

# OS 课设-任务管理器实现

王凌峰<sup>1</sup>

2021 年 12 月 17 日

## 简介

概览

使用的 API 与库

## 总体结构

需求分析

开发方法与软件结构

模块独立性分析

## win32 API

在程序中使用 win32 API 的方法

win32\_common 封装的功能

win32\_common 的实现

pid\_thrd 的获取

进程名的获取

module 路径的获取

进程所属用户的用户名的获取

结束进程

CPU 型号的获取

创建新进程

Taskmgr 的权限提升

## GUI

界面分解

signal 与 slot

Taskmgr 类的实现

详细信息页的实现

详细信息视图

小图标视图

用户页的实现

性能页的实现

load 类的实现

## Taskmgr 的部署

## 功能与性能分析

功能分析

与 Win10 任务管理器的对比

性能分析

## 简介

### 概览

### 使用的 API 与库

## 总体结构

### 需求分析

### 开发方法与软件结构

### 模块独立性分析

## win32 API

### 在程序中使用 win32 API 的方法

### win32\_common 封装的功能

### win32\_common 的实现

#### pid\_thrd 的获取

#### 进程名的获取

#### module 路径的获取

#### 进程所属用户的用户名的获取

#### 结束进程

#### CPU 型号的获取

### 创建新进程

### Taskmgr 的权限提升

## GUI

### 界面分解

### signal 与 slot

### Taskmgr 类的实现

### 详细信息页的实现

#### 详细信息视图

#### 小图标视图

### 用户页的实现

### 性能页的实现

#### load 类的实现

## Taskmgr 的部署

## 功能与性能分析

### 功能分析

### 与 Win10 任务管理器的对比

### 比

### 性能分析

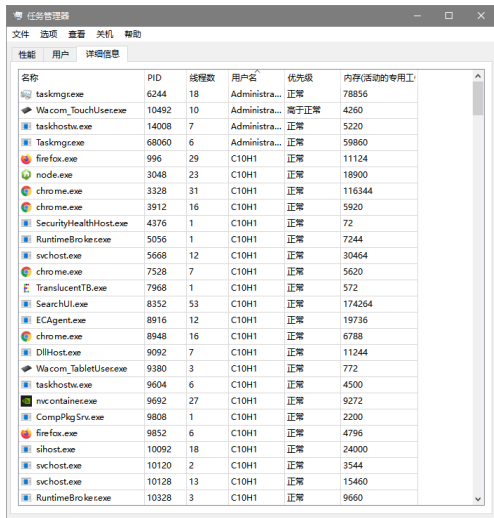
# 概览

此任务管理器使用了 Win32 API 和 Qt 的 GUI 库，语言是 C++。仿照 Windows 10 任务管理器的风格实现了：

- ▶ 显示所有进程的信息（图标，可执行文件名称，PID，优先级，内存使用，线程数量，所属用户）
- ▶ 可选择以详细列表或图标形式显示进程信息，以详细列表显示时可以以任何一个属性升序或降序排序
- ▶ 通过右键菜单结束选中进程
- ▶ 显示不同用户开启的进程，支持按用户展开和折叠
- ▶ 创建新进程。可以向新进程传递命令行参数，如果指定的路径不是可执行文件，能根据文件的类型调用合适的应用程序打开

- ▶ 可以选择 3 种预设的刷新频率，可以立即刷新
- ▶ 可通过菜单栏关机或注销
- ▶ 可使窗口始终在最前
- ▶ 读取 CPU 逻辑内核个数，在图表中实时显示 60s 内各个内核及平均的利用率
- ▶ 显示 CPU 的具体型号，基准频率以及所有逻辑内核平均的实时频率
- ▶ 实时显示系统中进程、线程、句柄总数，以及系统正常运行时间
- ▶ 显示内存总量和内存利用率，并在图表中实时显示 60s 内已使用内存的变化情况
- ▶ 实时显示使用中、可用、已提交、已缓存、分页缓冲池，非分页缓冲池的大小
- ▶ 帮助菜单栏中有关于本软件和关于 Qt 的按钮

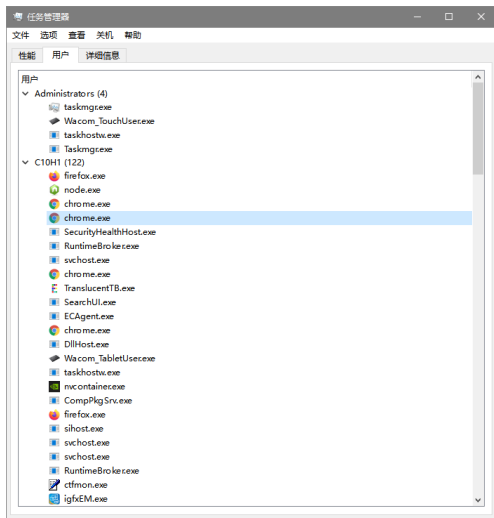
此任务管理器实现的主要是 Windows 10 任务管理器的“性能”、“用户”和“详细信息”三个页面。GUI 风格见图1 (p6)、图2 (p7)、图3 (p8)。



The screenshot shows the Windows Task Manager application with the 'Details' tab selected. The processes are sorted by 'User Name' in ascending order. The table contains the following data:

名称	PID	线程数	用户名	优先级	内存(活动的专用工
taskmgr.exe	6244	18	Administra...	正常	78856
Wacom_TouchUser.exe	10492	10	Administra...	高于正常	4260
taskhostw.exe	14008	7	Administra...	正常	5220
Taskmgr.exe	68060	6	Administra...	正常	59860
firefox.exe	996	29	C10H1	正常	11124
node.exe	3048	23	C10H1	正常	18900
chrome.exe	3328	31	C10H1	正常	116344
chrome.exe	3912	16	C10H1	正常	5920
SecurityHealthHost.exe	4376	1	C10H1	正常	72
RuntimeBroker.exe	5056	1	C10H1	正常	7244
svchost.exe	5668	12	C10H1	正常	30464
chrome.exe	7528	7	C10H1	正常	5620
TranslucentTB.exe	7968	1	C10H1	正常	572
SearchUI.exe	8352	53	C10H1	正常	174264
ECAgent.exe	8916	12	C10H1	正常	19736
chrome.exe	8948	16	C10H1	正常	6788
DllHost.exe	9092	7	C10H1	正常	11244
Wacom_TabletUser.exe	9380	3	C10H1	正常	772
taskhostw.exe	9604	6	C10H1	正常	4500
nvccontainer.exe	9692	27	C10H1	正常	9272
CompPkgSrv.exe	9808	1	C10H1	正常	2200
firefox.exe	9852	6	C10H1	正常	4796
sihost.exe	10092	18	C10H1	正常	24000
svchost.exe	10120	2	C10H1	正常	3544
svchost.exe	10128	13	C10H1	正常	15460
RuntimeBroker.exe	10328	3	C10H1	正常	9660

图: 详细信息页-详细表格视图 (以用户名升序排序)



图：用户页（已展开）

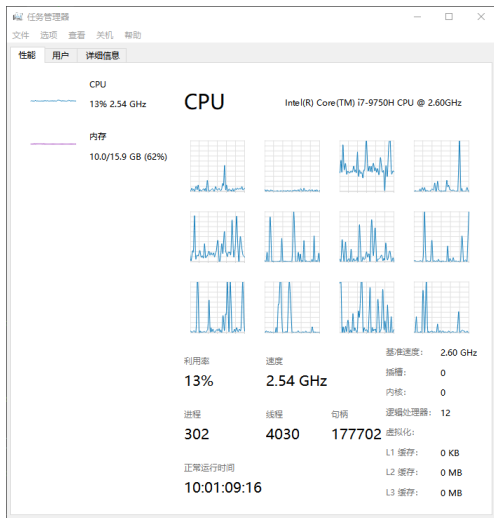


图: 性能页-CPU



没有实现的是应用程序管理和联网状态。由于 win10 任务管理器没有专门的联网状态信息，而在性能页中显示实时网速成本太高，所以我决定不实现。应用程序管理没有实现的原因是我没观察到 win10 任务管理器的进程页中判定一个进程属于应用还是后台进程的规则；图形界面程序并不都属于应用，而应用中也不全是图形界面程序。

为了方便，在下文中称此任务管理器为 Taskmgr。

# 使用的 API 与库

Taskmgr 主要分为两部分：GUI 部分，调用 Windows 接口的部分。

调用 Windows 接口的部分使用的操作系统 API 是 win32 API。win32 API 提供了大量关于系统的 API，C++ 是官方推荐的 win32 API 调用语言之一。

GUI 使用 Qt 的 GUI 库。Qt 是庞大的跨平台 C++ 开发框架，广泛应用于 GUI 开发。Qt 库含有各种模块，其中可能也有对 Windows API 的包装，但此项目只使用了 Qt 的 GUI 库，与 GUI 无关的部分都由另一个部分完成。

代码总行数：2354。

## 简介

## 概览

## 使用的 API 与库

## 总体结构

## 需求分析

# 开发方法与软件结构

## 模块独立性分析

## win32 API

## 在程序中使用 win32 API 的方法

## win32\_common 封装的功能

## win32\_common 的实现

## pid\_thrd 的获取

## 进程名的获取

## module 路径的获取

## 进程所属用户的用户名的获取

## 结束进程

## CPU 型号的获取

## 创建新进程

## Taskmgr 的权限提升

## GUI

## 界面分解

## signal 与 slot

## Taskmgr 类的实现

## 详细信息页的实现

[详细信息视图](#)

