混沌工程技术方案

文档历史

修订日期	修订内容	修订版本	修订人
2022-04-12	创建文档	v0.1	liangming.huang@shopee.com
2022-04-18	补充故障注入实现细节及数据收集埋点细节	v0.2	liangming.huang@shopee.com
2022-04-28	根据评审意见修改故障注入和权限管理相关内容	v0.3	liangming.huang@shopee.com

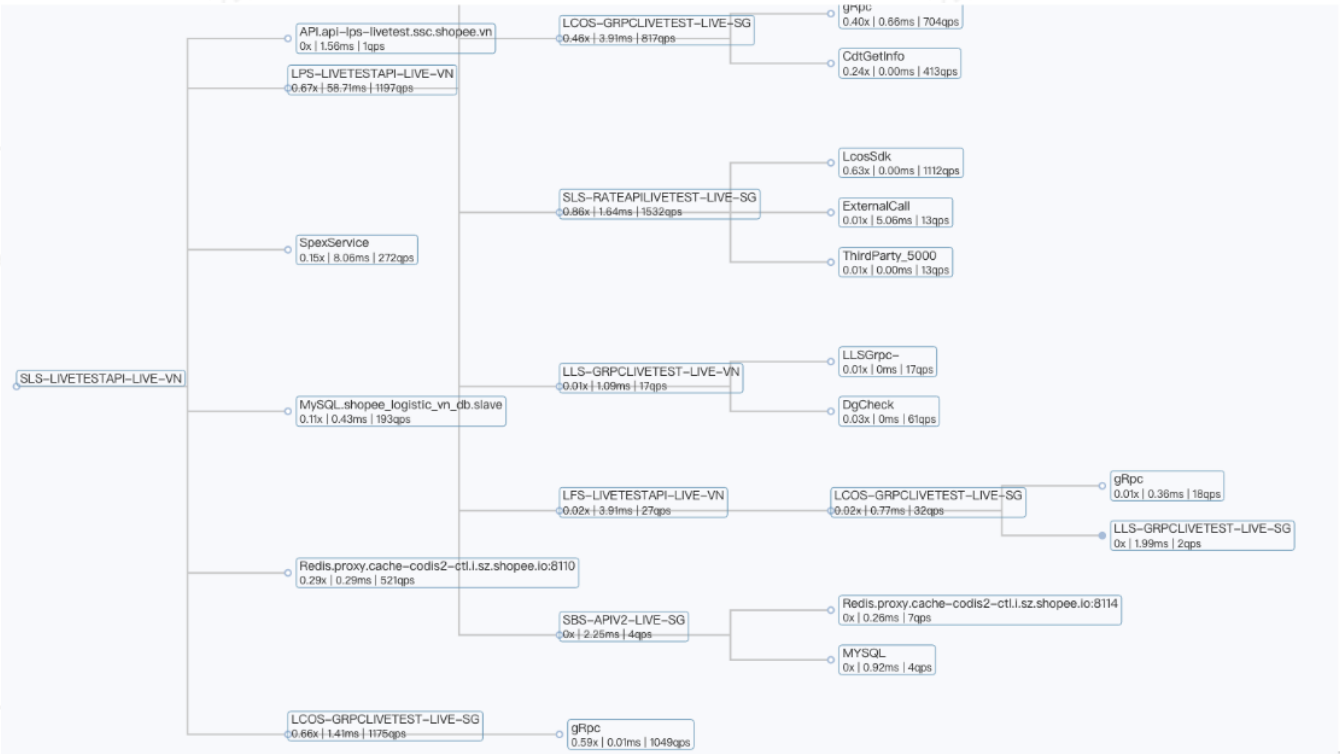
摘要

编写目的

分析混沌工程的实现，为混沌工程开发和测试提供技术参考。

项目背景

随着服务拆分，系统链路变长，系统可能出错的地方也随之增加。一个常见的系统链路如下：



本项目整理集成了常见问题检查，为系统稳定性评价提供判断依据。

参考资料

混沌工程(Chaos Engineering) 总结

阿里巴巴混沌测试工具ChaosBlade

字节跳动混沌工程实践总结

混沌工程原则

Golang 常见性能问题总结

在Go中使用Failpoint注入故障

任务概述

对于混沌工程，主要需要实现以下功能：

任务	描述
实验场景分析	按场景的维度进行混沌工程实验
混沌工程实验	实施混沌工程，比如对系统部件注入故障
混沌工程实验结果分析	验证实验结果，发现系统薄弱部分

规范与约定

1. 代码规范参考：代码及开发规范
2. 应用分为接口层、应用层、核心领域层和支撑层，按照CQRS原则可酌情合并应用层和核心领域层
