Lua bridge -0.1 light-weight, dependency-free library for binding Lua to C++.

Example:

```
luabridge::module m(L);
    m.function("foo", &foo)
     .function("bar", &bar);
```

support types:

primitive types: int, bool, float, double,

String : const char*, std::string

objects : pointers, references, and shared_ptr

```
m.class_<MyClass>("MyClass")                    构造ctor的signature.
 .constructor<void (*) (/*parameter types*/)>()
 .method("method1", &MyClass::method1)
 .method("method2", &MyClass::method2);

m.subclass_<MySubclass, MyBaseclass>("MySubclass")
 .constructor<...>
 ...
```

class module Helper class 从上面的example可知，提供接口，方便注册函数，类到lua环境中。

```
template <typename T>
class_<T> class_(const char* name);
```

注册类的方法

```
template <typename subclass_, typename Baseclass>
class_<T> subclass_(const char* name);

template <typename T>
class_<T> class_(); → 注册更多方法.
```

注册函数

```
template <typename FnPtr>
module& function(const char* name, FnPtr fp);
```

class class_

```
template <typename FnPtr>
class_<T>& constructor();

template <typename FnPtr>
class_<T>& method(const char* name, FnPtr fp);
```

描述构造ctor的函数签名

```
template <typename T>
class classname {
    static const char* name_; // 初始化为unknown
    static const char* name() { return name_; }
    static void set_name(const char* name_) {
        name_ = name;
    }
};
```

给定任一类，classname<T>::name() 便可知道类的名字.

如果类已经注册，则 name() != unknown.

在注册某个类时，M.class_<T>("T")

会返回一个class_object 对象，

它构造时会帮忙设置 classname<T>::set_name("T");

- 类可以与注册到 lua 中的类名(lua table名)不同.

typelist

```
typedef void nil;
template <typename Head, typename Tail=nil>
struct typelist {};

template <typename Typelist>
struct typevallist {};
template <typename Head, typename Tail>
struct typevallist <typelist<Head, Tail>>
{
    Head hd;
    typevallist<Tail> tl;

    typevallist(Head hd_, const typevallist<Tail>& tl_)
      : hd(hd_), tl(tl_)
    { }
};
```

```
template <typename FnPtr>
struct fnptr{};          ⇒ 主模板
```

对指针函数的偏特化

```
template <typename T, typename Ret, typename P1>
struct fnptr<Ret (T::*)(P1)>
{ typedef typelist<P1> params;
  static Ret apply(Ret T::*obj,
                   Ret (T::*fp)(P1),
                   const typevallist<params>& tvl)
  {
    return (obj->*fp)(tvl.hd);
  }
};
```

```
template <typename Ret>
struct fnptr<Ret (*)()> ⇒ 模板偏特化 有返回值,无参数的函数
{
  typedef nil params;
  template
  static Ret apply(Ret (*fp)(), const typevallist<Params>& tvl)
  {
    (void)tvl;
    return fp();
  }
};
```

```
template <typename Ret, typename P1>
struct fnptr<Ret (*)(P1)> ⇒ 模板偏特化
                          有返回值,有1个参数的函数
{
  typedef typelist<P1> params;
  static Ret apply( Ret (*fp)(P1), const typevallist<params>& tvl)
  {
    return fp(tvl.hd);
  }
};
```

```
template <typename Ret, typename P1, typename P2>
struct fnptr<Ret (*)(P1,P2)>  ⇒ 模板偏特化
                              有返回值,有2个参数的函数
{ typedef typelist<P1, typelist<P2>> params;
  static Ret apply (Ret (*fp)(P1,P2), const typevallist<params>& tvl)
  {
    return fp(tvl.hd, tvl.tl.hd);
  }
};
```

3个参数: `typedef typelist<P1, typelist<P2, typelist<P3>>> params;`

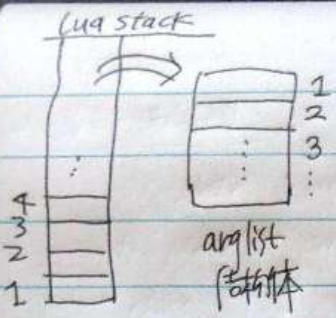
```
template <typename T, typename typelist>
struct constructor {};   // 主模板 T: 对象类型, typelist: 类构造函数参数列表

template <typename T>
struct constructor <T, nil>  // 模板偏特化
                             无参数的构造函数.
{
  static T* apply (const typevallist<nil>& tvl)
  { tvl;
    return new T;
  }
};
  template <typename T, typename P1>
  struct constructor <T, typelist <P1>> // 模板偏特化
                                        有1个参数(P1)的构造函数.
  {
    static T* apply (const typevallist<typelist<P1>>& tvl)
    {
      return new T(tvl.hd);
    }
  };
```

注册1个freefunction到lua

```
template <typename FnPtr>
module& module::function(const char* name, FnPtr fp)
{
  lua_pushlightuserdata (L, (void*)fp); // 作为 cclosure的 upvalue
  lua_pushcclosure(L, &function_proxy<FnPtr>::f, 1);
                                                  ==> 1个upvalue
  lua_setglobal(L, name);   静态函数, 推导FnPtr的签名
  return *this;
}
```

```
template <typename FnPtr, typename Ret = typename fnptr<FnPtr>::returntype>
struct function_proxy {
  typedef typename fnptr<FnPtr>::params params; // typelist
  static int f(luaState* L) { // lua要求的函数签名
    FnPtr fp = (FnPtr)(lua_touserdata(L, lua_upvalueindex(1));
    arglist<params> args(L); // 从lua stack中获取函数调用的参数
                             (聚合成1个typevallist)
    tolstack<Ret>::push(L, fnptr<FnPtr>::apply(fp, args));
                                  ==> typevallist 类型(要求).
    return 1;
  };
};
```

