# 1.工具背景

## 1.1工具目标

1. 使用python开发界面工具

## 1.2工具特点

1. 用户在界面上进行操作
2. 工具需要处理大量数据（CPU+内存）
3. 工具需要进行持续性任务（高CPU占用）

# 2.界面程序设计方案

## 2.1方案一：单进程+多线程

### 2.1.1设计思路

界面和任务处理线程分开，避免执行任务时界面出现卡死的情况，影响用户体验。

### 2.1.2详细设计

图1 单进程+多线程

1. 主线程启动界面，对界面进行初始化；
2. 主线程启动一个线程，作用是后台任务处理；
3. 界面（主线程）和后台任务线程之间使用Queue进行数据传输，界面线程写数据，后台线程读数据；
4. 后台线程的主线程是一个while循环，用于循环监听Queue；
5. 用户点击界面的按钮等操作产生一个请求任务，用户每次操作界面，程序并不是直接执行任务，而是将请求任务的代码以及必要附加数据写入Queue；
6. 后台线程读取到任务代码后，查询任务处理程序，将任务代码和附加数据发送给任务处理程序，任务处理完成之后，将任务处理结果emit到界面显示；

### 2.1.3方案有效性

1. 对于轻量级的应用，即后台不存在高CPU和高内存占用的任务，此方案可以正常工作；

### 2.1.4方案缺陷

1. 但是由于python并不是真正的多线程，所以，如果后台长期存在高CPU和高内存占用的任务，那么可能会出现界面卡死的情况；
2. 另一个问题是，由于后台任务处理采用串行处理方式，如果某一个任务处理占用时间长，此时用户请求的其他任务会长时间得不到处理；
3. 工具处理的任务并不全是一次性任务，而是存在长期处理任务，即后台处理线程会启动更多的常驻线程，比如trap监听任务，需要在后台启动线程进行长期监听，导致整个程序实际启动的线程数量多，影响工具的性能；

## 2.2方案二：多进程+多线程

### 2.2.1设计思路

界面和任务分离。

借用多进程解决方案一中线程的问题，

### 2.2.2详细设计

图1 多进程+多线程

1. 主进程启动界面，对界面进行初始化；
2. 主进程启动一个子进程，作用是后台任务处理；
3. 进程之间使用Pipe进行数据传输，线程之间使用Queue进行数据传输，主进程和子进程各有一个Pipe和一个Queue；
4. 主进程有两个线程，一个是界面主线程，一个是while循环监听线程；
5. 主进程：
6. 界面包含主界面和多个子界面；
7. 各个界面的一个任务由任务代码和附加数据定义，用户在界面上进行操作后，程序将任务请求代码和附加数据写入Queue；
8. 监听线程读取任务代码，如果任务需要本进程中的界面执行，则将任务请求代码和附加数据发送给目标界面处理，界面处理完任务之后，处理结果在两个界面中显示（按需）；
9. 监听线程读取任务代码，如果任务需要子进程中执行，则将任务请求代码和附加数据写入Pipe，发送给子进程；
10. 监听线程从Pipe中读取任务处理结果，在界面中更新，展示给用户；
11. 子进程：
12. 子进程启动后，初始化所有任务处理程序；
13. 子进程的主线程执行while循环监听任务；
14. 监听线程读取Pipe的任务处理请求后，查询任务处理程序，将任务请求代码和附加数据发送给任务处理程序；
15. 任务处理类处理完后，将处理结果写入Pipe，发送至主进程；
16. 子进程中，各个任务处理程序或者功能模块如果需要其他任务处理程序执行任务，同样将任务请求代码和附加数据写入Queue，主线程将读取数据，执行任务；

### 2.2.3方案有效性

1. 多进程真正分离了界面和任务处理，界面不会出现卡死的情况，且子进程由于没有界面的干扰，可以快速的进行任务处理；

### 2.2.4方案缺陷

1. 并不是所有的任务都适用于放置在后台子进程，比如snmp采集任务，一是界面需要读取MIB数据库的内容并显示在界面，二是采集的大量结果需要频繁的显示在界面上。如果采用放置在子进程的方案，则进程之间通信的数据的量会非常大，且存在延时。这是我的担忧，实际上是否有问题并没有验证；
2. 如果主进程界面卡死崩溃了，子进程会残留，不会随着主进程的退出而退出；

