软件自动更新

Updater是一个简单、轻量级的升级工具，以Http Server作为文件服务器，可自动升级托管的用户软件。

Updater使用python开发，基于PyQt5实现，支持windows7、windows xp系统。

安装

Release

在`example/tools`目录下`vbuilder.exe`和`update/update.exe`为已经打包的可执行程序。

自动安装

> pyinstaller打包的exe文件在启动时会有几秒延时, 生产环境建议**手动打包**。

运行install.py文件即可, (需要安装python环境),
install.py脚本文件使用pyinstaller安装`vbuilder`和`update_gui`工具。

```
python .\install.py
```

> 需要注意python3.8为支持windows7的最后版本, python3.4.4为支持windows xp的最后版本。

使用

vbuilder构建软件发布包

> vbuilder暂时不支持GUI界面，只支持命令行方式构建!

借助`vbuilder`工具可以轻松构建软件发布包，使用命令如下:

windows

```
vbuilder.exe  LeftPath RightPath PatchPath
```

linux & macos

```
vbuilder  LeftPath RightPath PatchPath
```

参数	解释	是否必须	备注
LeftPath	旧版程序目录	是	
RightPath	新版程序目录	是	
PatchPath	补丁目录	是	
-i	忽略文件或目录	否	以:区分

参数	解释	是否必须	备注
-d	版本描述	否	
-v	更新版本号	否	
-f	开启增量更新	否	与-nf是一对
-nf	关闭增量更新	否	

`vbuilder`生成更新包之后，将更新补丁目录打包成zip文件，然后拷贝到文件服务器相应目录。

启用更新程序Updater

Updater exit code:

code	说明	备注
0	无新版本更新，或更新成功	
1	找不到用户程序	
2	Http错误，网络错误	
3	发布包错误，网络错误	
4	其它错误	

以C#为例，提供更新代码：

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
using System.Threading.Tasks;

namespace App
{
    class AutoUpdater
    {
        public static int Update()
        {

            IniFile iniFile = new IniFile(@".\update.ini");

            if (!File.Exists(@".\update.ini"))
            {
                Console.WriteLine("更新程序配置文件不存在!");
                return -1;
            }

            // 读取exitCode
```

```
            var exitCode = Convert.ToInt32(iniFile.IniReadValue("update",
"exitCode"));
            if (exitCode == 0)
            {
                // 启动更新
                Process updater = new Process();
                updater.StartInfo.FileName = @".\update_gui.exe";
                updater.StartInfo.CreateNoWindow = true;
                updater.StartInfo.UseShellExecute = false;
                updater.Start();
            }
            else
            {
                iniFile.IniWriteValue("update", "exitCode", "0");
            }

            return 0;
        }
    }

    class IniFile
    {
        public string Path;

        public IniFile(string path)
        {
            this.Path = path;
        }

        #region 声明读写INI文件的API函数
        [DllImport("kernel32")]
        private static extern long WritePrivateProfileString(string section,
string key, string val, string filePath);

        [DllImport("kernel32")]
        private static extern int GetPrivateProfileString(string section, string
key, string defVal, StringBuilder retVal, int size, string filePath);

        [DllImport("kernel32")]
        private static extern int GetPrivateProfileString(string section, string
key, string defVal, Byte[] retVal, int size, string filePath);
        #endregion

        /// <summary>
        /// 写INI文件
        /// </summary>
        /// <param name="section">段落</param>
        /// <param name="key">键</param>
        /// <param name="iValue">值</param>
        public void IniWriteValue(string section, string key, string iValue)
        {
            WritePrivateProfileString(section, key, iValue, this.Path);
        }

        /// <summary>
```

```
        /// 读取INI文件
        /// </summary>
        /// <param name="section">段落</param>
        /// <param name="key">键</param>
        /// <returns>返回的键值</returns>
        public string IniReadValue(string section, string key)
        {
            StringBuilder temp = new StringBuilder(255);

            int i = GetPrivateProfileString(section, key, "", temp, 255,
this.Path);
            return temp.ToString();
        }