小图标视图

## 用户页的实现

## 性能页的实现

## load 类的实现

## Taskmgr 的部署

## 功能与性能分析

## 功能分析

## 与 Win10 任务管理器的对

比

## 性能分析

# 需求分析

需求主要就是前文提到的已经实现的功能加上以下还未实现的比较次要的功能：

- ▶ 显示 CPU 物理内核数量
- ▶ 显示 CPU L1, L2, L3 缓存大小
- ▶ 显示 CPU 插槽数
- ▶ 显示 CPU 是否支持虚拟化
- ▶ 显示内存插槽数
- ▶ 显示内存频率
- ▶ 显示内存外形规格
- ▶ 显示为硬件保留的内存

# 开发方法与软件结构

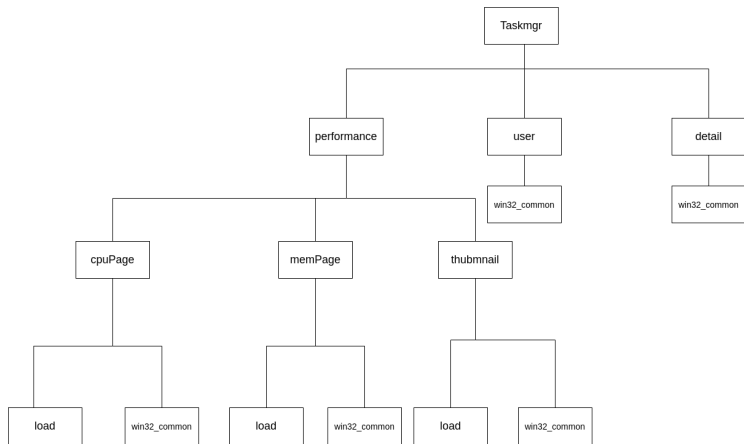
主要采用面对对象的开发方法。

类间没有继承关系，比较简单。具体见表1（p13）。

类	描述
Taskmgr	主窗口
performance	性能页
cpuPage	性能页中 CPU 页面
memPage	性能页中内存页面
thumbnail	性能页中左侧缩略图
load	在性能页中绘制图表的类
user	用户页
detail	详细信息页
win32_common	调用 win32 API 的封装类

表: 任务管理器项目中的类

## 不同模块的层次方框图



`win32_common` 类是对 win32 API 的封装，提供了各种 `get` 方法获取系统相关信息，还避免了 win32 API 返回的 Windows 定义类型（如 `TCHAR`, `DWORD` 等）带来的麻烦，因为所有 Windows 定义类型都限制在类的内部。`win32_common` 所有公开函数的传入参数和传出参数都是标准 C++ 类型（如 `std::string`, `std::vector` 等）。`win32_common` 内所有函数的声明见第3 节 (p23)。

win32 API 中查询函数返回的通常是一个结构体，结构体中包含几种相关的信息。为了不在调用几种请求的数据性质类似的 `get` 函数时重复调用相同的 API，`win32_common` 内设置了缓冲区。与其他可能被请求的信息存在同一个结构体的数据都存入 `win32_common` 内的缓冲区，调用相应的 `get` 方法时直接返回缓冲区中的值。缓冲区的刷新通过 `win32_common::refresh()` 完成。



其余的类都是继承自 QWidget 的 Qt 类，作用是获取、显示系统数据和与用户交互。这些类代表的图形组件之间存在组成关系，例如主窗口由性能页、用户页与详细信息页组成，性能页又由 CPU 页面、内存页面和略缩图组成，等等。具体组成关系由表1（p13）应该容易理解。

## Taskmgr 中的数据流图非常简单:

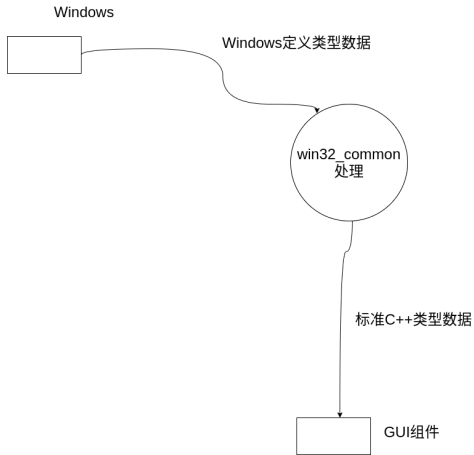


图: DFD

所有类都有一个 `refresh()` 函数 (`Taskmgr` 类中是 `do_refresh()`)，用于刷新自身及包含的所有组件。`Taskmgr` 的刷新通过这种递归刷新实现

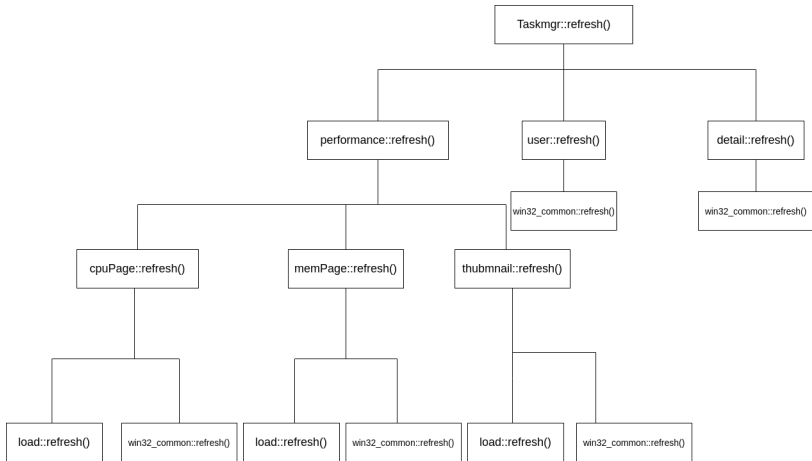


图: refresh 调用关系

# 模块独立性分析

为了避免每个的需要访问 `win32_common` 类中非静态类型（具体非静态函数见第3节（p23））的类在被其 `parent` 类（Qt 中 `parent` 与 `child` 的概念见第4节（p49））调用 `refresh()` 时都刷新一次 `win32_common` 缓存区，增加不必要的开销，将 `win32_common` 类移入 `Taskmgr`，其他类只保留指向 `win32_common` 的指针。刷新时，由 `Taskmgr` 先对 `win32_common` 进行一次刷新，其余类刷新时只调用 `get` 方法即可。

不过这样就使其他类访问了 `Taskmgr` 类内部的数据，出现了内容耦合。内容耦合应该避免，但在这个项目中我认为是合理的，主要原因有：

1. 本项目规模很小，存在一定的高耦合也不是不能接受的
2. 消除这种耦合十分容易，只要修改有组成关系的 Qt 类的构造函数和析构函数即可
3. 如果程序在后续需要增加新功能，很可能仍然使用扩充后的 `win32_common` 类，这种内容耦合的结构实际上表明了软件的性质，没有必要强行遵循一般的模块独立性准则
4. 最重要的是这种结构显著提升了程序性能。在第??节 (p??) 性能分析中，可以看到 win32 API 调用是 `Taskmgr` 的性能瓶颈，应该尽量减少 `win32_common::refresh()` 调用的次数。

## 简介

## 概览

## 使用的 API 与库

## 总体结构

## 需求分析

## 开发方法与软件结构

## 模块独立性分析

## win32 API

## 在程序中使用 win32 API 的方法

## win32\_common 封装的功能

## win32\_common 的实现

## pid\_thrd 的获取

## 进程名的获取

## module 路径的获取

## 进程所属用户的用户名的获取

## 结束进程

## CPU 型号的获取

## 创建新进程

## Taskmgr 的权限提升

## GUI

## 界面分解

## signal 与 slot

## Taskmgr 类的实现

## 详细信息页的实现

[详细信息视图](#)

小图标视图

## 用户页的实现

## 性能页的实现

## load 类的实现

## Taskmgr 的部署

## 功能与性能分析

## 功能分析

## 与 Win10 任务管理器的对

比

## 性能分析

## 在程序中使用 win32 API 的方法

win32 API 以在源代码中 `#include` 头文件并指定库文件的方式使用。一些 Windows 上的 C++ 标准库，如 MinGW 自带一些 windows 头文件，但由于不是 C++ 标准库，不能保证不缺少文件以及文件的新旧。应该使用 Windows SDK 中的头文件与库。Taskmgr 项目使用 qt creator 作为 IDE，项目管理文件是一个 .pro 文件。为了使用 win32 API，需要在 .pro 文件中指明头文件路径与需要使用的库：

```
win32: LIBS += -lPdh -lpowrprof  
INCLUDEPATH += 'D:/Program Files  
(x86)/Windows Kits/10/Include/10.0.22000.0'  
DEPENDPATH += 'D:/Program Files (x86)/Windows  
Kits/10/Include/10.0.22000.0'
```

## win32\_common 封装的功能

win32\_common 要提供的在系统运行中可能实时变化的 Windows 的属性以及这些属性所属的结构体见表2 (p25)。找出所有含有一个以上 win32\_common 需要提供的动态变化的属性的结构体, 这些结构体中 win32\_common 需要的属性可能在一个刷新周期内被 win32\_common 访问不止一个, 所以需要全部放进缓存区。由于与 win32\_common 实例的状态有关, 缓存区中属性对应的 get 函数自然是非静态的, 而其他的都是静态函数。



