要回到回忆中去，哦不，是西雅图的家里去。不要再特么给我提“山路”旅行了，不要再特么提了！！！我不要出来刷经验值了，我回家玩《巫师3》游山玩水打昆特牌算了！！！以上真的是我当时在1号公路北段上翻山越岭时内心的真实写照。

我也深知现实跟理想会有点差距，至少我也明白：

俄勒冈加油站的老美不可能是这样的……



但是俄勒冈油站的小哥的确很热情，主动帮忙加油并说“This is Oregon”让西雅图乡村出来的村姑受宠若惊。

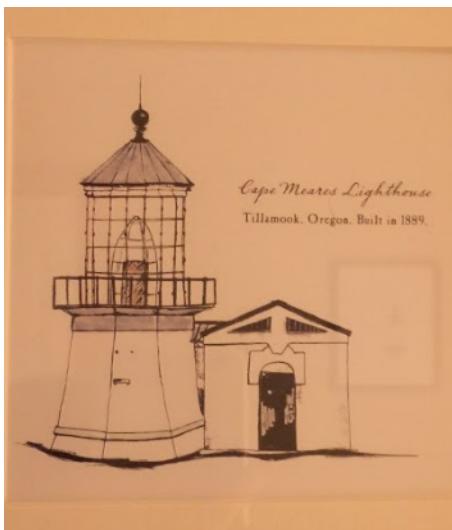


我也理解我们三个人不可能像理想中那么拉轰，毕竟少了一个人嘛……（不对！）

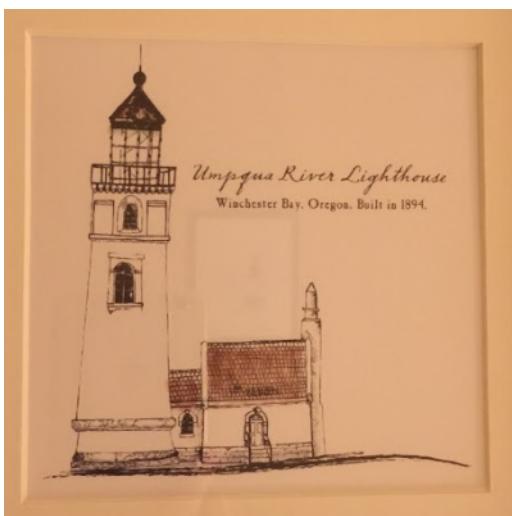
但是，我万万没想到，[@空明流转](#)居然给我们安排了那么多灯塔！！！虽然是灯塔国，也不用去那么多个灯塔的景点吧！！！在101公路沿线，途径俄勒冈州的时候，我们穿过了无数我觉得大同小异的灯塔……



就连住宿的旅馆也挂了灯塔的照片！！！！！！



Cape Meares Lighthouse
Tillamook, Oregon. Built in 1889.



Umpqua River Lighthouse
Winchester Bay, Oregon. Built in 1894.

灯塔太多了，我实在遭不住（路途上某个state park的厕所实在太暗黑了）了，坐在车上，我的心情是这样的…



当空明菊苣告诉我接下来还有一堆灯塔的时候，我.....



萨菲罗斯：圣诞节快乐！快递麻烦你签收一下！

收到快递，心情好像好一点点了（san值好像进一步下降了）。虽然灯塔有点多，但还是去了一些不错的地方，拍了些照片。





曾经炸爆阿克蒙德的世界之树→→





杰日天经常发现水鬼和装备的浅滩→_→





当然沿海线旅行是少不了看海的（后面是灯塔我会告诉你们么！）

全程都是[@空明流转](#)菊苣在开车，不容易啊。他还帮我们拍了很多萌萌的照片，带领我们度过了san值跌宕起伏的6天，真是感人啊~作为回礼，他也从我身上学到了一句口头禅“慌得一逼”，获得了成为谐星的潜质，也算是可喜可贺，不枉此行。

现在想来，旅行就是在于体验啊，体验有好有坏，在脑后插管实现之前，我想我还是会兴高采烈地去旅行的。一开始看大家吐槽FF15，认为亡国王子跟基友愉快地进行轻松的公路旅行是很不搭调的。公路旅行哪里轻松了！旅行要早上7点爬起来赶路、在1号公路北段被山路荡得神志不清、早上冷得一逼、手机没信号、荒郊野岭没厕所、这么多怪（灯塔）、女主角还不入队（只有AI）……

圣诞期间的公路旅行

嗯，带轮逛非常辛苦，尤其载着轮子哥的车更辛苦。

六天开了两千迈，最后一天开了近800miles，回来一番狂睡就看见陈萌萌吐槽了~啊我这里要反（狡）驳（辩）一下！[@陈萌萌 @vczh](#)

我就是想说，灯塔哪里有不好了嘛！虽然现在是冬季，经常遇到阴天是没错。。。

但是阴天的灯塔也可以挺好看啊！



$\tilde{w}_u,$
 \tilde{y}_e°

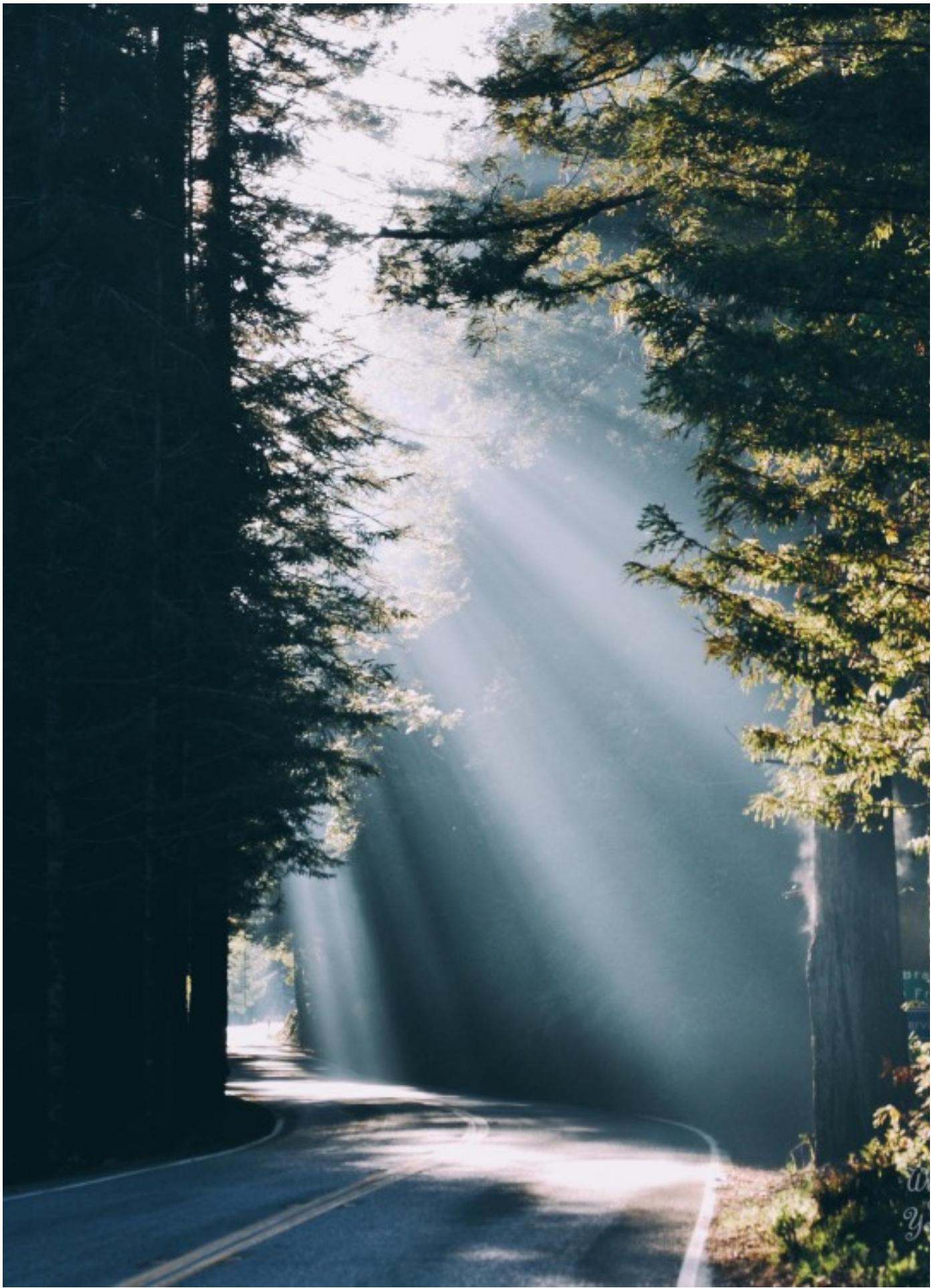


而且灯塔就建在海岸边上，不看灯塔也有海可以看：





而且加州一号除了海岸外，翻山公路也不错啊，虽然颠的七荤八素，但是这景色也并非浪得虚名啊~



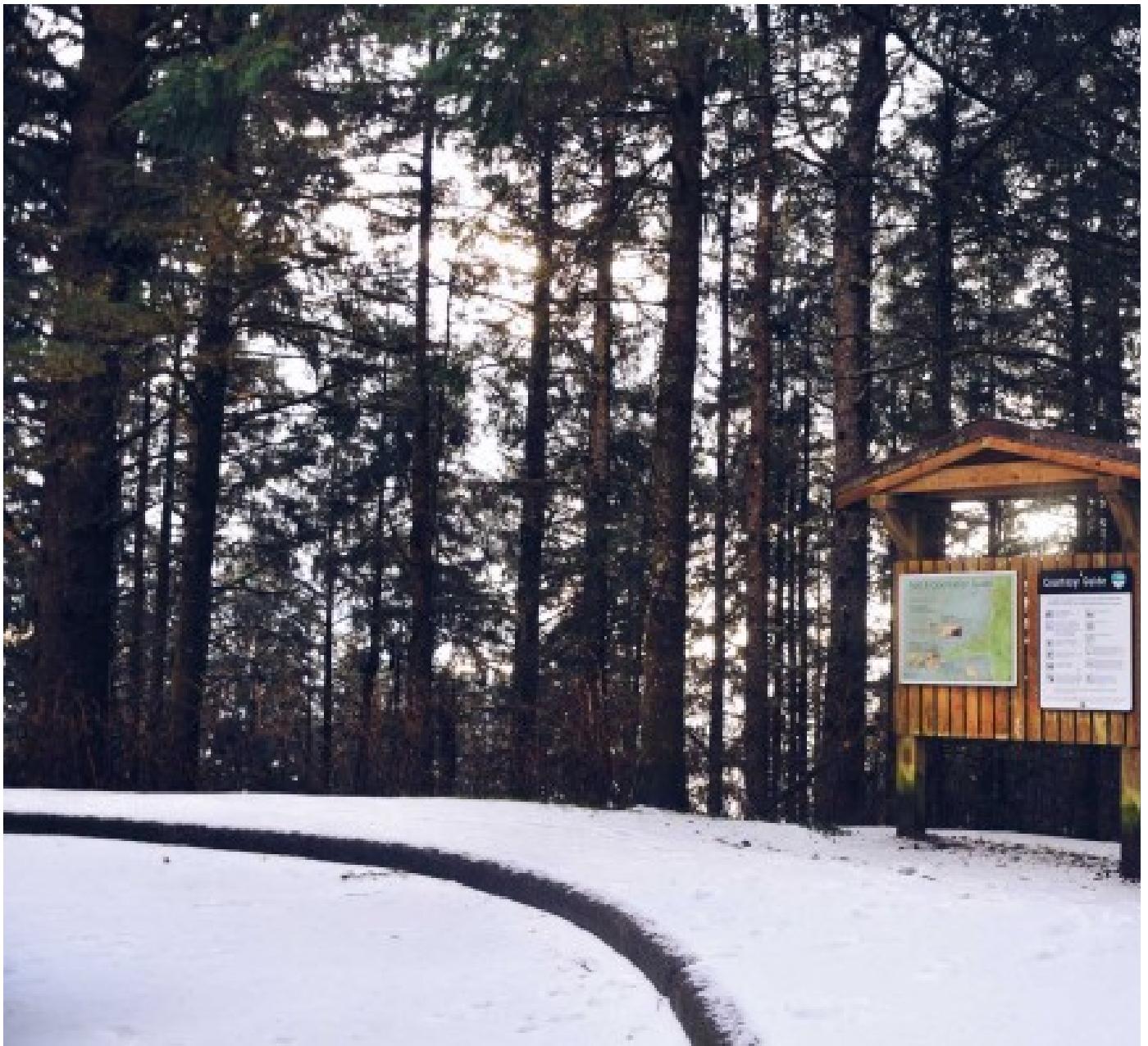
俄勒冈一段海岸岩石也不错啊。



小镇住宿也不错啊，虽然比不上贵谷希尔顿旗下的大酒店。。。



而且还见识了一场大冰雹，陈萌萌还以为下雪了：



但是转身的时候就能看见清楚的一道的彩虹啊还有什么不满足的！



P.S. 原来灯塔真的是能发出光束的啊，好屌。。。



哔，你们就是不热爱灯塔这种大自然的鬼斧神工啊！

再添几张照片：





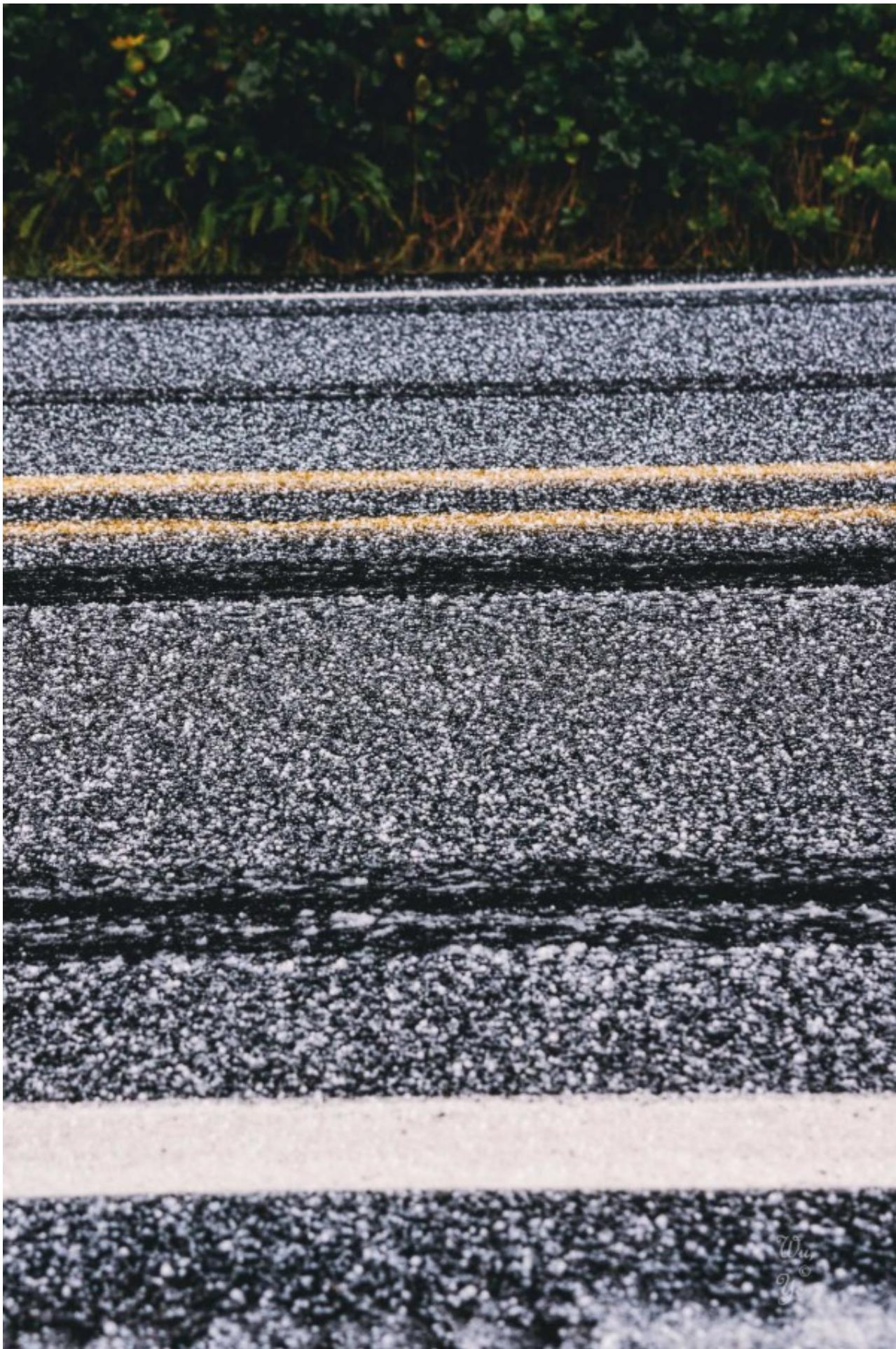
\tilde{w}_u ,
 \tilde{y}_e^c



$\mathcal{W}_u,$
 \mathcal{Y}_e°







\mathcal{W}_u
 \mathcal{Y}^o





GacUI: 初步完成Workflow脚本转C++的工作

GacUI一直有一个问题，就是使用XML开发界面的时候，生成的release模式的exe体积多达16M。主要的原因是XML需要反射。但是这个问题的解决，从今天开始已经看到曙光了！

Workflow作为一个脚本语言，他唯一的功能就是跟C++对象通信。所以Workflow对类型的支持也是直接照搬C++。这个脚本语言不仅可以调用C++的类，也可以被C++的类调用，你甚至可以在脚本里面创建一个类，多重继承自C++的接口和其他类，最后实例化成一个C++的接口的指针，丢给C++使用。Workflow与C++的交互是完全的。

之所以Workflow需要这么强大的与C++交互的功能，是因为GacGen.exe会把你的XML文件完全编译成Workflow脚本。这些脚本的内容其实就是几个继承自GuiWindow或者GuiCustomControl的类，然后类的构造函数会去创建你的子控件，设置好你的数据绑定，最终创建一个GuiWindow*的对象给你用。所以GacUI用到了C++的什么功能，Workflow就必须能够使用C++的什么功能。

为了做到这一点，我做了一个C++的反射。这个反射唯一的作用就是为了让Workflow存在。反射的内容有很多，你可以注册你自己的类、成员函数、属性、事件和接口，每一个类型最终变成一个ITypeDescriptor的实例。因此你就可以使用这个对象来调用函数，甚至是创建一个接口的实现，还有动态生成一个从C++的类继承下来的新类型等等。GacUI的代码本来就很多，所以反射需要的代码就更多，实际上release模式下那个16M的exe的其中15M就是在做这些事情的。

但是我相信绝大多数GacUI的程序都不需要动态加载UI的能力，所以实际上这个反射就完全没有用。但是没有反射Workflow又跑不了。所以我就想到了一个最简单的解决方法：只要能把Workflow编译成C++，那你直接把这些代码文件链接进去就可以了！

于是到了今天为止，我就完成了Workflow到C++翻译的基本工作。有单元测试就是好啊，只要我把所有单元测试里面的Workflow代码都翻译成C++，然后也能运行通过，最后检查一下code coverage没问题，那我对我的转换程序就有足够的信心了。大家可以在[vczh-libraries/Workflow](#)看到我所有的测试用例，以及在[vczh-libraries/Workflow](#)看到所有生成的C++代码。

当然这只是第一阶段的工作，为了方便C++代码的生成，因此就产生了很多多余的字符，把代码搞得很难看。不过反正生成出来的代码是不应该被阅读的，所以如何美化他就变成一个平行的、不紧急的、低优先级的工作了。为什么要去管那些不需要阅读的代码是不是好看呢？哈哈哈哈哈哈。

当然仅仅是这样的翻译还是不够的。接下来我还会继续开发GacGen.exe，让它生成的代码可以自动merge用户修改的代码（主要指的是用户插入事件处理函数的能力），最终让XML开发的UI摆脱15M的反射，轻松愉快地跑起来。这样用户就有两个选择，不需要动态特性的就全部生成C++链接进程序里，需要动态特性的就留下Workflow的byte code在资源文件里面，不生成C++代码。

说实话，最近这几个星期再做这个事情的时候，我感觉十分的无聊。这种完全没有挑战性的苦力业务代码写起来就是很不爽的，但是又不得不写。我觉得这就可以回答之前在知乎上提到的一个问题。什么是纯业务逻辑的代码？这（[vczh-libraries/Workflow](#)）就是纯业务逻辑的代码。这种代码只要你心情好，手的乳酸不要积累太多，一天要写多少行都可以，完全不需要经过大脑思考。做这个东西也是我最近不怎么刷知乎的主要原因，白天工作回来，晚上写这个都累死了，根本不想动。

GacUI的XML转纯C++做出来啦！exe体积直降90%！

其实这是标题党，因为还没有集成到GacGen.exe里面，还是有一点工作要做（逃。由于测试代码会经常改，所以有兴趣的人应该打开[Github上这个版本](#)的代码来看。

之前花了差不多一个月的时间，把Workflow脚本转C++的几乎所有工作都做了。不过还剩下一个，就是friend class的传播问题。由于Workflow支持匿名函数和匿名类的各种概念远超C++能够直接表达的概念，所以我把所有这些闭包的东西都真的创建出了一个类。因此类A friend了类B，却没有friend到B里面的一些闭包的类，因此目前只能把所有东西都public。等这个问题解决了之后，就把能protected的东西都protected了。

GacUI还有少部分的魔法代码，主要是跟把ObservableCollection绑定到列表控件相关的东西，暂时还无法直接生成C++，只能靠反射。这个问题时一定要解决的。还有一些小问题，都在这个长长的[ToDo列表](#)里面。等Working On和Comming Soon都做完了之后，GacUI就可以发布下一个Release了，大家就可以生成只有10%体积大小的exe了！

现在的Demo使用的XML资源是[这个文件](#)，这个文件含有一个普通的GUI，Host运行了之后会在TestCppCodegen的[这个文件夹](#)里面生成一堆代码，还有在Main.cpp旁边生成一个UI.bin（没有check in，自己运行的时候先跑一下Host）。TestCppCodegen的Main函数代码就很简单了，先把UI.bin加载进来，然后创建demo::MainWindow，整个程序就搞定了。直接生成C++，运行的时候跳过反射和虚拟机，速度就是快啊，爽的一比。

通常来讲，写一个的时候，如果没有加refCodeBehind="false"，那么就会给这个UI类单独生成一对C++文件。这主要是用来让你人肉填写事件处理程序的。譬如说大家在[生成的MainWindow.cpp](#)里面就可以看到，第一个函数是USERIMPL(...)开头的，这就是在提醒编译器，下面这个函数的代码是人写的。如果你改了这个函数的内容，那么当你XML更新了之后，我重新生成了C++代码，会把你的函数内容合并进去。当然了，其他所有地方的修改会被我直接忽略。直接填写事件处理程序的设计其实是很落后的，但是对于某些情况特别方便（特别是我还没支持XML编写动画的功能之前），所以我还是留着。其他的所有逻辑内容都应该通过MVVM模式：

- 写一个interface
- 在<Instance>里面写<ref.Parameter>声明这个interface类型的参数
- 在Main函数里面实现这个interface，传进窗口的构造函数里

这样你的逻辑代码对UI的部分就不会有任何依赖，就可以脱离GacUI运行单元测试了。

在接下来的几个月的时间里，我会把ToDo的东西都做完，然后把[Release repo](#)的全部例子都更新了，最后就来完成[www.gaclib.net](#)的首页，还有你们催了好几年的GacStudio（逃

GacUI应用程序终于实现体积骤降的目标！

之前那么多工作没白做啊！在之前完成的一系列的工作之后，现在普通地使用Xml来开发GacUI程序，在运行时已经完全摆脱对反射的需要了！上图！

1	Cpp.exe	2017/1/24 2:29	应用程序 1,611 KB
2	CppXml.exe	2017/1/24 2:29	应用程序 1,644 KB
3	MVVM.exe	2017/1/24 2:29	应用程序 1,822 KB
4	WorkflowScript.exe	2017/1/24 2:33	应用程序 12,792 KB
5	Xml.exe	2017/1/24 2:32	应用程序 11,529 KB

这里面是 [HelloWorlds 文件夹](#) 下面的几个Demo，分别用来介绍GacUI的几种用法：

- **Cpp**：完全使用C++来编写GacUI程序。
- **CppXml**：使用Xml来编写GacUI程序，GacGen.exe会帮你把Xml资源文件生成为一系列的C++代码，还有一个二进制的资源文件（所有的文字图片什么的就都打包进去了）。
- **MVVM**：最简单的MVVM模式的GacUI程序。
- **WorkflowScript**：完全使用脚本来编写GacUI程序，只不过因为虚拟机不是exe，所以还是要用C++代码去启动它。
- **Xml**：使用Xml来编写GacUI程序，GacGen.exe不生成C++代码，只生成一个二进制的资源文件。这个资源文件跟CppXml的区别就是会把本来应该是C++代码的东西，都变成了字节码放在里面。程序启动的时候会跑一个虚拟机来运行。这也是之前的运作方法。

显而易见地，前面三个不需要GacUI的反射，后面两个需要。所以可以很清楚地看出，其实之前每个Demo的体积都那么大，里面其实有大约85%的代码都是跟反射相关的。

当然通过这几个例子的测试，我还是发现了更多的bug，所以接下来还是要修改一下。不幸的是，目前在把ObservableCollection绑定到列表控件的时候，还必须使用反射（主要是获取元素的属性显示到不同的列里面），所以接下来的那个Release会有那么两三个Demo体积还是比较大。不过这个问题最终都是会得到解决的。

我的目标就是，完全脱离反射，程序性能暴涨！

GacUI_已经让几乎所有_Demo_“完全去掉反射”

现在就剩下两个跟列表绑定相关的Demo还需要进一步处理了。今天刚刚把内存大户 [ControlTemplate/BlackSkin](#) 也给转换了，这是一个使用XML来实现Visual Studio 2013的黑色皮肤的例子，也是编译的时候最慢的。因为每一个皮肤最终都是一个类，所以做语法检查的时候需要进行大量的工作。在去年的时候，所有的Demo都需要反射才能运行，当然也包括这个Demo，因此初始化的时间也可以被肉眼观测。

但是现在情况已经完全不同了！XML转C++的功能终于覆盖到了这个倒数第三个Demo了，因此初始化就是一大段C++代码跑完，窗口就出来了。而且拖动窗口的边缘让程序（主要是其他的TabPage）进行“实时”排版的时候，就算是Debug模式也根本看不见任何延迟。Release模式下甚至只要以双击BlackSkin.exe，一瞬间就看到了这个窗口。

现在还有一个小问题，就是Release模式的exe高达3.5M，当然这其中一个原因是XML是在写的有点复杂，内容多程序自然就大了。不过3.5M比起之前地34M已经处于可以忽略的情况了，等以后再继续优化。BlackSkin.bin这个编译好的二进制资源文件，因为之前需要包含所有XML编译出来的字节码，竟然有18M那么大，所以加起来一共有34M。现在不需要字节码了，只剩下848KB，其中的843KB是目录下面的png文件，全部都打包进去了，部署超级方便。

当然BlackSkin.bin也可以不成为一个文件的，你完全可以把它当成exe的资源文件直接链接在exe里面，运行的时候用API读出来，转成 [vl::stream::MemoryWrapperStream](#)，就可以被加载了。跟加载文件是一样的。使用DecoderStream + LzwDecoder，你甚至可以直接读取GacGen.exe根据你的配置创建的压缩后的BlackSkin.bin。

时隔两年，GacUI的丝般顺滑feature终于也覆盖这种超复杂的XML了，而且占用的内存比WPF小一大截，响应也比WPF要更快。果然纯粹的native代码就是迅猛啊。觉得自己真是太牛逼了（逃

当然这个UI乍一看其实也不是很复杂，但是因为我的排版大部分是面向Table的（就像css的grid和flex的组合），所以但凡任何东西都使用Table，这个截图的几个TabPage加起来，应该有几百个Table了。这么多Table根据属性互相制约，每次拖动窗口的时候都要不断地计算，收敛到一个结果才算排版玩，然而哪怕是被VC++调试的Debug模式下，也是一点感觉都没有，不断的拖动窗口边框，排版的计算比屏幕的刷新率还跑得快。Release就更不用说了。

一个好的架构和算法带来的好处，就是可以让使用者胡来，不需要有心理负担。就像在写C#程序一样，Linq乱写一气，程序又快又稳定。

值得注意的是，上面的那个日历是根据Windows的界面语言跑的。如果你在英文模式下打开，那就是英文的。就算你装了绿绿语，他也会变成绿绿语。富文本框也是支持绿绿语这种从右向左中间插着从左向右，几十个wchar_t粘成一个字符的这种语言的。

重构是永远都无法避免的

GacUI的大部分功能已经做完了，现在正在完善工具链。在把GacGen.exe做得更好的过程中，遇到的一个问题就是，报告错误的时候如何指出错误的地方在XML文件里面的位置。其实听起来好像也没什么，但是这里面有两个事情把报告位置变得困难了。

第一个是GacUI的XML资源支持类似CSS那样的精神的东西，那么当你的样式匹配到了一个错误的节点上，从而错误地操作了对象的属性的时候，我不仅要告诉你展开样式的地点（界面文件里），我还得指出到底是那个地方弄错了（样式文件里）。因此每次展开节点的时候，我得把构造节点的各种信息的来源位置铺满在整棵树里面。这个问题已经解决了，就是苦力已点，到处都要处理位置信息。

第二个是GacUI的XML资源允许你写 [Workflow脚本](#)。这个脚本的种类有很多，譬如说就是完整的脚本文件声明了一些库函数，譬如说属性的data binding的表达式，譬如说控件的事件的处理函数，甚至是分散在XML里面的各种名字啊。所有的这些信息最终会合成为一个完整的脚本文件拿去编译。错误的XML当然会生成错误的脚本，但是有些东西是没有办法在生成前就检查的，所以势必生成后的脚本也可能会产生编译错误。那这个时候我如何从脚本的编译错误的位置信息（在这里是AST的指针）给你还原成XML文件里面的位置信息呢？

其实实现起来也不难，就是你在生成的脚本的时候，每一个AST上面都挂一些东西就好了。但是实际操作起来特别苦力，这次苦力已经苦力到让我不想写了，但是东西又不能不做，所以只能从 [Vipp项目里面提供的Parser生成器](#)（概念上等同于ANTLR）开始改，让它帮我生成Parser的时候，顺便按需生成大量的苦力代码，到时候直接用就好了。

因此，为了报告错误信息，我就得去改编译器的前端生成器，就不得不继续做一个中等规模的重构了。**添加新功能觉得很痛苦，就是一个需要重构的信号。而如何才能安心重构呢？这就需要你有单元测试**。在一个项目里面，这些东西一环扣一环，少了哪一个，你都会在需求变更的时候觉得很蛋疼。

Workflow脚本的定位其实是C++的一个语法糖。我还是建议大多数东西用C++来做，最后通过Workflow脚本做胶水，通过MVVM完美粘进GUI里面。这样写GUI的时候可以直面GUI的概念（包括状态机和数据绑定等），实现逻辑的时候可以依赖C++的所有优点，东西做起来就更容易了。

在这里预告一下，因为这次要重构前端，顺便改Workflow的编译器，我决定把coroutine的工作往前提。在 [TODO.md](#) 的下半部分有一个我写了差不多的“草案”，讲的就是如何在Workflow里面添加coroutine的语法，于是就可以完成C#的yield return和async await（跟kotlin一样可以让你自己写库来实现），但是在我这是做成stackless的。除此之外，为了更加方便地实现UI的逻辑，这次coroutine也可以把一个状态机描述成一个接口的实现，切换状态的时候，个个边的输入其实就是对接口的成员函数的调用。这么做之后，MVVM威力将会猛增，配合await甚至可以绑定到一个服务器端的工作流（就像当年WPF、WCF和WF一起用的时候那样）。最后coroutine还会提供对动画的支持。

当然这么说好像听起来很复杂，但是其实这是一个非常简单的东西，给一个语言添加好用的coroutine，不需要多少知识就可以做到。我已经跟 [@RednaxelaFX](#) 说好了，等coroutine写完了，我就写一篇教程发到他的专栏里面去。标题我也想好了，就叫《考不上三本也能读懂系列——如何实现coroutine》，文章里面将尽量只依赖于编程常识和不超过高中的数学知识（逃

ParserGen生成预定义好的各种visitor

写编译器的时候经常会需要对一大堆有继承关系的抽象语法树节点类型进行运算。如果是函数式编程语言的话，都有pattern matching的支持。譬如说做一个四（简化成二）则运算的AST和求值，大概是这么写的：

```
data Expr = Value Double | Add Expr Expr | Mul Expr Expr

Eval (Value x) = x
Eval (Add x y) = (Eval x) + (Eval y)
Eval (Mul x y) = (Eval x) * (Eval y)
```

然后你 `Eval(Mul(Add 1 2)(Add 3 4))` 就能得到21。

但是这种鸡把代码用C++来写就会变得非常麻烦。当然了，根本原因是因为很多函数式语言都擅长处理递归数据结构，专门为这种语法优化了很多，而C++不仅没有，也并没有这么做的打算。那么到底要怎么写呢？大概会变成这样：

```
class Expr
{
public:
    virtual ~Expr() = default;

    virtual double Eval() = 0;
};

class ValueExpr : public Expr
{
public:
    double value;

    double Eval() override
    {
        return value;
    }
};

class AddExpr : public Expr
{
public:
    shared_ptr<Expr> x;
    shared_ptr<Expr> y;

    double Eval() override
    {
        return x->Eval() + y->Eval();
    }
};

class MulExpr : public Expr
{
public:
    shared_ptr<Expr> x;
    shared_ptr<Expr> y;

    double Eval() override
    {
        return x->Eval() * y->Eval();
    }
};
```

使用Visitor模式的话，一般会做成这样：

```
class Expr;
class ValueExpr;
class AddExpr;
class MulExpr;

class IVisitor
{
public:
    virtual ~IVisitor() = default;

    virtual void Visit(ValueExpr* node) = 0;
    virtual void Visit(AddExpr* node) = 0;
    virtual void Visit(MulExpr* node) = 0;
};

class Expr
```

```

public:
    virtual ~Expr() = default;

    virtual void Accept(IVisitor* visitor) = 0;
};

class ValueExpr : public Expr
{
public:
    double value;

    void Accept(IVisitor* visitor) override
    {
        visitor->Visit(this);
    }
};

class AddExpr : public Expr
{
public:
    shared_ptr<Expr> x;
    shared_ptr<Expr> y;

    void Accept(IVisitor* visitor) override
    {
        visitor->Visit(this);
    }
};

class MulExpr : public Expr
{
public:
    shared_ptr<Expr> x;
    shared_ptr<Expr> y;

    void Accept(IVisitor* visitor) override
    {
        visitor->Visit(this);
    }
};

class EvalVisitor : public IVisitor
{
public:
    double result;

    double Call(Expr* node)
    {
        node->Accept(this);
        return result;
    }

    void Visit(ValueExpr* node) override
    {
        result = node->value;
    }

    void Visit(AddExpr* node) override
    {
        result = Call(node->x.get()) + Call(node->y.get());
    }

    void Visit(MulExpr* node) override
    {
        result = Call(node->x.get()) * Call(node->y.get());
    }
};

double Eval(shared_ptr<Expr> node)
{
    EvalVisitor visitor;
    node->Accept(&visitor);
    return visitor.result;
}

```

这个设计模式非常好懂，也很直接，对比一下虚函数的例子，可以看出他们的一一对应的关系。顺便，研究函数式语言如何实现，有一个简单的办法就是去把F#生成的exe反编译成C#，可以发现很多惊喜。

其实Visitor模式讲的就是在不需要扩充新的子类的时候，如何添加新的虚函数而不需要修改原有代码。当然虚函数也有它的好处，就是添加新的子类的时候不需要修改原有代码。所以看你的业务逻辑，到底是添加新子类多，还是添加新虚函数多，从而

选择要不要把程序写成基于Visitor模式的样子。

对于编译器来说，整个处理流程那么复杂，所以等于需要经常添加虚函数，因此就都把本来是虚函数的东西改成了各种Visitor。这个时候，如果你修改了语法，那么每一个Visitor都会曝出语法错误，所以这等于变相通知你所有需要修改的东西在哪里——如果你能坚持不因为偷懒而使用dynamic_cast的话。

那这个跟上一篇文章说的事情又有什么关系呢？因为Workflow脚本的AST大约有100个类，每次真的这么一个一个覆盖虚函数，写到Google倒闭都写不完（逃。而且一些事情之间是有共性的，所以我决定给ParserGen.exe加入三个与定义好的Visitor。

第一个是EmptyVisitor，其实很好理解，就是所有的虚函数都覆盖了，全都留空。大概的意思就是，如果你的一个虚函数仅针对有限的几个类型要做运算，那么就可以使用EmptyVisitor，然后覆盖你有限的几个运算。所以ParserGen给EmptyVisitor生成的一大坨代码，实际上就是实现了Haskell的“_”字符（逃

第二个是TraverseVisitor，这个Visitor会提供一系列新的、空的虚函数Traverse给你覆盖。当你给他一个AST的类型的时候，他就会从这个节点开始自动去解除下面的全部子节点，然后针对每一个子节点的继承树上的所有类型去调用一系列Traverse函数。

这也有一个简单的例子。Workflow脚本支持两种lambda表达式的写法，其中一种是比较精简的，适合你写简单的运算：

```
function<int(int)> adder = [](int a, int b){ return a + b; };
```

V.S.

```
var adder : func(int):int = [$1+$2];
or
var adder = [$1+$2] of func(int):int;
```

注释：

之所以需要指明类型，是因为C++和Workflow的关系，就如同ObjC和Swift，或者C++和C++/CX一样。所以在GacUI里面，Workflow除了预定义的容器以外没有泛型，跟Go类似，不然从Workflow生成C++代码和从C++里面反射进Workflow就会多无尽的麻烦（C++/CX其实也一样，但是人家多（逃，硬是把这个事情给做了）。如果是C++的话，可以直接把lambda表达式写成泛型的。

这个语法看上去好像很好，但是实际上你会遇到这样的问题：表达式

```
[f($1,[$1+$2])]
```

的\$2到底算谁的呢？在这里我做了一个简单的规定，就是嵌套的中括号里面新出现的名字，在外面就当没看见，所以实际上这个表达式讲的就是：

```
[=](auto _1){ return f(_1, [=](auto _2){ return _1 + _2; }); }
```

那么在进行语义分析的时候，我自然就有必要去搜索一个中括号lambda表达式里面，他到底有多少参数。大概的算法就是，我把他所有子节点都遍历一遍，如果遇到了里面的一个中括号lambda表达式，就不继续往下找了。那么我就可以使用ParserGen给我生成的wl:workflow:traverse_visitor:ModuleVisitor（Visitor有若干个，每一个根类型都有），简单的写成这样：

```
class SearchOrderedNameVisitor : public traverse_visitor::ModuleVisitor
{
public:
    WfLexicalScope* scope;
    SortedList<vint>& names;

    SearchOrderedNameVisitor(WfLexicalScope* _scope, SortedList<vint>& _names)
        :scope(_scope)
        , names(_names)
    {
    }

    void Traverse(WfOrderedNameExpression* node) override
    {
        vint name = wtoi(node->name.value.Sub(1, node->name.value.Length() - 1));
        if (!names.Contains(name))
        {
            WfLexicalScope* currentScope = scope;
            while (currentScope)
            {
                if (currentScope->symbols.Keys().Contains(node->name.value))
                {
                    return;
                }
                currentScope = currentScope->parentScope.Obj();
            }
        }
    }
};
```

```
        }
        names.Add(name);
    }

void Visit(WfOrderedLambdaExpression* node) override
{
    // names in nested ordered lambda expression is not counted
}
};
```

大家不用去管中间一坨操作Scope的代码，其实大概内容说的就是，在表达式

```
[f($1,[$1+$2])]
```

内部的

```
[$1+$2]
```

里，\$1是从外面捕获来的，而不是这个lambda表达式的参数。

那么在traverse_visitor:ModuleVisitor出现之前，代码又是怎么样的呢？请大家看这个文件：[WfAnalyzer_SearchOrderedName.cpp](#)，很明显可以看到，这里面四百多行代码里面，有四百行代码其实都是垃圾代码。在其他地方也需要做类似的事情，所以以前整个编译器都充满了垃圾代码，这次就一次性删清光了。添加新的跟这个业务无关的表达式，也不用改Visitor了。

第三个是复制AST的。例子就不举了，大概就是说，如果你有一个算法，大概是需要在复制一棵语法树的时候，修改里面的一点点东西，那么你就直接用copy_visitor:xxxVisitor，然后覆盖里面的一点点虚函数就可以了，不用每一次都写一大堆垃圾代码，就为了遍历和复制这些东西（旧：[WfAnalyzer_GenerateBind.cpp](#) v.s. 新：[WfAnalyzer_GenerateBind.cpp](#)）。

当然copy_visitor的另一个作用就是，赋值表达式的时候，绝对不会漏掉表达式里面的所有位置信息，这对GacUI的错误信息格式化非常重要（逃

做到这里，ParserGen的修改也差不多结束了，接下来对Workflow再做一些小修改，就可以开始做Coroutine了！

Coroutine的文章终于可以动工了！

给Workflow加上Coroutine并撰写《考不上三本也能读懂系列——如何实现coroutine》之前的最后一个任务终于完成了！现在只需要继续添加单元测试，把分支都覆盖了，就可以开始动手了。这个文章大概会有那么三四篇，我给Workflow的功能加到哪，我就写到哪。

GacUI的XML资源最后会被编译成一个巨大的Workflow脚本，脚本编译完会生成C++。你可以在资源里面写各种各样的东西，譬如说data binding的一个表达式，或者调用外部API的事件处理函数等。这些脚本的碎片自然也会被parse之后粘进最后的脚本里面去。那如果你写错了怎么办呢？GacUI自然要给你报告错误信息。但是我怎么才能知道脚本里面的碎片原本是在XML的什么地方呢？

其实原理十分简单粗暴。我把每一个XML里面的脚本碎片parse完之后，遍历所有的AST节点（所以才有[ParserGen生成预定义好的各种visitor - 知乎专栏](#)），把节点里面的位置信息改为XML里面的位置信息。这一步还是很简单的，因为根据脚本碎片第一个字符的位置，我就可以很轻松的算出来每一个节点的位置是什么。后面会做一些琐碎的事情，确保最后生成的大脚本的每一个节点的位置都是有记录的。Workflow的错误信息的位置直接就给的AST节点的指针，所以最后重整成字符串输出的时候，就很方便。

题图显示的最近做的工作的效果。上半部分的六条红线就是下半部分六个错误信息的具体位置。看起来还是很准确的。

一个有点规模的项目，除了激动人心的部分以外，总是有很多琐碎的事情。而这些琐碎的事情是不得不去完成的。我也想只做有挑战的部分，但是开发GacUI又不像MSRA可以招实习生，只能自己来了。这就是为什么GacUI开发的那么慢。

其实[GacUI的第一个commit](#)还是在天国的Codeplex.com上面的，而这个commit里面的一些代码（就是现在的GacUI\Source\NativeWindow\Windows），其实是大三在微软实习的时候，[每天晚上在宾馆做一个封装Win32 API的GUI库](#)里面的代码。所以虽然GacUI已经是写了六年的项目了，但实际上里面的一部分代码的年龄已经超过了10年。而这些代码使用的[跨平台的基础库](#)，前身其实是当年Borland变傻逼，我决定弃用Delphi之后，在VC++2005的年代重新实现一部分VCL以便学习C++而写出来的。

所以经历过那个时代的人，大概还可以在我的代码里面看出一点痕迹，虽然改着改着已经不是很多了。不过如果你们去看我的博客的很多早期的代码的话，你们会发现一开始我甚至把Object Pascal的命名规则带了进来（逃。但是C++跟这个规则结合起来还是太丑，于是现在就改成了.net framework的命名规则。还是.net的代码看起来最舒服。

考不上三本也能懂系列——前言

在准备了好几天之后，给GacUI的脚本加上Coroutine的主要工作在两个晚上就完成了，所以我终于可以开始写这个系列文章了。这个系列当然不仅仅包括Coroutine，只是之前答应了[@RednaxelaFX](#)说，跟语言有关的就放在他那里。这样这个系列的文章就不一定全部都在这个专栏里面了。为了让大家也可以从我的专栏里面访问到，我就写个前言，顺便当目录用。

- 前言（就是本篇）
- [考不上三本也能给自己心爱的语言加上Coroutine（一） - 知乎专栏](#)
- [考不上三本也能给自己心爱的语言加上Coroutine（二） - 知乎专栏](#)
- [考不上三本也能给自己心爱的语言加上Coroutine（三） - 知乎专栏](#)
- [考不上三本也能给自己心爱的语言加上Coroutine（四） - 知乎专栏](#)
- [考不上三本也会实现数据绑定（一） - 知乎专栏](#)
- [考不上三本也会实现数据绑定（二） - 知乎专栏](#)（新！）

为什么要写这个系列呢？主要目的是为了破除迷信。在多年知乎上摸爬滚打之后，我终于明白了知友们的言论，都是很极端的。一个东西牛逼，就一定要拿诺贝尔奖你说话才算数。一个东西简单，那么小学生都要会做。贫穷指的都是山区里面吃不饱饭的，富裕则一定要使用存款利息就能买学区房的。那么编程也一样，要么就说凭什么中专就得被扔简历，要么就说你不读到博士你就什么都没学。这都是狗屁！真理永远只有一个，标准也永远是全球统一的。

人类的世界是残酷的，你贫穷就是因为比你有钱的人都想干你，所以维持你很贫穷，这就是唯一的原因了。所谓的秩序就是这个样子的，不牺牲落后的人，怎么保证人类可以在耗光资源之前，能达到征服银河系的目标，从而让种群永远的存活下来？

既然走错一步就会死，那么破除迷信就显得很重要了。编程这个事情，是不需要智商的。大三的时候我曾经抓了一个新鲜热辣的大一的妹纸（who is 汕头人）教编程。其实所谓的教，就是每个星期手把手地讲一堂课，系统地我认为要学会的东西都按顺序讲出来，目的是为了看我自己对编程的理解对不对。我就是抱着一种教废了就算了的心情去教的。然而结果人家大二就能自己实现正则表达式引擎，大四就已经能从0开始执行一个动态语言了。后来我用这个语言写了個Linq，给她当测试用例，这么复杂的代码跑过了，证明实现是有质量的。这个人后来拿过微软网易百度腾讯的offer，干得风生水起。

然而你说她聪明吗，其实根本不聪明（逃，就是一个普通人，而且还有点蠢，很多简单的东西讲了多少遍才明白（侧面证明了拿offer也是不需要智商的（逃）。这证明了，你要在本科四年里面搞出这么多事情，你只要有一颗正常的大脑就可以了。她能做到，你们当然也可以做到。但是这跟知乎上那些从五花八门的专业全都要跳来当码农是不一样的，因为虽然你可以做到，但不代表你可以马上学会，立刻赚钱。这就跟传说中爱迪生找钨丝的故事一样，实验了多少遍才找到结果。学编程也是一样的，只要你练习了一百万行代码（每天500行，也就是6年而已，6年很长吗？），你什么基础知识语言类库不能学会？当然并没有人愿意去练习一百万行代码，殊不知这就是唯一的办法。

你们也不要觉得一百万行很吓人，其实做个GacUI的那本脚本引擎，才屁大点事，四万多行就没有了，其中做个Coroutine的核心功能就没了一千多。GacUI那么简单的类库，十几万行就没有了。所以一百万行其实做不了多少事情的，也就写几个程序而已。大学的时候我有一个同学，高中的时候给知名P2P软件写过代码，上大学自己维护了一个VB写的聊天程序卖给了一大堆大学和公司，他自己说那个时候他已经写过一百万行代码了。然而就算是这样，毕业的时候也因为面试出了岔子，只好去读个硕，然后去了苹果公司。这么牛逼的人本科毕业的时候都找不到靠谱的工作，你觉得你半路出家行吗？

所以不想努力的人，趁早滚蛋，不然以后栽在这上面，那多不好。但是你愿意努力，只是觉得自己蠢，这没有问题。蠢无非就是多花点时间嘛。虽然考不上三本也能学会Coroutine，但是你跟我的区别无非就是能不能在两个晚上做出来而已。难道做一个月就不是做，就不算做出来了吗？当然不是。

当然，做出来的意思并不是说做出来一个Demo。Demo都是很容易做的，你要是发现谁谁谁说自己会什么，其实只是做了个Demo，还是说只是完成了老师布置的作业，那么其实就等于“可能还”不会。一个真正的程序，当然要处理现实世界会出现的一些复杂的情况，你还要懂得怎样保证自己的程序在绝大多数情况下是对的，所以这也需要很多技巧。有些人可能会说，可以用Coq等系统验证程序啊。好吧，你用Coq给clang++写个证明，应该可以写到80岁，然后发现写不出来。

当然我知道我光这样说你们也不会相信的，所以这个系列的存在意义也是如此。我会挑一些表面上看起来高大上的知识，然后告诉你们，为什么考不上三本也能学会，也能把这个程序做出来。所以这个系列也是一个指南针，因为我说的这么直白，要是屏幕前的你发现自己恰好就是那一小撮还是学不会的人，那你可以好好考虑一下自己的前程，是不是真的要做程序员（逃