        /// <summary>
        /// 读取INI文件
        /// </summary>
        /// <param name="Section">段，格式[]</param>
        /// <param name="Key">键</param>
        /// <returns>返回byte类型的section组或键值组</returns>
        public byte[] IniReadValues(string section, string key)
        {
            byte[] temp = new byte[255];

            int i = GetPrivateProfileString(section, key, "", temp, 255,
this.Path);
            return temp;
        }
    }
}
```

在主程序中使用：

```
namespace App
{
    internal static class Program
    {
        /// <summary>
        ///  The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            // 启动更新
            AutoUpdater.Update();

            // To customize application configuration such as set high DPI
settings or default font,
            // see https://aka.ms/applicationconfiguration.
            ApplicationConfiguration.Initialize();
            Application.Run(new Form1());
        }
    }
}
```

示例

文件服务器搭建&发布

`example`文件夹中包含了示例程序，其中`App1.0`、`App2.0`、`App3.0`为三个不同版本的（需要发布的）windows应用程序。

`patch`文件夹为补丁目录，存放各个版本的补丁文件、版本描述文件等。本示例程序中`patch`目录也为**文件服务器发布目录**，所以在`patch`目录中创建`App2.0`、`App3.0`目录，然后参考文件服务器部分，将`patch`目录搭建为文件服务器发布目录。

假设现在需要发布App2.0版本，需要通过`example\App1.0`与`example\App2.0`比对生成版本补丁与版本描述文件(version)，使用`vbuilder.exe`生成，命令如下：

```
vbuilder.exe .\App1.0 .\App2.0 .\Patch\App2.0 -v 2.0 -f
```

其中`-f`为增量更新,然后将`example\App2.0`中文件压缩为App2.0.zip，那么App2.0发布成功。

如果想要发布App3.0版本，启用全量更新，那么使用如下命令：