含义	Windows 结构体中的属性
PID	PROCESSENTRY32.th32ProcessID
进程的线程数	PROCESSENTRY32.cntThreads
进程映像文件名称	(无, 直接由 GetModuleBaseName() 获得)
进程映像文件路径	(无, 直接由 GetModuleFileNameEx() 获得)
进程优先级	(无, 直接由 GetPriorityClass() 获得)
进程占有的物理内存	PROCESS_MEMORY_COUNTERS.WorkingSet
进程所属的用户名	(无, 由 LookupAccountSid() 获得)
CPU 时钟频率 (MHz)	PROCESSOR_POWER_INFORMATION.CurrentFrequency
CPU 利用率	PDH_FMT_COUNTERVALUE
系统中进程数	PERFORMANCE_INFORMATION.ProcessCount
系统中线程数	PERFORMANCE_INFORMATION.ThreadCount
系统中句柄数	PERFORMANCE_INFORMATION.HandleCount
系统启动以来的时间 (ms)	(无, 直接由 GetTickCount() 获得)
物理内存大小 (B)	PERFORMANCE_INFORMATION.PhysicalTotal
可用物理内存 (B)	PERFORMANCE_INFORMATION.PhysicalAvailable
已提交内存 (B)	PERFORMANCE_INFORMATION.CommitTotal
系统已缓存内存 (B)	PERFORMANCE_INFORMATION.SystemCache
分页缓冲池 (B)	PERFORMANCE_INFORMATION.KernelPaged

## 全部的函数声明见表3 (p26)。

---

bool	refresh()
const std::map<unsigned long, unsigned long long>&	getPid_Thrdcnt() const
static bool	getProcName(unsigned &name)
static bool	getProcPath(unsigned &path)
static bool	getProcCla(unsigned long &cla)
static bool	getProcMem(unsigned long &mem)
static bool	getProcUserName(unsigned &username)
static bool	killProc(const unsigned long)
unsigned	getLogicCpuCnt() const
double	getCpuLoad(unsigned long)
unsigned long long	getProcCnt() const
unsigned long long	getThrdCnt() const

大部分 `get` 函数的返回值是 `bool`，指示是否运行成功。另外，所有函数中在获取要求的属性前将输出变量设置为能够指示成功与否的值。例如在 `getProcName()` 中：

```
60 bool win32_common::getProcName(unsigned long
    pid, std::string &name)
61 {
62     name = std::string("unknown");
```

在 `Taskmgr` 中，使用后一种方式，函数运行失败后使用 `"unknown", 0` 这类值。

## win32\_common 的实现

接下来描述 win32\_common 中一些函数的实现思路。

## pid\_thrd 的获取

pid\_thrd 是 win32\_common 内部的 map 容器，包含所有进程 PID 到其线程数的映射。

首先向 CreateToolhelp32Snapshot() 的 dwFlags 参数传递 TH32CS\_SNAPPROCESS 获得系统中所有进程的快照<sup>2</sup>，再用 Process32First() 和 Process32Next() 取出系统快照中每个进程的信息，依次存入 PROCESSENTRY32 结构<sup>3</sup>。主要代码见图6 (p30)。

---

<sup>2</sup><https://docs.microsoft.com/en-us/windows/win32/api/tlhelp32/nf-tlhelp32-createtoolhelp32snapshot>

<sup>3</sup><https://docs.microsoft.com/en-us/windows/win32/api/tlhelp32/nf-tlhelp32-process32first>

```
303     HANDLE hProcSnap;  
304     PROCESSENTRY32 pe32;  
305  
306     hProcSnap = CreateToolhelp32Snapshot(  
307         TH32CS_SNAPPROCESS, 0);  
308     if(hProcSnap == INVALID_HANDLE_VALUE)  
309         return false;  
310  
311     pe32.dwSize = sizeof(PROCESSENTRY32);  
312     bool retval = Process32First(hProcSnap, &  
313         pe32);  
314     while (retval) {  
315         pid_thrd[pe32.th32ProcessID] = pe32.  
316             cntThreads;  
317         retval = Process32Next(hProcSnap, &  
318             pe32);  
319     }  
320     CloseHandle(hProcSnap);
```

一个 `PROCESSENTRY32` 结构包含一个进程的相关信息，其中的 `th32ProcessID` 和 `cntThreads` 字段分别是该进程的 PID 和线程数<sup>4</sup>，在 `win32_common::refresh()` 中将映射关系存入 `map` 容器即可。

---

<sup>4</sup><https://docs.microsoft.com/en-us/windows/win32/api/tlhelp32/ns-tlhelp32-processentry32>

不分别获取 PID 和对应的线程数是因为 Windows 似乎没有提供通过 PID 获取线程数的 API，如果不通过 `PROCESSENTRY32` 或 `THREADENTRY32`<sup>5</sup> 预先保存系统快照中的 (PID, 线程数) 对，就需要在获取线程数时遍历线程的快照，计算创建它的进程的 PID 与请求的 PID 相同的数量，复杂度为  $O(N^2)$ 。

---

<sup>5</sup><https://docs.microsoft.com/en-us/windows/win32/api/tlhelp32/ns-tlhelp32-threadentry32>



# 进程名的获取

Windows 中，Module 是一个可执行文件或 DLL，每个进程都由一个或多个 module 组成<sup>6</sup>，进程的名称就是该启动该进程的可执行文件或 DLL 文件的文件名。

GetModuleBaseName() 接收一个进程句柄和一个模块句柄，返回该模块的模块名<sup>7</sup>。模块句柄的获得通过

EnumProcessModules，设置 cb 参数为一个模块句柄的大小以获得一个模块句柄<sup>8</sup>。

---

<sup>6</sup><https://docs.microsoft.com/en-us/windows/win32/psapi/module-information>

<sup>7</sup><https://docs.microsoft.com/en-us/windows/win32/api/psapi/nf-psapi-getmodulebasenamew>

<sup>8</sup><https://docs.microsoft.com/en-us/windows/win32/api/psapi/nf-psapi-enumprocessmodules>

OpenProcess() 根据 PID 返回对应进程的句柄。值得注意的是需要在 OpenProcess() 的 dwDesiredAccess 参数中指定对进程对象的访问权限<sup>9</sup>。为了获取进程的模块，需要 PROCESS\_QUERY\_INFORMATION 和 PROCESS\_VM\_READ 权限<sup>10</sup>。

---

<sup>9</sup><https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess>

<sup>10</sup><https://docs.microsoft.com/en-us/windows/win32/procthread/process-security-and-access-rights>

代码如下:

```
60 bool win32_common::getProcName(unsigned long
    pid,std::string &name)
61 {
62     name = std::string("unknown");
63     TCHAR szProcessName[MAX_PATH] = TEXT("<
        unknown>");
64     HANDLE hProc = OpenProcess(
        PROCESS_QUERY_INFORMATION |
65                                     PROCESS_VM_READ
        , //needed
        for access
        name
66                                     FALSE,
67                                     pid);
68     if (hProc == nullptr)
69         return false;
70
```

## module 路径的获取

这个功能在获取进程的图标时使用。

类似于 `GetModuleBaseName`, `GetModuleFileNameEx` 返回进程的模块的完整路径<sup>11</sup>。

Windows 中用 SID (security identifier) 标识用户 (user)、组 (group) 及计算机账户 (computer accounts)<sup>12</sup>, 为了获取进程所属的用户的名称, 要先获取进程所属用户的 SID, 然后通过 `LookupAccountSid()` 得到 SID 对应的用户名<sup>13</sup>。

---

<sup>11</sup><https://docs.microsoft.com/en-us/windows/win32/api/psapi/nf-psapi-getmodulefilenameexw>

<sup>12</sup><https://docs.microsoft.com/en-us/windows/win32/secgloss/s-gly>

<sup>13</sup><https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-lookupaccountsida>

进程所属用户的 SID 的获取通过向 `GetTokenInformation()` 的 `TokenInformationClass` 参数传递 `TokenOwner` 得到<sup>14</sup>。这个 API 接收一个 access token 的句柄为参数，特定进程 access token handle 的获取通过 `OpenProcessToken()`<sup>15</sup>，而该 API 接收的 `ProcessHandle` 参数由前面介绍的 `OpenProcess()` 得到，其中 `dwDesiredAccess` 为 `PROCESS_QUERY_INFORMATION`。

---

<sup>14</sup><https://docs.microsoft.com/en-us/windows/win32/api/securitybaseapi/nf-securitybaseapi-gettokeninformation>

<sup>15</sup><https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocesstoken>

代码如下:

```
157 bool win32_common::getProcUserName(unsigned
    long pid,std::string &username)
158 {
159     username = std::string("unknown");
160     HANDLE hProc = OpenProcess(
        PROCESS_QUERY_INFORMATION,
161                                     FALSE,
162                                     pid);
163     if (hProc == nullptr)
164         return false;
165
166     HANDLE hToken;
167     if (!OpenProcessToken(hProc,
        TOKEN_ALL_ACCESS,&hToken))
168         return false;
169
170     CloseHandle(hProc);
```

## 结束进程

结束进程 API 为 `TerminateProcess()`，获得进程句柄时需要向 `OpenProcess()` 的 `dwDesiredAccess` 参数传递 `PROCESS_TERMINATE`<sup>16</sup>。

代码如下：

```
189 bool win32_common::killProc(unsigned long pid)
190 {
191     HANDLE hProc = OpenProcess(
192         PROCESS_QUERY_INFORMATION |
193         PROCESS_TERMINATE,
194         FALSE,
195         pid);
196     if (hProc == nullptr)
197         return false;
198     bool rtval = TerminateProcess(hProc, -1);
199     CloseHandle(hProc);
200     return rtval;
```

## CPU 型号的获取

`__cpuid()` 产生 x86 和 x64 上可用的 CPU 指令，查询 CPU 支持的特性和 CPU 的类型<sup>17</sup>。`__cpuid()` 接收 `function_id` 参数指定要查询信息的类型。当其为 `0x80000002`, `0x80000003`, `0x80000004` 时，分别表示 CPU 制造商 (manufacturer), 型号 (model), 基准频率 (Clockspeed)。

