SpringRain\_Engine结构概述

**引擎结构**

**结构简图：**

**数据部分：**

引擎的数据由多重链表组成，主要用根节点指针作为参数传递方式。数据共分为5类：引擎基本状态数据，游戏对象数据，游戏关卡数据，游戏渲染资源数据，渲染缓存数据。

**引擎基本状态数据**：引擎基本状态数据记录了包括渲染分辨率，动态帧率，全局变量，各模块之间的传递数据的临界资源等基本引擎数据。

**游戏对象数据**：游戏对象数据包括游戏对象的名称，游戏对象的状态变量，游戏对象的行为函数，游戏对象的渲染信息。渲染信息中，大部分数据储存于游戏渲染资源数据中，在对象数据中的部分仅为其指针。

**游戏关卡数据**：游戏关卡数据主要保存游戏关卡的场景数据（即地图数据）、需要引用的对象列表，需要引用的资源列表，数据加载和分配与回收规则，相关的事件触发数据和函数，关卡流程和控制数据。即加载关卡后所涉及到的所有关卡数据。

**游戏渲染资源数据**：渲染资源数据用于存放渲染需要参考使用的光栅数据，相关的变换数据。光栅数据保存了从对象对应的图片文件中提取的未处理过的位图或矢量图数据。实际使用中不会直接修改光栅数据，而是修改光栅数据对应的变量数据。引擎渲染时根据变量数据对光栅数据进行变换计算。同时光栅对应的变量用于场景的碰撞监测和场景模拟。

**渲染缓存数据**：渲染缓存数据保存了从光栅数据到实际输出时的中间状态。即根据需要渲染的对象参照相关变换参数渲染出的用于输出的图层数据。该部分数据会直接用于帧的渲染，这时渲染器将不会也没有必要再从资源库中调取资源。资源库的资源将回交回场景模拟和碰撞监测模块进行实时演算。

**函数部分：**

**构造函数/析构函数**：用于初始化引擎启动时候的各项基本数据（渲染分辨率，动态帧率，全局变量，各模块之间的传递数据的临界资源等）。并调用初始的地图（游戏初始界面）。在游戏运行结束之后用于保存相关数据和释放资源。

**数据提取与加载函数**：数据提取加载函数用于加载关卡资源和相关的对象。该函数首先加载关卡文件的引导文件，根据引导文件从资源文件夹中提取相关的数据。如加载对象资源文件时，根据其属性在内存中开辟适合的存储空间，解压文件中的压缩数据，调用脚本解释器对脚本代码进行处理。加载完成后返回数据加载的报告供引擎判断下一步的操作。

引导文件指向的资源文件有:1、关卡基本信息文件。该文件包含了关卡的基本信息：关卡的大小，名称，版本号等数据。2、地图场景绘制文件。该文件包含了绘制地图场景所需要的对象和对应的文件位置，绘制场景的相关参数，判断标准表，出发事件函数脚本代码，关卡的流程控制脚本代码。3、地图所包含的对象数据文件的位置。

**断言处理函数**：断言处理函数从游戏开发的角度上是供游戏测试调试用的开发函数。在用户使用的角度上用是提供游戏的暂停保存功能，以及高级的控制台功能的函数。在渲染和内部架构上是为渲染器提供了数据保护的功能。该功能会在渲染器的描述中和引擎的执行流程中进行详细说明。

**事件触发和处理函数**：时间触发和处理函数是游戏进行的主函数。该函数负责场景的模拟，各项游戏事件的判定，接受用户输入并做出响应的重要函数。该函数以游戏中的时间轴为基准，将各对象的属性状态列入时间轴。用对象属性的变化和相关的状态来模拟整个游戏场景。函数运行时，会根据对象属性的判定表来监测当前时间点是否有事件被触发。并根据脚本解释器提供的功能模块（函数）对时间进行相应，并修改相关的对象属性设置该属性的下一次判定时间点。即在对游戏空间进行模拟的同时还进行了预测，为渲染器的预渲染，游戏数据的预加载提供了重要的参考信息。

