# 代码规范

## 基本思路

正确性是第一位的，重要性高于运行效率，不要过早优化。

## 内存和指针使用规范

除底层逻辑之外，不动态分配内存。

只在必要时，才使用指针，尽量少使用指针。

可以使用引用时，尽量使用引用。

防止引用已经释放的内存或者无效的内存，使用指针前必须判空。

指针在被delete之后，必须置为nullptr。

使用new[]分配的指针，必须用delete[]释放，使用new分配的指针，必须使用delete释放。

禁止指针与其他类型互转。

不得返回局部变量或者其他临时变量的指针或引用。

不重载operator new delete。

不使用std::auto\_ptr。

## 数组与使用规范

使用solar::array代替原生数组，除用于格式化输出char[]外，不使用原生数组。

使用solar::vector代替std::vector，不使用std::vector。

访问数组或者vector之前，必须判断下标。

## 类设计规范

类不需以“C”做开头。

类成员变量以m\_开头。

构造函数无比正确初始化各个成员变量，可抛出异常，但要保证资源不会泄露。

单参数构造函数声明为explicit。

析构函数必须正确释放资源，绝不能抛出异常，避免宕机。

子类重置虚函数不省略virtual关键字，必要时增加override关键字。

类中有引用类型，比如指针、引用、句柄等，则此类或者声明为noncopyable，或者自行实现拷贝构造函数和赋值操作符。

对delete基类指针这样的操作，必须保证析构函数为虚函数。

operator=需要规避右值是自己的情况。

不使用多重继承。

## 智能指针使用规范

* 智能指针是一个持有动态分配Object的指针的Object，其行为和C++的裸指针十分相似，且可在适当的时候自动delete所持有的Object，智能指针非常有助于在发生异常的时候确保销毁动态分配的对象。另外，在多个Owner之间共同持有一个Object时，也十分有用。从概念上来讲，智能指针就像一个指向Object的指针，并负责在不需要的时候delete掉指向的Object。
* 智能指针是一个对象，其主要作用是适时自动释放，异常安全。
* 禁止智能指针和裸指针混合使用。
* 禁止获取智能指针内含的裸指针，禁止使用智能指针的get、reset接口。
* 禁止使用裸指针变量初始化智能指针，建议使用boost::shared\_ptr<TypeName> sp(new TypeName)的形式初始化。
* 智能指针的类型转换使用（static/const/dynamic)\_pointer\_cast<T>的形式
* 智能指针可作为函数参数，传值即可不使用传引用。另外，一般不使用智能指针的引用。
* 智能指针可作为函数返回值，但必须是值返回而不是引用返回，时刻记住智能指针是一个对象。
* 智能指针可作为标准容器的元素。
* 智能指针相对裸指针在时间、空间效率上略有下降，但是在可接受范围内。
* 智能指针判空方式是if(ptr)或者if(!ptr)，或者跟nullptr判断。
* 智能指针变量的命名，建议是xxxPtr，不建议为pXXX。

## STL使用规范

* 容器外部不能以指针或引用形式引用容器内元素，如果必须这样，使用指针或智能指针作为容器元素。
* 在遍历容器的过程中，做出改变容器结构的操作（比如添加、删除）后，当前迭代器可能失效，必须重新获取有效的迭代器。
* 重载operator<操作符时，元素相等的的情况，必须返回false。
* 服务器端原则上，只有Routine底层才能使用STL。
* 总体原则是尽量不用，除非是能带来极大开发效率的提升。
* 一般STL用于长期存在对象的成员变量，例如服务器、场景，其他情况不要使用。

## CommonData、CommonFlag使用规范

CommonData和CommonFlag是公共资源，用于小量级的存储，避免为单独功能增加存储而使得逻辑复杂。

拿到ID之后需要在代码中增加这个枚举的定义（服务器是必须加的，客户端C#和lua中看使用情况增加必要枚举定义）。

## 服务器ObjID注意事项

* 服务器每个Obj有一个id，玩法常用记录此ID来在场景中找到这个Obj。
* 这个id只有在一个场景中才是有效的，每个场景内ObjID都是独立的，可能存在重复。所以使用ObjID时，要确定操作只在单场景内。
* 对一个Obj进行删除操作时（DelFromScene），这个Obj在下一个心跳才会删除，注意判断Obj是否有效使用GetActive。比如拾取掉落包消息，如果客户端连续发多个消息，如果没有判断active可能造成刷物品。