代码如下：

```
249     std::array<int, 4> integerBuffer = {};
250     constexpr size_t sizeofIntegerBuffer =
        sizeof(int) * integerBuffer.size();
251     std::array<char, 64> charBuffer = {};
252     constexpr std::array<unsigned long long,
        3> functionIds = {
253         0x8000'0002,    // Manufacturer
254         0x8000'0003,    // Model
255         0x8000'0004     // Clockspeed
256     };
257
```



## 创建新进程

ShellExecuteA() 能对文件做一些操作，操作的具体类型由参数 lpOperation 指定。lpOperation 可以为 edit, open, runas 等，例如对文档类型 (document) 可以使用 edit 打开相应的编辑器，对可执行文件可使用 open 执行文件<sup>18</sup>。

当 lpOperation 为 NULL 时，系统自动判断 lpFile 指向路径的文件或文件夹的类型，并执行该可执行文件或用合适的应用打开文档。这和 Win10 任务管理器中“运行新任务”菜单栏的功能十分类似。因此 Taskmgr 中创建新进程使用的 API 就是 ShellExecuteA()。win32\_common::openProc() 代码如下：

---

<sup>18</sup><https://docs.microsoft.com/en-us/windows/win32/api/shellapi/nf-shellapi-shellexecutea>

```
4 bool win32_common::openProc(const char *path,
    const char *params,bool asAdmin)
5 {
6     const char *verb = asAdmin ? "runas" :
        NULL;
7     if ((long long)ShellExecuteA(
8         NULL,
9         verb,
10        path,
11        params,
12        NULL,    // current working
            directory
13        SW_SHOWNORMAL
14        ) <= 32)
15         return false;
16     return true;
17 }
```

其中 verb 设置为"runas" 时，表示以管理员权限打开应用程序<sup>19</sup>。这在 3.4 小节（p45）Taskmgr 提升权限时会用到。

---

<sup>19</sup><https://docs.microsoft.com/en-us/windows/win32/api/shellapi/nf-shellapi-shellexecutea>

Taskmgr 的创建新进程实现了启动可执行文件的同时传递参数的功能。Qt 弹出“运行新任务”窗口后调用 `Taskmgr::parsePath()` 分析接受到的字符串（具体见节4 (p49)），从用户输入中分辨出文件路径和传递的参数，分别存于 `std::string &prog` 与 `std::string &param` 中。Taskmgr 再将这两个字符串转换为 C 字符串并作为参数传递给 `win32_common::openProc()` 函数，即可在 Taskmgr 中以特定命令行参数启动新进程。

# Taskmgr 的权限提升

为了获得足够的权限访问某些进程数据及结束某些进程，Taskmgr 需要获得管理员权限。Windows 在用户登陆时生成 access token，access token 包含该用户及其用户组的权限 (privileges)，以该用户身份执行的所有进程都包含 access token 的一个副本。Windows 通过 access token 验证进程执行的操作<sup>20</sup>。所以为了执行必要的操作，access token 中需要包括对应的 privilege。但 access token 中的 privilege 只能使能/激活 (enable) 已存在的权限<sup>21</sup>，也就是说进程的权限只可能在创建前指定。

---

<sup>20</sup><https://docs.microsoft.com/en-us/windows/win32/secauthz/access-control-components>


<sup>21</sup><https://docs.microsoft.com/en-us/windows/win32/secbp/changing-privileges-in-a-token?redirectedfrom=MSDN>

有几种以管理员权限运行程序的方法，例如右键以管理员身份运行，在 manifest 文件中指定 execution level<sup>22</sup>，或者使用 Windows 自带的 runas 命令让普通用户以管理员身份执行程序 and 命令。

为了方便，Taskmgr 在启动时调用

win32\_common::openProc()，以管理员身份启动 Taskmgr.exe 的另一个副本后结束原进程。win32\_common::openProc() 中根据传递的参数设置 verb 为"runas"，并将"noadmin" 作为参数传递给新进程（即 Taskmgr.exe 的副本），新进程接收到此参数后跳过 win32\_common::openProc() 代码段，进入正常运行。这个行为是在 main.cc 中实现的：

---

<sup>22</sup><http://msdn.microsoft.com/en-us/library/bb756929.aspx> 

```
9  int main(int argc, char *argv[])
10 {
11     if (!(argc == 2 && !strcmp(argv[1], "
        noadmin")))) {
12         win32_common::openProc(argv[0], "
            noadmin", true);
13         exit(0);
14     }
15     QApplication a(argc, argv);
16     Taskmgr w;
17     w.show();
18     return a.exec();
19 }
```

## 简介

## 概览

## 使用的 API 与库

## 总体结构

## 需求分析

# 开发方法与软件结构

## 模块独立性分析

## win32 API

## 在程序中使用 win32 API 的方法

## win32\_common 封装的功能

## win32\_common 的实现

## pid\_thr 的获取

## 进程名的获取

## module 路径的获取

## 进程所属用户的用户名的获取

## 结束进程

## CPU 型号的获取

## 创建新进程

## Taskmgr 的权限提升

## GUI

## 界面分解

## signal 与 slot

## Taskmgr 类的实现

## 详细信息页的实现

[详细信息视图](#)

### 小图标视图

## 用户页的实现

## 性能页的实现

## load 类的实现

## Taskmgr 的部署

## 功能与性能分析

## 功能分析

## 与 Win10 任务管理器的对

比

## 性能分析



# 界面分解

Qt 中有四种基本的布局：

- ▶ QVBoxLayout，垂直布局
- ▶ QHBoxLayout，水平布局
- ▶ QGridLayout，网格布局
- ▶ QFormLayout，表单布局

为了仿照 Win10 任务管理器的界面，根据 Qt 中的布局对 Win10 任务管理器进行一些分解。见图10（p50）。

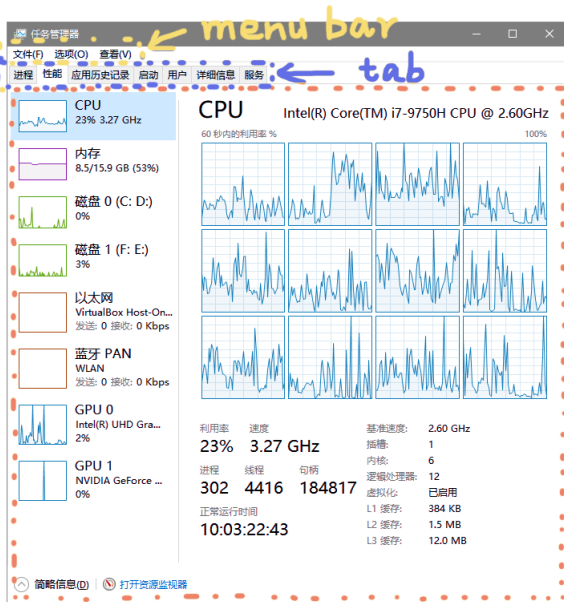


图: Win10 任务管理器的总体布局

性能页、用户页和详细信息页的切换可以使用 `QTableWidget` 实现，当用户点击不同的 Tab，即切换到相应的页面。

Qt 中 `QWidget` 可以作为其他 `QWidget` 的 parent 或 child。通过继承 `QWidget` 类，Taskmgr 项目中定义的类可以加入其他窗口类中。指定合适的布局，就能得到与 Win10 任务管理器类似的风格。

具体分解方法和使用的布局以 CPU 页面为例，见图11 (p52)，图12 (p53)，图13 (p54)，图14 (p55)，图15 (p56)，图16 (p57)，图17 (p58)。