3. 各层间定义DTO进行参数传递和数据返回
4. 数据的持久化必须在Repository中以便后续的数据层重构
5. 对外API返回采用retcode、message、data形式
6. 不允许随意向context添加参数进行隐式传递，必须经过审核阐述必要性

术语与缩略语

缩略语/术语	全称	说明
链路		系统中业务逻辑从开始到结束所需要经过的子系统组成的调用链路
混沌工程		由netflix工程师提出的，旨在随机注入故障来验证系统稳定性的工具
CKS	chaos knight system	混沌工程系统，本混沌工程服务命名 chaos knight 即“混沌骑士”
CKC	chaos knight client	开放给服务集成进行埋点上报信息的客户端，与cat client 类似，用于收集应用数据信息
CKA	chaos knight agent	和应用部署在同一个机器的代理，与cat agent 类似，用于实施故障注入和应用数据信息回传服务端
CKH	chaos knight home	服务端，管理故障演练，故障编排、实验结果分析和生成报表等
目标服务		进行混沌实验的服务

系统分析设计

系统设计目标

描述系统设计的目标，比如*pv/uv*，*tps*，容量，预估数据量，并发等，系统分析设计将以此作为目标展开。

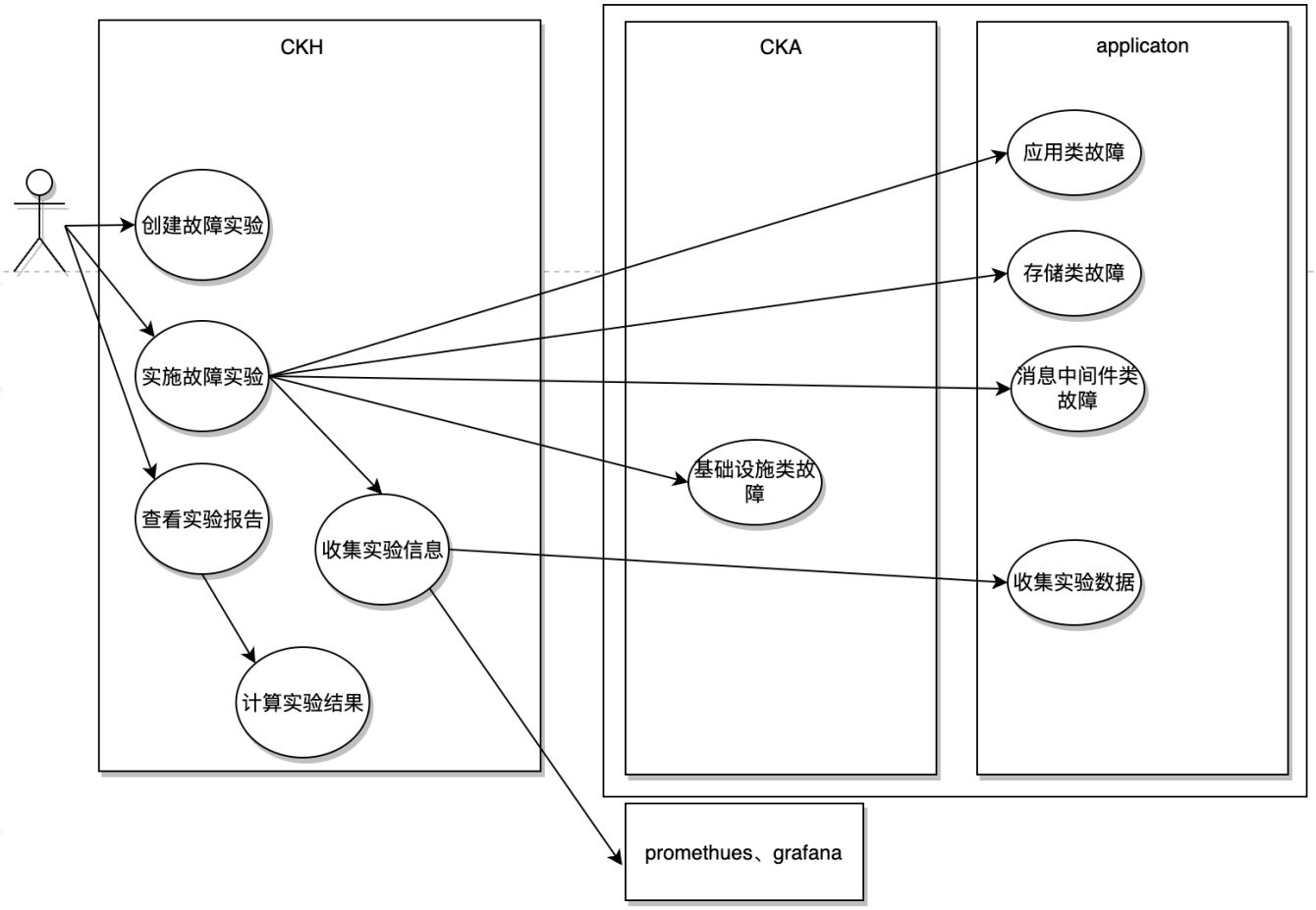
1. 故障容忍：CKS任何组件故障，都不能影响目标服务的正常运转
2. 全量数据：数据收集模块收集全量数据