## 2.3方案三：多进程+多线程+看门狗

### 2.3.1设计思路

在方案二的基础上，加入看门狗，如果主进程异常退出了，子进程可以正常退出，解决子进程可能残留的问题。

### 2.3.2详细设计

图1 多进程+看门狗

1. 看门狗是一个独立的线程，分为服务端和客户端；
2. 主进程和子进程启动时，首先启动看门狗线程，主进程启动服务端看门狗，子进程启动客户端看门狗；
3. 看门狗之间使用Pipe进行通信；
4. 服务端看门狗持续监听Pipe，如果收到投喂请求，则向返回一个投喂响应；
5. 客户端看门狗间隔一段时间向服务端看门狗发送一次投喂请求；
6. 客户端发送投喂请求后，等待一段时间，再查询Pipe是否收到投喂响应，如果连续多次未收到投喂响应，则通知子进程退出；
7. 子进程在收到退出事件后，执行退出程序；

### 2.3.3方案有效性

第一版的看门狗方案是借助已有的进程通信机制，由子进程的主线程定期发送投喂请求，主进程的主线程收到请求后返回投喂响应，但是此方案受其他任务处理影响很大，如果主线程执行任务耗时太长，投喂请求可能得不到及时的响应，导致子进程异常退出。因此才将看门狗在独立的线程中执行，实际效果证明该方案有效。

# 3.进程间通信方案

## 3.1方案一：类UDP通信

### 3.1.1设计思路

由于工具将界面和任务处理进行了分离，因此，工具的整体框架可以看成是由客户端和服务器组成，客户端发送请求，服务器响应请求并返回结果。

客户端即界面，用户在界面上操作，并最终在界面上得到操作的结果。界面由主界面和子界面组成，每个界面是一类相同任务类型的集合，如Snmp报文测试界面可以执行向设备发送snmp报文的任务，连接类界面则实现了SSH、Netconf等登陆设备的方式。

服务器则由任务分发模块和任务处理模块组成，任务分发模块将请求的任务分发至对应的任务处理模块进行处理。任务处理模块的功能是执行具体的任务请求，每个模块均是一类任务类型的集合，简单来看任务处理模块和界面之间可以存在一一对应的关系，如Snmp报文测试界面存在对应的Snmp报文任务模块，连接类界面存在对应的连接类任务模块。不同的是，一类界面可能同时打开多个子界面，如连接类子界面0、连接类子界面1，而任务处理模块则只存在一个实例，处理所有子界面的任务请求。需要注意的是，实际界面和任务模块之间并不是绑定的关系，即界面的请求根据其任务类型，可以发送给多个不同的任务模块。所以服务器的任务模块需要包含所有界面中的任务请求处理程序，如果服务器收到未知的任务请求，会返回任务未定义错误信息。



图1 界面和服务器

综上所诉，进程间通信需要解决的问题是，将某一界面的任务请求数据（任务请求代码和附加数据）发送至对应的任务处理模块，并且将任务处理结果返回至该界面。为了解决该问题，采用UDP通信的思想，给每个界面和任务处理模块分配一个IP，数据封装在报文中，报文包含源IP地址和目的IP地址，转发模块根据IP转发报文，达到通信的目的。



图2 数据传输问题

### 3.1.2详细设计



图1 通信设计图

#### 网络结构

1. 硬件结构：
   1. 硬件结构由Pipe和Queue组成，所有的报文均是生产者写入Pipe或者Queue，由消费者从Pipe或者Queue中读取报文，以达到报文在进程间和线程间转发的目的；
2. 转发模块：
   1. 转发模块主要负责定义数据包格式和地址格式；
   2. 解析接收到的报文的地址，选择路由进行转发；
3. 应用层：
   1. 发送数据的接口；

#### 报文结构

1. 报文由首部和数据两部分组成，首部的主要目的是标记路由信息，数据的主要目的是携带任务执行所需要的必要信息。



图2 报文结构

1. 如下图，首部包含源IP、目的IP、请求任务类型代码和报文编号。



图3 报文首部

1. 源IP：发送报文模块的IP地址；
2. 目的IP：报文的接收模块的IP地址；
3. 请求任务类型代码：任务类型大类，指定了由哪个任务处理模块处理本次任务请求，如指定连接类任务模块响应任务请求；
4. 报文编号：每条报文的唯一标识，可根据报文编号对应发送的报文和任务处理后的响应报文，报文编号由应用层自动生成；
5. 数据部分由下图所示五部分组成：