```
vbuilder.exe .\Blank .\App3.0 .\Patch\App3.0 -v 3.0 -nf
```

需要注意，全量更新比对生成时，`LeftPath`为空目录，该示例中`.\Blank`为空目录，`-nf`启用全量更新。

> 关于增量更新与全量更新的区别请查看这里。

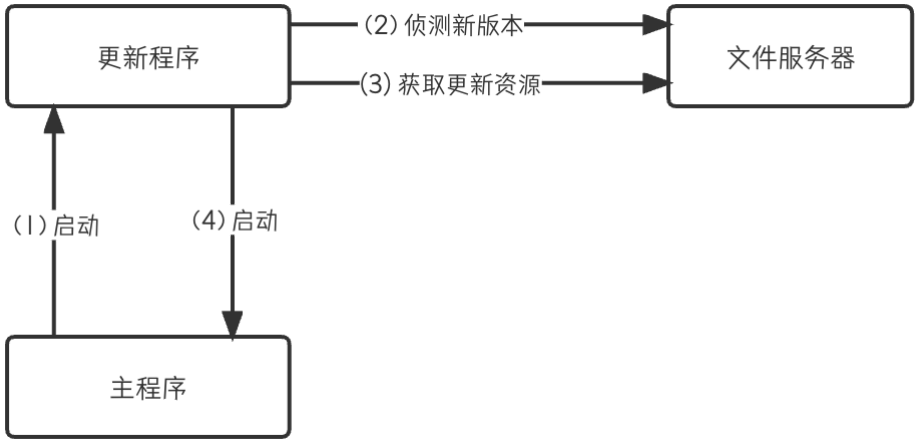
程序更新

将`update_gui.exe`与`update.ini`文件与1.0最初版本拷贝到客户端电脑。

其中`update.ini`文件需要根据要求修改。

程序框架

Updater程序整体逻辑如下图所示，**主程序**（用户程序）启动时调用**更新程序**，更新程序访问**文件服务器**，侦测是否发布新的版本。如果发布了新版本，再次访问文件服务器获取更新（补丁）等资源，然后**启动更新**，更新完成后启动主程序，然后退出更新程序；如果没有发布新版本，更新程序启动主程序，退出自身程序。

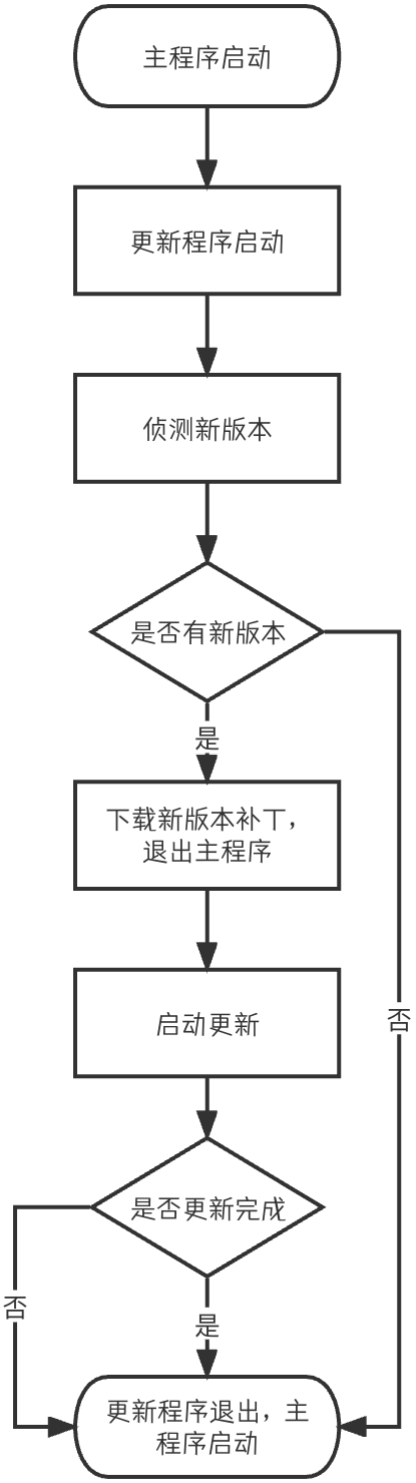


主程序逻辑

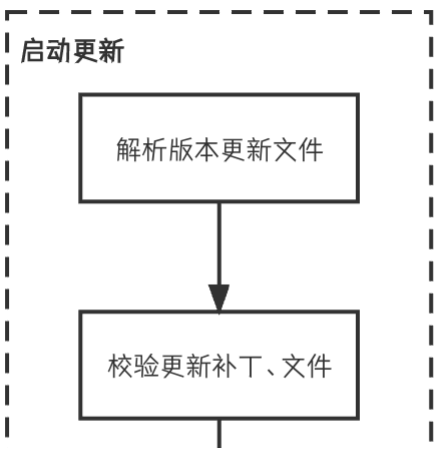
主程序（用户程序）只需要在启动时调用更新程序即可，其余操作可都交于更新程序进行处理。更新程序以埋点方式嵌入主程序，方便简单。

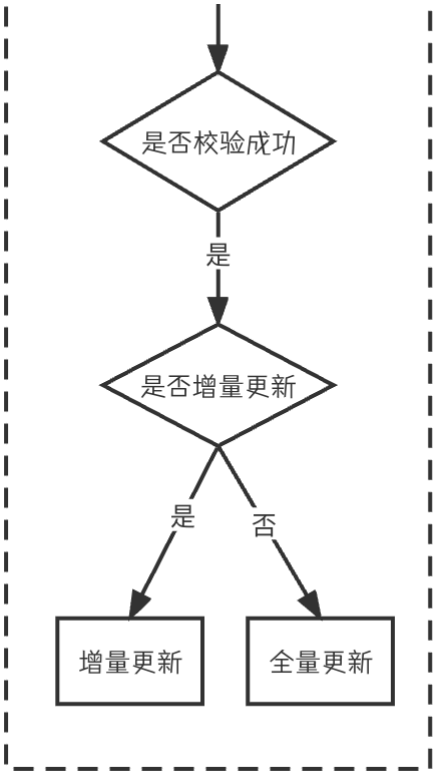
更新程序逻辑

主程序（用户程序）启动时，会调用更新程序Updater，更新程序Updater通过http协议访问文件服务器提取版本号，通过版本号比对判断是否有新版本，如果存在新版本，则下载新版本更新补丁等资源文件，并退出主程序，开始启动更新。
启动更新时会先解析**版本文件**，通过版本文件校验更新补丁资源等一系列操作，并判断是否需要启动增量更新，完成更新操作。


主程序启动
更新程序启动
侦测新版本
是否有新版本
是
否
下载新版本补丁，
退出主程序
启动更新
是否更新完成
否
是
更新程序退出，主
程序启动

启动更新流程：


启动更新
解析版本更新文件
校验更新补丁、文件



更新方式分为两大类，一类是全量更新；一类是增量更新；
全量更新是下载完整的安装包，然后重新安装一遍，无论原来是否存在内容。
增量更新是以打补丁的方式进行更新，在原来的基础上以增加补丁。

	是否省流	渣子问题	易于维护	顺序安装