总体架构分析

混沌工程是大促系统性能分析工具的一部分，大促系统架构如下：

其中混沌工程系统分为三部分：

用例分析



故障抽象

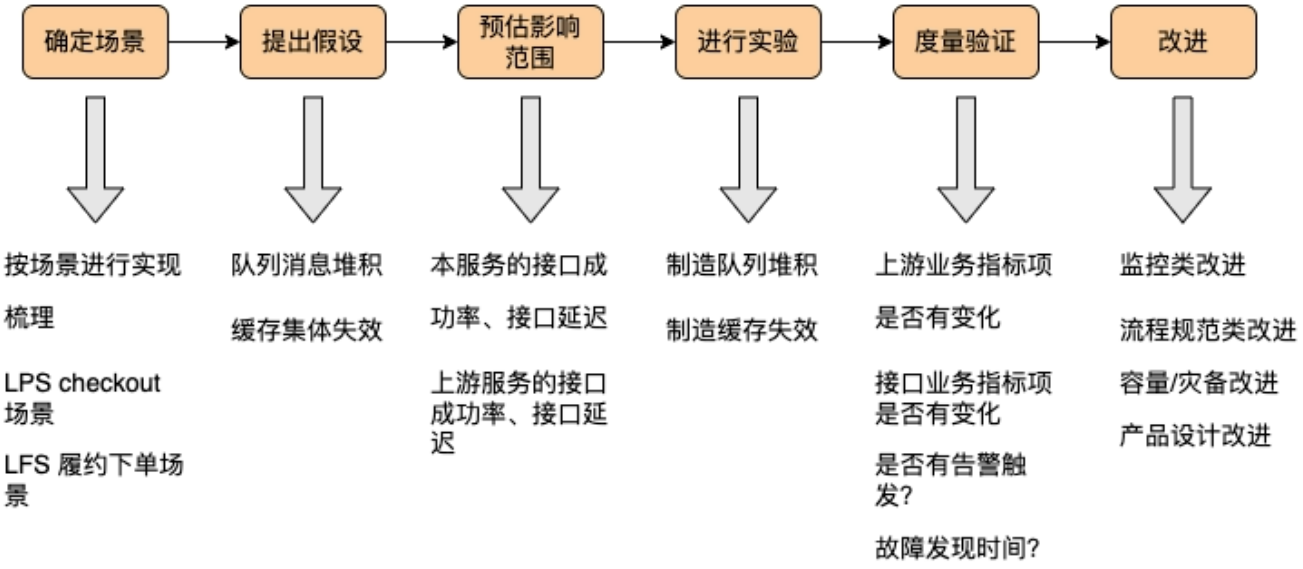
应用类	存储类	基础设施类	消息中间件
应用下线	数据库宕机	高IO	消息堆积
接口限流	连接池占满	高CPU	消息重复
内存占满	慢查询	网络延迟	消息丢失
...	数据库热点	网络丢包	...

