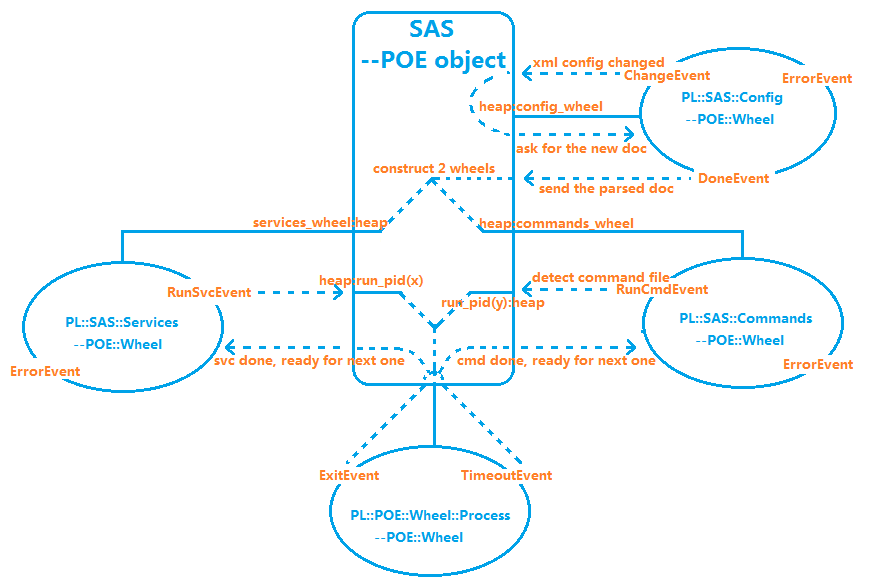
# 框架图



## 简介

SAS（SNS Automation Server），该工具的目的是因为SNS Server上，经常会有很多工具需要按时、按需运行，例如我们希望周期监控DUT是否需要断电，希望按照用户需求自动同步最新case等。而该工具就是用来按时、按需来运行各种第三方工具的，不仅如此，还将这些第三方工具统一管理起来，共享同一份配置文件。

SAS基于POE实现——一个是基于异步事件驱动的框架，因此该工具的实现，也是通过各种事件来驱动的。SAS重点使用了POE::Wheel——这是一个功能类似“轮子”的模块，所谓“轮子”只是形容其不断转动的行为，本质就是在一个封闭的空间中，Wheel自身维护一套独立的事件状态机，一旦触发用户指定的事件，就会向外部世界emit一个event，不仅如此外部世界还可以put某些东东发送给Wheel，以此来控制Wheel的一些行为。

SAS主要是实现了4个Wheel：

PL::SAS::Config, 用于监控配置文件是否发生修改，一旦修改，向外发送ChangeEvent，同时外部世界也可以向Wheel发送文档对象，此时Wheel内部会解析该文档对象，并将解析后的配置项作为DoneEvent的参数发送出去。

PL::SAS::Services, 用于周期轮询是否有某个Service需要执行，一旦Serivce到时间需要执行，则向外发送RunSvcEvent。此时外部世界收到该Event 后，就开始执行对于Service，一旦其结束，则向该Wheel发送一个Service结束指令，此时Wheel才会再次轮询是否需要再次启动该Service。

PL::SAS::Commands, 用于周期探测./command目录下，是否有已经在配置文件中注册的命令文件，一旦发现某命令文件，则向外部发送RunCmdEvent，此时外部世界收到该Event后，开始执行该Command，一旦其结束，则向该Wheel发送一个Command结束指令，此时Wheel才会再次探测是否有下一个Command。

PL::POE::Whell::Process，用于启动一个新进程（Support both Linux and Windows），并监控该进程是否结束，一旦结束，向外发送ExitEvent；如果timeout超时后，还未结束，则向外发送TimeoutEvent，并尝试强制结束，若强制结束成功，则也会向外发送ExitEvent。

# PL::SAS::Config

该包位于.\lib\PL\SAS\Config.pm，继承自POE::Wheel。使用示例：

PL::SAS::Config->new(

Config => $self->{conf},

ErrorEvent => 'err\_conf',

ChangeEvent => 'chg\_conf',

DoneEvent => 'start\_sas',

);