图4 数据结构

1. 请求任务具体类型：指定了任务处理模块中的具体任务，如连接类模块中的建立SSH连接任务；
2. 附加数据：执行任务所需的数据，如建立SSH连接任务需要设备IP、用户名和密码等数据，附加数据的数据格式由各个任务自行定义，无统一规定；
3. 任务是否成功标识：此部分由任务处理模块在响应报文中添加，标记了此次任务请求是否执行成功；
4. 任务错误信息：此部分由任务处理模块在响应报文中添加，记录任务执行过程中出现的错误信息；
5. 任务结果附加信息：此部分由任务处理模块在响应报文中添加，记录任务执行的具体结果数据，如请求向设备发送命令任务得到的设备回显信息；
6. Python实现的报文是字典类型，如下所示：

package = {  
 'source\_id': source\_buffer\_id,   
 'destination\_id': destination\_buffer\_id,   
 'task\_id': task\_id,  
 'msg\_id': msg\_id,  
 'data': data  
}

data = {  
 'task\_type': task\_type0,  
 'append\_data': [],

'task\_success': True/False,

task\_type0: None,

'error\_msg': error\_msg  
}

与报文的对应关系为：





图5 Python实现与报文的对应关系

#### 程序设计

1. IP格式：
   1. IP由两位数字组成，数字之间用\_分割，如1\_32；
   2. IP在局域网中是唯一的，跨局域网可能存在相同的IP；
2. IP分配：
   1. 首先要确定哪些对象需要分配IP，本工具对于具体的功能类分配唯一的IP地址，功能类是一类功能实体的集合，在程序中表现为类，如各个子界面、任务处理模块；
   2. 每一类的功能模块有全局唯一的代码，作为IP的首位，此代码也是task\_id的首位；



图6 功能模块代码

* 1. 一类界面的多个子界面IP的首位是相同的，比如两个连接类子界面的IP分别为5\_0、5\_1，即可以通过IP的首位确定是哪一类的功能模块；
  2. 将每个进程看做是一个局域网，进程启动时会分配到一个局域网IP。数据如需发送至其他进程，则报文中的目的IP需要包含局域网的IP，否则默认报文在当前局域网（进程）中转发；
  3. 子进程的IP由主进程在创建子进程时生成；
  4. 各个功能模块的IP由主线程在创建功能模块时生成；

1. 路由学习：
   1. 工具的路由是静态的，主进程在创建子进程时，会将所有进程的IP信息都发送给子进程；
   2. 进程内，各个功能模块的代码存储在主线程中，供转发线程查询；
2. 转发规则：
   1. 报文由报文转发线程进行转发，每个进程中只有一个转发线程，即界面程序设计方案2中的监听线程；
   2. 转发线程在接收到报文之后，首先分析目的地址：
      1. 目的地址只含有一个IP地址时，如果此IP地址与当前进程的IP地址相同，则分析task\_id，根据task\_id确定任务处理模块；
      2. 目的地址只含有一个IP地址时，如果此IP地址与当前进程的IP地址不同，则寻找路由进行转发；
      3. 目的地址包含多个IP地址时，提取首位IP，如果首位IP与当前进程IP相同，则根据第二个IP进行转发，且转发前，删除首位IP，如8\_0\_9\_0修改为9\_0；
      4. 目的地址包含多个IP地址时，如果首位IP与当前进程IP不同，则根据首位IP进行转发；
      5. 如果找不到转发路由，则返回功能未定义错误；
   3. 如果报文需要转发到其他进程，且源IP的首位IP不是当前进程的IP，则将在'source\_id'的头部加入当前进程的IP，标记报文经过了当前进程，如9\_0转发后为8\_0\_9\_0；
   4. 一个报文支持多目的地址转发，所有目的地址以list的形式存放在destination\_id对应的值中；
   5. 功能处理模块接收到报文之后，如果报文中的task\_type未定义处理方法，则返回功能未定义错误，如果task\_type已定义，则返回处理后的结果；
   6. 功能处理模块处理完请求的任务后，将接收到报文中的source\_id和destination\_id交换再交由转发线程转发；
   7. 当程序包含多个子进程时，子进程间的通信经由主进程实现，即子进程0的报文先发送至主进程，再由主进程转发至子进程1，目的是不需要建设过多的物理线路，简化程序设计；
   8. 传输大数据时，数据先存放在文件中，报文中只发送文件的目录地址；
3. 进程间转发：



