ZWMeshWorks 介绍和开发指南

广州中望龙腾软件股份有限公司
2021 年 1 月

前言

开发团队希望通过参考这本《ZWMeshWorks 介绍和开发指南》，用户能够了解并且更容易地使用 ZWMeshWorks 平台，或者基于此平台进行二次开发工作，集成自研求解器，共同打造国产 CAE 软件生态圈。

本指南主要由三部分组成。

第 I 部分介绍了 ZWMeshWorks 平台对外提供的基本功能和性能介绍，这一部分是 ZWMeshWorks 提供的基础能力，默认情况不需要二次开发者做任何修改即可以直接使用，其中包括数据导入导出能力，几何建模与清理能力，网格能力，后处理结果可视化能力等，另外还提供了一系列的产品级基础功能供二次开发用户直接使用。

第 II 部分介绍了 ZWMeshWorks 平台支持的扩展功能，以及系统介绍了整个平台的软件架构以及各模块的 API 接口。二次开发用户可以结合平台提供的对外头文件以及该部分的介绍，了解各个模块承担的功能以及各个 API 接口的作用及用法，基于此进行用户插件的二次开发。

第 III 部分是集成求解器的开发实例，以集成开源的流体求解器 SU2_CFD 为例，对示例代码的编译方法及源码结构进行了说明。

最后，非常欢迎大家了解和使用 ZWMeshWorks 平台，并提出宝贵的意见和建议，开发团队将会不断迭代更新平台功能，以期使大家更方便地集成自己的求解器。中望软件希望可以和大家携手，为国产 CAE 软件产品化的发展贡献自己的力量，合作共赢。

ZWMeshWorks 开发团队

本项目为广州中望龙腾软件股份有限公司拥有的机密及特许专卖产品。
任何未经中望授权的使用、复制或转让本产品行为都是违法且被严格禁止的。
版权(c) 2021 广州中望龙腾软件股份有限公司所有
（受限于有限制的对外发布权及有限制的对外披露权）
中望公司版权所有。

本手册由**广州中望龙腾软件股份有限公司**授权发布。

目录

第 I 部分 基本功能介绍 1
第 1 章 ZWMeshWorks 平台 1
1.1. 平台目的 1
1.2. 用户对象 1
1.3. 行业应用 2
1.3.1. 中望结构仿真 ZWSim Structural 2
1.3.2. 中望电磁仿真 ZWSim-EM 2
第 2 章 平台功能介绍 3
2.1. 数据接口 3
2.1.1. 导入导出 CAD 文件 3
2.1.2. 导入导出 CAE 文件 4
2.1.3. 中望自研标准 ZDF 文件 4
2.2. 几何建模及模型清理 4
2.2.1. 国产自主建模内核 4
2.2.2. 曲线、边、面、体编辑工具 4
2.2.3. 模型检测与修复 1
2.2.4. 特征简化及面简化 1
2.3. 网格能力 2
2.3.1. 中望网格引擎 2
2.3.2. 网格剖分 2
2.3.3. 网格质量 4
2.3.4. 网格拾取 5
2.4. 结果后处理及可视化 5
2.4.1. 结果后处理 5
2.4.2. 2D 显示 5
2.4.3. 3D 显示 6
2.5. 提交计算及过程监控 7
2.6. 脚本能力 7
2.7. 矩阵求解器 8
2.8. 并行能力 8
2.9. 产品级基础功能 9
2.9.1. 统一的软件交互界面 9
2.9.2. 图纸文件版本兼容 9
2.9.3. 单位设置灵活切换 10
2.9.4. 全模块参数驱动 11
2.9.5. 函数定义使用 11
2.9.6. 支持撤销/重做 12
2.9.7. 多语言切换 13

2.9.8. 仿真结果过期自动提醒 13
2.9.9. 平台兼容 14
第 II 部分 扩展功能及 API 介绍 14
第 3 章 平台扩展能力介绍 14
3.1. 界面 UI 定制 14
3.2. 材料属性扩展 16
3.3. 关系型阵列 16
3.4. 显示样式定制 17
3.5. 计算参数 Check 定制 17
3.6. 输出文件定制 17
3.7. 结果报告定制 18
第 4 章 开发环境及软件架构 18
4.1. 开发环境 18
4.1.1. Windows 环境准备 18
4.1.2. Linux 环境准备 19
4.2. 平台软件架构 19
4.2.1. 几何建模系统 19
4.2.2. 仿真分析系统 20
4.2.3. CAD/CAE 数据转换 21
4.2.4. 三维渲染系统 22
第 5 章 模块接口 API 说明 23
5.1. 头文件目录构成 23
5.2. 各模块接口说明 23
5.2.1. 资源文件路径规范 23
5.2.2. 插件的构建及注册 24
5.2.3. 仿真树模块 27
5.2.4. 自定义 UI 模块 27
5.2.5. 网格模块 30
5.2.6. 显示模块 30
5.2.7. 计算模块 32
5.2.8. ZDF 模块 34
5.2.9. 其他工具模块 34
第 III 部分 集成求解器 Demo 展示 35
第 6 章 集成求解器开发实例 35
6.1. Demo 功能说明（SU2_CFD） 35
6.2. Demo 目录 35
6.3. Demo 编译步骤 37
6.4. 使用 ZWMeshWorks 加载自研插件 38
6.5. Demo 源码说明 38
6.5.1. inc 39
6.5.2. Resource 39
6.5.3. src 39
6.5.4. PostBuild.cmd 41
6.6. 常见问题解决方法 41

6.6.1. 编译失败 41
6.6.2. 计算后求解器无法使用 41

第 I 部分 基本功能介绍

第1章 ZWMeshWorks 平台

作为通过虚拟仿真实现生产制造中降本增效的重要工具，国产自主 CAE 软件的研发对提高我国自主创新能力、实现制造强国至关重要。中望软件在 2018 年成立了 CAE 研发中心，延揽国内外优秀人才，持续加大在 CAE 领域的研发与创新投入。

ZWMeshWorks 是中望软件开发的具有自主知识产权的前后处理平台，采用模块化设计思想，并提供了标准接口，具有灵活的定制性和高效的拓展性。基于中望自主三维几何建模内核的强大的建模能力，先进的网格剖分技术，以及完善的前后处理功能，可集成多学科求解器，从而能够帮助开发者快速实现集前处理、求解计算、后处理于一体的开发需求，显著提升专业 CAE 软件产品的开发效率。

1.1. 平台目的

- 在 ZWMeshWorks 平台上，开发 CAE 软件的公用模块，提供产品级别的基础模块，帮助解决不同团队的重复开发所耗费的资源；
- 基于中望自主前后处理平台 ZWMeshWorks，集成国产自研求解器，定制化国产 CAE 软件；
- 基于中望已有业务市场，快速推广软件，合作共赢（目前拥有 16 种语言版本，中望产品畅销全球 90 多个国家和地区，建立超过 260 家合作伙伴，全球正版用户突破 90 万）；
- 基于中望广泛用户群，进行大量市场验证，快速迭代，使国产 CAE 软件越加强大。

1.2. 用户对象

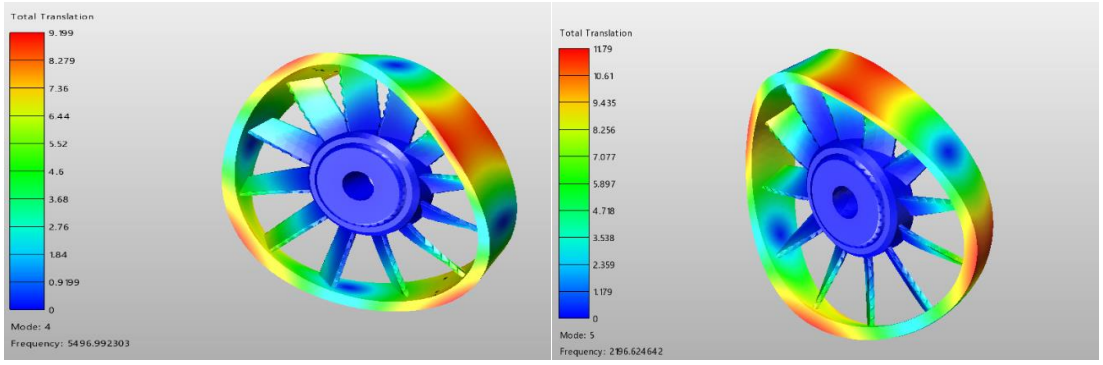
- 专业求解器程序开发者，正在规划研发自主仿真软件的开发人员，希望快速实现产品原型，通过 ZWMeshWorks 平台，可以方便拿到计算求解数据和绘制结果数据，进行迭代调试；
- 拥有自主求解器的客户，希望将求解器产品化，形成 CAE 软件在市场进行推广的客户；
- 有多学科仿真需求，希望在统一环境下使用的客户；

- 有二次开发需求的商业团队，希望自主定制一套完整的 CAE 软件产品。

1.3. 行业应用

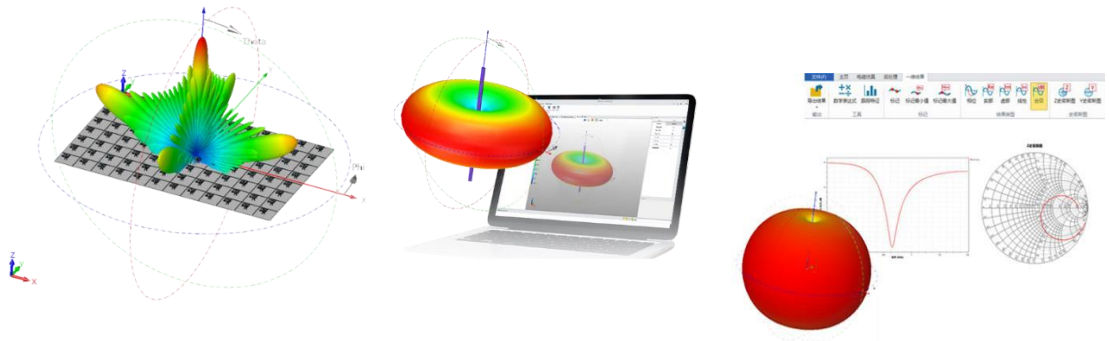
1.3.1. 中望结构仿真 ZWSim Structural

中望结构仿真软件是基于 ZWMeshWorks 平台进行二次开发的专业结构仿真分析软件。支持线性静力分析、线性屈曲分析、 模态分析、稳态热传导分析、瞬态热传导分析等，可帮助机械、航空航天、土木工程等领域工程师 准确评估产品结构设计的合理性与科学性，从而缩短企业的产品开发周期，实现降本增效。



1.3.2. 中望电磁仿真 ZWSim-EM

中望电磁仿真软件是基于 ZWMeshWorks 平台进行二次开发的专业电磁仿真分析软件，可对天线、微波器件、雷达等 产品进行快速准确的仿真，分析辐射、散射以及其他复杂结构宽带电磁问题，帮助企业用户提高电 磁仿真分析的效率、获得更精确的计算结果。



第2章　平台功能介绍

2.1. 数据接口

2.1.1. 导入导出 CAD 文件

ZWMeshWorks 具有**完全自主开发的通用中性数据** CAD 格式转换引擎，支持常见格式 IGES, STEP, Parasolid，STL, DWG/DXF, PDF, VDA, VRML 等的导入导出，同时都有对应的 API 接口提供，同时 ZWMeshWorks 平台还集成了商业格式交换引擎，支持常见主流商业系统文件的导入，包括 Catia，NX，Creo，SolidWorks，Inventor，SolidEdge 等

2.1.2. 导入导出 CAE 文件

ZWMeshWorks 将在 4 月份推出的 V1.2 版本支持 Nastran 的 BDF/OP2 文件的导入导出以及相互转换功能，其他文件格式的转换在未来版本持续开发迭代更新。

2.1.3. 中望自研标准 ZDF 文件

随着 CAE 软件的迅速发展，特别是多物理场耦合仿真，对适用于各学科的数据交互标准的需求非常迫切，然而目前公开的数据格式中并没有广泛采用。

ZWSoft Document File(ZDF)是 CAE 多学科仿真中重要的数据格式，其一般目的是为记录多学科仿真中数值求解有关的计算机数据提供标准。

ZDF 系统包含两部分：

- 记录数据的标准格式；
- 使用该格式读写仿真数据的软件。

ZDF 标准提供通用、易用、可兼容、可扩展的一系列约定，用于描述一个或多个仿真问题，并通过公开提供给仿真软件得到应用。ZDF 数据格式使软件之间、计算节点之间、不同平台之间的数据交互更为简单和稳定。基于 ZDF 标准数据格式可开发电磁、流体、力学、热学、声学、光学等多种学科仿真软件。

2.2. 几何建模及模型清理

2.2.1. 国产自主建模内核

ZWMeshWorks 拥有独立自主的几何内核，可以支持中国企业基于此内核技术进行行业产品开发。ZWMeshWorks 支持参数化建模和直接建模的交叉混合使用，参数化模型可以自由转换为无参模型；同时曲面建模和实体建模均可自由在实体、曲面结果之间切换，曲面和实体可以直接交叉布尔运算。

强大的 Overdrive 建模内核

直接编辑技术

混合建模技术

2.2.2. 曲线、边、面、体编辑工具

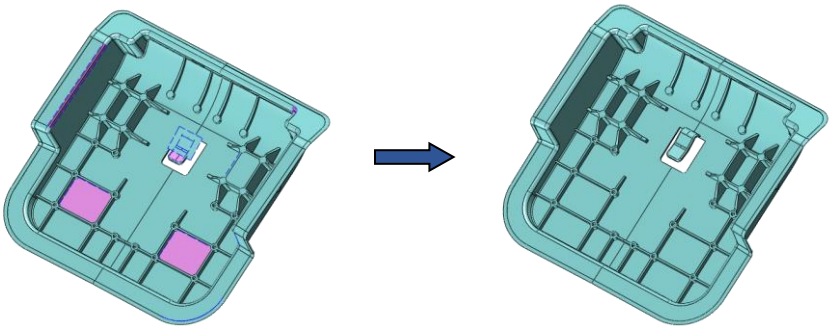
在 CAE 仿真分析中，当导入几何模型以后，经常需要先对模型的边、面、体作出一定的处理，以保证网格剖分中的单元质量符合算法要求，在 ZWMeshWorks 中，提供了很多工具可以方便用户对几何模型进行调整，主要包括对曲线、边、面、体的编辑工具，具体功能建议阅读 ZWMeshWorks 的帮助文档进行详细了解。

- 曲线编辑
  - 曲线边界编辑
  - 连接曲线
  - 修改 NURBS 曲线
  - 打断曲线
  - 延伸曲线
  - 延伸成角
- 面编辑
  - 合并面
  - 自动面合并
  - 扩大面
  - 曲面-曲面分割
  - 曲面-曲线分割
  - 简化-闭合间隙
- 边编辑
  - 拟合
  - 分割边
  - 连接
- 体编辑
  - 缠绕阵列到面
  - 圆柱折弯
  - 圆环折弯
  - 扭曲
  - 锥削
  - 炸开面
  - 修改圆角
  - 修改孔
  - 简化特征

2.2.3. 模型检测与修复

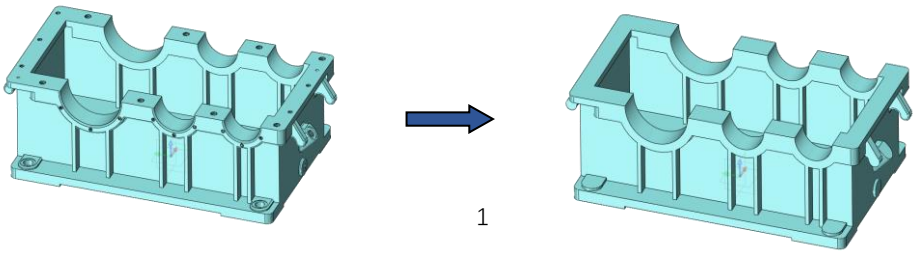
ZWMeshWorks 可以智能检测修复几何，提升清理几何效率。提供了自动和手动的方法，具备强大的修复能力，可以处理面边间隙及不匹配顶点，分析并修复多种拓扑条件，获得完美实体模型。

通过 ZWMeshWorks 提供的工具集，可以精确定位开放边界，导入数据快速修缮。



2.2.4. 特征简化及面简化

ZWMeshWorks 提供了特征简化功能，通过该特性可以去除对仿真精度没有太大影响但影响仿真效率的小孔，腰槽以及小圆角等；同时提供了面简化功能，可以合并小平面和狭长面，提升整个分网的质量。



2.3. 网格能力

2.3.1. 中望网格引擎

ZWMeshWorks 内集成了中望的网格引擎和开源 Netgen 引擎。
中望网格剖分引擎高效稳定:

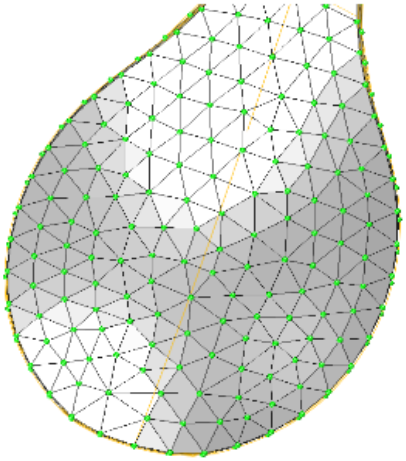
单元类型		中望网格ZWMesh	Netgen/Tetgen	Gmsh
1D		√	X	X
2D	三角形	√	√	√
	四边形	√	X	√
3D	四面体	√	√	√
	六面体	√	X	√

