```
对文件和标准输出写入而言,各语言的标准库均在用户空间开辟了缓冲区,写入数据时将首先写入至缓冲区中
                                                                           不管是在文件、终端还是通过流的形式发送给sockt并通过网络传输至日志服
                                                                           务器,都绕不开一个内容: 缓冲
                                                                                                                                     而对缓冲区写操作的函数基本上都是非线程安全的,在多线程处理中,完全有可能出现相互覆盖的情况
                                                                                                                                     所以,必须使用互斥量来同步不同线程之间的写入,使得并行的写入串行化
                                                       以日志输出功能为例
                                                                                                                                 假设有实例A、B、C,如果使用成员变量的互斥量,那么实例将持有不同的互斥量
                                                                           使用互斥量解决了缓冲区并发写入的问题,那么,互斥量在哪里分配? 🔾
                                                                                                                                 使用静态变量(类变量)分配可以解决此问题,但是依赖于语言的实现
                                            节省资源
                                                                           不管是否使用单一实例,并发安全问题均需要考虑: 所有实例应持有同一个互斥量 〇 既然如此,使用单例模式可节省更多的内存以及节省实例创建的开销
                  为什么要使用单例模式
                                                                        在容器化应用中,程序从环境变量中读取配置,并封装成自身语言级别的类对象
                                                       以配置信息为例 🤤
                                                                        配置信息全局中存在一份即可,超出一份没有必要且浪费内存空间
                                                                                                                                            ratic Singleton getInstance() {
n instance;
                                                      利用Java类加载线程安全的初始化静态变量特性实现,但无法实现运行时初始化(或者说,懒加载) 🗿
                                            饿汉式 😑
                                                      对于需要很长时间初始化的单例类,可以选择饿汉式实现,将耗时时间很长的任务在应用提供服务之前完成,个人认为是个更好的选择
                                                      利用线程同步的方式在运行时加载,由于每次获取单例对象时均需要加锁解锁,会造成较大的性能影响,所以应用不是很多 😊
                                                                       eton {
: Singleton instance;
                                            懒汉式 ⊖
                                                                : static synchronized Singleton getInstance() {
f (instance == null) {
  instance = new Singleton();
单例模式
                                    Java
                                                                  基于懒汉式实现,但并不是在每次获取实例时均使用同步,而仅是在初始化时可能出现的并发进行同步 ◎
                                            懒汉式的改进: 双重校验 ⊖
                                                                  低版本的JDK会有指令重排的问题,所以需要给 instance 成员变量添加 volatile关键字,以禁止指令重排
                                                          Java类加载的时机包括: 创建类的实例,调用静态成员函数或获取静态变量,初始化某个类的子类,或者使用class.forName()等反射机制获取类信息
                                                          静态内部类正是利用了JVM在进行类加载时,保证类的唯一以及线程安全性等特性实现的
                                            静态内部类

      Singleton getInstance() {

      ogletonHolder.instance;
      /* 获取静态变量时JVM加载类,并保证线程安全

                                            使用枚举类 💿 枚举类的枚举类型本身为 public static final 类型,只不过JDK对其进行了封装,所以本质上也是利用类加载机制实现的,实现上更加的简洁 🔘
                                                                instance = None
lock = Lock()
                                                                 def __new__(cls, *args, **kwargs):
    if cls.instance:
        return cls.instance
                                                                   with cls.lock:
    if not cls.instance:
        cls.instance = object.__new__(cls.instance)
                                                                                                                    __new__方法在Python中为一个类的构造方法, 默认返回一个类的实例。 而__init__方法则是在该实例上进行属性的添加
                  实现单例模式
                                                                                                             instances = {}
lock = threading.Lock()
                                                                                                             @wraps(cls)
def get_instance(*args, **kwargs):
    if cls not in instances:
        with lock:
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
    return instances[cls]
return get_instance
                                              双重校验实现 😊
                                                             有时候不想在原类上进行修改,也可以使用装饰器来实现 😑
                                    Python
                                                         singleton_class = Singleton()
                                                           om singleton import singleton_class
                                             import \Theta
                                           相较于Java和Python,Go要更为特殊一些: 没有类的概念,只有结构体
                                           sync.Once 

一次性初始化,不管有多少个协程同时调用,只会执行一次注册的函数 

类似于POSIX线程中的 pthread_once
                                               config *Config
                                               AppId string
AppSecret string
                                   Go ⊖
                                              inc GetConfig() *Config {
  once.Do(func() {
    config = &Config{}
    config.AppId = os.Getenv("appId")
    config.AppSecret = os.Getenv("appSecret")
}
```

当应用使用容器部署时,若所有的单例模式使用初始化时加载,可能在一瞬间超出容器所限制的CPU使用率, 而初始化时间较长的实例又需要应用启动时即加载。所以,到底使用何种单例方式,与容器资源相关、与业务逻辑相关 不能单一而论