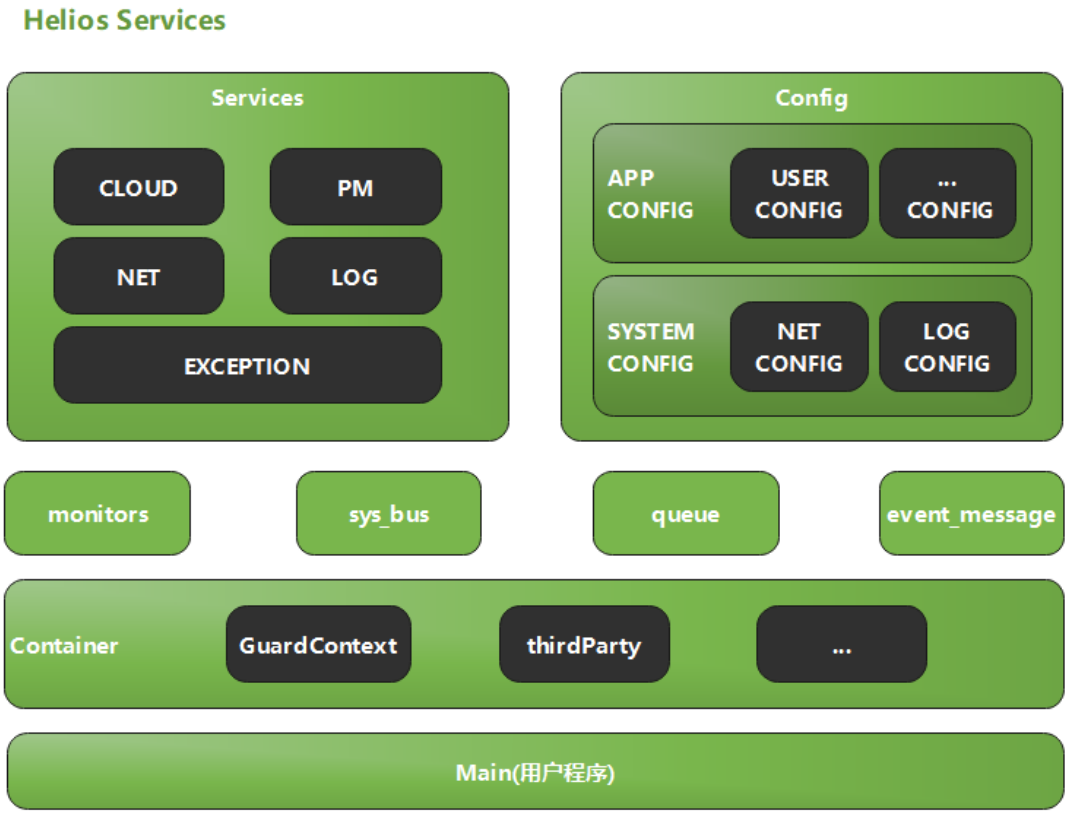
Helios Services指南(3)-高级

Helios Services高级概述

此篇主要说的是, helios services的软件设计的思考, 和后续将要支持的功能, 以及通过我们软件设计知道内部的通信原理, 了解和理解内部的机制等等...

Helios Services软件架构设计

架构设计



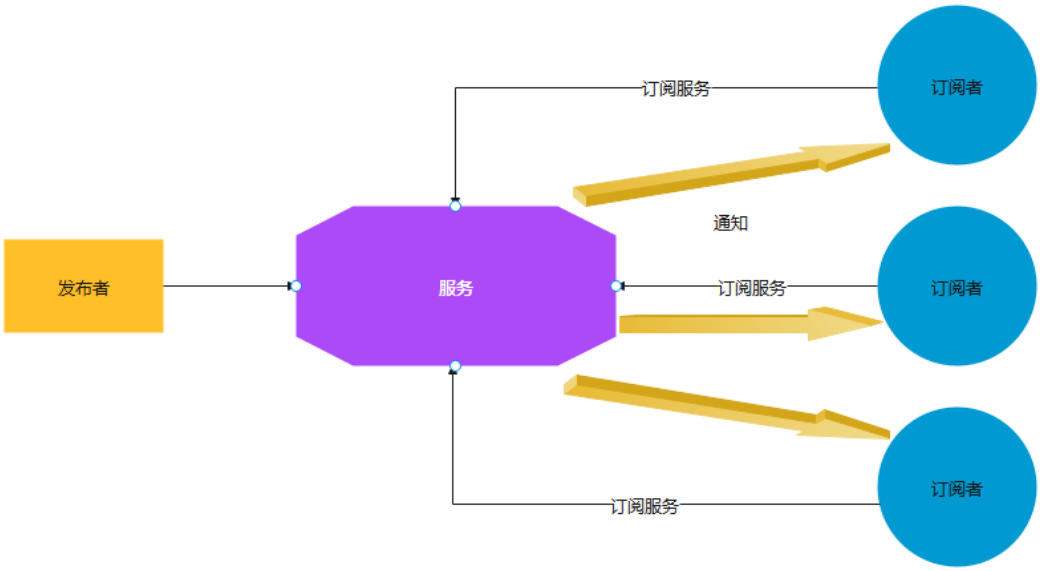
名词解释:

- `Services`: 服务组件
- `Config`: 配置存储
  - `app_config`: 存储用户配置文件的目录
  - `system_config`: 系统配置文件的目录
- `monitos`: 监控器,监控各个服务的状态
- `sys_bus`: 会话的处理总线主要处理, 生产者和消费者模型, 目前支持异步模式
- `queue:` 普通队列, 用于线程间通信

- `event_message`: 异步的带有命名空间的高级消息队列
- `GuardContext`: 上下文容器, 统一管理所有的服务和配置, 还有监控等组件, 提供服务, 配置, 监控等获取的接口
- `third_party`: 三方组件目录, 拥有我们提供和封装的一些组件以便客户敏捷开发和二次开发

设计原理

服务设计原理



1. 我们将写好的一些系统级别的提供了一些服务这些服务的发布者可能来自, 系统底层或者用户发布等
2. 用户只需要订阅网络服务, 当网络信号不好或者, 断网服务会发起自动重连,并且通知所有订阅的者函数, 通知各种状态
3. 例如日志服务等是需要用户自己承当发布者的作用的
4. 发布者和订阅者是毫无关系的, 发布者无需关注订阅者, 只用把发布的数据给用户即可
5. 发布者发布到服务是, 服务再到订阅者是异步的, 我们同时也提供了同步的配置和支持
6. 服务的中间本质实现是一个, `带命名空间的高级的消息队列`+其他一些组件来组成的

优点:

- 我们替客户实现好了一些系统级别的服务, 并支持后续迭代, 解决了客户客户还需要维护系统接别的代码问题, 只提供对外的API供客户调用
- 降低了业务之间的耦合, 订阅者和发布者之间的解耦
- 订阅者,只需要订阅服务, 或者使用服务, 既能使用我们维护并提供的功能

缺点:

- 我们限定死了服务的类型, 客户无法的到定制, 比如我现在有个需求是, 想自己业务上有两个或者多个业务解耦, 客户无法实现自己的服务, 这些服务都是我们写好的, 解决方案`sys_bus`
- 首先我们是由一个高级消息队列组成, 所以有可能存在服务还在, 但是消息队列崩溃的现象, 所以我们在服务崩溃的时候给他拉起来, 解决方案`monitor`

示例:

订阅服务

```
>>> from usr.bin.guard import GuardContext
>>>guard_context = GuardContext()
>>> guard_context.refresh()

[    OK      ] create sys monitor net service
[    OK      ] create sys monitor log service
[    OK      ] create app monitor media service
[    OK      ] create app monitor exception service
[  FAILED  ] load cloud monitor error reason:[cloud service load error]
[    OK      ] create app monitor pm service
>>> def func(*args, **kwargs):
...     print("func args = {} kwargs = {}".format(args, kwargs))
...
...
...
>>> net_ser = guard_context.get_server("net")
>>> net_ser.subscribe(func)
>>>
func args = ('anonymous',) kwargs = {'message': {'message_id': 2, 'sender': 'anonymous', 'message': {'IPv6_DNS1': '::', 'IPv6_DNS2': '::', 'ip_type': 0, 'sim_status': 1, 'net_status': 0, 'datacall_status': 0, 'profile_id': 1, 'IPv4': '0.0.0.0', 'IPv4_DNS1': '0.0.0.0', 'IPv4_DNS2': '0.0.0.0', 'IPv6': '7f0:2ab9:8a48:4069:7047:8949:20:4861'}, 'from_event': 'NET', 'msg_type': 1}}

func args = ('anonymous',) kwargs = {'message': {'message_id': 3, 'sender': 'anonymous', 'message': {'IPv6_DNS1': '2409:8030:2000::1', 'IPv6_DNS2': '2409:8030:2000::2', 'ip_type': 0, 'sim_status': 1, 'net_status': 1, 'datacall_status': 1, 'profile_id': 1, 'IPv4': '10.199.37.35', 'IPv4_DNS1': '211.138.180.2', 'IPv4_DNS2': '211.138.180.3', 'IPv6': '2409:8930:4b3:4817:4d1b:742e:2f03:7498'}, 'from_event': 'NET', 'msg_type': 1}}
```