Coroutine虽然简单，但是写的时候你的代码也要处理一亿个分支逻辑，所以还是花了点时间，做了出来还是觉得很爽。这几天用了消耗了这么多脑力，刚才称了一下都只剩下17斤了，果然就算是写自己早就知道的东西，也需要消耗大量的脂肪。那么大家就期待明天的第一篇——《如何实现Coroutine》吧。Coroutine一篇是写不完的，所以我会分开写，看完第一篇还没学会也不要害怕，后面还有，都不会再说（逃。关注了这个专栏但是没有关注我的，你们可以通过轮询这篇文章来发现即将发在别人的专栏上的正片。

考不上三本也能给自己心爱的语言加上Coroutine（一）

你现在所阅读的并不是第一篇文章，你可能想看[目录和前言](#)。

我发现发给了R菊苣的专栏之后再投稿给自己就好了，真机智！

终于要开始进入正题了。昨天在前言下面就看到有人问Coroutine是什么，其实这也是正常的，虽然Coroutine很常见，但你不一定能直接用得上。像古时候Windows 3.x系列的协作式多线程，其实就是Coroutine的一种表达形式。你需要主动放弃对CPU的占用，然后CPU就可以让别人进来。所以你会发现很多古老的API都有提到这一点，譬如[GetMessage函数](#)。GetMessage在这里就变成了Coroutine的一个operator。以前不能主动中断的时候，如果API还不引诱你去调用这些函数，那整个系统就只有你一个程序可以运行了。

所以Coroutine满大街都是，核心的想法就是你写一个函数，然后函数自己决定什么时候中断自己的执行，等待别人唤醒。所以我们可以看到很多语言很直接粗暴地就这样实现Coroutine了，譬如VC++早期的`_await`关键字，就是通过跑了一半把堆栈（其实就是一组寄存器）换掉来实现的。既然Coroutine本来就是要让几个函数交替执行，那我直接交替执行他们不就行了嘛？你们还可以从很多脚本语言里面看到类似的东西。

这种做法有一个好处，就是Coroutine跟其他所有的feature都是正交的，你什么代码都不用改，直接在编译器上做点手脚，然后改虚拟机就好了。但是如果你不是这一系列工具程序的owner，那你就会很蛋疼。万一我hack了半天，结果人家上游下来一个改动，造成了一万个conflict怎么办？或者说，我根本就不能决定语言用的是哪个runtime怎么办？所以只有当你真的拥有这门语言的时候，你才能这样做。

既然不改虚拟机，那只能改编译器了。通常来讲还有另一种办法，就是要做全文CPS变换。当然这听起来好像不知所云，其实核心思想很简单，在我自己以前的[vczh/tinymoe](#)项目里面就实现过。这是一门把自己编译成C#的语言，语言本身暴露了continuation。所谓的continuation的意思就是说，你可以在任何地方下个“断点”，然后编译器会把“剩下的部分”包装成一个闭包（或者通俗一点叫lambda表达式）给你。如果你直接问，那包装到底是怎么做的呢？很多人可能会让你去看论文。但是如果你不在乎优雅的话，其实不看也罢，我也是做出来了之后才发现原来连这种东西也可以发论文的，真是大开眼界（逃）

举个简单的例子，我有一个这样的函数：

```
int Fuck(IEnumerable<int> xs)
{
    int sum = 0;
    foreach(var x in xs)
    {
        SHIT!
        sum += x;
    }
    return sum;
}
```

现在执行到SHIT!这里，我打算做一个断点，让编译器把剩下的部分包装成一个闭包给你。那么这个闭包长啥样子呢？首先你要把foreach这个语法糖解开：

```
int Fuck(IEnumerable<int> xs)
{
    int sum = 0;
    var _xs = xs.CreateEnumerator();
    while (_xs.Next())
    {
        var x = _xs.Current;
        SHIT!
        sum += x;
    }
    return sum;
}
```

那么在执行到了SHIT!之后，如果你把剩下的部分写成一段代码（注意这个函数重新执行到SHIT!之后仍然要停止），自然就变成了：

片段1：（这是第一个SHIT!前面的部分）

```
int sum = 0;
int x;
var _xs = xs.CreateEnumerator();
if (_xs.Next())
{
    x = _xs.Current;
    /* SHIT! 向片段2 */
}
else
{
    return sum;
}
```

片段2（这就是剩下的部分）：

```
sum += x;
if (_xs.Next())
{
    x = _xs.Current;
    /* SHIT! 向片段2 */
}
else
{
    return sum;
}
```

那么你从片段1开始，每次遇到SHIT!的时候就停下来，等到恢复的时候，就执行当初SHIT!的目标。譬如说第一个片段，如果你狗屎运`_xs.Next()`返回false，直接return了，那就没有什么SHIT!了。但是万一你执行到了SHIT!，那么函数到这里也就结束了，等别人唤醒你的时候，你就从片段2开始执行，一直到什么时候遇到return为止。这是不是很像给程序打了个断点？

CPS变换的意思就是说，随便给你一个SHIT!，然后你要照着原来的程序，把剩下的部分写成上面那样。当然实际情况比这个更复杂，因为你要考虑到这个SHIT!可能会出现在你要调用的函数的里面，那事情就没这么好办了。

所以[vczh/tinymoe](#)暴露continuation的意思也就是说，你可以在语言任意的地方写上SHIT!，然后编译器就想办法把“剩下的部分”，通过全文CPS变换，编译成一个闭包（也就是lambda表达式、函数对象、托管函数指针，etc），直接给你，然后你自己想办法去调度。当初我也是为了练习编程才做成这样的。直接的结果是什么呢？看看[这个文件](#)就知道，语言只需要提供递归跟分支结构就可以了，剩下的部分全部都可以写成库，哪怕是循环和异常处理都能做出来。

注：SHIT!在某些Lisp语言里面叫call-cc。

所以大家就会在项目一开始看到，这个语言的其中一个例子就是如何几十行就地做出一个yield return。这也是如何添加Coroutine的一个例子。如果语言本身提供continuation，那实现Coroutine根本不是事。

只不过这个文章的标题是[给自己心爱的语言加上Coroutine](#)，而且除了Lisp以外，估计不会有任何一门提供continuation的语言会成为谁心目中心爱的语言，那么这个方法也就行不通了。

那最后剩下什么呢？这也是除了修改虚拟机以外，大部分语言的做法，特别是自从C#做出了await之后，被各种语言广泛抄走，用的也是我现在要讲的最后一一种办法，这要求你规定SHIT!不能默默地在你调用的函数里面出现，如果他一定要出现，那么你要用特殊的语法来调用这个函数（譬如说使用`await`关键字）

这是什么意思呢？考虑一下下面这个程序：

```
int Shit(int x, int sum)
{
    SHIT!
    return sum + x;
}

int Fuck(IEnumerable<int> xs)
{
    int sum = 0;
    var _xs = xs.CreateEnumerator();
    while (_xs.Next())
    {
        var x = _xs.Current;
        sum = Shit(x, sum);
    }
    return sum;
}
```

这里的SHIT!就出现在了Fuck调用的Shit函数里面。那么你说这样的函数我要怎么解continuation呢？我在看Fuck的时候我怎么会知道Shit里面有SHIT!？万一Shit函数是个虚函数怎么办？万一这个虚函数还是另外的dll提供的怎么办？是吧，这就是语言不提供continuation，你也不能修改虚拟机（其实修改虚拟机也就等于提供continuation）的时候，你要给调用Shit函数的地方打一个标记的原因。那么我们可以把代

码改成这样：

```
SHIT_CALLABLE! int Shit(int x, int sum)
{
    SHIT!
    return sum + x;
}

SHIT_CALLABLE! int Fuck(IEnumerable<int> xs)
{
    int sum = 0;
    var_xs = xs.CreateEnumerator();
    while (_xs.Next())
    {
        var x = xs.Current;
        sum = SHIT_CALL! Shit(x, sum);
    }
    return sum;
}
```

就可以了！其实想想很容易明白，如果一个返回int的函数执行到SHIT!的时候就会停下来等我再次唤醒它，那么它怎么可以返回int呢，返回成int我要上哪唤醒？这就像yield return要求你返回IEnumerable<T>，await要求你返回Task<T>一样，我们也可以要求包含SHIT!的函数返回一个我们定义的接口：

```
interface IShitCallable<T>
{
    T Result {get; set;}
    bool ShitCall();
}

SHIT_CALLABLE! IShitCallable<int> Shit(int x, int sum)
{
    SHIT!
    return sum + x;
}

SHIT_CALLABLE! IShitCallable<int> Fuck(IEnumerable<int> xs)
{
    int sum = 0;
    var_xs = xs.CreateEnumerator();
    while (_xs.Next())
    {
        var x = xs.Current;
        sum = SHIT_CALL! Shit(x, sum);
    }
    return sum;
}
```

这样我们调用它Fuck，他就返回一个IShitCallable<int>。我们拿到这个IShitCallable<int>，就不断地ShitCall它，直到返回true为止，然后去取Result属性。当我们解开Fuck函数的时候，由于Shit函数返回的也是一个IShitCallable<int>，我们也可以完美地做出continuation了。

现在我们就来尝试一下解语法糖，把SHIT、SHIT_CALL和SHIT_CALLABLE!从代码里面拿掉，变成普通的C#代码！

首先我们要对付的是Shit函数，Shit函数其实比较简单，因为没有分支也没有循环，那么我们粗暴的拆成两半就可以了。根据之前提到的做法，SHIT!前和SHIT!后是两段不同的代码，中间的SHIT!会告诉你下一段代码是什么，所以我们用一个int给他们编号就好了。然后就变成这样：

```
class Shit_IShitCallable : IShitCallable<int>
{
    public int state = 0;
    public int x;
    public int sum;

    public int Result {get; set; }

    bool ShitCall()
    {
        while (true) // 其实每一个分支都会退出所以这个while (true)等于没写
        {
            switch (state)
            {
                case 0:
                    {
                        /* SHIT! 就编译成下面两行 */
                        state = 1;
                        return false;
                    }
                    break;
                case 1:
                    {
                        /* return 就编译成下面三行 */
                        Result = sum + x;
                        state = -1;
                        return true;
                    }
                    break;
                default:
                    throw EatShitException();
            }
        }
    }
}

IShitCallable<int> Shit(int x, int sum)
{
    return new Shit_IShitCallable() { x=x, sum=sum };
}
```

然后我们要对所有的SHIT_CALL!都解开变成普通的函数调用，其实就是把它弄成一个带SHIT!的循环：

```
SHIT_CALLABLE! IShitCallable<int> Fuck(IEnumerable<int> xs)
{
    int sum = 0;
    var_xs = xs.CreateEnumerator();
    while (_xs.Next())
    {
        var x = xs.Current;
        var_Shit = Shit(x, sum);
        while (!_Shit.ShitCall())
        {
            SHIT!
        }
        sum = _Shit.Result;
    }
    return sum;
}
```

你们看，只要加上了特殊的语法（SHIT_CALL!），那么其实我们根本就不关心Shit里面到底长什么样子，因为所有的SHIT!都只会直接出现在函数里面，出现在被调用的函数里面的SHIT!我们都可以置之不理。

因此剩下来就很简单了，基本上就是每个while变成两个片段。这里要注意的是，由于我们有多个互相嵌套的while，所以不能直接展开，需要添加一些“软SHIT!”，可以大大降低算法的脑力负担：

```
while (CONDITION)
{
    STATEMENTS;
}
==>
while (true)
{
```

```
软SHIT! // 也就是虽然打了断点，但是如果命中了它，就立刻继续执行
if (!CONDITION) break;
STATEMENTS;
}
```

展开完成后就变成这样：

```
class Fuck_IShitCallable : IShitCallable<int>
{
    public int state = 0;
    public IEnumerable<int> xs;
    public int sum;
    public IEnumerator<int> _xs;
    public int x;
    public IShitCallable<int> _Shit;

    public int Result {get; set; }

    void ShitCall()
    {
        while (true)
        {
            switch (state)
            {
                case 0:
                {
                    sum = 0;
                    xs = xs.CreateEnumerator();
                    /* 其实这相当于我们认为在每一个while的最前面插入SHIT!，然后不中断，不然直接展开会陷入死循环。SHIT!只出现在嵌套的while里面就会这样。这种软SHIT!就是跟着continue语句的，代表我们并不想中断，j */
                    state = 1;
                    continue;
                }
                break;
                case 1:
                {
                    if (_xs.Next())
                    {
                        x = xs.Current;
                        _Shit = Shit(x, sum);
                        state = 2;
                        continue;
                    }
                    else
                    {
                        Result = sum;
                        /* 这是硬的return */
                        state = -1;
                        return true;
                    }
                }
                break;
                case 2:
                {
                    if (!_Shit.ShitCall())
                    {
                        /* 这是硬的SHIT! */
                        state = 3;
                        return false;
                    }
                    else
                    {
                        sum = _Shit.Result;
                        state = 1;
                        continue;
                    }
                }
                break;
                case 3:
                {
                    state = 2;
                    continue;
                }
                break;
                default:
                    throw EatShitException();
            }
        }
    }
}

IShitCallable<int> Fuck(IEnumerable<int> xs)
{
    return new Fuck_IShitCallable{ xs=xs };
}
```

到了这里，相比大家已经明白了SHIT!、SHIT_CALL!和SHIT_CALLABLE!是怎么回事了，应该很快就可以把它们对应到各种语言的牛逼的功能里面去了（譬如await）。

总结一下，实现Coroutine主要有三种方法：

1. 改虚拟机

- 1. 好处：实现简单，跟语言的其他功能是正交的
- 2. 坏处：只要你的改动不能merge回主分支，你就会一辈子蛋疼地conflict下去

2. 语言直接提供continuation

- 1. 好处：有continuation可以实现非常强大的控制流语句，Coroutine也只是其中的一个作用而已，你不需要专门为Coroutine做什么
- 2. 坏处：这样的语言并不常见

3. 要求SHIT!只能出现在SHIT_CALLABLE!函数里面，并且调用SHIT_CALLABLE!函数要用特殊的语法SHIT_CALL!

- 1. 好处：continuation毕竟是闭包，各种闭包群P容易给GC造成压力（这是Don Syme告诉我的，当初我发邮件问他为什么F#的computation expression的循环不支持break，他说这样就不能编译成状态机了），而直接改状态机并没有这个问题，甚至是像C++这样没有GC只有shared_ptr的语言也可以完美支持
- 2. 坏处：需要改语法

第一篇就讲到这里了，现在你们应该学会人肉实现Coroutine了吧？这也是考不上三本的人所应该具有的基本能力。人肉实现总归是比较简单的，但是你们能不能写一个程序来代替自己人肉实现Coroutine呢？这就是接下来的几篇要讲的内容。

除此之外，我还将介绍[GacUI的Workflow脚本语言](#)是如何在提供基本的Coroutine语法结构的情况下，可以让你自己来实现yield return和async await的。这样你就可以创造自己无穷多的Coroutine类型，而不用等语言来帮你做。

最后大家可能会有个疑问，如果我要实现方法2，那这到底有多难？这当然是非常难，要考上三本才会做！

考不上三本也能给自己心爱的语言加上Coroutine（二）

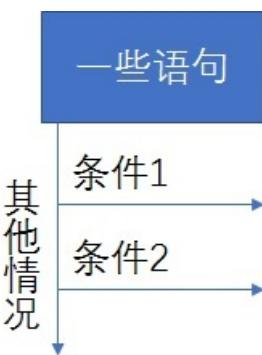
你现在所阅读的并不是第一篇文章，你可能想看 [目录和前言](#)。

上一篇提到了一些考上了三本的人必然会明白的知识，毕竟只是改写代码而已，让你把代码从一个样子翻译成另一个样子，这种事情就像把主动（push）模型的代码和被动（pull）模型的代码来回改一样容易——毕竟说的就是同一件事情嘛。写个for循环输出1到10，是push。写个接口人家调用你10次依次返回1到10，是pull。push转pull无非就是把本来函数里面的变量放到了成员变量里面去，然后再搞点事情。很多架构上的问题，都可以通过在push和pull之间来回穿插来解决，后来就衍生出了很多概念。如果不熟悉push和pull的做法的话，这些概念你可以学很久。

不过这毕竟是人在做，人在做总是可以不断的试错最后得到一个方法。但是要写程序去做怎么办呢？其实这用的知识更简单，是高中上数学课就讲的（年纪大一点的可能没有，广东是比我小两届的人开始学的）画流程图的问题。以前学习的都是把流程图表达成程序，现在要反过来，把程序表达成流程图。如果你做到了这一点，那么这个实现Coroutine的算法基本上就完成了90%了。

今天先讲简单的那部分，不涉及对try-catch语句的处理。处理try-catch一直都是一件麻烦的事情，早期C#的await的实现甚至禁止你把await写在try里面，最大的原因其实是想要不出bug，因为细节实在太多（逃

那现在先来普及一下流程图的概念。这里用的流程图跟高中的流程图有一点区别。高中的流程图每一个结构都很简单，但是功能很多。现在我这个流程图的一个节点比较复杂一点，但是就只有这一个节点。节点长这样：



每一个节点都会包含要执行的一些语句，执行完之后，会按照（有顺序的）条件跳转，如果都不满足，或者干脆没有条件，就跳转到“其他情况”的方向。

现在让我们来回顾一下之前的那个函数（首先要解语法糖，包括SHIT_CALL!语句）：

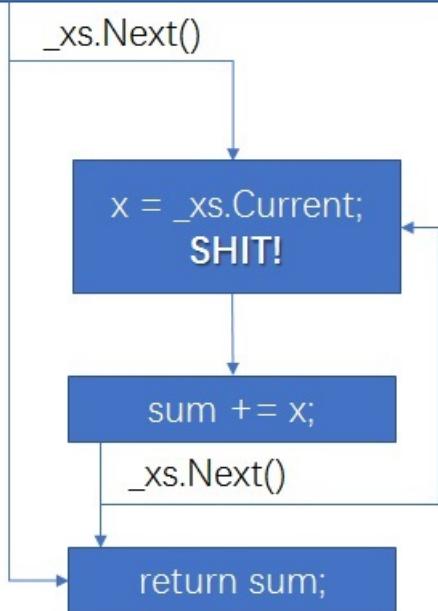
```
int Fuck(IEnumerable<int> xs)
{
    int sum = 0;
    var _xs = xs.CreateEnumerator();
    while (_xs.Next())
    {
        var x = _xs.Current;
        SHIT!
        sum += x;
    }
    return sum;
}
```

现在我们要做两件事情，第一个是要把所有的变量都声明在函数的最前面（于是就不会被包括在代码里），然后变量声明就变成了普通的赋值，第二个就是画流程图了。其中要注意的一点事，我们也要把SHIT!变成一个单独的语句，而且SHIT!后面的语句必须开启一个新的节点。在这里我们并没有要求说一个流程图的节点只能包含一条语句，你当然想塞多少就塞多少。节点做了少你难免需要在不同的地方重复相同的语句，节点做了多你浪费在while(true)里面那个switch的时间就多，该怎样适可而止，是一个权衡的问题，我就不深入讲了。现在大家可以尝试一下在纸上画出这个函数的流程图，记住 SHIT!后面的语句必须开启一个新的节点，被跳转的地方当然也只能新开节点，画完应该全部都是长下面这个样子：

```

    sum = 0;
_xs = xs.CreateEnumerator();

```



注意到这里的`_xs.Next()`被重复了两次。通常直接用程序生成的流程图，可能直接就给个菱形——也就是没有语句只有条件的节点，然后在`sum+=x;`后面指过去。这当然也是可以的，最后要不要变成这个图的画法也是大家的选择。现在我在Workflow的实现就没有输出这么紧凑的流程图，以后发现有性能问题了再改（逃）

那有了这个流程图之后怎么办呢？之前说搞定流程图就搞定了90%，剩下的10%自然是直接翻译成代码啦。在翻译的时候要注意下面几点：

- 所有在不同的节点里面都用到的变量都要搬进成员变量里（其实你偷懒全部搬进去也可以）。
- 每一个节点都是一个相应的case语句。
- 生成正确的节点间跳转的代码（在下面）。

在这里软SHIT!都没有画出来。其实只有SHIT!结尾的节点的跳转才是硬的，其他的都是软的。还记得软硬SHIT!出来的代码的区别吗？遇到`return`也要生成相应的代码。其实死代码处理的方法也是类似的，你从函数的开头开始，不断的做深度优先搜索，遇到`return`就停下来。搜完之后所有没碰到过的代码都是死代码。

现在我们给流程图里面的四个节点分别编号为0、1、2、3，然后先给出一个大的框架。这个框架除了case的数目和变量的数目有变化以外，其他的都是死的。

```

class Fuck_IShitCallable : IShitCallable<int>
{
    public int state = 0;
    public int Result {get; set; }

    public IEnumerable<int> xs;
    public int sum;
    public IEnumerator<int> _xs;
    public int x;

    void ShitCall()
    {
        while (true)
        {
            switch (state)
            {
                case 0:
                    // 待填
                    break;
                case 1:
                    // 待填
                    break;
                case 2:
                    // 待填
                    break;
                case 3:
                    // 待填
            }
        }
    }
}

```

```

        break;
    default:
        throw EatShitException();
    }
}
}

IShitCallable<int> Fuck(IEnumerable<int> xs)
{
    return new Fuck_IShitCallable{ xs=xs };
}

```

函数进来的第一个节点永远编号为0，把state初始化成0也就是这个意思。函数的返回值就给存到Result里面。然后执行下面几步：

- 遇到跳转就编译为 { state = ?; continue; }
- 遇到SHIT!就编译为 { state = ?; return false; }
- 遇到return就编译为 { Result = ?; state = -1; return true; }
- 其他的照搬

然后我们就可以分别把4个待填的地方分别填入：

case 0:

```

sum = 0;
_xs = xs.CreateEnumerator();
if (_xs.Next()) { state = 1; continue; }
{state = 3; continue; }

```

case 1:

```

x = _xs.Current;
{ state = 2; return false; } // SHIT!

```

case 2:

```

sum += x;
if (_xs.Next()) { state = 1; continue; }
{state = 3; continue; }

```

case 3:

```

{ Result = sum; state = -1; return true; }

```

大功告成！

对比一下上一篇文章的代码，一眼看上去结构差很远，但实际上表达的是同一个意思。push转pull带来的一个问题就是代码会不会很好懂，所以语言才需要提供这样的一个机制，好让我们写出好懂的代码。

同样是把普通函数转成Coroutine，是不是今天感觉就简单好多了，很多昨天还觉得云里雾里的，是不是今天就看得比较清楚了？你这么感觉就对了！本来这就是考不上三本也会的内容嘛，要是看起来觉得难才应该检讨一下自己是不是真的毕业了要做程序员。

今天就写到这里了。到了这里，在实现Coroutine的道路上，我们已经从昨天的：

函数 -(人肉处理)→ Coroutine

变成了今天的：

函数 -(人肉处理)→ 流程图 --(机械化处理)--> Coroutine

了，我们经过了一篇文章的时间，就已经把问题简化了。那么从函数弄出来流程图是不是也有机械化的方法呢？答案当然是有的。但是我先想让大家消化一下，毕竟考不上三本的人也不会是什么天才，不要浮躁，慢慢学才是硬道理。

下一篇将会讨论如何做出一个函数到流程图的算法，顺便把try-catch的事情也考虑进去。

考不上三本也能给自己心爱的语言加上Coroutine（三）

你现在所阅读的并不是第一篇文章，你可能想看 [目录和前言](#)。

昨天写了那么多高中程度的东西，今天继续来讲一些对于普通人来说上了大学才会接触到的。回顾一下之前几篇的内容：

(一)： 函数 -(人肉处理)-> Coroutine

(二)： 函数 -(人肉处理)-> 流程图 --(机械化处理)--> Coroutine

那么今天文章的内容将是：

(三)： 函数 -(人肉处理)-> 语法树和符号表 --(机械化处理)--> 流程图 --(机械化处理)--> Coroutine

既然文章的标题是《给心爱的语言加上Coroutine》，那么我们当然可以假设已经存在了一个完整的编译器，只是缺少Coroutine，因此语法树和符号表都是现成的——这些东西对于绝大多数语言来说，也超不过三本的内容。你们有兴趣的话，我以后再讲，这几天先把Coroutine讲完。

大家也不要觉得符号表很害怕，我们这里用到的东西都很简单，大概就只有：

- 知道每一个break（或continue等）语句到底break的是哪一个循环
- 知道代码里访问的每一个变量到底是在哪里定义的

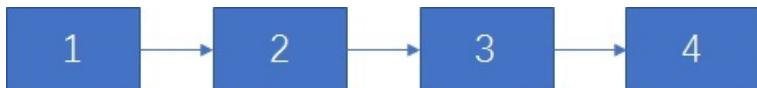
这就够了！符号表的意义就在于，把代码从头到尾读几遍，然后把这些信息都整理出来，以后可以有各种用途。

相比起前面的几篇文章，今天的内容比较复杂，需要读者提前掌握的内容有

- 指针：这没什么好说的，我们也不玩复杂的东西，你只要会new、delete和->就够了。
- 链表：也不需要懂什么复杂的东西，你只需要会添加删除节点就可以了。
- 递归：啊，递归！
- 深度优先搜索：语法树也是树，深度优先搜索就是，在你能想到的所有“把树里面所有的指针都读一遍”的方法里面，最简单的那个。你不需要知道这个概念，你第一个写得出来的方法，肯定就是深度优先搜索。

所以大家不要害怕，考不上三本也是没有问题的！

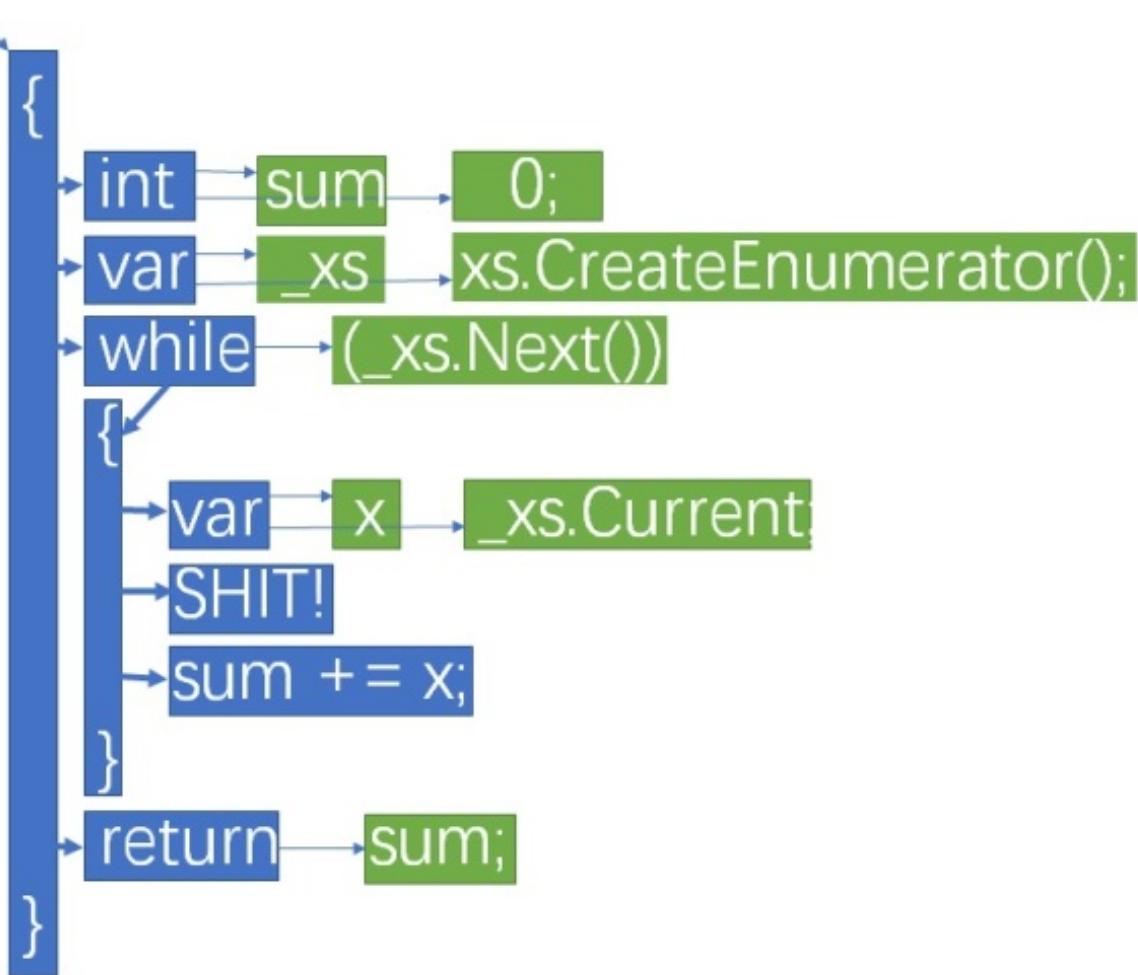
听说很多学校都没有好好讲编译原理的内容，那么自然就会有很多人不知道语法树是什么东西。其实这个概念是很简单的。我们回想一下，课本上是怎么表达一个链表的？



一个方框就是一个对象，而一个箭头就是一个指针成员变量（如果用的是C#那就更简单了）。那么语法树当然可以用这种方法表达出来。区别只在于，语法树根据代码内容的不同，每一个节点可能有不定数量的指针成员变量。那么当我们把下面的这段代码使用语法树来保存在内存里的时候：

```
int Fuck(IEnumerable<int> xs)
{
    int sum = 0;
    var _xs = xs.CreateEnumerator();
    while (_xs.Next())
    {
        var x = _xs.Current;
        SHIT!
        sum += x;
    }
    return sum;
}
```

就会变成这样：



这个图也是一样的，每一个框代表一个对象，左上角的箭头指向的是代表一整个函数体的语句，其中每一个蓝色的对象和每一条粗的箭头，就是我们需要遍历的所有语句的结构。这个图的意思是，从最大的{}开始，指向了四条语句，其中第三条语句是while，while又包含一个{}，里面有三条语句。这个递归的结构就是树。你要遍历树里面的所有节点，最容易的当然是递归地做深度优先搜索。

当然这里我省略了表达式的内部结构，也就是绿色的部分，毕竟我们之前定义的SHIT!这个Coroutine不处理这些内容，把他们当成一个整体就好了。

这里我们要面对的主要是一下这几类语句：

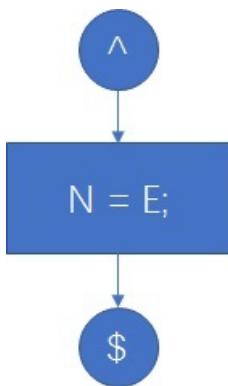
- 变量声明
- 表达式
- if
- while
- try-catch
- break / continue / return / throw

然后我们就要想办法机械化地从语法树里面得到流程图，然后按照上一篇文章的方法（从流程图得到Coroutine），那么我们就要从语法树得到Coroutine了！那么接下来我们就挨个来讲一下，每一种语句要怎么处理。

我为什么要专门说一下语法树呢？因为接下来介绍的每一种语句如何变成流程图的这个内容，实际上需要你从函数体递归地遍历进去，按顺序把每一个语句都变成流程图，然后拼接在一起。所以必然需要访问语法树。几乎所有编译器在做语法树和符号表用的都是同一个套路，所以可以举一反三（VC++的旧版本除外）。

变量声明： var N = E;

这个很简单，其实就是在实现IShitCallable<T>的成员变量里，然后把它改成普通的赋值语句：



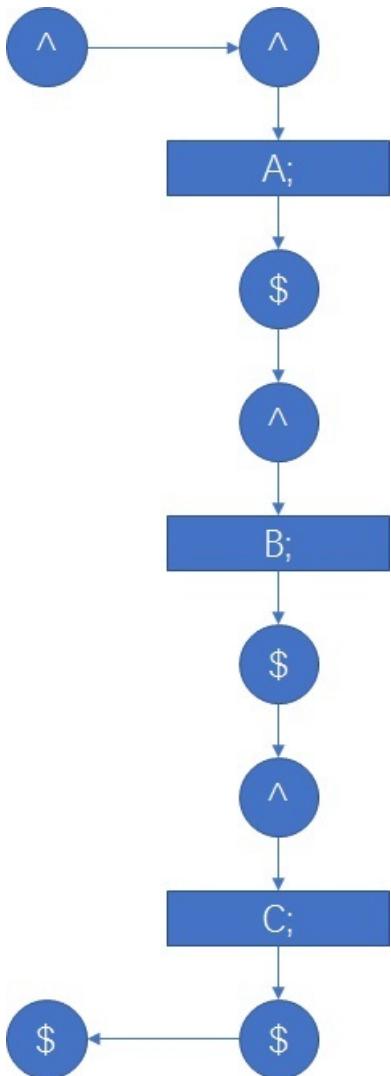
需要说明的是，^和\$分别代表开始和结束，用来跟其他语句生成的流程图连在一起，然后“优化”掉。我自己的做法是，保证只有一个流程图节点指向\$，那么你拿到的流程图，^和\$都可以指向唯一的、真实存在的节点，在这里就是“N=E;”。但是画图还是这样画容易理解。

表达式： E;

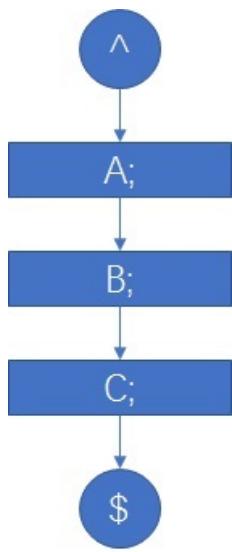
这个就简单了，原样复制，不画。

块语句： {A; B; C;}

块语句需要做的，就是分别生成A、B、C的流程图，然后把他们首尾相接起来：



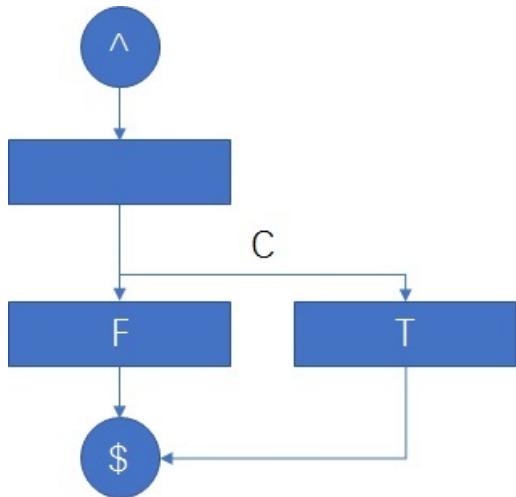
注意左边的^和\$才是属于{}自己的，右边的分别属于A;、B;和C;。在这里我们可以发现，让三条语句按顺序执行的方法，就是把各自的^和\$首尾相接。最终我们就可以把连在一起的^和\$给“内部消化”掉，然后就变成这样：



考虑到A;、B;、C;也可能是一些复杂的东西，画图的时候如果你觉得理解起来比较难，可以不辞辛苦地保留所有^和\$，然后再把他们拿掉。等下面的实战例子我就来演示一遍，到底这些图如何用在真正的代码上。

if (C) {T} else {F}

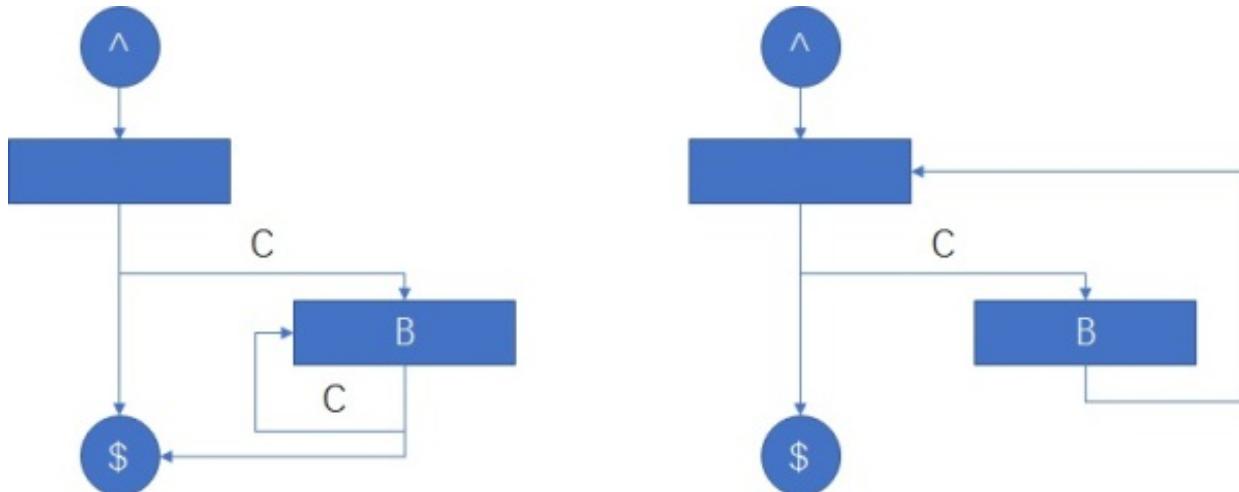
从if语句开始，就要使用分支结构了：



我这里把F和T的^和\$都省略掉了。这个图的意思很简单，我们先构造一个流程图的节点，他没有语句，只有跳转的条件。如果C成立了，跳到T。否则跳到F。T和F都连到相同的\$上面。

while (C) {B}

既然是循环，那么这些箭头自然就会连成一个圈：



循环有两种风格可供选择，我自己倾向于第一种。其实这个图的意思也很简单，一上来先到达一个没有语句的流程图节点，然后看看C是否满足。如果满足了，跳向B。B执行结束之后，又会看一下是否满足C，如果满足了，重新跳向B。不满足条件直接就跳向\$，结束循环。

break / continue / return

return就直接跳到整个函数的\$那里去（记得要把返回值写进Result属性）。而break和continue则要让你先找到到底要操作的是哪个循环，然后分别跳转到该循环的\$或者^节点。

我为什么强调说今天需要你先学会链表，因为你要解决“break的时候到底break的是哪个循环”的问题。假设说我们展开流程图的函数是这样写的：

```
Tuple<Node, Node> CreateFlowChart(Statement statement);
```

这个函数接收一条语句，然后返回^和\$。如果这个时候statement是“break;”语句，我要怎么拿到循环的^和\$节点呢？所以我们需要做一条链表：

```
class Scope
{
    public Node Begin; // ^
    public Node End; // ^
    public ScopeType Type; // enum{ Function, Loop, TryCatch, .... }
    public Scope Parent;
}
```

然后把Scope类型的对象添加到函数的参数里。这样当我们处理到了“break;”的时候，就可以沿着当前的Scope，往Parent一个一个找上去，直到发现第一个Type==Loop的Scope对象位置，然后使用包含在Scope里面的节点信息——譬如说告诉你break和continue要往哪跳。

那么我们在处理while语句的时候，自然就要往链表里面添加一个新的、代表自己的Scope对象了：

```
Tuple<Node, Node> CreateFlowChart(Statement statement, Scope scope)
{
    switch (scope)
    {
        ...
        case WhileStatement whileStatement:
        {
            var begin = new Node();
            var end = new Node();
            var whileScope = new Scope
            {
                Begin = begin,
                End = end,
                Type = Loop,
                Parent = scope
            };

            var bodyFlowChart = CreateFlowChart(whileStatement.Body, whileScope);
            begin.AddIf(whileStatement.Condition, bodyFlowChart.Item1);
            begin.AddElse(end);

            bodyFlowChart.Item2.AddIf(whileStatement.Condition, bodyFlowChart.Item1);
            bodyFlowChart.Item2.AddElse(end);

            return Tuple.Create(begin, end);
        }
        break;
    }
}
```

这个函数依次做了下面的几件事情：

- 创造begin（在这里代表的是^ 和 它指向的第一个没有语句的节点，因为的确没有必要创建两个）
- 创造end（代表\$）
- 往scope链表添加一个保存当前while语句信息的对象
- 获得循环体的流程图的^和\$节点
- 连接while的^到while的\$（条件失败）和函数体的^（条件成功）
- 连接循环体的\$到while的\$（条件失败）和函数体的^（条件成功）
- 返回while语句的^和\$

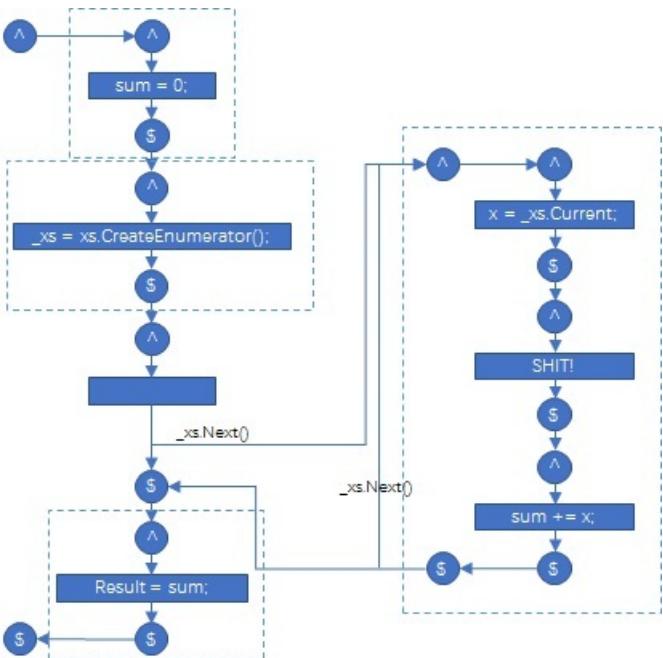
索；在需要的时候往Scope上面加节点，这就是链表；最后Statement类型和他所有的子类加在一起是一棵树。其实无论你写多么复杂的程度，你会发现大部分只需要超级简单的知识就可以了。

实战

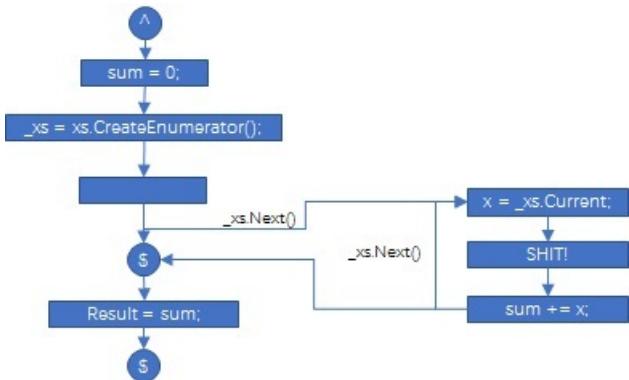
让我们来人肉练习一下怎么把完整的语法树转换成流程图吧！我们还是使用我们熟悉的函数：

```
int Fuck(IEnumerable<int> xs)
{
    int sum = 0;
    var _xs = xs.CreateEnumerator();
    while (_xs.Next())
    {
        var x = _xs.Current;
        SHIT!
        sum += x;
    }
    return sum;
}
```

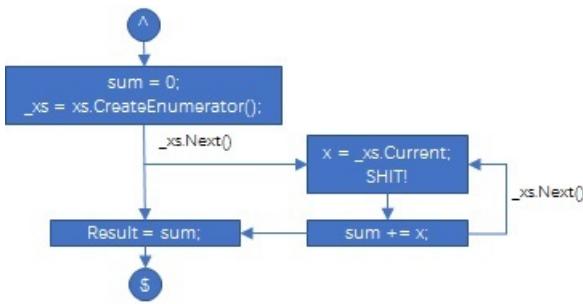
这里我就不再展开递归了，我选择保留所有的^和\$节点，你们去对照上面的图，就可以发现优化前的流程图的结构跟语法树的结构其实是完全一致的。



大家有没有发现，每次我们从一个语句或者复合语句做出流程图的时候，我们总是保证这个流程图有一个明显的开始节点和结束节点？这其实是为了实现的时候方便。一开始我们先生成粗糙的流程图，让后再优化一下减少明显不必要的跳转（也就是说不明显的不管他就可以了！）。通常来讲，我们可以先去掉多余的^和\$节点：



然后再把那些能合并的节点合并了（还记得SHIT!只能出现在节点的末尾的要求吗？）：

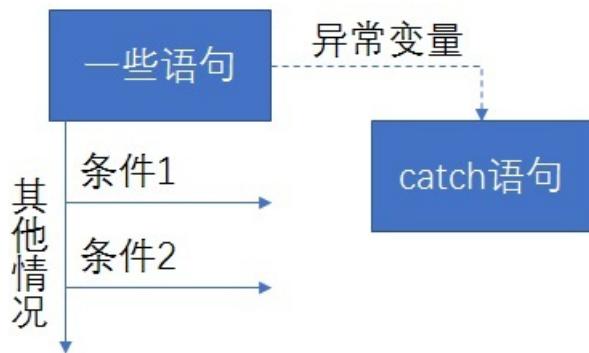


这样就得到上一篇文章我们画出来的流程图了！

作业：总结出一个规律，可以被去掉和合并的节点都有什么特征？

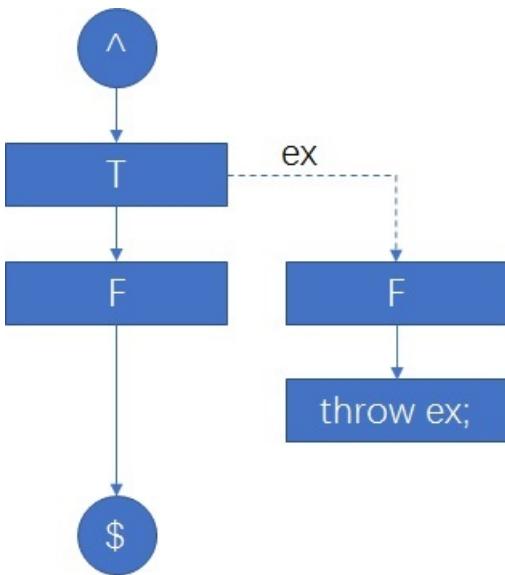
那么接下来就轮到try-catch了。在实现Coroutine的过程中，try-catch简直就是恶魔，C#在刚开始的时候甚至不支持你在try-catch里面用yield return和await，可能是来不及测试完（逃

在讨论具体的流程图之前，我们先要对流程图做一个小修改，就是要加上一个新的虚线箭头，代表这个方框里面的代码如果出现了异常，那么就跳转到另一个流程图的节点里面去：



异常变量的意思就是说，因为根据实际情况我们可能不能在生成的try-catch语句里面，套上“catch语句”里面的内容，所以在这种情况下，先把异常保存到成员变量里面，然后跳走。现在我们可以开始来处理try-catch语句了：

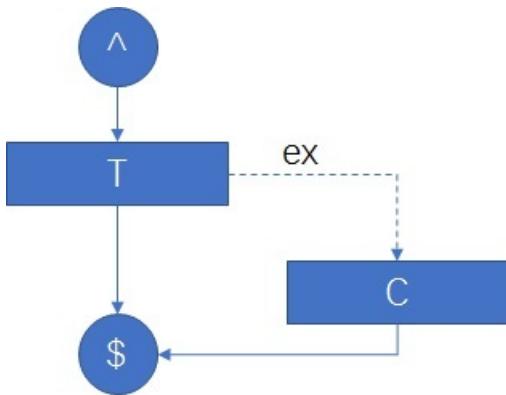
try {T} finally {F}



虽然看起来比较复杂，其实意思很简单。我们先执行T，如果没有出现任何状况，就执行F然后结束。如果出现了状况了，先执行F，执行完了重新把异常抛出来。虽然这样做可能会丢失当时的一些堆栈信息，不过你都弄成Coroutine了，顾不了那么多了（逃

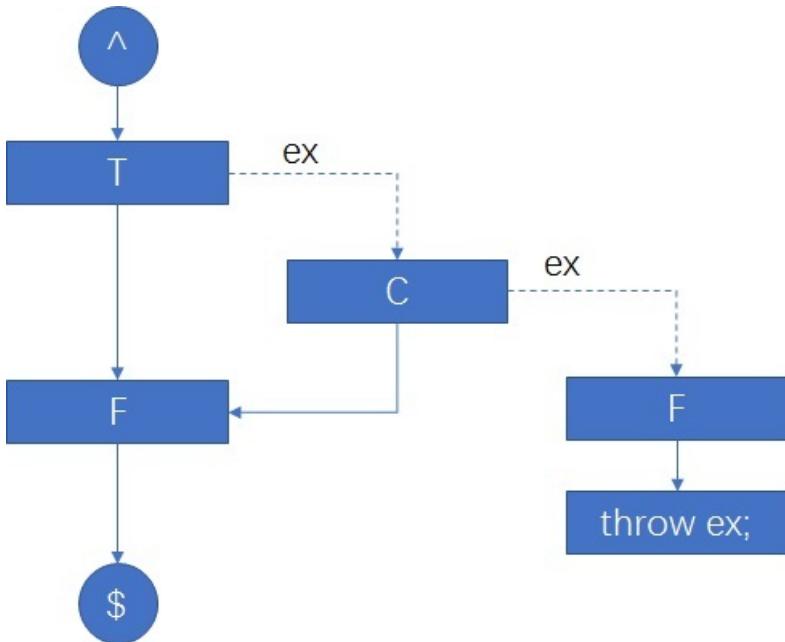
在这里，throw ex;并没有后继的代码了。因为如果这一片东西也被包含在另一个T里面的话，那么throw ex;那里将有一条新的虚线，执行到throw ex;就会跳转过去。如果没有的话，整个函数就这么结束了。但是这个时候你可能需要将Coroutine接口标记上一个结束状态，所以你在**while (true)**循环外面还要套一层try-catch，来做这个事情。

try {T} catch {C}



这个没啥好说的。如果语言里面像C++和C#一样支持对异常的类型做分支的话，这里的--(ex)-->C要变成很多条。

try {T} catch {C} finally {F}



这里我觉得需要解释一下。首先我们执行T，如果没有出现问题，那么就执行F然后结束。如果出了异常，我们就执行C，然后执行F，然后结束。如果在C，也就是catch语句里面的时候，又出现了一个新的异常，那我们就先把这个新的异常留下来，执行F，重新抛出去。

