Sora新增内容说明

1. SoraFunction 仿函数模板
2. SoraSimpleFSM 模板状态机
3. SoraSignal Signal模板
4. SoraEventWorld Event消息机制
5. SoraEntity 组件系统实践
6. SoraTimer 新的Timer
7. SoraThread 新的Thread实现
8. 新增宏
9. 新增模板
10. 新增机制
11. SoraFactory Factory模式实践

SoraFunction 仿函数模板

SoraFunction是boost::function的山寨实现，用法基本用boost::function一样. 模板接受一个参数, 为函数签名

```
SoraFunctio<Signature>
```

函数签名利用C++的模板函数签名会退化成函数指针类型特性，例如有模板

```
template<typename sig>
struct MyClass {
	typedef sig FuncType;
};
```

然后使用MyClass<void(int, int)>, 则MyClass::FuncType实际上会被理解为void (*)(int, int)的函数类型.

所以要使用SoraFunction代表一个函数非常简单，例如有函数

```
float doSomething(float a, float b) {
	return a * a + b * b + 2 * a * b;
}
```

则SoraFunction的表示为

```
SoraFunction<float(float, float)> myFunc;
```

然后可以对这个对象赋值.

```
myFunc = doSomething;
```

然后就可以像使用函数一样使用myFunc, 例如

```
myFunc(10.0, 10.0);
```

注意，SoraFunction支持函数参数和返回值的隐式转换，例如

```
SoraFunction<int(int, int)> myFunc2;
```

令

```
myFunc2 = doSomething;
```

也是被允许的.

除了C函数外，SoraFunction还支持类成员函数和仿函数.
对于类成员函数，有两种方式来代表

例如有类

```
struct MyClass {
	void doSomething(int a) {
		return a + 10;
	}
};
```

则

1. 使用this指针

```
	SoraFunction<void(MyClass*, int)> myFunc;
	myFunc = &MyClass::doSomething;

	MyClass instance;
	std::cout<<myFunc(&instance, 1);
```

2. 使用SoraBind

SoraBind是一个模板helper, 类似于std::bind, 可以绑定一个对象和他的函数到一个仿函数上, 但是不像stl的bind最大只支持2个参数，SoraBind支持最多12个参数. 例如

```
	SoraBind<MyClass, void(int)> myBind(&instance, &MyClass::doSomething);
```

然后可以像仿函数一样使用这个对象

```
	myBind(10);
```

则实际上调用了instance.doSomething(10);

由于SoraFunction支持仿函数，所以这种方式实际上是用过仿函数实现的

```
	SoraFunction<void(int)> myFunc(Bind(&instance, &MyClass::doSomething));
	myFunc(10);
```

注意这里使用的是类似于std::bind的sora::Bind函数，这个函数返回一个SoraBind仿函数对象.

注意stl或者别的库里的仿函数也可以直接用于SoraFunction。

利用SoraFunction，我们就有了更抽象的用于表示任意函数的对象，而不用为了C函数和类成员函数去特化，或者只支持一边。回调或者代理机制得以更好的实现。

* 关于SoraFunction和SoraSignal注册函数的使用

最好使用模板, 例如

```
template<typename T>
void myRegisterFunc(const T& func) {
	myFunction = func;
}
```

这样可以忽略具体类型，也不用麻烦用户来手动构造SoraFunction(以SoraFunction作为参数的形式)

然后使用就是 myRegisterFunc(myFunc / SoraBind(myObject, ObjectFunc) / MyFunctor)

SoraSimpleFSM 模板状态机

SoraSimpleFSM是SoraFSM的简化版，只支持状态的设置，转移和获取，并不直接支持类状态。但是他的状态类型和转移类型都是模板的，所以可以是任意类型，方便了具体目标的实现。

template<typename state_type, typename event_type>
class SoraSimpleFSM;

模板参数2个，第一个代表类状态的表示类型，例如std::string, 也可以自定义class类型. event_type则是事件类型，用于定义类状态转移.