图7 进程间转发

* 1. 某个功能模块将待发送的数据通过报文生成模块生成原始报文，如Trap界面（集成在了监听界面）发送请求启动Trap监听的报文，

其中：

9\_0：监听界面的IP；

6\_0：表示报文的目的地是任务中心进程：

1\_2023...：请求由Trap功能模块执行任务；

0：执行的具体任务代码为0，即'start\_listening'；

{'source\_id': '9\_0', 'destination\_id': '6\_0', 'task\_id': '1\_20230321105253598910', 'msg\_id': '20230321105253598564', 'data': {'task\_type': 0, 'append\_data': [{'port': '160', 'ipv4': True, 'ipv6': True}]}}

* 1. 主进程的转发线程在接收到报文之后，由于需要转发的其他进程，在'source\_id'的头部加入当前进程的IP，如上述报文处理后被转发：

{'source\_id': '8\_0\_9\_0', 'destination\_id': '6\_0', 'task\_id': '1\_20230321105253598910', 'msg\_id': '20230321105253598564', 'data': {'task\_type': 0, 'append\_data': [{'port': '160', 'ipv4': True, 'ipv6': True}]}}

* 1. 子进程接收到报文后，检查destination\_id，如果目的地址就是当前进程，无下一跳地址，则分析task\_id，根据task\_id首位确定报文请求的服务应该发送至哪个功能模块进行处理，如上述报文将转发至Trap模块。如果功能模块未定义，则子进程返回未定义错误；
  2. 功能模块接收到报文之后，提取task\_type，并在定义的服务列表中寻找该task\_type，如果task\_type已定义处理方法，则调用方法执行任务，处理完成后返回处理结果，如果task\_type未定义，则返回未定义错误。上述报文在处理完成后返回的报文如下：

{'source\_id': '6\_0', 'destination\_id': '8\_0\_9\_0', 'task\_id': '1\_20230321105253598910', 'msg\_id': '20230321105253598564', 'data': {'task\_type': 0, 'append\_data': [{'port': '160', 'ipv4': True, 'ipv6': True}], 'task\_success': True, 'error\_msg': ''}}

* 1. 子进程转发线程将功能模块处理完成后的报文根据目的地址进行转发，如上述报文经过子进程转发后为：

{'source\_id': '6\_0', 'destination\_id': '8\_0\_9\_0', 'task\_id': '1\_20230321105253598910', 'msg\_id': '20230321105253598564', 'data': {'task\_type': 0, 'append\_data': [{'port': '160', 'ipv4': True, 'ipv6': True}], 'task\_success': True, 'error\_msg': ''}}

* 1. 主进程在接收到报文后，根据转发规则转发报文，如上述报文经过转发后为：

{'source\_id': '8\_0\_6\_0', 'destination\_id': '9\_0', 'task\_id': '1\_20230321105253598910', 'msg\_id': '20230321105253598564', 'data': {'task\_type': 0, 'append\_data': [{'port': '160', 'ipv4': True, 'ipv6': True}], 'task\_success': True, 'error\_msg': ''}}

目的地址为9\_0，报文将会转发到IP为9\_0的监听界面。

* 1. 最终，处理后的报文返回到发起服务请求的界面，完成了从发起服务请求，到处理服务请求，最后返回服务请求结果的完整流程。

1. 进程内转发：



图8 进程内转发

进程内转发与进程间转发的差异在于报文的目的地址首位IP不指向其他进程的IP，转发线程在接收到报文后，寻找路由进行转发。其他过程与进程间转发一致。

1. 主动上报：



图9 事件上报

功能模块在响应任务返回结果之后，后台可能依然会存在任务执行线程，该线程会记录任务对应的源IP等信息。如果线程在执行任务过程中出现异常事件，线程会将该事件根据记录的源IP上报给请求服务的模块。如trap模块在后台持续执行监听任务过程中，如果遇到异常退出监听的事件，监听任务线程会将事件上报给监听界面，告知用户后台的任务运行情况。