注意：代码改成流程图的时候，可能一些不同的变量会撞名，需要酌情重命名。这个时候需要访问符号表。

一旦我们有了finally的部分，break、continue和return也要做相应的改动。break、continue和return都会从当前的块里面跳出去，这一跳可能会跨若干个Scope（上面说的Scope对象），譬如说在一个循环里面嵌套一个try-catch，然后再在里面跳出循环。这个时候你要依次执行所有被你跳过的finally语句，然后再走。这个过程很简单，访问Scope.Parent到循环之前，遇到的所有Type==TryCatch的Scope，你看着办就好了。需要注意的是，不同的finally语句里出现的异常可能会要求你跳转到不同的catch里面去。

实战

大家看到这么复杂的流程图不要怕，我们来实际做一下就明白了。先看看这段代码：

```

int Fuck(IEnumerable<int> xs)
{
    int sum = 0;
    var _xs = xs.CreateEnumerator();
    try
    {
        while (_xs.Next())
        {
            var x = _xs.Current;
            SHIT!
            sum += x;
            if (sum > 10) break;
        }
    }
}

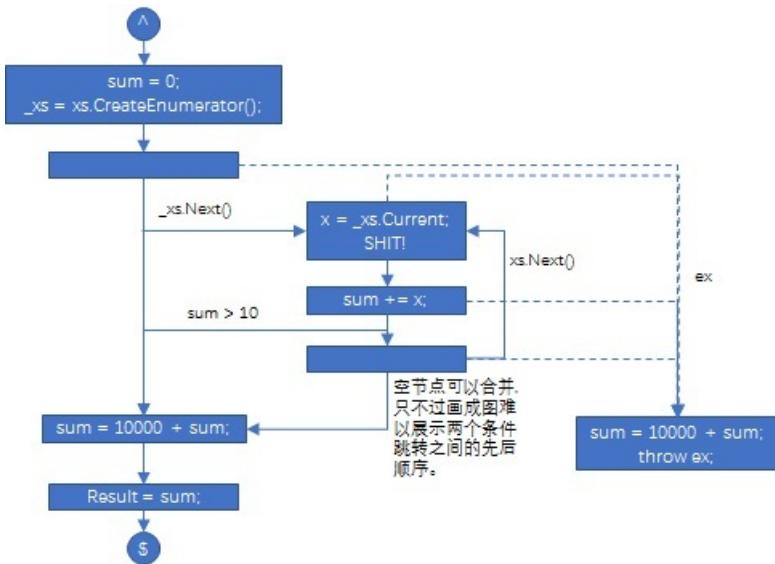
```

```

finally
{
    sum = 10000 + sum;
}
return sum;
}

```

大家可以尝试自己在纸上画一遍，最后得到的结果应该跟下面这个图是差不了多少的。如果要验证的话，可以人肉把机械化生成出来的代码打进Visual Studio的C#工程里面，立刻就知道结果了！在这里需要提醒大家，try-catch里面我很轻松地就从T那里引了一些虚线出来。如果T是一个复杂的流程图，那么在里面所有的节点，都要引那一条虚线出来，直到被新的try-catch语句的虚线覆盖为止。因此最后的结果可能看起来会比较混乱：



你们都做出来了吗？

到这里，如何把SHIT_CALLABLE!函数转成Coroutine的内容就介绍完了。但是这个系列还没完，因为后面还有yield return和await要讲。毕竟光给你这几个关键字，要写出真的程序来，也很别扭。所以下一篇文章开始，会介绍如何把各种各样奇怪的功能，一个一个映射到SHIT_CALLABLE!和SHIT!上面去。这样一来，当你真正实现这些功能的时候，只需要把他们统一改写为Coroutine的形式，再编译成普通的代码就好了，节省了大量的时间。

只要有扎实的编码技巧，哪怕你考不上三本，其实都没有问题，绝大多数的程序都是写得出来的！剩下的就是去背诵架构上的套路。

从硬盘里翻出来2010年写的带IntelliSense编辑器

记得很多年前也想做一个跟LLVM差不多的项目，就是这一次尝试让我意识到了，一个人的力量是有限的，这种东西一个人要写到80岁。于是写GacUI的时候才能那么淡定（逃。但是这并没有让我浪费时间，因为后来我还是完成了一次IntelliSense的尝试，处理一个内核是C的语言，但是我给他加上了type class，于是变成了一个怪胎。

这里的做法跟前几天Channel 9上面看到 [Anders讲Roslyn如何支持VS编辑器的视频](#)（感谢 [@RednaxelaFX](#) 去翻，这个视频让我学到了helicopter还能活用为那种意义的动词），核心方法是完全一致的。大概的意思就是说，整个东西是跑在后台的，但是你前台需要智能地识别出，到底要如何取一小块正在编辑的代码，才可以在UI线程里局部地更新语法树，然后用（有1%可能是错误的）之前留下来的语义分析的结果来立刻弹出IntelliSense的内容。同时后台也在运行，全文分析完之后，如果你的字还没打完，然后发现了列表的内容很不幸处于1%的错误的时候，就立刻更新列表。整个编辑的过程非常流畅。

当时还实现了鼠标指名字把声明弹成Tooltip，函数调用弹Tooltip，还实现了Visual Studio里面按‘For[TAB]’之后出现的模板式代码编辑。

每次我都在想，为什么需要一些高级技巧的时候，都会找不到资料，等到一个人折腾出来了，才发现dalao们终于肯在网上上课。

这个程序是C#写的（包括那个带错误恢复的parser），使用了当年火爆的Ribbon，无奈Windows Forms里面并没有控件，于是我想素级copy了Word2007的效果。如果是Windows 7的话，上面的标题栏还是半透明的，工具栏的背后还会有点阴影。为什么叫Turtle IDE呢，因为这其实是一个小海龟程序，写完代码之后就可以运行，看到一只海龟在画图（逃

那会儿还在cppblog发了[开发日志](#)一共十几篇，几年后回头看发现其实也没讲清楚。原本我已经把相同的算法做进了GacUI的项目里面，不过其实还是有点残废，无法像Visual Studio一样处理嵌套语言（主要是ParserGen的锅），而这又是GacUI的必备内容。

根据现在的进度，今年内GacStudio十分有希望可以开工，到时候也要实现IntelliSense。这次我准备把语言服务的内容都做进独立的进程里面，毕竟这才是现代的做法。那个时候我会做一个完整的工具箱，让大家也可以使用GacUI来开发属于自己的编辑器，顺便给《三本》系列添加新的内容。别看IntelliSense用的时候酷炫吊炸天，其实也没什么知识是难以学会的。

最近在给Workflow加上“自己写库提供yield return和await”的支持，写完了之后我就继续发《三本Coroutine（四）》，把这部分的内容包含进去。

考不上三本也能给自己心爱的语言加上Coroutine（四）

你现在所阅读的并不是第一篇文章，你可能想看 [目录和前言](#)。

这是《三本》的Coroutine系列的最后一篇了。虽然《三本》以后还会有，但是Coroutine从此就没有了。今天要说的是，如何把yield return和await这样的东西编译成Coroutine。如果你想把这两个语法写死在语言里面的话，过程是相当粗暴的。但是我现在要说的是 [GacUI的Workflow脚本语言](#) 是如何做的。

在做这种事情的时候，你先要在纸上不断地尝试各种不同的语法，最后找到一个最有感觉的定下来。这次我给脚本加上Coroutine的功能的时候，选择了这样的语法：

```
func GetNumbers() : int{}  
{$  
    for(i in range[0, 4])  
    {  
        $Yield i;  
    }  
}
```

熟悉C#的人应该相当容易理解。"\${}"这两个符号之间其实省略了一个名字，这个名字指向了一个Coroutine的Provider，这个Provider就是具体实现\$Yield等功能的一个类。我这里还提供了\$Join，让你直接yield一个集合出去，不用跟C#一样要写个for循环里面慢慢yield return。

大部分语言都是使用偏特化（或者pattern matching，或者type class，随你喜欢怎么叫）来让语言找到这个Provider的。但是Workflow作为一门专门用来跟C++交互的语言，对象模型必然就跟COM差不多，那么也就没有什么模板。C++/CX表面看起来好像支持泛型，实际上你去找生成的IDL文件用MIDL的CX版本去编译，会有惊喜（逃

没有模板的语言，要支持Provider的查找，自然有自己的办法。我这里选择了使用类型的名字去匹配。既然在这里我们把"\$Enumerable{"省去变成了"\${}", 那么只能从函数的返回值名字入手了。int{}这个类型其实是不存在的，它的全称是system:Enumerable<int>^。而Workflow里面又没有泛型，所以system:Enumerable<int>^实际上就是system:Enumerable^（^就是v1:Ptr，跟shared_ptr是同一个意思）。那么在这里我就会去枚举这个类型自己和所有基类的名字，按顺序列出来：

- system:Enumerable
- system:Interface

然后加上一个后缀

- system:EnumerableCoroutine
- system:InterfaceCoroutine

然后看看第几个名字最先存在。如果你写的是"\$Enumerable{"的话，我会在当前的上下文里查找Enumerable类型，如果存在的話就试试看加上Coroutine后缀，如果失败了就直接找EnumerableCoroutine。

那么显而易见的，这次的Coroutine Provider就是一个叫做system:EnumerableCoroutine的类，这个类实际上是在C++里面的。作为一个专门跟C++对象交互的脚本，当然可以随便调用C++的东西啊。在[这个文件](#)的62行，我们可以发现这个类真实的身份是v1:reflection:description:EnumerableCoroutine，它的声明和实现分别在[GuiTypeDescriptorPredefined.h:234](#)和[GuiTypeDescriptorPredefined.cpp:137](#)。

背景就先讲到这里。那么到底要如何去定义EnumerableCoroutine类呢？我们先来思考一下，如果要直接使用Coroutine来实现一个返回0到4的IEnumerable<T>，那么到底要怎么做？在这里先简单介绍一下Workflow里面Coroutine接口的样子：

```
enum class CoroutineStatus  
{  
    Waiting,  
    Executing,  
    Stopped,  
};  
  
class CoroutineResult : public virtual IDescriptable, public Description<CoroutineResult>  
{  
protected:  
    Value result;  
    Ptr<IValueException> failure;  
  
public:  
    Value GetResult();  
    void SetResult(const Value& value);  
    Ptr<IValueException> GetFailure();
```

```

void SetFailure(Ptr<IValueException> value);
};

class ICoroutine : public virtual IDescriptable, public Description<ICoroutine>
{
public:
    virtual void Resume(bool raiseException, Ptr<CoroutineResult> output) = 0;
    virtual Ptr<IValueException> GetFailure() = 0;
    virtual CoroutineStatus GetStatus() = 0;
};

```

这跟之前几篇文章的IShitable类其实是差不多的，只是这里多了一些细节。Resume的参数先别管，实际上ICoroutine类只有Resume函数是重要的，你一直调用它，直到返回true为止。

我们先尝试一下用Workflow的Coroutine语法来表达循环输出0到4：（在这里\$coroutine就等于SHIT_CALLABLE!，而\$pause就等于SHIT!，区别只是\$coroutine是一个lambda表达式）

```

$coroutine
{
    for(i in range[0, 4])
    {
        $pause
        {
            DO_SOMETHING_WITH(i);
        }
    }
}

```

这个表达式就返回一个system:Coroutine^接口的实例，就是上面的ICoroutine。然后我们就可以来着手写一个Enumerable接口的实现了。我这里的Enumerable接口跟C#完全一致，所以大家不需要学习也应该知道如何做（这里的object相当于COM的Variant类型）：

```

func GetNumbers() : int{}
{

return cast int{} new Enumerable^
{
    override func CreateEnumerator() : Enumerator^
    {
        return new Enumerator^
        {
            var coroutine : Coroutine^ = null;
            var current : object = null;
            var index = -1;

            override func GetCurrent() : object { return current; }
            override func GetIndex() : int { return index; }

            override func Next() : bool
            {
                if (coroutine is null)
                {
                    coroutine = $coroutine
                    {
                        for(i in range[0, 4])
                        {
                            $pause
                            {
                                DO_SOMETHING_WITH(i);
                            }
                        }
                    };
                }
                if (coroutine.Status == Stopped) return false;
                if (!coroutine.Resume(true, null)) return false;
                return true;
            }
        }
    }
};
}

```

显而易见地，这个DO_SOMETHING_WITH(i)应该去修改current的值，让它等于i，然后index自增。\$pause的这个大括号其实意思是说，在设置了Status==Waiting并且写好下一个跳转的目的，到暂停之间，你要干一些什么事情。对于Enumerable来说，DO_SOMETHING_WITH在\$pause之前做还是在\$pause里面做都是一样的，因为毕竟是一个单线程的东西。所以大家也不要纠结这个。

于是我们就得到了这样的代码：

```
$pause
{
    current = i;
    index = index + 1;
}

好了，我们这个Enumerable已经完成了。现在要开始来想，如果我们要把$coroutine语句拿到这个类的外面去，那要怎么办？显然$pause里面的current和index，就要把this指针替换成一个名字，代码我们先拆开一半：

interface MyClosureEnumerator : Enumerator
{
    func OnYield(value : object) : void;
}

func CreateCoroutine(impl : MyClosureEnumerator*) : Coroutine^
{
    return $coroutine
    {
        for(i in range[0, 4])
        {
            $pause
            {
                impl.OnYield(i);
            }
        }
    };
}

func GetNumbers() : int{}
{

return cast int{} new Enumerable^
{
    override func CreateEnumerator() : Enumerator^
    {
        return new MyClosureEnumerator^
        {
            ....
            override func Next() : bool
            {
                if (coroutine is null)
                {
                    coroutine = CreateCoroutine(this);
                }
                ....
            }
        };
    }
};

}

这两份代码是完全等价的。现在看起来已经有点$Yield的样子了。但是还有一个问题没有解决。就是显然对于所有想要产生Enumerable^的Coroutine来说，那一大段new Enumerable^的代码肯定是相同的，而不同的内容有不同的CreateCoroutine函数，那到底要怎样才能把依赖翻转过来呢？实际上针对这个具体的例子，事情是相当的简单，只要把CreateCoroutine变成一个参数就可以了：
```

```
interface MyClosureEnumerator : Enumerator
{
    func OnYield(value : object) : void;
}

func CreateCoroutine(impl : MyClosureEnumerator*) : Coroutine^
{
    return $coroutine
    {
        for(i in range[0, 4])
        {
            $pause
            {
                impl.OnYield(impl, i);
            }
        }
    };
}

func CreateEnumerable(creator : func(MyClosureEnumerator*:Coroutine^) : Enumerable^
```

```

{
    // 里面对CreateCoroutine的调用就变成了对creator的调用
    return new Enumerable^ { ... };
}

func GetNumbers() : int{}
{
    return cast int{} CreateEnumerable(CreateCoroutine);
}

```

但是这仍然不足以让我们实现完全的自动化。一个最终的目标，就是用户提供一个EnumerableCoroutine类，然后我们把Coroutine编译成对EnumerableCoroutine的调用。那么这个EnumerableCoroutine类里面到底要有什么东西呢？其实到这里应该很清楚了：

- CreateEnumerable
- impl.OnYield(i);
- interface MyClosureEnumerator*

这样我们就可以尝试整理一下：

```

class EnumerableCoroutine
{
    interface IImpl : Enumerator
    {
        func OnYield(value : object) : void;
    }

    static func Yield(impl : IImpl*, value : object) : void
    {
        impl.OnYield(value);
    }

    static func Create(creator : func(MyClosureEnumerator*) : Coroutine^) : Enumerable^
    {
        return new Enumerable^ { ... };
    }
}

func GetNumbers() : int{}
{
    return cast int{} EnumerableCoroutine.Create(
        func(impl : IImpl*) : Coroutine^
        {
            return $coroutine
            {
                for(i in range[0, 4])
                {
                    $pause
                    {
                        EnumerableCoroutine.Yield(impl, i);
                    }
                }
            };
        }
    );
}

```

现在应该很清楚了。这已经是一个带EnumerableCoroutine的完整的程序了，对比一下一开始的代码：

```

func GetNumbers() : int{}
${
    for(i in range[0, 4])
    {
        $Yield i;
    }
}

```

实际上编译器在看到这个代码之后，要做的就是：

- 通过规则找到EnumerableCoroutine，在这里我们叫P
- \$Yield i; 翻译成 \$pause { P.Yield(impl, i); }
- 整个函数翻译成 { return P.Create(func(impl : <从P获得的impl类型>) : ICORoutine^ { return \$coroutine { ... }; }); }

当然光做到这里是不行的，我们还有一些细节要做，譬如说return怎么办，譬如说\$Await要获取返回值怎么办，虽然这些都是问题，但已经不是什么大问题了，使用类似的手段就可以解决。

不知道大家还记得Resume的第二个参数CoroutineStatus^，其实这就是用来获取上一次\$pause之后的结果的，譬如说\$Await的

返回值。\$Await函数本身当然是没有返回值的，因为事情没做完要先\$pause，返回值肯定以后才能给。因此你就想办法在下次调用Resume的时候传进来就可以了。这里加一个参数就很合适。

在实现EnumerableCoroutine的同一对文件里面，就有IAsync和AsyncCoroutine的实现。这里的AsyncCoroutine实际上只是个壳，而真正的事情是IAsyncScheduler做的。每一个线程有自己的一个scheduler，一个Async（C#的Task）在哪个线程启动，就回去拿到哪个线程的scheduler。到时候具体的实现当然是GacUI提供的，而单元测试里面的scheduler显然不可能真的去跑多线程，不然就没有一个固定的调用顺序可以测量了。

\$Await就一定要放在\$pause里面，因为当\$Await没执行完的时候，可能被\$Await的Async对象已经瞬间跑完要执行continuation了，这个时候如果我们没有先设置好暂停状态，那么他就会读到Executing，然后GG。至于为什么AsyncCoroutine要那样写，就做为我布置给你们的一个作业，你们自己去看。

Coroutine到这里就圆满结束了，如果你们有什么还看不懂，害怕自己不是一个合格的三本学生的话，可以来留言，如果东西多我就再写一篇。接下来我将会继续GacUI的开发，然后抽空写《三本》系列的下一部：《考不上三本也能实现数据绑定》。不要觉得现在XAML也好，各种离谱的前端UI框架也好，数据绑定是一个神奇的东西，这当然不是！实现数据绑定所需要的所有知识里面，并没有什么是考不上三本就一定学不会的！

当然了，这一篇肯定不会跟Coroutine一样过几天就发布了。我先继续折腾GacUI，应该也不需要等太久，总之你们先慢慢等（逃

考不上三本也会实现数据绑定（一）

你现在所阅读的并不是第一篇文章，你可能想看 [目录和前言](#)。

前言

经历了GacUI的一次超级大重构之后，终于又有空写文章了。这次重构在保留了所有功能的前提下，删掉了很多类跟接口，积累了很多魂来传火，过些日子再围绕相关的话题来谈一谈重构。今天先说一下数据绑定的事情。

MVVM

在我讲数据绑定的时候，背景是设定在 [MVVM模式](#) 上面的，在这里简单介绍一下MVVM。MVVM把我们的程序——其实大多数时候就是UI或者网页——分成了三大块，分别是Model、ViewModel和View。Model很好理解，指的是数据源，通常是数据库或者文件。View也很好理解，就是最终的UI，或者是测试用例，或者是一些别的东西。那ViewModel是什么呢？大多数时候我们也不用死抠概念。ViewModel出现的根本原因就是，Model和View的结构经常是不一样的。

举个简单的例子，Model是每门科目各一份的成绩单，View是教室门口贴的总分前10名的大红榜。显然Model和View的结构就是完全不同的。在这里Model按照科目来分，告诉你每个人的成绩是多少。View则告诉你总分前10名都是什么人。那ViewModel是啥呢？其实ViewModel在这里就是红榜的内容了。View——红，ViewModel——榜，没毛病（逃

因此ViewModel大多数时候就是用来负责实现具体的计算过程的，譬如从一堆成绩单里面算出总分前10名，这就是ViewModel要做的事情。那么针对这份ViewModel，我们就可以做一堆View，譬如说：

- 让老师把前10名念出来
- 写进大红榜贴在教室门口
- 打印成文件给校长用来做发奖学金的参考

这三个View虽然长的完全不同，但是其内容都是一致的，而且的格式跟Model有巨大的区别。所以这就是ViewModel存在的原因，你需要一些复杂的计算，而且这些计算可以被重复使用，所以要独立出来，正确处理好依赖：让View去依赖ViewModel。

数据绑定

好了，那为什么会有数据绑定呢？因为到了这一步，ViewModel的格式跟View其实就差不多了，那么你把数据从ViewModel复制到View，或者从View反馈回ViewModel，理论上只需要一些简单的步骤就可以实现。UWP就假设这些步骤可以简单到你只需要给一串（通常只有一个）属性的名字就好了，复杂点的可以用Converter，再复杂就证明你的ViewModel做的不好。

GacUI的数据绑定并没有这个假设，我可以让你写无限复杂的表达式，不过同时也付出了没有双向绑定的代价——如果你需要双向绑定，那你就正反两边都绑定好了——因为GacUI的数据绑定除了联系View和ViewModel以外，还有别的事情要做。

UWP的xbind提供了三种绑定的形式，分别是一次性绑定、单向绑定跟双向绑定。其中一次性绑定就跟写在构造函数里没有区别，而双向绑定可以理解为从一个单向绑定计算出相反的单向绑定要怎么写然后自动替你写好，因此我们就只需要考虑单向绑定要怎么做就可以了。

在这里不得不提到的是，我看到有些JavaScript的界面库在实现数据绑定的时候，采用了一些花式作死的方法，其中我印象最深刻的一种就是，他先跑一下你的表达式，看看一共用到了多少属性，然后挨个给他们挂上事件处理程序。居然存在使用采样的方式来实现的编译器，不得不佩服作者的想象力实在是太强大了，是谁这么做我已经不记得了。不过他根本不需要这么做，只要把JavaScript换成TypeScript，这个事情就可以完美的解决。

为什么呢？因为你既然要做数据绑定，那你总要知道你写的这个表达式到底需要响应多少事件——其实也就是说，到底要在ViewModel的什么部分被修改的情况下刷新View——这样才能做出完整的功能。而JavaScript这个语言在阅读的时候是没有上下文的，所以你根本没办法做这样的计算，因此才需要用采样的方法。而采样的一个缺点就是，万一你的表达式有分支怎么办？有些人可能会说，你不应该写分支。其实这个说法很对，毕竟这么复杂的逻辑应该放进ViewModel里，但是与此配套的，你应该在遇到分支的时候直接爆炸，而不是就这么默默的接受了。TypeScript就是有上下文的，所以可以清楚地把依赖关系都计算出来。

所以这个系列的前提就这么确定了，我们在MVVM的模式下，使用一个强型的语言做数据绑定。那我们需要依靠什么三本的知识来实现数据绑定，就是接下来的文章要讲述的事情。当然说是说系列，多半（二）就会把所有的事情都说完了，毕竟数据绑定的内容很少（逃

一个例子

在文章结束之前，最后讲一下我们要如何对数据绑定这个功能进行建模。在GacUI使用的 [Workflow脚本语言](#) 里面，我给了一个

bind表达式，具体的用法就是：

```
bind(obj.A + obj.B)
```

这个表达式会返回下面的接口的实例：

```
interface system::Subscription
{
    /* 一些无关紧要的其他函数 */

    event ValueChanged(object);
}
```

然后只要obj.A + obj.B这个表达式改变了——其实也就是A或者B属性的其中一个发生了变化，ValueChanged事件就会触发，参数就是这个表达式当前的值。你只要挂了这个事件，自然就可以把最新的数据显示到UI上面了。譬如说下面的这个[测试用例](#)：

```
module test;
using test::*;
using system::*;

var s = "";

func Callback(value : object) : void
{
    s = $$$(s)[$(cast string value)];
}

func main() : string
{
    var x = new ObservableValue^();
    var subscription = bind($"The value has been changed to $(x.Value)");
    subscription.Open();
    attach(subscription.ValueChanged, Callback);

    x.Value = 10;
    x.Value = 20;
    x.Value = 30;

    subscription.Close();
    return s;
}
```

最终就会返回下面这个字符串：

```
[The value has been changed to 10][The value has been changed to 20][The value has been changed to 30]
```

因为x.Value一共改变了3次。

待续

今天这篇文章就说到这里了，接下来我们会重点描述下面的三个问题

- 如何跟踪属性变化
- 如何分析属性之间的依赖关系
- 如何把数据绑定重写为对回调函数的调用

这三个问题搞定了，数据绑定也就做出来了。敬请期待。

考不上三本也会实现数据绑定（二）

你现在所阅读的并不是第一篇文章，你可能想看 [目录和前言](#)。

我想了想还是拆开写。今天只讲一种简单的表达式的形状，也就是XAML用的属性链。属性链指的就是a.b.c这样的表达式，同时你也可以带有下标，但是下标也只能是属性链。不过在大多数的实现里面，像a[b.c]这样的表达式，只会监视b.c有没有变化，而不会在b.c固定的情况下监视a[x]有没有变化，所以实际上跟只没有下标处理起来是差不多的。

例子

让我们来看这样的一个例子，我们要把语句：

```
attach(bind(a.b.c.d).ValueChanged, callback);
```

变成一个触发callback的事件，从而你可以监听他的变化，那到底要处理多少东西呢？如果我们假设每一个属性都有对应的变化事件，那么其实就等于监听

- a.bChanged
- a.b.cChanged
- a.b.c.dChanged

这几个事件。同时我们也做出下面的假设：

- a不会变
- 多次读取一个属性只会返回相同的值，也就是这个表达式的副作用将被忽略

而且a.b变化了，那么原先我们监听的a.b.cChanged就没有用了，我们要解绑它，重新监听一个新的。通常来讲，解绑一个事件，要么是通过直接操作监听事件返回的cookie（如.net的IObservable），要么是直接把cookie交给被监听对象的属性来完成。如果我们取的是后者，那么我们还得把旧的a.b留下来，然后才能正确的解绑事件。你要不这么做的话，那么我们就会一直在试图监听旧的a.b的cChanged事件，那么新的a.b.c再怎么变化你也不会接到通知。

这就意味着，如果我们监听一个对象o的属性p，那么我们就得把o给cache下来，当o变了的时候，我们才能正确的解绑事件pChanged，然后挂一个新的callback到新的o.pChanged上面去。因此对于表达式a.b.c.d，首先假设a不会变，那么我们要cache下来的就是a.b和a.b.c了。

代码生成

所以我们大概会生成这样的代码：

```
class MyDataBinding : Subscription
{
    var cache_a : A^ = null;
    var cache_a_b : B^ = null;
    var cache_a_b_c : C^ = null;
    var handler_a_b : EventHandler^;
    var handler_a_b_c : EventHandler^;
    var handler_a_b_c_d : EventHandler^;

    ctor(a : A)
    {
        // 给a.bChanged上绑
        cache_a = a;
        handler_a_b = attach(cache_a.bChanged, OnChanged_a_b);
        // 绑上剩下的所有事件
        Update_a_b(cache_a.b);
    }

    func Update_a_b(b : B) : void
    {
        // 如果a.b变了，那么要正确处理a.b.cChanged
        if (cache_a_b == b) return;
        // 把旧的a.b.cChanged解绑
        if (handler_a_b_c is not null)
        {
            detach(cache_a_b.cChanged, handler_a_b_c);
        }
        // 把新的a.b.cChanged上绑
        cache_a_b = b;
    }
}
```

```

    handler_a_b_c = attach(cache_a_b.cChanged, OnChanged_a_b_c);
    // 既然a.b变了，那么a.b.c多半也变了
    OnChanged_a_b_c();
}

func Update_a_b_c(c : C) : void
{
    // 如果a.b.c变了，那么要正确处理a.b.c.dChanged
    if (cache_a_b_c == c) return;
    // 把旧的a.b.c.dChanged解绑
    if (handler_a_b_c_d is not null)
    {
        detach(cache_a_b_c.dChanged, handler_a_b_c_d);
    }
    // 把新的a.b.c.dChanged上绑
    cache_a_b_c = c;
    handler_a_b_c_d = attach(cache_a_b_c.dChanged, OnChanged_a_b_c_d);
    // 既然a.b.c变了，那么a.b.c.d多半也变了
    OnChanged_a_b_c_d();
}

func OnChanged_a_b() : void
{
    Update_a_b(cache_a_b);
}

func OnChanged_a_b_c() : void
{
    Update_a_b_c(cache_a_b.c);
}

func OnChanged_a_b_c_d() : void
{
    // 触发ValueChanged事件，调用callback
    ValueChanged(cache_a_b_c.d);
}
}

```

然后原来的那段代码就变成了：

```

var myDataBinding = new MyDataBinding^(a);
attach(myDataBinding.ValueChanged, callback);

```

计算过程

如果bind(a.b.c.d)这样实现的话，那么其实可以看到，如果a.bChanged触发了，那么旧的a.b.cChanged和a.b.c.dChanged都会被解开，然后OnChanged系列函数就挂到了新的a.b.cChanged和a.b.c.dChanged上面去。所以不管是b、c、d这三个属性哪一个变了，ValueChanged事件最终都会触发。

上面这个代码有一个瑕疵，就是构造MyDataBinding的时候，ValueChanged会被触发一次。不过这种小事只要不影响profiling，都不用管。生成代码的算法尽量以简单为好。

生成MyDataBinding的策略很简单，因为我们涉及的属性有

- a.b
- a.b.c
- a.b.c.d

这三个，因此就有以下三个步骤：

- 对于a.b
 - cache一下a，然后挂上a.bChanged
 - 生成ctor
- 对于a.b.c和a.b.c.d
 - cache一下a.b，然后挂上a.b.cChanged
 - 生成Update_a_b和OnChanged_a_b
 - cache一下a.b.c，然后挂上a.b.c.dChanged
 - 生成Update_a_b_c和OnChanged_a_b_c
- 触发ValueChanged事件
 - 生成OnChanged_a_b_c_d

对于`a[b.c]`这样的表达式，道理也是一样的，就是分别给下标表达式生成`MyDataBinding`这样的类，最后把它们组合起来，这是一个递归地步骤，留给大家课后练习一下。今天就说到这里了，下一篇就会讲到处理复杂表达式的方法。

其实在这里，`a.b`的变化会导致`a.b.cChanged`重挂的事情，就是一个`a.b.c`对`a.b`的依赖关系。在复杂的表达式里面，就会有各种各样的依赖关系，它们共同组成了一个偏序的关系。举个简单的例子：

```
let a = x.y in (a.b + a.c)
```

`a.b`和`a.c`就共同依赖于`a`，也就是`x.y`，因此`x.yChanged`的事件处理函数里面，就要把`a.bChanged`和`a.cChanged`都进行重挂。这样的依赖关系当然是不会有环的。

通常来讲，做出这样的一个关系图倒是不难，难的是当你的表达式类型特别多的时候，你做的这些类似把`a.b.c`替换成`cache_a_b_c`和`cache_a_b.c`这样的工作，就容易出bug。这些还得靠你多写几个测试用例来保证。

后注

UWP的`{x:Bind}`比起`{Binding}`先进的地方在于，`Binding`需要运行时的反射，而`x:Bind`是不需要的。但是XAML的编译器在处理`x:Bind`的时候，怎么知道`a`、`a.b`、`a.b.c`和`a.b.c.d`的类型，从而生成出正确的代码呢？这通常有两个办法：

- 你可以想办法让XAML编译器知道他们的类型。对于`x:Bind`，这是通过`cl`、`midl`、`midmerge`和`xamlcompiler`这几个程序共同实现的。其中的辛酸，只有手写`idl`文件的人才能明白（逃。这种方法不仅可以支持C++，也可以支持WinRT上面的任何编程语言。
- 如果是想支持C++的话，其实你完全可以通过模板的技巧来计算出所有的这些东西，只要你使用同样的语法来在所有可以绑定的对象身上读取属性和挂事件就可以。`x:Bind`的灵感其实是从Office借去的（包括以前的XAML），而Office又怎么有办法修改`c`呢？所以内部的工具就通过生成模板类和函数的方法来解决。

这些AI都不给人类点面子

如图，真敷衍。

[张远远：C++ #include " " 与 <>有什么区别？](#)

知乎程序员初学者组，在今天成立啦！

当然功能主要是促进组里的、譬如说我和R大这样的

新手

互相地、广泛地交流（聚餐）。目前只有[@RednaxelaFX](#)响应，所以组里就只有三个人（含AI）。本组不设置管理层，在人少的时候，没有任何组织形式，人多了说不定开个微信群什么的。

为了控制人数，我有一个简单的规则。除了我邀请（含他人推荐被批准）的人以外，其他人要加入的方法，就是请 $\max(3, \text{全组一半})$ 组员吃饭，即可成为程序员初学者。由于组员分布世界各地，不要求一顿把所有人请完（逃

回国新体验

整个7月份基本都在国内，因为积累了太多假期，不知道干嘛，于是回国吃吃吃。但是发现广州居然没有西雅图直飞，于是只好多去几个城市。

我选择了深圳作为飞机的落脚点，主要图城际铁路去广州方便，顺便让当年的学生给我做了两天导游，然后见了一下传说中的[@Milo Yip](#) 菊苣。本来只是发了条信息跟他说吃饭，结果他居然把整组人都叫来了，于是吃饭的活动就变成了team building。菊苣向我推荐了八合里牛肉火锅，然而我作为一个汕头人，我还从来没吃过。亲测味道十分还原，以后你们想吃牛肉不用跑来汕头了（逃

叶菊苣在大会上说我们认识已经有七年了，这可能是从加QQ号开始算的。我已经不记得是叶菊苣喷某本书先发生，还是他写文章安利《爱丽丝》的头发先发生了，总之我是在其中一件事情里知道有这个人的，然后就在网上搭讪。没想到时间过得这么快，我一直以为只有三四年。

总的来说，我在深圳只有三个感觉：

- 首先是（含全广东）人民的驾驶习惯真是太不能接受了。广州深圳都是大城市，为什么开车就不能像北京上海一样规矩。马路上几乎都是蛇形走位和连续变道。不过这次在汕头终于没有见到擎天柱逆行了，可能是在贤哥的带领下，人民的素质有了提升。
 - 注：这里只讲开车的时候的素质，至于这个司机人怎么样（逃
- 第二个就是这个深圳的物价真是太离谱了。深圳的消费水平跟北上广差不多，但是服务质量就像二线城市一样，让我觉得十分不值，完全没有在北上广那种花了钱觉得爽的感觉。
- 第三个就更恶心了，几个旅行箱的锁在香格里拉酒店里面同时坏掉，屁股也知道发生了什么事情。

于是从此深圳在我的心目中基本上就定格为暴发户了，从北上广深四人组除名。

玩了几天之后就去了广州和汕头，东西还是一如既往的好吃。要是我在美帝赚够了钱，肯定回广州呆着，每天吃饭都开心。自从十几年前在华南皇家理工大学上学开始，就基本上没怎么回汕头了，每次回去都看到有巨大的变化。不过这次看起来形势终于朝着正确的方向发展了，该干的事情都在干，很有前途。

临走前去上海呆了三天，先后跟一些朋友吃了饭，然后就回美国了，并没有什么时间到处参观。在广东呆了几个星期之后，从上海飞机场出来，第一个感受就是，司机真他妈有素质！

忘记说了，KFC新出的北海道雪糕，真好吃！希望美国也有。

P.S.

听说在我离开中国之后，国内的支付行业有了翻天覆地的变化，人民群众纷纷表示只带手机出门就可以活了。于是我本着一个好奇的心情去体验，没想到第一天就发生了旅行箱的事情，于是接下来每一天都只能背着所有不能丢的东西出门。这就像很多软件，明明一个新feature涉及得很好，但是却因为bug的原因使之丧失了应有的作用，结果就等于没做。因此，我仍然不相信现阶段人民群众可以真的只带一个手机出门，而没有新的坏处。等过几年你们再来说服我（逃

总的来说，微信支付的覆盖要比支付宝强一万倍，不过在广州还是有若干次，在搭taxi要付钱的时候，我问司机要二维码，司机是这么说的：

微信鸡付？冬银系现金啦！

油管搬运：《如何提升自信心》

https://www.youtube.com/watch?v=HO5tws_CjOA

看了这个李锡锐的很多视频，大部分观点我都很认同。这是一个在台大讲政治的、推崇主观意识的人。其中有几个观点很值得大家参考，大概的意思就是，一个人的成就主要还是靠主观意识的强大（深入的了解之后其实跟唯物主义是不矛盾的），而一个人的主观的精神力量是否强大，跟环境是很有关系的。但这跟阶级固化还不一样，不是越有钱你就越牛逼，而是你经受到的挫折越大，你没有被打败，你就越牛逼。

今天我看到这个视频很有趣，所以截了个图贴过来（滚到最下），虽然跟我上面说的东西关系也不是特别大，仅供参考。在这里说几句题外话。

想当初在中学学编程的时候，父母经常以学习成绩为理由阻挠我。我那个时候成绩的确也不怎么样，初中的时候都怕考去烂高中，高一的时候属于一本都不知道能不能上的状态，高三突然开窍了，于是就考上了皇家理工大学。所以在相当长的时间里，我的成绩都是无法让大家满意的。所以大家也不要意淫我从小就是个学霸，我从来都只是很普通的成绩，跟这个称号一点关系都没有。

但是我很喜欢编程，于是整天搞编程，当然也就会受到大量的阻挠，包括但不限于：

- 限制我只有周末可以打开电脑，而且如果是在打游戏，还要对我进行精神污染。这个我相信只要是普通人都一定有经验。
- 发现我不开电脑的时候居然不是在学习，撕掉我的书。记忆很清晰，那是一本PowerPoint的教程。虽然后来我也发火了，要求他们一页一页粘好还给我。
- 经常对我说别人的小孩如何如何。所幸我也不是毫无优点，虽然很多科目一塌糊涂，但是至少数学总是稳定在年级最前面几个，所以还没有说教的那么厉害。但是这也有坏处。大家都觉得数学那么鸡巴难，我都能靠那么好，所以其他科目绝对是因为懒所以才没有拿第一名的。我都不知道要怎么回答。你说是吧，显然又不是。你说不是吧，又说不出理由。
- 当我有一次表达了我战略上藐视高考的观点的时候，用拖鞋砸我。

不过他们也没有100%反对，至少我想去书店买书的时候，还是会给我买的，只是让我必须把这件事情放在比高考更次要的位置上。当然我从来都予以坚决的反对，虽然我那个时候从来没有考虑过万一考了一百本要怎么办的问题。

车到山前必有路嘛，所以我在战略上藐视高考，战术上你还是要复习的，不能完全置之不理。只是我没有像别人一样沉迷学习。后来为什么成绩突然就变好了呢？我也不知道，对比起学习编程来讲，可以认为我从来就没有认真对待过高考。后来我总结了一下原因，大概就是我觉得没意思不想学的东西，在高考里面真的他妈就比重低，我觉得这运气也是爆表了。

当然，我觉得我学习编程对我的成绩的提高是有促进作用的，毕竟MSDN的中文从来就是屎，但是你英文版总得看懂才能写出东西来，抽象思维也得练习才不会出岔子，是不是？

而且当时我非常擅长的物理竟然成了所有科目中最低的，我就不说了。不过我门考前填志愿的时候，我已经为这种情况做了准备。结果高考的时候物理一道大题看错题意傻逼了，但是已经填好了的志愿，并没有因此而傻逼。人对自己要有正确的把握。

当时高三成绩突然就提高了，不过我想的不是要奋斗一下上清华北大，至少交大复旦什么的，而是既然我这么牛逼，华南皇家理工软件工程（当时因为学费贵所以只要过了线就能进去），绝对没有问题啦。我更应该把更多的时间放在编程上。我也真的这么做了。最后的结果怎么样呢？当时在微软做实习生的时候，总之我得到了两个评价。

第一个是我们部门的Director，突然有一天跑到我工位上来说：“听说你很屌啊，你这样的人我们都喜欢”。第二个是当时面试我的面试官，有一天我听到他在别人面前吹我的牛，说什么我已经忘记了，总之我听的很爽（逃

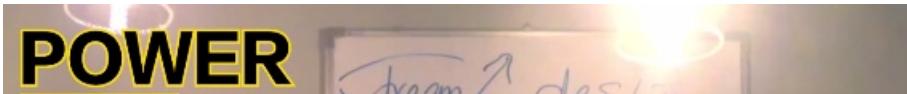
我觉得这已经可以有一个结论了。虽然我也没有说我宇宙第一，但是至少比大多数人写代码强是没有问题的。而这是为什么呢？正是因为我当初顶住了压力，没有屈服，也没有动摇。那我得到了什么好处呢？其实你也不要钱，比我烂的人拿的钱比我多的例子要多了去了，但是我至少心灵上是爽的，人生过得是充实的，合法萝莉也娶到了。所以我不嫉妒这些例子，心态一直都很好，每天都过得很开心。

所以人的主观上的精神力量，是很重要的。

而今天的开心，跟我当初一个月拿两千多块钱的时候，其实是差不多的。而且我今天还在写自己感兴趣的代码，譬如说GacUI。你说我做这个事情有啥用？从功利的角度上来看，并没有用。但是我过得爽呀。甚至我有时候还做一下白日梦，要是我买彩票中奖了（在美帝中头奖可以买北京房子100套），我就吃利息，在家写GacUI。

人活着是为了什么，不就是为了爽嘛？

不过有的人可能会说，你比起那些深山老林里面的猴子已经强多了，凭什么这么跳？这个问题正确的看待方法是：跟我一样家庭背景差不多的人那么多，每一个家长都是要对小孩指指点点的，说出来的话也都一样，那他们当初又做了一个什么选择？那些跟我明显不在同一个环境里面的人，你哪来跟我比，其实比不出东西的，你自己也不能得到有效结论。



點點名

任何東西你去做 但是你做了以後
**不但這個東西很熟練
你自信心就出現了**

我告訴各位一個非常有趣的例子
我一個朋友的一個的年紀比我輕啦

他媽媽跟我年紀差不多

他媽媽在做甚麼事呢 退休以後

那個捷運不是有施工工程嗎

那個路旁就被做高架在挖喔

那他媽媽喔 就沒事幹

就是去應徵一個職業就很有趣

那個職業就是手裡拿一個牌子

那個牌子就畫一個箭頭

叫甚麼呢 汽車改道

他手裡拿一個像乒乓球拍一樣

那個乒乓球牌就畫一個箭頭

車來他就這樣 做到很怨你知道嗎

一個月三萬塊還不錯 6000人民币

然後每天有七八小時

後來他就講到說 他媽媽喔不做了

想要換一個工作

(但)她這種年紀能到哪裡做事

我就後來喔 我就打電話給她媽媽

我就叫她嫂夫人喔

我說嫂喔 呃最近怎樣
啊你怎樣 心情壞 很糟

是怎麼樣糟 做這個(舉牌)做得要死

我就跟她聊天 我說那妳身體怎麼樣

她說我身體還OK啊 我說有沒有毛病啊

她說有啊 血壓太高啊

我說啊那我告訴你

血壓高是因為血液循環不良

血壓高做一個運動

你如果做三個月馬上

你現在多少 她說大概一百六

馬上可以降到一百五

喔有這種秘方喔 我說不用吃藥

(她說)做甚麼事呢

我說你現在就可以做

我說你每天都舉牌都這樣舉

這樣舉 你一天算算

你如果能夠舉兩千下

三個月一定可以降十度

喔有這種事情 我試試看

真的不到一個月 從一百六變到一百五十五

做了兩三個月真的變一百五了

然後她現在開始開心了

現在不只做兩千下 沒車來她也這樣

POWER
點點名

請審慎參酌
個人經驗分享喔



哈哈哈哈哈哈哈哈
哈哈哈哈哈哈哈哈
哈哈哈哈哈哈哈哈
哈哈哈哈哈哈哈哈
哈哈哈哈哈哈哈哈

做得很開心你知不知道
你知道她那個狀況是甚麼呢
生命的自信心她恢復了

她有成就感 我不靠醫生
我不靠甚麼了 我也不靠吃藥
聽說她現在立志把它降到一百二

所以我們會常常覺得說
媽媽給你一個印象 很囉唆甚麼都想管
講起來落落長也不停 一直講一直講
你知道為什麼嗎 那就是信心

他們很有信心
那很可憐是她們的信心沒有著力點
只能管孫子 只能管兒子嘛

那兒子又不常回來
好不容易你一回家被她逮到了
喔你最近怎麼樣 然後就高談闊論

其實那個就是自信心
所以剛剛這個小妹妹啊

她會認為說自信心怎麼恢復
我可以告訴你

你不要去管自信心這三個字
你就挑一個你一直做一直做的事
那個做久以後 自信心很奇怪

自信心它是一個 你敲牆壁敲久以後
它會傳染到你的腦子面來
就會變成敲牆壁會給你自信心

你只要一動一重複

變成skillful(熟練)

skillful以後變成art(功夫)

所以自信心基本上是個art

它是一個我們心靈裡面的一個思維裡面的一個art

所以甚麼叫自信心呢

它會傾向於甚麼東西

(都)會傾向於正向思考

大家覺得不可能的時候

一個有信心的人他會說騙肖欸

闽南话：肖=疯子

他就會去研究就會去試試看

一個沒有自信心的人

明明這條路很寬

唉呦前面不知道會不會變小欸

你看講這種屁話

乾脆就不走了

一個沒有自信心的人怎麼樣都沒有自信心

但是那個自信心的培養 我建議各位

要找一個著力點

那個著力點把它著下去以後

你不要計較

你就一直做一直做一直做

所以一個口才不好的人

我可以告訴你 你找一個著力點

譬如說你的好朋友

對他一直講 看一本書

欸我昨天看一本書怎麼樣

啊你不想聽我還繼續講

他都不知道你在練習你知道嘛
你把它當成那個實驗品
口才就這樣練起來了

C++_v.s._C#

背景

很久以前我曾经给GacUI写过一个工具，功能是把大量的h和cpp文件打包成若干个大的文件，譬如说GacUI最终会输出

- GacUI.h / GacUI.cpp
- GacUIThreads.h / GacUIThreads.cpp
- GacUIReflection.h / GacUIReflection.cpp
- GacUICompiler.h / GacUICompiler.cpp

就是这个工具做的。因为我懒得在Ubuntu下面折腾dotnet core（其实搞过，也很简单，但是不想加入我的一键装环境脚本里面（逃），于是原本在Visual Studio 2010下面用C#写的CodePack，我大概花了加起来一整天的时间，用C++重写了一遍，就可以在Ubuntu下面用clang++来编译了。

重写前：[CodePack \(C#\)](#)

重写后：[CodePack \(C++\)](#)

写完之后很有感触，于是今天我来比较一下C++和C#在处理这些乱七八糟的事情的时候的区别。

首先大家需要对这个工具有一个基本的概念。如何把一大堆文件打包成几组文件呢？大概的想法就是：

- 把h文件一个一个拼起来，cpp文件一个一个拼起来
- 拼h文件的时候要注意顺序，要把被别人include的那些排在前面
- 拼完之后的大文件也要互相include

里面涉及到一个拓扑排序的算法。大家不要学我这么写，因为我是出于实践输入的文件个数只有几百个的现状，从而用最快的速度随便弄了一个拓扑排序，搞出来有 $n^3 \log n$ 那么复杂。我猜实际上是不应该的，因为这个复杂度看起来有点大。下面是最耗复杂度的一个片段：

```
n-> FOREACH_INDEXER(T, category, index, unsorted)
{
    if (!deps.Keys().Contains(category))
    {
        sorted->Add(category);
        unsorted.RemoveAt(index);
        for (vint i = deps.Count() - 1; i >= 0; i--)
        {
            if (deps.Keys()[i] == category)
            {
                deps.Remove(deps.Keys()[i], category);
                break;
            }
        }
    }
}
Worst Case: n * (logn + n + n + n * nlogn) = n^3 logn, 不过平均情况要好得多
```

在实践中，Release编译的时候跑这段函数，就算输入几百上千个文件和它们的include目标列表，排序的时间也是快到看不见，所以就不改了（逃）

现在就来开始一一比对。虽然我并没有使用STL，但是我用到的自己的字符串、正则表达式、容器、foreach和Linq（C++即将加入标准的ranges有几乎一致的功能），STL里面都有。所以看起来很公平。

1：扫描磁盘文件

C#的写法很简单，因为他已经内置了一个搜索文件的函数：

```
static string[] GetCppFiles(string folder)
{
    return Directory
        .GetFiles(folder, "*.cpp", SearchOption.AllDirectories)
        .Select(s => s.ToUpper())
        .ToArray();
}
static string[] GetHeaderFiles(string folder)
{
    return Directory
        .GetFiles(folder, "*.h", SearchOption.AllDirectories)
        .Select(s => s.ToUpper())
        .ToArray();
}
```