SoraSimpleFSM有defTrans函数用于定义一个状态转移的边, 然后通过procEvent来进行状态转移，如果当前状态拥有处理这个Event的能力，则会转移到defTrans定义的下一个状态。同时procEvent还支持两个仿函数参数用于回调状态进入和退出。

类函数如下
add(state_type state)
添加一个状态

setState(state_type state)
设置当前状态

clear()
清除所有状态

defTrans(state_type state, event_type event, state_type stateto)
定义一个状态转移, state的启示状态，event是事件，stateto是目标状态

delTrans(state_type state, event_type event)
删除一个状态转移

procEvent(event_type event)
根据当前状态和event转移到下一个状态(如果存在)

template<typename enter_op, typename exit_op>
procEvent(event_type event, enter_op enter, exit_op exit)
和上边一样，enter函数在状态转移之前被调用，exit函数在状态转移之后被调用，函数接受(SoraSimpleFSM&, state_type)

curr()
获得当前状态

SoraSignal Signal模板

SoraSignal是boost::signal的山寨实现，基于SoraFunction，用法同boost::signal基本一样。
Signal相当于多重回调机制。你可以连接多个函数到一个Signal，然后在触发时他们会被逐一回调，并且返回最后一个返回值。并且支持优先级排序.
这个机制在需要支持多重回调的地方很有用，例如listener, 或者一个事件(比如文件下载完毕)发生时需要通知复数个对象的时候.

```
template<typpename SIG>
SoraSignal;
```

模板参数也是函数签名，记得SoraFunction那个函数签名么，含义一样.
你可以通过SoraSigal::connect来连接到一个signal, 通过sig或者operator()来执行这个signal.
例如

```
float myFunc(float a, float b) {
	return a + b;
}

double myFunc2(float a, float b) {
	return a + b;
}

SoraSignal<float(float, float)> mySignal;
mySignal.connect(myFunc);
mySignal.connect(myFunc2);

float result = mySignal.sig(10.f, 20.f); // or mySignal(10.f, 20.f) directly
```

则result实际上是myFunc2的返回值(注意和SoraFunction一样，Signal也支持函数参数和返回值的隐式转换)

但是如果在连接的时候使用

```
mySignal.connect(myFunc);
mySignal.connect(1 /* priorirt */, myFunc2);
```

则返回的是myFunc的值(优先级排序，注意SoraSignal总是返回最后调用的函数的返回值)

同时如果要断开连接的话，则可以使用SoraConnection，SoraSignal::connect实际上返回SoraConnection对象

```
SoraConnection myConnection = mySignal.connect(myFunc);
```

用户需要使用的只有disconnect函数，表示断开和signal的连接。
并且SoraConnection基于SoraAutoPtr的引用计数，在所有SoraConnection对象被销毁时，自动断开连接。

SoraEventWorld 新的Event消息机制

SoraEventHandler已经被扩展, 支持监听某个EventWorld的某个Channel, Sora默认最多支持24个Channel，不同Channel的Event走不同通道分发。
同时SoraEvent也已经扩展，支持Receiver和Channel。
当然以前默认的机制还是可以用，没有做出改动。扩展的部分是用于方便实现Event监听体系。

SoraEventWorld是消息的发布者，如果一个Handler要监听某个EventWorld的消息，则必须先Enter某个World(通过SoraEventWorld::enter(SoraEventHanlder*)函数
你可以通过SoraEventWorld::defaultWorld这个静态函数来获取全局的EventWorld。

同时所有进入某个EventWorld的handler可以选择监听UpdateEvent(SoraEventHandler::enableUpdate(receiveEvent), 如果receiveEvent为真，则Handler在收到UpdateEvent的同时还会发布这个Event到handleEvent, 否则只是调用SoraEventHandler::onUpdate), 来达到某种意义上的自动Update功能

同时SoraEventHandler可以选择监听某个Channel的Event(SoraEventHandler::setChannel). SoraEventChannle是一个模板trait生成类，在默认32bit数据的情况下，最大支持24个Channel(8个system, 16个user), 你也可以通过更改SoraPlatform内的Channel数据类型(例如int64)来达到更高的通道宽度.
你可以通过GetSystemChannel<N>或者GetUserChannle<N>来获取某个通道的值，或者通过SoraEventChannel.fill()来填充通道值来支持所有通道.