中望网格ZWMesh划分效率(Intel Core i7 3.6 GHZ)

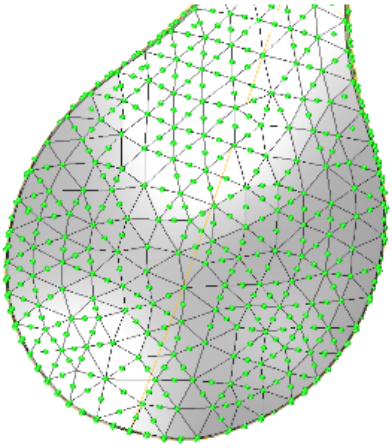
- 三角形：平面 ≥ 30000 triangles / s，复杂曲面 ≥ 3000 triangles / s
- 四边形：平面 ≥ 6000 elements / s，复杂曲面 ≥ 1000 elements / s
- 四面体：≥ 33000 tetrahedrons / s

2.3.2. 网格剖分

- 支持多种单元类型的剖分
  2D-Triangle
  2D-Quad Hybrid
  2D-Quad All
  3D-Tetrahedron
  3D-Hexa Hybrid
  1D-Curve
- 支持一阶、二阶、各项同性、各项异性

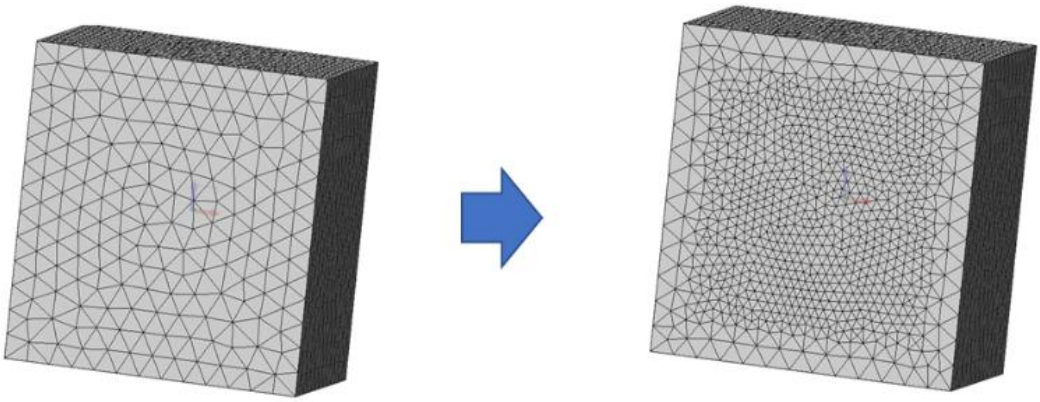


一阶网格

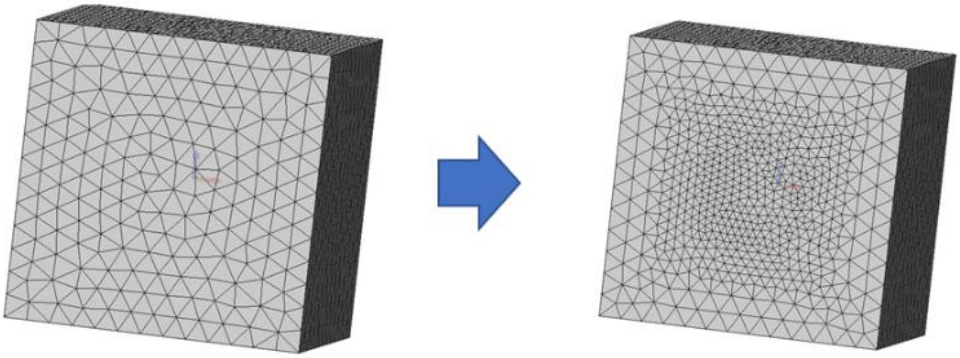


二阶网格

- 边、面、体局部尺寸控制，硬点控制

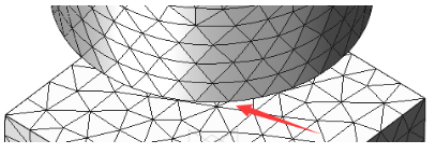


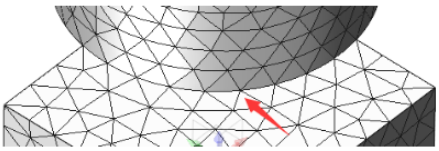
单元尺寸变化过渡剧烈

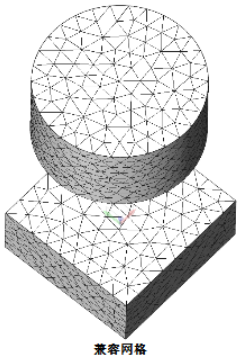


单元尺寸变化过渡平缓

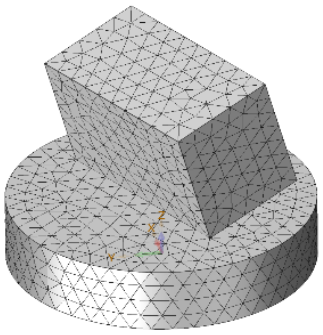
- 一键生成兼容网格



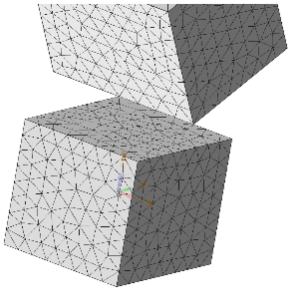




兼容网格



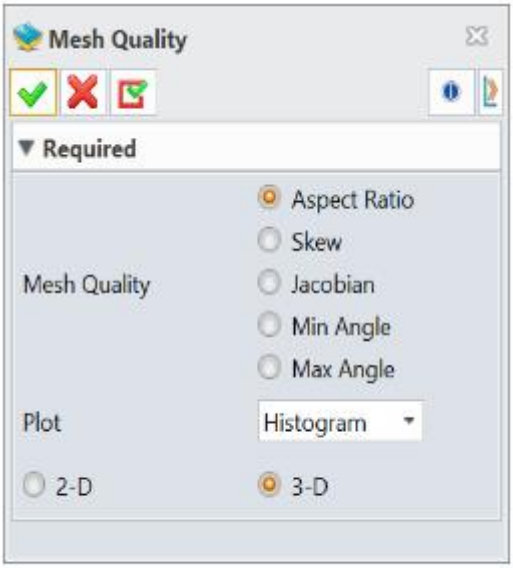
兼容网格



兼容网格

2.3.3. 网格质量

网格质量可以对网格划分后的模型进行网格质量检测，得出结果后以图表的形式展现网格的质量情况。



网格质量

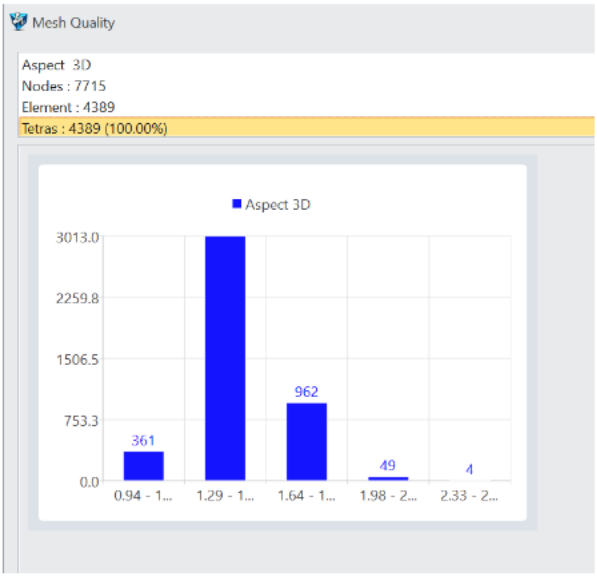
网格质量：可选择 Aspect Ratio（长宽比）、Skew（扭曲度）、 Jacobian（雅克比比率）、Min Angle（最小角度）和 Max Angle（最大角度）进行网格质量计算。

绘制：可选择结果绘制柱状图或直方图。

2-D：选择平面单元网格检测。

3-D：选择体单元网格检测。

设置以后可以得到网格质量的检测结果并以柱状图或者直方图来表现。



网格质量检测结果

2.3.4. 网格拾取

目前暂不支持网格的拾取，在即将推出的版本中将支持网格拾取，同时会串通网格拾取，几何拾取，ui 联动等各种场景，提供给二次开发用户方便且友好的网格拾取接口集。

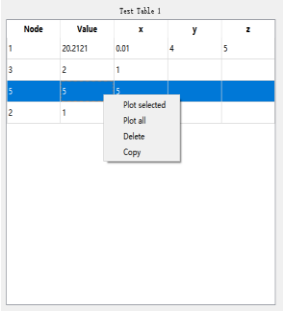
2.4. 结果后处理及可视化

2.4.1. 结果后处理

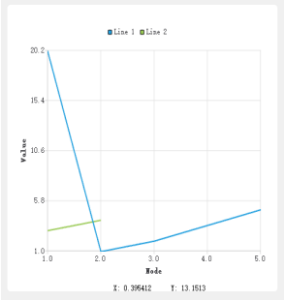
目前 1.1 版本的结果后处理更多地是由学科 app 自己进行计算，在即将推出的 1.2 版本中，ZWMeshWorks 平台将支持流线图的计算，同时在未来版本将会统一对后处理计算进行封装抽象，提供更多便捷的 API 供 app 使用。

2.4.2. 2D 显示

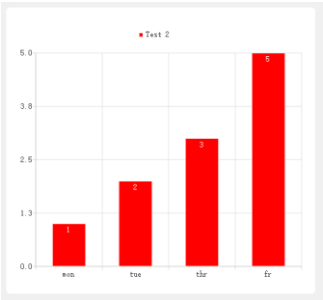
ZWMeshWorks 目前可以显示的 2D 结果包括如下：



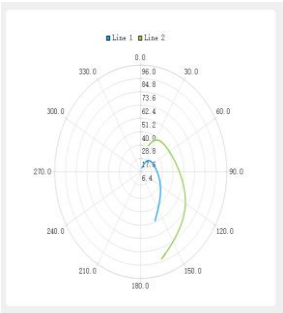
表格



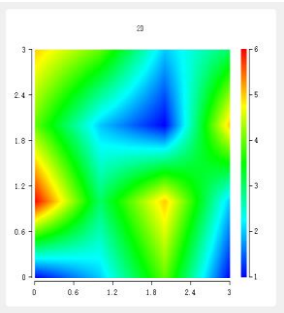
笛卡尔



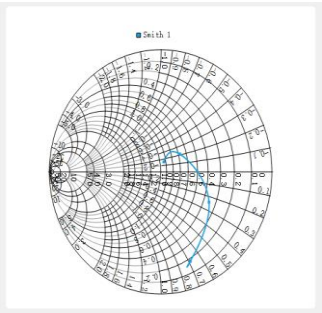
柱状图



极坐标图

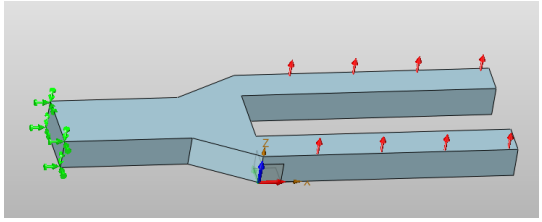


2D 图

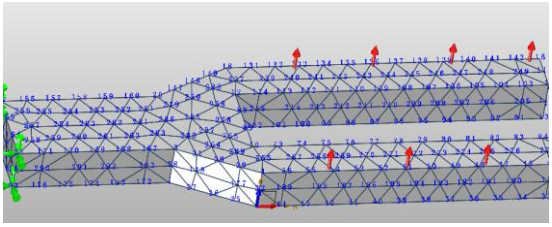


史密斯图

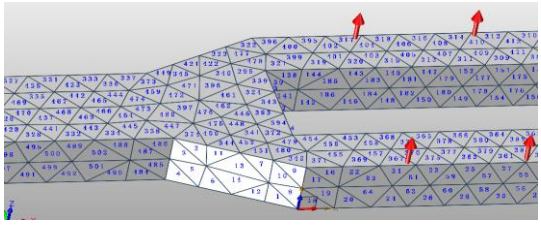
2.4.3. **3D** 显示



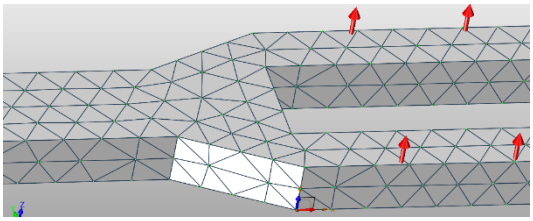
前处理对象标识



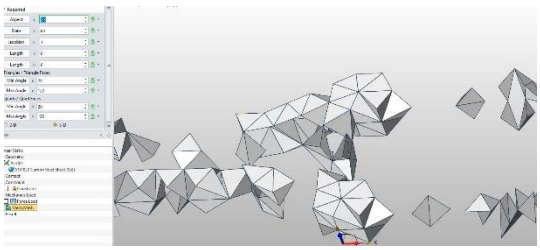
网格节点编号



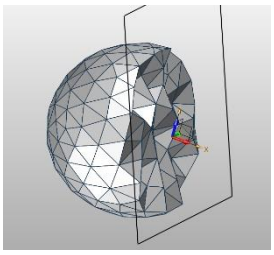
网格单元编号



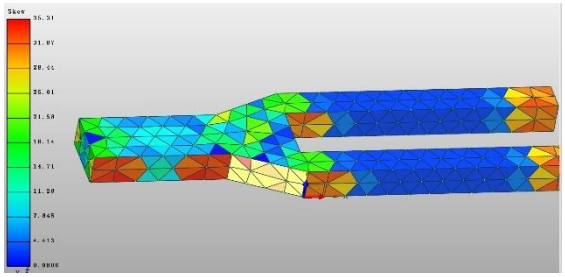
网格单元节点显示



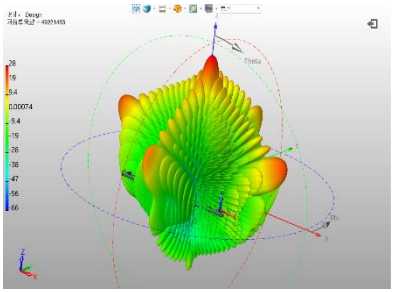
网格过滤显示



网格剖切显示



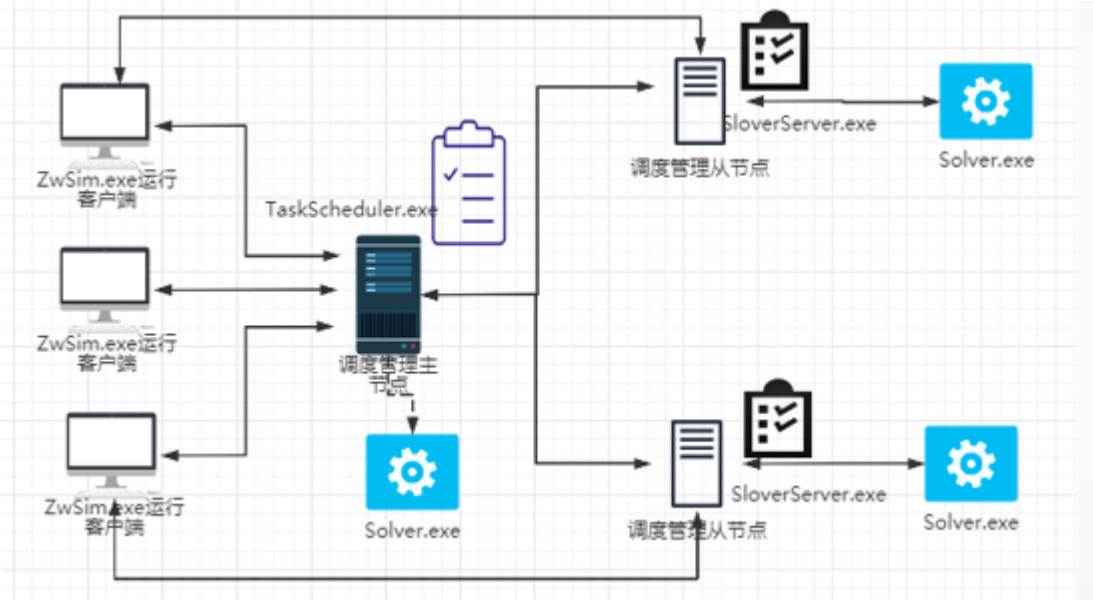
3D 图



3D 图

2.5. 提交计算及过程监控

ZWMeshWorks.exe 和求解器之间是通过 TCP/IP 进行通信的，在整个架构中的逻辑如下：



对于二次开发者，不用去关心具体调度内部的行为，而只需要在在计算模块中填写相应的业务逻辑即可。其中的调度管理主节点完成了资源管理，负载均衡，任务控制等功能，这些都由 ZWMeshWorks 内部完成。