## QHBoxLayout

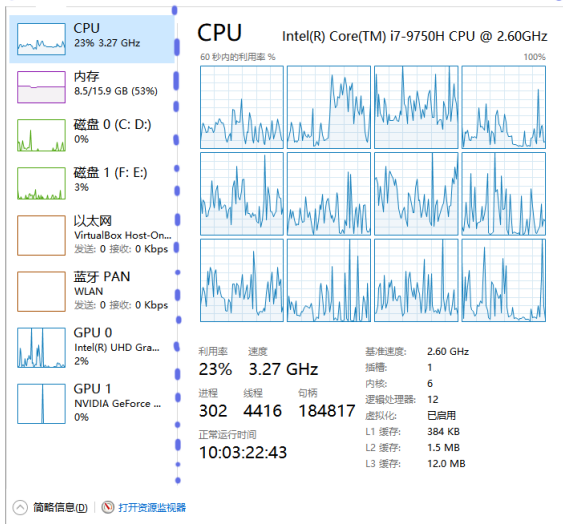


图: CPU 页面

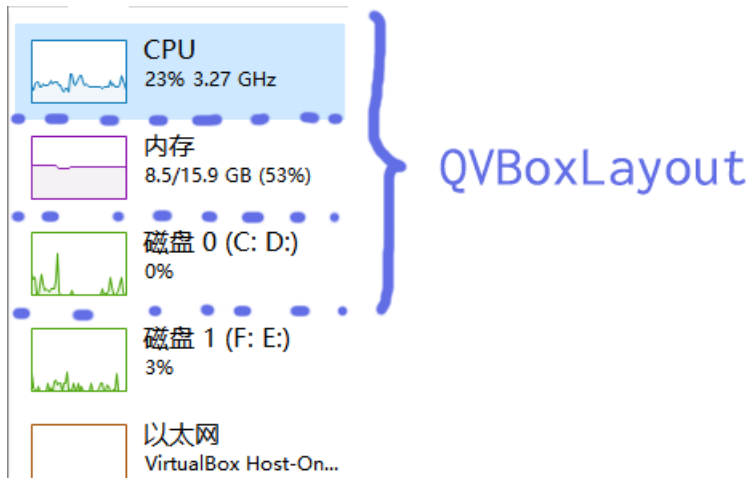


图: CPU 页面左侧

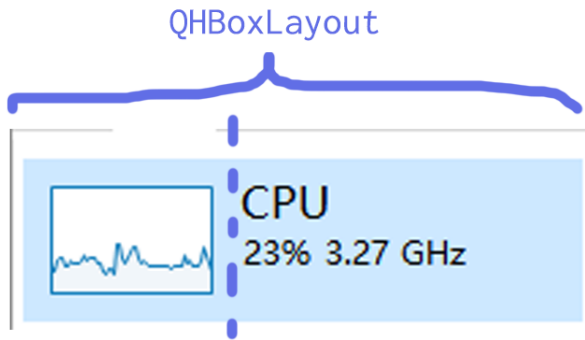


图: CPU 页面左侧缩略图



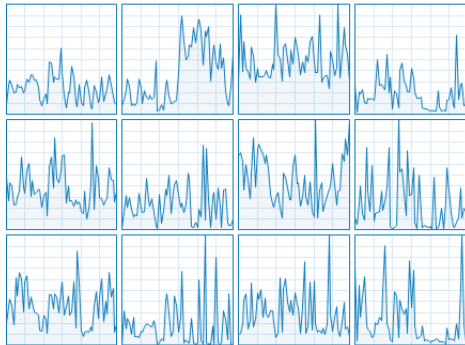
图: 缩略图右侧

# CPU

Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz

60 秒内的利用率 %

100%



利用率 速度

23% 3.27 GHz

进程 线程 句柄

302 4416 184817

正常运行时间

10:03:22:43

基准速度: 2.60 GHz

插槽: 1

内核: 6

逻辑处理器: 12

虚拟化: 已启用

L1 缓存: 384 KB

L2 缓存: 1.5 MB

L3 缓存: 12.0 MB

QVBoxLayout

图: CPU 页面右侧





QGridLayout

图: CPU 页面右侧中部

# QHBoxLayout

利用率	速度		基准速度:	2.60 GHz
23%	3.27 GHz		插槽:	1
			内核:	6
进程	线程	句柄	逻辑处理器:	12
302	4416	184817	虚拟化:	已启用
正常运行时间			L1 缓存:	384 KB
10:03:22:43			L2 缓存:	1.5 MB
			L3 缓存:	12.0 MB

QGridLayout

QGridLayout

图: CPU 页面右侧底部

## signal 与 slot

Qt 中 signal 与 slot 是不同对象间沟通的机制<sup>23</sup>，特定操作会发射 signal，而与这个 signal 关联的 slot 响应 signal。

Qt 中 signal 与 slot 的关联是完成特定功能的重要方式。Taskmgr GUI 界面的显示逻辑，与用户的交互，调用 win32\_common 类的时机都是通过 signal 与 slot 实现的。

大多数 Qt 组件都预先定义了不少 signal，只需要自定义 slot 并将 signal 与 slot 关联即可。不过 performance 类中也创建了新的 signalclicked，当鼠标点击 performance 类时发射此 signal，完成一些功能。

---

<sup>23</sup><https://doc.qt.io/qt-5/signalsandslots.html>

# Taskmgr 类的实现

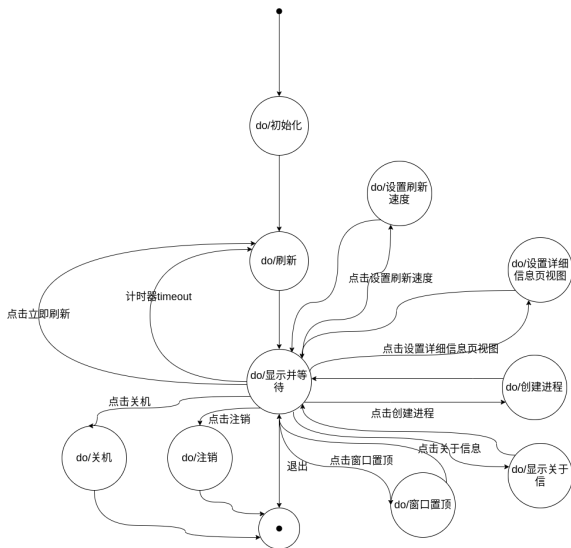


图: Taskmgr 类状态图

定时刷新使用 `QTimer` 类。将 signal `QTimer::timeout()` 与 slot `Taskmgr::do_refresh()` 关联。使用 `QAction::setChecked()` 确保更新速度默认为正常 (1000 ms)。

菜单栏通过创建一个从属于主窗口的 `QMenuBar` 类添加，各个菜单项，如“文件”等，由 `QMenuBar::addMenu()` 函数添加。具体的项目，如“退出”等，由 `QMenu::addAction()` 添加。

`QAction` 类提供了抽象的用户界面的动作，`QAction` 类的实例可以插入不同的 `QWidget` 组件，并根据所在的组件以不同的风格显示<sup>24</sup>。`QAction` 类有预定义 signal `QAction::triggered`，在 `QAction` 形成的组件被激活时发射。

---

<sup>24</sup><https://doc.qt.io/qt-5/qaction.html>

一组 action 可以是互斥的，即同一时间只能有一个 action 被激活。在 Taskmgr 菜单栏的更新速度菜单中的项目和视图菜单中的项目就属于这种情况。创建一个 QActionGroup 实例并把同组动作加入其中即可。

Taskmgr 类中 action 与 slot 的对应关系见下图



在 `do_newProc()` 中，首先使用 `QInputDialog::getText()` 函数弹出窗口提示用户输入路径，规则是：可以输入可执行文件路径，也可输入其他类型文件或文件夹路径；可以向可执行文件传递参数；路径中若出现空格，需要使用双引号包围。

收到用户输入后调用 `Taskmgr::parsePath()` 得到文件路径和命令行参数的字符串，再根据这两个信息调用 `win32_common::openProc()` 打开新进程。

`Taskmgr::parsePath()` 代码如下：



```
259 void Taskmgr::parsePath(std::string path, std::
    string &prog, std::string &param)
260 {
261     bool inquote = false;
262     bool in = false;
263     std::string::size_type beg, pos;
264     for (pos = 0; pos < path.size(); ++pos) {
265         if (!inquote && path[pos] != ' ') {
266             if (path[pos] == '\\'){
267                 beg = pos+1;
268                 inquote = true;
269             }
270             else if (!in){
271                 in = true;
272                 beg = pos;
273             }
274         }
275         else if (inquote) {
276             if (path[pos] == '\\'){
```

# 详细信息页的实现

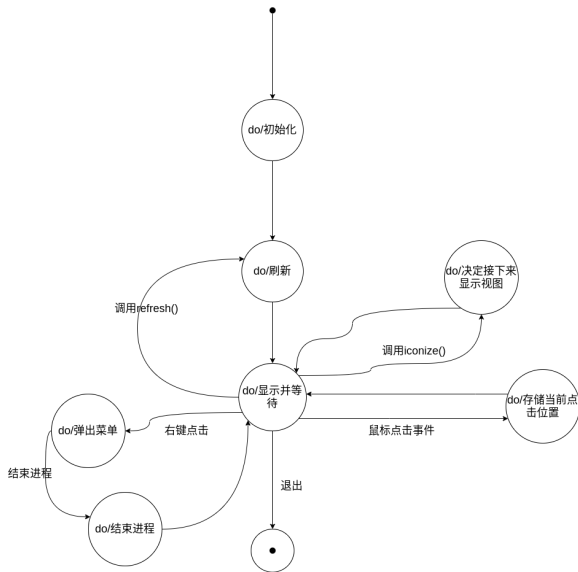


图: detail 类状态图

详细信息页有两种视图：详细列表与小图标。这两个视图分别由 `QTableWidget` 和 `QListWidget` 实现。  
`detail` 类中的函数见表5（p67）。

---

```
        detail(win32_common *buffer, QWidget *parent = nullptr)
void    iconize(bool x)
void    initListView()
void    initIconView()
void    refresh()
void    refreshListView()
void    refreshIconView()
void    do_table_pressed(int row)
void    do_table_sort(int index)
void    do_icon_pressed(QListWidgetItem item)
void    do_terminate()
void    contextMenuEvent(QContextMenuEvent event) override
```

---

表: detail 类中的函数

detail 类中含有 bool 型变量 iconized，用于指示本类处于详细列表视图还是小图标视图。在构造函数和刷新函数中，通过判断 iconized 的值调用不同的组件初始化和组件刷新函数。

iconized 的修改通过公共函数 detail::iconize() 函数实现。该函数接受一个 bool 类型参数，只有当原视图与此参数代表的视图类型不同时才切换至新的视图，同时更新 iconized 的值。在小图标与详细列表切换的过程中需要删除和添加组件，这通过在页面的主 layout 中移除和添加 Widget 实现。

detail::iconize() 在切换视图时从 layout 中移除 QTableWidgetItem 或 QListWidget 对象，并调用相应初始化函数，在各自的初始化函数中组件将自身加入布局。

右键弹出包含“结束任务”菜单由 `contextMenuEvent()` 实现(代码如下)。对于右键结束进程来说，当右键点击列表或图标时有两个 signal 是有关的：单元格点击事件与鼠标右击事件。前者已经关联了 `detail` 类中的 `do_table_pressed()` 或 `do_icon_pressed()`，将当前激活的组件的位置记录到 `detail` 类内部变量 `cur_row` 中；后者弹出右键菜单，其中包含结束进程的 action。

```
24 void detail::contextMenuEvent(  
    QContextMenuEvent *event)  
25 {  
26     QMenu menu(this);  
27     menu.addAction(terminateAc);  
28     menu.exec(event->globalPos());  
29 }
```