全量更新	❌	❌	✅	✅
增量更新	✅	✅	❌	❌

全量更新

全量更新的逻辑是将主程序（用户程序）及相关文件删除，（可以通过配置忽略一些配置、日志文件），然后从文件服务器将新版本程序完整下载并解压。

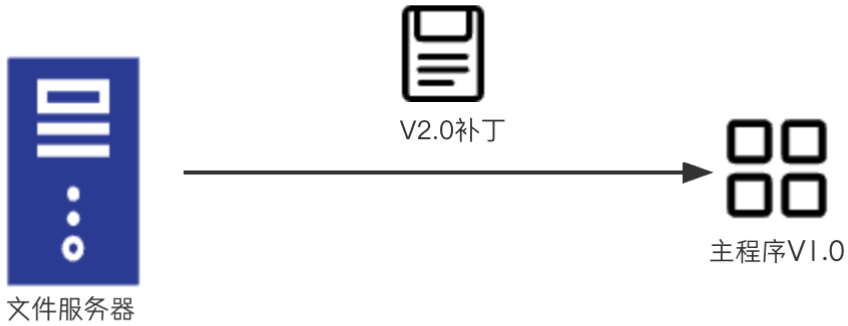
全量更新的优点是发布重构大版本时方便、快捷；并且方便未来扩展版本回溯功能。缺点是每次发布的包太大，浪费网络资源，造成流量浪费。

> 一般全量更新会采用**覆盖安装**的方式，但是本程序并未采取该方式，因为覆盖安装可能会导致渣子问题，比如v1.0版本含有test.dll文件，v2.0不含有test.dll文件，那么采用覆盖安装就会导致test.dll文件一直保留，如果test.dll存在漏洞，有可能会被黑客利用，同时发布多个版本后，会造成存储容量的浪费。所以本程序会先将原来的程序删除，然后重新安装的方式。

增量更新

增量更新相对于全量更新来说，每次发布的不是完整的程序包，而是基于上一版本生成的补丁文件进行更新，只将程序变动修改的地方以补丁的形式发布，更新程序将补丁合并到原程序。增量更新是基于bsdiff&bspatch技术实现，原算法与实现由Colin Percival提供，算法细节可以参考他的论文Naïve Differences of Executable Code，关于该算法的更多信息，可以参考 http://www.daemonology.net/bsdiff/。

增量更新的优点是每次只需要发布补丁，文件较小，能够大幅节省空间资源。缺点是版本回溯时比较麻烦，同时只能顺序安装，也就是说只能根据v1.0 -> v1.1 -> v1.2的顺序进行安装，当用户大规模未更新或者大版本更新时，省流优势就没有了。



增量更新还有可能会导致软件维护复杂度变高，因为我们不可能在发布v1.0之后，后续的所有版本都采用补丁方式发布。因为当发布了N个版本后，补丁的大小一定会远远超过程序本身的大小，而增量更新只能使用顺序更新的方式，那么就远远不及重新下载一个安装包，使用全量更新来的划算。所以我们一般小版本采用增量更新发布，大版本采用全量更新，比如v1.1发布小版本，v2.0发布大版本。

文件服务器

文件服务器以http协议提供文件资源，服务器本身不提供接口，因此易于搭建，可以使用Nginx或者IIS等工具快速搭建一个文件服务器。

本人测试时直接使用python命令快速搭建一个简易文件服务器，运行如下命令即可：