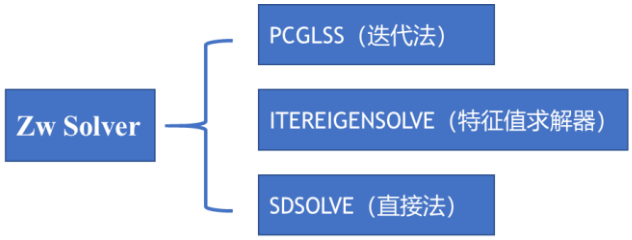
2.6. 脚本能力

ZWMeshWorks 当前有 Macro 机制，在用户主动开启后可记录下用户的操作，二次开发用户可以使用该模块来快速完成案例的创建，也可以使用该工具来保障二次开发的质量。

ZWMeshWorks 目前正在开发全模块的 python 驱动，敬请期待。

2.7. 矩阵求解器

工欲善其事必先利其器，优秀矩阵求解器便是决定 CAE 软件求解速度与精度的最强利器。中望矩阵求解器支持多种求解方式，适用于多种不同的领域。
中望求解器包含直接法求解器，基于 PCG 的迭代法求解器和基于 Lanczos 的特征值求解器。其求解精度和求解速度基本满足了单机用户的需求。一千三百万自由度的问题，在 Dell XPS 13 notebook 上的求解速度为 44 分钟。



矩阵求解器	DOF	求解时间	CPU
ZwSolver	**13, 823, 972**	**44min**	**Intel i5 @2.20 GHz, 8GB RAM**

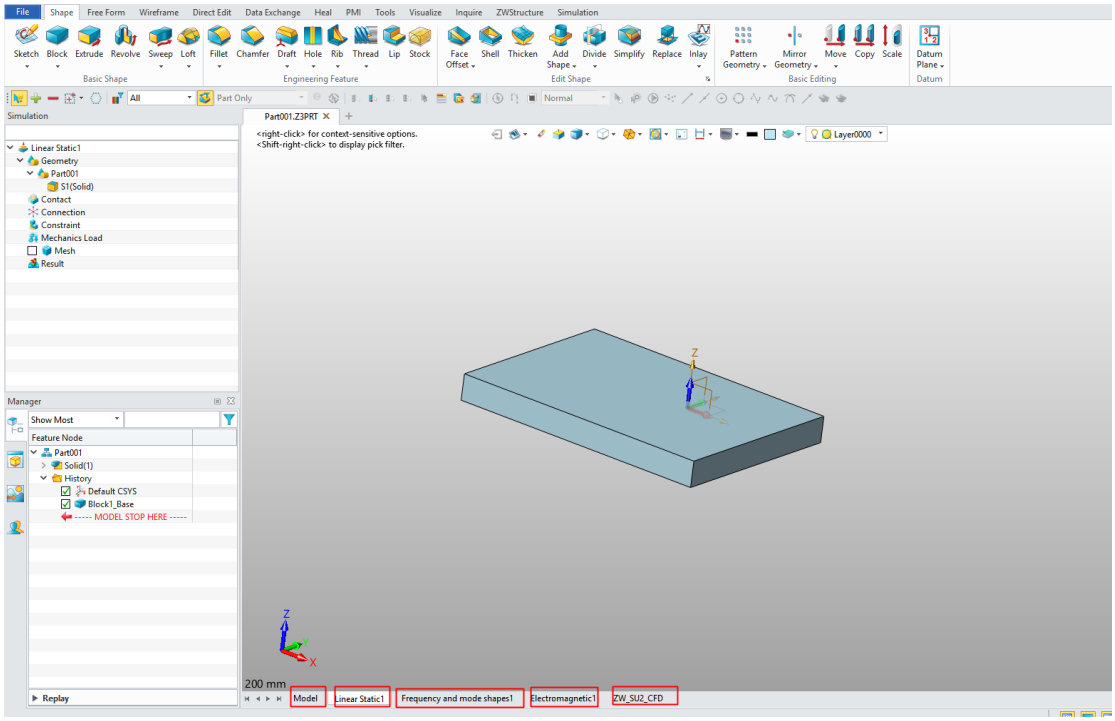
除此之外，中望矩阵求解器还支持自动组装矩阵，支持多工况和 MPC，用户所做的只是将单元矩阵、约束、载荷等信息传入求解器，大大优化了用户的体验。

2.8. 并行能力

ZWMeshWorks 的并行模块正在开发中，后续稳定后将推出详细介绍文档。

2.9. 产品级基础功能

2.9.1. 统一的软件交互界面



ZWMeshWorks 搭建了一个统一的操作逻辑，在视图区有仿真树和几何模型树，中间是视图区，下方是每个仿真任务。

ZWMeshWorks 可以在一个环境中，同时进行几何建模，网格剖分和仿真数据配置等功能，同时支持多文档，多仿真任务同时计算，数据间互不影响。这个强大的优势可以保证基于 ZWMeshWorks 开发的 app 具有统一的操作逻辑，即可以形成独立的产品，又保留了未来多物理场仿真的可能。

2.9.2. 图纸文件版本兼容

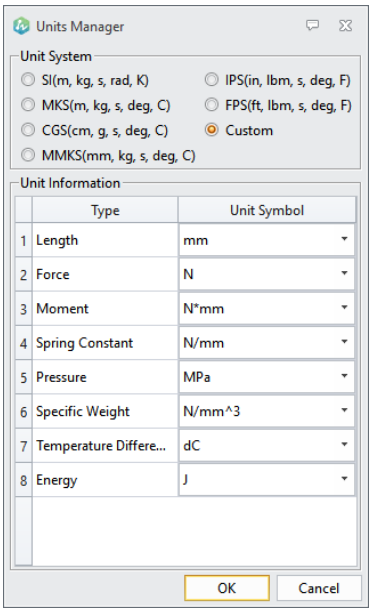
ZWMeshWorks 形成的文件有 Z3PRT 和 Z3ASM 文件，同时文件后缀支持自定义，该文件是二进制文件，具有强大的产品级优势：

- CAD 数据和 CAE 数据在同一个文件中且互不影响；
- 文件体量小，保存的都是二进制数据；
- 学科间数据兼容，互不影响；
- 版本兼容，在大版本间保持向后兼容，在小版本间保持前后互相兼容；
- 数据安全，二次开发用户的数据只有自己知道具体含义，其他人无法解析。

2.9.3. 单位设置灵活切换

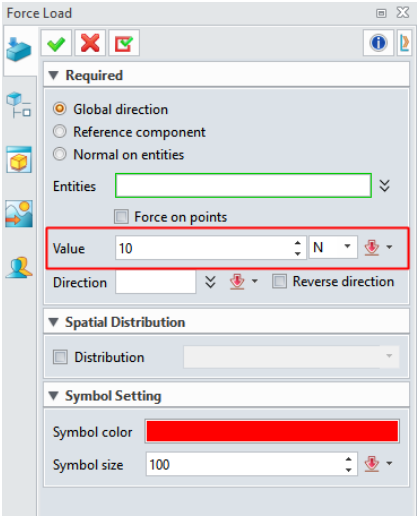
ZWMeshWorks 支持全局的单位设置和在界面输入时的单位设置。

- 全局单位设置



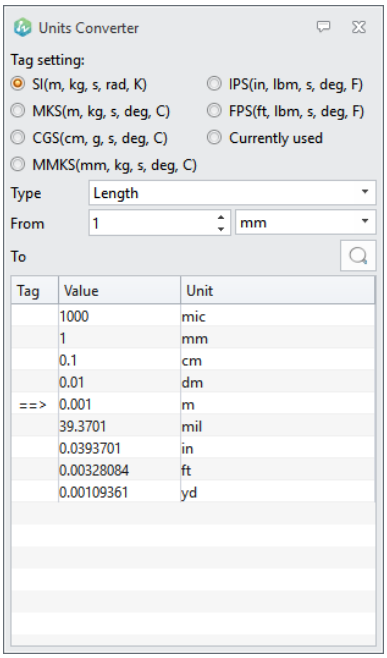
支持单位系统的选择，包括 SI/IPS/MKS/FPS/CGS/MMKS 六种单位系统的选择，同时也支持用户自定义物理量的单位，修改之后，所有仿真数据的数值都会相应更新，按最新的单位设置进行展示。

- 局部单位设置



在界面输入数值时，默认以全局单位设置进行显示，也可以在输入时动态的局部调整。

- 单位转换工具



ZWMeshWorks 提供了一个单位转换计算的工具，可以供用户在配置案例时进行计算，相应的数值进行填写。

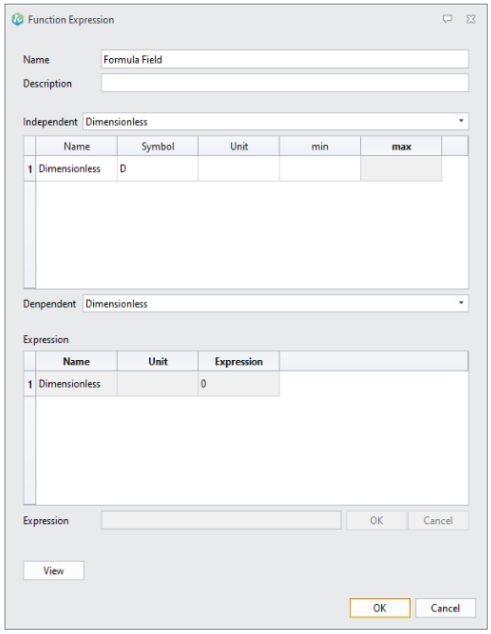
2.9.4. 全模块参数驱动

ZWMeshWorks 在所有支持输入数值的地方都支持直接输入变量或者表达式，不用二次开发者额外添加任何代码，当在 ZWMeshWorks 提供的变量管理器中更新变量的数值以后，所有使用到该变量的地方都会自动更新。

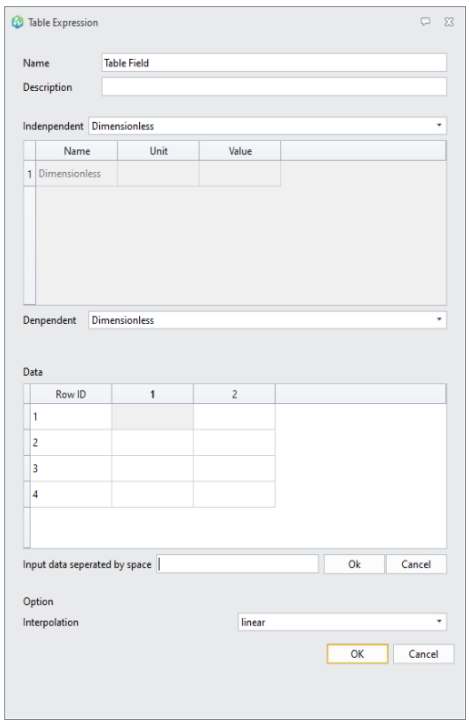
这一点对于集成于 ZWMeshWorks 上的求解器来说，可以很方便地使求解器具备参数扫描和参数优化的功能。

2.9.5. 函数定义使用

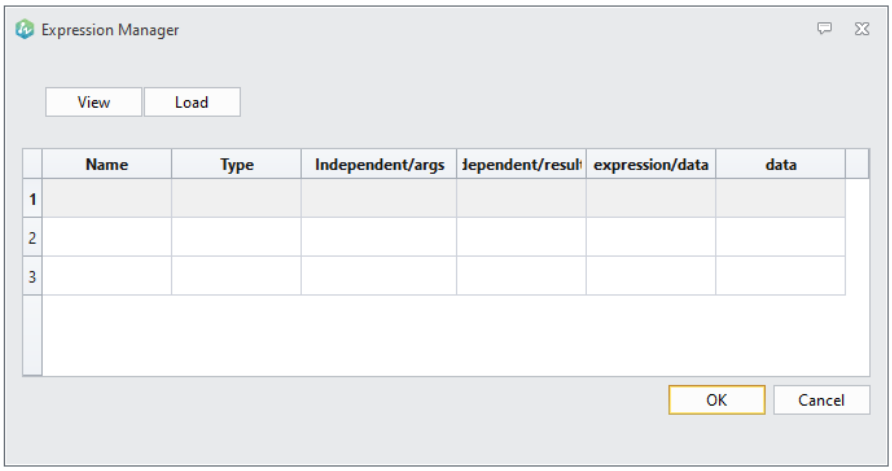
在仿真中，经常需要用到非线性函数的输入，ZWMeshWorks 也已经提供了相应的功能，用来查看，创建，修改函数表达式，表格表达式。



函数表达式对话框



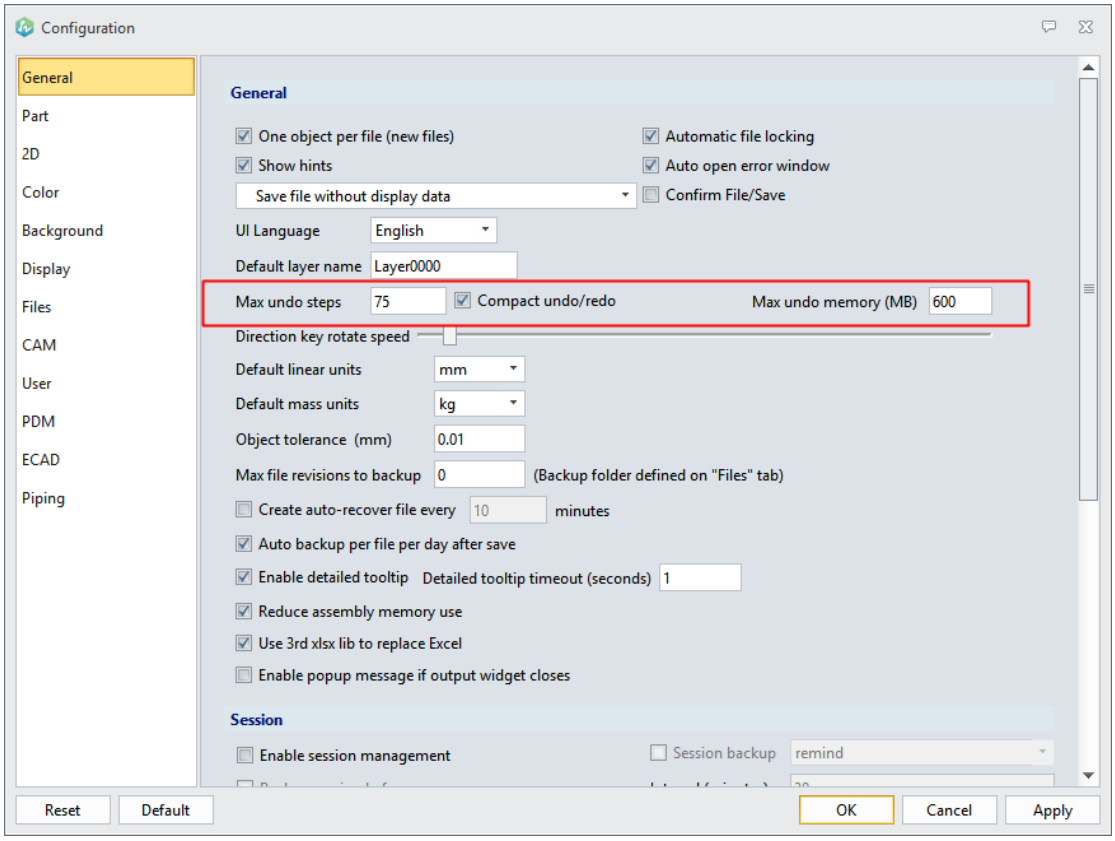
表格表达式对话框



表达式管理器对话框

2.9.6. 支持撤销/重做

ZWMeshWorks 天然支持撤销、重做功能，并且可以对撤销、重做的最大次数以及最大缓存进行配置。这一点对二次开发用户来说是极大的方便，只需要负责数据的读写即可，所有备份回退工作都已经在 ZWMeshWorks 平台内部完成。



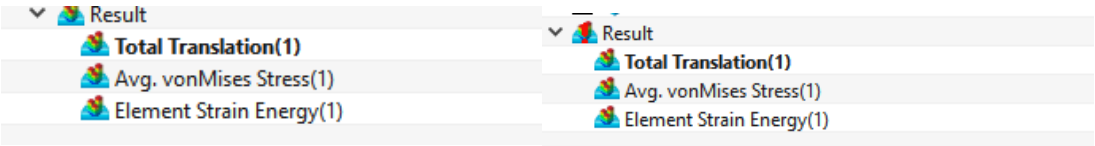
在配置对话框中，可以完成对撤销/重做的配置。

2.9.7. 多语言切换

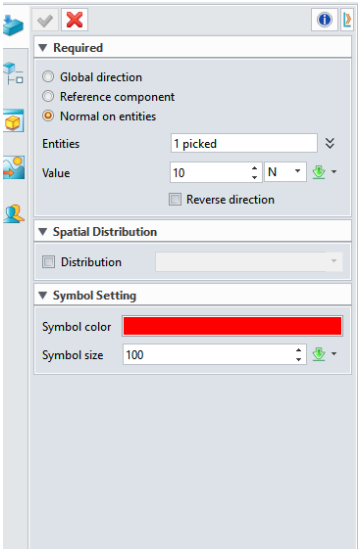
ZWMeshWorks 拥有 16 种语言版本，对于集成到 ZWMeshWorks 平台的 app 开发者来说，只需要将其特有的资源文件进行翻译替换即可，无需修改任何代码即可以完成多语言版本的切换。