**图形渲染器**：根据图形资源和游戏主函数提供的参考数据进行渲染的函数。其详细结构和工作流程会在下文中进行详细描述。

**音频渲染器**：根据主函数提供的指令和音频数据库进行混音和音频的输出。

**外部接口：**

**I/O接口相关**：提供键盘鼠标手柄等输入设备的输入获取与转换。将获取的按键信息转换为控制的指令。

**窗口和系统api相关**：将引擎内部的硬件调用命令和数据与平台的系统api进行互动与转换，保证引擎的平台无关性。

**引擎数据结构**

**关卡数据结构：**

在这里不对表的形式做出定义。详见后文的描述。

**图形资源数据结构：**

表根节点下有图层表，图层表下有对象节点，对象节点下有对应的光栅数据和状态值。由于状态类型的不固定，所以应使用合适的数据结构。该表直接用于空间模拟。参考值会用于各类型的判定。

**资源对象数据结构：**

这里存放的是资源对象的类结构，因为资源对象的不可知性，不能在编写引擎时进行编译，且可以不考虑对象接口继承等问题。所以采取相关函数用脚本语言编写，属性和数据存放于文件或内存的形式来保存和使用。由于光栅数据涉及到到渲染，需要及时从磁盘文件中读取相关的数据。保证更新渲染缓存时能及时读取到需要的光栅数据。

**渲染缓存数据结构：**

渲染缓存结构与图形资源结构相似，但各参考数据没有了参考函数，图形数据统一渲染出最终结果——适配输出大小的光栅位图数据.

**图形渲染器概述**

**工作流程：**

**步骤简述：**

1：根据之前的数个单帧的渲染时间和缓冲池的指令数来确定渲染的频率。并根据频率向相关函数发出渲染指令。在尽可能不影响用户体验的情况下降低系统的消耗。

2：渲染器接受到渲染单帧的指令之后，立刻向事件触发处理函数（空间模拟器）发送渲染断言。此时事件触发处理函数并不停止对事件的模拟，而是修改了输出和读入的方式。当函数进行输出时，它会向缓冲池发出一条指令，指令被保存在缓冲队列中。当它读取数据时会优先向缓冲池查找修改记录，若无修改记录则向资源库中查找数据。从而实现了时间模拟过程和渲染器的独立。两者的功能不会产生冲突，最严重的情况也只是影响了模拟的速度。这也为以后做游戏服务器主机端和网络端提供了框架的支持。

3：更新渲染缓存的数据时，仅读取图形资源库中的数据而不读取缓冲池中的数据。这一读取方式保证了当前渲染的帧能准确的对应某一时间点。更新时，首先处理的是参数的对比。参数相同时不进行更新。参数不同时更新渲染缓存的参数数据，并将记录压入渲染缓冲区的更新队列。而读取和更新的数据是以镜头的位置和镜头移动的轨迹预测来决定的。镜头中包含的数据对象会全部放入渲染缓存，在移动轨迹上，且在短时间内可能出现的数据也会被置入渲染缓存。以此来保证渲染的完整性和灵活性。

4：向事件触发和处理函数发送渲染断言解除命令。恢复渲染资源库的可写性。并优先把缓冲池中的指令进行执行，更新相关数据。若缓冲池中还有更新指令，则将新指令放入缓冲池，若无指令则直接执行更新指令。

5：渲染单帧时，先根据渲染缓冲区更新队列中的节点依次计算和更新光栅数据。完成之后根据图层树来对各图层进行叠加与相关特效的渲染。最终根据相机的位置对整个场景进行裁剪。舍弃不需要的部分，然后调用于输出单帧的函数。

6：输出函数收到输出命令，根据不同的软硬件环境输出单帧到显示器。至此完成单帧的渲染和输出。

**关卡数据结构**

因暂时没有脚本和脚本解释器的资料，本部分暂不做描述。