而我只有遍历目录的函数，所以要用一个递归。在这里要提出的是，Windows API提供了C#的那个功能，所以你想直接调用也可以，我只是突然发现有这个需求之后，懒得封装，等以后有更多的需要再写进去。

```
LazyList<FilePath> SearchFiles(const Folder& folder, const WString& extension)
{
    auto files = MakePtr<List<File>>();
    auto folders = MakePtr<List<Folder>>();
```

```

folder.GetFiles(*files.Obj());
folder.GetFolders(*folders.Obj());

return LazyList<File>(files)
    .Select([](const File& file) { return file.GetFilePath(); })
    .Where([=](const FilePath& path) { return INVLOC.EndsWith(path.GetName(), extension, Locale::IgnoreCase); })
    .Concat(
        LazyList<Folder>(folders)
            .SelectMany([=](const Folder& folder) { return SearchFiles(folder, extension); })
    );
}

LazyList<FilePath> GetCppFiles(const FilePath& folder)
{
    return SearchFiles(folder, L".cpp");
}

LazyList<FilePath> GetHeaderFiles(const FilePath& folder)
{
    return SearchFiles(folder, L".h");
}

```

2: 对文件进行分类

分类的意思，其实就是按预先设置好的pattern把文件分组。在这里有必要展示一下配置文件：

```

<?xml version="1.0" encoding="utf-8" ?>
<codegen>
    <folders>
        <folder path=".\\Source" />
        <folder path=".\\Import" />
    </folders>
    <categories>
        <category name="vlpp" pattern="\Import\VLPP."/>
        <category name="wfruntime" pattern="\Import\VLPPWorkflow."/>
        <category name="wfcompiler" pattern="\Import\VLPPWorkflowCompiler."/>
        <category name="gacui" pattern="\Source\">
            <except pattern="\Windows\" />
            <except pattern="\WindowsDirect2D\" />
            <except pattern="\WindowsGDI\" />
            <except pattern="\Reflection\" />
            <except pattern="\Compiler\" />
        </category>
        <category name="windows" pattern="\Source\GraphicsElement\WindowsDirect2D\" />
        <category name="windows" pattern="\Source\GraphicsElement\WindowsGDI\" />
        <category name="windows" pattern="\Source\NativeWindow\Windows\" />
        <category name="reflection" pattern="\Source\Reflection\" />
        <category name="compiler" pattern="\Source\Compiler\" />
    </categories>
    <output path=".">.
        <codepair category="vlpp" filename="VLPP" generate="false"/>
        <codepair category="wfruntime" filename="VLPPWorkflow" generate="false"/>
        <codepair category="wfcompiler" filename="VLPPWorkflowCompiler" generate="false"/>
        <codepair category="gacui" filename="GacUI" generate="true"/>
        <codepair category="windows" filename="GacUIWindows" generate="true"/>
        <codepair category="reflection" filename="GacUIReflection" generate="true"/>
        <codepair category="compiler" filename="GacUICompiler" generate="true"/>
    </output>
</codegen>

```

<categories>下面的就是文件的分类，基本上是说哪些文件要被分成一组。<output>规定了每一组最终输出的文件名。首先看C# 的代码。这个函数虽然比较长，但是内容是很简单的。上一步我们已经获取了所有的文件，那么现在即使匹配所有的//category@pattern 和//category/except@pattern，把他们筛选出来，最后把重复的文件干掉。

```

static Dictionary<string, string[]> CategorizeCodeFiles(XDocument config, string[] files)
{
    Dictionary<string, string[]> categorizedFiles = new Dictionary<string, string[]>();
    foreach (var e in config.Root.Element("categories").Elements("category"))
    {
        string name = e.Attribute("name").Value;
        string pattern = e.Attribute("pattern").Value.ToUpper();
        string[] exceptions = e.Elements("except").Select(x => x.Attribute("pattern").Value.ToUpper()).ToArray();
        string[] filteredFiles = files
            .Where(f =>
            {
                string path = f.ToUpper();
                return path.Contains(pattern) && exceptions.All(ex => !path.Contains(ex));
            })
            .ToArray();
        string[] previousFiles = null;
        if (categorizedFiles.TryGetValue(name, out previousFiles))
        {
            filteredFiles = filteredFiles.Concat(previousFiles).ToArray();
            categorizedFiles.Remove(name);
        }
        categorizedFiles.Add(name, filteredFiles);
    }
}

```

```

foreach (var a in categorizedFiles.Keys)
{
    foreach (var b in categorizedFiles.Keys)
    {
        if (a != b)
        {
            if (categorizedFiles[a].Intersect(categorizedFiles[b]).Count() != 0)
            {
                throw new ArgumentException();
            }
        }
    }
}
return categorizedFiles;
}

```

最后一步只是防御性的，如果最终发现同一个文件同时出现在不同的分组里面，就崩溃。我也懒得报错，崩溃了就直接debug这个程序，马上就看见了（逃

下面是C++的代码。可以看到基本上是没什么区别的。但是从现在开始，会发现C++写lambda表达式比C#要啰嗦。当然lambda表达式的参数类型本是不应该写出来的（要用auto替换），只是我的Linq比较旧，当年写的时候C++的lambda表达式还不能是模板函数，所以现在就不能兼容这个新feature了。找个时候把他改了。

至于具体的原因，因为在Linq里面，对于任何的lambda表达式F，我都用

取函数指针返回值的模板类<decltype(&F::operator())>

来获取返回值，从而生成Linq函数的结果类型。如果F的operator()是一个模板函数的话，显然这样写是不行的，以后还是要直接 decltype(declval<F>()(参数))来获取。

```

void CategorizeCodeFiles(Ptr< XmlDocument> config, LazyList< FilePath > files, Group< WString, FilePath >& categorizedFiles)
{
    FOREACH(Ptr<XmlElement>, e, XmlGetElements(XmlGetElement(config->rootElement, L"categories"), L"category"))
    {
        auto name = XmlGetAttribute(e, L"name")->value.value;
        auto pattern = wupper(XmlGetAttribute(e, L"pattern")->value.value);

        List< WString > exceptions;
        CopyFrom(
            exceptions,
            XmlGetElements(e, L"except")
                .Select([](const Ptr<XmlElement> x)
                {
                    return XmlGetAttribute(x, L"pattern")->value.value;
                })
        );
    }

    List< FilePath > filterFiles;
    CopyFrom(
        filterFiles,
        From(files).Where([](const FilePath& f)
        {
            auto path = f.GetFullPath();
            return INVLOC.FindFirst(path, pattern, Locale::IgnoreCase).key != -1
                && From(exceptions).All([](const WString& ex)
            {
                return INVLOC.FindFirst(path, ex, Locale::IgnoreCase).key == -1;
            });
        })
    );

    FOREACH(FilePath, file, filterFiles)
    {
        if (!categorizedFiles.Contains(name, file))
        {
            categorizedFiles.Add(name, file);
        }
    }
}

FOREACH(WString, a, categorizedFiles.Keys())
{
    FOREACH(WString, b, categorizedFiles.Keys())
    {
        if (a != b)
        {
            const auto& as = categorizedFiles.Get(a);
            const auto& bs = categorizedFiles.Get(b);
            CHECK_ERROR(!From(as).Intersect(bs).IsEmpty(), L"A file should not appear in multiple categories.");
        }
    }
}

```

3：递归枚举这个文件直接或者间接#include的所有文件

从这里开始，C#和C++的程序会有一点点区别。上面我提到说把所有头文件拼接起来的时候是要注意顺序的，而这个顺序就是GetIncludedFiles所要做的。然而在C++里面，因为我的容器跟C#的容器有一些区别，导致我无法原汁原味地复制这份代码，于是就把(4)的拓扑排序函数改成了模板函数，最终在(6)的Combine函数里面排序。

因此，在C#的版本里面，排序是GetIncludedFiles做的，而C++的版本则是在Combine函数里面做的。

```
static Dictionary<string, string[]> ScannedFiles = new Dictionary<string, string[]>();
static Regex IncludeRegex = new Regex(@"^s*#\include\s*""(<path>[^"]+)\s*\"");
static Regex IncludeSystemRegex = new Regex(@"^s*#\include\s*\<(<path>[^"]+)\>\s*\");

static string[] GetIncludedFiles(string codeFile)
{
    codeFile = Path.GetFullPath(codeFile).ToUpper();
    string[] result = null;
    if (!ScannedFiles.TryGetValue(codeFile, out result))
    {
        List<string> directIncludeFiles = new List<string>();
        foreach (var line in File.ReadAllLines(codeFile))
        {
            Match match = IncludeRegex.Match(line);
            if (match.Success)
            {
                string path = match.Groups["path"].Value;
                path = Path.GetFullPath(Path.GetDirectoryName(codeFile) + @"\" + path).ToUpper();
                if (!directIncludeFiles.Contains(path))
                {
                    directIncludeFiles.Add(path);
                }
            }
        }

        for (int i = directIncludeFiles.Count - 1; i >= 0; i--)
        {
            directIncludeFiles.InsertRange(i, GetIncludedFiles(directIncludeFiles[i]));
        }
        result = directIncludeFiles.Distinct().ToArray();
        ScannedFiles.Add(codeFile, result);
    }
    return result;
}
```

GetIncludedFiles的内容也很简单，就是把这个文件每一行都用正则表达式找到#include "", 然后打开被#include的文件继续做，直到做完为止。结果会被缓存到scannedFiles变量里面，不会重复使用。C#的版本把后发现的文件放在前面，因为很显然，如果a.h#include了b.h，那么肯定要先打开b才能发现a。但是我写的C++容器没有关键的InsertRange函数，于是我只能大段大段地往后添加。但是你说添加完Reverse一下嘛，他们是不等价的。所以C++版本我就把保证顺序的事情挪到了后面。

```
Dictionary<FilePath, LazyList<FilePath>> scannedFiles;
Regex regexInclude(LR"/(^s*#\include\s*\"(<path>[^"]+)\s*\"/)");
Regex regexSystemInclude(LR"/(^s*#\include\s*\<(<path>[^"]+)\>\s*\"/");

LazyList<FilePath> GetIncludedFiles(const FilePath& codeFile)
{
    vint index = scannedFiles.Keys().IndexOf(codeFile);
    if (index != -1)
    {
        return scannedFiles.Values()[index];
    }
}

List<FilePath> includes;
StringReader reader(ReadFile(codeFile));
while (!reader.IsEnd())
{
    auto line = reader.ReadLine();
    if (auto match = regexInclude.MatchHead(line))
    {
        auto path = codeFile.GetFolder() / match->Groups()[L"path"][0].Value();
        if (!includes.Contains(path))
        {
            includes.Add(path);
        }
    }
}

auto result = MakePtr<List<FilePath>>();
CopyFrom(
    *result.Obj(),
    From(includes)
    .Concat(From(includes).SelectMany(GetIncludedFiles))
    .Distinct()
);

scannedFiles.Add(codeFile, result);
return result;
}
```

可以看出，两个版本的代码仍然是相当接近的。可见处理字符串的时候，其实C++和C#的区别，也就只有语法区别。

4: 拓扑排序

上面已经贴过了，故省略。如果大家对比一下C++和C#的版本，会发现他们主要的区别是：

- C++推崇值类型容器，于是Linq就不可能有C#的ToArray、ToList和ToDictionary这些东西，也更不会有Dictionary<string, string[]>了。
- 后面这个一对多的容器我用Group<WString, WString>来代替，但是Group不能表达key存在但是value不存在的情况。

因此这两个主要区别导致了两个版本的代码在逻辑上稍微有点差异。

5: 最长公共前缀

在查找最长公共前缀的时候，C#的版本算的是字符串，而C++算的是路径，导致了C#的版本如果每一个文件不仅文件夹一样而且文件名的前缀也一样的话，文件名还会被砍掉一部分。不过输出的文件名只存在于注释里，错就错了无所谓，所以一直没有改。我借着这次重写的机会把这个bug修了。

C#:

```
static string GetLongestCommonPrefix(string[] strings)
{
    if (strings.Length == 0) return "";
    int shortestLength = strings.Select(s => s.Length).Min();
    return Enumerable.Range(0, shortestLength + 1)
        .Reverse()
        .Select(i => strings[0].Substring(0, i))
        .Where(s => strings.Skip(1).All(t => t.StartsWith(s)))
        .First();
}
```

C++:

```
FilePath GetCommonFolder(const List<FilePath>& paths)
{
    auto folder = paths[0].GetFolder();
    while (true)
    {
        if (From(paths).All([&](const FilePath& path)
        {
            return INVLOC.StartsWith(path.GetFullPath(), folder.GetFullPath() + WString(folder.Delimiter), Locale::IgnoreCase);
        }))
        {
            return folder;
        }
        folder = folder.GetFolder();
    }
    CHECK_FAIL(L"Cannot process files across multiple drives.");
}
```

6: 把一堆文件粘起来

这个函数太长我就不贴了，总的来说C#和C++两个版本的写法都是一模一样的，除了C++的版本因为上面提到过的原因，要调用一下SortDependencies函数。

7: Main函数

也是太长所以我只贴一个显著有区别的片段，这个片段是用来计算分组之间的依赖关系的，最后输出的结果就是GacUI.h会去#include "Vlpp.h"。先来看C#的版本：

```
var categoryDependencies = categorizedCppFiles
    .Keys
    .Select(k =>
    {
        var headerFiles = categorizedCppFiles[k]
            .SelectMany(GetIncludedFiles)
            .Distinct()
            .ToArray();
        var keys = categorizedHeaderFiles
            .Where(p => p.Value.Any(h => headerFiles.Contains(h)))
            .Select(p => p.Key)
            .Except(new string[] { k })
            .ToArray();
        return Tuple.Create(k, keys);
    })
    .ToDictionary(t => t.Item1, t => t.Item2);
```

多清爽呀！把每一个category的所有cpp文件所直接或者间接include的h文件找出来，然后反向查找它们的category（懒得做索引所以暴力算）。然后看C++的版本：

```
Group<WString, WString> categoryDependencies;
```

```

CopyFrom(
categoryDependencies,
From(categorizedCppFiles.Keys())
.SelectMany([&](const WString& key)
{
    SortedList<FilePath> headerFiles;
    CopyFrom(
        headerFiles,
        From(categorizedCppFiles[key])
        .SelectMany(GetIncludedFiles)
        .Distinct()
    );
});

auto keys = MakePtr<SortedList<WString>>();
CopyFrom(
    *keys.Obj(),
    From(categorizedHeaderFiles.Keys())
    .Where([&](const WString& key)
    {
        return From(categorizedHeaderFiles[key])
        .Any([&](const FilePath& h)
        {
            return headerFiles.Contains(h);
        });
    })
);
keys->Remove(key);

return LazyList<WString>(keys).Select([=](const WString& k)->Pair<WString, WString>{ return {key,k}; });
}
);

```

又臭又长（逃。不过实际上他们是没有区别的，这完全都怪C++的lambda表达式语法太罗嗦，我按照我的喜好排版，只好把一行分成好多行。

结论

总的来说，只要C++的lambda表达式能跟C#一样写的话，那这两门语言的区别也就在于GC和shared_ptr了。在做很多不是极端在意性能的事情的时候，都可以一阵胡写，开发效率应该是相当接近的。

大家还会注意到，C++里面我有时候直接写List<int>，有时候MakePtr<List<int>>，其实主要的区别在于，lambda表达式如果在退出这个函数之后还要被执行，那引用一个List<int>的局部变量就要跪（因为已经被释放了）。这个时候我用智能指针把它拿住，就没有这个问题了。

同样是写正则表达式，C++的LR"FuckShitBitch(abcd)FuckShitBitch"比C#的@"abcd"，要好用一万倍！

如果你做的事情非常在乎性能，那你可能需要考虑一下，一个项目在经过profiling的检验之后，把一小部分C#代码用C++来写。**C++可以通过付出把代码写得超级丑陋并且无法用低薪程序员维护的代价，来换取无敌的性能**，这一点C#就不行，当然这也不是C#设计出来的目的。

最近问我职业规划的好像有点多

实际上很多人的问题我都是没办法给出最好的解决方案的，因为我并没有经历过那些情况，也不知道他们在想什么。我个人认为，只有满足下列情况的其中一项的人来问我相关的问题才能得到比较好的答案：

- 你最大的愿望就是开发出一个工程上很漂亮的系统（而不是拿它当手段）。
- 你是一个程序员，觉得中国环境太恶劣，想走。
- 你再学不会编程就会被杀死。

至于为什么？其实很简单。我做过的事情我当然知道应该怎么做，而我没做过的事情我就不一定可以给出完美的做法。但是实际上问我职业规划问题的人，他们的情况我都没有经历过。其中典型的情况有：

- 什么都不会，但是想混进大公司——只要你放弃马上混进去，没问题。
- 本科毕业找不到开发工作——找个垃圾工作，赚点白饭钱，每天晚上好好学习编程，总有一天会找到的。
- 想学编程，因为钱多——老实说，去殡仪馆抬尸体，性价比要高多了，还不会过劳死。
- 想拿offer，好解决OPT的问题，留在美国——降低时薪期望马上可以解决。
- 想跳槽涨薪——跳。
- 毕业了什么都不会，想马上混进大公司，因为钱多，还能解决OPT的问题留在美国，以后还要跳槽涨薪——问题这么复杂你可能不能指望我免费告诉你（逃

作为一个经验丰富的前面试官，我当然懂得怎么混面试。但是我并不想教大家这个，因为对于他们来说，拿到这个offer可能不是最快、最好的办法。作为一个喜欢fix bug要找root cause的人，如果一个人要解决的问题，编程不是最好的办法，那我就不想告诉他如何通过编程来解决这个问题。举个例子：

- ——方法实在太多了，为啥一定要编程？
- 你想留在美国——方法实在太多了，为啥一定要编程？
- 你想拿很多钱——方法实在太多了，为啥一定要编程？
- 你想留在美国拿很多钱——方法实在太多了，为啥一定要编程？
- 你想被人崇拜——方法实在太多了，为啥一定要编程？
- 你想留在美国被人崇拜方法实在太多了，为啥一定要编程？
- 你想拿很多钱被人崇拜——方法实在太多了，为啥一定要编程？
- 你想留在美国拿很多钱被人崇拜——方法实在太多了，为啥一定要编程？

知乎上那么多例子，其中一些还很出名，其实不需要我教你，你都应该学会。

- 你想实现阶级跃迁——只要你父母穷到爆炸，那这也不是一件很难的事情，你只要付出时间就好了，然后保持一个健康的身体，死前绝对可以跃迁。只是我不保证你可以享受跃迁的结果，那是你孙子要考虑的事情，因为那个时候你儿子可能也差不多退休了——如果你找得到老婆的话（逃

好了，那最后就剩下那么几种人了：

- 你最大的愿望就是开发出一个工程上很漂亮的系统（而不是拿它当手段）。
 - [如何成为牛逼的程序员](#)
 - [靠谱的代码和DRY（图片是GacUI）](#)
 - [为什么我们需要学习（设计）模式](#)
 - 私信、值乎、邀请，etc
- 你觉得中国的自然环境很恶劣，想走——加入跨国公司（微软、谷歌，etc），干几年翻墙，公司的律师会教你怎么办。
- 你再学不会编程就会被杀死——C#。

最后应该有人问，为啥我不帮助他们呢？道理很简单，要我关心的人那么多，你都跟我没见过面，我不想关心你——

因为我想打Hitman 2016，这游戏太他妈好玩了（逃

听说有人向爆照号卖我的赞

今天听到这个消息的时候真是震惊了。本来我就是想让大家看看好看的照片，我也不要求答主一定要素颜，一定要长那个样子，毕竟我点的是图片又不是人。结果发现有人在做这个生意，三个赞就可以买我的台式机了，觉得自己亏了几个亿（逃。他妈的我给微软打工都没有这么容易捞钱。

我觉得我需要先讲清楚，首先没有任何“中介”跟我谈过给爆照帖子点赞的事情，其次我都是看图片质量和心情来点赞的。如果这件事情是真的，那么想必肯定有很多买家发现你跟“中介”谈了然后我没点吧？那么多人at我，我最终觉得好看的其实也没多少，点赞了的屈指可数。现在想想说不定有一些是“中介”雇佣的人肉水军，在赌我会不会点赞。万一我点了呢对吧？

我跟你们说，你们不要浪费这些钱了。想我给好看的照片点赞很容易，你觉得一定要给我点什么的话，就来跟我吃顿饭。要是我觉得你真的长的好看，不仅我以后还会支持你，我们还可以再吃几顿，做个朋友 对不对？我也不收的你钱，还能多交几个朋友，你也满足了你的愿望，这对大家都有好处对不对？

万一你觉得不在美国不方便，你自己at我就可以了，不要去买。等你什么时候来旅游了，我回国了，我们再吃也来得及（逃为什么要这么做高成本的事情？划不来。人长的好看是一件好事，不要搞这些。

【硬广】星际2谐星语音包出炉了！

FBI WARNING

If you want to become rich and "Xie", please stay. What? You don't? Mark my word. You'll encounter weight because you have already gone far...

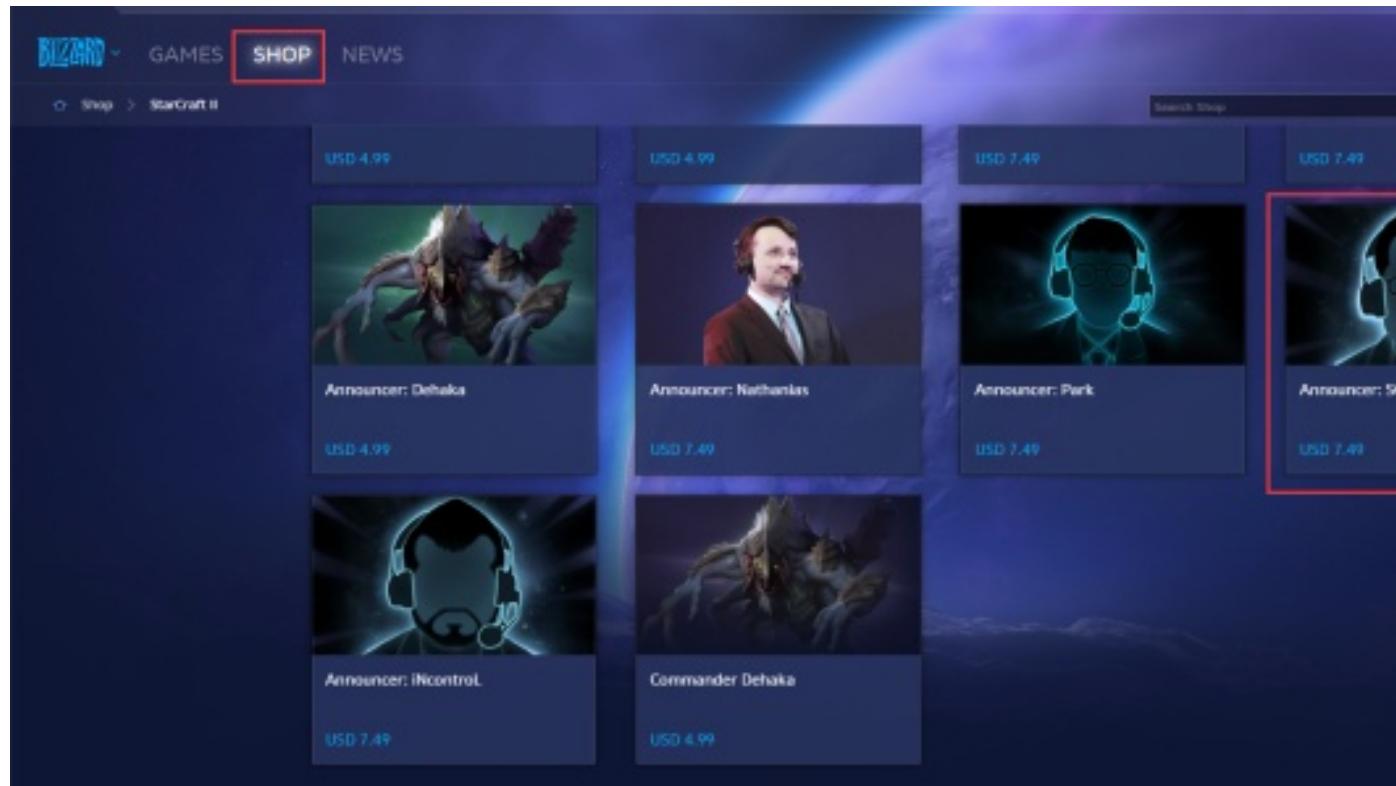
特大喜讯

中国星际2人气（女装）偶像组合星际老男孩推出SCBoy语音包了！！！只需7.49刀！只需7.49刀！走过路过不要错过！想要变强（谐）吗！想要获得（守护者的）力量吗！赶快打开玻璃渣客户端订购吧！

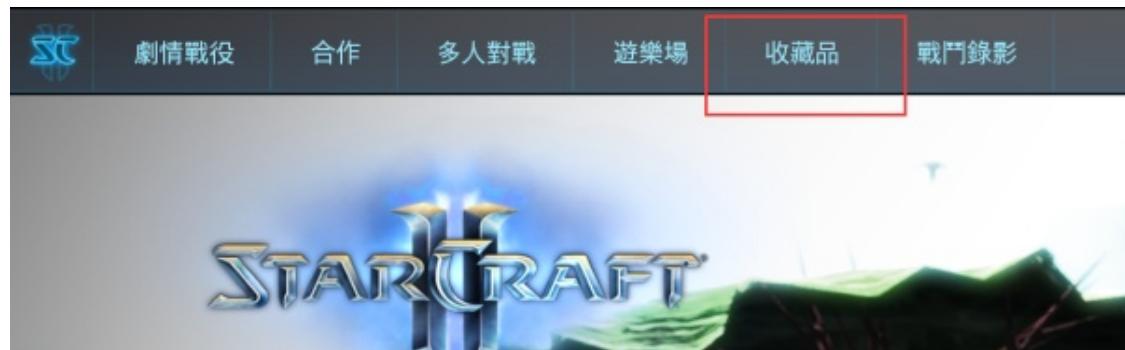
想要召唤守护者吗？想要体会东方神秘力量吗？人生还在期盼第二春却诸多顾虑吗？生怕RTS打不来？不会星际2而导致跟男/女朋友没有共同话题？

统统不怕！智障也有春天！跟攻略赶紧入手吧！

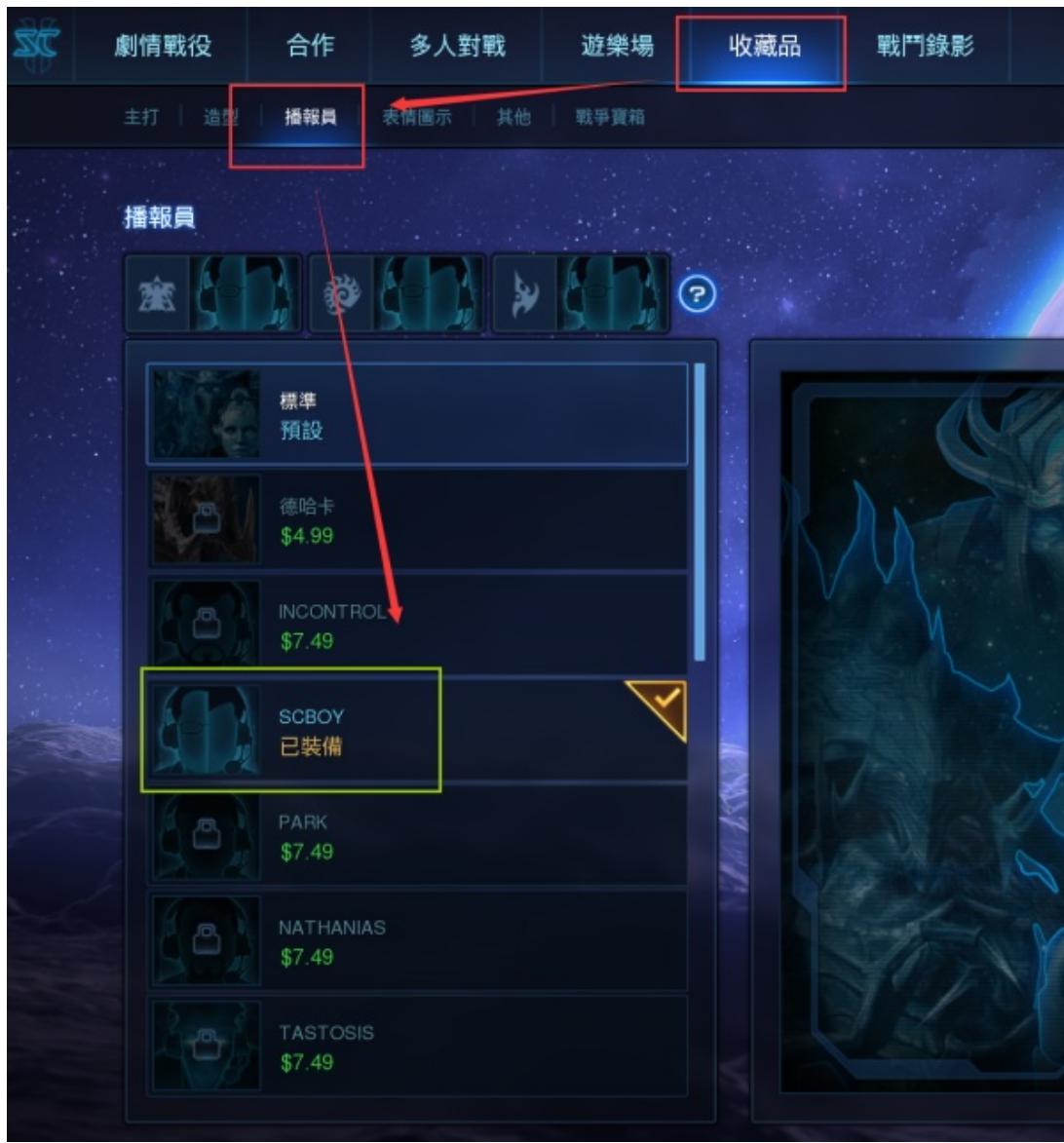
首先打开玻璃渣客户端点击左上角的“商店”，往下滚一波红尘找到谐星。



支付购买，进入游戏，点击“收藏品”开启称霸（谐星）天梯之路！！！



选中“播报员”，解锁专业解说，哦不，专业语音_(3`∠)_



干死黄旭东，光头能对空！想立刻体验却因为在上班，撩妹，做不可描述的事而暂时无法登陆游戏体验？没关系！

孙Dva，黄毒奶谐星语音包提前预览版：

[\[星际争霸2\]星际老男孩语音包全语音试听_电子竞技_游戏_bilibili 哔哩哔哩](#)

赶紧打开去聆听 好哥哥 正义凛然又带点销魂，风致绰约又带点刚猛的星际好声音吧~

一点私货：

据说老仙和会长会把语音包的收入投入到中国星际的事业中去。因此我跟你们的轮带逛 [@vczh](#) 都赶紧入手，人手一份语音包。

我始终坚信，没有老男孩和一种谐星们的坚持和努力，就没有今时今日的中国星际2。老男孩给广大水友带来了很多欢乐，真的很不容易，我们也应该行动起来：人人献出一份爱（money），让智障也有春天！！！



（孙Dva爱你们~）

写代码一定要敢于重构

以前一直不知道为啥Xaml的控件要做成直接把control template赋值进control的属性里，于是我自己在做GacUI的时候，就写了一个IStyleController接口，其实就等与control template对象。

写着写着，我发现有一些控件是得有固定外观的（譬如ScrollView系列，总要有两个滚动条），于是就没办法继续开放ScrollView控件的IStyleController了，只能转而ScrollView提供一个固定的IStyleController实现，然后用template method pattern，做了一个IStyleProvider。所有IStyleProvider的基类跟IStyleController一摸一样，写到这里我就隐约觉得肯定有什么弄错了。

于是两三年过去了，到了2014年，我开始允许GacUI在XML里面写皮肤了，但是XML跟XAML的设计差不多，其实都是在写构造函数，那么用XML去实现接口反而就很别扭，于是我又给每一个控件做了一套Template类，让XML去构造Template，然后用预先写好的代码把相应的Template处理成相应的IStyleController或者IStyleProvider，于是大家就会在GuiControlTemplateStyles.h里面发现一大堆这样的adaptor。

没读过GacUI代码的人可能有些混乱，我就用C#简单的缩写一下：

```
public class Control
{
    public interface IStyleController
    {
        void SetText(string value);
        void SetFont(Font font);
        // composition 其实就是用来排版的方框
        // 方框里面可以放几何图形
        // 整个窗口其实是一颗排版树构成的矢量图
        // 而控件就是一个让你方便地控制一颗子树的facade模式的产物
        // 因为你总不想为了改一个按钮的文字去遍历那棵树吧
        Composition BoundsComposition {get;}
    }

    public interface IStyleProvider
    {
        void SetText(string value);
        void SetFont(Font font);
    }

    public Control(IStyleController controller);
}

public class Label : Control
{
    public interface IStyleController : Control.IStyleController
    {
        void SetColor(Color color);
    }

    public Label(IStyleController controller);
}

public class ScrollView : Control
{
    public interface IStyleProvider : Control.IStyleProvider
    {
    }

    private class StyleController : Control.IStyleController
    {
        // 放两个ScrollBar，需要的参数放进IStyleProvider里面
        public StyleController(IStyleProvider provider);
    }

    public ScrollView(IStyleProvider provider)
        :base(new StyleController(provider))
    { ... }
}

class ControlTemplate : Composition
{
    public string Text{get;set;}
    public Font Font{get;set;}
}

class ControlTemplate_StyleController : Control.IStyleController
{
    private ControlTemplate template;

    public ControlTemplate_StyleController(ControlTemplate template)
```

```

{
    this.template = template;
}

public void SetText(string value) { controlTemplate.Text = value; }
public voidSetFont(Font font) { controlTemplate.Font = value; }
public Composition BoundsComposition { get { return controlTemplate; } }
}

```

然后是控件皮肤的XML:

```

<Instance ref.Class="MyControlTemplate">
    <ControlTemplate Text=Fuck, Font=Shit>
        <!-- 一些装饰品 -->
    </ControlTemplate>
</Instance>
<!--
上面这段代码生成了class MyControlTemplate : ControlTemplate { ... }
-->

```

然后用皮肤来创建控件:

```

<Window>
    <Control ControlTemplate="MyControlTemplate"/>
</Window>
<!--
实际上控件是这么被new出来的:
new Control(new ControlTemplate_StyleController(new MyControlTemplate))
-->

```

依稀记得当年还跟[@装配脑袋](#)讨论过这个问题，后来实在没什么办法，先做成这样。但是这种写法的味道其实很差，主要原因如下：

1. IStyleProvider和IStyleController的功能是一致的。
2. 由于某些控件对外观的限制，导致IStyleController被占用，只好创造IStyleProvider。其实这就意味着IStyleController的继承很难，不该这么设计。那以后要是某个IStyleProvider又被占用了那咋办？再发明一个新的接口，名字就不够用了，而且给继承的深度来定制名字显然是一件愚蠢的事情。
3. XML是构造函数，构造函数只能修改类而不能继承接口，于是又要有一堆adaptor。而adaptor其实就是不断地把函数转发给template，把pull模式和push模式互相变一下。
4. 这样每次添加一个新控件就会超级麻烦。

当然其实我也不是因为犯傻了才这么写的，早期GacUI并没有XML定制控件和皮肤的功能，因此做出来的选择很难把未来发生的事情都完美的考虑进去。这个事情也告诉我们，软件随着功能的变化，以前哪怕是好的设计也可能会变差，一定要敢于重构，才能让软件开发可以持续几十年，不需要推倒重写。

几年前我还曾经想过要写一个《GacUI与设计模式》系列，后来想想这个问题还没有得到妥善解决，因此就先作罢，以后肯定会改的。当然这个设计不行，并不代表这个设计用到的知识是不好的。这里面使用了很多关于Inverse of Control的思想，大家一定要去好好学习一下，而且切记不要使用JavaEE作为学习材料。不知道为什么，那些东西到了JavaEE上都会变味，可能跟他有反射导致大家可以胡乱搞有关系。人一忍不住，就喜欢乱搞。

于是又过了三年，我把很多事情处理得七七八八了，于是重新来想这个问题。后来我发现，Xaml直接让control去跟control template沟通是一件多么明智的选择，完美的避开了上面提的所有问题。

这6年来写GacUI的时候，遇到过很多设计上的烦恼。凡是我自己想出来的一个后来证实非常完美的做法，后来好奇去看Xaml怎么做的时候，都会发现Xaml也是那么做的。凡是我自己想出来的跟Xaml有不一样的地方，多半是我想的不对。

Xaml的前身Avalon（beta版）是2001年就发布了，而Avalon其实是山寨了Office部门内部的一个Win32的XML控件工具包。不过这个工具包已经很老了，成为了Office开发新功能的重要障碍之一。不过当年开发它的其中一个元老，就是我翻墙的时候最后一面的面试官，都已经去做Distinguished Engineer了。剩下有几个爬上了Partner，还有一堆人留在Principal Engineer就不走了。微软的Principal讲道理不是那么好当的。

当然这只是历史了，Xaml很多出彩的东西其实在Office原本的这个工具包里面是没有的，所以功劳还是应该算在Xaml身上。

所以最近就在着手删除IStyleController和IStyleProvider接口，让control直接跟control template沟通。在几万行代码里面搞这种翻天覆地的重构，一直都不是一件容易的事情。但是能让代码大幅度变好的事情，当然是要做的。所以大家会在TODO.md里面看到最近添加了三步：

1. 把所有IStyleProvider换成IStyleController，多出来的要用来继承的实现就转移进相应的ControlTemplate类里面。（今天刚做完）
2. 把IStyleController删掉，让控件直接跟ControlTemplate沟通，实际上就等于把GuiControlTemplateStyles.h的那一大坨几乎都只有一行的函数直接inline到控件里面去。
3. 最后把构造函数的ControlTemplate*参数改成一个可以运行时修改的属性。

重构也不能一口吃成个胖子，要一步一步来，安全地进行重构，把代码慢慢改好。

啊，终于变得跟Xaml一样了（逃。下面的C#代码大概就是重构后的样子：

```
public class ControlTemplate : Composition
{
    public string Text {get;set;} // GacUI的属性可以自带Changed事件
    public string Font {get;set;}
}

public class Control
{
    public virtual ControlTemplate Template {get;set;}
}

public class LabelTemplate : ControlTemplate
{
    public Color Color {get;set;}
}

public class Label : Control
{
    // 检查传进来的真的是LabelTemplate
    public override ControlTemplate Template {get;set;}
}
```

干净清爽！Xaml大法好！

知乎竟然就默默的加入了一个大杀器

找个机会试试，永远告别抢前排。

我来在这篇文章开启一下试试看

P.S.

竟然还是有那么多人抢前排。在显示之前就先拉黑，爽！

我也终于用上Powershell了

背景

众所周知，GacUI的源代码在github上是以一个[organization](#)来维护的，不同的模块放在了不同的repo里面。起初这是为了互相依赖的时候，方便并行开发，但是随之而来的问题就是如何才能简单的更新依赖。好在我用的是C++，那么我其实只要把整个项目的代码合并成一对或者少数几对.h文件和.cpp文件就可以了。一来比起直接给你用几百个文件可以节省80%以上的编译时间，二来很多著名的库也是这么干的，想必大家都可以接受。于是我当然也就采取了这种办法。

那么互相更新就很容易了。每一个repo有一个Release文件夹，每当需要更新的时候就跑个脚本把所有的C++代码都粘起来，然后[正确处理掉那些#include](#)，最后放在Release文件夹就可以了。每个repo也有一个Import文件夹，里面的代码是从隔壁repo的Release复制过来的。那么这个跨版本的依赖关系就成立了。上游repo不管怎么更新，我的Import只要不变，那我仍然依赖的是旧的模块。有些时候特别方便。

其实我一开始也想过用git submodule，但是这个煞笔东西不能解决菱形依赖的问题，which在软件开发的过程中基本上是不可避免的，于是只好弃疗，用我现在的办法了。

然而这带来一个问题，就是互相更新依赖的时候特别烦，要跑很多脚本。因此我在刚开始的时候是用C#来干这个事情的，后来把这个巨大的程序的不同的部分抽出来成为独立的程序，用bat粘起来。再后来为了方便我移植到Ubuntu，不想乱七八糟折腾，我就把他们都用C++和Vlpp重写了一遍。那么最近我做的事情，就是把所有的bat都换成了powershell的ps1了。

总的来讲，其实bat调用msbuild才更方便，因为msbuild官方就只给你cmd的环境。但是现在Windows 10为了推广powershell，淘汰落后的cmd，竟然默认把“在这里打开cmd”隐藏了，只有powershell了。再加上Github for Windows已经在powershell里面把git配置好了，我就觉得现在是个机会，因此花了几分钟学习powershell，[把整个脚本重新做出来了](#)。

使用GacUI的同学们会发现，我其实在Release里面附带了几个小的bat文件，用来跑UI的XML描述，生成跨平台的代码（当然在这里主要解决的是跨x86和x64的问题）。现在这个bat没有了，都换成ps1了（逃

PowerShell

之前我也算是大规模使用过bash和cmd的人了，对于如何在这两门垃圾语言里面使用数据结构具有丰富的经验，这使得我一直很想自己开发一门为了命令行优化与法的脚本语言。当然因为有别的事情干，我一直没有做。这几天学习powershell之后，我觉得这个东西就做得挺好，除了命名方式有一点不对我胃口以外。不过这并不重要，因为我早就习惯了在不同的语言里面使用不同的习惯了。当然从一开始是JavaScript那个语言规范逼的，他妈的大括号换行还能改变程序的意思，这是我人生中第一次有这样的体验。

总的来说，我觉得powershell有以下几个优点

1. 很简单，装完Windows就有了，不用折腾。
2. powershell具有丰富的和Windows交互的能力，而且很多软件也支持，譬如说SQLServer也可以用powershell来管理。
3. powershell的设计是具有一致性的，你不需要总是去记得各个不同的命令之间微妙的区别，特别是rm *遇到一个叫做-rf的文件怎么办的时候的问题，在powershell就不存在。因为powershell是有语法的，参数不同的位置的-rf，代表的到底是参数名还是文件名，是可以通过语义分析找到的（逃
4. powershell的字符转义不多，特殊语法基本没有，而且名字也是完整的单词，不会看着什么sed/grep一脸懵逼。我在不知道的时候，怎样也想不到正则表达式相关的命令叫这个。在这里我就想到之前在ubuntu下面也是做如何扫描硬盘输出makefile的脚本，正则表达式跟数组混在一起这代码简直不能看，[现在我都不知道自己的写什么了](#)。
5. powershell是区分类型的，也就是说你不可能把array和string搞混，大多数情况下运行前就能给你抓到，因为powershell的编译器是有类型推导的，跟F#一样（虽然没有那么厉害）。他会整个文件读进去，一行一行校对完没问题，再给你执行。有一次甚至抓到了我不小心把null传进一个需要string的参数的命令里面的错误，看到这个报错的时候简直惊呆了。
6. 最重要的，他能调用.net类库。再也不需要去找别人写的命令了，有什么系统管理的事情是.net做不了的，没有（逃。甚至是遇到一些极端情况，我还可以借助.net的力量来调用Windows API——而不用先用C++/C#写最后编译完code和binary一大堆文件都打包在一起——你只要在powershell里面框起来直接写C#就可以了。
7. powershell启动进程的语法很方便，也很自由，基本上可以做到跟.net里面使用Process类一样多的事情。这使得我最终调用msbuild的时候非常方便。我写了个powershell的函数，模拟了Developer Command Prompt for VS2015的启动过程，然后在里面接着跑一堆命令来编译我的代码。
8. powershell能写函数，有异常处理。需要中断的时候无论callstack有多少东西一个throw就出来了。最外面catch一下善后，拜拜。到处jump和if error变量的日子一去不复返。
9. 流行的Linux发行版上也有powershell。

尾声

总的来说，我其实最终是要完成一件事情，就是在我往github送代码的时候，有一个服务器可以监视我的活动，然后自动build我的代码，把所有的unit test跑一遍，如果过了的话就更新依赖，重新测试，打包GacUI，发布release，更新网站上面的函数参考文档，所有事情一气呵成。不过目前远远没这么自动化，但是powershell让我看到了希望，用bash和cmd的时候想都不敢想。

当然在这里不得不指出，bash已经这么烂了，但是我由于先学bash后学cmd，体验了一把由奢入俭难的过程，他妈的cmd更烂。不过反现在都能用powershell了，管他麻痹（逃

最近很多知友都很热心

总是发私信来让我关注或者回答什么东西。这个我觉得没有问题，我通常都会看一看的。只是问题在于，这些私信通常都是下面这样的：

“关注一下我提的问题”——哪个问题？

“关注一下我新写的答案”——哪个答案？

有的人比较好一点，会告诉我这个问题到底是什么，但是知乎的垃圾搜索功能怎么可能搜的出来呢？

很多时候我就只能点进这些人的记录里面看他到底干了什么，用智商判断一下到底要我关注的是哪一个。一开始还好，但是现在已经有将近62万粉丝了，这么干不是办法。这些缺乏上下文的问题自然语言处理是搞不定的。

我只用浏览器上知乎的，下次记得贴个链接。你们趁机学习一下，如何找到你们的问题和答案的链接。很简单的。App也可以做到。

写代码一定要敢于重构（2）

重构

时隔两个月，《[写代码一定要敢于重构](#)》所提到的GacUI三个重构任务，终于完成了。现在GacUI的控件和列表都拜托了一大堆，以用XML开发GUI为前提的情况下，再也不需要的Inverse of Control系列接口。而且控件也终于可以在运行时切换皮肤了。

之间修了几个layout的bug，有一些是XML写的不好，终于把黝黑皮肤也基本改好了。除了窗口的一些属性（譬如说隐藏最小化按钮）还没反映到皮肤上面以外，我觉得基本可以用了。

我也发现了一个问题，就是如果你把子控件的AlignmentToParent属性绑定到父控件的Bounds的计算结果上面的话，就会出问题。不过这个问题是没法解决的，只能让大家不要这么写。这是在我做progress bar的皮肤的时候发现的。原理其实很简单：

- 假设现在progress bar是50%，那么里面绿色的部分AlignmentToParent就设置为{left:0 top:0 right:(parent.Bounds.y2/2) bottom:0}
- 假设progress bar有100像素宽，那么绿色的部分就是50像素宽。
- 然后progress bar外面的人想要计算progress bar宽度的最小值，结果由于绿色的部分的AlignmentToParent的意义就是：“我距离父控件的右侧至少50像素那么大”，结果传播上去也就变成了progress bar至少要有50像素那么大，才能够让绿色的部分距离右侧50个像素那么大。
- 于是这就造成了错误的结果，因为progress bar的宽度的最小值理应是0。

解决这个问题有两个简单的办法：

- progress bar的MinSizeLimitation设置为NoLimit，这样他在计算自己的最小值的时候，就不会去管绿色的部分的设置，永远都返回0。但是这样的坏处是，如果有人在progress bar里面放控件（虽然我猜不会有人这么做），那么子控件就没办法把progress bar撑大了。这就变成了另一个bug。因为GacUI的控件默认都是要被子控件自然撑大的，这其中也包括按钮被文字自然撑大。
- progress bar的绿色部分的位置不要通过AlignmentToParent来实现。这也是我最终采取的做法，把他从<Bounds>换成<PartialView>就可以了，刚好PartialView的功能就是按照百分比依照父控件排版，然后告诉父控件计算最小值的时候不要管自己的设置。于是所有问题都消失了。

不过这仍然依赖于用户自己去修改XML来做到，GacUI没办法改，不然就要做太多额外的计算了。

文档

最近有人抱怨GacUI文档的问题。其实我也觉得现在的文档不行，因为他就是把代码里的注释贴到网页里面去给你们看。这个问题终究是要解决的，但是要写成tutorial，得先做到下面几点：

- 要有足够多的时间，把文字写详细。
- Tutorial里面的Vlpp部分的程序要能够高亮、单击名字的时候跳到现在的函数参考网页上、还能自动给出结果。
- Tutorial里面的Workflow部分，参考Vlpp，但是他不是C++，还得做一遍。
- Tutorial里面的GacUI的部分，使用XML写例子之后要能自动生成截图，还要在文件夹下面留下一大堆VS工程以供大家把玩。
- 在Tutorial之余还要给出Reference，其实这就还是那个把注释变成网页的工作。现在的C++ parser做的不够好，我再想办法到底是用clang++的呢，还是自己用C++再写一遍。