化作为 typevallist




```
template <typename Typelist, int start=1>
struct arglist {};
```
默认值，影响所有子模板

```
template <int start>
struct arglist<nil, start>: public typevallist<nil> //递归终止模板
{
    arglist(lua_State* L) {(void)L;}
};

template <typename Head, typename Tail, int start>
struct arglist<typelist<Head, Tail>, start>:
    public typevallist<typelist<Head, Tail>>
{
    arglist(lua_State* L):  // 构造函数
        typevallist<typelist<Head, Tail>> //父类
        (tdstack<Head>::get(L, start),
        arglist<Tail, start+1>(L))
    {}
};
```

start 1 → 父类1
2 → 父类2
(start+1)

arglist是个空壳
?: 帮助推导，直到 nil 为止

```
template<typename T>
struct tdstack {};

template <>
struct tdstack<int> // 模板特化
{
    static void push(lua_State* L, int data)
    {
        lua_pushinteger(L, data);
    }
    static int get(lua_State* L, int index)
    {
        return luaL_checkint(L, index);
    }
};
```

如何注册构造函数?

```
template <typename T>
template <typename FnPtr> //根据签名来调用
class_的 class_<T>::constructor()
{
    luaL_getmetatable(L, classname<T>::name());
    lua_pushcclosure(L,
        &constructor_proxy<T, typename fnptr<FnPtr>::params>, 1);
    lua_setglobal(L, classname<T>::name());
    return *this;
}
```

隐式帮助fnptr推导出参数类型.

1 → metatable作为upvalue.

luacode: `a = A(2,3);`

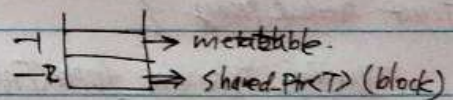
⇒ 函数名对应类名.

a是个shared_ptr<T>用作索引

```
template <typename T, typename Params>
int constructor_proxy(lua_state* L) //符合lua注册函数的签名
{
    void* block = lua_newuserdata(L, sizeof(shared_ptr<T>));
    arglist<Params> args(L); //根据参数typelist中含有的参数,记录有多少stack arguments
    new(block) shared_ptr<T>(constructor<T, Params>::apply(args)); //placement new
                                  ⇒ typevallist 返回 new T(...)
    lua_pushvalue(L, lua_upvalueindex(1));
    lua_setmetatable(L, -2);
    return 1;
}
```

⇒ metatable

⇒ 设置用户数据的 metatable



并且如何设置/创建类A的metatable呢? 在 class_的ctor中 metatable存储在registry中, 类名作为key

```
luaL_newmetatable(L, name);
lua_pushcfunction(L, &subclass_indexer);
lua_setfield(L, -2, "__index");
lua_pushstring(L, name);
lua_pushcclosure(L, &destructor_dispatch<T>, 1);
lua_setfield(L, -2, "__gc");
lua_pushstring(L, name);
lua_setfield(L, -2, "__type");
lua_pop(L, 1); //弹出metatable
```

```
template<typename T>
int destructor_dispatch(lua_state* L)
{
    ((shared_ptr<T>*)(checkclass(T, 1,
        lua_tostring(L, lua_upvalueindex(1)))))->reset();
    return 0;
}
```

将栈清空.

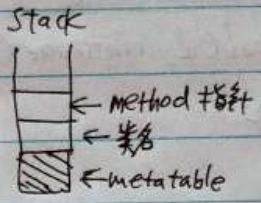
如何调用类的方法?

a = A(1, 2).

a:method_call(xx)

- 类的方法注册:

```
template <typename T>
template <typename FnPtr>
class_<T>& class_<T>::method(const char* name, FnPtr fp)
{
    luaL_getmetatable(L, classname<T>::name());
    lua_pushstring(L, classname<T>::name());
    void* v = lua_newuserdata(L, sizeof(FnPtr));
    memcpy(v, &fp, sizeof(FnPtr));
    lua_pushcclosure(L, &method_proxy<FnPtr>::f, 2);
    lua_setfield(L, -2, name);
    lua_pop(L, 1);
    return *this;
}
```



lua_setfield(L, -2, name); → metatable中的name, cclosure包装method_proxy

lua_pop(L, 1); → metatable

method_proxy是个结构体模板, 提供静态f函数, 所以为结构体, 且可以特化返回值为void类型的method.

```
template <typename FnPtr, typename Ret = typename fnptr<FnPtr>::result_type>
struct method_proxy {
    typedef typename fnptr<FnPtr>::classtype classtype;
    typedef typename fnptr<FnPtr>::params params;
    static int f(lua_State* L)
    {
        classtype* obj = ((shared_ptr<classtype>*)checktype(L, 1,
            lua_tostring(L, lua_upvalueindex(1))))->get();
        FnPtr fp = *(FnPtr*)lua_touserdata(L, lua_upvalueindex(2));
        arglist<params, 2> args(L);
        tdstack<Ret>::push(L, fnptr<FnPtr>::apply(obj, fp, args));
        return 1;
    }
};
```

是个shared_ptr指针, 在lua中.

对象object, userdata

它设置了metatable, 与name对应的metatable一样.

根据类名获取类对应的object*

从第2个stack value开始是params. 那第1个stack value是什么??

是a对象指针(userdata)