如上图所示，混沌工程实施主要分为三步：根据场景创建故障实验、注入故障进行实验和实验结果分析输出报告。

核心业务规则

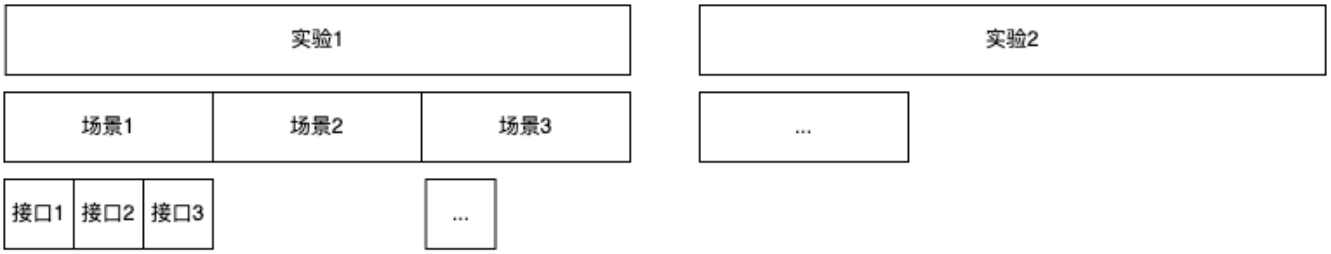
下面具体描述混沌工程各项用例的实现。

故障实验流程

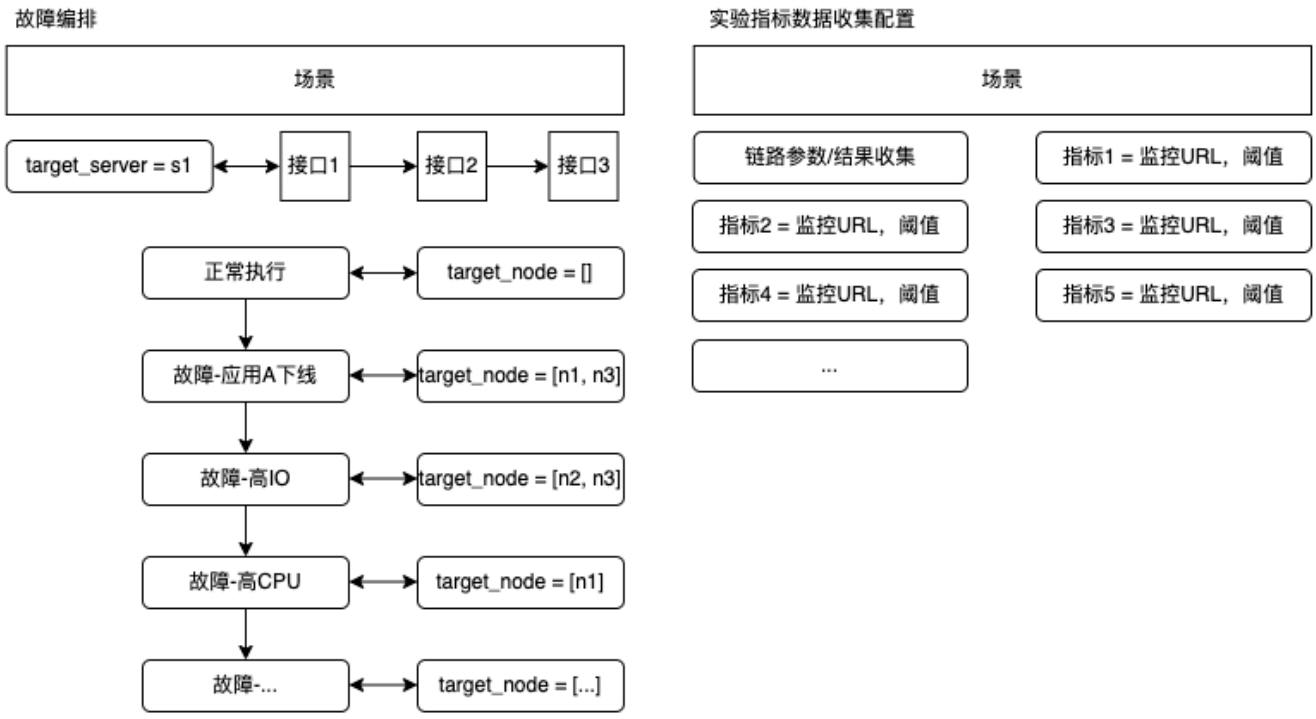


创建故障实验

每次进行实验的时候都必需确认好实验的场景和关注的指标。场景实质上是一组业务性质相近的接口的集合，一次可以实验多个场景。层次结构如图：



