# 图像预处理：

“预处理”是针对整个肿瘤膀胱壁识别框架而言的，在输入Unet神经网络之前，对训练图或测试图所进行的通用图像处理（某些算法来自OpenCV图像处理库）称为预处理。

## 预处理框架

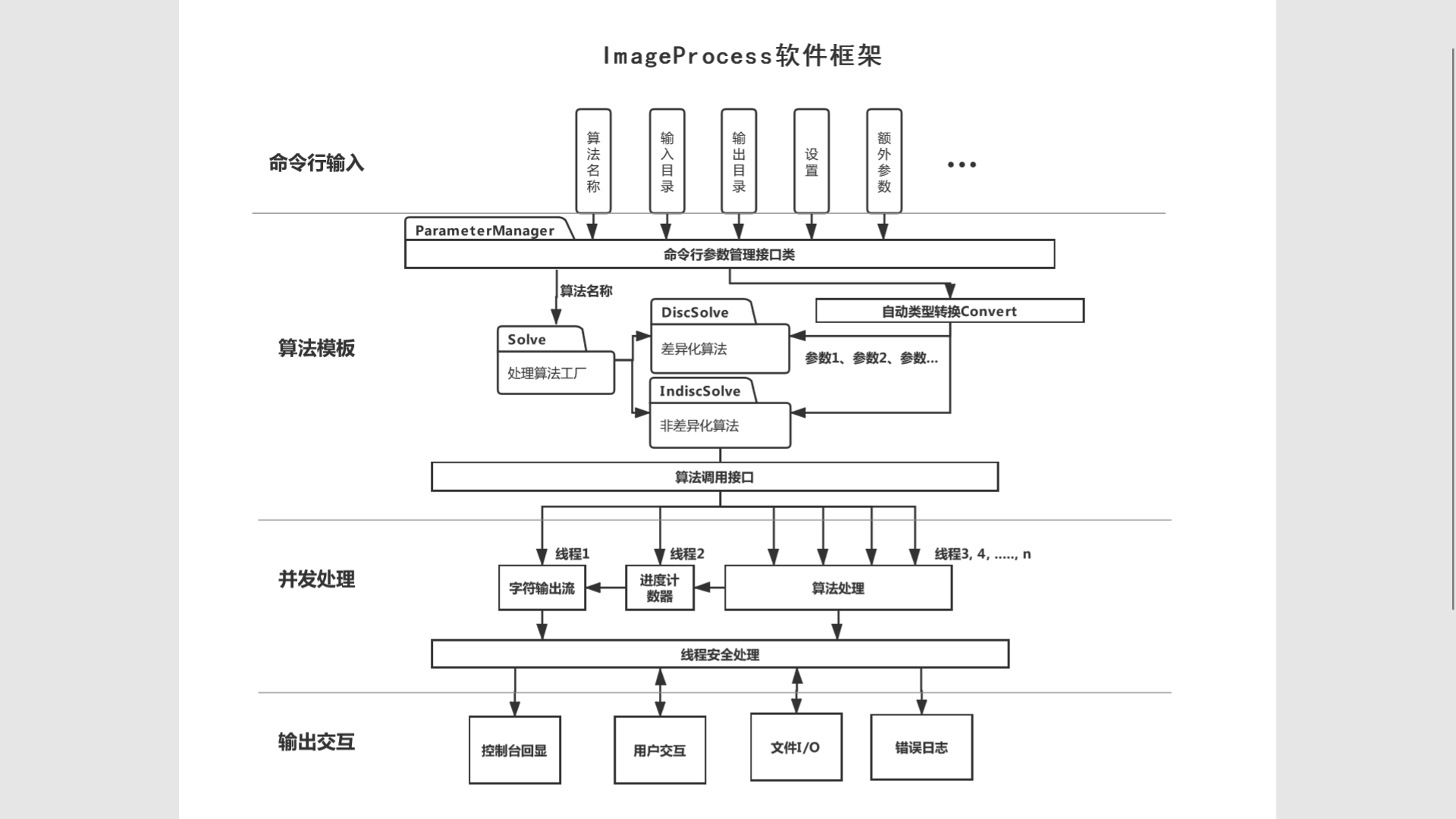


图 1.1‑1 ImageProcess软件框架

图像预处理部分软件命名为ImageProcess，负责连接神经网络处理模块和通用图像处理模块。ImageProcess具有如下特性：

* 向上为神经网络模块提供**命令行调用接口**，向下为通用图形算法提供**命令行参数捕获**及**自动类型转换**接口。
* 对通用图像处理维护者，提供**线程安全的并发处理平台**，具有反射机制特性的**算法工厂**提供动态创建对象统一接口
* 对用户，提供控制台提示界面，list命令搜集所有支持的通用算法接口，提供示例调用方式。