这里Config是需要监控的文件名，一旦该文件发生修改，就会向外发送ChangeEvent，并附带修改后的doc对象，但如果解析该文件发生任何错误时，则向外发送ErrorEvent。该Wheel实际还接受外部传入的文档对象——即调用$wheel->put($xml\_doc)，一旦Wheel收到该doc对象，则在内部分析处理该文档对象，结束后向外发送DoneEvent，并附带分析结果。

该Wheel内部定义两个状态：

SELF\_STATE\_MONITOR\_CONFIG，该STATE每间隔MONITOR\_CONF\_INTERVAL seconds就会发起一次，此时会检测’Config’参数传入的配置文件的modified time，一旦改变，则emit ChangeEvent，如果发现配置文件读取、解析发生错误，则emit ErrorEvent。

SELF\_STATE\_LOAD\_CONFIG，该STATE只能由用户调用$wheel->put($doc)方法时，才会触发，此时Wheel会分析用户传入的doc对象，分析主要是将配置文件中的用户定义的Services和Commands解析，并关联凡是注册了对应Service/Command的Project，以便对应Service/Command执行时，知道需要在哪些Project中执行。分析完成后，emit DoneEvent。

# PL::SAS::Services

该包位于.\lib\PL\SAS\Services.pm，继承自POE::Wheel。使用示例：

PL::SAS::Services->new(

Services => $heap->{services},

ErrorEvent => 'err\_svc',

RunSvcEvent => 'run\_svc',

);

这里Services是需要监控的Services配置项，这些配置项，都是来自于PL::SAS::Config模块emit DoneEvent时，给出的Service配置项。Wheel会遍历用户传入的Services list，一旦发现有需要执行的Service，则emit RunSvcEvent。至于ErrorEvent，计划是在Service配置项有问题时发送，但目前实现还未向外发送该事件。

该Wheel内部定义四个状态：

SELF\_STATE\_MONITOR\_SERVICE，每间隔MONITOR\_SVC\_INTERVAL seconds，监控内部queue中是否有Service，如则pop之，并emit RunSvcEvent。此时由收到Event的外部世界负责执行。

SELF\_STATE\_QUEUE\_SERVICE，该STATE会收到一个service name的参数，将该name push到queue中。

SELF\_STATE\_CLASSIFY\_SERVICE，会遍历用户传入的Services list，计划是将不同Service类型分类到不同queue中，目前仅实现一个queue——凡是定义了interval的Services都会触发上面定义的SELF\_STATE\_QUEUE\_SERVICE。

SELF\_STATE\_COMPLETE\_SERVICE，该STATE只有外部调用了$wheel->put($svc\_name)才会触发，此时Wheel会参看对应service是否需要再次执行，若interval > 0，则delay指定interval后，触发上面定义的SELF\_STATE\_QUEUE\_SERVICE。

# PL::SAS::Commands

该包位于.\lib\PL\SAS\Commands.pm，继承自POE::Wheel。使用示例：

PL::SAS::Commands->new(

Commands => $heap->{commands},

ErrorEvent => 'err\_cmd',

RunCmdEvent => 'run\_cmd',

);

这里Commands是需要监控的Commands配置项，这些配置项，都是来自于PL::SAS::Config模块emit DoneEvent时，给出的Commands配置项。

该Wheel内部定义三个状态：

SELF\_STATE\_MONITOR\_COMMAND，每间隔MONITOR\_CMD\_INTERVAL判断所有已经注册的命令文件是否存在，若存在则触发下面的内部状态SELF\_STATE\_PREPARE\_COMMAND。

SELF\_STATE\_PREPARE\_COMMAND，该State会收到将要执行的command，然后会判断当前是否已经有还未执行完毕的命令，若有，则忽略该命令（由于没有清理命令文件，在下一个命令监控周期，该命令还会再次进入prepare状态）；否则，判断配置enabled，则向外emit RunCmdEvent，并清除命令文件，以及记录当前正在执行的command。

SELF\_STATE\_COMPLETE\_COMMAND，该STATE只有外部调用了$wheel->put($cmd\_name)才会触发，此时Wheel会清理当前正在执行的command记录。