SoraEventHandler只会接收到正在监听的通道的消息(除非明确指明Receiver的Event)

同时注意，所有通过SoraEventWorld发布的消息必须使用SoraEventFactory创建，因为SoraEventWorld的实现会缓存发布的消息，在下一次同步再发送，然后destroy，所以至少必须使用new创建(不建议)

SoraEventFactory默认已经注册了所有系统事件类型，你可以通过全局静态函数CreateEvent(name)来创建已经注册的Event类型(RegisterEvent同于注册), 已经注册的包括
PlaybackEvent
FileChangeEvent
KeyEvent
TimerEvent
ConsoleEvent
MenubarEvent
HotkeyEvent
MessageEvent

SoraEntity 组件系统实践

所谓组件系统，就是和继承体系不同，认为所有的物件都是由一个个的组件(Component)组成，例如人是由手，脚，头等等, 每个组件负责不同的功能。
SoraEntity(SoraLightWeightEntity)就是这个想法的实践，他可以动态的添加和删除SoraComponent, 每帧会update, render component, 在收到消息时会转发的Component，Component可以通过owner和另外的Component通信，同时为了支持Data Driven, 还支持字符串作为key的动态Property，例如

```
SoraEntity myEntity;
myEntity.addProperty(“type”, “tank”);
myEntity.addProperty(“id”, ID_TANK);

SoraRenderComponent* comp = CreateComponent(“BasicRenderer”);
comp->setSprite(L”pics/tank.png”);
myEntity.addComponent(comp);

SoraPositionComponent* pos = CreateComponent(“Position”);
myEntity.addComponent(pos);
```

则创建了一个tank对象(假设), 这样的话就避免了越来越复杂的继承体系，而所有对象都可以通过Entity代表，不用再区分类型，他们不同之处只有他们的Component不同，且Component是可以动态添加和删除的。

这里RenderComponent会要求Position Component的属性，可以通过很多方式实现，这里也就不详细写了。(例如加载时通过onComponentAdd获取PositionComponent并保存指针, 或者动态向entity owner申请)

SoraComponent本身也支持动态Property，所以可以很简单的把一个带有Component的Entity映射到配置文件(xml，json etc)，且可以通过配置文件动态更改(data driven)，所有的key都是字符串，所以配置文件是直接映射来达到序列化的目的。

通过SoraEntity，也可以直接访问到某个Component的Property(如果是通过Property模式)
例如

```
myEntity.setProperty(“Position.x”, 10.f);
```

则实际上设置了Position Component的属性x(如果存在，否则是异常)

如果为了效率不使用动态Property，也可以这样

```
myEntity.getComponent<SoraPositionComponent>->setPositionX(10.f);
```

这个取决你的Component的实现方式, 并非强制.

同时Entity和Component都支持动态继承(addParent, delParent), 当然仅限于使用动态Property的情况下。

* Property和TypeSerializer

为了效率， SoraProperty并非使用RTTI而是使用SoraTypeSerializer来特化自身和判断类型转换，默认的SoraTypeSerializer已经支持Sora定义的所有类型和任意指针类型. 你也利用模板扩展,

template<typnename T, typenmae S = SoraTypeSerializer>
第二个参数就是TypeSerializer, 默认是SoraTypeSerializer

TypeSerilizer必须给一个类型提供
	默认值
	TypeId(int)
	从string转换
	转换到string
4个功能

* SoraLightWeightEntity和SoraEntity

SoraEntity在SoraLightWeightEntity的基础上加上了fsm, scriptVM, listener支持

* SoraComponent的heavyWeight参数

HeavyWeight的Component在加入时或者别的Component加入时，会被回调onComponentAdd来通知现有Component和新加入的Component，在移除时会被通知onComponentRemove
而非HeavyWeight的Component就只会在被加入时被通知所有现有的Component(参见ComponentHolder实现)

SoraSimpleTimer 新的Timer

基于SoraFunction，回调函数原型bool(SoraSimpleTimer*, TimeType/*float*/), 所以不必再使用SoraTimerEvent那种麻烦的方式了，支持任意类型函数.
这个函数返回true这Timer会被自动停止

可以通过CreateSimpleTimer创建一个SimpleTimer(不会开始)
这个函数返回一个SoraAutoPtr<SoraSimpleTimer>, 你必须手动保存这个对象并且使用他来开始某个Timer，由于使用智能指针，当对象析构Timer也就被删除，所以你必须使用某种方式来保存这个对象直到你不需要这个Timer了