2.9.8. 仿真结果过期自动提醒

当配置的案例计算出结果后，如果用户继续去修改数据，ZWMeshWorks 平台会自动识别并标记出结果过期提醒。



在计算过程支持查看配置的数据，同时 ZWMeshWorks 会自动地将用户的 UI 中的 OK 和 APPLY 按钮给屏蔽，防止用户修改到数据。



2.9.9. 平台兼容

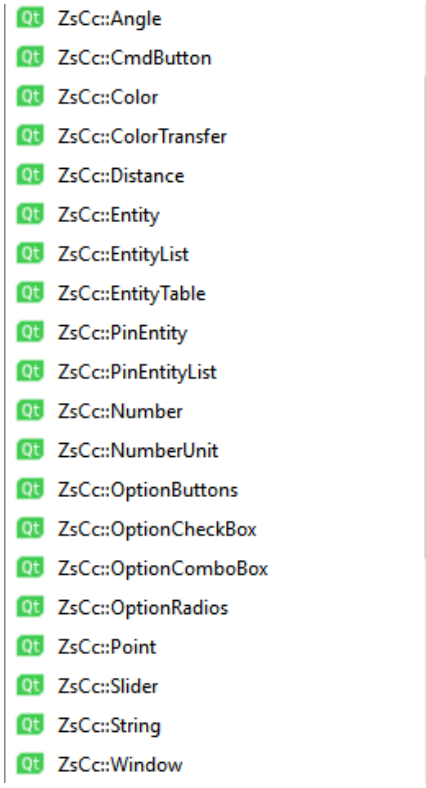
ZWMeshWorks 目前仅支持 windows 版本，后续迭代版本即将支持 Linux。目前提供出去的二次开发接口都是平台无关的，待后续版本更新后，用户代码只需要按照配置进行重新编译链接即可以将自己的 APP 加载在 Linux 版本的 ZWMeshWorks 上。

第 II 部分 扩展功能及 API 介绍

第3章 平台扩展能力介绍

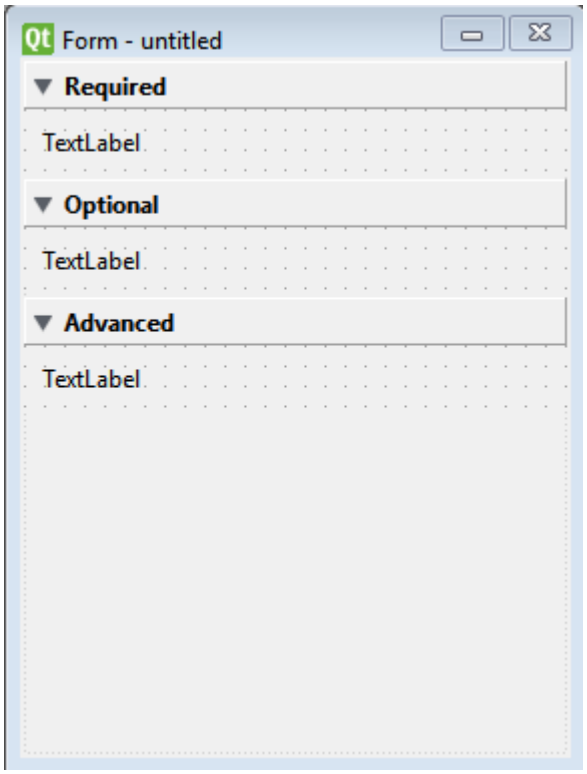
3.1. 界面 UI 定制

基于 QT Designer 进行扩展的 ZW 插件包含了众多方便 CAE 仿真时使用的控件，包括如下内容：



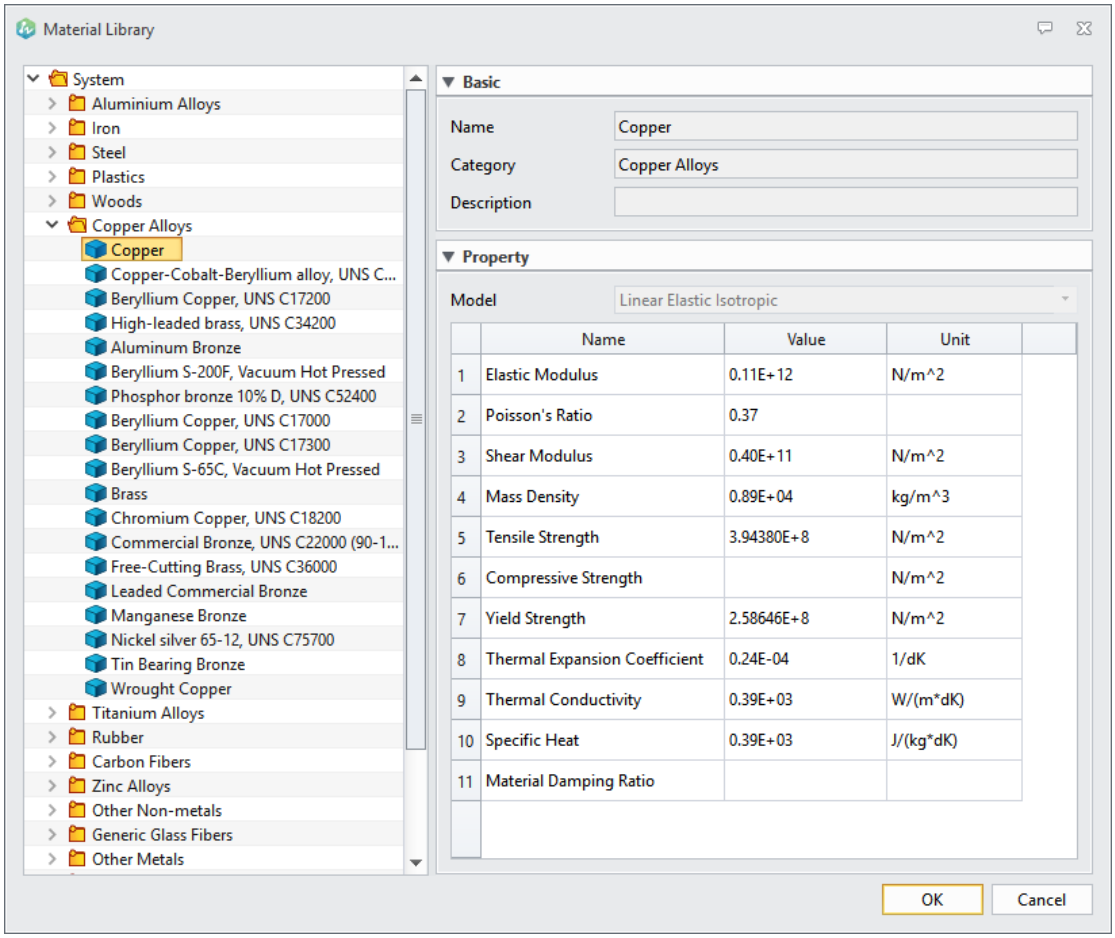
其中的 ZsCc::Entity 控件可以方便地和视图区的几何进行交互，并支持动态化地配置拾取的过滤器；ZsCc::Point 控件可以灵活地在视图区中获取点、中点、交点等；ZsCc::NumberUnit 控件可以进行单位的配置。

当二次开发用户需要进行新的 ui 绘制时，只需要打开加载 ZW 插件的 QT Designer，即可以快速地形成一个新的 ui，并和视图区几何进行联动。



3.2. 材料属性扩展

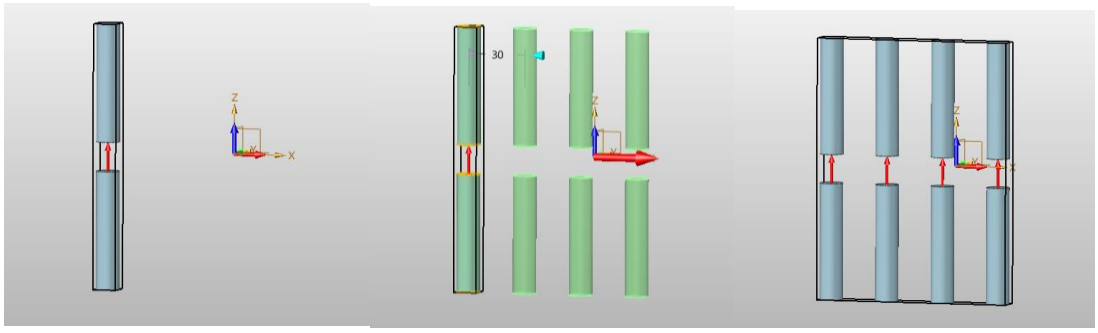
ZWMeshWorks 的材料库界面如下：



其中右下角的材料属性部分支持用户自定义扩展。

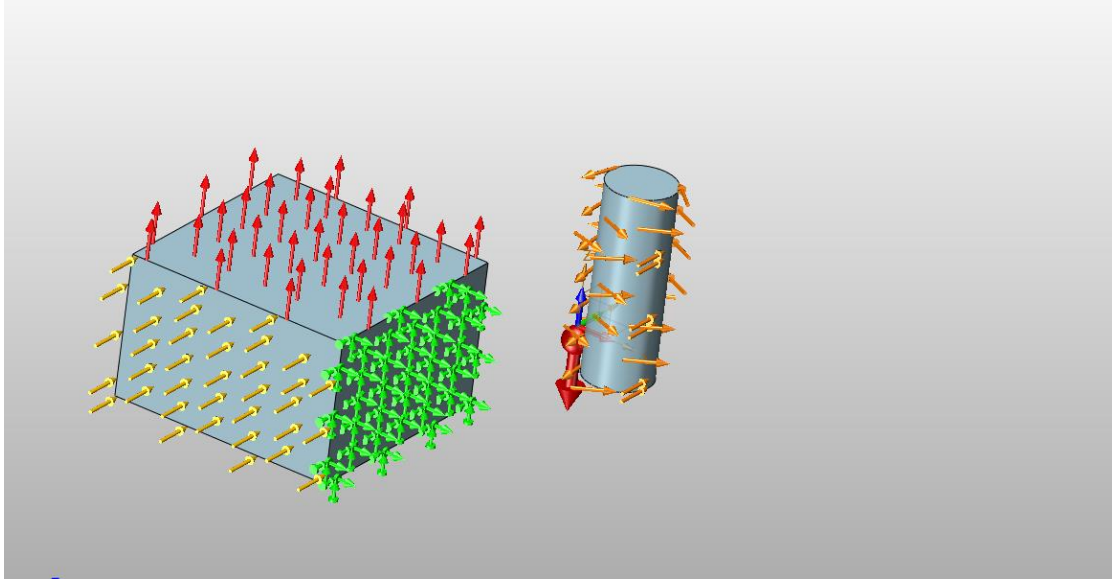
3.3. 关系型阵列

ZWMeshWorks 平台支持用户进行扩展吗，当进行几何阵列时，可以自动地将用户的仿真数据进行阵列，具体展示如下：



3.4. 显示样式定制

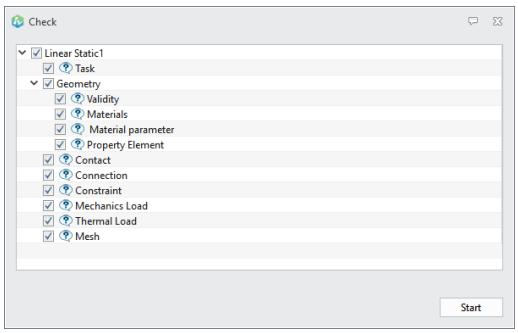
目前 ZWMeshWorks 默认提供了多种前处理显示对象的样式，用户只需要调用相应的接口就可以在对应几何边界上进行显示。

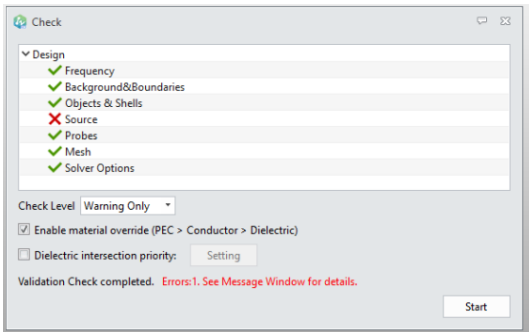


同时，如果默认显示样式不满足使用需求是，用户也可以通过显示模块的接口自由定制显示样式。

3.5. 计算参数 Check 定制

ZWMeshWorks 在计算前会自动调用 check 模块进行数据的校验，不同 app 可以有自己不同的 check 项，这些都可以通过接口设置，进行定制化。



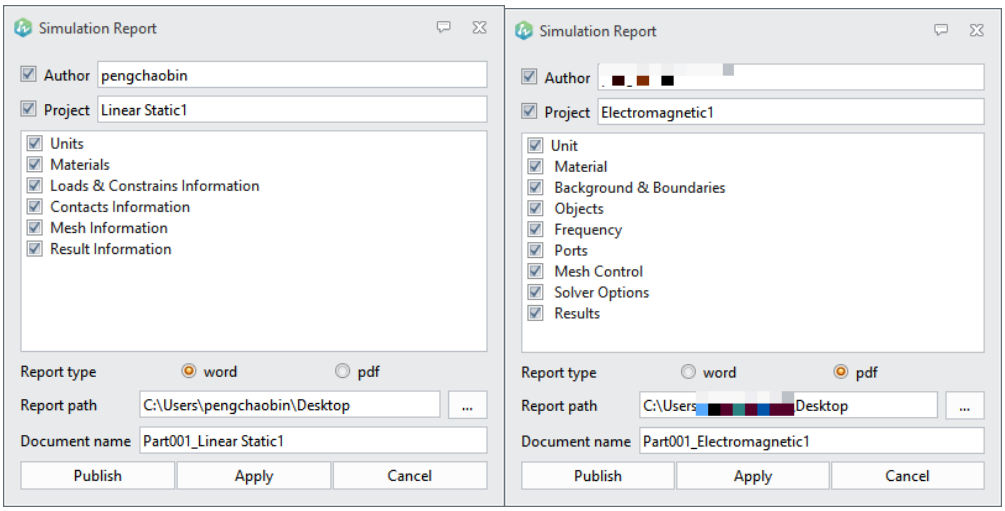


3.6. 输出文件定制

ZWMeshWorks 目前输出文件定制需要编写代码进行转换，在后续迭代版本中将逐步支持校本化转换和模板化转换，进一步降低用户获取求解数据的复杂度。

3.7. 结果报告定制

ZWMeshWorks 支持导出 word 版和 PDF 版的结果报告，具体的导出项由 app 自行扩展，都有相应的二次开发接口使用。



第4章　开发环境及软件架构

4.1. 开发环境

ZWMeshWorks 平台采用 C++语言编写，目前支持到 C++14 标准，当前提供的 API 接口都是 C++接口，在后续版本迭代中，将逐步支持 Python 等脚本语言接口。

ZWMeshWorks 平台目前仅支持 x64 模式，不支持 x86 模式。

4.1.1. Windows 环境准备

Windows 环境下需要以下软件，按照默认步骤安装即可，具体版本要求如下：

- ZWMeshWorks 2021 版本

  必需安装，用于使用 ZWMeshWorks 平台的功能，同时可以加载用户二次开发的动态库，将 ZWMeshWorks 平台和用户求解器的功能串联起来。

- Visual Studio 2015 及以上

  必需安装，目前需要的 PlatformToolSet 是 v140 版本，用于编译插件。

- Qt Designer 5.9.7，带 ZW 自研扩展控件

  按需安装，如果用户需要可视化绘制 UI 文件，则需要安装，同时必需加载 ZW 自研的扩展控件才可以正常使用。目前 ZWMeshWorks 平台是基于 QT 5.9.7 开发，但对外接口不涉及任何 QT 接口，一方面是方便内部 QT 库的升级不影响用户代码，另一方

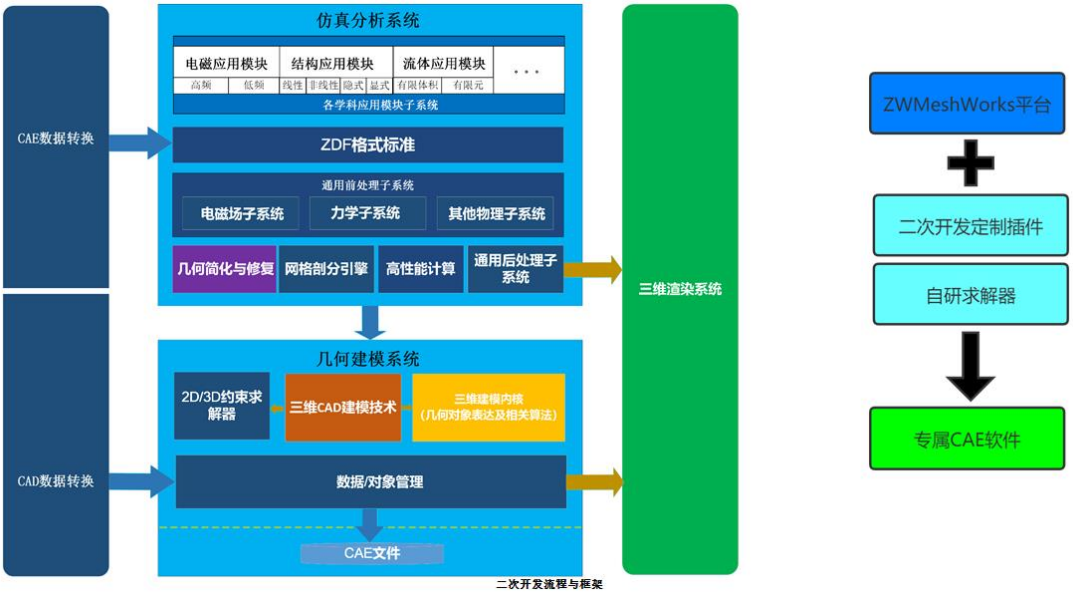
面也是方便替换其他 UI 库。

- MPI 库，推荐采用 MSMPI
  按需安装，如果用户需要使用 MPI 库功能，则需要安装。

4.1.2. Linux 环境准备

ZWMeshWorks 的 Linux 化正在开发中，近期将推出调度级别的 Linux 化，可以调用在远端服务器上的 Linux 求解器，后续版本将支持全平台的 Linux 化版本。具体环境要求待后续 API 提供后再详细说明。