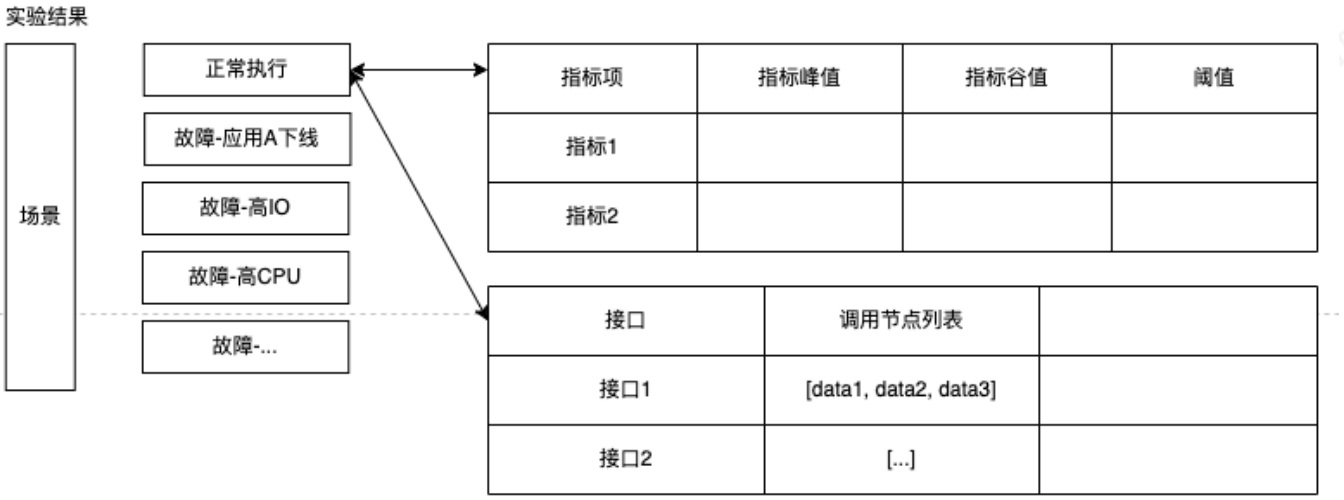
确定好场景后，需要为场景配置好实验的故障和关注的指标。



故障会根据链上的节点一个一个执行，每个节点上的执行详情为：记录开始执行时间---触发故障---执行接口1---执行接口2---执行接口3---恢复故障---记录结束时间---根据时间段查询各指标的信息---收集链路参数/结果信息—下始下一个节点的故障实施。其中执行接口可以执行一次请求也可以执行多次，因为有些问题可能在高并发下才出现。