做到这几步需要很多劳动，而且到底要文档先写还是GacStudio先写我现在还没决定，但是总之这也是一个大工程。

以后争取做的跟MSDN一样好用，每一个单元都凑齐About、Using和Reference三大部分。

GacUI 1.0 进度

当然还是参考[这个文件](#)。主要内容就是，在开始做GacStudio和摆弄文档之前，还得把下面的工作做完：

- 黝黑皮肤不够，还得加一个纯洁色的。
- 处理好更多的皮肤细节，譬如说窗口的各种外观属性，还有焦点，还有Tab等等。
- 让XML生成Workflow的部分变得更好看，现在写的还是有点乱。
- 富文本框真正实现剪切功能，现在只能支持纯文本剪切。
- 让XML写的窗口和控件可以被继承。
- XML写动画（目前只能用C++写）！
- 资源之间可以相互依赖，譬如说一个Resource.xml里面可以使用另一个Resource.xml的新类和新控件。
- 重做ParserGen.exe，生成的语法树要能够跟字符串双向绑定（偶也！），还要有一系列其他高级功能，帮助IDE的开发。我觉得地表最强的parser生成器就要诞生了（逃。因为现在的所有parser生成器主要还是面向编译器的，出来的代码都没有为交互功能做准备，导致各大IDE基本上还是靠硬撸来完成各种炫目的功能。不过我也是第一次做这个，不知道要花多久。

所有的这些功能还得在GacStudio里面做好编辑器。总的来说，到底是美帝下一轮经济危机先来，还是GacUI 1.0先完成，这个真说不准（逃

觉得很有必要再次推荐编程自学界神文

《[Teach Yourself Programming in Ten Years](#)》

不过我知道大家肯定是不想看英文的了，所以我给大家找了[中文版](#)。所有试图自学编程的人都应该发自内心的同意它的说法（除去少数过时的具体技术部分）。

正文

你们着什么急？

随意步入一家书店，满目都是《7天搞定Java编程》这种速成书目，同样的书籍还包括Visual Basic、Window系统、Internet互联网等等，它们都承诺在几天，甚至几小时之内就让你能够学会相关技术。我在亚马逊网站上做了如下的条件检索：

[pubdate: after 1992 and title: days and](#)

[\(title: learn or title: teach yourself\)](#)

出版日期：1992年以后，题目关键字：“天”，“学会”或者“自学”

然后得到了248条搜索结果。头78条都是计算机类书籍（第79条记录是《30天学会孟加拉语》）。我将“天”关键字换成了“小时”，不出意外地搜索到了253条记录，其中头77条记录是计算机书籍，第78条的搜索记录是《24小时语法和样式自学手册》。在总共搜索到的头200条记录中，有96%是计算机书籍。

从上面的搜索结果可以看出来，要么就是人们对计算机技术的学习如饥似渴，要么就是计算机技术实在太简单，不费吹灰之力就能学会。相比于计算机技术书籍的如此“速成”，在其他领域的书籍里，你却很难找到诸如：“三天学会贝多芬”，或者“五天搞定量子力学”，这种速成教材，甚至连《狗狗喂养手册》这种宠物指南，都鲜有“几天搞定”的说法。Felleisen et al. 在他们的著作《如何设计程序》一书中明确指出了这种“速成”的趋势，并评论到：“垃圾的编程技术当然非常容易，傻子都能在21天之内学会，哪怕他天生就是个白痴。”

让我们来仔细看看《[3天学会C++](#)》这种速成教材实际上意味着什么：

●**学会：**在3天时间里你几乎没有时间去写任何有意义的程序，就更不要谈什么从编程中获得经验和教训这种事情了。你也不可能有时间和有经验的程序员一起工作和交流，也不会体验到在真正的C++环境下工作是什么感觉。长话短说吧，你就是没时间，也学不到什么。所以这种书籍最多也就让你有个粗浅的印象，但是绝对不可能有深入的理解。就像亚历山大教皇说的那样，“浅尝辄止是很危险的”。

●**C++：**如果你有其他编程语言的基础，那么3天之内你也许可以学到C++的一些语法，但即使是这样，你还是无法了解如何使用该语言编程。简言之，如果你之前是一个Basic程序员，那么经过3天的学习，你会成为一个“能使用C++语法编写Basic风格程序的程序员”，不过这样是没法发挥出C++语言本身的优势的（说句不好听的，你连怎么犯C++的典型错误都不会）。仅仅知道一点语法意味着什么呢？Allan Perlis曾经说过：“一个无法改变你思维方式的编程语言是不值得学习的。”；另一种可能性是，你可以只学一点点C++知识（类似的，或者一点点JavaScript，或者一点点Flex Script），然后就可以利用现有的工具制作应用接口，完成特定的编程任务了。但是这样的行为并不意味着你“会”编程了，你只是会使用这个工具完成任务而已。

●**3天：**很不幸，3天是远远不够的，往下看你就知道了。

研究人员([Bloom\(1985\)](#)、[Bryan & Harter \(1899\)](#), 见文后参考书目)、[Hayes \(1989\)](#)、[Simmon & Chase \(1973\)](#), 见文后参考书目)的一系列调查研究显示，在各个领域内，要想获得专业级别的水平，大约需要10年时间的努力。参与此项调查的领域包括：国际象棋，作曲，发报，绘画，钢琴演奏，游泳，网球等。科学家们从神经心理学和拓扑学的角度对这些领域进行研究，并得出结论。若要在某一领域内达到专家级的水平，其关键在于“审慎地重复”，也就是说，并非是机械地，一遍又一遍地练习，而是要不断地挑战自我，试图超越自身当前的水平，通过不断的尝试挑战，并在尝试的过程中和尝试之后对自身的表现进行分析和总结，吸取经验，纠正之前犯过的各种错误。把这一“审慎”的过程不断重复，才能取得成功。

所谓的“捷径”是不存在的，即使对于莫扎特这种天才来说，也没有捷径可走，尽管4岁就开始作曲，可是他也花了13年的时间，才真正地写出了世界级的作品。再举一个例子，甲壳虫乐队(The Beatles),他们似乎在1964年凭借一系列热门单曲和其在艾德沙利文秀(The Ed Sullivan show)上的演出一炮而红，但是你也许不知道，他们早在1957年就在利物浦和汉堡两地进行小规模演出了，而在此之前的非正式演出更是不计其数。甲壳虫乐队的主要成名曲《*Sgt. Peppers*》，则是1967年才发行的。

[Malcolm Gladwell](#)公布了他对柏林音乐学院所作的一项研究报告，该研究对比了一个班里的学习成绩为上、中下三个档次的学生，并逐一询问他们进行音乐练习的时间

这三个档次中的所有人，大约都是在5岁的时候开始练习音乐的，一开始的时候大家练习音乐的时间都差不多，大约一周2到3小时。但是到了八岁左右，大家的区别就开始体现了。后来成为班里最好的那一部分学生开始比别的学生练习得更多，大概每周6到9小时，12岁的时候每周8小时，14岁的时候每周16小时，往后则越来越多，直到20岁左右，他们每周练习音乐的时间已经超过30小时了。在20岁的年纪，那些精英级别的演奏家们都有累计超过10000小时的音乐练习时间。相比之下，仅有部分优秀生能达到8000小时的累计练习时间，而那些音乐教师级别的学生，他们的累计练习时间只有4000小时左右。

所以，也许这个让你能达到专业等级的神奇时间应该是10000小时，而不是10年。（Henri Cartier-Bresson (1908-2004)说过，“（作为摄影师），你所拍摄的头10000张照片都是垃圾”，但即使是垃圾作品，他拍一张照片也要花接近一小时。）Samuel Johnson (1709-1784)认为这个时间应该更长：“在任何一个领域要想做到极好，势必穷尽一生的精力，否则根本无法企及。”Chaucer (1340-1400)也发出过“生命如此短暂，技能如此高深”的感叹。Hippocrates (c. 400BC)因写下了如下的句子而被人称颂：“ars longa, vita brevis”，该句是来自于一个更长的引用：“Ars longa, vita brevis, occasio praecipit, experimentum periculosum, iudicium difficile”，这段话翻译成英语就是：“生命很短暂，但是技艺却很高深，机遇转瞬即逝，探索难以捉摸，抉择困难重重”。这段话是用拉丁文写的。在拉丁文里，ars可以翻译为“技艺”或者“艺术”，但是在古希腊文里，ars只能做“技能”的意思，而没有“艺术”的意思。

你想当程序员么？

下面是我列举的程序员成功“食谱”

- **沉醉**于编程，编程是为了兴趣。保持这种充满兴趣的感觉，以便于你能将其投入到你的10年/10000小时的编程时间中。
- **程序**。最好的学习方式是“[在实践中学习](#)”。更技术一些地说：“一个人在某个专业领域方面能够达到最高水平，并不是因为这个人经验增长了以后而自动获得的，而是这个人为了进步所做出了专门的努力之后产生的结果。”([p. 366](#))“最有效的学习包括以下几个要素：明确并且难度适当的任务，适应学习者个人情况，及时的信息反馈，有重新开始和改正错误的机会”(p. 20-21)《[Cognition in Practice: Mind, Mathematics, and Culture in Everyday Life](#)》这本书提供了上述有趣的观点
- 同其他程序员 **交流**，多阅读其他人写的程序。这些远比你看书或者上培训班重要
- 如果你愿意的话，就选择去读一个计算机科学专业吧（当然你还可以去念这个专业的研究生）。如果你能做到这点，那么你就有机会找到一些需要计算机学位认证的工作，也会让你对这个行业有更深的理解。不过，如果你不是上学的料，那么你可以（当然要有足够的毅力）靠自己学习，或者通过工作来积累经验。无论你采用哪种途径，光依靠书本是远远不够的。“如果说仅仅靠学习油画和调色技术无法创造出顶尖的画家的话，那么光学习计算机科学课程更不能造就顶尖的程序员。”，Eric Raymond这样说过，他著有《新黑客字典》一书。我所聘用过的最好的程序员仅仅只有高中文凭；他写了很多伟大的软件，他有他自己的新闻组，并且通过股权赚够了钱，还开了家属于自己的夜店。（作者说的这个人是Jamie Zawinski，他是网景浏览器（Netscape）的早期开发这者之一，也是开源项目Mozilla和[XEmacs](#)的主要贡献者，他开了一家叫做DNA_lounge的夜店，位于旧金山的SoMa区——译者注）
- 与其他程序员一起 **做项目**。在某些项目中要尽量做到最好，在某些项目中却别做那么好。当你是最好的时候，你的领导能力就会得到锻炼，并激发你高瞻远瞩的视野。当你做得不好的时候，你就能知道你的领导怎么做事，以及他们不喜欢哪些事（因为领导总是把那些他们不爱做的杂事丢给他们认为不得力的人去做）
- 尝试 **跟随**其他程序员一起做项目。尝试去理解其他人所写的代码。看看如果你无法找到代码的作者本人的情况下，理解和修正他写的代码需要花费什么样的代价。同时也思考，如何规划你自己的程序代码，让它们更容易被其他人理解和维护。
- 至少学习半打 **编程语言**。包括一种支持类抽象的语言（例如Java或者C++），一种支持函数抽象的语言（例如Lisp或者ML），一种支持语法抽象的语言（例如Lisp），一种支持声明式编程的语言（例如Prolog或者C++模板），一种支持协同程序的语言（例如Icon或者Scheme），一种支持平行并发编程的语言（例如Sial）
- 牢记“计算机科学”中包含着“计算机”这个词。了解计算机需要花多长的时间执行一条指令，花多长时间从内存中获取一个字(word)（包括缓存命中和不命中两种情况），如果连续从磁盘中获取数据，时间消耗如何？以及需要花多少时间才能再磁盘上定位一个新的位置？
- 尽量参与语言的 **标准化**过程。往大了说，你可以试着加入ANSI C++委员会这样的专业组织，往小了讲，你也可以从自己的代码规范入手，限定代码缩进是需要2个空格宽还是4个空格宽。无论采用哪种方式，你都需要了解其他人对于语言的喜爱，以及他们的喜好的程度，甚至你要知道他们为什么产生这样的喜好的原因。
- 有良好的意识，能尽快适应语言标准化的成果。

要掌握上面所说的所有内容，光靠看书学习应该是很难做到的。当我的第一个孩子出生的时候，我几乎阅读了市面上所有的《如何...》指南书籍，但是我读完了以后还是觉得自己是个菜鸟。30个月以后，我的第二个孩子快出生时，我难道还要做一个书虫么？不！相反，我此时更依赖我的个人经验，这些经验相比于那些上千页的书籍，则更加有效和让我放心。

Fred Brooks所著的著名的论文《[No Silver Bullets](#) | 没有银弹》里向我们揭示了发现和培养软件设计人才的三步骤：

- 1.有组织地辨认顶尖的软件设计人才，越早越好
- 2.安排一个职业导师，为其职业前景指点迷津，并谨慎对待自己的职业履历
- 3.为成长中的设计师们提供机会，让他们能够互相激发促进。

即使一部分人已经具备了成为优秀软件设计人员的潜质，也需要经历工作的慢慢琢磨，方可展现才华。[Alan Perlis](#)则说得更加直接：“任何人都可以被‘教’成一个雕塑匠，但米开朗基罗则被‘教’如何不要成为一个雕塑匠，因为他要做的是雕塑大师，。这个道理放到编程大师身上同样管用。”Perlis认为，伟大的软件开发人员都有一种内在的特质，这种特质往往比他们所接受的训

练更重要。但是这些特质是从哪里来的呢？是与生俱来的？还是通过后天勤奋而来？正如Auguste Gusteau（动画电影《料理鼠王》里的幻象大厨）所说，“谁都能做饭，但只有那些无所畏惧的人才能成为大厨！”我很情愿地说，将你生命中的大部分时间花在审慎地练习和提高上，这很重要！但是“无所畏惧”的精神，才是将促使这些练习成果凝聚成形的途径。或者，就像是《料理鼠王》里那个与Gusteau作对的刻薄的美食评论家Anton Ego说的那样：“不是任何人都能成为伟大的艺术家，不过，伟大的艺术家在成名前可能是任何人。”

所以尽管去书店大买Java/Ruby/Javascript/PHP书籍吧；你也许会发现他们真的挺管用。但是这样做不会改变你的人生，也不会让你在整体经验上有什么提高。24小时，几天，几周，做一个真正的程序员？光靠读书可读不出来。你尝试过连续24个月不懈努力提高自己么？呵呵，如果你做到了，好吧，那么你开始上路了……

参考书目

Bloom, Benjamin (ed.) [*Developing Talent in Young People*](#), Ballantine, 1985.

Brooks, Fred, [*No Silver Bullets*](#), IEEE Computer, vol. 20, no. 4, 1987, p. 10-19.

Bryan, W.L. & Harter, N. “Studies on the telegraphic language: The acquisition of a hierarchy of habits. *Psychology Review* , 1899, 8, 345-375

Hayes, John R., The [*Complete Problem Solver*](#) Lawrence Erlbaum, 1989.

Chase, William G. & Simon, Herbert A. [*‘Perception in Chess’*](#) *Cognitive Psychology* , 1973, 4, 55-81.

Lave, Jean, [*Cognition in Practice: Mind, Mathematics, and Culture in Everyday Life*](#), Cambridge University Press, 1988.

问答

典型PC系统各种操作指令的大概时间

execute typical instruction

执行基本指令

1/1,000,000,000 sec = 1 nanosec

fetch from L1 cache memory

从一级缓存中读取数据

0.5 nanosec

branch misprediction

分支误预测

5 nanosec

fetch from L2 cache memory

从二级缓存获取数据

7 nanosec

Mutex lock/unlock

互斥加锁/解锁

25 nanosec

fetch from main memory

从主内存获取数据

100 nanosec

send 2K bytes over 1Gbps network

通过1G bps 的网络发送2K字节

20,000 nanosec

read 1MB sequentially from memory

从内存中顺序读取1MB数据

250,000 nanosec

fetch from new disk location (seek)

从新的磁盘位置获取数据（随机读取）

8,000,000 nanosec

read 1MB sequentially from disk

从磁盘中顺序读取1MB数据

20,000,000 nanosec

send packet US to Europe and back

从美国发送一个报文包到欧洲再返回

150 milliseconds = 150,000,000 nanosec

附录：如何选择语言

很多人曾经问过我，他们应该选择什么编程语言作为入门之用？我想这个问题很难有一个确切的答案，但是下面几点可以用来作为选择的参考。

● **随大流**。当被问到“我应该使用什么系统呢？Windows, Unix还是Mac？”，我的回答通常是：“看你的朋友们用什么你就用什么。”这么做的好处是，有了你朋友的帮助，你就能有效地回避操作系统固有的一些差异，对于选择编程语言来说，也是同样道理。同时，你还要有点儿战略眼光：如果选择了一种编程语言，并成为其编程社区的一员，那么你选择的语言和社区是正在不断壮大？还是奄奄一息？如果你有编程方面的问题，能不能从相关的书籍，网站以及在线论坛中得到解答？你是不是跟论坛里的人合得来？这些都是要考虑的。

● **简单实用**。诸如C++以及Java这样的编程语言都是非常专业的开发语言，适用于有经验的大型团队进行开发，需要时常考虑代码的运行效率。所以，这类的编程语言就适合于那样（复杂）的编程环境。如果你是一个初学者，那么就不要搞那么复杂。你所需要的是一个简单易学的编程语言，你靠你自己就可以搞定的语言。

● **交互**。给你两种选择去学钢琴：第一种，常规做法，也是互动的做法，也就是你每敲一下琴键就能听到琴音；第二种，批量模式，等你把所有该按的琴键都按了一遍，然后再一次性放给你听。你选择哪一个呢？显然，交互式的方式对于钢琴学习来说更容易，对于编程学习也是如此。那么就坚持使用交互式模式学习编程吧。

基于上述的观点，我所推荐的编程入门语言应该是Python或者Scheme。但是读者自身的环境是非常复杂多变的，所以你们也许会有其他更好的选择。如果你的年龄还不到两位数，那么你们应该考虑Alice语言或者Squeak语言（很多成年的初学者也认为他们很有趣）。当然，做出选择并开始行动，这个最重要。

附录：书籍和其他资源

● **Scheme**: 《[Structure and Interpretation of Computer Programs \(Abelson & Sussman\) | 计算机程序的构造和解释](#)》 这本书或许是计算机科学最好的入门书籍，本书从计算机科学的角度入手，教你如何进行编程。你可以在[online videos of lectures](#)观看本书的在线视频教程，以及[complete text online](#)的在线文字版。学习本书是需要一些挑战的，相信该书会让一部分人望而却步的。

● **Scheme**: 《[How to Design Programs \(Felleisen et al.\) | 程序设计方法](#)》是一本好书，该书介绍了如何用优雅并且有效的方式进行程序设

● **Python**: [Python Programming: An Intro to CS \(Zelle\)](#) 是一本介绍Python的好书

● **Python**: [Python.org](#) 官方网站上提供了一些在线教程 [tutorials](#) .

● **Oz**: [Concepts, Techniques, and Models of Computer Programming \(Van Roy & Haridi\)](#) 被看做是Abelson & Sussman（经典计算机教程《计算机程序的构造和解释》的作者——译者注）的当代传承者。该书绘制了一幅关于计算机编程的宏观蓝图，它囊括了比

Abelson & Sussman的经典教材更广泛的知识领域，也更加通俗易懂。此书中使用了一种编程语言，Oz，这种语言在工业领域内几乎不被使用，其主要目的就是针对于教学

备注

T. Capey指出，Amazon网页上那个[Complete Problem Solver](#)页面把《21天搞定孟加拉语》以及《21天学会语法》这两本书移到了“购买此书的用户还购买过这些产品”这个区域内。我估计大部分人就是从这个区域看到这本书的。感谢Ross Cohen的帮助。

神文系列（2）——喜鹊开发者

在[上一篇文章](#)的评论里面，有人提到了《The Magpie Developer》。这也是一篇好文章，只是应用范围没有《Teach Yourself 10 Years of Programming》那么大。不过没关系，该看的还是要看。

英文版：[The Magpie Developer](#)

中文版：[喜鹊开发者（The Magpie Developer）](#)

正文

我经常感觉，开发人员很像我们所说的喜鹊，以不停的获取很多小玩意来装饰他们的窝而著称。就像喜鹊一样，开发人员通常都被定义为聪明的、好奇的生物。但是我们太容易被一些时髦的新鲜事物分心。

Scott Hanselman的终极开发工具列表（Ultimate Developer Tool list）不会再使我有新鲜感。相反，它越发使我疲劳。软件开发的改变速度是非常迅速的，而我们太沉迷于一些自身概念就在不断瓦解的新鲜事物，就像一个英语单词如果一遍一遍的不断重复就会变成毫无意义的元音和辅音，新事物最终都会变为平凡常见的，当他们被称为新事物时他们便不会是独一无二的、有趣的。最终，你会厌倦这无止境的新鲜玩意儿。

无独有偶，Jeremy Zawodny也注意到了《新事物的不断黯淡无光》（the diminishing luster of shiny new things）：

一年前我退订了Steve的博客，因为他每天不断的更新最新最潮的一些小东西，频率实在是太高了。而我认识中的很多人都被卷入到了这令人窒息的新事物的喧嚣中，而往往忘记了去思考那些新出现的事物在我们的长期发展中是不是真的那么重要。

Dave Slusher也一致同意：

[Robert Scoble]说他收到过太多通过邮件来获取他的PR releases，但这并没有什么卵用。他建议你应该到他的Facebook wall中留言，Dear god and/or Bob... 在我关注Scoble期间，我看到他说了太多类似这样的话：别发邮件、发推特，别发推特、通过Jaiku联系我，在留言板留言、发短信给我，不要打我电话、发邮件给我，不要发邮件给我、不要打电话给我... 真的是够了！我甚至都没尝试去联系过他，我发现我已经厌倦了他高频的从一个平台迁移另一个平台，这简直就是Bullshit！当我一年前抛弃TechCrunch时我也有这样的感觉。我已经十分反感听到用另一种只有细微差别的方法来代替我们现在已经在做的事情，这些细微差别何以能让我们抛弃一切然后奔向它？我正式宣布放弃搜寻一些光鲜亮丽的事物。

不只有永无止境的新技术，而且还有永不消停的软件宗教战争（thousand software religious wars）都使我们疲乏，就像激流中的巨石那么的讨厌。我相信David Megginson概括的这些过程听起来会十分熟悉：

1. 一些顶尖的开发者们注意到太多的菜鸟们都在使用他们正在使用的编程语言，然后他们开始寻找一些新的东西来区分他们更优秀于一些普通的同事。
2. 他们会丢掉一些他们看起来非常烦人的一系列旧东西，从而寻找一种新的、鲜为人知的语言让他们看起来明显成为少数精英部分。
3. 他们开始促进新语言的发展，提交代码，编写框架等等，推广新语言。然而，高级开发者们也跟着这些顶尖开发者们转向新语言，创造一些列的图书，培训等等，并且加速发展着这门新语言。
4. 这些高级开发者对社区有着很大的影响，开始把新语言推向平常的工作中。
5. 大量菜鸟们又开始意识到他们必须去买一些书，上一些课来学习这门新语言。
6. 顶尖的开发者们注意到太多的菜鸟们都在使用他们使用的编程语言，然后他们开始寻找一些东西来区分他们更优秀于一些普通的同事。

希望你安静的坐着，因为我还有一个坏消息给你。你对Ruby on Rails很感兴趣对吗，但它已经过时了，我们已经抛弃它使用新的东西了。

大量主力的开发人员从没有接触过任何一门动态语言，更别说Ruby。但一些动态语言的特征已经慢慢的多层次的渗透进了Java和.NET的堡垒。这些所谓的思想领袖留下了一座其他人没有机会到达的虚拟鬼城。

我成为了一名开发者是因为我热爱计算机，然而热爱计算机，你必须拥抱变化，然而，我愿意。但是我在想，喜鹊开发者们有时候喜欢通过改变来削弱他自身的技能。Andy Hunt 和 Dave Thomas在2004 IEEE column (pdf)上说的很好：

用户才不关心你是否用了J2EE、Cobol或者一些奇妙的东西。他们只需要他们的信用卡授权被正确的处理，账单被正确的打印出来。你帮助他们发现了他们真正的需求和一个他们设想中的系统。

相反的，被想要艰难登上最新技术的巅峰而冲昏头脑的，Pete正在集中精力为客户构建一个系统(使用COBOL)。它很简单，几乎是简单的最高标准。但是它易用，易懂，可快速部署。Pete的框架混合了一些技术：模型，核心的生成器，可复用的组件等等。他实践了最基本实用原则、什么技术适合用什么技术，而不只是什么最新使用什么。

当我们尝试想造出一个全功能的应用框架来替换市面上所有的应用框架，我们肯定会失望。也许是压根没有这一类大一统的理论。就像后现代主义的其中一个印记：没有宏大叙事，没有大事件来指引我们，相反的，是有许多小的故事而组成。一些人认为，这就是我们这个时代的特点。

别因为自己没有去尝试那些新鲜事物而感到自己不够好。谁会去关心你使用了什么技术，只要它运行的流畅，你和你的用户都满意，这就足够了。

新事物的美丽之处在于：永远都会有新的东西出现。不要让追求新鲜事物无意中成为你的目标。避免成为一个喜鹊开发者。有选择性的追求一些新的东西，你将会发现你会成为一个更好的开发者。

听说江湖大学的空明流转也要开live教高考了，那么我

在live支持Windows Phone之前，我还是不会开的。但是WP也一副要完蛋的样子，不过开发WP的技术跟开发Windows 10平板应用的技术是完全一致的，所以支持UWP，能在平板上跑起来就可以，技术上就等于支持WP了。

虽然我再过十几天我的IpHONEx就要到了，但是作为[@Edi Wang 国的国民](#)，我这个条件是不会变的。而且还会遵守“条约”，到手了马上安装微软全家桶，特别是自己部门的东西（逃

参考：[想开个Live教如何九周高考211，一周211本校备研](#)

使用coroutine实现状态机

很久以前给GacUI的Workflow脚本加入了[coroutine](#)的功能，现在马上就要有状态机了。状态机是实现GUI或者他大量的东西，使用很频繁的一种抽象手段。但是每一次做的时候都没有办法在代码写得好看的前提下把状态机抽象成库。其原因很容易理解，因为状态机也是一个控制流的问题，而不改语法是没有办法修改控制流的。

一直以来都在尝试把状态及翻译为coroutine，直到最近终于找到了一个比较简单的办法。《凌波微步》曾经讲到一个类似的问题，如何用状态机来实现计算器。现在我也以这个作为例子，来描述一下我的这个想法。

首先我们给出一个简化后的计算器的ViewModel应有的样子：可以输入数字、小数点、还有加号、乘号、等号、清除。写成代码大概就是这个样子：

```
class Calculator
{
    prop Value : string = "0" {const}

    func Digit(i: int): void;
    func Dot(): void;
    func Add(): void;
    func Mul(): void;
    func Equal(): void;
    func Clear(): void;
}
```

这个类给了你一个只读的、带有ValueChanged事件的Value属性，还有象征按钮的几个函数。一个简单的ViewModel就这样做出来了。在GUI上面，只需要把Value属性绑定到为本框里面，然后把按钮绑定到几个函数上，这样程序就能用了。因此关键就在于实现这几个函数。

刚开始学编程的时候，如果试图做一个这样的东西，都会有比较大的挑战。因为这并不是那种两个输入框+一个输出框的计算器，而是只有一个显示屏幕，上面显示的内容，有时候等于计算结果，有时候等于编辑了一半的数字，很容易搞错。不过如果我们用状态机的思维来思考他，虽然不能降低复杂度，但是至少在思路上就可以理清楚，写代码的时候也就有底气了。

现在让我们先不要考虑状态本身应该由什么变量组成。状态机的结构无非就是状态加上一些象征状态转换的边，边通常意味着GUI上的输入，那么很明显，下面的6个函数自然就是边了。同一个函数在不同的状态下面会执行不同的内容，并且跳转到其他状态下。因此如果直接来实现这个东西的话，就相当于把整个逻辑都打散了。

因为我们所期望的代码组织方法是，把逻辑按照当前状态分类，把一个状态下面按不同按钮的行为写在一起。而不是像现在这样按输入来分类。

那到底这种代码要怎么表达呢？目前我提供了一个思路，虽然还没有最终定稿，实现出来可能会有一些细微的差别。既然要按照状态分类，那我们就把状态看成一个声明语句：

```
class Calculator
{
    var valueFirst : string = "";
    var op : string = "";
    prop Value : string = "0";

    func Update(value : string) : void
    {
        SetValue(value);
        valueFirst = value;
    }

    func Calculate() : void
    {
        if (valueFirst == "")
        {
            valueFirst = value;
        }
        else if (op == "+")
        {
            Update((cast double valueFirst) + (cast double Value));
        }
        else if (op == "*")
        {
            Update((cast double valueFirst) * (cast double Value));
        }
        else
        {
            raise $"Unrecognized operator: ${op}";
        }
    }

    $state_machine
```

```

{
    $state_input Digit(i : int);
    $state_input Dot();
    $state_input Add();
    $state_input Mul();
    $state_input Equal();
    $state_input Clear();

    $state Digits()
    {
        $switch(pass)
        {
            case Digit(i)
            {
                Value = Value & i;
                $goto_state Digits();
            }
        }
    }

    $state Integer(newNumber: bool)
    {
        $switch(pass)
        {
            case Digit(i)
            {
                if (newNumber)
                {
                    Value = i;
                }
                else
                {
                    Value = Value & i;
                }
                $goto_state Digits();
            }
        }
    }

    $state Number()
    {
        $push_state Integer(true);
        $switch(pass_and_return)
        {
            case Dot()
            {
                Value = Value & ".";
            }
        }
        $push_state Integer(false);
    }

    $state Calculate()
    {
        $push_state Number();
        $switch
        {
            case Add():      {Calculate(); op = "+";}
            case Mul():     {Calculate(); op = "-";}
            case Equal():   {Calculate(); op = "=";}
            case Clear():
            {
                valueFirst = "";
                op = "";
                Value = "0";
            }
        }
        $goto_state Calculate();
    }

    $state
    {
        $goto_state Calculate();
    }
}

```

就像这样。一个类的一旦以\$state_machine结尾，那么他就会给你用\$state_input的留个函数之外，还给你生成一个RunStateMachine函数用来初始化。RunStateMachine就等于从\$state{...}开始执行。现在我们来人肉执行一下。

假设我们输入的东西是 $1.2+30*2$ ，那么结果自然是 $(1.2+30)*2=62.4$ ，这也是普通的计算器会给出的结果。在一般情况下

, valueFirst、op和Value分别代表左操作数、操作符和右操作数。当你按下1+2之后，如果在某个时间点触发了Calculate函数，那么valueFirst将等于3，Value也等于3（为了显示在GUI上，ViewModel当然是直接跟GUI挂钩的），op等于"+”。这个时候按下了数字键，数字键在不同的情况下，要么把数字增加到Value后面，要么直接替换掉Value，根据当前处于不同的状态有不同的行为。

\$state什么都不干先跳转到了Calculate（虽然名字跟Calculate函数一样但实际上 是不同的东西）状态，\$Calculate也什么都不干先在对栈上压入一个Number状态。主义\$goto_state和\$push_state的区别。\$push_state会把指定的状态当成一个新的状态机的初始状态来运行，等退出以后回到\$push_state的下一行语句开始运行，他们的关系就像goto和函数调用一样。Number也什么都不管先压入了Integer，Integer则开始执行\$switch。

\$switch是一个等待输入的语句，这个时候代码就停在这里了，直到我们调用了Digit函数，也就是按下数字键的时候才开始执行。注意这里的\$switch(pass)，pass代表了如果输入不满足要求的话，就直接跳过\$switch运行下面的东西。自己推演的话会发现，（如果没有写错）直接按下 "+" 会导致左操作数当成0处理。这也就是为什么Value属性的初始值是0。

好了现在我们按下了"1"，那么case就进去了。Integer进来的时候newNumber是true，所以Value就被赋值成了"1"。现在我们跳转到了Digits状态，在这个状态里面，又开始执行\$switch。

现在我们按下了"."。\$switch(pass)匹配失败，所以Digit状态就结束了，整个状态机也就结束了。这个状态及是从Number状态的第一行压进去的，所以这个时候就退到了\$switch(pass_and_return)这里。\$switch里面一共可以写

- 不写：匹配错误就抛异常。
- pass：匹配错误就接下去执行，让下一个\$switch匹配。
- pass_and_return：匹配错误就直接结束当前状态机，让下一个\$switch匹配。
- ignore和ignore_and_return也差不多，区别在于他不会让下一个\$switch去匹配，而是直接把输入忽略了。下一个\$switch会进入等待状态。

现在我们命中了，所以Value变成了"1."。然后有进入了Integer(false)状态。

现在我们按下了2，Value变成了"1.2"，Number开始的状态及也结束了。这个状态机是Calculate的第一行压进去的，于是又进入了下一个\$switch。

现在我们按下了"+"，因此执行Calculate函数，结果很明显，现在valueFirst是"1.2"，op是"+"，Value还是"1.2"没有变。然后开始了新一轮的循环。剩下的你们自己模拟一下。

我们可以发现，当我们按下数字键的时候，取决于我们是在Integer(true)还是Integer(false)里面，Value会被不同的方式修改。因此我们已经完成了把逻辑按照状态来组织的目标了。唯一的问题就是实现它。

实现其实并不难，毕竟coroutine已经早就有了，我们把它编译成coroutine，最后coroutine又会被编译成普通代码，几轮codegen下来，控制流就被分解为正常编程语言的东西了。

在这里贴一下我手写的生成出来的代码：[vczh-libraries/Release](#)。这几天我就把它实现了，然后就可以出一个新的计算器Demo。XML脚本里面就是这个简单易懂的状态机，而生成的C++代码，当然只能是以输入来组织代码。大家就能看到我是如何把这个东西拆开成普通地控制流语句的了。

使用coroutine实现状态机（2）

自从[使用coroutine实现状态机](#)以来已经有两个星期了，在这两个星期里我把这个想发给实现了。语法做了一点点小修改，但是总的来说没有区别。上一篇文章的计算器的例子作为单元测试的一部分被添加了进去，大家可以在看这两个文件：

[CoSmCalculator.txt](#)：测试用的状态机，支持加法和乘法。

[CoSmCalculator.cpp](#)：生成的C++代码。C++当然没有状态机的功能，所以Workflow脚本的状态机自然被改写成了普通的代码，变成了一个真正的状态机。

Workflow脚本要求使用了\$state_machine的类必须继承自system:StateMachine，再加上Workflow不允许多重继承里面出现同一个类多次，所以\$state_machine的父类当然不能是\$state_machine。这个限制是草稿里面没有的。

整个状态机被编译成了一个使用[coroutine](#)表达的状态机，而coroutine又被编译成了一个状态机，所以生成的代码里面两个嵌套的状态机被合并到了一起，这么扫一眼根本看不懂（逃

Workflow的状态机支持\$goto_state和\$push_state两种模式。\$goto_state说的是你跳转过去之后就不回来了，如果那个状态结束了之后没有跳转到任何其他状态，那么整个状态机就结束了。\$push_state就是还要回来的疑似。\$goto_state就像jump，\$push_state就像函数调用。函数调用自然是堆栈的，所以\$push_state自然也有堆栈——这个被生成的coroutine的previousCoroutine变量隐式表达成了一个外部访问不了的链表。

这个链表的根节点放在了system:StateMachine类里面，类的[ResumeStateMachine](#)函数识别了多个coroutine之间切换的意图，把他们有机地结合了起来。因为一个coroutine的暂停和结束，有可能意味着\$push_state和"pop_state"，自然不能让ResumeStateMachine暂停，而是要继续跑下去，直到状态机真的要开始等待外部的输入为止。

状态机的输入是用\$state_input定义的，最后他们就变成了函数。你调用这些函数，就是在输入数据。因此最后在测试用例里面，我们的用例长这样子：

```
func main():string
{
    var c = new SMCcalculator^();
    s=$"[${c.Value}]";
    var handler = attach(c.ValueChanged, func():void
    {
        s=$$(s)[${c.Value}]";
    });

    c.Digit(1); // 1
    c.Dot(); // 1.
    c.Digit(5); // 1.5
    c.Add();
    c.Digit(2); // 2
    c.Digit(1); // 21
    c.Dot(); // 21.
    c.Digit(2); // 21.2
    c.Digit(5); // 21.25
    c.Mul(); // 22.75 (1.5 + 21.25)
    c.Digit(2); // 2
    c.Equal(); // 45.5 (22.75 * 2)
    c.Clear(); // 0

    detach(c.ValueChanged, handler);
    return s;
}
```

这个代码模拟了用户点击计算器按钮的过程，Value属性代表了液晶屏上显示的数据，每次有改变的时候，这个值就会被append到全局变量s里面。最后输出的结果自然应该是：

```
[0][1][1.][1.5][2][21][21.][21.2][21.25][22.75][2][45.5][0]
```

因为你在点计算器的时候，上面的数字的确就是这么变化的。

使用coroutine实现状态机（3）

状态机的GacUI做出来了！源代码见：[CalculatorAndStateMachine](#)

在这里简单描述一下，使用GacUI开发一个计算器的简单过程。

一、使用状态机来实现一个ViewModel

上一篇文章已经见过了，略过不表，具体的脚本代码见：[Resource.xml](#)

二、把GUI写出来，并做好数据绑定

这里需要指出 Resource.xml 所使用到的GacUI的功能。

首先是排版，这个如果用过GacUI的都很容易，写个<Table>把窗口分成5行4列，马上就能做出上面的UI来。

其次是数据绑定。这里我们需要把显式文本的部分绑定到Calculator的Value属性上：

```
<SolidLabel ... Text-bind="calculator.Value"/>
```

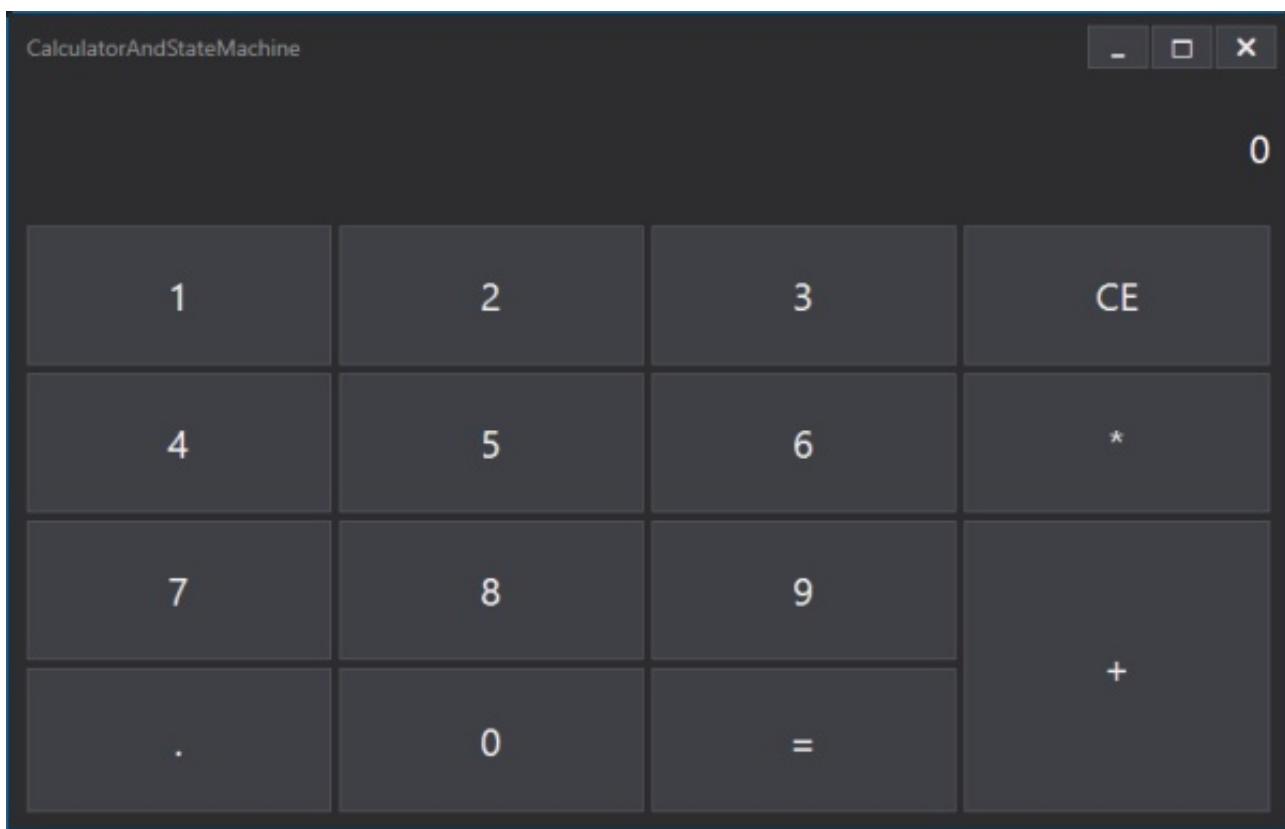
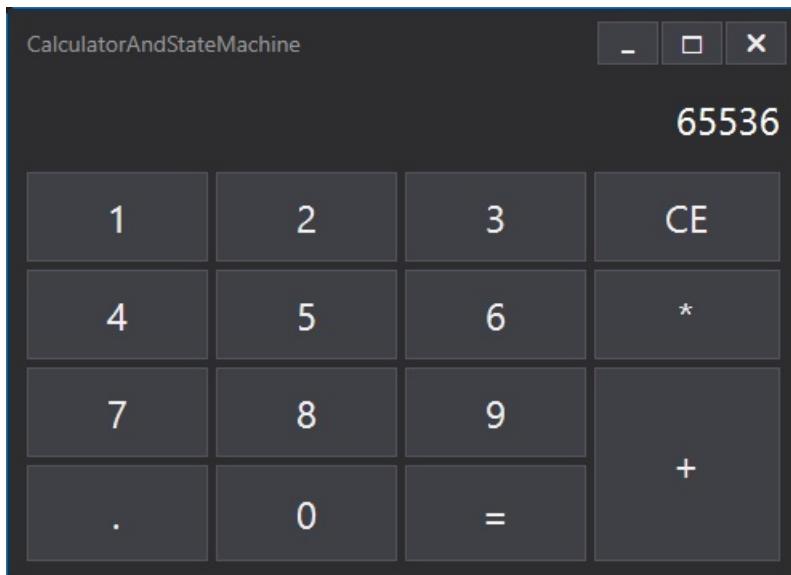
然后就是一堆按钮，用来调用Calculator的几个函数：

```
<Cell Site="row:1 column:0">
    <Button Text="1" ev.Clicked-eval="calculator.Digit(1);"/>
</Cell>
<Cell Site="row:1 column:1">
    <Button Text="2" ev.Clicked-eval="calculator.Digit(2);"/>
</Cell>
<Cell Site="row:1 column:2">
    <Button Text="3" ev.Clicked-eval="calculator.Digit(3);"/>
</Cell>
<Cell Site="row:2 column:0">
    <Button Text="4" ev.Clicked-eval="calculator.Digit(4);"/>
</Cell>
<Cell Site="row:2 column:1">
    <Button Text="5" ev.Clicked-eval="calculator.Digit(5);"/>
</Cell>
<Cell Site="row:2 column:2">
    <Button Text="6" ev.Clicked-eval="calculator.Digit(6);"/>
</Cell>
<Cell Site="row:3 column:0">
    <Button Text="7" ev.Clicked-eval="calculator.Digit(7);"/>
</Cell>
<Cell Site="row:3 column:1">
    <Button Text="8" ev.Clicked-eval="calculator.Digit(8);"/>
</Cell>
<Cell Site="row:3 column:2">
    <Button Text="9" ev.Clicked-eval="calculator.Digit(9);"/>
</Cell>
<Cell Site="row:4 column:1">
    <Button Text="0" ev.Clicked-eval="calculator.Digit(0);"/>
</Cell>

<Cell Site="row:4 column:0">
    <Button Text"." ev.Clicked-eval="calculator.Dot();"/>
</Cell>
<Cell Site="row:4 column:2">
    <Button Text="=" ev.Clicked-eval="calculator.Equal();"/>
</Cell>
<Cell Site="row:1 column:3">
    <Button Text="CE" ev.Clicked-eval="calculator.Clear();"/>
</Cell>
<Cell Site="row:2 column:3">
    <Button Text="*" ev.Clicked-eval="calculator.Mul();"/>
</Cell>
<Cell Site="row:3 column:3 rowSpan:2">
    <Button Text)+" ev.Clicked-eval="calculator.Add();"/>
</Cell>
```

注意ev.Clicked-eval。如果不写-eval的话，那属性名就是一个函数名，然后把refCodeBehind="false"去掉，这样你们就会得到一个MainWindow.cpp，然后就可以自己去填写事件响应函数的内容了。不过在这里没有必要，直接用脚本语言都写上。我相信大家不可能看不懂。

最后就是统一设置按钮的属性，这里需要让按钮的字体变大，然后还要让他填充满整个单元格，实现排版效果：



我们可以使用样式功能来完成。首先定义一个样式，放在Resource.xml的根目录下面，这样它的名字就是res://Styles

```
<InstanceStyle name="Style">
<Styles>
  <Style ref.Path="//Cell/Button" Font="fontFamily:'Segoe UI' size: 24 antialias:true">
    <att.BoundsComposition-set AlignmentToParent="left:0 top:0 right:0 bottom:0"/>
  </Style>
</Styles>
</InstanceStyle>
```

在这里我们匹配所有Cell里面的Button（语法是XPath的简化版，随便看眼XML文档就明白），然后做

- button->SetFont(...)
- button->GetBoundsComposition()->SetAlignmentToParent(...)

最后在窗口里面引用这个样式：

```
<Folder name="MainWindow">
<Instance name="MainWindowResource">
  <Instance
    ref.CodeBehind="false"
    ref.Class="demo::MainWindow"
    ref.Styles="res://Style">
```

...

三、生成C++代码，编译出exe运行

生成后的cpp文件见：[DemoPartialClasses.cpp](#)

在这里需要调用Release\Tools\GacGen.ps1， -FileName参数是Resource.xml，马上就给你产生Source文件夹，里面的代码拖进vcxproj，立刻可用。

马上就完成了！真是容易啊！

必须声明一下，我只有知乎、微博和Github是这个头像和名字。其他的都是假的。

仅指各大网页账号，其中微博叫 [@GeniusVczh](#)。其他几个死掉的我就不提了，关注了也不会看到东西的。

App号则只有微信和Steam，其他App我都不使用，如果说有的话以后会用专栏验明正身。

虽然QQ也用，但是已经不私聊了，你们加我我也不会通过的（逃

健身长肌肉果然是没有骗人的

今天跑去微软开的好好活着健康中心又测试了一下body composition，发现比起健身前，虽然体重从180斤增长到了188斤，但是脂肪只长了半斤，体脂率稳步下降。

说不定再练个几年就可以恢复到200斤，成为香蕉君一样自由的男人了（逃

如何正确安装Visual_Studio_2017

- 1、下载在线安装程序 [下载 | IDE、Code 和 Team Foundation Server | Visual Studio](#)
- 2、右下角点击“如何离线安装”
- 3、通过正确理解页面里的内容，最终会把你带到 [Install on low bandwidth or unreliable network environments](#)

于是就很容易了，先下载Visual Studio 2017 社区版的安装程序，然后开个admin权限的cmd输入以下内容：

```
vs_community.exe --layout C:\Fuck --lang en-US
```

保持网络通畅，最好翻个墙（如果你要开发Android或者Xamarin等），等一段时间，30G的东西会下载到C:\Fuck里面。以后你把它保存在U盘或者是把目录共享出去，所有人都可以离线安装Visual Studio 2017的任何组件。

Surface Book 2 初体验 (1)

昨天公司终于发了SB2给我，第一件事当然是要把公司的软件都装好，然后开个新账号把GacUI和Steam装进去测试一下。在今天有限的时间里面我只完成了“编译GacUI”的操作。这项操作在我家里2016年买的那台

i7-6700k + 32GDDR多少来着 + GTX 1080 + 1T SSD

的台式机上运行了 40分钟 。现在这台顶配15寸Surface Book 2是

i7-8650U + 16G + GTX 1060 + 512G SSD

一共运行了 65分钟 。作为一台笔记本性能跟我的台式机只差1/3，而且发热量也还行，冬天摸上去还挺舒服，我觉得已经满足了（逃

如果你们想要自己运行测试

你们可以在自己的电脑上编译一下GacUI，做个对比。步骤很简单：

1. 安装最新版Visual Studio 2017，选上C++和.net的所有桌面开发选项
2. 把 <https://github.com/vzjh-libraries> 的 Vlpp、Workflow、Tools、GacUI、Release 这5个repo都clone到同一个目录里
3. 添加环境变量 VLPP_VSDEVCMD_PATH，内容是VS2017自带的Developer Command Prompt快捷方式加载的那个 VsDevCmd.bat。如果你们安装的是专业版，那默认的路径就是 C:\Program Files (x86)\Microsoft Visual Studio\2017\Professional\Common7\Tools\VsDevCmd.bat。这个你们要自己找一下。
4. 使用PowerShell运行Tools\Tools\Build.ps1
5. 这个编译的过程一共有若干个步骤，到了倒数第二步Release，如果你的CPU只有8核的话，CPU使用率将高达100%，整台电脑将无法使用。
6. 如果你的locale不是GB2312的话，到时候可能会有三个文件（不含复制）被代码生成器修改，反正你们也没权限check-in，不用管（逃。这个问题以后再修。这是因为我还少量的中文注释没有清理。
7. 等到结束，最后会打印出来总时间。

运行的时候插着电，电量一直保持100%，表明SB2拿来当高性能开发机完全是没问题的。今晚回家再跑个游戏试试看。

Surface_Book_2_初体验_(2)

顶配SB2玩了半个小时的黑魂3，没有发生传说中的插着电源还掉电的现象，谣言不攻自破（逃

不过那个触摸板有一个问题，首先是左右键没法一起按，其次是因为键盘有两个按键按下去，右键基本就不识别了。玩黑魂的时候还有平时流畅打字所没有的半秒钟延迟现象。于是玩黑魂不能一边走一边砍，不能一边走一边开始巨盾，不能马上拼刀，不能马上滚。本来Gundyr的动作就那么灵敏，要比正常再提前半秒预判基本是不可能的，等看到前摇的时候就已经晚了。

家里没有多余的有线鼠标，不知道插上到底怎么样，是不是会好一点。用手柄测试了一下，0延迟，药都不用嗑，只被打了1/6血，Gundyr就直接跪了。果然SB2的性能还是有的，就是触摸板不行。

结论： **SB2玩游戏，虽然性能都在，但是不能用触摸板打，还是要用手柄**。跟Thinkpad的小红点比起来还是差太远了，毕竟小红点打[求生之路](#)、[使命召唤](#)还是[打人2016](#)我都试过，毫无问题。Thinkpad的小红点真是旷世之作，打败了世界上所有高性能笔记本。

Surface_Book_2_初体验_(3: 大结局)

最后的一个测试，就是把屏幕从筋肉基上面拔出来，然后重新跑一遍 [第一篇文章](#) 的脚本。我发现SB2在拔出来之后估计是会自动降频，因为本来65分钟的任务现在跑了105分钟，多花了60%的时间。不过这个电量很坚挺，居然最后还能剩下10%。

但是当我插回去的时候，电量马上回到了76%，可能筋肉基里面还有一个3倍大的电池。

既然SB2的屏幕拿来编译都可以撑2个小时，那插上筋肉基应该可以上一天班了。拔出来刷刷知乎应该也可以撑很久。

总的来说，十分满意。

作为21岁就开始在外企混的，我是不想改了，爱看不看。

<EOM>

P.S.