右键菜单结束进程动作被激活后相应的 slot 从 `cur_row` 中取出刚才点击单元的位置，得到相应的进程 PID 和进程名信息，从而正确完成相应动作。与右键菜单中结束进程动作关联的 slot `do_terminate()` 代码如下：

```
204 void detail::do_terminate()
205 {
206     QString procName;
207     if (!iconized)
208         procName = table->item(cur_row,0)->
                text();
209     else
210         procName = iconList->item(cur_row)->
                text();
211
212     QMessageBox msgBox;
213     msgBox.setWindowTitle("任务管理器");
214     msgBox.setText("<font size=\"5\" color
        =\"#003399\">你希望结束 " + procName +
        " 吗? </font>");
215     msgBox.setInformativeText("如果某个打开的
        程序与此进程关联, 则会关闭此程序并且"
216                               "将丢失所有未保
```

当然这里两个 slot 存在执行次序限制：do\_terminate() 必须在 do\_table\_pressed() 或 do\_icon\_pressed() 后执行，否则结束的是上一次点击的进程。但受到神经元电信号传导速率的限制，当人类用户移动鼠标并点击“结束任务”时后者已经完成了，所以实际上不需要采取措施限定它们的执行顺序。



# 详细信息视图

QTableWidget 用于表示表格，其中每个元素称为 cell。创建 QTableWidgetItem，设置相应的值并按一定的行列规则加入 QTableWidget 就能表示每个进程的不同属性（名称、PID 等）。在详细信息视图中，每一行表示一个进程，所以鼠标点选应该一次选中一整行。这种行为通过 QAbstractItemView::setSelectionBehavior() 设置；一次选中的行数只能是 1，这通过 QAbstractItemView::setSelectionMode() 实现。

QTableWidget 含有 header, header 中有不同的 section。当 section 被点击时发射 signal `QHeaderView::sectionClicked`, 将其与 `QTableWidget::sortItem()` 关联即可实现点击某列时按此列排序。

为了实现多次点击同一列 header 时对这一列交替地升序排序和降序排序, detail 类中保存 `bool` 类型数组用以保存此前对应列的排序方式, 数组长度与列长相同。当一个 section 被点击, 与其关联的自定义 slot 取出此前该列的排序状态, 对其取反并作为新值回存, 再根据这个值决定 `QTableWidget::sortItem()` 的升降序。代码如下:

```
194 void detail::do_table_sort(int index)
195 {
196     SectionClicked = true; //点击过section
197     recentSec = index;
198     if (order[index] = !order[index])
199         table->sortItems(index, Qt::SortOrder::
200             AscendingOrder);
201     else
202         table->sortItems(index, Qt::SortOrder::
203             DescendingOrder);
204 }
```

QTableWidget 使用 MVC 模型, Qt::ItemDataRole 影响 Model 和 View 解释数据的方法。QTableWidgetItem 默认保存 QString 类型数据, 排序方法是字符串排序。

为了对 PID, 线程数和内存列采用数值排序, 需要改变相应 QTableWidgetItem 的 role。使用 QTableWidgetItem::setData() 设置 role 为 Qt::DisplayRole 并传入包含 unsigned long long 类型的 QVariant, 这样就指定了特定 QTableWidgetItem 对象数据源的类型为数值型。

detail::refreshListView() 中相关的部分代码如下:

```
111         item = new QTableWidgetItem;
112         item->setData(Qt::DisplayRole, QVariant
            ((unsigned long long)beg->first));
113         table->setItem(i,1,item);
```

当行被点击时，与 `QTableWidget::cellPressed` 关联的 slot `detail::do_table_pressed` 保存当前行数至 `row`，供进程结束命令使用。代码如下：

```
184 void detail::do_table_pressed(int row)
185 {
186     this->cur_row = row;
187 }
```

## 小图标视图

QListWidget 默认按行显示，需要向 `setViewMode()` 传递 `QListView::IconMode` 参数，设置其显示方式为图标。小图标视图不需要实现排序，功能较简单。

当图标被点击时，和详细列表的行被点击时的行为类似，与 signal `QListWidget::itemPressed` 关联的 slot `detail::do_icon_pressed` 使用 `QListWidget::row()` 函数保存当前单击图标的位置至 `row`，供结束进程命令使用。代码如下：

```
189 void detail::do_icon_pressed(QListWidgetItem *  
    item)  
190 {  
191     cur_row = iconList->row(item);  
192 }
```

# 用户页的实现

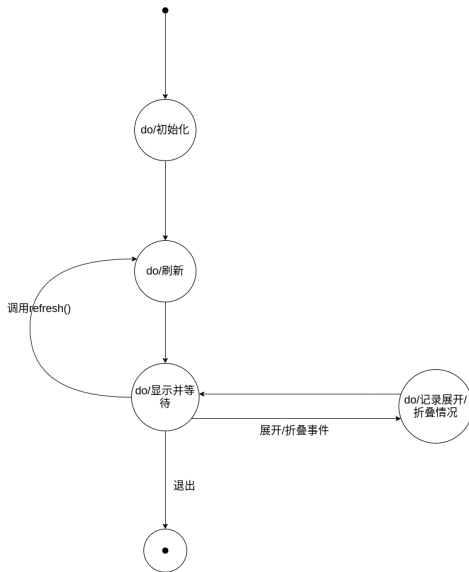


图: user 类状态图

用户页的功能是展示系统中所有用户和用户所创建的进程。这种结构通过 `QTreeWidget` 实现两层的树形布局，可以点击用户账户名称展开其创建的所有进程。

刷新时调用 `QTreeWidget::clear()` 清空所有内容，再重新填入新的用户和运行的进程。

当一个组件的下级组件被添加、删除时，这个组件会自动折叠。

为了尽量保持更新前后视图一致，在 `user` 类中使用 `std::vector<bool> expanded` 记录各个用户组件的展开情况，刷新时访问 `expanded` 决定是否展开某个用户的进程：如果刷新前后用户数量不变，则认为用户没有发生变化，展开上次展开过的用户；若刷新后用户数量改变，则更新 `expanded` 的值为全 `false`，折叠所有用户组件。

`expanded` 的更新通过 slot `do_expanded()` 和 `do_collapsed()` 实现，这两个函数使用

`QTreeWidget::indexOfTopLevelItem()` 获取被展开或折叠的用户组件序号，据此更改 `expanded` 中的元素。



do\_expanded() 与 do\_collapsed() 代码如下:

```
77 void user::do_expanded(QTreeWidgetItem *item)
78 {
79     expanded[tree->indexOfTopLevelItem(item)]
        = true;
80 }
81
82 void user::do_collapsed(QTreeWidgetItem *item)
83 {
84     expanded[tree->indexOfTopLevelItem(item)]
        = false;
85 }
```

刷新时 user::refresh() 中根据 expanded 确定是否展开:

```
70         // 展开 / 收起
71         if (expanded[i])
72             tree->expandItem(topItem);
73         ++i;
```

# 性能页的实现

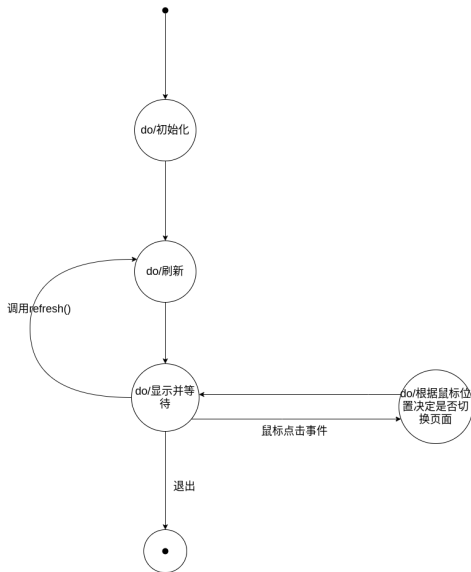


图: performance 类状态图

性能页由 CPU 页、内存页和两个缩略图组成。

CPU 页和内存页的布局采用 `QStackLayout`，同一时刻两者只能有一个显示。

为了达到点击缩略图显示对应详细页面的效果，增加了鼠标点击事件发生时发射的 `signal clicked`。将 `clicked` 与 `mouseClicked()` 关联，当 `clicked` 发射时 `mouseClicked()` 检查点击区域是否位于其中一个缩略图之内，若是，则使用 `QStackLayout::setCurrentIndex()` 设置此页面为当前页面：

```
45 void performance::mouseClicked()
46 {
47     if (cputhumbn->geometry().contains(
48         mousePos))
49         slayout->setCurrentIndex(0);
50     else if (memthumbn->geometry().contains(
51         mousePos))
52         slayout->setCurrentIndex(1);
53 }
```

性能页主要用于展示，交互功能只有以上一个。

# load 类的实现

CPU 页面，内存页面和缩略图都使用了 load 类以实现实时数据图表显示。

绘图功能使用 Qt 的 Charts 模块。这不属于 Qt Core，为了使用 Charts 模块，需要在 .pro 文件中加入：

```
QT += charts
```