4.2. 平台软件架构



二次开发流程与框架

从二次开发的角度，用户只需要提供一个基于 ZWMeshWorks 平台开发的定制动态库插件，以及提供自研求解器，即可定制自己的专属 CAE 软件。

在 ZWMeshWorks 软件平台上，对外主要抽象了几个子系统，每个子系统都具备单独扩展二次开发的能力，同时每个子系统会完成特定的功能。

4.2.1. 几何建模系统

负责完成图纸二进制数据的管理和几何拓扑关系的维护与更新，是 ZWMeshWorks 平台的最底层模块。

- 数据/对象管理
  对用户的图纸数据做了抽象包装，是多学科兼容，版本间数据兼容的基础，同时以二进制形式存储了数据，只有插件开发者能解析该数据，具有一定的数据安全性。对外提供了数据对象的读写接口，用户通过该模块可以获取对应的数据。

- 几何内核

主要包括了 ZWMeshWorks 的建模内核部分数据，在该模块提供了几何拓扑数据读取的接口，用户可以通过这些接口组装自己的建模特征，也可以直接使用 ZWMeshWorks 的 UDF 功能进行组合；用户也可以通过该模块获取几何拓扑的相关数据。

4.2.2. 仿真分析系统

负责完成 CAE 数据及业务功能的管理，具备很大的自由开发灵活度，是 ZWMeshWorks 二次开发用户扩展功能时的必备模块。

- 几何简化与修复

对于 CAE 仿真来说，会比 CAD 系统提出更多的几何简化与修复需求，该模块是基于底层的几何内核模块开发而来，对外提供的更多的满足 CAE 系统要求的几何简化与修复工具。

- 网格引擎

目前集成了中望网格引擎和开源的 NetGen 引擎，对外提供了网格剖分工具。使用该模块的接口，用户可以集成自己的网格剖分引擎，也可以通过该模块获取网格单元、节点、拓扑数据。

- 高性能计算

ZWMeshWorks 的 HPC 模块，负责完成计算资源的分配和管理，并且对计算过程进行监控，同时对外提供了并行算法库接口，在用户程序中，可以直接使用该模块的接口提供的功能，调用对应的并行算法库接口。

- 通用前处理子系统

结构、电磁、流体、声学、光学等学科强关联的模块都有自己的特定边界条件，比如 Load， Pressure，Inlet，Outlet 等，当二次开发用户开发相应的模块时，肯定不想重复开发这些界面，所以在 ZWMeshWorks 平台上，会根据具体学科，提供相应的边界条件的读写接口，使用户可以快速地复用逻辑。同时，在未来这些边界条件将支持二次开发用户进行配置，允许在特定插件上进行扩展或裁剪。

- 通用后处理子系统

通用后处理子系统，对外提供了用于将计算结果数据转换为显示数据的通用接口，包括流线图计算接口，3D 云图单元值转换，节点值转换等通用接口，只需要将计算结果数据按指定的规则传递给该系统，即可以自动地完成结果数据到显示数据的转换，并绘制在视图区。

- ZDF 格式标准

随着 CAE 软件的迅速发展，特别是多物理场耦合仿真，对适用于各学科的数据交互标准的需求非常迫切，然而目前公开的数据格式中并没有广泛采用。

ZWSoft Document File(ZDF)是 CAE 多学科仿真中重要的数据格式，其一般目的是为记录多学科仿真中数值求解有关的计算机数据提供标准。

ZDF 系统包含两部分：

- 记录数据的标准格式；
- 使用该格式读写仿真数据的软件。

ZDF 标准提供通用、易用、可兼容、可扩展的一系列约定，用于描述一个或多个仿真问题，并通过公开提供给仿真软件得到应用。ZDF 数据格式使软件之间、计算节点之间、不同平台之间的数据交互更为简单和稳定。

ZDF 标准适用于下列数据：

- 结构网格、非结构网格、YEE 网格；
- STL 网格模型、CAD 几何模型；
- 量纲单位和无量纲信息；
- 材料信息；
- 变量和函数信息；
- 单元属性信息；
- 边界条件；
- 仿真计算参数；
- 仿真结果；

基于 ZDF 标准数据格式可开发电磁、流体、力学、热学、声学、光学等多种学科仿真软件。

- 学科应用模块

各学科基于通用前处理子系统之上获取到的数据是基于中望的几何拓扑数据和中望的网格拓扑数据内容，针对不同学科内部的通用计算属性配置要求，以及不同求解算法所需要的特殊数据处理流程，都包装在学科应用模块中。在学科应用模块中，完成了数据模板转换的功能，该部分接口是用来直接和求解器数据文件交互而设计提供的。

4.2.3. CAD/CAE 数据转换

负责完成 CAD/CAE 数据的导入导出，使 ZWMeshWorks 可以正常使用数据转换交互功能。

- CAD 数据转换

ZWMeshWorks 具有**完全自主开发的通用中性数据** CAD 格式转换引擎，支持常见格式 IGES, STEP, Parasolid，STL, DWG/DXF, PDF, VDA, VRML 等的导入导出，同时都有对应的 API 接口提供，同时 ZWMeshWorks 平台还集成了商业格式交换引擎，支持常见主流商业系统文件的导入，包括 Catia，NX，Creo，SolidWorks，Inventor，SolidEdge 等

- CAE 数据转换

ZWMeshWorks 将在 4 月份推出的 V1.2 版本支持 Nastran 的 BDF/OP2 文件的导入导出以及相互转换功能，其他文件格式的转换在未来版本持续开发迭代更新。

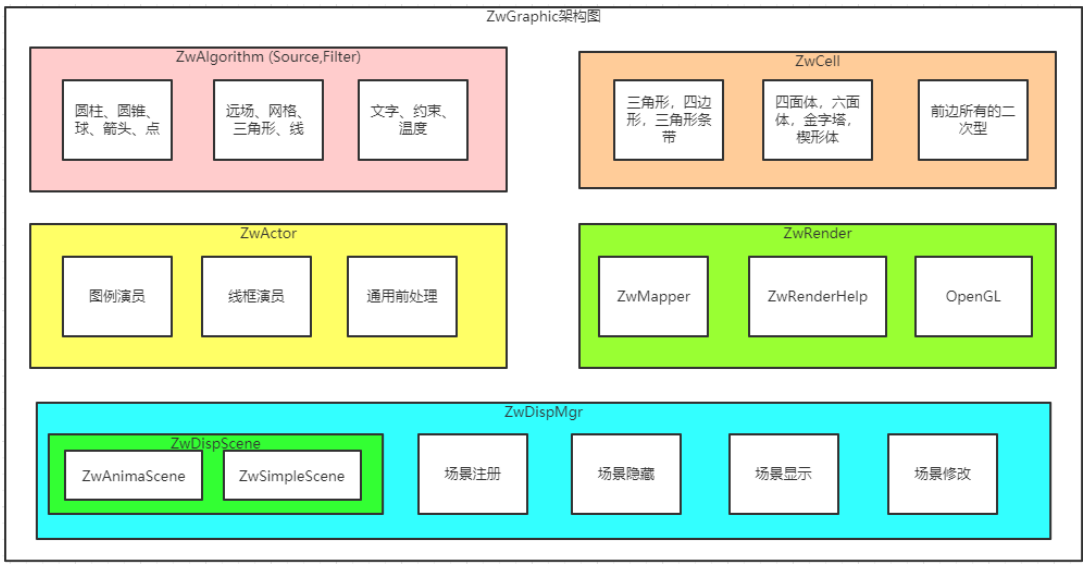
4.2.4. 三维渲染系统

ZWMeshWorks 提供了一套通用前后处理对象的三维显示系统，同时辅以表格和图表二维控件进行一定形式的显示，能更加直观地让使用者观察数据的特性，进行有关的数据分析等操作。

三维渲染系统提供了如下功能：

- 提供表格和二维图表显示功能，包括表格，笛卡尔，柱状图，极坐标图，2D 云图和史密斯图等接口；
- 三维渲染系统提供动画，剖切，等值面/等值线显示功能；
- 提供通用三维显示对象接口，包括色带，标注，文字，云图，矢量图等；
- 提供通用前处理的显示对象接口，包括网格，约束，载荷，接触等；
- 提供通用后处理显示对象接口，包括结构场，远场，电磁场，电流场和温度场等。

三维显示系统的架构图如下：



第5章　模块接口 API 说明

5.1. 头文件目录构成

SDK › Inc › ZWSim

Name	Date modified	Type	Size
calculate	1/22/2021 10:10 AM	File folder	
check	1/22/2021 10:10 AM	File folder	
communication	1/22/2021 10:10 AM	File folder	
db	1/22/2021 10:10 AM	File folder	
dbOperator	1/22/2021 10:10 AM	File folder	
expression	1/22/2021 10:10 AM	File folder	
form	1/22/2021 10:10 AM	File folder	
geom	1/22/2021 10:10 AM	File folder	
graphic	1/22/2021 10:10 AM	File folder	
gui	1/22/2021 10:10 AM	File folder	
material	1/22/2021 10:10 AM	File folder	
mesh	1/22/2021 10:10 AM	File folder	
result	1/22/2021 10:10 AM	File folder	
resultBaseLine	1/22/2021 10:10 AM	File folder	
scheduler	1/22/2021 10:10 AM	File folder	
utils	1/22/2021 10:10 AM	File folder	
zdf	1/22/2021 10:10 AM	File folder	
ZwArithmetic.h	6/12/2020 9:21 AM	C/C++ Header	1 KB
ZwImpl.h	9/5/2020 3:09 PM	C/C++ Header	5 KB
ZwModuleMgr.h	9/29/2020 10:29 PM	C/C++ Header	2 KB
ZwPackageMgr.h	9/29/2020 10:29 PM	C/C++ Header	4 KB
ZwType.h	9/15/2020 10:32 AM	C/C++ Header	3 KB

在 ZWMeshWorks 的安装目录路径下，/SDK/Inc/ZWSim/中可以查到目前版本提供的头文件，每个文件夹下是各个模块对外提供的扩展接口。具体可以参考头文件中的代码注释，更详细和系统的 API 接口将在后续迭代版本中不断细化说明。

5.2. 各模块接口说明

5.2.1. 资源文件路径规范

在动态库同级目录下，应该有一个 Resource 文件夹，其中包含以下截图中的文件夹。

Name	Date modified	Type	Size
Commands	1/22/2021 10:10 AM	File folder	
Configures	1/22/2021 10:10 AM	File folder	
Context	1/22/2021 10:10 AM	File folder	
Forms	1/22/2021 10:10 AM	File folder	
Icons	1/22/2021 10:10 AM	File folder	
Languages	1/22/2021 10:10 AM	File folder	
qrc	1/22/2021 10:10 AM	File folder	
schemas	1/22/2021 10:10 AM	File folder	
text	1/22/2021 10:10 AM	File folder	

- Commands
  UI 界面的扩展配置信息，包括数值大小限制，单位设置，几何拾取过滤器等设置，同时也包括控件的显示隐藏逻辑配置等 UI 逻辑，该文件的字段与 ui 文件中的字段一一对应，通过修改该文件，不修改代码即可完成 UI 的定制化。

- Configures
  仿真任务对应 Ribbon 栏中需要展示的图标排列顺序，通过修改该文件，即可以配置界面的 ribbon 展示，无需重新编译链接代码。

- Context
  仿真树节点右键菜单和视图区右键菜单配置，通过修改该文件，即可以配置界面的菜单显示，无需重新编译链接代码。

- Forms
  基于 QT Designer 加 ZW 自研插件绘制出来的 qt 的 ui 文件，该文件可以手动编辑，也可以在 QT Designer 中打开，进行可视化的修改配置。

- Icons
  用于存放 Configures、Context、Forms 这些界面资源相关中需要用到的 icon 资源。

- Languages
  多语言产品化资源文件夹，通过在该文件夹中增加相应的资源翻译语言包，即可以使自己开发的插件显示多个语言版本，形成真正的软件产品。

- qrc
  存储一些自动生成的压缩资源等。

- schemas
  作为 Commands 的补充，用于校验 xml 文件中的元素，如果不合法会在编辑时直接提醒用户。

- text
  在代码中涉及显示文字时，如果该文字需要多语言翻译的话，则需要在该文件中进行补充，后续在产品化过程中就只需要翻译该文件夹即可。

5.2.2. 插件的构建及注册

相关联的头文件在“gui/ZwSImAppRegister.h”中。

- 创建空插件

1. 用 VisualStudio 创建一个 DLL 工程。

确保工程选项 Target Platform Version 为 8.1，Platform Toolset 为 Visual Studio 2015(v140)，Configuration Type 为 Dynamic Library(.dll)。

2. 生成 DLL，导出注册/反注册接口。

首先，从 ZWSim 的安装目录中查找到名为 include 文件夹，并将该文件夹的路径添加到工程的头文件引用目录。接着，实现注册和反注册函数，如下图所示

```
// PluginDemo.h
#include "ZWSim/ZWPackageMgr.h"
#include "ZWSim/ZWModuleMgr.h"

#ifdef __cpluscplus
extern "C" {
#endif
// PluginDemo in PluginDemoInit and PluginDemoExit is the name of your dll, and you should rename them accordingly
__declspec(dllexport) int PluginDemoInit(ZWSOFT::ZWSIM::ZWPackageMgr& package_mgr, ZWSOFT::ZWSIM::ZWModuleMgr& module_mgr);
__declspec(dllexport) int PluginDemoExit(ZWSOFT::ZWSIM::ZWPackageMgr& package_mgr, ZWSOFT::ZWSIM::ZWModuleMgr& module_mgr);
#ifdef __cpluscplus
}
#endif

// PluginDemo.cpp
#include "PluginDemo.h"

int PluginDemoInit(ZWSOFT::ZWSIM::ZWPackageMgr& package_mgr, ZWSOFT::ZWSIM::ZWModuleMgr& module_mgr);
{
    return 0;
}
int PluginDemoExit(ZWSOFT::ZWSIM::ZWPackageMgr& package_mgr, ZWSOFT::ZWSIM::ZWModuleMgr& module_mgr)
{
    return 0;
}
```

PluginDemoInit 是 DLL 的初始化函数，在插件加载前调用；而 PluginDemoExit 为 dll 的退出函数，在插件卸载前调用。在该例子中，两个函数什么都没做，而且返回 0 表示正常。需要注意的是，PluginDemoInit 和 PluginDemoExit 中 PluginDemo 是 dll 的文件名。应该根据 dll 的名字修改这两个导出函数的名字。最后，最后编译工程，确保生成 dll。

3. 加载插件。

ZWSim 的插件以 dll 格式表示，你需要将前面生成的 dll 加载到 ZWSim。首先，打开一个 ZWSim 程序，点击 New 按钮，新建一个文件。这时候就会看到一个新的仿真文件。接着，点击 Ribbon 栏上的 Simulation。在弹出的菜单栏点击 PluginLoader。然后在弹出的对话框上点击 Add。这时将弹出另外的对话框，而你需要选择之前生成的 dll 路径，并点击 OK 关闭该对话框。这时你将在剩余的对话框上看到 dll 的名字，如 PluginDemo。这时你需要勾选 PluginDemo 名字前的勾选框并点击 OK，这时候会弹出一个提示框，选择 Yes。到这时你已经将插件添加到加载列表，但是插件还没加载到 ZWSim。接下来你需要关闭软件：点击右上角的关闭按钮，在弹出的对话框上点击 No。然后重启软件，重复之前的步骤：打开软件，点击 New 按钮，点击 Ribbon 栏上的 Simulation，点击 PluginLoader。这时候如果你看到对话框上显示 dll 的名字，并且名字前的对话框被勾选，那就说明你的你的 dll 已经被成功加载。

- 插件的注册/反注册流程

ZWSIM 仿真平台是 ZW3D 的一个插件，因此加载流程的第一步是将 ZWSIM 加载到 ZW3D 平台。整个加载流程如下：

1. 将 ZWSIM 仿真模块加载到 ZW3D（必须重启生效）
2. ZWSIM 预加载 ZwSimPreLoad.xml 列出的模块
3. ZWSIM 预加载 ZwSimAppPreLoad,xml 列出的插件
4. 用户手动加载/卸载插件（必须重启生效）