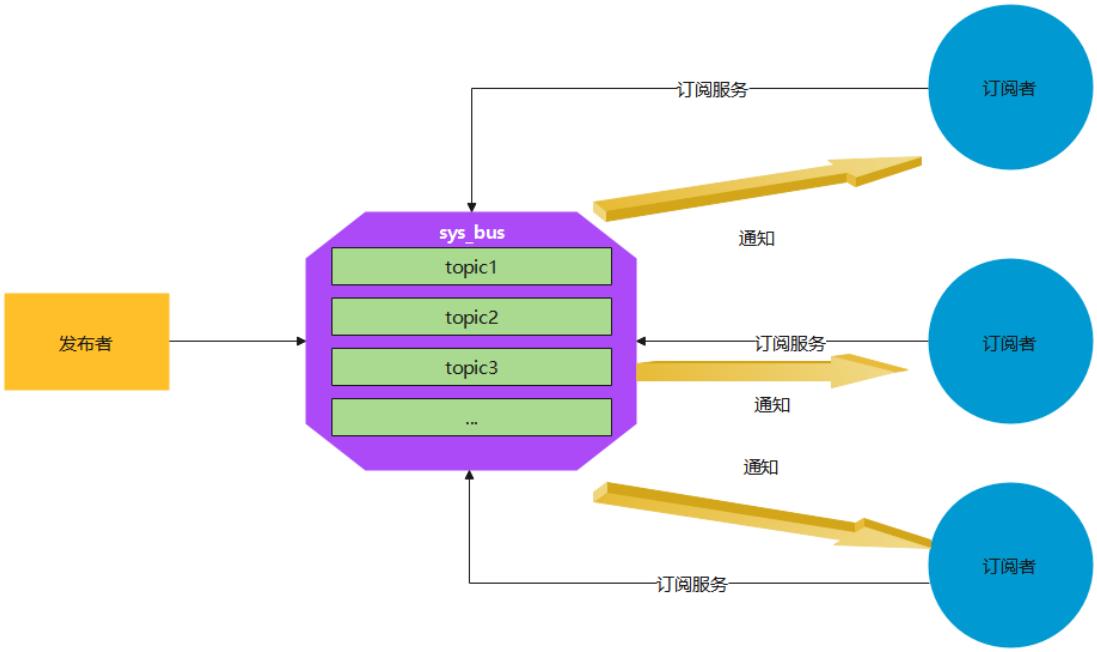
获取网络服务

func去订阅网络状态的

网络失败情况下的通知

网络从连接失败后又重新连接的通知

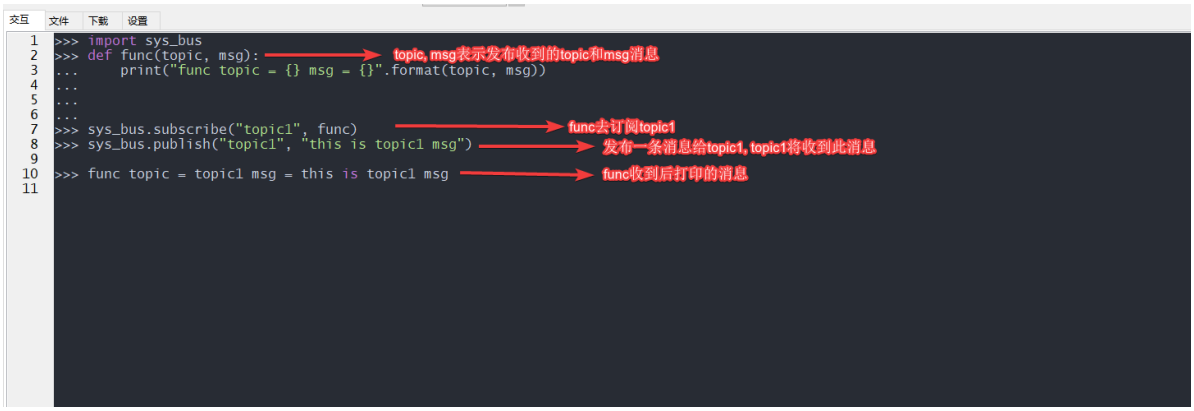
sys_bus实现原理



上面服务是我们通过服务的形式来给用户体现的, 缺点中我们也明确了，用户无法定制服务, 反之sys_bus即可实现用户定制服务的需求

- 异步的模式, 用户可以订阅和发布,还有解绑服务
- 将发布和订阅分离, 模块化分离, 支持多个发布者, 和服务的区别是,这里发布者和订阅者都是用户去维护的