7. SoraThread 核心Thread库

位于Sora/Thread, 所有的东西的用处和他的名字一样，这里也就不详细写了。需要注意的是现在在windows下默认不是使用pthread vc而是Windows原生线程。如果需要强制使用pthread(毕竟可能有微秒的语意不同), 需要定义 SORA_WIN32_PTHREAD(SoraPlatform.h)

8. 新增宏 New Macros

SoraPlatform.h

SORA_JOIN 合并两个名称，例如
SORA_JOIN(a, b)将展开为ab

sora_static_assert 静态断言
用于编译期的断言, 例如检测一个迭代静态量为某个值

SORA_STATIC_INSTANCE_DECLARE
SORA_STATIC_INSTANCE_IMPL

静态Singleton实现(非继承体系, 前者必须用于类中，后者必须用于类实现

SORA_UNIQUE_NAME(name)

获取一个基于代码行数的独一无二的名字，因为基于代码行数，所以跨文件可能会有重复(如果使用同一个名字)

SORA_STATIC_RUN_CODE_I(name, code)

在静态初始化时运行一段代码, name用于防止SORA_UNIQUE_NAME造成的命名重复, code是代码，可以多行，例如
SORA_STATIC_RUN_CODE_I(mycode, a = b + c; c = a + b; b = a + c;)

SORA_STATIC_RUN_CODE(code)

静态初始化时运行一段代码, 采用默认的sora_static_(__LINE__)作为名字，跨文件使用可能导致命名重复而无法运行

SORA_STATIC_INIT_I(name, fn)

静态运行一个函数, name用于方式SORA_UNIQUE_NAME造成的命名重复

SORA_STATIC_INIT(fn)

静态运行一个函数，采用sora_static_init_##FN默认命名

SORA_STATIC_RUN_FN(fn)

等同于SORA_STATIC_INIT(fn)
SORA_STATIC_RUN_CODE_FN(name, code)

采用SORA_STATIC_INIT的模式在静态初始化运行一段代码(宏生成函数)

SORA_FSM_USE_NULL

如果此宏被定义，则SoraSimpleFSM会默认插入一个null状态，否则不会，默认开

SORA_WIN32_PTHREAD

如果此宏被定义，则SoraThread库会使用pthread作为win32下的thread库，否则采用Win32 API原生

SORA_AUTOMATIC_REGISTER

如果此宏被定义，则支持AUTOMATIC_REGISTER特性的插件会在静态初始化时自动注册到Sora, 参照10. 新机制的AutomaticRegister

SoraPreDeclare.h

SORA_DEFINE(name)
SORA_IF_DEFINED(name)

运行时定义检测，实际上是map查表，所以很慢，不知道用来干啥

SORA_CLASS_DEF_FIELD_SET_GET(type, name)
SORA_CLASS_DEF_FIELD_SET_GET_P(type, name, pref)

方便在一个类中定义一个可被set和get的变量, type为类型，name为名字，pref为名字前缀，例如

```
SORA_CLASS_DEF_FIELD_SET_GET_P(int, Principle, m)
```

将生成

```
private:
    int mPrinciple;
public:
    int getPrinciple() const {
        return mPrinciple;
    }
    void setPrinciple(int val) {
        mPrinciple = val;
    }
```

SoraEvent.h

SORA_EVENT_IDENTIFIER

现在会插入一个静态函数方便类型直接比较，并且现在SORA_USE_RTTI默认为关，因为效率问题。
所以这个宏在Event类实现中默认为强制

接下来是关于工厂模式的宏，具体参见11. SoraFactory

SoraGlobalFactory.h

REG_GLOBAL_PRODUCT_0(T, NAME, FN)
REG_GLOBAL_PRODUCT_1(T, NAME, FN, A0)
REG_GLOBAL_RPODUCT_2(T, NAME, FN, A0, A1)

工厂产品注册宏，实际上是
SoraAbstractFactory<T(, A0, A1)>::Instance()->reg(name, fn)

SoraSpriteFactory.h

GlobalSpriteFactory

将被替换为SoraSpriteFactory::Instance()

REGISTER_SPRITE_PRODUCT(name, fn)

将被替换为GlobalSpriteFactory->reg(name, fn)

SoraComponent.h

SORA_DEF_COMPONENT(cls)