# PL::POE::Wheel::Process

该包位于.\lib\PL\POE\Wheel\Process.pm，继承自POE::Wheel。使用示例：

PL::POE::Wheel::Process->new(

Program => $program,

ProgramArgs => [@args, $args\_str],

ProgramTimeout => MAX\_WORKER\_TIMEOUT,

TimeoutEvent => 'worker\_timeout',

ExitArgs => [$worker\_type, $worker\_name],

ExitEvent => 'worker\_sig\_child',

);

这里Program是需要执行的程序路径，ProgramArgs是命令行参数array，ProgramTimeout是设置程序的超时时间——超时会强制kill program，TimeoutEvent是在强制kill process后向外发送的Event，ExitEvent是process结束后（无论是自动结束还是强制kill）向外发送的Event；ExitArgs是指定ExitEvent事件发送时，携带的参数。

该Wheel同时支持Windows和Linux，在Linux上，实际是封装了POE::Wheel::Run——使用fork新开进程；在Windows上，Wheel是自实现的，新进程由Win32::Process实现——Win32 API，稳定但功能上，相比Linux的fork要限制很多。

# Command/service实现脚本

无论command还是service的实现，和SAS的关系——仅在参数传递上存在依赖。因为command/service的实现脚本最终都是SAS调用的，那么参数自然也是SAS传递的，由于SAS传递参数时，是一个复杂的数据结构（含有project配置信息），因此SAS会将数据结构序列化为一个扁平ASCII字串，方便命令行参数传递。

因此所有command/service实现的入口函数，首先都是解析传入的ASCII字串为对应的hash，该hash含有两个key：

args，脚本收到的基本参数，在配置文件中command/service的para标签中指定，例如可以是svn用户名、密码等

project，脚本一般需要根据不同project有不同行为，因此需要一个详细project信息，这里key对应的value是一个hash，包含了所有register了command的project配置信息。

# ./bin/auto\_case.pl

## 简介

该脚本是auto\_case command的实现。

首先遍历传入的project参数，将所有project的STA配置文件解析，获取project所有DUT的STA路径信息，然后遍历所有DUT的STA路径，一旦发现Excel版的case文件“CNSS SnS x86 testplan.xlsx”，则基于该Excel文件，自动生成XML case。

## PL::AutoCase(./lib/PL/AutoCase.pm)

这是一个工具类，主要解析Case使用了很多模块，参数使用很复杂，因此使用该类来一次性传入多个模块的不同参数。举例：

my $auto\_case = PL::AutoCase->new;

$auto\_case->load($excel\_file,

Variable => ['GLOBAL\_CONFIG', 'VARIABLE'],

TA => 'TA\_DETAIL', TA\_Template => $ta\_temp,

TC => 'TC\_DETAIL', TC\_Template => $tc\_temp,

TS => 'TEST\_SUITE', TS\_Template => $ts\_temp,

);

如上，PL::AutoCase根据传入的：

Variable参数——Variable相关的Excel Sheel name，自动构造PL::AutoCase::Variable对象，因为这些Variable Sheets定义了其他Sheet使用的变量，因此需要PL::AutoCase::Variable来解析变量，如下面的TA、TC、TS对象，都将获得这里的Variable对象的引用，用以解析其可能用到的变量名。

TA/ TA\_Template参数——指定定义Test Action所在的Sheet name，并根据每行的TA name，构造多个PL::AutoCase::Case::TA对象，且每个TA对象将得到TA\_Template指定的模板文件，由该template得知如何根据TA对象，扩展为对应的Case文本。

TC/ TC\_Template参数——指定定义Test Case所在的Sheet name，并根据Sheet中定义的TC name，构造多个PL::AutoCase::Case::TC对象，且每个TC对象将得到TC\_Template指定的模板文件，由该template得知如何根据TC对象，扩展为对应的Case文本。

TS/ TS\_Template参数——指定定义Test Suite所在的Sheet name，并根据Sheet中定义的TS name，构造多个PL::AutoCase::Case::TS对象，且每个TS对象将得到TS\_Template指定的模板文件，由该template得知如何根据TS对象，扩展为对应的Case文本。

## PL::AutoCase::Case