- 用户只需要关注topic而不用关注sys_bus等本身和订阅者本身
- 优点:
    - 客户可以自己定制和发布相关的数据等
    - 我们帮客户维护了sys_bus的稳定性
- 缺点:
    - 过于灵活可能会导致, 用户的一些订阅代码需要用户自己去维护

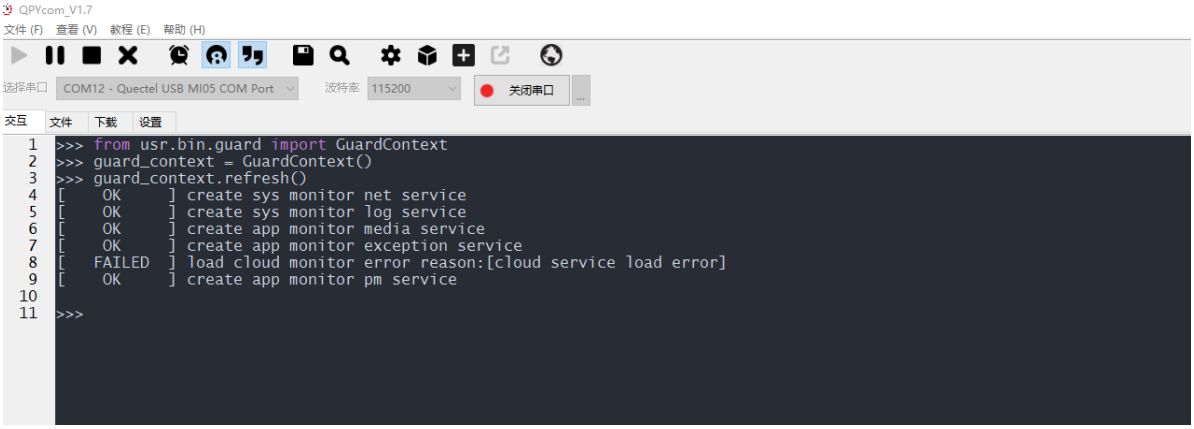
示例:



monitor设计原理

- 负责监控每个服务在每隔15秒给服务一个心跳, 如果, 服务, 收到这个心跳包, 会将这个发给的指定的接收人的, monitor接收到心跳包, 即认为, 服务运行正常
- 可以设置, 当服务几次运行失败后执行什么样的行为, 比如说失败三次后执行, 重启设备或者