## 命令行调用接口

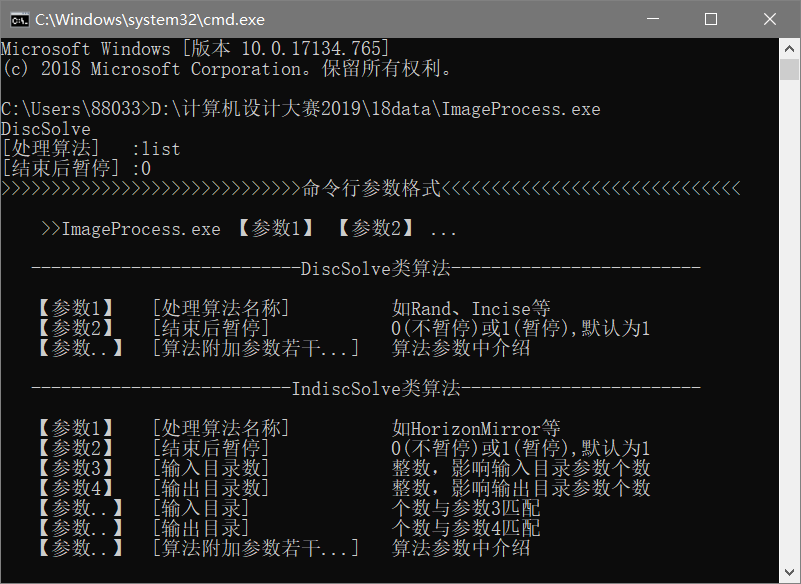


图 1.2‑1 命令行调用

ImageProcess将通用图像处理分为两个种类：差异化算法DiscSolve，非差异化算法IndiscSolve。分别制定两种公共参数列表，如上图所示。差异化/非差异化算法概念定义如下：

* **非差异化算法**：算法模板处理目标为已读入并解码的图像像素矩阵。对于任何输入算法模板的像素矩阵，都执行相同的矩阵处理操作。

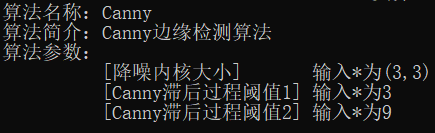


图 1.2‑2 非差异化算法参数示例

* **差异化算法**：算法模板没有固定处理目标。算法输入参数为一个字符串容器，其中一般包含若干个算法所需图像所在目录，及若干个算法所需图像输出目录。

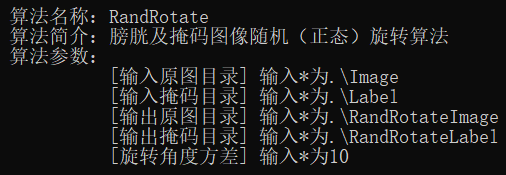


图 1.2‑3 差异化算法参数示例

命令行调用程序API为Windows提供，与语言无关。如C++中调用方式为：

表 1.2‑1 C++命令行调用方法

|  |
| --- |
| system("ImageProcess.exe list"); |

## 参数捕获及转换接口



图 1.3‑1 命令行参数管理类图

C++ main()函数入口保留了命令行调用参数输入获取方式，即主函数签名方法：

表 1.3‑1 主函数签名

|  |
| --- |
| int main(int argc, void\* argv[]) |

参数管理类ParameterManager使用以上两个形参argc, argv构造实例。提供变量绑定方法Push()/PushBatch()，和变量填充（参数捕获）方法Catch()。

其完整签名如下：

表 1.3‑2 变量绑定方法完整签名

|  |
| --- |
| void Push(const Str& para\_name, const Str& para\_default, const ActFun& action); |
| template<typename Ty>  void Push(const Str& para\_name, const Str& para\_default, Ty& para\_ref); |
| template<typename Ty>  void PushBatch(const Str& para\_name, const Str& para\_default, std::vector<Ty>& para\_vec); |
| template<typename Ty>  void PushBatch(const StrVec& para\_name\_vec, const Str& para\_default, std::vector<Ty>& para\_vec); |
| template<typename Ty>  void PushBatch(const Str& para\_name, const StrVec& para\_default\_vec, std::vector<Ty>& para\_vec); |
| template<typename Ty>  void PushBatch(const StrVec& para\_name\_vec, const StrVec& para\_default\_vec, std::vector<Ty>& para\_vec); |

表 1.3‑3 变量填充方法完整签名

|  |
| --- |
| int Catch(bool report = true); |

在ParameterManager实例作用域，将局部或全局变量绑定至ParameterManager实例上，当调用Catch方法时，ParameterManager自动将命令行参数填充至绑定变量。

绑定方法Push()/PushBatch()具有泛型特性，将根据绑定变量实际类型选择正确的类型转换方法。另外ParameterManager还提供索引器泛型重载，可通过指定返回类型方式，获得索引下的命令行参数对应类型转换值。签名如下：

表 1.3‑4 ParameterManager索引器泛型重载

|  |
| --- |
| template<typename Ty>  Ty operator[](unsigned int idx) |

其中，类型转换方法为Convert()重载函数，当前重载支持转换类型有：int, float, bool, string，及这四种类型为元类型的std::vector泛型容器。

## 线程安全并发处理平台

并发处理平台提供了一个统一的并发调用接口：

表 1.4‑1 带计数器并发调用函数接口

|  |
| --- |
| inline void ParallelWithCounter(int first, int last, const std::function<void(volatile long&, int)>& func, const Str& info, long maxCount = -1); |