有些人不明白为什么我要广而告之，因为这样的人我已经干掉几十了，拉黑很麻烦的。

GacUI_动画系统_(1)

很久以前 GacUI 就有一个动画系统了，不过当时是为了让C++手写的Windows 7皮肤的按钮具有动画功能而做的。现在终于要实现用XML+Workflow来写动画了，所以那个架构不是很合适，于是我就开始重构。

重构的内容很简单。以前一个动画接口直接添加进 GuiWindow 里面，但是 XML 创建的对象也有可能是皮肤和控件，所以一个动画接口只能添加进 GuiInstanceRootObject 里。但是 timer 实际上还在 GuiWindow 里，于是就会有把控件从窗口里面那开之后，因为失去了 timer 要暂停，然后为了节省 timer 广播的资源，还要有一系列的其他动作等等。

这个设计其实很简单，从 GuiInstanceRootObject::AddAnimation 点进去，浏览一下，就差不多了。

总的来说就是，凡是有动画的 root 被添加进窗口，或者已经添加进窗口活着本身就是窗口的 root 被添加了第一个动画之后，就要挂个 callback 在 timer 里面。这个对象属于 root 和 timer 共享。

- 当动画播完了，或者 root 被拿走的时候，root 就往 callback 里面记一下然后脱离关系，下次 timer 调用 callback 的时候就会知道他已经完蛋了，然后自己从列表里删除。
- 如果窗口要被析构了，timer 会直接把 callback 干掉，callback 自然就会随着 root 的析构而洗头了。
- 如果一个窗口一个动画也没有，那么 timer 里面自然没有任何 callback，非常节省资源。

完美！

于是我写了一个 test case 试了一下，大概长下面这个样子。等整套动画做完了之后，这些东西最终将会出现在 BlackSkin 里面，还有 controls 里面多一个例子。在我计划中的动画，

- 有一部分可以用 XML 来完成（譬如说定义一个渐变的动画，没有理由要你写代码）
- 有一部分可以你自己通过 IGuiAnimation::CreateAnimation 来创建
- 最后一部分可以通过 coroutine 的方法把若干个小动画拼接起来。拼接的结果当然也是一个 IGuiAnimation，但是它可以被随时暂停和恢复，但是你写的时候只要定义动画 A 接在 B 和 C 后面，D 延迟多少同时执行这样的直观的函数调用。[coroutine 真是好用啊！](#)

最后来看第一个例子，现在才刚刚重构，所以暂时只支持 IGuiAnimation::CreateAnimation。这个函数有两个重载，分别用来做有限动画和无限动画。

```
<Resource>
<Instance name="MainWindowResource">
    <Instance ref.Class="demo::MainWindow" xmlns:demo="demo::*">
        <ref.Members>
            <![CDATA[
                @cpp:Private
                func CreateCircleAnimation(container: GuiBoundsComposition*, ball: GuiBoundsComposition*): IGuiAnimation^
                {
                    return IGuiAnimation::CreateAnimation(
                        func (time: UInt64): void
                        {
                            var circleTime = (cast double (time % (cast UInt64 1600))) / 1600;
                            var angle = circleTime * Math::Pi() * 2;
                            var sin = Math::Sin(angle);
                            var cos = Math::Cos(angle);

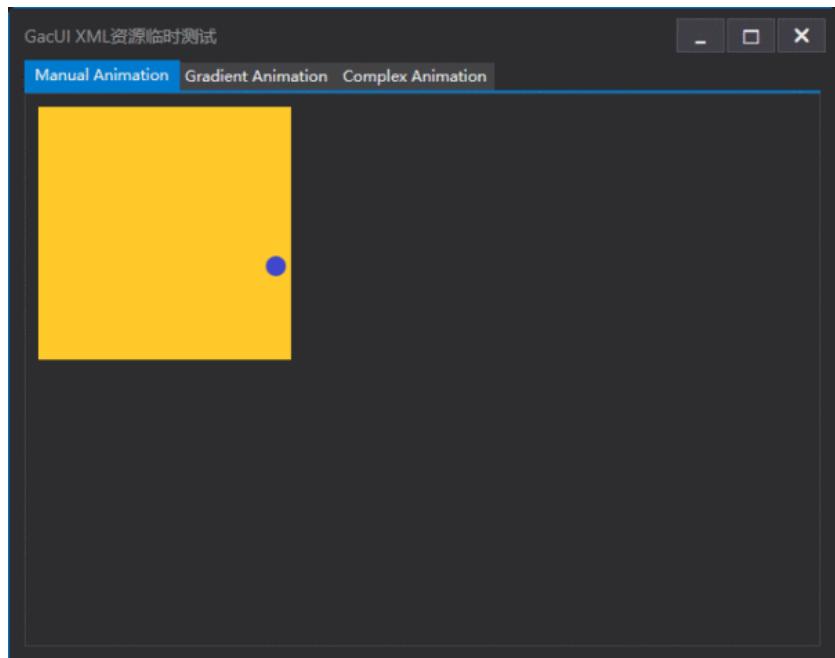
                            var cx = (container.Bounds.x2 - container.Bounds.x1) / 2;
                            var cy = (container.Bounds.y2 - container.Bounds.y1) / 2;
                            var radiusBall = (ball.Bounds.x2 - ball.Bounds.x1) / 2;
                            var radiusOrbit = Math::Min(cx, cy) - radiusBall;

                            var x = cast int Math::Round(cos * radiusOrbit + cx);
                            var y = cast int Math::Round(sin * radiusOrbit + cx);
                            ball.AlignmentToParent = {left:(x - radiusBall) top:(y - radiusBall) right:-1 bottom:-1};
                        });
                }
            ]]>
        </ref.Members>
        <ref.Ctor>
            <![CDATA[
                {
                    self.AddAnimation(CreateCircleAnimation(containerMA, ballMA));
                }
            ]]>
        </ref.Ctor>
    <Window ref.Name="self" Text="GacUI XML资源临时测试" ClientSize="x:640 y:480">
        <att.BoundsComposition-set PreferredMinSize="x:640 y:480"/>

        <Tab>
            <att.BoundsComposition-set AlignmentToParent="left:5 top:5 right:5 bottom:5"/>

            <att.Pages>
                <TabPage Text="Manual Animation">
                    <BoundsComposition ref.Name="containerMA" AlignmentToParent="left:10 top:10 right:-1 bottom:-1" PreferredMinSize="x:200 y:200">
                        <SolidBackground Color="#FFC929"/>
                    <BoundsComposition ref.Name="ballMA" PreferredMinSize="x:16 y:16">
                        <SolidBackground Shape="shapeType:Ellipse" Color="#3F48CC"/>
                    </BoundsComposition>
                </TabPage>
            <TabPage Text="Gradient Animation">
            </TabPage>
            <TabPage Text="Complex Animation">
            </TabPage>
        </att.Pages>
    </Tab>
    </Window>
</Instance>
</Instance>
</Resource>
```

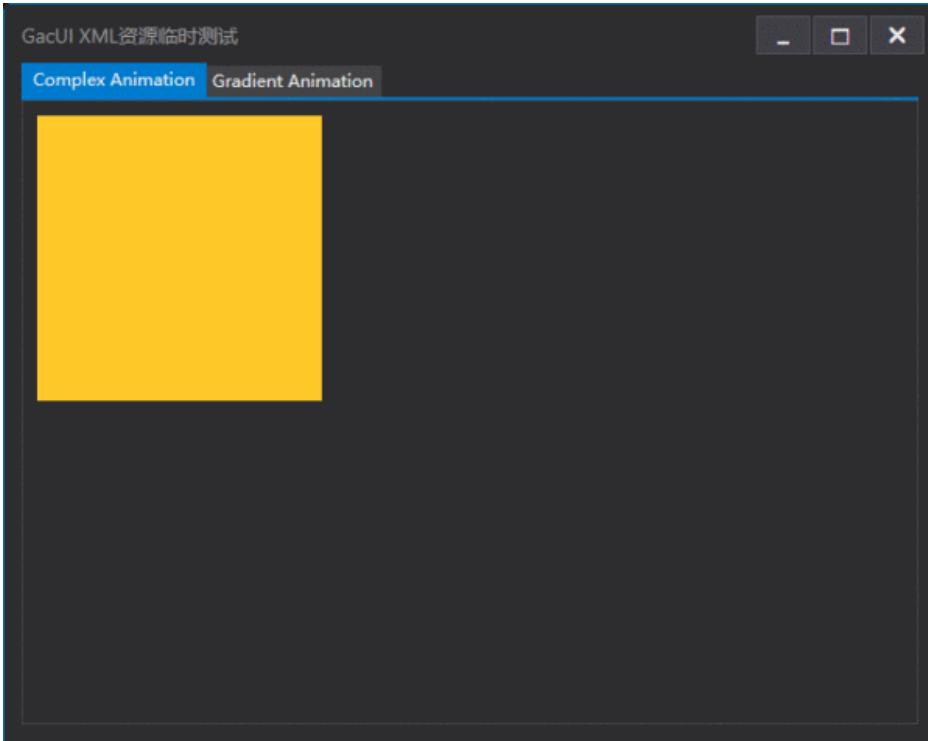
效果很简单：



接着马上就要实现coroutine动画了！

GacUI_动画系统_(2: 复杂序列动画)

先放效果图！



经过了几个小时的奋斗，这个序列动画终于做出来了，使用了coroutine+Workflow。重要的实现都在IGuiAnimationCoroutine里，可以在[GuiAnimation.h](#)和[GuiAnimation.cpp](#)看到。至于Workflow里面的coroutine是如何实现的，翻我专栏的置顶文章就有。

我来解释一下这个例子所用到的脚本。第一个当然是一个球在绕圈圈了。这个没什么好解释的，纯粹是一些简单的几何公式：

```
@cpp:Private
static func F(x: double): double
{
    var y = x * x * 2;
    return y;
}

@cpp:Private
static func BallAnimation(container: GuiBoundsComposition*, ball: GuiBoundsComposition*): IGuiAnimation^
{
    return IGuiAnimation::CreateAnimation(
        func (time: UInt64): void
        {
            var circleRatio = (cast double time) / 2000;
            if (circleRatio < 0.5)
            {
                circleRatio = F(circleRatio);
            }
            else
            {
                circleRatio = 1 - F(1 - circleRatio);
            }

            var angle = circleRatio * Math::Pi() * 2 + Math::Pi() * 1.5;
            var sin = Math::Sin(angle);
            var cos = Math::Cos(angle);

            var cx = (container.Bounds.x2 - container.Bounds.x1) / 2;
            var cy = (container.Bounds.y2 - container.Bounds.y1) / 2;
            var radiusBall = (ball.Bounds.x2 - ball.Bounds.x1) / 2;
            var radiusOrbit = Math::Min(cx, cy) - radiusBall;

            var x = cast int Math::Round(cos * radiusOrbit + cx);
            var y = cast int Math::Round(sin * radiusOrbit + cx);
            ball.AlignmentToParent = {left:(x - radiusBall) top:(y - radiusBall) right:-1 bottom:-1};

            if (not ball.Visible)
            {
                ball.Visible = true;
            }
        }, cast UInt64 2000);
}
```

第二个函数就是使用BallAnimation函数，来创造一个“在前面等一段时间然后接着转圈”的动画。这里开始使用Coroutine。我们可以很清楚地看到两步，第一步是等，第二步是播放转圈圈的动画。

```

@cpp:Private
static func BallAnimationWithDelay(container: GuiBoundsComposition*, ball: GuiBoundsComposition*, delay: int): IGuiAnimation^
${
    $Wait (cast UInt64 delay);
    $PlayAndWait BallAnimation(container, ball);
    ball.Visible = false;
}

```

第三段就是本篇文章的精髓了。这个函数虽然看起来比较长，不过前面的一半是在创建GUI对象，可以忽略。

```

@cpp:Private
static func WaitingAnimation(container: GuiBoundsComposition*): IGuiAnimation^
${
    var balls = {} of GuiBoundsComposition*[];
    for (i in range [0, 8))
    {
        var ball = new GuiBoundsComposition*();
        balls.Add(ball);

        ball.Visible = false;
        ball.PreferredMinSize = {x:16 y:16};

        var element = new SolidBackground^();
        element.Shape = {shapeType:Ellipse};
        element.Color = cast Color "#3F48CC";
        ball.OwnedElement = element;

        container.AddChild(ball);
    }

    while (true)
    {
        $Wait (cast UInt64 500);
        for (i in range [0, balls.Count))
        {
            $PlayInGroup BallAnimationWithDelay(container, balls[i], 150 * i), 0;
        }
        $WaitForGroup 0;
    }
}

```

在这里主要提一下这个死循环。死循环代表函数不会结束，那么这当然是一个无限长度的动画了。在每一个循环里面，我们先等500毫秒：

```
$Wait (cast UInt64 500);
```

然后一共8个球，每个球等 $150 * i$ 毫秒然后转圈：

```
$PlayInGroup BallAnimationWithDelay(container, balls[i], 150 * i), 0;
```

最后等着8个球的动画放完，这个循环才能结束。这里就是Coroutine需要解决的问题了，动画肯定会不断地播放，那这个函数到底是如何不断地被暂停然后继续执行的呢？要知道Workflow脚本最后都是编译成C++，而C++当然是没有这些高级功能的。欲知详情，请看本专栏置顶的文章。

```
$WaitForGroup 0;
```

这里的0指的就是\$PlayInGroup的第二个参数了。\$PlayInGroup不会等待动画播放完才往下走，所以这8个动画是并行执行的。但是最后我们还要等圈圈跑完一个流程才算完。

好了，我个人觉得这套表现为Coroutine的GacUI 动画API应该是相当容易理解的，我就接着继续做XML描述的渐变动画了。完整的例子可以看这里：<https://github.com/vcjh-libraries/GacUI/blob/75c997571ac5b378eab3aa612b0393ee87bba068/Test/GacUISrc/TestXml/Resources/Resource.xml>。

沈向洋： You_Are_What_You_Write，大家都要看

原文：<https://www.linkedin.com/pulse/you-what-write-harry-shum?from=timeline&isappinstalled=0>

Are Twitter, PowerPoint, Facebook, Instagram and texting eroding our ability to think?

There is a Chinese proverb that says “见文如见人,” which literally means “reading the document is the same as seeing the author.” If we are what we write, then who have we, as a society, become?

I was sitting in a technical review recently, listening to one of our reviewers grill the engineer who was presenting. Why did you choose that design? Why is the service showing bad results? How many users will switch to the solution?

The presenter’s answers lacked depth. It seemed like he hadn’t done enough rigorous thinking, the kind where you sit quietly, sift through research, contemplate options, determine what you know, don’t know and where more work is required. The kind of thinking I did as a young researcher when peers took me and my work apart when I took short cuts. Back then, I practiced a disciplined approach, spending hours just thinking, and even more hours on the hardest part—writing it down.

Today, long-form writing is being replaced. Tweets pass for dialogue. PowerPoint condenses thoughts to bullets. Words have been traded for emojis and GIFs. And we’ve become addicted to the noise. What happens in an Internet minute? 16 million text messages. 1.8 million snaps. 452,000 Tweets. 156 million emails. Who has time to think, let alone write?

And maybe we, in the technology industry, have shaped this reality. We created the phones, apps and 24/7-connected world. We’ve enabled society to put down the pen. The only writing I do today is email or quick WeChat posts.

So now I worry that we’re losing a valuable tool that helps us to think deeply, express who we are at our greatest and expand the intellect of those around us. And for us in the technical community, this is especially troubling. The stakes are higher than ever before with AI. We’re under enormous pressure to ship quickly, to achieve more, faster, but we can’t do this at the expense of the highest engineering quality. We have to think carefully about consequences and alternatives. Who gets blamed when a self-driving car hits someone? The engineer who wrote the code is the driver. Who is accountable for the AI algorithm with bias? The engineer who created the AI.

I see fewer engineers writing and sharing deep thinking, but this is what will lead to far more true innovation across the industry. How will we achieve the big transformative breakthroughs versus the incremental milestones?

By writing. Because the way to think is actually to write.

Putting pen to paper forces you to develop and refine your thinking by iterating, revising and exploring alternatives. Anyone who can think deeply can write beautiful code, inspiring papers or develop the plan to bring the next big thing to life. I encourage you to read Reid Hoffman’s Series B pitch for LinkedIn in which he shares the thinking that helped him succeed. At the time, he shares that a partner in a venture firm was exposed to around 5,000 pitches, looked more closely at 600 to 800, and did between 0 and 2 deals.

Writing offers the possibility to create lasting artifacts. I think of papers I published that endure, albeit perhaps as reference materials. Plenoptic Sampling. Lazy Snapping. Poisson Matting. These are my work’s contribution to the field of computer vision and graphics. They will survive me and, if I’m lucky, even help shape a mind or two.

One of my favorite professors at Carnegie Mellon, Takeo Kanade, said that you have to write research papers like detective novels. You need story, suspense, surprise and ‘aha’ to explain your ideas to peers, to inspire others to contribute and advance your work and the whole field.

Writing is an equalizer to get the best from the whole team. At Amazon, presentations are done with the six-page paper. Meetings kick off with everyone reading followed by comments and questions to the author. Everyone operates from the same context, and introverts, extroverts and non-native speakers have an equal chance to get their thinking across. It’s not about the presenter’s personality, but the words.

Ultimately, writing helps make you successful. You might be the smartest person with the best idea, but if you can’t communicate your thinking in a compelling way, you won’t get far. Two engineers in our AI+R team who inspire me with their regular writing habits are Bill Ramsey and Ronny Kohavi. Bill has written over 250 blog posts at Microsoft, benefitting our entire technical community. With Ronny, you don’t even need to meet him—his highly cited A/B test experimentation papers say it all, and he’s publishing for the benefit of the industry on LinkedIn.

As you’re reading this, you may be logging your objections: I need to drive results, so I need to go straight to code. I’m known for my code, so I don’t need to write papers. I’m not a native speaker, and I speak better with my code. I don’t know what to write about. I don’t have time... But please set them aside—for your own success, for your company’s, for the industry’s advancement—and start writing.

I see so many occasions for building long-form writing back into the engineering culture—planning documents, project proposals, technology LRP’s, review articles—to inspire us to work together, collectively creating and cultivating big ideas and big thinking.

I took a first step recently, writing a research paper with my colleagues Xiaodong He and Di Li. From Eliza to XiaoIce: Challenges and

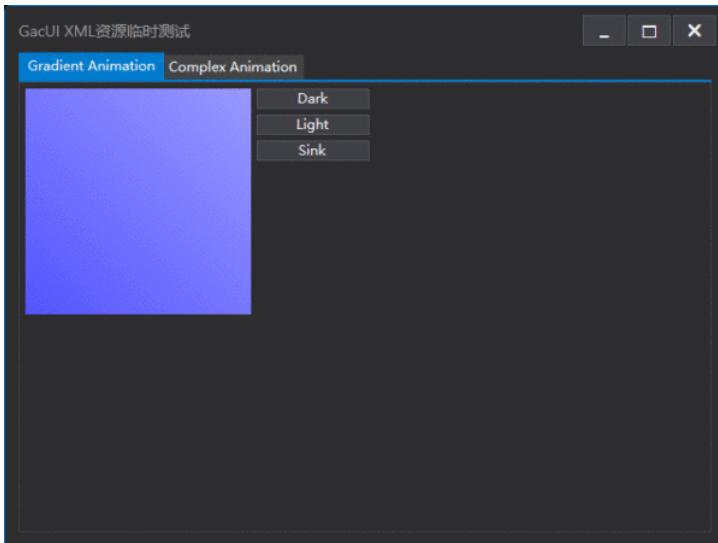
Opportunities with Social Chatbots, for the first time in years, so please no judgment, only constructive feedback!

I challenge everyone reading this piece to write 500 words per week. If you've got an idea or you see a problem, write your proposal and share it!

Let's rewrite our standards for thought leadership and engineering quality by writing more!

GacUI_动画系统_(3: 漂移动画)

惯例先上效果图：



这次写的基本上就是代码生成器。你用XML定义这个渐变动画，然后会生成Workflow脚本，最后Workflow脚本会由你执行脚本来生成C++代码。然后这些动画最终就使用C++来运行。后台那个转圈圈的菊花动画还在跑，就算是虚拟执行Workflow脚本CPU占用率也<0.5%，使用C++的话就更低更快了。

首先，定义你需要的颜色和一些其他数据的数据结构。这个结构必须是一个带有默认构造函数的class，而它的属性都得是struct：

```
class ColorDef
{
    prop Top : Color = cast Color "#000000" {}
    prop Bottom : Color = cast Color "#000000" {}
    prop Shadow : Color = cast Color "#000000A0" {}
    prop Thickness : int = 0 {}

    static func Dark() : ColorDef^
    {
        var def = new ColorDef^();
        def.Top = cast Color "#9999FF";
        def.Bottom = cast Color "#5555FF";
        def.Thickness = 0;
        return def;
    }

    static func Light() : ColorDef^
    {
        var def = new ColorDef^();
        def.Top = cast Color "#DDDDFF";
        def.Bottom = cast Color "#9999FF";
        def.Thickness = 0;
        return def;
    }

    static func Sink() : ColorDef^
    {
        var def = new ColorDef^();
        def.Top = cast Color "#5555FF";
        def.Bottom = cast Color "#0000FF";
        def.Thickness = 10;
        return def;
    }
}
```

然后用XML定义一个动画类demo:ColorAnimation

```
<Animation name="ColorAnimation">
    <Gradient ref.Class="demo::ColorAnimation" Type="demo::ColorDef">
        <Interpolation>
            <![CDATA[G]]>
        </Interpolation>
        <Targets>
            <Target Name="Top"/>
            <Target Name="Bottom"/>
            <Target Name="Shadow"/>
            <Target Name="Thickness">
                <Interpolation>
                    <![CDATA[ [$1] ]]>
                </Interpolation>
            </Target>
        </Targets>
    </Gradient>
</Animation>
```

生成的demo:ColorAnimation会包含有Current属性和CreateAnimation函数。其中Current属性的类型就是你指定的demo:ColorDef。

那么接下来，我们要做一个UI，然后把外观的属性绑定到这个类的Current属性里面的其他属性上去。在此之前我们要在窗口里面new一个ColorAnimation^放进成员变量里，这样才有东西可绑：

```
<Instance name="MainWindowResource">
    <Instance ref.Class="demo::MainWindow" xmlns:demo="demo::*">
        <ref.Members>
            <![CDATA[
                @cpp:Private
                prop GradientColorDef : ColorAnimation^ = new ColorAnimation^(ColorDef::Dark()) {const, not observe}

                func PerformGradientAnimation(target: ColorDef^): void
                {
                    AddAnimation(GradientColorDef.CreateAnimation(target, (cast UInt64 500)));
                }
            ]]>
        </ref.Members>
    </Instance>
</Instance>
```

```

    var counter : int = 0;
]]>
</ref.Members>
<Window ref.Name="self" Text="GacUI XML资源临时测试" ClientSize="x:640 y:480">
<att.BoundsComposition-set PreferredMinSize="x:640 y:480"/>

<Tab>
<att.BoundsComposition-set AlignmentToParent="left:5 top:5 right:5 bottom:5"/>

<att.Pages>
<TabPage Text="Gradient Animation">
<Table AlignmentToParent="left:0 top:0 right:0 bottom:0" CellPadding="5">
...
<Cell Site="row:0 column:0 rowSpan:4">
<Bounds PreferredMinSize="x:200 y:200">
<GradientBackground Direction="Slash" Color1-bind="self.GradientColorDef.Current.Top" Color2-bind="self.GradientColorDef.Current.Bottom"/>
<Bounds AlignmentToParent="left:0 top:0 right:0 bottom:0">
<InnerShadow Color-bind="self.GradientColorDef.Current.Shadow" Thickness-bind="self.GradientColorDef.Current.Thickness"/>
</Bounds>
</Bounds>
</Cell>
...
</Table>
</TabPage>
...
</att.Pages>
</Tab>
</Window>
</Instance>
</Instance>

```

剩下的，我们只要放上三个按钮，然后使用ColorDef里面预先定义好的配色（写在了静态成员函数里），调用GacUI提供的API，加上ColorAnimation的CreateAnimation函数就可以了。这个函数的内容就在PerformGradientAnimation里面。按钮怎么调用它我就不说了，用过GacUI的人都知道。

好了，现在动画已经做完了，只要把剩下的可以把正在运行的动画取消这一个给做了，GacUI对动画的支持就差不多到这里了。几乎所有动画都可以用这三篇文章的内容，加上熟练掌握的中学几何知识来实现。

GacUI的XML基本上还是用来定义和实现GUI的，至于一些花俏的设计功能，还是得做进未来的GacStudio里。

GacUI_1.0_眼看着就要写完了

心急的看 [TODO.md](#) 就好了。现在剩下的就是一些小修小补的事情了，譬如说

GacUI 1.0

- 修现在黑皮肤窗口几个按钮没有响应属性变化的问题
- 做一套新皮肤，运行时换肤
- 菜单项可以被绑定到列表上
- 富文本框支持RTF/HTML剪贴板格式
- 国际化/本地化支持
- 给所有列表控件加上一个给ItemTemplate用的“上下文”参数
- 让XML写的窗口和控件可以继承
- 把Visual State做的跟XAML一样好，简单来说就是你大幅度改变窗口大小的时候，控件可以重新排列，甚至连图片都可以换掉等这些事情。其实本来很简单的，不知道做手机app那帮人为什么要搞得鸡飞狗跳。

说来这个皮肤的事情，我找了一个老乡给我做个设计。不过这个设计主要是用来炫耀GacUI的各种高级功能的，你们可能多半要根据自己的要求，在我这两套皮肤里面学习怎么做你自己的皮肤（逃

还有剩下的一些边角料，主要是把代码写得更好。

ParserGen

GacUI 1.0做完之后，就要开始为IDE写一套新的parser生成器了。除了正常的GLR支持以外，还要对代码的格式有支持，修改语法树或者修改代码的时候要互相同步，还要给出足够的hint来把一些类似智能提示啊、重构的这些事情，无论从算法上还是多线程上都简单化。估计要几个月。

GacStudio

ParserGen做完之后当然就是GacStudio，除了正常的拖拉控件以外，我准备把Tutorial也写进GacStudio里面，你可以再把玩的时候开启教学功能，或者直接按照目标来告诉你你应该怎么做等等。GacUI具有丰富的排版和脚本功能，大部分常用的部分我也会提供一套UI来替你编辑XML和Workflow，而不是让你干什么都得自己亲自写XML和Workflow。

新的ParserGen主要就是在XML和Workflow的全部或者局部的编辑上，最后你还可以自己debug你自己的脚本。Debugger的窗口本身也会随着GacUI一起发布，你们也可以直接把它嵌入到你们自己的程序里面，如果你们要做一些二次开发的东西的话，就像Office的VBA。

GacJS

GacJS的唯一目的就是想把相同的XML和Workflow都能跑在浏览器里，并没有什么共享代码的事情。如何做一个跨平台的UI，你们自己决定，我不提供死方法。做出来了之后我就把网站用GacJS重写一遍，现在的网站就扔了。

完善其他平台的移植工作

目前其他平台的移植工作是别人做的，他们不是很积极，我现在也没有空，所以只能等上面的事情都做完，如果他们还没完成的话，我就自己来。

GacUI 2.0

昨晚所有的这些东西，GacUI 2.0当初设下的目标就完成了。后续除了你们有什么需求我维护之外，如果人类的UI技术没有什么显著发展好让我抄的话，我就继续尝试其他领域的东西了。

这一套下来估计也要有一段时日吧。

重启《GacUI的设计与演化》

几年前挖过两个坑，一个是GacUI在segmentfault上的教程，另外一个是GacUI与设计模式。教程当然只能等1.0做完之后再写了，这次主要说的是后面那个。本来是想借着开发GacUI的过程，给大家讲一讲设计模式的知识的。后来写了一半坑了，坑的时候刚好在给GacUI添加大量的新东西。后来想想这坑的也是好，因为GacUI的设计经历了一次大规模的变动，就算写了这个文章，也要作废了（逃

如果关心GacUI和我的专栏的，应该都知道之前发生的这个事情，我就不再提了。之前看了沈大侠的文章，又让我记起了这个没填的坑，于是我就想把它写完。当然大家不要等，我也不会马上写，写的过程也需要很久，一篇一篇慢慢发也要看心情的。不过这次题目我把它定为《GacUI的设计与演化》，一方面是剽窃了BS大爷的idea，另一方面也可以写一下GacUI这七年来的变化。

大家看各种架构和设计模式的书和文章估计也很多了，不过谈演化的我很少见到。虽然GacUI早期的设计也有很多傻逼的地方，但是让大家看看GacUI是如何一步一步变化到今天这个样子，应该也能学到很多东西。

为什么我要在这个时候来说呢？因为GacUI已经临近尾声了，架构不会改了，就一直这样了。在开发之余换换口味写写文章，也是不断推进进度的一种办法。最近玩游戏玩的也比较多，应该收敛一下。

这里主要提一下GacUI更早以前的历史。开发UI库的习惯由来已久了。第一次尝试是高中的时候给自己的ARPG做UI库和脚本引擎，然后后面就慢慢变化，写了好几遍，到了[这里](#)终于有一个超级简单的雏形了。那个时候还不懂得怎么开发UI的layout的系统，于是弄了个废物出来。后面当然发现不行就坑掉了。那个时候才刚在微软入职，已经是9年前的事情了。不过使用矢量图来绘制UI的感觉还是体会到了。

后来我觉得还是应该学习WPF，于是就粗略看了一遍，然后做下一个UI库。[这个UI库就有一些简化过的dependency property的想法](#)，然后实现了一些简单的控件和排版功能，最后发现C++模拟这个需要浪费的计算力实在是太大了，于是又GG了。

因此在2011年，我把10年前封装[Windows API的那套UI库](#)的基础部分拿了出来，在上面加一层interface，把跨平台的部分和平台相关的部分分开，[然后开始写GacUI](#)。之前积累了几年的经验教训，让我粗略的到了一个正确的设计，因此就慢慢演变到今天。[这个地方](#)还有我早期开发GacUI的笔记。

《GacUI的设计与演化》会从一开始的想法开始写，不过早期的代码有些已经找不到了，因为我是到了工作之后才用TFS的，所以估计也记不清多少了。但是GacUI的所有历史都在，包括codeplex上面的都迁移到github的历史仓库里了。我可以慢慢看，回忆一下以前的想法，然后写进文章里。

文章估计会用markdown写在github上，每篇改好了我就发到知乎来。segmentfault看看他到美帝的那个傻逼延迟到第能不能解决，不能我就不用了。这次不知道会不会坑，不过反正也是个长期的事情，看缘分了（逃

一个图可以看出知乎有什么问题？

其实我不是说为什么编程问题赞这么少（38已经突破天际了（逃）。可乐的那个答案是我在西雅图今天晚饭时间回答的。所以可见，“18:00”应该是对的，然而这个“昨天”该怎么理解呢？我只能认为，知乎计算“昨天”的时候，用的是西雅图的本地时间跟服务器的本地时间比，得出了“昨天”。

因此我猜这个代码是这么写的：

1. 提交答案的时候，服务器采用了自己所在地区的时间，转成了UTC保存了下来。不过中国没有夏令时，也没有时区的区别，知乎的码农如果觉得自己的服务器一辈子都不可能搬去国外的话，直接保存北京时间也是有可能的。
2. 显示答案的时候，不知道是服务器那边计算的，还是浏览器这边计算的，总之把UTC的时间换了回来。
3. 计算出“昨天”的这个代码，有两个猜测：
 1. 如果第二条的时间是浏览器计算的，那么“昨天”也是在浏览器里面计算的。但是浏览器到底用这个时间跟谁比呢？一般都会跟“现在”比。但是我猜“现在”在代码里面应该是模板里面的一个空，服务器用自己的本地时间，把这个空填上了，因此造成了浏览器得以用西雅图的本地时间与服务器的本地时间相比，得到了“昨天。”。
 2. 如果第二条的时间是在服务器里计算的，那么“昨天”应该也是在服务器里面计算的。服务器知道目标浏览器的时区，于是贴心的帮我把时间转成了西雅图本地时间。但是转完了之后估计那个码农就忘记了，拿西雅图的本地时间，和服务器的本地时间比了一下，得到“昨天”。
3. 个人认为第二个猜测可能性比较大，第一个脑洞有点大。

不知道我猜的对不对（逃

美帝感冒见闻

这几天偶得感冒，我猜可能是由于知乎用户不相信的原因导致免疫力下降，结果刚好暖气坏了叫人来修，所以关了一天暖气结果着凉了。后面就沿着喉咙发炎 -> 发烧 -> 咳嗽 -> 流鼻涕的既定路线走下去。健身课都打电话cancel了两节，班也在家上。第一天体温将近38℃，但是由于我本来身体的体温就比正常人的情况要低0.5℃，我也不知道为什么，所以这应该相当于别人的38.5℃，在国内再高点肯定就一定要去北京的三甲医院输液打医生了。

微软对员工生病这种事情反应比较大，有点事情就让你滚不要来影响到别人，所以我发了个邮件就可以呆在家里了。本来还想着都发烧了要不要去医院，但是美帝医院的尿性就是一次要约到至少2-4个星期后的，到那个时候病早就好了。微软为了关怀自己的员工，自己外包了家要工卡才能看病的小诊所。但是由于公司总部的人比所在城市的居民数量还要多，其实也好不了多少。于是我就叫老婆下班的时候去买药。

药房的人听到了我这个情况，说这点事吃什么药，喝热水等死，哦不，等它好就行了。谁不知道感冒大多都可以自己好啊，吃个药还不是为了让自己过得舒服一点，所以最后按照推荐买了美帝版本的日夜百服宁，就是图片中的那个。NyQuil明显是NightQuil，但是Night太长了。不过我不知道为什么Day不干脆写成Dy，模仿一下中部某些地方的口音，多对称呀，就像C++左右大括号都一定要换行才好看一样。如果不换行，那我建议左大括号左边不换行右边换行，右大括号右边不换行左边换行，也很对称。

这让我想起小时候身体一直都不是很好，经常生病。高中中有一次中了病毒性感冒，在家里躺了2个星期，体重掉了20斤。这次也是，虽然只持续了两天低烧，但是体重也狂掉，于是我想着这样也不是办法，中午就出去吃点高热量的东西。点了一角cheese cake之余，看到有个菜花居然超过1000卡，我凭着好奇就要了一份。结果果然是一大颗菜花，和目测约有1000卡的沙拉酱。沙拉酱没碰，就靠那角cheese cake居然到了晚上一直都不饿，于是弄了点poke对付了一下，我觉得今天又等于没吃饭了。

自从我离开汕头之后，每走远一点，身体素质就会得到提高。刚开始到北京的时候也是。那个时候北京的空气真是太他妈好了，根本想不到从2012年开始就会有严重的雾霾蔓延。不得不说那个雾霾真是有毒，2013年连中两招，为了避免看病难问题，我跑去非三甲医院输了几瓶液，终于搞好了。于是就来了西雅图，好的环境就是厉害，连这些鸡毛蒜皮的小病，也到了这几天才终于来了一次。

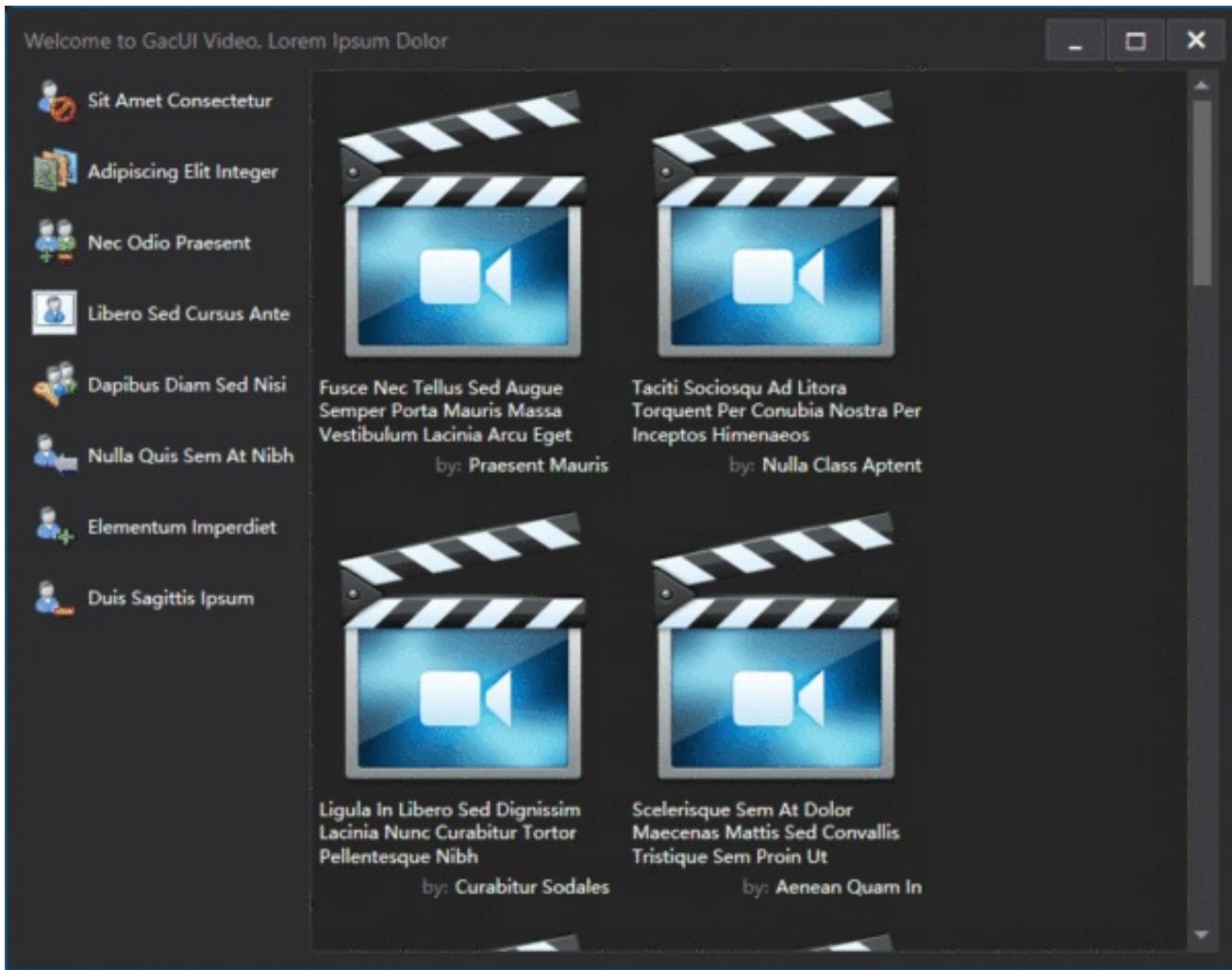
想到公司每年都有两次大规模流感爆发，大量员工集体work from home。我要是这两天去上班的话，按照他们那个身体素质，应该可以引发第三次。

刚才Surface_Book_2突然屏幕暗了，启动不了，键盘也没有反应了。

于是我果断同时抚摸 Power 和 Volume+ 长达20秒，进了UEFI，选择重启，就好了。笔记本有个强制重启的按钮很重要。

GacUI_响应式布局

效果如图：



结构比较简单。GacUI提供了如下4种响应式布局的元素：

- View: 你直接设定每一个布局的具体的样子
- Stack: 这个布局元素里面的布局元素会按他们的大小依次变化
- Group: 这个布局元素里面的其他布局元素会同时变化
- Fixed: 这个布局里面的元素不随着上级的要求而变化

最后给一个Container，会根据自己的尺寸来调用装在里面的布局元素的LevelDown或者LevelUp函数，从而实现自动变化。

所以这个例子写法就很简单：

最外面是一个View，View里面的两个子布局分别是Group和Fixed，代表了有左边的按钮的样子，还有没有左边但是有菜单的按钮的样子。Group里面装着一排左边的按钮以及右边的列表，而每一个按钮里面还是一个View，View里面的两个子布局分别是有文字和没有文字的时候的样子。

因此当窗口被缩小的时候，首先Container会问最外层的View能不能继续缩，而这个时候View会问Group能不能继续缩，Group查看了每一个按钮发现他们可以去掉文字，因此成功缩小一级。然后窗口再缩小，Group发现按钮已经没有办法继续变小了，于是通知View，View觉得第一层View缩小到这里已经差不多了，于是切换到第二个View——带菜单的模式。

真是容易啊！

后面很快就出一个Release，然后把globalization和localization的文字支持工作做了，接着就可以用这套响应式布局的工具来实现Ribbon了。等Ribbon做好了，我就把GacUI_Controls/DocumentEditor的菜单都改成Ribbon。

距离GacUI 1.0又近了一步=_=

GacUI终于获得第一个愿意跟我沟通的商业的case了

以前一直都有很多人在使用GacUI的过程中遇到了问题来咨询我，不过他们获得了答案之后就不理我了。所以我也想知道他们到底用的怎么样。这次这个case的区别，就是终于肯让我track他的进度了（逃

当然人家都还没做好我就不透露太多了，反正是面向海外客户的，你们多半以后也看不见。一开始本着负责任的态度，我向他推荐了XAML，他说不要，一定要用C++，而且看着我这个东西编译出来体积小（2M）没有依赖，所以QT和UWP也不想用，一定要试试GacUI。既然人家一定要试试，还质问我为什么开发了GacUI还要向他推荐别的，由于我[Before 1.0](#)的事情做完之后才会开始写文档，所以只能自己来做人肉文档了。在GacUI的Telegram的人应该目睹了这起事件的发生，后来就转到了微信上。

这个人自称十几年前是个Java程序员，现在当老板了，但是还是做一些investigation的工作，等prototype写出来了就让手下的人去改成产品。虽然在写代码的过程中，我发现人家有大量的一般程序员所拥有的缺点，不过既然人家自称已经不是程序员了，那我也就不好说什么。本来GacUI也是设计给普通程序员用的，应该满足他们的需求。

GacUI抄袭了XAML的先进思想，把很多使用过程中大家抱怨的东西顺便也改掉了，而且做成了更合适C++使用的样子。结果竟然是因为体积小而得到了别人的兴趣。世界真奇妙。

随着咨询的进一步深入，我发现对方的知识基本停留在MFC的时代，而且它告诉我业内很多人写MFC/QT的GUI也是hardcode了按钮大小之后根据窗口的变化人肉调整位置的。我听了都惊呆了，XAML都出来20年了现在人们还在做这种原始的事情。我觉得我有点曲高和寡自绝于人民的感觉。

设计GacUI的时候我有意让直接设置坐标的几个函数显得不那么明显，所以我没告诉他还有这个功能，逼他全部用布局来做。于是有了如下几个发现：

- 存在一些人是完全不介意在XML里面使用脚本的（在他们得知会生成C++代码之后），而且似乎总是尽量想再XML里面写脚本。
- 我改良的这套可以写任意表达式的data binding，每个人都说用的超级爽，从来没有在其他UI框架上体验过。
- 只要他们发现MVVM是他们唯一的solution，那他们就会愿意花时间学习MVVM。
- 只要他们不知道怎么设置坐标，最后还是会愿意学习Layout，而且一定会觉得有Layout更好。

其中一个例子是，这个人想用GacUI做异质的TreeView（也就是每个item长的不一样），GacUI里面最自然的方法当然是用一颗异质的树绑定到BindableTreeView上面去，最后写ItemTemplate来解决。因为你要使用data binding，你还要做这个TreeView，自然你就只能用MVVM的这套方法了。经过了痛苦的学习，他终于学会了这个跟GacUI抄袭了XAML的、20年前一直领先到现在的先进设计。

做前端的人还是要好好改良他们的工具。到了现在JS的observing系列feature还没定稿，导致data binding和MVVM没有形成统一标准，最后Layout做UI的时候还那么几把复杂（我承认HTML那套东西写起文章来还是挺舒服的，但是做UI特别别扭）。什么时候才能全盘抄去呢？20年过去了还这样，难道要30年？

本来一直很抗拒MVVM，最后还是把整个程序都改成了MVVM的样子，不过看他的编程思想过于陈旧，我没有要求他必须使用纯粹的方法来做，还是告诉他GacUI留了很多可以给你hack的地方，是用来应付变态需求的。最后他告诉我，经过了这几个星期的折腾，他深刻地领会到了**UI与逻辑分离的好处**，决定要把他们公司的其他app也全部改成GacUI，人肉Layout的部分说是占用了他们超过了50%的UI部分的代码，剩下的一大半可以用几行data binding代替，不能再这样继续傻下去了。

这个人进来的时候天时地利人和具备，刚好几个月前GacUI经历了一次超级大重构，设计应该是不会改了。要是他早点来，说不定后面还得再折腾一下（逃

虽然我的License很宽松，不过人家觉得既然他写的软件要卖钱，那就得支持我一下才行。经过了讨论，我获得了把他的代码的GacUI XML和图标的部分抄来当demo的许可（因为我的License认为这部分代码开发者说了算我默认没有权利使用，跟GPL系列恶魔License有着本质的区别），最后还要写篇文章给我吹吹牛，我把他的东西以后放进网站里做case studying。他们发现的bug要全数上报，如果他们自己修了也得让我知道我合并进master。

不过这个过程中居然只发现了两个bug，果然慢工出细活（逃

总的来说，**GacUI获得了普通程序员的肯定，比那些高手说GacUI牛逼要让我开心一个数量级**。因为这是我当初开发GacUI的时候，给自己定下几个目标里面最难达到的——用得爽。牛逼的人如果说GacUI好，多半还是出于他们自己所坚守的开发哲学跟GacUI相匹配。这一点我并不关心，反正每个人都有自己的想法，恰好跟我一致也不能说明什么问题。而普通程序员说GacUI好，才是出于感性的、直觉上的体验。

[@邹欣](#)在上他的软件工程课的时候，要求学生做的作业要发到网上去让大家下载使用，从而获得反馈。其实开发GacUI也是一样，一直以来人们问完问题就跑了，这样也是不行的。难得有个人愿意持续使用和反馈，我觉得是个良好的开始。

决定入驻知识星球

知识星球的投资人催了我一年了，我想想还是加入了（逃。受到上一篇文章说的事情的启发，我觉得这是一个很好的机会。

以后我会把《GacUI设计与演化》和其他相关内容发在里面。如果我用了几次发现不好用，那我就全数退款关掉。如果知识星球不支持退款，那就给你们个微信账号，转账完拜拜。

至于所有文章的内容，**我决定让好好使用GacUI的人免费获取**。你们来了以后，给我一个github的repo，证明你真的在用，我会指导你们怎么合理使用GacUI，算是前期社区没起来之前做的一些义务劳动。如果你们最终把一个有一点点价值的程序写出来了，**我就把钱退给你们**。牛逼的那些，我就放进主页当case study，在大家使用的过程中，我还可以找找bug，双赢也。

这个系列发完之后，我看心情发别的。他们的dev已经承诺说要支持markdown了，我觉得很好，比知乎专栏强多了。这样我就可以在<http://visualstudio.com>里面写完，然后贴进来。

现在我什么都没发，所以定在了3000块，你们不要进来。没有二维码应该找不到，万一被你们找到了想加我也不通过的。以后我会设置一个合适的门槛，然后写文章通知。至于第一篇文章什么时候发布，我也不知道，心情好的时候就来写。

希望到时候所有人都能获得退款，这才是我加群收费的真正目的（逃

想戒除焦虑吗？想变得快乐吗？

文章列表

1. [vczh: 如何处理找对象的焦虑](#)
2. [vczh: 成年人如何处理来自父母的管束焦虑](#)
3. [vczh: 如何处理经济造成的焦虑](#)
4. [vczh: 如何理解戒除焦虑和积极行动的矛盾](#)
5. [如何在贩卖焦虑的世界中保持清醒](#)
6. [焦虑与攀比的内在关系](#)
7. 待续