故障执行理由状态是系统自动执行及恢复，但不排除系统无法实现的情况，所以会有人工介入创造故障的情况。

实验结果信息记录结构为：



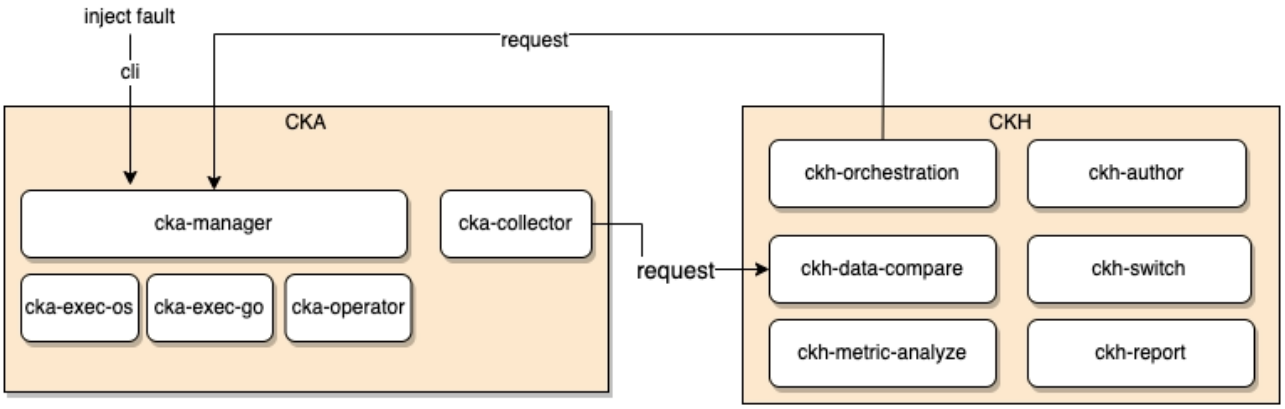
Data

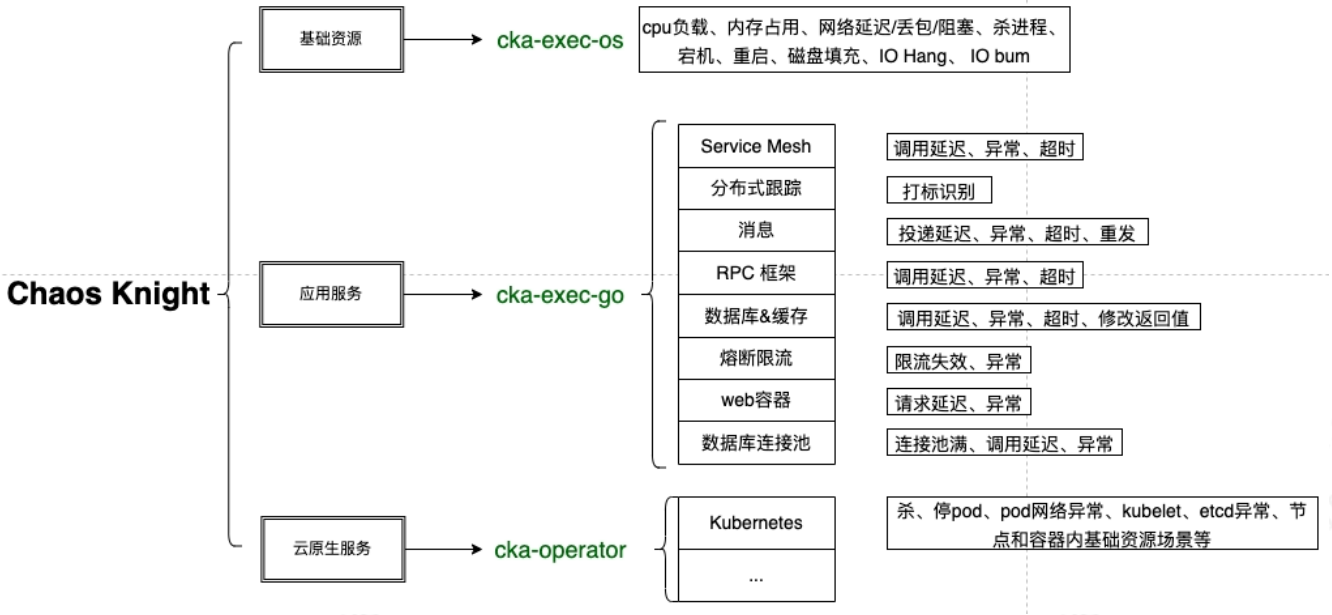
{module_name, interface_name, status, input, output}