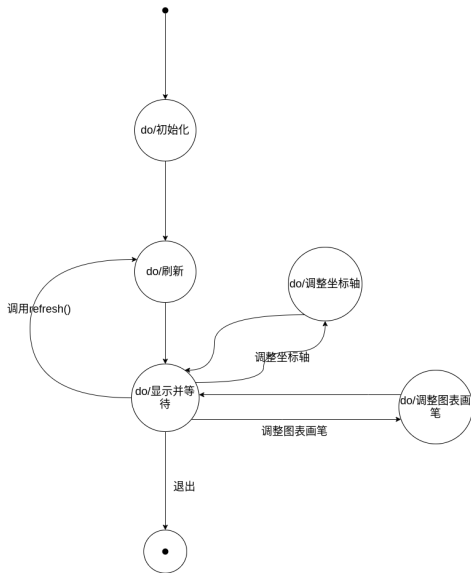


图: load 类状态图

load 类使用样条曲线作图。创建一个 `QSplineSeries` 类的实例，设置与其绑定的 X 轴与 Y 轴值域分别为  $[-60s, 0s]$  和  $[0, 100]$ 。X 轴的值域设置是为了显示 60s 内的数据变化情况，且最新的数据在最右侧，和 Win10 任务管理器一致。更新时首先判断最旧的点横坐标值，即其创建的时刻是否距现在已超过 60s。若是则删除这一点，否则直接添加新的一点。这样就达到了实时数据的刷新：

```
60 void load::refresh(double load)
61 {
62     auto now = QDateTime::currentDateTime();
63     axisX->setMin(now.addSecs(-60));
64     axisX->setMax(now.addSecs(0));
65     if (series->count() && (series->at(0).x()
        < now.addSecs(-60).toMsecsSinceEpoch())
        )
66         series->remove(0);
67     series->append(now.toMsecsSinceEpoch(),
        load);
68 }
```

load 类还提供了 `setSeriesPen()`, `setXtick()` 等函数供设置绘制风格。



## 简介

概览

使用的 API 与库

## 总体结构

需求分析

开发方法与软件结构

模块独立性分析

## win32 API

在程序中使用 win32 API 的方法

win32\_common 封装的功能

win32\_common 的实现

pid\_thrd 的获取

进程名的获取

module 路径的获取

进程所属用户的用户名的获取

结束进程

CPU 型号的获取

创建新进程

Taskmgr 的权限提升

## GUI

界面分解

signal 与 slot

Taskmgr 类的实现

详细信息页的实现

详细信息视图

小图标视图

用户页的实现

性能页的实现

load 类的实现

## Taskmgr 的部署

## 功能与性能分析

功能分析

与 Win10 任务管理器的对比

比

性能分析

# Taskmgr 的部署

把 ico 格式的图标放入项目文件夹，在 .pro 项目文件中添加：

```
RC_ICONS = Taskmgr.ico
```

Qt 程序在运行时需要访问 dll 图形库，所以除了最终生成的 exe 文件外还需要把相关库放入合适的位置。

以 Release 模式生成可执行文件后将 exe 文件复制到目标文件夹，运行 windeployqt Taskmgr.exe 将需要的 dll 复制到目标文件夹下即可。只要拥有此文件夹，就可正常运行 Taskmgr。

## 简介

概览

使用的 API 与库

## 总体结构

需求分析

开发方法与软件结构

模块独立性分析

## win32 API

在程序中使用 win32 API 的方法

win32\_common 封装的功能

win32\_common 的实现

pid\_thrd 的获取

进程名的获取

module 路径的获取

进程所属用户的用户名的获取

结束进程

CPU 型号的获取

创建新进程

Taskmgr 的权限提升

## GUI

界面分解

signal 与 slot

Taskmgr 类的实现

详细信息页的实现

详细信息视图

小图标视图

用户页的实现

性能页的实现

load 类的实现

## Taskmgr 的部署

## 功能与性能分析

功能分析

与 Win10 任务管理器的对比

性能分析

# 功能分析

没有实现的功能已经在第?? 节 (p??) 列出。

除了功能的不完全以外，已实现的功能中有一些问题：

- ▶ CPU 页面时钟频率始终是 2.54GHz，但 API 调用应该没有错。可能是这个 API 比较古老，对处理器的支持不好
- ▶ 无法获取一些进程的信息，可能是因为需要比管理员更高的权限或者需要其他权限
- ▶ 界面不够美观

# 与 Win10 任务管理器的对比

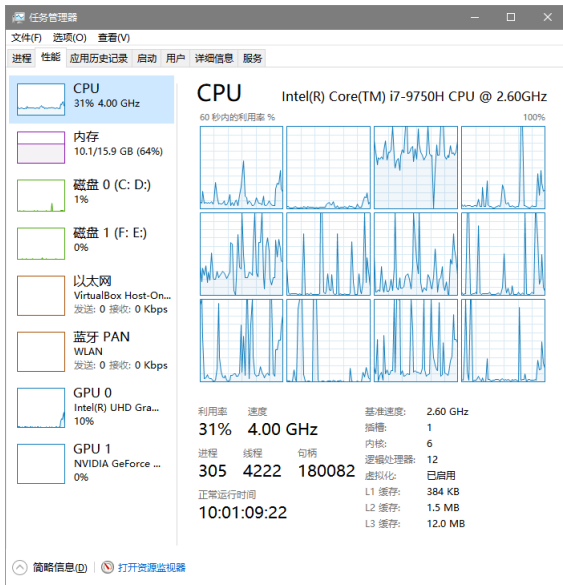


图: Win10 任务管理器性能页-CPU

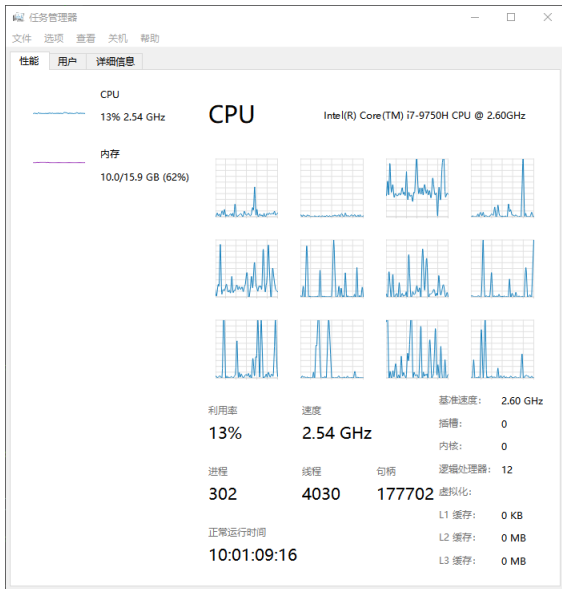


图: Taskmgr 性能页-CPU

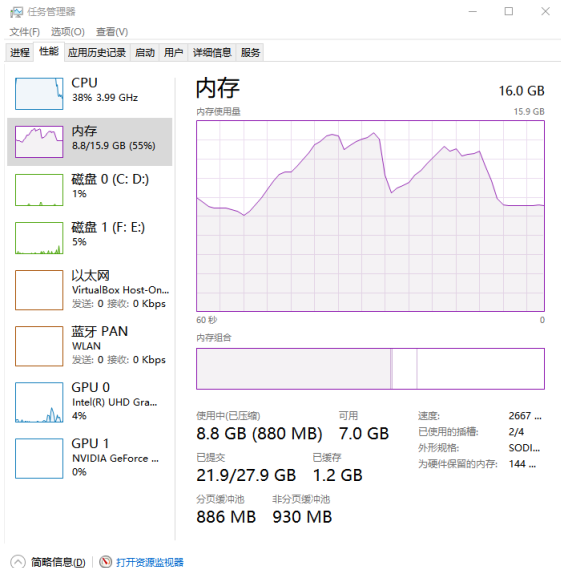


图: Win10 任务管理器性能页-内存



图: Taskmgr 性能页-内存



# 性能分析

主要从 Taskmgr 消耗的 CPU 资源、内存资源和实际体验评价性能。最后通过 qorbit 注入采样分析程序的性能瓶颈。

这里通过 Windows 10 自带的任务管理器粗略地查看 Taskmgr 消耗的资源。不使用 Taskmgr 自身的原因是因为它是被测对象，而且功能也没有自带的任务管理器齐全。

刷新闻隔为 1s 时，Taskmgr 在性能页、用户页、详细信息页列表视图时消耗的 CPU 资源与内存分别为 (1.8%, 25.6MB), (1.6%, 25.6MB, (2.0%, 25.7MB)。 (见图27 (p98), 图28 (p99), 图29 (p100))。这个资源占用是可以接受的。

## 任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	状态	21% CPU	60% 内存	1% 磁盘
应用 (7)				
> Firefox (16)		15.6%	792.6 MB	0.1 MB/秒
> Google Chrome (10)		0%	246.6 MB	0 MB/秒
> Microsoft OneNote		0%	18.6 MB	0 MB/秒
> Qt Creator (5)		0%	1,857.8 ...	0.1 MB/秒
> Taskmgr.exe		1.8%	25.6 MB	0 MB/秒
> Windows 资源管理器		0%	24.2 MB	0 MB/秒
> 任务管理器		0.6%	56.2 MB	0 MB/秒
后台进程 (152)				
> Activation Licensing Service (32 位)		0%	1.6 MB	0 MB/秒
> Adobe Acrobat Update Service (32 位)		0%	0.1 MB	0 MB/秒

## 任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	状态	19% CPU	60% 内存	0% 磁盘
应用 (7)				
> Firefox (16)		15.2%	798.7 MB	0 MB/秒
> Google Chrome (10)		0%	246.2 MB	0 MB/秒
> Microsoft OneNote		0%	19.9 MB	0 MB/秒
> Qt Creator (5)		0%	1,857.9 ...	0 MB/秒
> Taskmgr.exe		1.6%	25.6 MB	0 MB/秒
> Windows 资源管理器		0%	24.4 MB	0 MB/秒
> 任务管理器		0.2%	57.2 MB	0 MB/秒
后台进程 (153)				
> Activation Licensing Service (32 位)		0%	1.7 MB	0 MB/秒
> Adobe Acrobat Update Service (32 位)		0%	0.1 MB	0 MB/秒

## 任务管理器

文件(F) 选项(O) 查看(V)