```
python -m http.server [port]
```

其中`[port]`为文件服务器监听的端口号。

> 文件服务器只提供简单的下载接口，其余复杂接口均不提供，这样不但易于搭建测试，也方便使用，但是建议在生产环境中做好负载均衡以及宕机热备等措施，减少因文件服务器宕机而造成无法更新的损失！

发布更新资源

借助工具`vbuilder`自动生成补丁文件并打包成zip文件，只需要将zip文件放置到文件服务器对应目录，即发布更新包成功。

Nginx搭建文件服务器

本文介绍使用Nginx搭建文件服务器的方法，在`example/tools`中提供了Nginx程序，通过修改`conf/nginx.conf`文件配置文件服务器：

```
    server {
        listen 1024;
        server_name localhost;
        root ..\..\Patch;

        location / {
            autoindex on;  # 显示目录
            autoindex_exact_size on;  # 显示文件大小
            autoindex_localtime on;  # 显示文件时间
            charset utf-8;
        }
    }
```

然后使用命令`nginx.exe`启动nginx程序，想要关闭nginx服务，使用命令`nginx.exe -s stop`。

关于nginx更多信息请参考官网。

其它

版本号

版本号由一个或多个**修订号**组成，各修订号由一个 "." 连接。
每个修订号由多位数字组成，可能包含前导零 。
每个版本号至少包含一个字符。修订号从左到右编号，下标从 0 开始，最左边的修订号下标为 0 ，下一个修订号下标为 1 ，以此类推。
例如，2.5.33 和 0.1 都是有效的版本号。比较版本号时，请按从左到右的顺序依次比较它们的修订号。
比较修订号时，只需比较忽略任何前导零后的整数值。也就是说，修订号 1 和修订号 001 相等。
如果版本号没有指定某个下标处的修订号，则该修订号视为 0。例如，版本 1.0 小于版本 1.1 ，因为它们下标为 0 的修订号相同，而下标为 1 的修订号分别为 0 和 1 ，0 < 1 。

> 因此为了发挥全量更新与增量更新的优势，建议小版本修改右侧修订号发布，采用增量更新；大版本修改左侧修订号，采用全量更新方式。

版本描述文件version

版本描述文件由`vbuilder`工具生成。

```
{
  "version": "1.0",
  "description": "文件描述",
  "packageTime": "2022-09-20",
  "incUpdateFlag": true,
  "files": [
```

```
    {
      "patch": true,
      "file": "./app.exe",
      "patchFile": "./app.exe.patch",
      "md5": "ddd34fsdf2jiojfsdjfsdfj"
    },
    {
      "patch": false,
      "file": "./update.txt",
      "md5": "ddd3fsssf2jiojfsdjfs1fj"
    }
  ]
}
```

该文件主要用于描述版本更新信息，其中`incUpdaterFlag`为是否启用增量更新标志。

程序配置文件update.ini

程序配置文件采用ini文件格式，因为本更新程序客户端主要在windows平台使用，所以采用ini文件格式，但是其它系统平台也能兼容该文件格式。

```
[program]
name = app
application = app.exe
version = 1.0
path = .
patch_path = ./tmp
ignore_files = App.runtimeconfig.json
log_path = ./log/update.log

[server]
url = http://localhost:1024/

[update]
exitcode = 0
```

其中name为项目名称；application为需要启动更新的程序名称(含后缀)；version为当前程序版本号；path为项目目录，默认为当前目录；patch_path为补丁目录；ignore_files为全量更新删除忽略文件、目录，防止因为渣子问题导致文件误删；log_path为日志路径。

url为文件服务器地址。

exitcode为更新程序退出代码。