### 3.1.3方案有效性

此方案能够实现进程内和跨进程的通信功能，且不存在丢包的情况。

### 3.1.4程序示例

https://github.com/AlvinsFish/UiExample

# 4.python+QTextbrowser内容显示

本章主要解决的问题：

1. 利用QTextbrowser显示文本数据。
2. 探究数据转换为str类型对QTextbrowser显示性能的影响。
3. 如何显示各种类型的数据。
4. 管理多QTextbrowser显示。
5. 解决QTextbrowser持续写入数据导致占用内存多导致程序崩溃的问题。

## 4.1单QTextbrowser显示

QTextbrowser继承自QTextEdit，提供了一个带有超文本导航的富文本浏览器，只能浏览不能编辑。

QTextbrowser通过append的方式将数据显示在控件的尾部，由于程序中可能存在不同的功能类需要显示内容至同一个QTextbrowser，为了方便管理内容的显示，定义一个显示管理类。显示管理类主要的目的是将数据写入和数据显示两个步骤进行解耦，任一需要显示信息的功能类（如界面）只需要将待显示内容写入指定的缓存即可，不需要关心显示的细节。显示管理类定期读取缓存中的内容，append到QTextbrowser中显示，整体显示流程如下图所示。

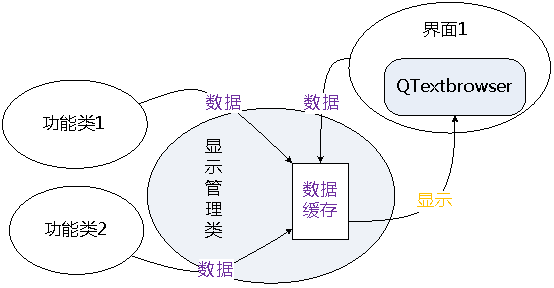


图1 显示管理类

显示管理类继承自QThread，可以作为线程独立运行，也可以作为普通的类由外部驱动显示。在本程序中，为了减少线程数量，显示管理类由循环监听线程驱动，监听线程会定期检查Queue中是否有数据需要显示。需要注意的是，该方案虽然可以减少一个线程的数量，但是会受到其他任务执行的影响，如果任务执行时间长可能带来显示延迟的问题。

创建显示管理类时，会初始化一个数据缓存，数据缓存是一个Queue，各功能类将数据写入此Queue。由于只能将字符串类型的数据append到QTextbrowser中，所以在显示管理类读取到数据之后，需要对数据进行判断转换，再显示，数据类型转换的方式有如下方式：

1. 通过str()的方式直接转换为字符串类型。但是如果数据量大，数据直接转换之后显示到QTextbrowser中可能会出现性能慢、界面卡死的问题，原因未知。
2. 针对不同的数据类型进行转换。为了规避直接转换带来的性能问题，需要对不同的数据类型进行更精细的转换，目前显示管理类支持对list、dict和tuple类型的数据进行转换，其他类型的数据暂时采用str()的方式，代码如下所示。

def convert\_type\_to\_str(content=None):  
 *"""将其他类型的数据转为str类型"""* msg\_to\_text = ""  
 if isinstance(content, str):  
 msg\_to\_text += content + '\n'  
 elif isinstance(content, list):  
 msg\_to\_text += '['  
 for data in content:  
 msg\_to\_text += convert\_type\_to\_str(data).strip() + ', '  
 msg\_to\_text = msg\_to\_text.strip().strip(',')  
 msg\_to\_text += ']\n'  
 elif isinstance(content, dict):  
 msg\_to\_text += '{'  
 for key, value in content.items():  
 msg\_to\_text += convert\_type\_to\_str(key).strip() + ': '  
 msg\_to\_text += convert\_type\_to\_str(value).strip() + ', '  
 msg\_to\_text = msg\_to\_text.strip().strip(',')  
 msg\_to\_text += '}\n'  
 elif isinstance(content, tuple):  
 msg\_to\_text += '('  
 for arg in content:  
 msg\_to\_text += convert\_type\_to\_str(arg).strip() + ', '  
 msg\_to\_text = msg\_to\_text.strip().strip(',')  
 msg\_to\_text += ')\n'  
 else:  
 msg\_to\_text += str(content) + '\n'  
 return msg\_to\_text