对于Component实现必要的宏，将插入Component名字，建议使用类名防止混乱

SoraLightWeightEntity.h

SORA_DEF_ENTITY(class, description)
SORA_IMPL_ENTITY(class)

SoraEntity的定义宏，将插入几个静态函数create, destroy, 并且插入SoraDynRttiClass信息. 如果你选择实现你自己的Entity，建议加入

SoraDynRttiClass.h

SORA_DEF_DYN_RTTI_CLASS(class, description)
SORA_IMPL_DYN_RTTI_CLASS(class)
SORA_IMPL_DYN_RTTI_CLASS(class, parent)

定义、实现一个DynRttiClass(在类中插入DynRttiClass的静态信息), DEF必须在类中使用, IMPL必须在类定义中使用. 会插入一些静态函数和变量，包括getClassName, getClassDescription, getRttiClass, rtti_cls;

SoraConsole.h

SORA_DEF_CONSOLE_EVT_FUNC(func, cmd)

把一个C函数注册为SoraConsoleEventHandler, cmd为要接受的命令，可以用分隔符分开，例如
get;set;lalala;lololo

方便写Console命令解析

SoraEvent.h

SORA_DEF_FUNC_AS_EVENT_HANDLER(func, evtType)

把一个C函数注册为SoraEventHandler，会生成一个叫做func##EventHandler的类，没搞清楚有什么用，其实没用大概(?)

9. 新增模板
新增的模板大部分位于各个机制内(他们都是模板基类), 所以这里就懒得都写一遍了

SoraEventChannel.h

template<int N>
GetSystemChannel()

template<int N>
GetUserChannel()

用于获取第n个System/User SoraEventChannel

SoraAny.h

template<typename T>
isAnyType(const SoraAny& any)

判断某个SoraAny是否为T类型

10. 新增机制

AutomaticRegister

利用Sora核心的静态运行宏，支持这个特性的插件都会在静态初始化时自动注册自身(如果编译到了源代码内)，而无需再手动注册这些插件.
现在Sora核心基本所有需要注册的插件都已经支持这个特性.
你可以在SoraPlatform的SORA_AUTOMACTIC_REGISTER宏关掉他，然后手动注册

* 拥有AutomaticRegister后Sora的初始化

Sora核心新增了InitAndCreateSoraCore函数，这个函数接受两个参数
第一为要创建的Window
第二为配置参数，类型为SoraCoreParameter, 目前支持3个，按顺序为
	LoadPlugins
	UseFullscreenBuffer
	PostErrorUsingMessageBox

所以启用SoraAutomaticRegister的情况下一行就可以启动Sora

```
sora::InitAndCreateSoraCore(new myWindow,
                                    sora::SoraCoreParamter(false, false, false));
```

* 插件的AutomaticRegister支持

用SORA_STATIC_RUN_CODE(_I, STATIC_INIT等)宏就可以了

例如

```
#ifdef SORA_AUTOMATIC_REGISTER
	SORA_STATIC_RUN_CODE_I(SMRegister, SoraCore::Instance()-
>registerSoundManager(new mySoundManager))
#endif
```

LuaAutoExport

这个和AutomaticRegister机制类似，也是利用静态运行代码来达成新的Lua导出体系
新的Lua导出体系中
你使用SoraLuaExporter这个静态类来连接需要的导出函数，然后在创建SoraLuaObject或者手动使用OnExport时，这个导出函数就会通过SoraSignal被调用，然后在函数内你可以导出你需要导出的函数

原有的体系已经被更新为支持这个特性，所以guilib等lua导出放回到了原有的插件内, 只要在编译单元内就行(如果启动AutoExport，否则你需要手动连接到SoraLuaExporter, [SoraLuaExpoter::ConnectExporterFunc(const T& func)])

而LuaAutoExport就是在静态初始化时静态连接导出函数到LuaExpoter的功能，这样既有了扩展性也方便了导出(只要在编译单元内)，也可以配置需要导出的内容(SoraLuaConfig.h)
当然这也取决于导出函数到底是否支持这个特性(需要用宏手动申明, 下述)

SoraConfig.h
这里定义了具体导出的配置, 注意这里的宏必须定义为0或者1
SORA_LUA_AUTO_EXPORT_ALL
导出所有，导出函数的特性支持实现应该包含这个的检测，下述
这个宏表示是否导出编译单元内所有可导出的导出函数, 会覆盖掉其余所有配置
默认为0(关)