## 其他注意事项

* 1. switch语句中所有的case必须有break。
  2. 变量使用前必须进行初始化。
  3. 用宏定义表达式时，要使用完备的括号，将宏定义的多条表达式放在大括号里。
  4. 除法和求模运算时，对于除数变量需要做非0判断。
  5. 类型装换前必须进行有效性判断，不允许直接强转类型，优先使用C++的转型风格，禁止使用const\_cast和reinterpret\_cast。
  6. 尽可能使用const。
  7. 使用前置声明，避免在头文件中包含过多文件。
  8. 向下类型转化前，必须判断目标的类型，确定可以转换再转换。
  9. 使用模板还是面对对象的设计模式，关键看不同类型对行为是否产生影响。
  10. 消除所有的warning。
  11. 注意溢出错误、除0错误、数组越界。
  12. 不得使用std::vector<bool>，使用solar::bit。
  13. 可以使用solar::string代替char[]。
  14. 取当前时间使用solar::ansitime\_sec和solar::ansitime\_milsec，分别是取当前秒和毫秒。
  15. 小心使用dynamic\_cast，用于类层次间的上行转换和下行转换，下行转换时具有类型检查的功能。只能用于具有虚函数的类，否则会编译报错，这是因为运行时类型检查需要运行时类型信息，而这个信息存储在虚函数表中。
  16. 注意服务器在处理CG消息包的时候，会直接拿到客户端发来的数据。这些数据需要经过校验才能使用，恶意发送的数据不经过校验可能会造成巨大影响。
  17. 注意服务器在处理玩家利益相关操作的时候，需要先处理扣除，再处理给予。防止扣除不成功，但是直接给予利益的情况。
  18. 独立的功能应该在Config文件中增加开关，可以在线动态控制功能的运转，控制力度要跟策划商定。可能也会包含客户端的屏蔽。
  19. 注意功能是否涉及跨天、跨周、跨月、跨年、跨上下线等，多注意考虑极端情况。
  20. 可使用金钱兑换到的货币（比如元宝）禁止各种途径的放出（寄售行类除外）。
  21. 注意服务器避免短时间内大量复杂操作（比如某些全服操作，大循环，大量玩家操作并发），可能会影响性能。
  22. 尽量避免玩家利益回滚操作，可能造成刷钱的漏洞。

# 功能自查

功能是否有开关

功能需要存库的数据已正确存储

是否不相信客户端

是否先删后给

stl erase循环删除是否正确

副本逻辑玩家断线重连是否正常

副本复用是否初始化了各种状态

不能使用递归，确需使用时提出公议

向下类型转换是否先判断了类型

日志是否完整

大世界如何处理

# 服务器处理CG消息包的逻辑

* 将CG消息包的发送者当做恶意攻击、欺骗的人，不能信任CG消息包中的内容
* 将CG消息包的内容当做精心伪造，想要不当获益或攻击服务器的内容
* CG消息包一般只能发送行为数据（即客户端告诉服务器自己要干什么），不能发送结果数据（结果由服务器根据客户端的请求运算得出）
* 具体常见需要注意的情况：
* CG消息包中的变量，需要做合法性检查
* CG消息包中的变量，作为数组下标的必须严格判断范围
* CG消息包中的变量，作为循环上限的，必须设置额外的上限判断，防止死循环
* CG消息包中的变量，类似金钱金额，某种东西的数量之类的变量，必须严格判断非负等情况
* CG包涉及频繁操作的，需要考虑效率问题，比如设置玩家冷却操作锁
* CG包是否有效率问题，比如发一个包导致服务器进行大量运算或者大量读存数据等
* CG包理论上不发自己的GUID，服务器可以取到自己的GUID，处理函数中也不能依赖于客户端发来的自己的GUID。 可以发其他玩家的GUID。
* CG包涉及打日志的，应评估日志量的大小
* 其他注意事项：
* 如果处理CG消息包的逻辑是将其转发给其他玩家或Routine，需要追踪过去检查，确保不遗漏
* 如果CG消息包涉及到给玩家收益的话，同时检查其逻辑是否遵循“先删后给”原则

# 服务器质量保障

1. 增量检查

2. 专项功能检查

3. 多种工具检查

4. CG消息包

5. erase检查