前言

在知乎上摸爬滚打了这么久，很多热门问题其实都围绕着焦虑和制造焦虑。我也看到大量的人陷于焦虑之中不能自拔而不自知。我决定向大家贡献一点人生的经验。每个人都可以追求成功，但是就算你没追求到，你也可以有快乐的、充实的、没有焦虑的生活。而且，人的一生并没有什么事情是必须要去完成的，如果你的梦想难度太大，你可以换（逃

焦虑从何而来

大家千万不要相信焦虑是什么成功学啊大V贩卖智商税而创造的，这些也是他们让你们相信从而制造焦虑的一种手段。回想一下，当你们开始有焦虑的时候，是什么时候？

没错！就是初中快毕业的时候了。我国每个人的第一份重要的焦虑，基本都来自于中学老师和你们的父母。差不多中考的时候，中学生已经分为两种人了，一种是考得上高中的，另一种是考不上高中的。考不上高中的，每天都要被骂。而考得上高中的，马上就要迎来高考。想想，到底是谁总是不断的告诉你们高考很重要的呢？显然作为一个高中生，绝大多数人是无法自发理解高考是多么重要的。

但是我在那里并不是要反高考，因为高考的确很重要。但是这种教育的方式，已经在我们的心目中埋下了焦虑的种子。原则上来说，告诉你们高考很重要，有两种方法。第一种方法，就是好好跟你说，高考是如何影响你后面的道路的，万一考不上你又有什么路可以走，要怎么挽回。第二种方法，就是直接跟你说，考不上你这一辈子就完了，考上了你就再也不需要努力学习了。显然大多数父母和老师都是这么跟你们说的。

高考砸了就真的完了吗？显然不会。而高考考得好就成功了吗？显然也没有。你的人生还很长，你还需要做大量的事情来让生活随着你的希望而前进。但是成功什么的不说，焦虑你反正是先得到了。几乎所有人都认为高考没发挥好就完了，这正是群体焦虑的一部分，形同被洗脑。如果你还想要完成你的愿望的话，你就得冷静下来好好思考。而如果你想冷静，那你就必须得不被焦虑影响你的判断。有焦虑并不是问题，但是焦虑控制了你，你成为了焦虑的奴隶，这就是问题了。

显然，焦虑是一个我国人民普遍存在的一个现象，而这个现象的根基是稳固的，还普及地很成功。在互联网开始之前，大家早就焦虑了，根本不需要成功学和大V来推波助澜。如果你只是为了让你舒服一点，而找个理由来解释你的焦虑，然后控诉别人的话，你的心态很不健康。

为什么焦虑不好

焦虑当然也是有两面性的，适当的焦虑也可以成为你的动力，不好的只是被焦虑控制。被焦虑控制你就会不淡定，失去冷静思考的能力，就很容易上当受骗。所以我们需要想办法来让焦虑下降到一个健康的水平，让焦虑变成好的东西。

人活着是为了什么

每个人对这个问题都有自己的看法，但是总有一个事情是不会变化的，人活着当然是为了获得快乐。哪怕是那些一辈子勤勤恳恳付出的人也不例外。为什么有些人一直在帮助他人？帮助他人对被帮助的人来说当然是一件好事，但是如果这个人觉得做好事很痛苦，那自然是不会去做的。正是因为做了好事会让他们快乐，这才成为了他们一辈子帮助他人的内心的动力。当然有些人是不会从这些事情上获得快乐的，譬如我（逃

每个人获得快乐的方法是不一样的，每个人追求快乐的方法也是不一样的。人焦虑自然也是因为无法获得足够的快乐。想想，每个人都有已经获得的东西，和想要去拥有的东西。前者带给你快乐，而后者带给你焦虑。焦虑是一定会存在的。如果你的快乐打败了你的焦虑，那焦虑也就成为了丰富生活的一道开胃菜而已。但是如果你的焦虑打败了你的快乐，那你的人生就真的充满焦虑，被焦虑控制，进而成为了焦虑的奴隶。

在这里，我觉得有一点需要强调的是。我并没有打算让大家放弃自己的梦想。有梦想当然会有焦虑，哪怕是我开发GacUI看着TODO列表那么长我也会有焦虑，但是显然这种焦虑是不能打败我的。梦想是要追求的，你追求梦想也获得了焦虑，然而并没有什么道理你必须被这种焦虑充满。

所以为了降低焦虑，甚至让焦虑成为无足轻重的一部分，你应该去追求快乐，从而抵抗焦虑。

怎么追求快乐呢

首先需要指出，快乐当然是不能以吸毒的方式去获得的。这里并不是真的指毒品，而泛指所有临时会让你麻木的、临时的活动。在这里我举个简单的例子：买东西。你有了钱，你当然会想买东西，而有些时候你可能控制不住报复性地购买你以前得不到的东西。这包括男人的汽车，女人的衣服包包鞋子腰带等等。拥有这些东西你当然会觉得快乐。但是，你觉得买了一辆BMW很爽，买第二辆一样的车你就会觉得一样爽吗？显然是不会的。你有了一个2.55你很开心，再买一个2.55，你可能没有感觉了。一旦你试图从买东西中获得快乐，你的欲望会膨胀的特别厉害，这就像吸毒。成本不可控制，只会让你陷入更大的焦虑。

而且只是几千几万花出去说不定还可以刹车，但可惜社会上的宣传，让你们还想买房子，这就不得了了。你好不容易买了一座小的，以后可能想换大的，或者换个好的学区。我认为如果你第一套房子就已经让父母和你自己都倾家荡产的话，后面基本不要想买更好的房子了。于是这个愿望只膨胀了一级，就马上让你受不了了。你的快乐只持续了一小段时间，伴随而来的是这辈子无法摆脱的焦虑。

当然，我也并没有让你不要买房子。你可以有一万条理由买房子，但是千万不要期望从里面得到快乐。买房子可能只是你人生中的计划的一个步骤，譬如说你喜欢的女人他父母就是特别爱房子，你为了不放弃爱情，你可以去买。但是你买房子是为了收获一个老婆，所以这种时候一个健康的结局，是你收获了老婆让你快乐，而不是买这套房子让你快乐。

你在战术上重视买房子，战略上轻视买房子，房子就不会给你带来焦虑，也不会让你陷入吸毒式的快感里。如果在你的价值观里面，买房子并不增加你的价值，这个时候就最健康，最不会焦虑了。显然你是通过花你自己的钱去买的房子，这是一个等价交换，本来就是不会增加你的价值的。但是如果你从内心认同这一个观点的话，那买房子从一开始就会变成一个手段，而不是你的目标。不能完成目标会带给你焦虑，不能完成一个你认为很重要的目标会给你带来巨大的焦虑。但是——如果你为了解决一个问题，使用买房作为一个手段而失败了（买不起也好，买了之后发现没用也好），这并不会增加你的焦虑，因为你还有很多种其他手段可以尝试。

做同样的事情，心态很重要，不同的心态会给你带来不同的感觉。

什么样的快乐才是健康可持续的呢

这个问题其实很容易解答。这个世界上有很多东西，不仅做了会给你带来快乐，而且还会提高你的阈值，你只要隔一段时间再做一次，就会给你带来相同的快乐。追求这个事情，就很健康。因为你的成本是可以控制的，只要你觉得一切都可以控制，自然你的焦虑感就降低了，而且你的快乐还会进一步冲淡你的焦虑，让你的人生走上正轨。

对于有些人来说，吃好吃的东西可以给他带来快乐。我就是这样的一种人，只可惜吃也会带来肥胖，但是我惊喜地发现，健身业会给我带来快乐，命真是太好了（逃。打游戏也可以给某些人带来快乐，跟老婆玩耍也可以给某些人带来快乐，甚至是每天下下棋，也可以给某些人带来快乐。

你还可以有一些更高级的快乐，就是你不仅觉得做这件事情本身就很爽，而且还会给你带来正面效益。譬如对于我来说，编程就是这样的一个事情。我沉迷于编程，正是因为编程本身给我带来快乐。而我从来没有去追求编程可以带给我什么，结果发现他妈的竟然能赚钱，命简直太好了。对于我而言，编程就像大家打篮球打游戏一样，就是每天做一做觉得很开心的事情。如果你有一个可以给你带来经济效益的快乐，那每一天过的简直不能再舒服了。

你还可以有更加高级的快乐，譬如说一些可以给你带来社会地位和威望的。比如说，喜欢当义工，你不仅得到了快乐，别人也得到了快乐，大家还觉得这样的人了不起。物质上的肯定显然不如精神上的肯定，那你得到的快乐，要比你发现你的快乐能赚钱，还要多一个数量级。简直就是跟神仙一样爽了。那这个时候如果你还有一些愿望难以满足，有一点焦虑，又怎么样呢？

充实生活

所以，大家应该去发掘和追求这些可以带来健康的快乐的事情。你可以在思考你如何完成一些重大目标（如买房）的同时，不要让这些事情成为你生活的重点。你需要主动去发掘一些可以让你健康地感受到快乐的事情，让他充满你的生活。当你感受到了巨大的快乐的时候，你再来思考一下你的那些重大目标，焦虑感已经被快乐冲刷干净了，剩下的只有冷静地思考。唯有冷静地思考才能帮助你真正达成你的愿望，而冲刷焦虑的快乐可以起到重要的作用。

你的梦想健康吗？

显然如果我的梦想是变得跟比尔盖茨一样有钱的话，我觉得我这辈子是永远实现不了的。但是如果我又觉得非实现不可，那这个焦虑就会伴随我的一生，直到你痛苦的死去。这种梦想就是不健康的。平时YY一下可以，但是倘若你去追求了，你这辈子就完了。

所以在你获得快乐而冷静下来的时候，你要好好思考一下，你的真实的愿望到底是什么。

不管你追求什么，有一些事情你总是要做的

显然，遵纪守法，保持身体和精神健康，不管你做什么你都必须注意。如果你身体垮了，那你什么事情都做不了，还要去医院受苦，把自己的钱都花光，简直不能再惨了。所以千万不能因为一些短期目标而牺牲你的身体健康。

举个最简单的例子，如果你朝九晚五地过了好几年，发现有一个可以加班的工作带给你+100%的收入，你到底要不要跳槽呢？有更多的钱当然是有助于你健康地获取更多的快乐的，但是如果你发现，你没办法在加班的前提下做这些让你快乐的事情，你也没有办法在加班的前提下维持你的身体健康，那你最好就不要跳槽。但是正如上面所说，有些人会因为工作而获得快乐，那这样你也不妨试一试。

人的欲望是无穷的，欲望可以是正面的，也可以是负面的。如果你的欲望就是打炮，打炮本身是中性的，你当然可以在保证不过度纵欲和炮友的身体健康的前提下，去多打点炮。但是任何事情都是有两面性的，打炮你控制不住后续的发展，可能会伤害别人，或者被别人伤害。因此遵纪守法就是一件重要的事情。

关于遵纪守法，我来说一个大家容易理解的。高速公路上前车驾驶员脑子进了屎突然停了下来（譬如说错过了出口）。正确的方法当然是你撞上去，因为你急着打方向盘的话，很大概率会翻车。如果高速公路的围栏外面没东西的话，你就翻山下面去了。但是你撞上去就一定没有责任吗？当然不是，只有在你维持了安全距离的前提下你才没有责任。但是很多人都说，维持安全距离会被人插队。

这个时候就进入了重要的时刻。如果你的生活很充实，你已经找到了让自己快乐的东西，那么让人插队带来的一点点不爽其实是不能影响你的，而且实际情况下你就算因为一直保持安全距离被插队，其实你也慢不了多少的，不信可以自己去做实验。那么这个时候，因为你的快乐，使得你可以淡定地保持安全距离，那么不管发生什么事情，你都将很难受到法律要求你承受的损失。

保持快乐是多么地重要。

所以我决定创办《淡定组》

为此我造了一个QQ千人大群【淡定组】：171319457，大家有兴趣的可以进来讨论。大家也不要当成是上来上课，我自然也无法深入地理解每一个阶层遇到的具体问题，所以应该共同讨论和进步，大家才能变得淡定，获得真正的快乐。

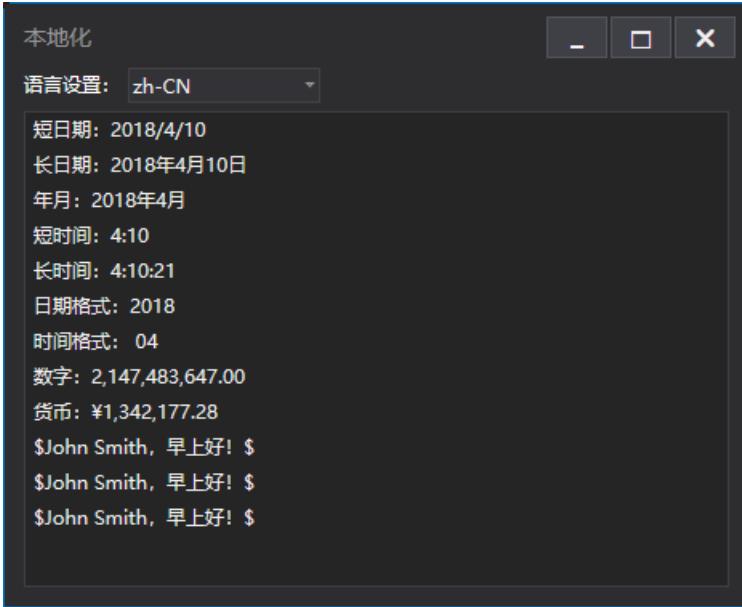
如果大家喜欢的话，我可以根据群里讨论的内容和大家的疑问，以后陆续写几篇文章，给出一些可操作性高的方法。你们有什么看法也可以写，大家互相帮助，成为精神上健康快乐的人。每个人都有很多人生的经验，当然不是一篇文章就能写完的。今天这一篇文章只是抛砖引玉，但是给出了我认为正确的纲领——

遵纪守法，充实生活，冷静思考，淡定行动，非常快乐

（逃

GacUI_支持运行时切换语言

先看效果：



使用方法很简单。先把不同语言的字符串定义好：

```
<LocalizedStrings ref.Class="demo::StringResource" DefaultLocale="en-US">
<Strings Locales="en-US">
    <String Name="ShortDate" Text="ShortDate: ${0:ShortDate}"/>
    <String Name="LongDate" Text="LongDate: ${0:LongDate}"/>
    <String Name="YearMonthDate" Text="YearMonthDate: ${0:YearMonthDate}"/>
    <String Name="ShortTime" Text="ShortTime: ${0:ShortTime}"/>
    <String Name="LongTime" Text="LongTime: ${0:LongTime}"/>
    <String Name="DateFormat" Text="DateFormat: ${0:Date:yyyy}"/>
    <String Name="TimeFormat" Text="TimeFormat: ${0:Time:HH}"/>
    <String Name="Number" Text="Number: ${0:Number}"/>
    <String Name="Currency" Text="Currency: ${0:Currency}"/>
    <String Name="Sentence" Text="$($)Good morning, ${0}!${($)}"/>
    <String Name="Title" Text="Localization"/>
    <String Name="Label" Text="Selected Locale:"/>
</Strings>

<Strings Locales="zh-CN">
    <String Name="ShortDate" Text="短日期: ${0:ShortDate}"/>
    <String Name="LongDate" Text="长日期: ${0:LongDate}"/>
    <String Name="YearMonthDate" Text="年月: ${0:YearMonthDate}"/>
    <String Name="ShortTime" Text="短时间: ${0:ShortTime}"/>
    <String Name="LongTime" Text="长时间: ${0:LongTime}"/>
    <String Name="DateFormat" Text="日期格式: ${0:Date:yyyy}"/>
    <String Name="TimeFormat" Text="时间格式: ${0:Time:HH}"/>
    <String Name="Number" Text="数字: ${0:Number}"/>
    <String Name="Currency" Text="货币: ${0:Currency}"/>
    <String Name="Sentence" Text="$($)${0}, 早上好! ${($)}"/>
    <String Name="Title" Text="本地化"/>
    <String Name="Label" Text="语言设置: "/>
</Strings>
</LocalizedStrings>
```

然后在窗口上声明这个窗口需要引用这个字符串表格。你可以写很多个，名字不一样就可以：

```
<ref.LocalizedStrings Name="Strings" Uri="res://StringResource" Default="true"/>
<Window ref.Name="self" Text-str="Title()" ClientSize="x:640 y:480">
```

字符串表格会根据refClass生成一个这样的类：

```
module <localized-strings>demo::StringResource;

namespace demo
{
    class StringResource
    {
        interface IStrings
        {
            func Currency(<ls>0 : ::system::String) : (::system::String);
```

```

func DateFormat(<ls>0 : ::system::DateTime) : (::system::String);
func Label() : (::system::String);
func LongDate(<ls>0 : ::system::DateTime) : (::system::String);
func LongTime(<ls>0 : ::system::DateTime) : (::system::String);
func Number(<ls>0 : ::system::String) : (::system::String);
func Sentence(<ls>0 : ::system::String) : (::system::String);
func ShortDate(<ls>0 : ::system::DateTime) : (::system::String);
func ShortTime(<ls>0 : ::system::DateTime) : (::system::String);
func TimeFormat(<ls>0 : ::system::DateTime) : (::system::String);
func Title() : (::system::String);
func YearMonthDate(<ls>0 : ::system::DateTime) : (::system::String);
}

static func Get(<ls>locale : ::system::Locale) : (IStrings^)
{
    /* 略 */
}

new ()
{
}
}
}

```

<ref LocalizedString>的意思，就是给窗口加上一个demo:StringResource:IString^类型的Strings属性，作为一个缺省的字符串表格（在-str绑定里面就可以省略掉Strings这个名字）。这个属性的内容会根据GetApplication()->GetLocale()的结果而自动变化，切换成不同语言的IStrings实现。

最后使用他就很简单了：

```

<TextList HorizontalAlwaysVisible="false" VerticalAlwaysVisible="false">
<att.BoundsComposition-set AlignmentToParent="left:0 top:0 right:0 bottom:0"/>
<att.Items>
<_ Text-str="ShortDate(self.dateTime)"/>
<_ Text-str="LongDate(self.dateTime)"/>
<_ Text-str="YearMonthDate(self.dateTime)"/>
<_ Text-str="ShortTime(self.dateTime)"/>
<_ Text-str="LongTime(self.dateTime)"/>
<_ Text-str="DateFormat(self.dateTime)"/>
<_ Text-str="TimeFormat(self.dateTime)"/>
<_ Text-str="Number(self.number)"/>
<_ Text-str="Currency(self.currency)"/>
<_ Text-str="Sentence('John Smith')"/>
<_ Text-str="Strings.Sentence('John Smith')"/>
<_ Text-bind="self.Strings.Sentence('John Smith') ?? ''"/>
</att.Items>
</TextList>

```

欧耶！

如何处理找对象的焦虑

昨天开的1000人群没几个小时就满了，我只好一边蹲坑一边查攻略看看怎么升级到2000，然后过了一天也满了。看起来大家对这个问题十分感兴趣。做完一边写GacUI（见上一篇文章）一边刷知乎一边看QQ群，发现大家对几大人生终极问题都很感兴趣。不过有人还试图在里面问C++，我觉得这个离题比较远（逃

人找对象可以有很多理由，但是几个不健康的理由分别是：

- 父母催你相亲
- 你看别人有你也要
- 孤独寂寞

健康地社会上的事情是有一定的套路的，就像我在之前的一个回答里面，说没有目的的学习就不会觉得痛苦一样。如果你想要再找对象的过程中摒弃痛苦，那么你应该不为了什么目标而去找对象。但是如果父母催你相亲怎么办呢？这个如何改善“管束焦虑”的问题以后再说。

找对象的终极目的是为了结婚，所以你需要的是一个能长久跟你相处下去的人。两个人可以长久相处下去，必然意味着不可能在每天见面的时候装出一副人的样子，不然你就会越来越累。只有坦诚相待才可以得到精神上的放松。那么很明显，以下几个动作就不利于你获得一个稳定的长期关系：

- 献殷勤
- 追求的过程中处于明显不平等地位
- 交往的过程中处于明显不平等地位
- 害怕分手而扭曲自己的欲望与行为方式

如果你为了维持关系而必须做出违反你自己本性的努力的话，那显然你只会在相处的时候越来越感到累，累你就会在想要获得自由和想要维持关系的矛盾中变得焦虑，一旦焦虑超过了你的快乐，你就很容易爆发。这个时候就很危险，如果你刚好属于脾气暴躁的人，说不定就犯罪了（逃

因此在两个人确定关系之后，要马上暴露出自己的本性，也要鼓励对方暴露出自己的本性。如果你想要弄明白这个人是不是可以作你的配偶，那你还可以从同居开始。只有在同居的过程中互相暴露本性，你才能够很快的确定，你是否能够跟这个人长久的生活下去，而不会觉得焦虑。大概一年到两年左右的时间你就可以做出决定了。如果互相暴露本性一年一点事都没有，那恭喜你，可以去领证了。

但是很多人根本就走不到这一步，因为连可以确立关系的人都找不到。那怎么办呢？这个我以前已经讲过了，要从广撒网开始。广撒网不是让你一脚踏10船，而是要让你走出门，先认识很多异性，跟他们交朋友，然后在日常的相处中，看看谁比较合适。如果你连candidate都没有，你还怎么发offer呢？

无论男女，都可以广撒网。

显然这是一个漫长的过程，这就是为什么人不能怀有目的地去找对象，因为一旦你有了目的，你就要猴急，一猴急你就无法维持高标准，然后跟一个无法跟你和谐相处的人确立关系的时候，你就会遇到大量的矛盾。两个人要在一起生活不是那么容易的，任何小事都有可能演变为灾难性的结果。

所以看了这么多，到底要怎样才能找到一个对象从而结束焦虑呢？显然，第一步就是要学习不去问这个问题。你为了获得一个对象而去找对象，这本身就是一个具有很强的目的性的行为，要戒。

人这一生中并没有什么事情是一定要做的，譬如结束单身生活。你有这个欲望，没有问题，存在欲望是一个很健康的事情。但是就跟[之前的文章](#)所提到的一样，首先你要有一个健康的生活，一个健康的生活是充实的。在寻找对象之前，你最好找到一两个可以让你获得非吸毒式的快乐的方法，充实你自己的灵魂。只有你的灵魂充实了，你吸引异性的效率才会提高，没有人想要和一个无聊的人相处。

显然，一旦你有了一个充实的生活，你就首先不会受到没有对象的这件事情的折磨，那么在合理的焦虑水平的前提下，你才能冷静的思考，冷静地行动，最后找到一个合适的对象，从而满足你的欲望。其实解决所有焦虑的方法都是一样的，只要你能够在生活中获得足够的快乐，你不觉得别人成功完成了这个事情（在这里就是得到伴侣）会使得对方的价值比你更高，你才能更加成功地完成这个事情。

因此，首先从放弃“我一定要找到对象”的这个念头开始，然后再去找对象。这个时候，找对象本身已经成为了你生活中增加一点味道的调味料，没有也无所谓，但是有会让你更开心。只要你有了这个心态，你就可以去认识很多异性。你只要跟够多的异性交朋友，你就能够频繁地跟他们接触，只有互相接触，互相产生了好感，你才能有进一步的进展。

如果你的到了足够多的异性朋友之后，就开始想我一定要在里面挑一个出来，那你就错了。这仍然是“我一定要找到对象”的念头。确立关系应该是一件自然而然地发生的事情，就跟同性结交成朋友是同一个意思，显然你不可能去追求一个同性来做你的朋友的，那为什么一定要去追求一个异性去做你的男女朋友呢？

不要追求，要让事情自然地发生。只要你有一个充实的生活，你自然就会变成一个有趣的人，人们就会更加受到你的吸引，从而你确立关系的概率就会增大。如果你没有一个充实的生活，整天想着去追求男神女神，那你作为一个无聊的人，显然对方是

不太可能看上你，哪怕是看上了多半也是找千斤顶/接盘。不管是千斤顶还是接盘，这都是一个不平等的地位，而不平等的地位是无法维持长久的，哪怕你临时确立了关系，你的愿望必将落空。

只有让两个人同时产生出对对方的好感，这种时候你才能自始自终地维持一个平等的关系，这是健康的长久关系的必要条件。任何人都不要试图做出牺牲，做出牺牲你就会有一种要对方报答的想法，最终会伤害你的感情。但是如果你并没有要对方报答，那对方也会变本加厉，最终伤害跟你的感情。这些事情最终都会让你们落入一个不平等的地位，从而在长远看来，造成关系的破裂。

所以在确立关系之前就有一个互相平等的地位，也是很重要的。所以如果你自己不是男神女神，你就不要去追求另一个神，这不可能有平等的地位的。人跟人才能在一起，人跟神是没有前途的（逃

说到这里，这个健康确立关系的流程就呼之欲出了：

1. 想办法获得非吸毒式的兴趣来充实你的生活
2. 摒弃“一定要找到对象”的想法
3. 尽可能结交更多的异性朋友
4. 于每一个人以朋友的关系想相处，维持一个平等的地位
5. 万一有一个人与你互相对对方产生好感，你就可以进一步发展恋爱关系了，但是不要去追求
6. 在你觉得十拿九稳的时候表白
7. 暴露本性，鼓励对方暴露本性，同居
8. 结婚

显然，相亲这件事情从一开始就应该抛弃，因为不管你到底是因为什么而相亲，对方有相亲压力的可能性是相当高的。就算你对谈恋爱这个事情又健康的看法，对方不一定也有健康的看法，那你无法走完这个流程的几率就非常大。所以不要相亲。至于如何对付父母的压力，以后我会再写文章阐述。

大家如果对找对象的这个事情一直有焦虑的话，看看这8条，你是从哪一条开始不满足的。**如果你从第一条就不满足，那你最好先放下所有的事情，好好充实一下自己，让自己成为一个有趣的、快乐的、给人一种散发正面感情的感觉的人，你才能够在结交异性朋友之后，成功获得对方的好感，从而满足你对建立关系的欲望。**

最后有必要再强调一下，你在寻找一个对象的时候，千万不要去追求，而应该让你的对象在你的一群朋友之中，通过日常的正常的、平等的相处，慢慢的自动筛选出来。一旦你去求了，你就容易求不得，你就会焦虑。

成年人如何处理来自父母的管束焦虑

前言

学生就没什么好说的了，你们跟父母要学费，被管也是没办法的。不过你要是牛逼到上大学就可以自给自足，那可以继续往下看（逃）。[办了个群](#)很容易看出来大家最关心的事情一共也就那么几件，很多人也在抱怨关于父母的问题。

不过在解决这个问题之前，我们得明白，基本上你是无法改变父母什么的，所以这主要是个策略问题。很多父母都把自己的精神寄托建立在小孩身上，其实这是不健康的关系。父母当然也有焦虑，父母的焦虑当然也应该通过充实自己的生活、找到自己人生的乐趣来解决。而把快乐建立在自己的小孩身上，也是吸毒式的。小孩今天关心父母，父母觉得很开心。关心了一年之后，父母就觉得理所当然了。父母的欲望被提高，父母的阈值被抬了上去，但是小孩有了自己的家庭之后，必然会把重心放在自己的老婆孩子上，给父母的关心肯定会越来越少。这是一个永远都无法满足的欲望，很多矛盾也来自于此。吸毒式的快乐是不可依赖的，所以父母也要充实自己的生活。

人老了，并不会因此跟年轻人就必须有什么不同。不过这基本上取决于每一个人，有些人就是愿意牺牲自己满足父母的各种要求，那我也只能说你觉得这样开心那没有问题。这篇文章的读者，是那些跟父母的观点矛盾的人。在无法改变父母的前提下，应该如何释放自己的焦虑，也是我今天要讨论的内容。

父母的开明性与所拥有的财富其实并没有什么关系，无论是穷的富的，我都见过有那种一定要把自己小孩绑在身边的。有钱还好，小孩没出息还可以挥霍一下，把问题留给孙子。那些没钱的家庭就这样把小孩的前途给葬送了。今天在群里就看到一个人抱怨，说父母因为东北没什么工作，所以把本来要读CS的小孩的志愿强行改成了医生，而且医生也赚不了什么钱。东北的码农不好过，难道别的地方的码农就不好过吗？所以这里的根本动机，其实就是不想小孩走。父母不想你走，你就真的不走了吗？

自己的专业自己做主

很多人就在这里遇到了第一个问题——高考志愿不是我填的怎么办？讲道理其实我觉得这个问题没什么办法，你们只能认命了。学生自己也是有责任的。高考很重要，并不意味着所有的时间都必须贡献给高考。父母是不太可能明白在我国迅猛发展下各个专业所蕴含的意义的，所以要是学生也不去了解，那到时候填志愿，你到底想用一个什么样的指导思想去填志愿呢？实际上就是很多人都没有指导思想，而且非常短视。还有的父母是根据自己30年前的经验来给小孩填志愿的，毕业了就知道脑子进屎是一种什么感觉。

总之，成年人要对自己负责。填志愿的时候你们已经是成年人了，父母是不会对你们负责的，他们是基于对自己负责来控制你们的。但是你们在本科毕业的时候，实际上还是有机会的。你们可以去考研。那些没考上本科的，可以考虑转本，虽然不容易。张雪峰先生虽然看起来很逗比，但是有一句话说的很对，就是考研你应该把他看成一个转专业的机会。考研父母总没办法给你们改志愿了吧，好好把握机会。

对于那些最终证明这个志愿还不错的人，你们工作也是一个机会。如果你们愿意换一个城市工作，不住在家里，那就轻松多了。

不管怎样，上面的所有准备工作，都是为了一件事情而准备的——那就是经济独立。经济独立是很重要的。人要生存，生存要花钱，如果连钱的来源都无法把控，那你的人生实际上就是别人的。不过有很多人会在这里遇到第二个问题，就是找工作。怎样解决找工作和收入的焦虑的问题以后再说，现在先说这个找工作，到底是你找工作，还是父母找工作。

自己的工作自己做主

为什么说工作要自己做主？因为第一份工作实际上定下了你人生的基调。这是你可以获得自己的人生的第一个转折点。如果你让父母来替你找工作，那父母肯定会按照自己的想法，来安排你的人生。说到这里要紧扣一下主题——焦虑。父母安排的人生你不一定满意，但是你如果被安排了，那你肯定会产生焦虑。这种焦虑基本上是没有办法通过调整自己的心态来解决的，解决它难度会更高。对于那些有选择的焦虑，实际上是你通过充实自己的人生，获得更大的快乐，冲掉焦虑，从而让自己可以冷静思考，最终解决问题，或者改变你的想法。而如果被父母安排了工作，你基本上是没有选择的，没有选择的人生是没有办法解决焦虑的，所以首先你必须让自己有所选择。而有所选择的第一步，就是自己找工作。

诚然，自己找工作的前提是自己能找到工作。对于那些工作能力强的人，这当然不是个问题。对于那些不行的人，如果父母也没有门路，那也不是个问题——反正你靠父母也没用。对于那些MADAO，父母还能提供一点帮助的时候，你就要好好想想，是自己的自由重要，还是有份好点的工作重要了。有些人可能会觉得，只要拿的钱多，以后一辈子被父母管也没有问题。我只能说，如果你觉得这样开心，那也挺好，成功通过把自己的价值观align到父母的价值观上降低了焦虑，也是一个好办法。

自己的经济自己做主

有了工作，大家要面临买房的问题。一旦父母给你出了首付，基本上你的居住问题也被父母控制了。这种控制的力量是强大的，哪怕你翻墙留学在美帝住了下来，在“自由”的国度生活了快10年，我也见过因为拿了父母的钱在美帝买房子，结果被远程控制的例子。真是不要太多。

为什么说是一个策略问题呢？因为，你可以接受父母的钱买房子，也可以不接受父母的钱买房子。对于那些没有钱的父母这当

然不是一个问题，但是倘若父母还有点钱呢，但是你却买不起房了呢？**你就要好好想想，是自己的自由重要，还是房子重要**。这个我也无法替你做决定，因为有些人觉得房子很几把重要，牺牲一切也要买，我只能说，如果你觉得这样开心，那也挺好，毕竟你最后有了房子，虽然贷款很重。

自己的老婆（丈夫）自己做主

如果你专业也是父母选的，工作也是父母选的，房子也是父母买的or你干脆就住在父母家里，那相亲这个问题肯定跑不掉。连经济都不能独立的，没什么靠谱的异性会喜欢你的。我见过一个例子，就是什么东西都是父母挑的走到了这一步，结果自己强行娶了一个父母不看好的老婆。成年人嘛，你打得过父母，硬要去领证，那父母其实也没你办法。但是接下来，因为双方父母在夫妻俩的生活细节上干涉太多，最后变成了一副悲惨的样子（逃

自由是一环扣一环的，如果你最终经济独立失败了，那你后面很大概率没什么好日子过。所以你想要掌控自己对配偶的选择，那么你首先要经济独立，你才能顶得住压力自由恋爱。你经济独立了，父母当然还是会催你结婚，催你相亲，但是你可以很有底气的拒绝，说你就是要自由恋爱。

万一你经济不独立，你自己找估计也找不到什么靠谱的人，父母相亲可能还会给你弄一个比你自己找要好的。**你就要好好想想，是自己的自由重要，还是能结婚重要了。**

那么，到了这一步，如何找到自己人生的伴侣，请参考[上一篇文章](#)。

自己的生活自己做主

如果你前面的几个问题都顺利解决了，那么这其实就不是个问题了。父母再也没办法管你要穿什么颜色的裤子，过年要不要洗头发，拉屎一次不能超过多少分钟，要给你的小孩穿多少衣服、生病找中医还是西医这样的事情了。你就获得了充分为自己负责的自由（逃

尾声

想想有很多人活着，实际上都不是自己活着。小时候被监护是没有办法的，结果高考志愿父母来填，工作后父母找工作，父母出首付买房，父母带你相亲，父母让你生小孩，你们工作父母给你带，孩子的价值观最后都是父母灌输的。**活到了四十多岁还是父母在做主，这是很可怕的，在座各位一定要努力，不要让自己变成这样。**

这里我们复习一下。与之前所讨论过的问题不同的是，父母的控制力量是强大的。如果你被父母控制，但是你们的观念不兼容，那么这种焦虑是没有办法通过改变心态来解决的，因为这会伴随你的一生。哪怕到了父母死去，这个影响都不会消失，你就得焦虑一辈子。

如果你的想法跟父母有本质的矛盾，而且又改变不了父母，那么你为了降低焦虑，必然只能通过远离父母的控制来解决这个问题，那么：

- 自己填志愿
- 自己找工作
- 不依赖父母的经济援助
- 自己找配偶
- 自己解决接下来的一系列生活难题

父母可能比你牛逼，但是一旦你决定要远离控制，那么你就要为自己的选择而负责，不要后悔。如果你比父母牛逼，但是你选择了愚孝，那也是你自己的选择，不要后悔。人生是很困难的。如果你失去了自由，那你就连用快乐冲刷焦虑的选择都没有了。**哪怕是被父母控制，我也希望那是因为你的选择，你可以随时结束它，而不是因为你没得选。**

如果你有了自由，你就有办法通过充实自己的人生，获得更大的快乐，冲掉人生中的其他焦虑，从而让自己可以冷静思考，最终解决问题。

如何处理经济造成的焦虑

今天看到一个老朋友在知乎上贴了这个视频 <https://weibo.com/tv/v/GbDfqdAVC>。视频里面一个人在喷现在世风日下人心不古，不为社会做贡献，每个人都在攀比，官大一级让你擦鞋。所以很多人因为穷觉得自卑。当然我肯定不是这样的人（指讲台上那个），作为一个目标是吃软饭不工作的人，我不想指责大家为什么不努力为社会做贡献。不过视频里面的说法还是可以借鉴的。

虽然焦虑和自卑是不一样的，但是如果你去比较，然后得出一个结论是你不如很多人，当然也会焦虑。但是你现在就是没办法一夜暴富，那怎么办呢？所以心态还是要摆正。不过我必须先说，我并不是叫大家放下赚钱的欲望，也不是叫大家不要赚钱。你能赚钱当然比不能赚钱要好啊，但是在你还没有成功赚钱的时候，其实你是可以不过得这么焦虑的。

经济上的焦虑源于比较。假如你身上负担着同样的危机，但是没有任何人过得比你好，你的焦虑感肯定会大大降低。因为你心里并没有一个什么是好的概念。一旦有人过得比你好，你就会开始想，原来我还有那么多机会和可能性可以失去，还有那么多糟糕的事情可以遇上，不月入百万的话简直不得了了。

讲道理，中国经济发展到这个份上，你能当个普通人，已经能排在全球很靠前的位置了。加上中国很多强行压低成本的方法（譬如让医生收入那么低），你肯定比赚一样多钱的发达国家的人要过得舒服多了。所以你的穷，可能是结构性的，因为人类的生产力水平就这样了，并不是靠你的个人努力就可以解决的。在讨论所有的事情之前，你先要明白这一点——作为一个好人，你赚钱的愿望，很有可能是无法实现的。

好了，反正我也不是来教大家怎么赚钱的，但是你在保持贫穷的前提下，降低焦虑感那是完全可能的。人高兴和不高兴的原因，是不一样的。当然这看起来像一句废话，不过更加具体的意思是，你解决了让你不高兴的事情，你可能不会变得更高兴。把你高兴的原因拿掉了，你也不一定会变得更加不高兴。这就是为什么可以在保持贫穷的情况下，降低焦虑感。有钱你高兴，但是没钱不是你焦虑的原因，你真正焦虑的，是你对危机失去了控制。有钱可以帮助你控制危机，但是没有钱，你一样可以借助其他的东西来控制。只要你成功控制住了越来越多的危机，你的焦虑感就会逐步下跌。

举个简单的例子，养老。全社会都提倡养儿防老，而且在制度上的确也在强行让很多不养儿的人失去防老的手段，从而让你养儿。但是真的是这样的吗？并不是。只要你好好计划，降低你现在的非必需支出，把省下来的一点点钱投入到理财计划中去（哪怕你一开始只有1000块钱），然后努力工作，你总有办法养老的。此乃开源节流。当然养老的意思，不是让你老了就能过上多么好的生活，毕竟你就这个水平了，老了能只是比有收入的时候差一点，已经很不错了。如果你生了小孩，你的防老跟养儿的解耦就更大，以后不仅对自己的老年生活更加有控制，还可以避免小孩为你做出无辜的牺牲，就像你现在觉得养你父母也有点难搞一样。

想想我父母，退休后收入直线下跌50%，这是很多人的常态，要做好心理准备。

再举个例子，有些人削尖脑袋挤进北上广，然后就拿着拿一点工资。讲道理，如果你反正买不起学区房，小孩没办法在北上广高考（其实广州没有高考优势，来汕头吧！），你去北上广不一定就会比你去其他地方要更好。首先这个房租就是很贵的，其次你的工资的差价就一定比你房租的差价要高吗？所以你事实上可以跟计算来得到，你到底是适合北上广，还是适合二线城市。考虑到二线城市各种比北上广便宜的东西，你是有可能过上更好的生活的。那你不去北上广到底失去了什么呢？失去机会嘛？其实并没有。很多时候你本来就没有机会，你只是失去了觉得有机会的这种感觉而已。

这两件事情的共性是什么？很明显，共性在于你要对你的经济做出规划。规划包含两部分，第一部分是怎么赚钱，第二部分是怎么花钱。当然具体怎么规划，我教不了你，在这里我只能给你提供这么一个思路，你要根据自己的具体条件，和对相关知识的学习，来很好的规划你的未来。一旦你觉得你对未来的控制，因为你的规划而提升了很多（或者预测的精确性更强了），你不仅会因为未知的事情的变少而降低焦虑，你还可能可以更加清晰地看出，你目前的问题在哪里，从而更有针对性地去解决。万一你命好，你根除了你的一部分问题，你一部分的焦虑就永远的消失了。

前面的文章其实也一直在指出，放下焦虑，是需要更多的快乐来冲刷的。如果你能找到适合你的成本可控或者非吸食式的兴趣的话，你就可以从中源源不断地获取快乐。只要你足够快乐，你的焦虑感自然就会下降，从而你的大脑就会更加清醒，你规划和执行规划的能力就会上升，从而更加降低你的焦虑，此乃良性循环也。

那到底什么样的事情，能给你源源不断地带来乐趣呢？其实这也是规划的一部分（逃

当然，在你规划之余，心态也是很重要的。这个事情对于很多人来讲可能会很难，但是你可以尝试着从不跟别人攀比开始。如果你正在找对象的话，你可能需要找一个认同这个观点的对象，来帮助你摆正心态。因为如果你找了个女朋友天天嘲讽你工资这么低屌这么细你还有什么用，那你怎么可能摆正心态呢？配偶之间相互合作，朝着共同的目标前进，是一件有必要而且美好的事情。

那如何不跟别人攀比呢？规划也是帮助你达到这一目的的其中一种手段。假设你老了会得一个大病，有病就要钱，没钱就得死。那你可能会觉得那些有钱的人没有这个问题多好，你在对比之下就得到了焦虑。其实你省吃俭用，买个正确的保险，也是可以不去死的（逃。而且你还不需要比较，你就知道自己不用死。焦虑肯定就下降了。

你在越来越多的有效规划中，你就会感受到你慢慢的觉得，别人那些你以前觉得牛逼的素质了，好像没有以前那么牛逼了。因为人家能抵抗而你不能的危机，你现在能抵抗一部分了，你跟别人的能力尽管没有任何变化，但其实别人对于你而言就不是那么牛逼了。这也是摆正心态的一种办法。你慢慢的就会把你的价值观调整到一个健康的状态上，从而在克服不了的攀比之中，你也不会觉得自己太惨了，从而焦虑感也就下降了。

如果你觉得能控制的东西都已经控制了，那些不能控制的就只能看命的时候，你对于未知的焦虑就会下降到极限，因为你实在没有办法了，干脆就不去想了，好好干活，多赚点钱，以前不行的事情，以后总是可以的。实在不行，退休年龄后你还可以继续工作呀（逃

当然，无论怎么样，“想过奢侈糜烂的生活”这个愿望，你最好不要有，不然你就没有救了。而且你也不要有一种“凭什么人家能花钱我就不行”的想法，不利于保护你的未来。最后一点，千万不要对国家的经济形势过度乐观。你不是专家，你的预言，多半是不行的。

挖坑（可能）不填文章目录



[vczh: 想戒除焦虑吗？想变得快乐吗？ ↗ zhuanlan.zhihu.com](#)

[vczh: 考不上三本也能懂系列——前言 ↗](#)



[zhuanlan.zhihu.com](#)



[vczh: 编程学习方法随笔目录 ↗ zhuanlan.zhihu.com](#)

如何理解戒除焦虑和积极行动的矛盾

人，当然是要追求成功的。

——vczh

(逃

零

虽然之前的文章一直在说如何戒除焦虑的事情，但是戒除焦虑跟戒除欲望本身是完全不同的，但是读者们很容易混淆。昨天在[跟群友交流](#)的时候，我想到了一个很好的比喻方法，来让大家体验一下这两者的根本区别——买彩票。

人买彩票嘛，当然是为了中奖。国内的福利彩票可能听起来没什么用，美帝的彩票动不动上亿美元，我觉得这种彩票非常符合我国人民的心情。于是你次都进场，一注两美元。很明显，你是冲着中奖去买彩票的，没有人是为了收集那些票票去买彩票。不过也许你并不一定是为了拿一等奖，想想其实拿个末等奖开心开心，小赌怡情。

但是，开奖的时候发现末等都没有，你这个时候是什么心情？

显然不会有特殊的心情。为什么不会有特殊的心情呢？因为你对这件事情在内心中的期望，就是不会中奖的。买彩票不会中奖，再正常不过了，就跟多吃几碗饭就一定会饱一样平常。

一

显然，谈到焦虑的主题，你不会因为彩票没有中奖就不焦虑，然而你还是会去买。零中奖焦虑还要去买彩票这两者之间，显然是没有什么矛盾的。这跟日常生活中的各种戒除焦虑和积极行动之间的矛盾是类似的——本不该有什么矛盾。

我觉得，以这种心态来迎接社会的困难，是没有问题的。

细心的读者可能会发现，我一直以一条很明显的主旨来贯穿之前几篇文章的想法，也是整个戒除焦虑的手段的核心内容。就是，你可以去追求成功，但是如果你最终没有成功，你不要觉得不开心。这种心态对于某些人来说可能很难获得，需要训练一下。我自己觉得是理所当然的，不知道为什么，可能从小长大看着身边的人都是这样做的，我也就这样做了。

二

追求成功如买彩票，学习如买彩票，找女朋友如买彩票，赚钱如买彩票，世上万事如买彩票。买个彩票没中奖，很多人都会觉得气馁继续买，而不中奖本身并不会给他们带来困扰。工作配偶赚钱都一样。你当然应该总是去好好学习，好好找女朋友，好好赚钱。但是不能马上成功怎么办？

千万不要觉得不开心。你只需要重新努力一遍就好了。享受这个过程。万一下次成功了呢？成功了你开心，不成功你内心毫无波澜。整个生活就只有爽。赚钱是为了什么？你们都应该问一下自己这个问题。我个人认为，赚钱就是为了爽。但是如果你可以在成功赚钱之前你就觉得爽，那万一你真的赚了钱，岂不是觉得巨爽？这种人生简直爽爆了，为什么活着要有焦虑呢？对吧。

三

不过人的焦虑都是源自压力。压力从何而来？其实仔细考虑一下就知道，所有的压力实际上都是外部的。人作为自己，首要目的是吃饱穿暖，其次是繁殖。吃饱穿暖对很多人来说其实都不是问题了，有问题的只是靠什么吃饱靠什么穿暖。如果你能够一直保持自己吃饱穿暖，那么你是不应该有压力的。但是人基本不可能没有压力，如果你们分析一下自己的压力的来源，都会发现是别人给的。

譬如说——丈母娘要你买房子才结婚。

想想，这件事情跟你女朋友要求你买房子才能结婚实际上是不一样的。有可能你女朋友自己也不同意她妈的这个做法。因为你没有房子，她想跟你结婚，但是她妈妈棒打鸳鸯，大家都不高兴。于是你产生了压力。这个压力，就是未来的丈母娘给的。

那这个时候怎么办呢？其实，这也是女朋友进入属于她自己的人生，所需要的成长的一环。让你女朋友亲自面对这个boss，打败丈母娘，跟你在一起。你可以助攻，但是解决问题的，终究必须是你的女朋友。不要跟现在的父母对待小孩一样，要什么都满足，遇到什么困难都替她解决，长大了就变成废物。显然你不应该让老婆变成废物的。

当然了，如果最后变成你女朋友要你买房子，那这就变成你的价值观的问题了——你想不想娶妈宝女当老婆？不要害怕丢弃沉没成本。

凡是外部的压力，最终都是可以自己解决的，而内部压力，得靠医生才能解决。

(逃

说远了。为了小孩读书可能你也得买房子，不过我个人觉得租房读书这个事情迟早都是要落实的，不然对社会发展不利。我国作为一个奉行集体主义的国家，没有理由放弃很明显的这种社会效率最大化的做法。今天实施起来比较难，有些房东还不肯给你租房证明因为他不交税你也不肯被涨价，所以有些人连办工作居住证都会遇到困难。这些都是很现实的问题。

但是想想，碍着你吃饱穿暖了吗？其实并没有。所以这些都是外部压力。过多的压力会造成焦虑，而有了焦虑，就会进入系列文章的主题，那么当然也被今天的文章所覆盖。你当然可以去追求北京户口，但是万一他不发给你呢？千万不要感到不开心。因为随便挑一个中国人，他很大概率都没有北京户口。但是并不要因此就放弃北京户口了，万一下次县太爷开恩了呢？

四

对抗焦虑，当然还是一个调整心态的问题。你当然可以通过成功来最终让焦虑消失，但是人的欲望总是超前的，所以你总是会有新的焦虑。如何让焦虑不要过多的影响你的心情，或者影响你的决策，也是很重要的。

人的一生，除了吃饱穿暖，并没有什么事情是一定要完成的。世上万事如买彩票，你每天都在努力为你的目标奋斗，成功了你会很高兴，但是在成功之前迎来的阶段性失败，你不需要有任何负面的感觉。但是保持这种心态，并不是教育你不要去赚钱。你真的连欲望本身都戒掉了，那就只能去龙泉寺学编程了（逃

但是做人也不能空虚。所以在迎来成功之前，你还是需要有充实的生活来源不断地给自己提供快乐。譬如说我写代码我就可以凭空产生快乐，对于你们来说，可能是朋友聚会、吃饭喝茶、打篮球玩游戏、去公园跟老爷爷下棋，这些都可以。找到属于自己地非吸毒式快乐。什么是吸毒式快乐？那些快乐门槛会被迅速提高的，就是吸毒式快乐。譬如吸毒（逃，买奢侈品，从自己小孩身上获取满足感等。[第一篇文章](#)描述了这个定义的详细内容。

一旦你有了充实的生活，每天都可以自发产生快乐的话，快乐就可以帮助你冲刷掉焦虑，有助于你冷静下来思考并解决问题。解决问题才是最终打败焦虑的办法。但是你在这之前，不要被焦虑打败，要提前获得快乐，你才能真正的、有效地、通过成功来打败焦虑。要让自己进入一个正向循环。