SORA_LUA_AUTO_EXPORT_FONT
SORA_LUA_AUTO_EXPORT_GUI
SORA_LUA_AUTO_EXPORT_SPRITE
SORA_LUA_AUTO_EXPORT_CORE
SORA_LUA_AUTO_EXPORT_SYMBOLS
SORA_LUA_AUTO_EXPORT_ANIMATED_SPRITE
SORA_LUA_AUTO_EXPORT_GIF_SPRITE
SORA_LUA_AUTO_EXPORT_BGM_MANAGER

这些就是具体的导出函数定义配置了，随着Sora可导出部分的增加这个列表可能会被扩展，默认全开

SoraLuaExporter.h

特性支持宏

SORA_LUA_CHECK_AUTO_EXPORT_SYMBOL(Symbol)
检测某个导出Symbol是否存在(或者被SORA_LUA_AUTO_EXPORT_ALL覆盖)

SORA_LUA_AUTO_EXPORT_FUNC(fn)
静态将一个函数注册到LuaExporter

所以一个典型的自动导出支持示例如下

```
void myExportFunc(LuaPlus::LuaState*) {
        LuaClass(state, “myclass”)
        .def(“xxx”, xxx)
        .def(“yyy”, yyy);
}
```

```
#if SORA_LUA_CHECK_AUTO_EXPORT_SYMBOL(MySymbol)
        SORA_LUA_AUTO_EXPORT_FUNC(myExportFunc)
#endif
```

注意SORA_LUA_CHECK_AUTO_EXPORT_SYMBOL宏会自动检测SORA_LUA_AUTO_EXPORT_ALL

原有的导出已经更新为支持这个特性，例如SoraLuaExport里就有

```
#if SORA_LUA_CHECK_AUTO_EXPORT_SYMBOL(SORA_LUA_AUTO_EXPORT_CORE)
        SORA_LUA_AUTO_EXPORT_FUNC(lua_export_sora_core);
#endif
```

等等

11. SoraFactory Factory工厂模式实践

SoraAbstractFactory是模板工厂基类，最多支持2个参数的构造函数和自定义构造函数
可以用默认(最多带两个参数的)的默认类型构造函数注册

```
SoraAbstractFactory::reg_ctor<product>(“my_product”);
```

也可以注册自定义构造函数

```
SoraAbstractFactory::reg(“my_product”, my_constructor);
```

然后通过

```
SoraAbstractFactory::createInstance(“my_product”)
```

创造实例

注意SoraAbstractFactory的模板参数为基类类型，也就是一个工厂只能创造某种类型以及这个类型的派生类型的产品

SoraSpriteFactory

SoraSpriteParser的工厂实现，SoraSpriteParser已经被标记为删除
同样利用静态运行代码特性，基于SoraSprite的类型，例如Animation和Gif都会在静态运行时注册自身到SoraSpriteFactory
同时SoraSpriteFactory的构造支持两个参数, 第一个为path，贴图路径，第二个是json::Value*, 用于传递参数(待修改为更抽象的模式, 这样对jsoncpp依赖太大)

这个Factory在SoraSprite的实现里已经支持了SoraSpriteParser所有支持的参数(shader, effect等等)
同时如果你编译了SoraSpriteAnimation和SoraGifSprite，也可以通过此工厂创建那两种类型(没有人工手动干预!一切都是自动的!)

默认类型就是sprite
例如

```
sora::SoraSprite* mySprite = CreateSpriteType(“sprite”, L”pics/my_sprite.png”);
```

或者在有gif的前提下

```
sora::SoraSprite* mySprite = CreateSpriteType(“gif”, L”pics/my_gif.gif”);
if(mySprite)
    sora::SoraGifSprite* gif = static_cast<sora::SoraGifSprite*>(mySprite);
```

同样对于Animation的类型为”animation”

如果你自己添加了新的Sprite类型，可以通过RegisterSpriteType(const std::string& type, const T& fn)来注册到工厂
如果加上静态运行就更好了 =w=

SoraDynamicFactory是另一种工厂实现方式，采用SoraInstatiator这个类来构造产品，但是没有SoraAbstractFactory灵活(默认只支持ctor)，还必须构造SoraInstantiator的子类，所以我也不知道有啥用