故障注入

故障注入主要由CKA和CKC实现，CKA和catagent 类似，和目标服务部署在同一个机器，跟随服务启动而启动。CKA对外提供http请求方式进行故障注入。

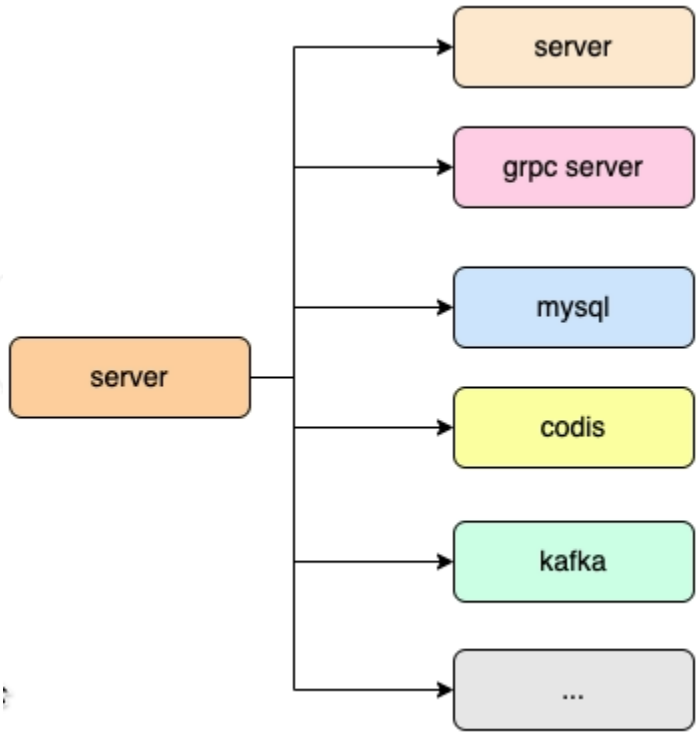
基本架构如图：





由于CKA和目标服务平行部署，只能注入服务级别(端口级别)的故障。要实现接口或是更细粒度(代码级别)的故障注入，需要使用到CKC。CKC是混沌工具提供给服务埋点使用的底层SDK，它除了可以埋点收集实验数据(下面实验结果收集在说明)，还可以埋点注入故障。

SDK提供通用的埋点接口，目标服务只需要在服务调用，数据库调用或是中间件调用处设计埋点方法即可， 如下图：

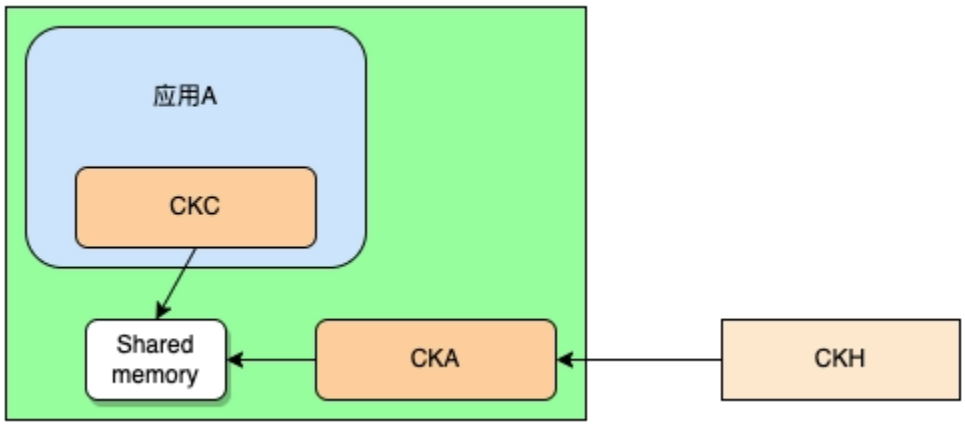


也可以提供包装的http(grpc) client 或是数据库client 供业务使用。该类可以最终可以通过以chassis通用统一组件的方式提供到业务代码中使用。