找女朋友也是类似的。没有人喜欢一个散发负面情感的人。如果你做到了上面的这一点，你自己从心态上就变成了一个具有正能量的人，吸引妹纸的效果，当然也会得到大幅度的提高（逃。如果你是那种看中了一个女神就不断去追求，那显然不满足上面的说法，于是追求女神本身，十分有助于你地追求石沉大海。如何正确的看待这个事情，[第二篇文章](#)也说了很多。

五

当然以后我还会时不时地根据各种灵感，来教大家如何面对各种垂直领域的焦虑。我自己也很随缘，我追求完成这个系列，我也不会因为这个系列暂时没有完成就不开心。这种心态，也是一个写作的好心态（逃

如何在贩卖焦虑的世界中保持清醒

最近工作比较忙，加上闲下来的时间都刷美剧了，所以更新就慢了一点。不过今天看了一条微博，让我觉得有些事情还是要继续强调一下。最近经过了管理员的残酷剥削，[淡定组](#)又空出了很多位置了（逃

正文

前几天跟一在上海工作多年的朋友聚会，朋友是国外某中小型科技企业中国区的总经理，年薪应该在百万元左右，说他这薪水在上海也就勉勉强强保持个基本生活，房贷、俩娃的学费、保姆费用、全家的基本开销，一年到头也存不住多少，连奢侈品都不敢买，压力很大，真不敢放松。要得夫妻二人都是年薪百万，才能在上海过得比较舒服些。唉，中年压力大啊，不管在硅谷还是上海，都一样

必须指出，随着互联网的发达，每个人都可以听见每个人说的话。当然我个人认为博主的本意并不是贩卖焦虑，因为这句话是说给跟他同一个等级的人听的。我也经常抱怨自己工资低爽不起来，但是我的内心十分清楚，我的爽跟别人的爽是不一样的，我们之间并不需要互相学习这部分。

但是想一想，年薪两百万才能“过得比较舒服些”，难道年薪只有二十万的时候就不能“过得比较舒服些了吗”？回想当年在北京只拿二十几万的时候，我也觉得每天爽的不行，除去雾霾对自己的身体造成的伤害意外，我甚至觉得比现在在美帝的日子过得还舒服。当然现在我已经处于随时可以买房子的状态，而当年基本上是不用想什么房子的事情的。为什么我就不会觉得焦虑呢？

在讨论这个问题之前，我必须指出，很多人之所以觉得焦虑，实际上是他们的心智根本就没有长大，觉得成年了还可以跟学生一下，把人生过程游戏打成就的那个样子，包括但不限于：

- 进入985并毕业
- 买房子
- 拿到BAT offer
- 拿到xx牛逼学校的phd
- 结婚
- 给自己小孩上私立学校
- etc

但是实际上这是一种错误的态度。就说买房子吧。人为什么要买房子？买房子无非就是想获得房子的一些价值，譬如说稳定的居住，譬如说让丈母娘可以把你女朋友的户口本掏出来，譬如说让小孩可以上好学校，譬如假装自己的资产增加了（但是因为你根本买不起第二套所以没有任何意义）。大家都说买房子这个好那个好，所以你觉得应该去买房子。但是你仔细想想就知道，你需要的是稳定的居住而不是一套房子，你需要的是让小孩可以上学而不是一套房子。完成这些东西就只有买房子一条路吗？当然不是。

这就是我说的把人生过程玩游戏打成就的一个典型的表现。有房子当然可以实现你的愿望，但是没有房子也可以，为什么你就一定要死磕买房这一条路呢？你看我现在没有房子，但是我丝毫不会觉得，这些愿望我再买房子前就实现不了。因为我已经调查清楚了，而不像大多数人一样，不愿意动脑，觉得买房子就什么都解决了所以就之想买房子。

这种事情就跟高考填志愿一样。你对专业一点认识都没有，所以你只能抓阄。你对待人生一件这么重要的事情，你都是靠抓阄来处理的。那你以后过得不好，要焦虑，也是理所当然的了。

当然这篇文章并不是像试图说服你不要买房子，其实你买不买关我屁事（逃。就像之前的文章所说的那样，人解决焦虑，就得解决造成焦虑的根源。而在此之前，你需要通过充实你自己的生活，让自己变得快乐，来使得焦虑不会过多的影响你的【冷静思考】，从而让你更有效的解决这些造成焦虑的问题的根源，于是你就摆脱一部分焦虑了。

有些人可能会说，我的焦虑就是穷啊？这仍然是把人生过程玩游戏打成就的一个典型的表现。有钱当然可以实现你的愿望，但是手上没有直接把他们买下来的钱，怎么办？

这是你的事情，你需要通过自己的智慧来解决。上面说到，对抗焦虑的其中一环是能够【冷静思考】，在此之前，你得真的会思考才行。至于怎么学会思考，怎么调查问题，怎么找到自己心中真正的欲望，这个我教不了你。你可能可以去复习或者学习一下《数理逻辑》（逃

当然人不得不面对的事情，就是你看着每天知乎微博上一堆人年入千万btc，觉得自己像条狗一样，怎么办？其实关键的问题，你还是要找到自己想解决的问题是什么，要是用自己的大脑。回到那个买房子的事情。你要解决的问题，绝对不是买一套房子，你需要的是房子带给你的价值。然而你现在经过计算，觉得自己多半是永远都买不起了，怎么办？你只能坐下来好好想一想，你买了套房子到底是为了什么，然后你就去解决这些具体的目标，最后你会发现，不买房子，你也可以达到目标了。

类似的事情有很多。譬如说学编程赚钱。赚钱当然是为了自己可以心无旁骛，没有任何压力地继续学习编程。倘若你把赚钱本身当成一个目标，久而久之你就会忘记了你当初的目的是想学习编程，从而变成工资的奴隶了。没有年入千万btc难道就不能安心学编程吗？当然不是。

于是你看着大家很有钱，你也想要钱，其实你想要的是他们用钱能够做到的你现在做不到的事情。合法解决一件事情的方法有

很多种，并不一定需要靠钱。如果你努力地去了解自己，知道自己到底想要那么多钱是为了什么，最后你通过了自己的聪明才智，在没有钱的情况下你满足了自己的欲望，那这个时候你还会羡慕别人有钱吗？

当然有些人的内心是扭曲的，觉得只有比别人过得好自己才开心。如果你放不下这种欲望，焦虑是没办法解决的，只能出家了。如果你的心态是健康的，但只是因为自己健康的欲望一时三刻满足不了，羡慕别人有钱。这没有关系，你需要做的，就是训练一下你的大脑，让他真正有用。不要在战略上懒惰，为了达成什么目标都一定要走有钱有房子这条路。只有你的战略是正确的，你的战术才有可能达到你想要的效果。

最后说一下我自己好了。显然我觉得我比大多数国人的情况是要优越的，但是我也不会看不起穷人，也不会羡慕富人。钱我当然是喜欢的，但是我要指出的是，我实现这些目标，主要原因显然也不是我爸的房子和钱。讲道理，我找工作以前，花在学习编程的成本，也就不到3万块。我用3万块钱购买的知识和设备，今天你要买也差不多是3万块。很多人都是完全有能力拿出3万块的，为什么那么多人就学不会呢？真的是因为上的小学比我差，中学比我差，大学比我差，你老爸房子比我老爸的小吗？当然不是。

那是不是因为我父母开明呢？讲道理，我在学习编程的时候，除了跟他们要这3万块，我同时还获得了非周末练习编程被拔电源赶回去写作业，一开电脑就要被骂，动不动就说某段时间不能摸电脑，找资料回来还要被抱怨说为什么不把时间投入到学习上，等所有人打游戏的时候也会遇到的情况。我认为这根本算不上开明。而且高三之前我的成绩其实也不是太好（当然这是由于学校的统计方法的问题导致的，我没有对提高成绩付出过任何努力），我早期也没觉得自己就一定能够考上一本。who cares，你要做的是调查，你就会发现，上不了一本也没什么，你还有广东某工业大学（不过听说最近该校promote了）啊。我到现在还是这么认为的。

把上985作为解决问题的手段，就跟上面提到的一定要买房子你才能如何如何一样，是战略上的懒惰。我在十几岁的时候，曾经跟我把激烈的探讨过这个问题，还挨了几个拖鞋。显然我爸作为一个传说很聪明的人，哪怕是在当年能考上大学（但是被我爷爷花钱落榜了（逃），工作光速promote，取得了周围人眼中世俗意义上的成功，也不一定可以摆脱社会造成的影响。所以我可以理解要做到这一点是很难的，但是这不是你可以不努力在战略上投入智商的理由。

不要在战略上懒惰，觉得自己做什么都得先赚钱买房子。同理，也不要觉得今天你无法取得什么成就，是因为你老爸没有赚钱买房子。你当然没有办法在老爸是穷逼的前提下毕业去做什么银行家，但是你要活成一个正常人，充实你自己的人生，取得你自己认为有价值的成就，不要过多去关心别人的看法，是完全没有问题的。

后记

我来简单分析一下这条微博带来的信息。博主说，两百万才能在过的舒服一些，然后他现在一百万的日子，是：“房贷、俩娃的学费、保姆费用、全家的基本开销，一年到头也存不住多少，连奢侈品都不敢买，压力很大”。解决了这些问题就舒服了吗？显然不是，因为我当年在北京的时候，没钱给首付，（学费就算了年纪太小没资格给小孩交学费）没钱请保姆，但是我通过了一个健康的财政计划，让全家的基本开销保持在了一个比较低的位置上，再扣除50%税后收入做存款，然后就全部拿来吃喝玩乐，爽的一笔。

假设我现在仍然过着这样的生活，然后突然要给两个小孩交学费，突然要交房贷，收入猛增至100万，你觉得多出来的税前80万搞不定房贷和学费吗？我认为完全是能搞定的。于是这些开支并不会影响我吃喝玩乐的部分，照样日子爽的一笔，何来不舒服。所以说，博主觉得不爽，是因为他对自我的认识产生了偏差所导致的，而不是没钱。

C++需要不断地练习——vczh

写答案的时候看了一眼github的记录，这4个repo分别是，GacUI从codeplex备份到github的数据，和终于换到github之后 <https://github.com/vczh-libraries> 的3个主要的repo（GacUI、Vlpp、Workflow），长达6年。

第一年的数据统计不了，因为GacUI一开始是codeplex另一个项目的文件夹，里面太多其他东西，虽然也一并被分到了github上。

这个故事告诉我们，下班之后自己搞点练习，提高一下编程技巧，完全是没有问题的。你看我整天泡老婆打游戏，都能抽出时间写+3.8百万 - 2.2百万行代码。你们只需要每年写个12万行，应该也够了。



焦虑与攀比的内在关系

之前的文章说了那么多，都是在不同的方面讲述同一个道理：人要没有焦虑，就要有确信自己能够完成自己的目标的理由。开心也是其中一种。人活着当然是为了开心，如果你已经很开心了，那这也是一种证明。还有什么比开心更重要的呢？没了。开心主要还是跟你的心态有关。不过今天来讲一讲另一个方面。

人为什么会焦虑，因为你不能证明自己会过得好。如果你已经有办法证明自己一定能过的好，那当然就不会焦虑了。譬如说你有很多钱但是欲望没有变，这是最直白的一种不焦虑的状态。但是真的要去得到一个证明，难于登天，所以人们通常都会选择各种简单的方法来欺骗自己。攀比就是其中一种。

在这么多种证明里面，攀比其实是最没有用的。譬如说你开宝马人家看福特，就能证明你未来就更好吗？其实是不行的。虽然宝马车主平均要比福特车主有钱，但是这个统计结果对你是没有意义的。因为你已经知道自己是谁了，各项指标都定下来了，统计结果对你已经没有帮助了。你提一个LV包就一定能比不提LV包的人过得好吗？当然不行，道理是一样的。

但是很多时候你并没有别的更好的办法可以麻醉自己，所以有些人会选择花很多钱，买辆宝马，买个LV包，光鲜光鲜，然后过着贫穷的日子。当然我并不是在说买宝马买LV的人就一定是这样。这两个命题不是互为逆否命题。几年前还有割肾买iPhone的，道理也是一样的。拿iPhone的人就一定比拿Nokia的人未来日子过的好吗？谁知道呢。但是你手上有一个很贵的东西，是可以在一定程度上麻醉自己的，求不要那么焦虑。

然而不焦虑是不能用钱买的。

不过要做到完全不攀比，其实也不是那么容易的。因为人少不了竞争。竞争是会有个结果分个高低的，成绩好的人拿的好处多。但是并不是所有的比较都会构成竞争，攀比就是在指这一类不构成竞争关系的比较。你比赢了是没有实质上的好处的，你得到的只是对自己的麻醉。

这就像分手了去喝酒一样，你并没有打算解决问题，你只是想觉得不那么惨而已。所以有些人为了不觉得那么惨，也不考虑一下怎样做才能真正提高自己，而选择去攀比。攀比就像吸毒，吸毒式的快乐是难以维持的。你今天跟这群人比赢了，明天去比更牛逼的人，很快你就会败下阵来——因为人家是真有钱。最后你只会获得挫败感，从而更加焦虑。

如果你可以从一开始就控制自己不要去攀比，那其实你会更安全一点。这种安全是有保证的，因为你的攀比的阈值并没有被抬高，你就更容易被满足。如果你意识到了这一点，那么只要不攀比，你的焦虑就应该马上降低一点点。

人们还会把这种东西延续到下一代身上，譬如说一定要让小孩在北京读书之类的。你在北京读书你就一定会更难牛逼吗？哪怕你去报课外辅导班，然而高考的性质决定了，只要你接收普通的教育没有困难，那么辅导班能让你提高的幅度是相当有限的。因为最终还是要靠你自己学。

当然，如果你能在北京高考，那上个好学校的难度是比较低的，但是这又回到攀比这个问题上——你上了个好大学，你就一定过得比上部好的大学的人好吗？当然从统计意义上来说是这样的，可惜你已经知道自己是谁了，你的各项指标都定下来了，统计结果对你已经没有帮助了。

实际一点讲，你读什么专业对你的未来的改变可能会更大。如果你在刚过去的CS风口前毕业，你在广东某工业大学学习CS，可能比你在北大学考古，将来日子还要过得舒服一点。但是这又回到了一个问题——读一个火的专业你的日子就一定能过的更好吗？这显然是未必的。世界风云变幻，你填志愿的时候这个专业好，你毕业了就说不准了。

当然上面说了这么多，不是说叫你不要去努力，不要去调查做选择。我只是在焦虑的层面上讲明白一个问题——只要你还是用这些直白的指标来麻醉自己的话，是没有用的。你需要的是更加实在的东西。人不要把自己的一生过成游戏打成就那样。你在小学中学的时候，因为评价系统比较直白，所以你可能演成了一种习惯，就是你只要完成一个一个的目标，你就会变得越来越优越。不过上了大学情况就不一样了。

不过基于同样的原理，如果你中学超级努力最后考上了一个非常理想的大学，但是我现在告诉你你考上了这个大学也不意味着什么，你可能会觉得愤怒，然后想反驳我。为什么呢？其实你还是内再地在攀比这些东西。你是试图要把自己以后过得好的这个证明简化，从而麻醉自己，临时降低一下焦虑的感觉。

我还是要再强调一下，我并不是说考上一个好大学没有用，但是你知道它到底好在哪里吗？这个理由不能太笼统，因为笼统意味着统计，而你已经知道自己是谁了，所以统计结果对你来说是没有意义的。你要再具体一点，微观一点，知道这个大学对你到底好在哪里，为什么两个差不多的大学你要选这个不要选哪个，两个差不多的专业你要选这个不要选那个。这样你才能细化你的证明，不仅能让你的焦虑更加稳固的下降，而且你可能因此就获得了一个更加详细的人生目标，为了这个目标你得出了一些看着就很靠谱的步骤，更加让你确信你将来一定会过的好。这才是解决焦虑的办法。

不要简化自己的目标，目标的细节要有理有据地复杂，才能真的起到效果。

如果你长期养成了一种攀比的心理的话，可能一开始会比较痛苦。但是只要你从今天开始把这个戒掉，以后好处会显示出来的。至少很直白的理解他——你不用花那么多冤枉钱了，你就可以用来投资自己，做更多更有用的事情，从而让你真的能够过得更好。

攀比是一种证明，人生的计划也是一种证明。只要你相信了这个证明，你的焦虑感自然会下降。但是攀比通常都会被证明是不可持久而且具有破坏性的。所以如果你觉得焦虑，你不要马上就去做一些事情来让你过得好受一点，你应该去分析你自身的问题。

题。[一些具体的问题我可能刚好以前写过文章](#)，但是总的来说，你都应该好好利用你的大脑，切实地去做一些有意义的事情，而不要选择麻醉。麻醉是最简单的，而且也是最没有用的。

现在给各位成年人留一个作业。你们都知道要赚钱娶老婆养家一线城市买房子给小孩上学。当然大方向来看是没有问题的，但是你们知道这个方案，是如何在微观上对你的这个具体的（而不是统计意义上的）小孩起到帮助，从而你非这么做不可的吗？好好考虑一下，给个证明，不要让自己不知不觉落入攀比的泥潭无法自拔。

什么样的人能通过读计算机专业而获得成功？

一个成功的人生，当然是充满快乐的。然而现在哪怕你4年后进了BAT，面对BAT当地的高房价，一样吃瘪。所以靠工资逆袭想获得快乐在概率上是很低的，我们可以合理的假设，读计算机是没法让你赚钱得到快乐的。赚钱当然还是为了爽，你要是不是仅赚的钱不够，做的工作还让你不爽，还要努力赚钱让别人爽，看起来就很傻逼。

如果拿工资跟房价比的话，一样是买郊区的房子，或者一样是买市中心的房子，BAT的月薪折算成平米，跟汕头的码农收入是没有区别的（而且汕头的小吃又好吃又便宜赚到了！可惜我从小在那里长大吃太多有点麻木）。而且有钱的互联网公司一年一共才收多少人，而读计算机的人又有多少？搞不定几个著名的公司，最后这些人都是去干嘛？其实你们也应该关心这些问题，作为你们风险评估的一部分。

在十几年前，计算机行业一直都处于一种，熟悉业务跳甲方混的更好的状态。其实今天也没有变化。如果你最终无法把自己变得优秀从而去有钱的几个互联网公司混的话，你照样会进入那种“熟悉业务跳甲方混的更好”的公司，而这才是大多数程序员在干活的地方，保守估计高达90%。当然这90%，还不包括大学几年就发现自己学不下去怒而转专业的人，也不包括毕业后不从事计算机行业的人。

你们也不要再把互联网跟计算机联系的那么理所当然，互联网只是计算机的一小部分，知道的人多而已。很多地方都是要程序员去干的。譬如说硬件公司啦（澄海玩具厂听过没有，按一下就大吼大叫的功能是不是要人做？），进销存软件专业户啦（著名的譬如说金蝶），普通外包公司啦，汽车厂啦，各种比较智能的电器家具供应商啦，林林种种，都是需要大量的程序员去干活的。你们去吃饭的时候，看见柜台的姐姐在摆弄那个占位软件，一样是需要程序员做的。

除了这些地方以外，很多国企也是需要底薪码农去干活的。你们去移动办卡的时候，去银行开卡的时候，去给你们的汽车上保险的时候，看见柜台的哥哥姐姐们噼里啪啦地打字，那个软件要不要人写？这些都是程序员的工作。很多国企不从外包公司要软件，他们会自己写软件。

再低端（不一定代表更底薪）一点，你们如果开车回家，在大门口刷卡滴一下栏杆就抬起来的那个东西，要不要程序员写？说来我以前还差点被请去破解并更新一个这样的软件，因为原作者已经倒闭了。可惜我并没有这方面的专长，没代码我是改不了软件的。不过那东西一看，就是Delphi 6的作品。听说过这个没有？现在还有很多公司用Delphi的。高中的时候认识了一大堆Delphi的程序员，前几年我回去群里看看，居然还很多人这么多年仍然在写Delphi，仍然是一线员工。你们将来毕业说不定就做Delphi了，什么JavaScript啊MySQL啊一辈子可能都用不上，数据库还必须是天国的Borland Database Engine。

当然中国那么多人读计算机，10%其实也有很多人了，所以很多人都会有幻觉觉得自己一定是前10%，谁知道呢？要是人真的可以那么了解自己的话，那以前高考考前填志愿填跪了的人就不会有那么多了。

读计算机专业，本质上也是一个学习的过程。你学习理论知识也好，学习编程技巧也好，你将来做researcher也好做dev也好，最终你都是要靠开发出一个系统来完成你的工作。MSRA的researcher一样要写代码来证明自己的论文不是在吹逼，而且很多都还是要靠自己，小intern们或许可以帮帮忙，总的来说作用可以忽略。

所以要通过读计算机专业，获得成功，也就是让自己的生活充满快乐的话。如果你可以在赚很多钱之前就让自己的生活充满快乐，那简直就是赚到了。人家辛辛苦苦赚钱也是为了让自己快乐，你还没开始怎么赚钱，你就已经先快乐了，那么你的人生的意义不久已经达到了吗？后面不就可以去追求很多更有价值的事情了吗？

所以答案已经呼之欲出了。如果学习编程本身就可以让你觉得爽到浑身发抖的话，你就可以通过读计算机专业而获得成功。这种成功甚至不需要靠你的高收入，简直易如反掌，爽到不行啊。想想，你现在在读书那么贫穷，都已经这么快乐了。那将来万一找到了一份好工作，娶到了一个善解人意的漂亮的妹纸，生两个女儿，岂不是成仙了？你都已经这么快乐了，那你要用计算机来谋求什么，其实就是一件次要的、完全可以随缘的、顺其自然的事情了。你的人生已经成功了，还有什么好烦脑的呢？接下来你要做的，那就是让自己不断地爽下去，学一辈子计算机，写一辈子代码，爽到妈妈都不认识你了。

那对于那些不开心的人怎么办呢？既然你们想为了10%的、反正也买不起房子的伪高薪概率而让自己一直坐在电脑前加班，做一辈子自己不喜欢的事情，那你们自己看着办吧。我从来没体会过编程让自己不开心，我也帮不了你们。

那你怎样才能知道，你能不能从学习编程本身获得高潮呢？我只能说，你在填志愿之前，试试看就知道了。时间越长越准确。

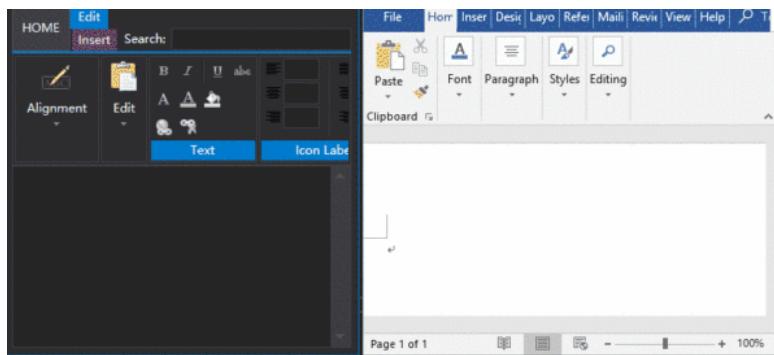
不过在此之前你们可以去看看，前些年从计算机行业退出的，去卖烧饼和水果的那些最终发现自己不喜欢编程的人，脸上洋溢着的幸福，重新长出来的头发，还有围绕在身边的老婆和女儿们，都是他们在写代码的时候，不曾拥有也无法想象的。

P.S.

我在软件工程有一个同学，也是不喜欢编程的，但是他在毕业的时候寻找到了自己的梦想，最后成功在毕业后几年，管一个消防大队。这么多年瘦小的身躯应该充满肌肉了吧。我觉得他的人生也十分成功。

GacUI实现把富文本复制为HTML格式

先看效果：



基本上就是按照 [MSDN的文档](#) 来实现。经过测试，写字板不支持HTML格式，所以我猜他是支持RTF格式的。RTF格式接下来做。所以这次的demo是复制到Word，因为也想不起来到底还有什么程序需要被粘贴超文本内容。显而易见地，并不是HTML的所有功能都可以使用。

总的来说，Windows的HTML格式是一个UTF-8的字符串，里面包含一个头和完整的HTML文件。你可以指定HTML文件的其中一部分作为剪贴板真正的内容，现在我不可能往剪贴板里面写废话，所以body的内容就被做了标记。

不过目前看来，Word会忽略标记以外的其他东西，一开始我尝试在header里面写一些css让html变得更短，然后发现全部当没看见。然后我就试图用ins和del来代替已经被HTML5 deprecate掉的u和s，发现Word竟然在看见del标记的时候就真的把内容删了（智能！），看见ins的时候真的打上了“插入”记号而不是下划线。但是我又不想用回u和s，所以干脆内联css，长就长吧，你们能复制多少文字。

生成HTML的过程简单粗暴，style里面会包含一些多余的东西。最后贴一下剪贴板的内容：

```
StartHTML:-1
EndHTML:-1
StartFragment:0000000210
EndFragment:0000000613
<!DOCTYPE html>
<html>
<header>
<title>GacUI Document 1.0</title>
<meta charset="utf-8"/>
</header>
<body>
<!--StartFragment--><p><span style="font-family:Microsoft YaHei UI; font-size:12px; color:#000000; ">This is a <span style="font-weight:bold; font-style:italic; color:#0000ff; ">link</span> to <a href="#">another page</a>.</span></p>
<!--EndFragment--></body>
</html>
```

CS新手如何以正确的态度开始学习

前言

填志愿的人应该差不多要点确定按钮了。反正既然你们已经入了坑，那我就告诉你们应该如何开始。美妙的幻想破灭得越早，对你越有帮助。那些单个程序已经倒了一万行的新手，如果你觉得自己的学习习惯不好的话，也可以借此机会互相切磋一下。写这篇文章的契机是我在开发GacUI的时候，被迫去读RTF2007的文档。这天杀的文档一共有300页——不过其实也不算很长，只是我已经有几年没有干过这种事情了。把程序写出来的时候，仿佛又有一种回到过去的感觉。

端正态度这个问题其实是这几年才出现的。在我刚开始学习编程的时候，应该是2000年，大家普遍的态度就是，有书就看，有电脑就借，光是有已经很了不起了，应该投入所有的精力在这上面才能对得起这个机会。当然现在情况已经不是这样了。而且人们被周遭的生活所培养出来的结果，就是越来越丧失耐心。然而学习CS最不能缺的就是耐心。

学习CS的日常是什么呢？大家不要觉得是玩游戏，也不是花里胡哨地摆弄界面，其实每天会发生的事情，就是坐在电脑前一整天，然后看书打字。这就是为什么说学习CS最重要的就是耐心，因为没有耐心，你就会想从电脑前离开。你一离开干别的事情一爽，你的时间就流失了，学习效率大幅下降。所以学得最快的那些人，通常就是觉得坐在电脑前写代码是最爽的，自然地不会去分心。当然学得快不意味着你就学的好，你还要坚持下去，连续20年都觉得写代码是最爽的，根本就不需要担心什么找工作，说不定你已经在创造工作岗位了（当然这不一定是指创业）。

一般来讲，“学习”要么就是看教程，要么就是看著作。现在的教程都是视频，不过我个人不是很推荐，因为手把手的东西看了会上瘾，这对你以后的成长很不利。我建议去看那些以书本形式出现的教程。你看书自然会遇到很多问题，譬如说作者可能讲漏了其中一个动作，或者用的软件版本跟你不一样等等。这些纯粹的操作上的问题，你要习惯于需要自己琢磨的状态。虽然这样一开始会慢一点，不过 [学习编程是以10年为单位的](#)，晚个10天又有什么所谓呢。

当然做事情不能矫枉过正，觉得既然慢慢来无所谓就去啃著作，这样也不太好。因为你可能会因为缺乏一些基本的知识导致你著作啃不下去。但是你又不可能不啃著作，因为这些中级入门材料时十分缺乏的。其实在早期世界范围内根本就不缺乏这样的材料，只是随着互联网的发达，人们不知道为什么就喜欢互相喷。你稍微写点什么，就有人抓住你辫子不放，仿佛把你比下去他就会成功一样。当然喷人也不是不行，只要你有理有据地喷，同时文明地表达你的观点，就像 [@Milo Yip](#)，没问题。为了区别，我把喷起来不像叶先生那么友好有道理的都叫“恶意喷人”。

跨越了初学者阶段进入下一个阶段的人是最多的（这里我一般把那些未来会放弃的初学者不当人看（逃），所以这个程度的文章自然也需求量最大。但是处于这个阶段的人，特别喜欢恶意互喷，所以搞到大家都不是很想分享自己的见解了。这就是为什么到了现在你会发现，这些文章找起来不像以前那么好找了。在200x年的时候，大家看对方不对，都是讲道理吵架的。现在人们吵架都是因为看你不爽而不是看你不对，所以动不动就到了你的私生活。我也很同情今年开始学CS的人。当然我并不是在讲人变坏了，而是因为科技的发达，能上网的人平均素质变低了，这是无法避免的（逃。就想买北京的房子过了2010年就变得很难一样。

正文

因为看教程实在是太简单没什么好说的，本文重点还是在教你怎么啃著作，这都是出于我自己的经验。这篇文章的一个前提，就是你坐得住。我觉得某种程度上我很幸运，在我中学的时候，不知道血栓的形成以及杀人机制，让我久坐在电脑前毫无负罪感，幸好年轻不会出事。现在大家都能上网了，就算你坐得住，你最好也隔一个小时，上来打一套太极拳，然后回去。反正写代码总是要思考的吗，打太极拳的时候也可以思考。不要干别的事情分心，除了吃饭拉屎睡觉。

只有坐得住才能啃著作，而怎样才能坐得住，一个最简单的办法就是切断跟外界的联系。你可以找一本书，然后去一个没有网络的地方，手机数据disable，然后开始看。如果你需要查资料，你可以去图书馆上网，反正超级便宜。总之要让你没办法access到各种娱乐信息。这是在你们还坐不住的时候的一种锻炼自己的方法。

在这里提醒一下，学CS最重要的还是写代码，你不会写代码懂得再多也没有用。所以你需要花更多的时间来练习，我一般推荐是10倍。你看书的时间加起来长达一年，然后写10年代码，然后你就出山了，非常科学。

啃著作会有一个常见问题，就是你不知道他在说什么。这是由于著作本身的特点决定的。他不是教程，所以书里面的信息，是不排序的，他会假设你什么都懂一点。当然只要花时间你肯定都能搞定。所以在乎我介绍三个常用方法。

一、看好几遍

这个方法是最简单粗暴的。假设一本著作的前置知识你已经都会了，这个时候来看著作。著作的章节之间也是有依赖关系的，但是有些时候这些依赖关系跟学习关系是不一致的。所以你第一篇看完可能云里雾里，知道了一些新概念，知道了他们的关系，但是就是不明白。这个时候你看第二遍，因为至少你已经知道每一个词是什么意思了，你就会知道更多的东西。一般看个两三遍我觉得你也懂得七七八八了。以后你就把这本书的目录背下来，需要什么你再去查就好了。

工具书也是同一个道理。很多人都说，著作需要的时候去查就好了，这是不对的。因为你如果不去看，大部分时候出现的情况，是你根本不知道要去查什么东西，关键字都不会组织，什么都看不到。所以还是要看。反正 [学习编程是以10年为单位的](#)，多看几本书又有什么所谓呢，又不是要赶着去投胎。著作看个七成熟，概念你都明白了，细节记不住没问题以后再查。要是你连概念都不明白，你什么都查不到。

我第一次明白这个事情，是我大概初三的时候。以前看过我博客的人估计知道，我是从一本图形学的书开始学习Visual Basic的。这本书很厉害，他定位的读者就是傻逼，所以连数学也会教你，现在都没有这么具有人文关怀的著作了。于是我看了一年，同时开始学习VB的语法和控件的知识，勉强可以做一些简单的图像处理，譬如说搞个凹凸效果啊，扫描个椭圆什么的。但是VB的性能很成问题，因为debug的时候不像其他语言，VB会选择把你的代码编译成P-Code然后模拟执行，连个exe都不产生。你不仅不优化，还要解释执行，性能低到不行。所以尽管exe速度还可以，但是调试的时候心情很糟糕。

另一个原因是，没什么人会真的用VB来搞这个，大家都使用C++在搞。我当时也想着干脆学个C++吧。学成之后，要是不能把人家的C++代码抄成VB，那我就去用C++好了。虽然日后跳槽去了Delphi，不过C++的确是在那个时候看的。那个时候大约是2002年左右，C++03都还不知道在哪里，看的是C++98的内容。

于是我就去书店找书，看到了一本Visual C++ 5.0 语法手册，上面印着个MSDN的logo，觉得很专业，我就买了。后来才知道这本书是直接从MSDN上打印出来的（mmp）。如果你们也去看MSDN的C++语法手册，就会发现它有个特点，就是他是按语法结构来描述C++的。这也就是说，当他一开始讲到表达式的时候，他已经在告诉你遇到模板的时候各种奇怪的表达式要怎么写了。我那个时候类是啥都不太清楚，怎么可能看的懂这个呢。所以第一遍看了几个月云里雾里，于是又看了第二遍，总算知道点东西了。后来又看了第三遍，终于把基本概念都搞明白，会用C++的语法来写一些简单的程序了。上了大学之后我又通过几本书学习了C++的各种奇技淫巧和细节，不过这些都是题外话了。

这件事情告诉我们，只要你掌握了一门编程语言，哪怕是C++，语法手册看三遍，你也可以学会。工具书著作没什么好怕的，你需要的只有耐心。

二、搭配各种导论文章看

当你看一本著作只是为了明白其中的一点点东西，而不是真的想去学会全部的时候，你需要的就是完全不同的阅读方法。就像最近GacUI要支持富文本复制到剪贴板里面变成RTF好让Word和Wordpad粘贴一样，找来找去都没什么靠谱的文章，估计新手们连RTF是啥都不知道。**RTF作为一种通用格式，早期WPS和Word都支持。WPS之所以后来跪了，就是因为他所见即所得的体验比起Word实在是差的太他妈远了。所以在WPS和Word都有盗版的情况下，眼睛雪亮的人民选择了Word。**不要相信什么微软靠盗版占有市场的这种胡说八道的东西，那个时候不管是谁家的软件都有盗版。价格都是0，谁赢了真的是质量取胜。不像现在，为了成功做一个人，你还得考虑一下软件成本。

GacUI的富文本是很基本的，但是RTF2007已经发展到了几乎可以表达Word的所有内容的状态了，复杂到连微软也没有一款软件是可以编辑100%的RTF内容的。你们如果也去看一下那个文档，就可以知道他妈的control word的数量之多，念一遍都可以让便秘的人从厕所坐到拉完屎。显然我需要的只是1%，那怎么办呢？

这个时候我当然不可能用看两遍RTF2007的方法来完成这个事情，我还得找一些其他资料。资料的来源有很多，譬如说我可以用Wordpad写一点简单的东西保存一下用记事本打开看看长什么样子，心中有个概念。然后随便遍几个问题来看看网上的人是如何解答的，对RTF有一个初步的了解。有了这些概念之后，我再去看RTF2007的文档，遇到一个章节就可以扫一眼判断到底这事不是我要的，然后迅速跳过所有我不需要的内容。这样速度就很快。我大概就花了两个小时吧，就把1%的内容从RTF2007里挑出来了，最后试验成功，就可以开始写代码了。

当然，大部分著作你是不能这么看的，因为学习编程不能太有目的性，在刚开始你不能挑说你想学什么不想学什么。编程涵盖的内容有很多，在你作为一个新手的时候，学习所有的知识都是为了你以后成为某一方面的专家而打下基础。哪怕你的梦想是去画网站，你也不可能完全不碰后端的，不然你到时候就会在protocol如何设计的角度跟同时吵起来。这就像PM经常搞出一些不切实际的需求给码农做一样，会被鄙视的。所以你什么都得学，不用学的太深入，只要能够写个10000行的、复杂点的程序就好了。

只有在你试图去解决一个特别狭窄的问题的时候，你才可以用这种方法来读著作，因为你知道估计以后再也用不上了，就像我生成RTF格式的字符串一样，我很难想到还有什么情况让我再去写一遍这个东西——说不定以后要写RTF parser呢，那个时候再说了。

三、一边看一边练习

有些书就适合这样看，譬如说大家喜闻乐见的《算法导论》。算法导论这本书，是不局限于任何一门语言的（Haskell等除外），也不是在介绍什么API的用法。这种书就不是纯工具，而是包含一些知识是需要你去学习的。你可以抱着一个目标（而不是目的）去看这本书，譬如说你就是想学会怎么用好语言里面提供的各种东西，所以你去学一下怎么开发他们，从各方面了解他们的特点，没有问题。

就像《算法导论》，光看就不是很合适，你还要去实践。当然我这里的实践，说的不是去做习题，做习题性价比太低。具体问题具体分析，具体到算法导论这本书，我推荐的办法，就是在理解了前面的关于复杂度的计算方法之后，把书里的数据结构的复杂度都背下来，把结构化为图形印在脑子里，最后把所有的伪代码都翻译成C++，写成STL那个样子。至此你的任务就完成了，你什么概念都明白了，以后需要什么细节你不记得了，马上就可以查到。

同理，《设计模式》和《编译原理》也是一样的。这些书都需要大量的实践来很好的掌握他们，所以你无论如何都得配合至少跟书一样厚的代码来练习。一本书500页，一页100行，也就是5万行而已，不一定够。

尾声

为什么要写这篇文章呢？很多CS新手都是因为不知道正确的学习方法，吊儿郎当，最后失败的。我觉得至少应该排除这一点干扰因素。如果你从第一天开始就用正确的态度来学习，从来只有好处没有坏处。学习CS，就是坐在电脑前看书写代码，每天8小时，10年过去你肯定已经是高手了，根本就不怕什么面试，Offer随便拿。你们志愿差不多都填了，距离入学还有两个月。如果你们可以跟我当年一样，放假的时候每天早上8点一只蹲到晚上12点，除了编程和生存什么都不干，我相信你们到了第一天开学的时候，至少掌握一门语言的基础知识，写点唬人的代码，已经没有问题了。如果你们可以一直坚持到毕业，至少毕业的时候就不会很难看。

中国的大学，反正都是乱来的。你们可以适当选择在保证期末及格的前提下，翘几节不重要的课（譬如什么马列毛邓三啊，我很推荐你们读马哲，但是应试就没必要了，先把书背下来，等你们长大了自然就懂了），在宿舍里写代码，就可以增加你们的学习时间，保证一天8个小时。

坚持7个学期，大概一万小时就凑足了。上课的时候可能没办法学习8个小时，你可以学习6个小时，放假的时候一天学习14个小时，平均下来还是每天8个小时，然后差不多也到了找工作的时候了，保证8000保底。

编程学习方法随笔目录

本目录已加入 [挖坑不填系列](#) 套餐

[vczh: CS新手如何以正确的态度开始学习](#) ↗ zhuanlan.zhihu.com



[vczh: 什么样的人能通过读计算机专业而获得成功?](#) ↗ zhuanlan.zhihu.com



[vczh: C++需要不断地练习 ——vczh](#) ↗ zhuanlan.zhihu.com



[vczh: 沈向洋: You Are What You Write, 大家都要看](#) ↗ zhuanlan.zhihu.com



[vczh: 神文系列 \(2\) ——喜鹊开发者](#) ↗ zhuanlan.zhihu.com



[vczh: 觉得很有必要再次推荐编程自学界](#)

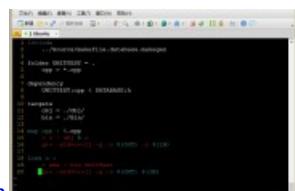
[vczh: 神文](#) ↗ zhuanlan.zhihu.com



[vczh: 最近问我职业规划的好像有点多](#) ↗ zhuanlan.zhihu.com



[vczh: 为什么我们需要学习\(设计\)模式](#) ↗ zhuanlan.zhihu.com



[vczh: 靠谱的代码和](#)

[DRY \(图片是GacUI\)](#) ↗ zhuanlan.zhihu.com



[vczh: 如何成为牛逼的程序员](#) ↗ zhuanlan.zhihu.com

GacUI_1.0_眼看着就要写完了_(2)

距离《[GacUI 1.0 眼看着就要写完了](#)》已经过去了5个月，对照一下当时列出来的TODO，以及[TODO.md](#)，明显可以看出放弃的

- 菜单项可以被绑定到列表上
 - 这主要是因为设计改了。菜单还是可以绑定的，就是要曲折一点。因为GacUI放菜单项的那块地方并不是真的只能放菜单项，你想放什么都可以。所以你就算放一个`<CustomControl>`，里面放个`<RepeatStack>`，然后写个`ItemTemplate`，每个东西里面放个菜单，菜单照样还是长菜单那个样子，该有的功能还是有。所以这个时候你只要把东西绑定到`<RepeatStack>`上面去就好了。所以就不专门做这个功能了。

做完了的

- 富文本框支持RTF/HTML剪贴板格式——还要一点小修补，不过已经基本上认为完成了
- 国际化/本地化支持
- 给所有列表控件加上一个给`ItemTemplate`用的“上下文”参数
- 让XML写的窗口和控件可以继承
- 把Visual State做的跟XAML一样好——这个被响应式布局代替了，还多了个Ribbon

还没做的

- 修现在黑皮肤窗口几个按钮没有响应属性变化的问题，和焦点的渲染
- 做一套新皮肤，运行时换肤
- 高DPI支持
- 窗口图标支持

新增的

- 一些跟C++代码生成相关的东西

进度喜人

等这些东西做完后，GacUI 1.0 计划内的功能也就全部完成了。接下来的事情。就是下面几件事：

- 重构文档生成工具，这次要把生成的文档重新组织一下，合并到英文版的Tutorial里面。以后GacUI就会有真正的文档了，而不是只能看一个光秃秃的类型参考和一堆Demo。
- 完成《GacUI的设计与演化》。大纲我已经想得差不多了，慢慢写。
- 看反馈修bug
- 重做网站

所有的这些事情完成之后，GacUI 1.0就可以正式发布了！偶也！

强连通分量与拓扑排序

前言

由于GacUI里面开始多处用上拓扑排序，我决定把之前 [瞎JB搞出来的算法](#) 换掉，换成个正式的。之前我自己弄了个写起来很简单的算法，然后每一处需要用到的地方我就重新做一遍。当然这样肯定也是不行的，我觉得也差不多积累到重构的临界点，于是重构一把。

我的需求是要在做拓扑排序的同时，识别出图的强连通分量。于是在经过短暂的考察之后，我选择了 [Kosaraju's algorithm](#)。这个算法设计的很精妙，虽然很简单，但是令我回味无穷。该算法claim说自己是线性的，虽然也没错，但是实际上为了构造出这个数据结构，本身花费的时间已经超过线性了，所以整个算下来并不是线性的。

GacUI需要用到拓扑排序的地方很多，包括但不限于：

CodePack.exe

一个把一堆C++代码打包成几个成对的h和cpp代码的工具。这里就需要拓扑排序。因为在配置文件里（譬如说 [这个](#)），我只定义了哪一些文件需要合并。而最后文件与文件的#include关系，是自动算出来的。拓扑排序在这里起到的作用，就是如果排序不成功，那我就要输出错误信息。

现在我输出错误信息只是告诉说你错了，并不能告诉你是谁跟谁搞在一起导致出错的。强连通分量在这里就起到了很好的效果，他识别出了循环引用的最小的集合，那么我就可以把这个集合输出到错误信息里，这样你就知道配置文件里面哪里写的不对。

Workflow 编译器

Workflow脚本语言支持C#那样子的class和struct。class可以继承，struct可以在成员里面引用别的struct。如果我们把class a继承自class b，和struct a用了struct b，都看成a依赖b的话，那么所有的class或者所有的struct就构成了一个图。这个图必须是偏序结构的，否则就意味着，要么你循环继承class，要么你虚幻嵌套struct，这都是错误的。

那强连通分量是什么作用呢？其实仍然是为了输出错误信息。如果你有一个很大的Workflow程序，我告诉你某个class循环继承了自己，看起来其实不是很友好。如果我可以告诉你到底是哪几个class互相继承，你改起代码来自然就方便多了。每一个强连通分量都代表了一个错误信息，很方便。

Workflow C++代码生成器

Workflow生成C++代码还有一些额外的要求。譬如说你在GacUI里面，指明了一个窗口的ref.CodeBehind属性为true，那么GacUI就会为你这个窗口单独生成一对C++文件，否则就全部加进大文件里。这样可以有效减少文件数量。你需要单独生成文件的理由，自然是你需要把自己的C++代码合并进这个窗口生成的C++代码里，就像流行的GUI库编辑器做的那样。典型的有事件处理函数，或者是你自己用C++添加的成员等等。

这就带来了两个问题。第一个问题是，如果你有三个窗口，a继承出b，而b继承出c。本来abc都是生成到同一个文件里面的，但是后来你给b加上了ref.CodeBehind=true，这会导致c也必须生成到一个独立的文件。因为如果ac在一个文件，b在另一个文件的话，你就没法正确#include。

显然，你ref.CodeBehind=true的一些窗口，使得ref.CodeBehind=false的一些窗口不一定可以全部放到一个头文件里。在这里识别出强连通分量就可以很好地减少分裂的头文件数量。当然并不是每一个强连通分量就是一个文件，这样也是很多余的。具体的方法我还没开始想，不过肯定是要水到渠成的问题，因为明显只要对每一个强连通分量按照一定的规则染色，就搞定了。

第二个问题是，Workflow的类可以有嵌套类，嵌套类也会影响生成文件的安排，但是就算你只有一个文件，还会带来另一个问题。譬如说我有这样的C++代码：

```
class Fuck : public Bitch::Dung
{
public: class Shit{};
};

class Bitch : public Fuck::Shit
{
public: class Dung{};
};
```

你会发现，不管你如何调整顺序，不管你如何向前声明，你都没办法让他编译通过。当然C#是不会有这个问题的，以C#和COM作为模板的Workflow自然也不会有这个问题。但是你真的这么写了，我就没办法替你生成C++代码。

那么自然，Workflow的C++代码生成器必须在这个时候报错。这里我们仍然要进行拓扑排序，但是图的每一个节点，其实就是每一个top level class和所有内部类的集合。在这里自然就是{Fuck, Fuck::Shit}以及{Bitch, Bitch::Dung}。在检查继承关系之后，我

们发现这两个节点是循环引用的，因此会被分配到同一个强连通分量里。如果排序的结果，有一个强连通分量有超过一个节点的话，那么就意味着这种代码没办法生成C++代码，因此就可以报错了。报错的时候，我又可以生成好看错误信息了。

实现

其他的我就不说了，还有很多。如果你们好好看了上面的维基百科的链接，就知道Kosaraju算法是表达为递归的。在敲代码之前，我也考虑过到底要不要把递归化为循环，让爆栈不容易发生。后来想想算了，因为这里的递归的层数，跟你C++代码#include的层数，和类继承的层数是一致的。如果你的Workflow类一共继承了1000层，那你也不要怪我GacGen.exe崩溃，我不管的（逃。因此我毅然选择了递归。

Vipp里面一共有三个文件：[PartialOrdering.h](#)、[PartialOrdering.cpp](#) 和 [TestPartialOrdering.cpp](#)。大家有兴趣的话就去看，里面有实现以及测试用例。

经过我的估算，这个类的三个主要函数的worst case复杂度分别是：

- `InitWithGroup`: $O(ElgV)$
- `InitWithFunc`: $O(V^2 + ElgV)$
- `Sort`: $O(V+E)$

总的来说，整个东西的复杂度还是会被控制在 $O(n \lg n)$ 或者 $O(n^2)$ ，还行。

之前瞎JB搞得算法的worst case是 $O(n^3 \lg n)$ ，看起来很吓人，不过因为我处理的图都是稀疏图，所以平均下来也不会这么难看。既然已经把靠谱的算法做进GacUI了，那么接下来就是把每一处用到垃圾拓扑排序的地方删掉，用新写的算法替换上。

尾声

写代码真是开心啊，每天都可以找到缺陷可以改进，每天都有代码可以写。希望有我同样热情的人，好好学习，不要被一些投机倒把的CS学生，把你们的大学学籍给挤掉，每一个喜欢编程的同学最终都能读上CS专业。

科研人员如何避免自己的计算程序在运行的时候被系统重启 打断

有两种办法。

NOTICE: 熟练掌握第二种办法，有可能导致贵机构因为长时间跑计算不结束而被类似wannacry等，开始运行程序的时候还没被发现的病毒入侵，更新装得太慢，所有工作毁于一旦。

一、把自己的程序编译为Windows Service而不是普通应用程序，然后你喜欢用NETWORK SERVICE启动也可以，用自己的账号启动也可以。Windows可以设置为重启的时候让用户自动登陆（仍然保持锁屏状态）。这样你的Windows Service就可以随着系统重启的时候继续运行。你只要把程序写成像断点续传那样的就可以，重启的时候之前计算过的东西可以尽量恢复。

二、自己监听操作系统关机的信号，然后调用ShutdownBlockReasonCreate和ShutdownBlockReasonDestroy。后面那个API如果你不在事情做完的时候调用的话，会导致Windows无法关机，代码好好写（逃

Easy!

至于大家下片也是一样的。一个合格的P2P软件，会使用第一种办法，安装的时候替你把这些步骤做完。这样的软件，不管Windows是因为你想要安装软件重启，还是你踢了机箱的Reset按钮，都会完美完成任务。除非你开机后进了安全模式。