软件关闭时，卸载流程如下：

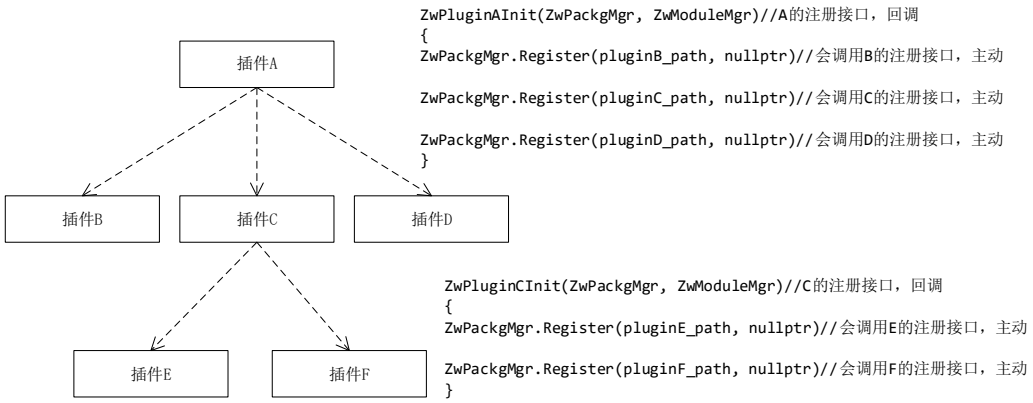
1. 卸载 ZwSimAppPreLoad,xml 列出的插件；
2. 卸载 ZwSimPreLoad.xml 列出的模块
3. 卸载 ZWSIM 仿真插件

4. 继续 ZW3D 关闭流程

需要注意的是，所有插件的加载 /卸载都必须重启后生效。

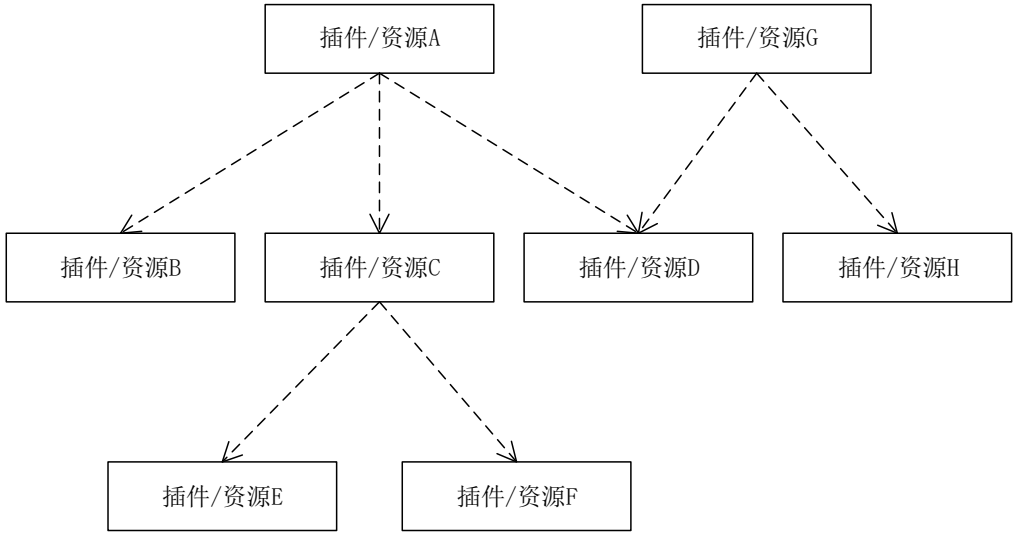
- 插件依赖树

插件加载的方式是 ZWSIM 主动调用插件的注册接口(也就是接口被 ZWSIM 调用)。但是，ZWSIM 同时支持用户主动调用注册接口的方式实现插件加载。使用主动调用的情况一般是因为插件之间存在依赖关系，比如，某个学科插件 B 是基于另一个插件 A 进行开发和扩展。那么当插件 B 被注册进 ZWSIM 时，需要注册插件 A。这时候就需要用户调用主动注册接口。通过主动和多次调用注册接口，插件之间构成了依赖树，如下图。



- 插件引用计数

插件之间构成了依赖树，那么就很容易出现多个插件依赖同一个插件，或者多个资源被多个插件共享的情况。在加载或者卸载的过程中，很容易出现资源冗余或者多次卸载的情况。ZWSIM 采取引用计数的方式：以插件路径或者资源路径为标识符，同一个标识符只会维护一份插件或者资源。当多次加载同一标识符的资源或者路径时，只会增加引用计数；卸载资源或者路径时，则会减少引用计数或者当引用计数为 0 时删除资源（资源删除只能重启再生效）。



5.2.3. 仿真树模块

目前仿真树模块正在迭代升级中，当前的接口参照头文件中的注释（/ZWSim/gui/node），具体使用方法可以参照“6.5.3 src”demo 中代码说明进行阅读了解。

5.2.4. 自定义 UI 模块

当前的接口参照头文件中的注释（/ZWSim/form）,具体使用方法可以参照“6.5.3 src” demo 中代码说明进行阅读了解，

- 注册与反注册(ZwFormBaseReg.hpp)

  提供了两种注册方式，即通过 API 的形式或者通过宏的形式。

  区别是通过 API 的形式需要自行保证注册和反注册的匹配，如果通过宏的形式，则只需要在文件开头声明即可。共同点是都需要这些基本信息：模块 ID，UI 名字，DB type，license，派生类。

  a. API 的形式注册

bool Form::Register(ZwUInt32 moduleId, const char *formName, ZwUint32 datatype, ZwUInt64 license, std::shared_ptr<Base> pForm);

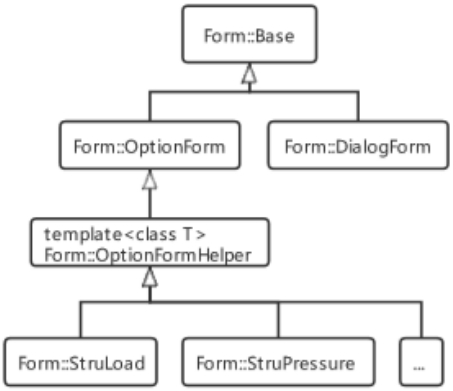
bool Form::Register(const char *formName);

bool Form::UnRegisterAll(ZwUInt32 moduleId);

  b. 宏的形式注册

  ZwFormReg(ModuleId, FormName, DataType, License, T, ...)

- FormBase 说明



```
Base
    // 析构函数+ ~Base();
    // 克隆函数
+ virtual Base* Clone() const = 0;
    // 读取DB，自动触发_DataAutoBindInfo和_DataDeserialize函数
+ bool Read(ZwHandle handle);

    // 数据相关受保护接口
    // 受保护的构造函数，避免用户直接创建Base对象
# Base();
    // 获取当前数据版本
# virtual ZwUint16 _DataCurVersion() = 0;
    // 保存DB函数，受保护，确保用户知会在类内部使用该函数
# bool _DataSave(ZwHandle *pHandle)
    // 自动绑定数据，用于读DB，写DB，注册DB
# virtual void _DataAutoBindInfo(DbBinder &binder, const ZwUint16 version);
    // DB注册的补充
#  virtual bool _DataRegister(ZwRegisterHelper &helper, const ZwUint16 version);
    // DB写的补充
#  virtual bool _DataSerialize(DbHelper &dbHelper, const ZwUint16 version);
    // DB读的补充
#  virtual bool _DataDeserialize(DbHelper &dbHelper, const ZwUint16 version);

    // UI相关受保护接口
    // 获取ui操作辅助类
#  GuiHelper _GetGuiHelper();
    // 初始化
#  virtual bool _OnFormInit();
    // ok
#  virtual bool _OnFormOkay();
    // term
#  virtual bool _OnFormCancel();
    // callback
#  virtual bool _OnFormCallBack(int field, int item);

- Base &operator=(const Base &) = delete;
```

```
OptionForm
  // 预览函数# virtual bool _OnFormEcho(ZwHandle
hEcho);
```

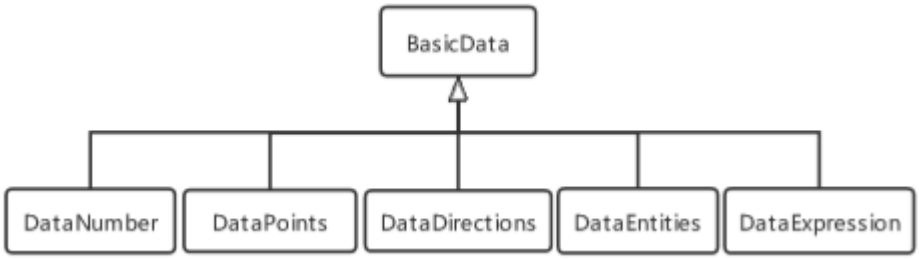
```
OptionHelper
  // 默认clone函数+ Base *Clone() const override
  {
        return new T(static_cast<const T&>(*this));
  }
```

```
Dialog Form
// TODO
```

- 基础数据辅助类说明

  本模块对于一些常见的数据类型分别作了简单的封装，使在外部使用起来更加方便。目前主要包装了以下类型：

  a. Number
  b. Directions，可能包括一个或多个方向
  c. Points， 可能包括一个或多个点
  d. Entities， 可能包括一个或多个几何
  e. Expression，对函数的功能做了扩展，多了一个 enable 状态



DataNumber

```
+ DataNumber();
+ ~DataNumber();
// 设置自定义数据
+ void SetData(double value, const ZwString &unit = ZwString())
// 单位转换，长度以mm为基准，其他单位以SI为基准
+ int toInt() const;
+ double toDouble() const;
+ ZwString toStr(const ZwString strSeperator = "", bool needUnit = true) const;
// 单位转换，用户指定单位系统（SI，MKS，CGS，MMKS，IPS，FPS）
+ int toInt(Unit::System system) const;
+ double toDouble(Unit::System system) const;
+ ZwString toStr(Unit::System system, const ZwString strSeperator = "", bool needUnit = true) const;
// 单位转换，根据当前单位系统的设置
+ int toIntInCurrentSetting() const;
+ double toDoubleInCurrentSettingI() const;
+ ZwString toStrInCurrentSetting(const ZwString strSeperator = "", bool needUnit = true) const;
// 获取原始数据，包括数值和单位
+ bool GetData(double &value, ZwString &unit) const;
```

DataPoints

```
+ DataPoints()
+ ~DataPoints()
// 设置自定义数据
+ void SetData(const ZwPoint3&point)
+ void SetData(const ZwVector<ZwPoint3> &points)
// 设置是否关联几何属性
+ void SetNeedAssociateGeom(bool associated)
// 获取点个数
+ ZwUInt32 GetSize() const
+ bool Empty() const
// 获取点的坐标，以mm为单位
+ ZwPoint3 toPoint(ZwUInt32 pos = 0) const
+ ZwVector<ZwPoint3> GetAllPoints() const
// 获取点的坐标，指定单位系统
+ ZwPoint3 toPoint(Unit::System system, ZwUInt32 pos = 0) const
+ ZwVector<ZwPoint3> GetAllPoints(Unit::System system) const
// 获取点的坐标，根据当前单位系统的设置
+ ZwPoint3 toPointInCurrentSetting(ZwUInt32 pos = 0);
+ ZwVector<ZwPoint3> toPointInCurrentSetting(ZwUInt32 pos = 0);
```

DataDirections

```
+ DataDirection()
+ ~DataDirection()
// 设置自定义数据
+ void SetData(const ZwPoint3 &dir)
+ void SetData(const ZwVector<ZwPoint3 &dirs>)
// 设置是否关联几何属性
+ void SetNeedAssociateGeom(bool associated)
// 获取方向个数
+ ZwUInt32 GetSize() const
+ bool Empty() const
// 获取方向
+ ZwPoint3 GetNormalizeDir(ZwUInt32 pos = 0) const;
+ ZwVector<ZwPoint3> GetAllNormalizeDir() const;
```

DataEntities

```
+ DataEntities()
+ ~DataEntities()
// 获取几何类型
+ Geom::EntType GetEntType(ZwUInt32 pos = 0)
// 获取几何个数
+ ZwUInt32 GetSize() const
+ bool Empty() const
// 获取pick path
+ Geom::PickPath GetGeomPickPath(ZwUInt32 pos = 0) const
+ ZwVector<Geom::PickPath> GetAllGeomPickPaths() const
// 高亮几何
+ void PickDisp(bool hi)
+ void PickDispColor(bool hi, ZwColor)
```

DataExpression

```
+ DataExpression()
+ ~DataExpression()
// 获取函数是否enable
+ bool IsEnable() const
// 获取函数类型，Foumla or Table
+ Expression::ExprType GetType() const
// 获取函数名字
+ ZwString GetName() const
```

- 几何新增对外接口

Geom::PickPath

```
+ PickPath()
+ PickPath(ZwHandle handle)
+ ~PickPath()
+ PickPath(const PickPath &pickPath)
+ PickPath& operator=(const PickPath &pickPath)
+ operator void*()
// 检查pickpath是否合法
+ bool IsValid() const
// 检查是否被抑制
+ bool IsUseable() const
// 获取几何类型
+ EntType GetEntType() const
// 获取曲线类型
+ bool IsCurve() const
+ CurveType GetCurveType() const
// 获取曲面类型
+ bool IsSurface() const
+ SurfaceType GetSurfaceType() const
// 获取参考面类型
+ bool IsDatum() const
+ DatumType GetDatumType() const
// 高亮显示
+ void PickDisp(bool hi) const
+ void PickDispColor(bool hi, ZwColor color) const
// 获取pickpath的handle
+ ZwHandle GetHandle() const
// 获取永久化的pickpath的handle
+ ZwHandle Persist() const
// 获取该pickpath是否装配体
+ bool IsAssemble() const
// 获取几何信息，包括原始handle和所属文件名
+ bool GetEntityInfo(ZwHandle &entity, ZwString &fileName = ZwString()) const
```

几何类型 Geom::EntType

```
Invalid
Curve
Sketch
Edge
Face
Shell
Component
Datum
```

曲线类型 Geom::CurveType

```
Invalid
Line
Arc
Circle
General
```

曲面类型 Geom::SurfaceType

```
Invalid
Plane
Sphere
Cone
Cylinder
Ellipsoid
Torus
General
```

参考面类型 Geom::DatumType

```
Invalid
Axis
Plane
Csys
```

- DB 辅助类 DbBinder 及 DbHelper 说明

```
DbBinder
+ DbBinder()
+ ~DbBinder()
+ operate void*();

// 绑定函数，用于注册、写、读流程
// T包括ZwInt8, ZwInt16, ZwInt32, ZwInt64, ZwUInt8, ZwUInt16, ZwUInt32
// ZwUInt64, float, double, bool, ZwHandle, ZwString, DataNumber,
// DataPoints, DataDirections, DataEntities, DataExpression+ bool
Bind(ZwUInt32 dataField, T& value)

// 绑定vector函数，用于注册、写、读流程
// T包括ZwInt8, ZwInt16, ZwInt32, ZwInt64, ZwUInt8, ZwUInt16, ZwUInt32
// ZwUInt64, float, double, bool, ZwHandle, ZwString, DataNumber,
//  DataPoints, DataDirections, DataEntities, DataExpression
+ bool Bind(ZwUInt32 dataField, ZwVector<T> &value)
```

```
DbHelper
+ DbHelper(ZwUInt32 module, ZwUInt32 type, ZwUInt16 version)
+ DbHelper(ZwHandle handle)
+ ~DbHelper()
+ const ZwHelper& GetHelper() const
+ ZwHelper& GetHelper() const
+ bool WriteName(const ZwString &name)
+ bool ReadName(ZwString &name) const

// T包括ZwInt8, ZwInt16, ZwInt32, ZwInt64, ZwUInt8, ZwUInt16, ZwUInt32
// ZwUInt64, float, double, bool, ZwHandle, ZwString, BasicData
+ bool Read(ZwUInt32 dataField， T &value)

// T包括ZwInt8, ZwInt16, ZwInt32, ZwInt64, ZwUInt8, ZwUInt16, ZwUInt32
// ZwUInt64, float, double, bool, ZwHandle, ZwString, BasicData
+ bool Write(ZwUInt32 dataField， T value)

// T包括ZwInt8, ZwInt16, ZwInt32, ZwInt64, ZwUInt8, ZwUInt16, ZwUInt32
// ZwUInt64, float, double, bool, ZwHandle, ZwString, DataNumber,
// DataPoints, DataDirections, DataEntities, DataExpression
+ bool Read(ZwUInt32 dataField， ZwVector<T> &value)

// T包括ZwInt8, ZwInt16, ZwInt32, ZwInt64, ZwUInt8, ZwUInt16, ZwUInt32
// ZwUInt64, float, double, bool, ZwHandle, ZwString, DataNumber,
// DataPoints, DataDirections, DataEntities, DataExpression
+ bool Write(ZwUInt32 dataField， ZwVector<T> value)
```

- UI 辅助类 GuiHelper 说明及函数 UI 辅助类

```
GuiHelper
+ GuiHelper()
+ GuiHelper(GuiHelper &&helper)
+ ~GuiHelper()
+ operator void*()+ ZwString GetFormName() const
+ bool SetVisible(ZwInt32 uiField, bool visible) const
+ bool SetFocus(ZwInt32 uiField) const
+ bool SetEnable(ZwInt32 uiField, ZwInt32 uiItemId, bool enable) const
+ bool ClearData(ZwInt32 uiField) const

// T包括ZwInt8, ZwInt16, ZwInt32, ZwInt64, ZwUInt8, ZwUInt16, ZwUInt32
// ZwUInt64, float, double, bool, ZwHandle, ZwString, BasicData
+ bool Read(ZwInt32 uiField, T&value) const

// T包括ZwInt8, ZwInt16, ZwInt32, ZwInt64, ZwUInt8, ZwUInt16, ZwUInt32
// ZwUInt64, float, double, bool, ZwHandle, ZwString, DataNumber,
//
DataPoints, DataDirections
+ bool Write(ZwInt32 uiField, T value) const
```

```
WidgetFunction
+ bool IsEnable(const char *form, ZwUInt32 uiField)
+ bool SetEnable(const char *form, ZwUInt32 uiField, bool enable)
+ ZwString GetLabel(const char *form, ZwUInt32 uiField)
+ ZwString SetLabel(const char *form, ZwUInt32 uiField, const char *label)
+ static bool InitComboBox(const char *form, ZwUInt32 uiField,
ZwVector<ZwInt32> &independent, ZwVector<ZwUInt32> &dependent)
```