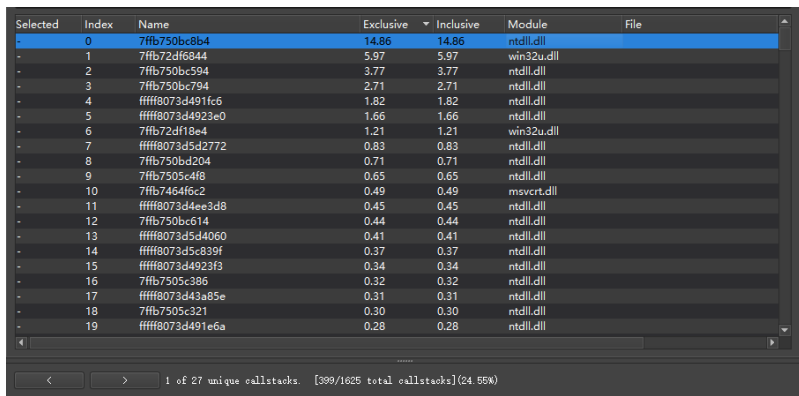
进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	状态	20% CPU	60% 内存	1% 磁盘
应用 (7)				
> Firefox (16)		14.9%	789.7 MB	0 MB/秒
> Google Chrome (10)		0%	246.7 MB	0 MB/秒
> Microsoft OneNote		0%	18.6 MB	0 MB/秒
> Qt Creator (5)		0%	1,857.8 ...	0 MB/秒
> Taskmgr.exe		2.0%	25.7 MB	0 MB/秒
> Windows 资源管理器		0%	24.3 MB	0 MB/秒
> 任务管理器		0.5%	57.0 MB	0 MB/秒
后台进程 (153)				
> Activation Licensing Service (32 位)		0%	1.5 MB	0 MB/秒
> Adobe Acrobat Update Service (32 位)		0%	0.1 MB	0 MB/秒

从实际体验来看，即使刷新闻频率为高，在各个 Tab 之间切换时没有卡顿，在详细信息页切换详细列表视图与图标视图时，以及鼠标滚动时也不会卡顿。  
总的来说性能可以接受。

接下来通过注入采样分析性能瓶颈。

在 1s 的刷新闻隔下，使用 qorbit 注入采样 14.9s。结果见图30 (p102)。



Selected	Index	Name	Exclusive	Inclusive	Module	File
-	0	7fb750bc8b4	14.86	14.86	ntdll.dll	
-	1	7fb72df6844	5.97	5.97	win32u.dll	
-	2	7fb750bc594	3.77	3.77	ntdll.dll	
-	3	7fb750bc794	2.71	2.71	ntdll.dll	
-	4	ffff8073d491fc6	1.82	1.82	ntdll.dll	
-	5	ffff8073d4923e0	1.66	1.66	ntdll.dll	
-	6	7fb72df18e4	1.21	1.21	win32u.dll	
-	7	ffff8073d5d2772	0.83	0.83	ntdll.dll	
-	8	7fb750bd204	0.71	0.71	ntdll.dll	
-	9	7fb7505c4f8	0.65	0.65	ntdll.dll	
-	10	7fb7464f6c2	0.49	0.49	msvcrt.dll	
-	11	ffff8073d4ee3d8	0.45	0.45	ntdll.dll	
-	12	7fb750bc614	0.44	0.44	ntdll.dll	
-	13	ffff8073d5d4060	0.41	0.41	ntdll.dll	
-	14	ffff8073d5c839f	0.37	0.37	ntdll.dll	
-	15	ffff8073d4923f3	0.34	0.34	ntdll.dll	
-	16	7fb7505c386	0.32	0.32	ntdll.dll	
-	17	ffff8073d43a85e	0.31	0.31	ntdll.dll	
-	18	7fb7505c321	0.30	0.30	ntdll.dll	
-	19	ffff8073d491e6a	0.28	0.28	ntdll.dll	

1 of 27 unique callstacks. [399/1625 total callstacks](24.55%)

图：注入采样结果（Exclusive time 降序排序）

占用 exclusive 时间最多的是 ntdll.dll，比排序第二的 win32u.dll 高了近两倍。在 ntdll.dll 的调用栈中（见图31（p104））可以看出在 ntdll.dll 调用的主要是读取虚拟内存、读取进程内存和读取进程的模块的函数。而在 win32u.dll 的调用栈中（见图32（p105）），主要是用于绘图的函数与 dll（GetGlyphOutlineW, Qt6Gui.dll 等）。

Hooked	Index	Function	Size	File	Line	Module	Address	Conv
-	0	NtReadVirtualMemory	0	0	0	ntdll.dll	0x7ffb750bc8a0	UnknownCallConv
-	1	ReadProcessMemory	0	0	0	KERNELBASE.dll	0x7ffb72e822b0	UnknownCallConv
-	2	EnumProcessModules	0	0	0	KERNELBASE.dll	0x7ffb72e83a10	UnknownCallConv
-	3	EnumProcessModules	0	0	0	KERNELBASE.dll	0x7ffb72e83a10	UnknownCallConv
-	4	7ffb1374f73b	0	0	0	Taskmgr.exe	0x7ffb1374f73b	UnknownCallConv
-	5	7ffb137453b4	0	0	0	Taskmgr.exe	0x7ffb137453b4	UnknownCallConv
-	6	7ffb13748e4c	0	0	0	Taskmgr.exe	0x7ffb13748e4c	UnknownCallConv
-	7	7ffb11349357	0	0	0	Qt6Core.dll	0x7ffb11349357	UnknownCallConv
-	8	7ffb110d8a9a	0	0	0	Qt6Core.dll	0x7ffb110d8a9a	UnknownCallConv
-	9	7ffb110bb3b7	0	0	0	Qt6Core.dll	0x7ffb110bb3b7	UnknownCallConv
-	10	7ffb109c5946	0	0	0	Qt6Widgets.dll	0x7ffb109c5946	UnknownCallConv
-	11	7ffb11086ed8	0	0	0	Qt6Core.dll	0x7ffb11086ed8	UnknownCallConv
-	12	7ffb111f9f2f	0	0	0	Qt6Core.dll	0x7ffb111f9f2f	UnknownCallConv
-	13	7ffb111fa8c0	0	0	0	Qt6Core.dll	0x7ffb111fa8c0	UnknownCallConv
-	14	7ffb73ce681d	0	0	0	USER32.dll	0x7ffb73ce681d	UnknownCallConv
-	15	7ffb73ce6212	0	0	0	USER32.dll	0x7ffb73ce6212	UnknownCallConv
-	16	7ffb111fe7ca	0	0	0	Qt6Core.dll	0x7ffb111fe7ca	UnknownCallConv
-	17	7ffb11999e55	0	0	0	Qt6Gui.dll	0x7ffb11999e55	UnknownCallConv
-	18	7ffb11090fd3	0	0	0	Qt6Core.dll	0x7ffb11090fd3	UnknownCallConv
-	19	7ffb1108f8b6	0	0	0	Qt6Core.dll	0x7ffb1108f8b6	UnknownCallConv

图: ntdll.dll 的调用栈



Hooked	Index	Function	Size	File	Line	Module	Address	Conv
-	0	NtGdiGetGlyphOutline	0		0	win32u.dll	0x7ffb72df6830	UnknownCallConv
-	1	7ffb7215c5dd	0		0	gdi32full.dll	0x7ffb7215c5dd	UnknownCallConv
-	2	GetGlyphOutlineW	0		0	GDI32.dll	0x7ffb73404550	UnknownCallConv
-	3	7ffb119c350d	0		0	Qt6Gui.dll	0x7ffb119c350d	UnknownCallConv
-	4	7ffb11c4a5f0	0		0	Qt6Gui.dll	0x7ffb11c4a5f0	UnknownCallConv
-	5	7ffb11898c73	0		0	Qt6Gui.dll	0x7ffb11898c73	UnknownCallConv
-	6	7ffb1192fae6	0		0	Qt6Gui.dll	0x7ffb1192fae6	UnknownCallConv
-	7	7ffb10a1d567	0		0	Qt6Widgets.dll	0x7ffb10a1d567	UnknownCallConv
-	8	7ffb10a25dec	0		0	Qt6Widgets.dll	0x7ffb10a25dec	UnknownCallConv
-	9	7ffb10a261c0	0		0	Qt6Widgets.dll	0x7ffb10a261c0	UnknownCallConv
-	10	7ffb10a298b8	0		0	Qt6Widgets.dll	0x7ffb10a298b8	UnknownCallConv
-	11	7ffb10aa45a5	0		0	Qt6Widgets.dll	0x7ffb10aa45a5	UnknownCallConv
-	12	7ffb446e25ee	0		0	qwindowsv***style.dll	0x7ffb446e25ee	UnknownCallConv
-	13	7ffb446db843	0		0	qwindowsv***style.dll	0x7ffb446db843	UnknownCallConv
-	14	7ffb10a39801	0		0	Qt6Widgets.dll	0x7ffb10a39801	UnknownCallConv
-	15	7ffb446e534f	0		0	qwindowsv***style.dll	0x7ffb446e534f	UnknownCallConv
-	16	7ffb10c47b96	0		0	Qt6Widgets.dll	0x7ffb10c47b96	UnknownCallConv
-	17	7ffb10c6c873	0		0	Qt6Widgets.dll	0x7ffb10c6c873	UnknownCallConv
-	18	7ffb10c75e6c	0		0	Qt6Widgets.dll	0x7ffb10c75e6c	UnknownCallConv
-	19	7ffb10a0b998	0		0	Qt6Widgets.dll	0x7ffb10a0b998	UnknownCallConv

图: win32u.dll 的调用栈

由此可见，系统调用所花的时间是 Taskmgr 的性能瓶颈，对 Taskmgr 的优化应该在于降低 win32 API 调用频率。这也说明了此项目中出现内容耦合的合理性。

谢谢。