示例

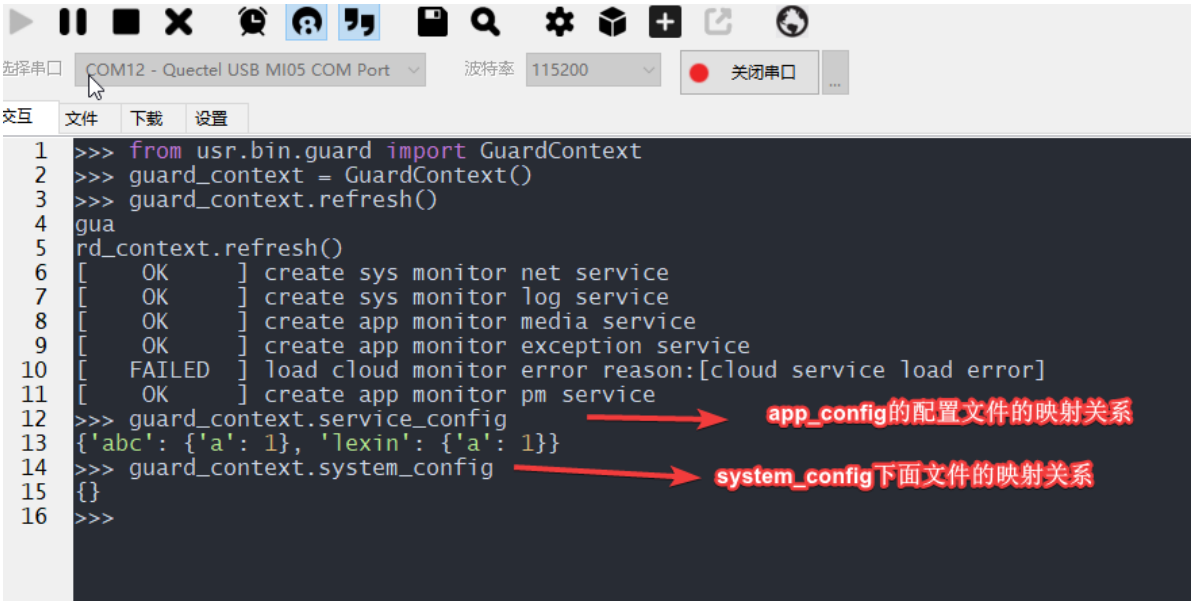


config设计原理

为了方便配置文件的读取和设置等, 特此讲配置文件路径统一到config下面设置按照下列规则的config.json文件

- app_config
  - 用户层级的配置文件设置, 用户按照一定的文件夹格式, 我们将会自动读取里面的配置文件, 在组织进容器中, 以便用户进行获取, 相关接口参考, 请参考`API文档`
- system_config
  - 系统级别的配置我们会按照系统级别的配置文件作为提供对外, 用于区分和读取, 因为后期我们有可能会提供相应的配置文件, 和相关服务

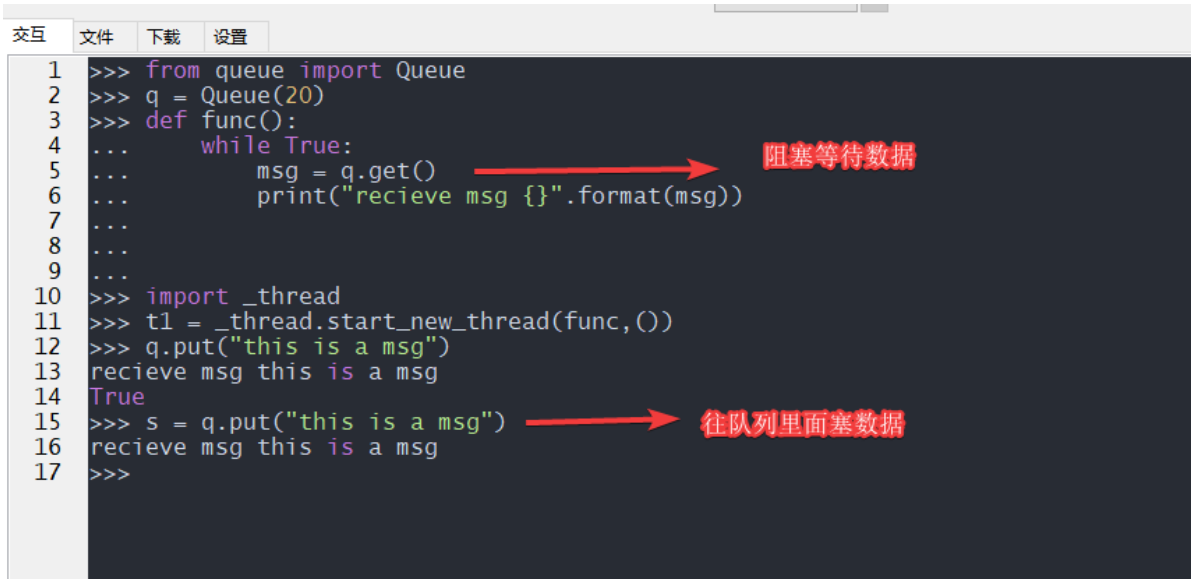
示例:



- 我们可以看出上面app_config下面存在
  - lexin
    - config.json
  - abc
    - config.json
  - 我们可以看到读取的service_config下面
    - {'abc': {'a': 1}, 'lexin': {'a': 1}}
    - abc和lexin就是我们的文件夹名字, 对应的value就是我们config.json中的内容
- system_config也是通app_config的对应关系一致
- 注意: 如若想读取必须按照此类调教key是文件夹名,value读取的是config.json中的内容书写

queue设计原理

普通队列用作消息通信, 创建队列后获取数据的时候会阻塞, 当有数据信号的时候将会被唤醒



三方组件和容器设计

GuardContext设计原理

上面说了那么多组件设计后, 我们说下, 我们说下容器设计

- 因为目前存在很多服务, 很多配置, 为了方便客户不用这导入一下那导入一下, 并且做统一化管理, 特此容器化了所有组件
- 用户只需要关注容器设计即

ThirdParty设计

pass