5.2.5. 网格模块

目前网格模块正在迭代升级中，当前的接口参照头文件中的注释（/ZWSim/mesh/）,具体使用方法可以参照“6.5.3 src”demo 中代码说明进行阅读了解。

5.2.6. 显示模块

当前的接口参照头文件中的注释（/ZWSim/ graphic/）。

- 2D 显示接口

通用：

构造函数：

```
ZwChartWidget(ZwChartType type, unsigned __int64 parentQtWinId = 0);
enum ZwChartType
{
        eTable,
        eCartesian,
        eBar,
        e2D,
        ePolar,
        eSmith
};
```

独立QWidget，或嵌入到parentQtWinId上

通用：

```
//显示独立QWidget
void Show();

//设置图表名称
bool SetChartTitle;

//设置大小
bool SetChartFixedSize(int width, int height);
bool SetChartMinimumSize(int width, int height);
bool SetChartMaximumSize(int width, int height);

//清空数据
bool ClearData();
```

表格：

```
SetColumnNames(); SetColumnName();GetColumnName();

SetHeaderResizeMode();
enum TableHeaderResizeMode
{
        eInteractive,          //用户可拖动，可在代码中调用函数设置
        eStretch,              //自动填满所有空余空间，用户不可拖动，可在通过代码设置
        eFixed,                //固定宽度，用户不可拖动，代码不可设置
        eResizeToContents      //根据列名的文字自动缩放，用户不可拖动，代码不可设置
};
```

表格：

```
AppendRow(ZwVector<double> &rowData);   //尾部追加一行数据
InsertRow(int row, ZwVector<double> &rowData);     //在第row行插入
SetRow(int row, ZwVector<double> &rowData);        //设置某一行
RemoveRow(int row, );   //删除某一行
GetRowData(int row, ZwVector<double> &rowData);   //获取某一行的数据

AppendColumn(); InsertColumn(); SetColumn(); RemoveColumn(); GetColumnData();

SetItemData(); GetItemData();  //设置、获取某一个单元格的数据

ClearTable();  //清除控件上的所有行列
```

笛卡尔图：

```
AddLine(ZwVector<ZwVector<double>> &pointSeries, ZwString lineName);
RemoveLine(ZwString lineName);
SetLineColor(ZwString lineName);
SetXAxisTitle(ZwString title); SetYAxisTitle(ZwString title);  //需要在已经使用AddLine添加数据
后调用
```

柱状图：

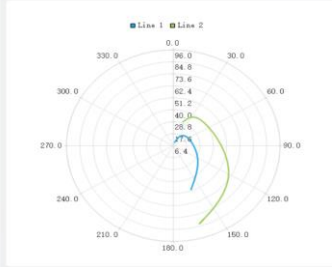
```
AddBar(ZwVector<double> &barSetValues,
        ZwString barName);
SetBarCategories(ZwVector<ZwString> categories);
SetBarColor(ZwString barName);
```

2D图：

```
Set2dData(size_t xDataCount, size_t yDataCount, ZwVector<double> &zData);

zData:
4, 19, 5, 3, 48, 24, 1, 2, 33, 23, 50, 22

xDataCount = 4; yDataCount = 3;
zData:
4, 19, 5, 3,
48, 24, 1, 2,
33, 23, 50, 22
```

极坐标图：

```
AddPolarLine(
ZwVector<ZwVector<double>>,
ZwString lineName);
SetPolarLineColor();
SetPolarAxis(double minAngular,
        double maxAngular,
        double minRadial,
        double maxRadial,
        int angularTickCount,
        int radialTickTount);

tickCount首尾相接
```



- 3D 显示接口

场景对象：一个方便的显示对象注册类，用户需要继承，组合自己的演员和算法

演员对象：在场景内由用户维护，控制显示属性

算法对象：在场景内由用户维护，控制显示数据

显示管理器：单例，提供场景管理接口，以及其他全局显示接口

ZwGui/Disp 中已实现场景：封装好了演员和算法，提供业务接口

场景对象ZwScene

- 作为用户场景基类，用户继承它实现自己的绘制场景。
- 构造函数：生成若干ZwActor和ZwAlgorithm
  - 每一个实例用户必须提供编号，作为场景标识
  - 用户必须在构造函数中完成可视化管线的搭建
  - source1->filter1->filter2->actor1
  - filter1->SetInput1(source1);
  - filter1->SetInput2(source2);
  - filter2->SetInput(filter1);
  - actor1->AddSource(filter2);
- 响应函数：
  - OnModify...渲染时会自动调用一次，也可通过管理器获取到场景手动调用
  - OnModify...约定：OnSource中填充source数据，OnActor中修改显示属性，……
  - OnGetActor：返回用户所维护的演员
- 通过ZwDispMgr提供的接口把ZwScene注册到显示系统
- ZwSimpleScene：简单场景，内部维护有一个演员，并提供获取接口

virtual ZwVector< **ZwActor** * >	**OnGetActor** () const =0 offer the actors user want to render 更多...
virtual **void**	**OnModifySource** ()=0 change the source data 更多...
virtual **void**	**OnModifyActor** () change the actor attribute 更多...
virtual **void**	**OnModefyFilter** () change the filter attribute 更多...
virtual **void**	**OnModefyCustom** () change the custom attribute 更多...

演员对象ZwActor

- 管理显示属性，渲染过程中会把显示属性设置到渲染管线
- 具体属性见基类头文件ZwActor.h

- 通过AddSource接口关联ZwAlgorithm。
- 内部维护一个ZwMapper，负责把algorithm处理完的数据转化成可渲染的数据

- ZwDispLegend: Legend演员，提供legend相关接口，见头文件
- ZwFrameActor: 坐标系演员，提供相关接口，见头文件

显示管理器ZwDispMgr

- 提供用户操作接口。
- 通过Register注册/卸载用户场景
- 通过RegisterHelper注册/卸载用户的ZwRenderHelp

- 提供场景获取接口（指定标识获取，或者，获取当前显示场景）
- show/hide接口

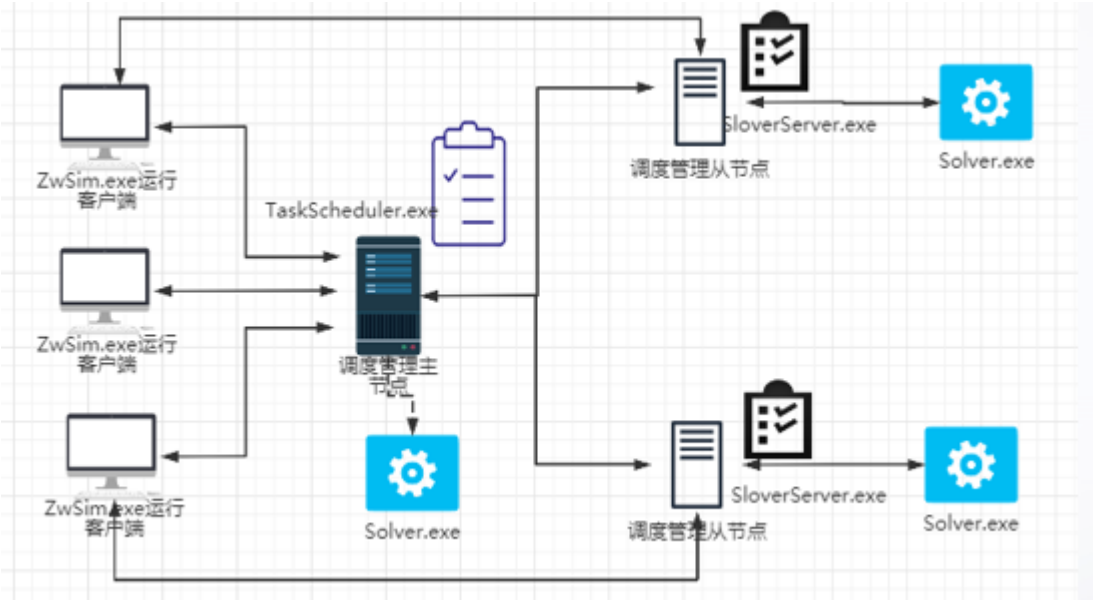
.....见头文件ZwDispMgr.h

GUI/disp中已实现场景

- 理论上来说，所提供出来的场景，用户只需要调用所提供接口就好，不需要了解actor、source、filter相关概念

5.2.7. 计算模块

ZWMeshWorks.exe 和求解器之间是通过 TCP/IP 进行通信的，在整个架构中的逻辑如下：



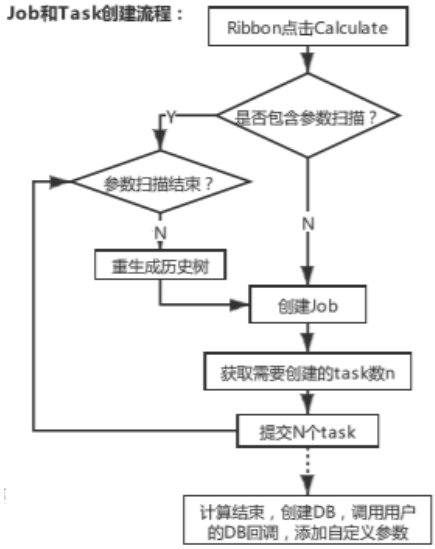
对于二次开发者，不用去关心具体调度内部的行为，而只需要在在计算模块中填写相应的业务逻辑即可。关于计算模块相关的头文件可以参照（/ZWSim/ calculate）。

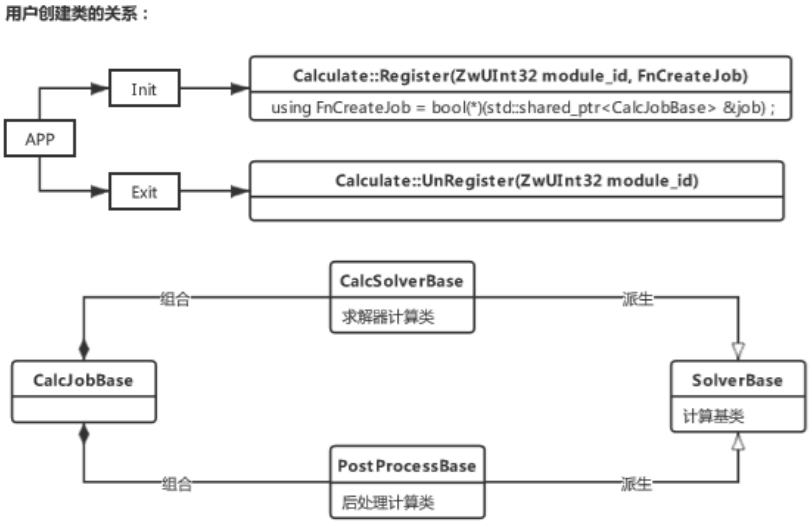
以下是关于计算模块的具体说明：

两个概念（Task 和 Job）：

Task：一个完整的 Task 包括一次求解器计算流程和一次后处理计算流程。对应接口类为一个 CalcSolverBase 对象和一个 PostProcessBase 对象。

Job：一个 Job 默认包含一个 Task，但对应于 EM 的逐次激励和 ST 的多工况场景，一个 Job 可以包含多个 Task。非参数扫描场景下，在 Ribbon 栏点击一次计算，相当于创建一个 Job。对应接口类为 CalcJobBase 类。





接口说明：

```
CalcJobBase
  // 在提交计算任务给调度管理之前的回调+ DoBeforeCommit(ZwHandle hTask)
  // 在提交计算任务给调度管理之后的回调，此处可以获得本次job的唯一id
+ DoAfterCommit(ZwHandle hTask, ZwUInt64 uniqueJobId)
  // 获取需要启动的task数，默认为1.（逐次激励场景，多工况场景，建议通过返回这个id来实现）
+ GetCalcCnt(ZwHandle hTask)
  // 纯虚接口。用于回调创建CalcSolverBase和PostProcessBase两个类。如果会创建多task，可以通过cnt参数来唯一识别。
+ CreateSolver(shared_ptr<CalcSolverBase> &calcSolver, shared_ptr<PostProcessBase> &postProcSolver, ZwUInt32 cnt) = 0

  // 整个job（所有task）计算完成的回调
+ OnComplete()
  // 计算主动终止的回调
+ OnAbort(bool support)
  // 计算异常的回调
+ OnException()
  // 计算进度更新的回调
+ OnProgress(int progress)
  // 计算暂停的回调
+ OnPause(bool support)
  // 计算恢复的回调
+ OnResume(bool support)
  // 计算刷新的回调
+ OnFresh(const vector<string> &files)

  // 可以用于返回报错信息，方便定位问题
# _SetErrorInfo(const ZwString &strInfo)

  // 内部数据
- shared_ptr<CalcJobBaseImpl> m_impl
```

```
SolverBase
  // 纯虚函数，返回求解器的唯一标识（目前任务调度内部是通过字符串来匹配的）
+ GetSolverSign():ZwString = 0
  // 在添加task之前的回调，cnt代表是第几个task+ DoBeforeCommit(ZwHandle
hTask, ZwUInt32 cnt)
  // 在添加task之后的回调，cnt代表是第几个task
+ DoAfterCommit(ZwHandle hTask, ZwUInt64 uniqueJobId)

  // 在solver被启动后的回调事件
+ OnLaunch()
  // 本solver计算完成的回调
+ OnComplete()
  // 本solver主动终止的回调
+ OnAbort(bool support)
  // 本solver计算异常的回调
+ OnException()
  // 本solver计算进度更新的回调
+ OnProgress(int progress)
  // 本solver计算暂停的回调
+ OnPause(bool support)
  // 本solver计算恢复的回调
+ OnResume(bool support)
  // 本solver计算刷新的回调
+ OnFresh(const vector<string> &files)
  // 本solver用户收到自定义数据的回调
+ OnReceiveUserData(void *pData, ZwUInt32 size)

  // 可以用于返回报错信息，方便定位问题
# _SetErrorInfo(const ZwString &strInfo)
  // 本solver主动通信的接口
# _PostUserData(void *pData, ZwUInt32 size)
  // 获取本solver在本地输入输出文件的路径（UUID）
# _GetCurrentPath():ZwString

  // 内部数据
- shared_ptr<SolverBaseImpl> m_impl
```

```
CalcSolverBase : public SolverBase
  // 纯虚函数，返回本solver的启动参数，一般在本函数中去生成zdf文件。
+ GetCommandArguments(ZwHandle hTask, CmdArguments &args) = 0
```

```
PostProcessBase : public SolverBase
  // 枚举，以函数的方式进行后处理计算
+ enum PostProcType::T_Function
  // 枚举，以exe的形式进行后处理计算
+ enum PostProcType::T_Process

  // 纯虚函数，决定以什么方式进行后处理计算，如果不需要后处理，则返回T_Function即可
+ GetType():PostProcType = 0
  // 函数计算后处理时的回调，可以不实现
+ PostProcess(const ZwString &currentPath, FnProgress fn, ZwString &outputZdfPath)
  // 以exe形式启动后处理时的回调
+ GetCommandArguments(const ZwString &pathIn, CmdArguments &args, ZwString &outputZdfPath)

  // 如果后处理计算后，能正常产生合法的zdf文件，将自动触发下面的回调，进行DB的创建
  // 创建结果根节点DB后的回调，可以通过结果DB接口自定义扩展字段
+ DBCallBackCreateResult(Result::ZwDbResult &result)
  // 用户决定是否创建结果曲线，返回true则自动创建结果曲线节点DB，用户可以给指定的nodes匹配相应的Int字段。
+ DBCheckElementNeedCreate(const ZwVector<ZwString> &nodes, ZwVector<ZwUint32> &types)
  // 创建结果曲线节点DB后的回调，可以通过结果DB接口自定义扩展字段
+ DBCallBackCreateResultElement(Result::ZwResultElement &element)
```

```
CmdArguments //命令参数辅助类
  // 获取当前任务的磁盘路径
+ GetPath()
  // 清空所有参数
+ Clear()
  // 添加普通启动参数
+ AppendStr(const ZwString &arg)
  // 添加输入文件启动参数，该文件必须存在。路径可以是全路径也可以是相对路径，内部会自动适配。
+ AppendInputFile(const ZwString &fileIn)
  // 添加输出文件位置。路径可以是全路径也可以是相对路径，内部会自动适配。
+ AppendOutputFile(const ZwString &fileOut)
  // 输出适配过的最终参数，可以用于调试
+ OutputCmdArguments():ZwString

  // 内部数据
-  shared_ptr<CmdArgumentsImpl> m_impl
```

5.2.8. ZDF 模块

目前 ZDF 模块正在迭代升级中，当前的接口参照头文件中的注释（/ZWSim/ dbOperator/）,具体使用方法可以参照“6.5.3 src”demo 中代码说明进行阅读了解。