程序采样第二种转换方法，使用方法有两个优点，一是避免了直接转换导致性能慢的问题，二是用户在显示数据时，不需要考虑数据的类型，可以将各种类型的数据直接写入数据缓存。

持续显示数据，QTextbrowser中数据量过大时，可能带来程序性能慢的问题，为了避免此问题，需要定期删除部分显示内容，释放占用的内存。删除内容有两种方案，一是直接删除所有的内容，清空显示，二是先截取部分最新的显示内容，再清空显示，再显示截取的内容，如此则有显示的连续性。

显示管理类提供了设置参数，支持对显示进行配置，配置选项如下图所示：

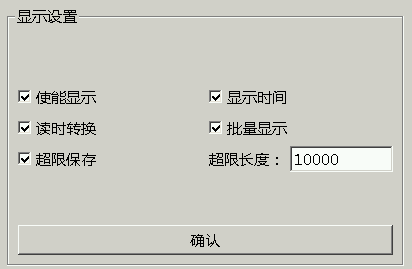


图2 显示设置

1. 使能显示：可以开关显示。
2. 显示时间：如勾选，则每次显示内容会附带显示当前时间。
3. 读时转换：如勾选，则在显示管理类读取数据时转换数据的类型，否则在写入数据时转换数据的类型。
4. 批量显示：如勾选，则显示管理类会读取Queue中所有的数据，再一次性显示，否则读取一次数据显示一次。
5. 超限保存：如勾选，则显示的数据如果超过超限长度，则数据只会显示一部分，并且将完整的数据保存到txt中，否则，数据依然只会显示一部分，但是完整数据不会被保存。
6. 超限长度：设置数据显示长度的阈值。

超限保存的功能是为了避免单次显示数据长度过长导致的性能问题。

总结针对QTextbrowser显示的性能优化：

1. 针对不同数据类型进行转换。
2. 定期清理显示内容。
3. 单次显示数据超限截断。

## 4.2多QTextbrowser显示

对于多个子界面均包含各自的QTextbrowser的程序，如果每个子界面各自管理QTextbrowser，存在冗余代码的问题，且不利于外部功能类显示数据。因此需要扩展显示管理类，使其能够统一管理程序内所有的QTextbrowser。

多QTextbrowser和单QTextbrowser显示的差异主要在于显示管理类新增了显示注册表，存储界面及其QTextbrowser的对应关系：{gui\_id: QTextbrowser}。显示的流程为：

1. 子界面初始化时，如果该界面含有QTextbrowser，则在显示管理类中注册。
2. 各功能类将需要显示的数据和期望显示界面的gui\_id写入数据缓存。
3. 显示管理类读取数据缓存中的数据，查显示注册表找到目标界面的QTextbrowser后进行显示。

显示管理类通过显示注册表管理QTextbrowser的方式的优点在于，为功能类和QTextbrowser搭建了桥梁，功能类可以将数据显示到任一QTextbrowser中，如界面1不仅可以将数据显示到本界面的QTextbrowser，也可以将数据显示到界面2的QTextbrowser中。

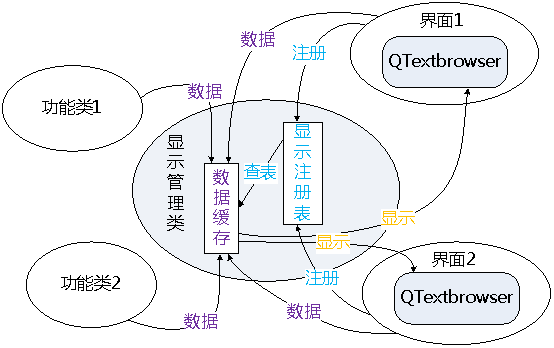


图3 多QTextbrowser显示

为了支持QTextbrowser的显示设置功能，子界面在注册时，不仅注册了QTextbrowser，同时也注册了显示设置参数，以便对每个QTextbrowser进行单独的显示设置。