接口使用了C++11中Lambda特性，依赖于C++11 ppl并发库，接口与库中concurrency::parallel\_for函数签名相似，但增加了计数器功能。接口自动捕获Lambda匿名回调函数func中对计数器的修改，并提交至输出流。调用该函数将并发出(last-first)个线程执行func回调函数。

线程安全输出流

图 1.4‑1 线程安全输出流类图

并发处理的线程安全特性，要求并发回调函数func中：

* 所有输出操作（即对函数外部变量的写操作）均使用提供的线程安全输出流实现。（文件I/O可除外）
* 对提供的volatile long类型计数器count，使用宏定义IncrementCounter(count)进行自增。

CountinueStream是OutputStream的内部使用类，对外无接口。

OutputStream使用单例设计模式，仅有一个全局实例，该实例由宏定义COUT引用。该实例的任何对外方法都是线程安全的。实例的输出流可重定向，提供三种重定向接口，签名及描述由下表给出：

表 1.4‑2 OutputStream重定向接口

|  |  |  |
| --- | --- | --- |
| **完整签名** | **重定向流目标** | **描述** |
| void RedefineBuf(std::ostream\* op); | op | 将OutputStream全局实例重定向到输出流op. |
| void OutputImmediately(); | std::cout | 将OutputStream全局实例重定向到输出流cout.  立即输出模式，使用COUT的语句立即打印到控制台。 |
| void OutputDelay(); | OutputStream::ss | 将OutputStream全局实例重定向到字符流ss.  延迟输出模式，输入储存在字符流ss中，下次重定向时迁移至目标输出流。 |

OutputStream的线程安全处理规则为**单语句时线程独占使用权，多语句时线程抢占使用权**：OutputStream实现左移位符重载，当线程单语句使用左移位符连续向OutputStream输入对象时，该线程独占OutputStream使用权（互斥元），以保证该语句不被其他线程中断干扰；当线程多语句向OutputStream输入对象时，每个语句执行时独占使用权，语句结束释放使用权，此时每个语句执行期间保证不被中断，但多语句间无法保证连续执行。

## 反射特性的算法工厂

反射机制允许运行时使用类名字符串创建对象实例。对于程序维护来说，我们不希望每增加一个新的算法，都需要修改算法接口提供类内部代码，以适应新的算法需求，这时反射机制优势就十分明显。如果存在反射机制，则只需要在运行时编译单元存在该类，就可以成功申请实例。

但C++并不支持反射机制，于是我们采用工厂模式，借助STL关联容器std::map存储类名与对应类创建方法，用宏模板间接实现反射特性。其详细实现方法此处不再赘述，添加算法类所需宏及其描述由下表给出：

表 1.5‑1 算法工厂宏模板

|  |  |  |
| --- | --- | --- |
| 宏名称 | 宏参数类型 | 描述 |
| ADD\_SUMMARY(solve\_name, sum) | 字段，字符串 | 为solve\_name算法类添加算法简介（非必需） |
| DISCSOLVE\_CLASSNAME(solve\_name) | 字段 | 差异化算法solve\_name  实际类名 |
| DISCSOLVE\_CLASS\_DEFINE(solve\_name) | 字段 | 差异化算法solve\_name  类声明 |
| DISCSOLVE\_CREATE\_FUNC(solve\_name) | 字段 | 差异化算法solve\_name  创建方法虚函数重写 |
| DISCSOLVE\_SOLVE\_FUNC(solve\_name) | 字段 | 差异化算法solve\_name  处理方法虚函数重写 |
| INDISCSOLVE\_CLASSNAME(solve\_name) | 字段 | 非差异化算法solve\_name  实际类名 |
| INDISCSOLVE\_CLASS\_DEFINE(solve\_name) | 字段 | 非差异化算法solve\_name  类声明 |
| INDISCSOLVE\_CREATE\_FUNC(solve\_name) | 字段 | 非差异化算法solve\_name  创建方法虚函数重写 |
| INDISCSOLVE\_SOLVE\_FUNC(solve\_name) | 字段 | 非差异化算法solve\_name  处理方法虚函数重写 |

C:\Users\88033\AppData\Local\Microsoft\Windows\INetCache\Content.Word\算法工厂.emf

图 1.5‑1 算法工厂类图

所有差异化算法显式继承DiscSolve类，所有非差异化算法显式继承IndiscSolve类。套用上表中的宏模板声明算法类，将自动向算法工厂注册类名，全局可反射。

差异化算法需重写签名为void ImageSolve()的图像处理虚函数；非差异化算法需重写签名为Mat ImageSolve(const Mat&)的图像处理虚函数。处理函数签名宏定义在上表中已给出。

无论哪种算法，均需要重写创建方法Creat()，宏定义在上表中已给出。在创建方法函数体内，可以指定该算法所需输入参数表，算法工厂依据该参数表向命令行提示输入需求（当命令行输入了该算法名时），并执行参数检查。

算法一旦声明，程序使用者即可使用命令行指令通过算法名调用该算法，或者在list指令中找到该算法的名称、简介、参数需求。