5.2.9. 其他工具模块

工具模块的所有接口说明将在后续版本的迭代中进行说明。

- Check 定制

  当前的接口参照头文件中的注释（/ZWSim/ check/）

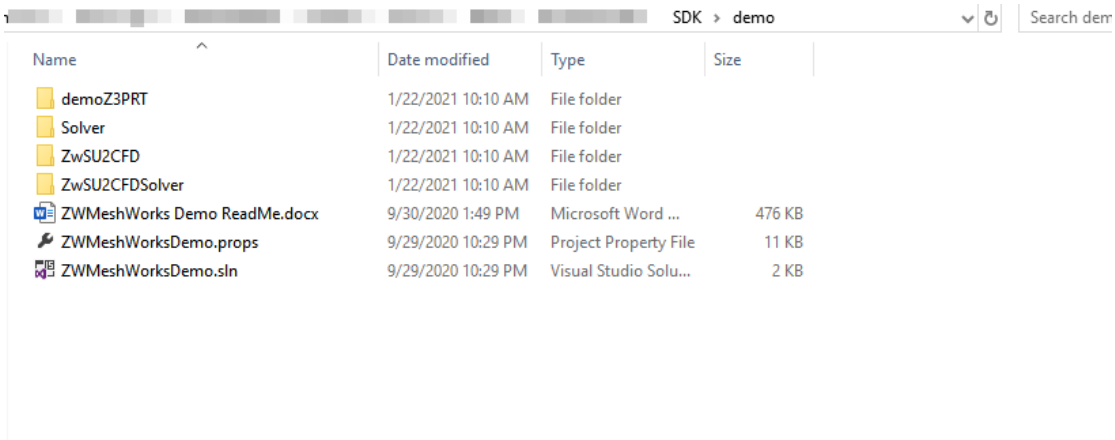
- 结果报告定制
  当前的接口参照头文件中的注释（/ZWSim/ utils/simlulationReport）

- 关系型阵列定制
  当前的接口参照头文件中的注释（/ZWSim/ utils/pattern）

第III部分 集成求解器 Demo 展示

第6章 集成求解器开发实例

在 ZWMeshWorks 的安装目录下，存放着一个 demo，用于展示集成开源求解器 SU2_CFD，基于平台开发的插件的代码。



6.1. Demo 功能说明（SU2_CFD）

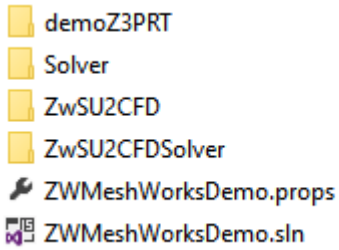
SU2 是美国斯坦福大学航空航天学院开发的高精度偏微分方程求解器，它是一款非结构网格求解器，并且是基于 C++开发的。它不仅开展串行计算，还可以进行并行计算，同时具备了变形网格功能和网格自适应功能等。

在本 demo 中集成的是 SU2 的流体计算模块，同时提供了两个配置好的案例用于用户使用调试，更多的案例参考见 SU2 官网说明。

在示例代码中，由于是用于展示的 demo，有一些流体设置的业务模块写得比较简陋，感兴趣的用户可以自行将代码进行完善。

6.2. Demo 目录

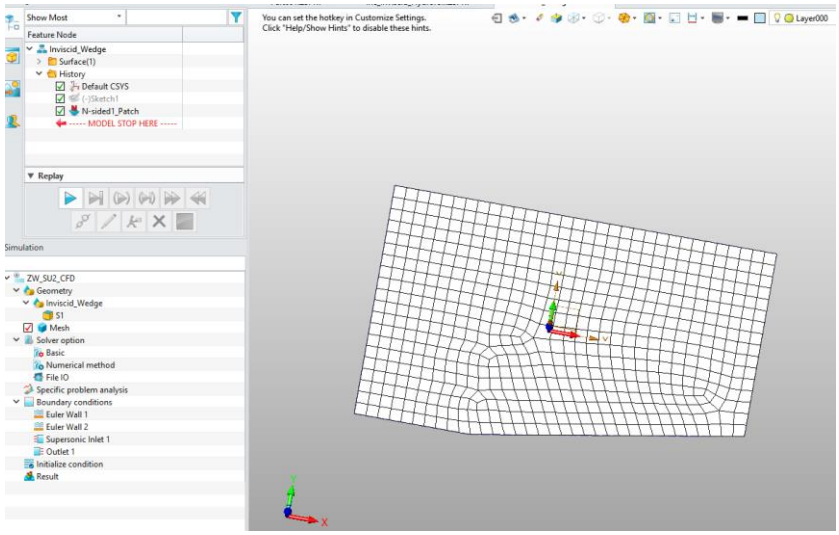
demo 文件夹下是基于“Inc”和“Libx64”编译的一个集成 SU2_CFD 的例子，具体如下：



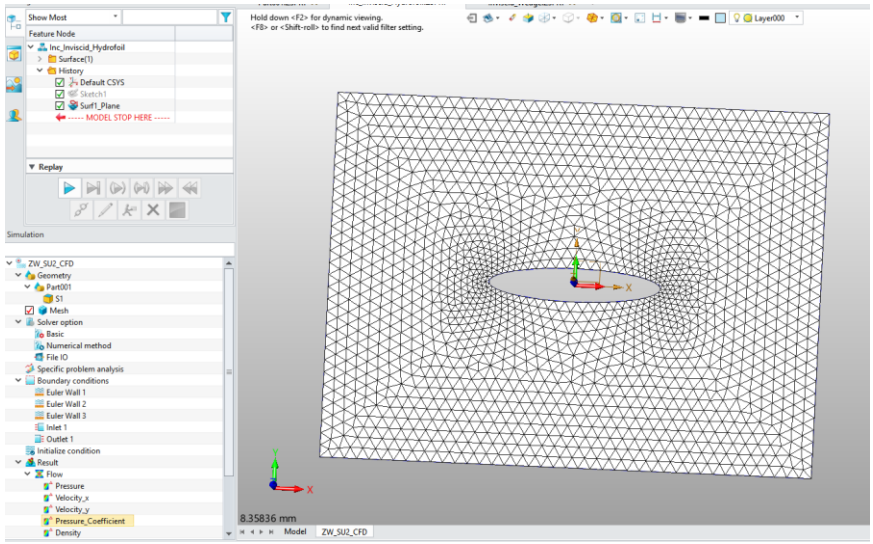
- demoZ3PRT

  里面是基于“ZWMeshWorksDemo.sln”编译出来的 dll，加载到 ZWMeshWorks 平台后创建的模型文件，对应的是 SU2_CFD 官方提供的两个 demo：

  / compressible_flow/ Inviscid_Wedge/ 是一个四边形网格的可压计算模型；



  / incompressible_flow / Inc_Inviscid_Hydrofoil / 是一个三角形网格的非可压计算模型。



- Solver

  SU2_CFD 官方提供的开源求解器。

- ZwSU2CFD

  用于加载到 ZWMeshWorks 平台的 dll 工程。

- ZwSU2CFDSolver
  对 SU2_CFD 求解器进行的简单包装，使其带有 TCP/IP 通信工程。

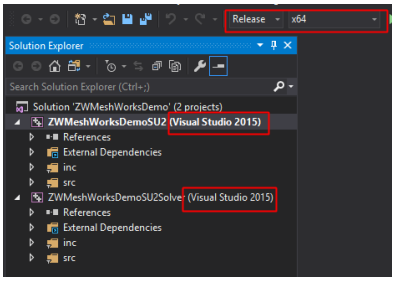
- ZWMeshWorksDemo.props
  ZWMeshWorks 提供的属性表，用户的工程只需要导入该属性表，则无需去配置编译链接信息。

- ZWMeshWorksDemo.sln
  demo 的解决方案文件。

6.3. Demo 编译步骤

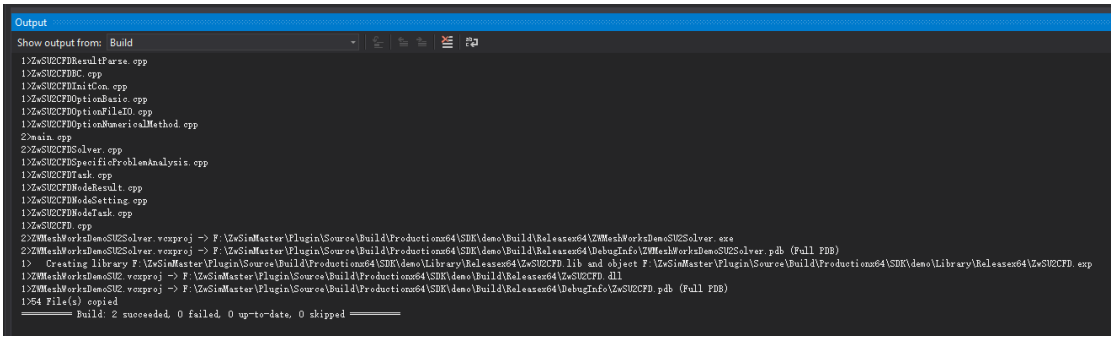
- 双击打开“ZWMeshWorksDemo.sln”

- 确保编译配置正确

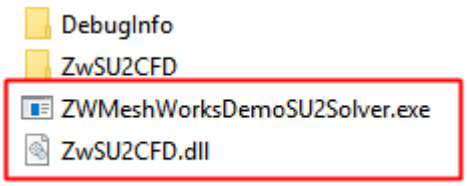


  a. 确保是 Release x64 版本，目前 ZWMeshWorks 平台只提供了 release 版本的 SDK
  b. 确保工程配置是 Visual Studio 2015

- 点击编译，即可以生成对应的 dll 和 exe，如下图。

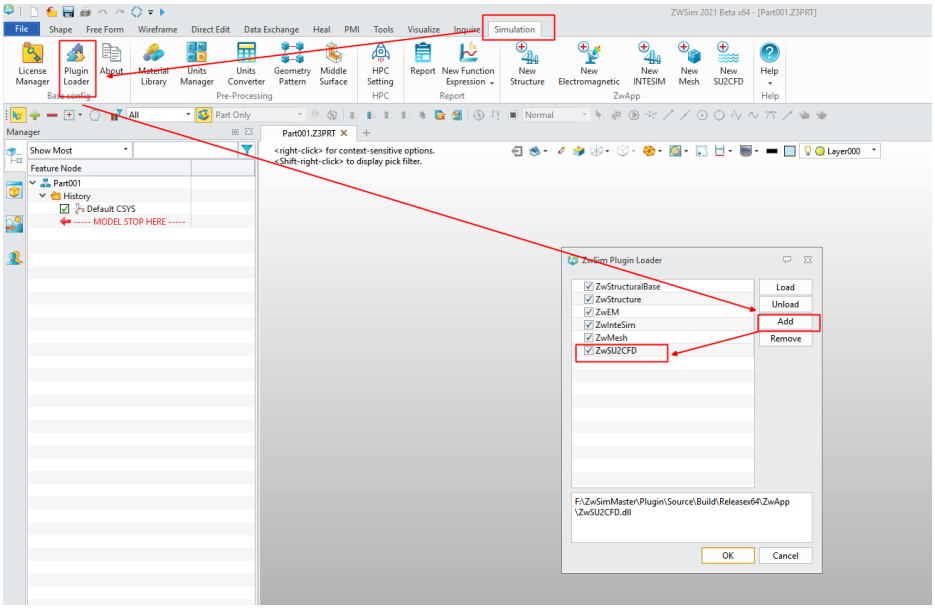


- SDK/demo/Build/Releasex64 下可以看到如下文件：



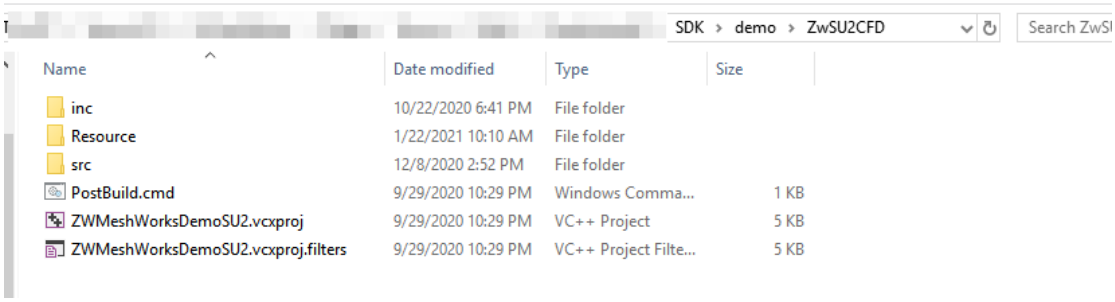
6.4. 使用 ZWMeshWorks 加载自研插件

点击 Plugin Loader，弹出对应的对话框，点击 Add，添加需要加载的插件，然后点击 OK。这时候 ZWSim 就会记录插件的加载信息。接下来重启软件后就会看到插件已被加载，同时点开 Plugin Loader 也会看到新加的插件。



6.5. Demo 源码说明

在 ZWMeshWorks 的安装目录下的/SDK/demo/ZwSU2CFD/目录下可以看到该插件的所有源码。



6.5.1. inc

Name	Date modified	Type	Size
ZwFile.h	9/29/2020 10:29 PM	C/C++ Header	8 KB
ZwPath.h	9/29/2020 10:29 PM	C/C++ Header	13 KB
ZwSU2CFD.h	9/29/2020 10:29 PM	C/C++ Header	1 KB
ZwSU2CFDPr.h	10/21/2020 2:09 PM	C/C++ Header	1 KB
ZwSU2CFDType.h	10/22/2020 6:41 PM	C/C++ Header	7 KB
ZwTimestamp.h	9/29/2020 10:29 PM	C/C++ Header	6 KB

头文件，包括在 src 中需要用到头文件和一些公共的头文件。

6.5.2. Resource

Name	Date modified	Type	Size
Commands	1/22/2021 10:10 AM	File folder	
Configures	1/22/2021 10:10 AM	File folder	
Context	1/22/2021 10:10 AM	File folder	
Forms	1/22/2021 10:10 AM	File folder	
Icons	1/22/2021 10:10 AM	File folder	
Languages	1/22/2021 10:10 AM	File folder	
qrc	1/22/2021 10:10 AM	File folder	
schemas	1/22/2021 10:10 AM	File folder	
text	1/22/2021 10:10 AM	File folder	

具体文件夹下各文件的含义，参照“5.2.1. 资源文件路径规范说明”。

6.5.3. src

Name	Date modified	Type	Size
calc	1/22/2021 8:51 AM	File folder	
cmd	1/22/2021 8:51 AM	File folder	
node	1/22/2021 8:51 AM	File folder	
ZwSU2CFD.cpp	12/8/2020 2:52 PM	C++ Source	3 KB

- calc
  - ZwSU2CFDCalcJob
    ZWMeshworks 进程通过调度模块和 ZwSU2CFD 求解器模块进行交互的地方，包括启动参数配置，求解文件生成，求解结果解析等过程的控制。

  - ZwSU2CFDGenCfg
    用于生成 SU2_CFD 求解器计算案例时所需要的文件，相当于将 ZW 的数据翻译成求解器输入文件的过程。

- ZwSU2CFDResultParse
  用于解析 SU2_CFD 算出来的结果文件，并将其进行管理，以供后续显示渲染模块读取对应数据字段并进行显示的过程。

- cmd
  - ZwSU2CFDTask
    这是用户界面操作这个动态库的功能入口，新建一个 SU2_CFD 的仿真任务。

  - ZwSU2CFDBC
    边界条件的 UI 代码逻辑，通过该文件可以了解到 ZWMeshWorks 平台的 UI 逻辑和数据存储相关的代码逻辑。

  - ZwSU2CFDInitCon
    初始条件设置 UI.

  - ZwSU2CFDOptionBasic
    SU2_CFD 求解器的基本属性 UI 设置。

  - ZwSU2CFDOptionFileIO
    SU2_CFD 求解器的文件 IO 属性 UI 设置。

  - ZwSU2CFDOptionNumericalMethod
    SU2_CFD 的数值方法求解属性 UI 设置。

  - ZwSU2CFDSpecificProblemAnalysis
    SU2_CFD 的特殊求解属性 UI 设置。

- node
  - ZwSU2CFDNodeTask
    SU2_CFD 仿真树创建的入口，通过该文件可以去自由构造求解器独有仿真树的层级关系，以及对各种用户触发的 UI 事件的处理。

  - ZwSU2CFDNodeSetting
    SU2_CFD 仿真树业务逻辑节点的逻辑，包括边界条件节点、基本属性设置、求解 IO 属性设置、求解数值方法属性设置、求解特殊属性设置等。

  - ZwSU2CFDNodeResult
    SU2_CFD 仿真树结果数据节点的逻辑。

- ZwSU2CFD.cpp
  这是整个模块的入口，其中包含各种回调事件的处理，插件的注册和反注册处理，以及插件的初始化信息配置。

6.5.4. PostBuild.cmd

编译链接成功后自动调用的批处理文件，用于复制所需要的资源文件形成可发布的文件夹，其中包括新生成的 ZwSU2CFD.dll 及其资源，以及 SU2_CFD 求解器。

6.6. 常见问题解决方法

6.6.1. 编译失败

确认是否因为头文件路径搜索不到，如果是可以自行修改工程的配置，或者手动打开“ZWMeshWorksDemo.props”文件，修改 ZwMeshWorksSDKDir 即可。

6.6.2. 计算后求解器无法使用

确保 ZWMeshWorks 平台下的用户目录下的“solverexe.xml”，是否有对应的求解器路径信息，注意该求解器是指用户编译出来带有通信功能的求解器。