## 4.3QTextbrowser直显

长期频繁的在QTextbrowser中显示数据，会导致程序占用的内存持续增长，最终导致程序崩溃。且如果界面存在频繁显示数据的需求，通过显示管理类进行显示存在显示效率低的问题。

为解决以上两个问题，开发的QTextbrowser直显功能，其特点为：

1. 数据直接显示到QTextbrowser。
2. 提升QTextbrowser显示清理的频率。
3. 降低单次显示数据的长度。

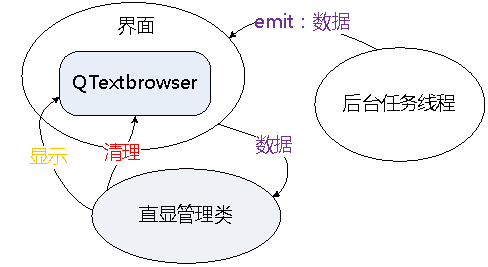


图4 QTextbrowser直显

如上图所示，后台任务线程将任务过程中产生的数据emit到界面，界面接收到数据之后，调用直显管理类的显示方法。直显管理类在接收到数据之后，并非将所有数据显示到QTextbrowser，而是将数据根据设定的超限长度进行截断后数据显示到QTextbrowser中，而原始数据则按需保存到txt。

直显管理类与显示管理类的差异在于，直显管理类没有数据缓存，在接收到数据之后直接显示到QTextbrowser，因此界面如果有直显需求，均需要创建直显管理类。

直显管理类中会记录显示数据的次数，如果达到设定的次数，则触发清空QTextbrowser的操作，及时释放占用的内存。

通过直显管理类，程序可以及时释放占用的内存，并且提升显示的效率。

## 4.4总结

本章的主要创新点为：

1. 统一管理界面内所有的QTextbrowser。
2. 数据写入和显示解耦。
3. 显示任意格式的数据。
4. 可设置每个QTextbrowser的显示方式。
5. 解决QTextBrowser导致界面崩溃的问题：
   1. 直显，不通过统一管理类。
   2. 长度超限数据截断显示，完整数据存为txt。
   3. 频繁清理已显示内容。

## 4.5程序示例

程序源码地址：https://github.com/AlvinsFish/UiExample

程序部分截图：



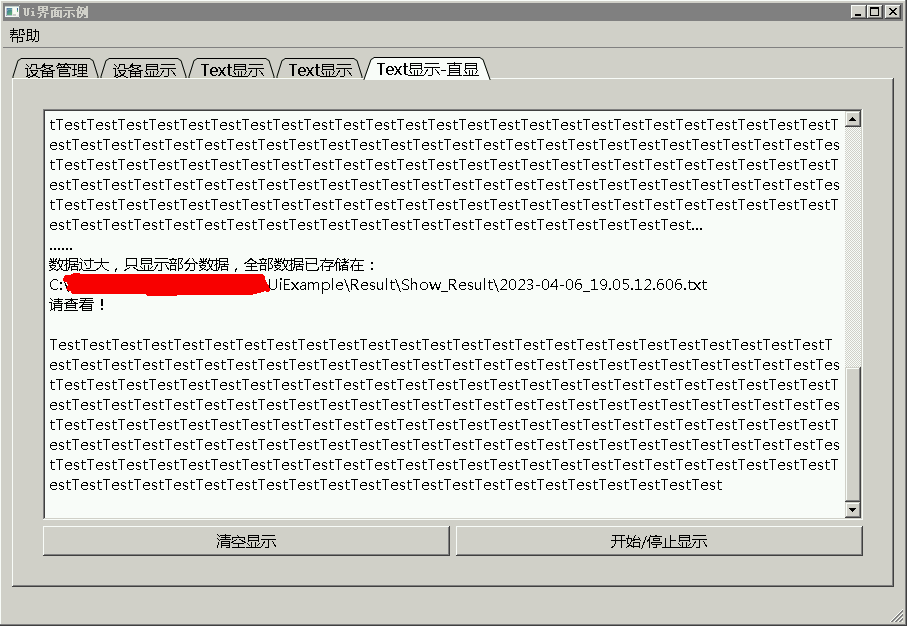


图5 程序截图