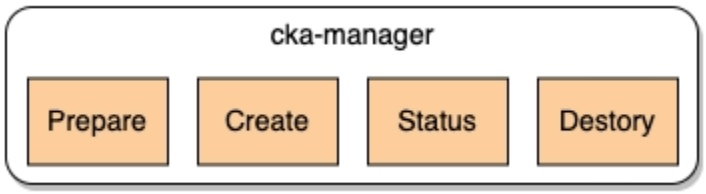
埋点代码在正常流程中不起作用，故障注入触发条件时才起作用：

```
if _, _err_ := failpoint.Eval(_curpkg_( name: "bananaDelay")); _err_ == nil {
    time.Sleep(time.Second * 2)
}
```

故障注入信号通过共享内存实现。在故障注入实验是，通过CKH将信息发送给CKA，CKA将故障注入信号写到共享的内存中，应用在工作时通过读取共享的信息数据是否触发故障代码。内存判断通信架构如下：



cka-manager 主要是对外接口和管理调度具体故障注入实现，它内部区分如图：



其下层cka-exec-os、 cka-exec-go、 cka-operator等是根据不同的场景进行故障注入实现的模块分类。

以下整理了一部分故障注入具体实现方式：

类型一	类型二	故障	实现方式一（非侵入式）	实现方式二（侵入式）
基础资源	cpu	cpu负载	写个程序让cpu一直运行	-
	磁盘	磁盘填充	使用 dd 命令： dd if=/dev/zero of=/chaos_filldisk.log.dat bs=1M count=1000 iflag=fullblock	-
	内存	内存填充		-
	IO	读IO负荷	使用 dd 命令： dd if=/dev/sda1 of=/dev/null bs=1M count=1024 iflag=dsync,direct,fullblock	-
		写IO负荷	使用 dd 命令： dd if=/dev/zero of=/tmp/chaos_burnio.log.dat bs=1M count=1024 oflag=dsync	-
	网络	延迟	使用 tc 命令： tc qdisc add dev eth0 root netem delay 300ms 10ms 30% [tc qdisc add dev eth0 parent 1:4 handle 40: netem delay 300ms 10ms && \ tc filter add dev eth0 parent 1: prio 4 protocol ip u32 match ip dst 127.0.0.1 match ip sport 8090 0xffff flowid 1:4 && \ tc filter add dev eth0 parent 1: prio 4 protocol ip u32 match ip dst 196.168.1.2 match ip dport 9090 0xffff flowid 1:4]	-
		丢包	使用 tc 命令： tc qdisc add dev eth0 root netem loss 10%	-
		重发	使用 tc 命令： tc qdisc add dev eth0 root netem duplicate 80%	-
		损坏	使用 tc 命令： tc qdisc add dev eth0 root netem corrupt 20%	-
		乱序	使用 tc 命令： tc qdisc change dev eth0 root netem delay 10ms reorder 25% 50%	-
应用服务	Service Mesh	延迟		使用代码埋点
		异常		使用代码埋点
		超时		使用代码埋点
	消息队列	堆积	人工停止saturn,先不让消息被消费	使用代码埋点
		重复	使用 tc 命令： tc qdisc add dev eth0 root netem duplicate 80%	使用代码埋点
		丢失		使用代码埋点
	数据库	延迟	"使用 tc 命令： tc qdisc add dev eth0 root netem delay 300ms 10ms 30% [tc qdisc add dev eth0 parent 1:4 handle 40: netem delay 300ms 10ms && \ tc filter add dev eth0 parent 1: prio 4 protocol ip u32 match ip dst 127.0.0.1 match ip sport 8090 0xffff flowid 1:4 && \ tc filter add dev eth0 parent 1: prio 4 protocol ip u32 match ip dst 196.168.1.2 match ip dport 9090 0xffff flowid 1:4]"	使用代码埋点
		异常	tc 中断或损坏	使用代码埋点
		超时	延迟的值非常大，大于设置的超时时间	使用代码埋点

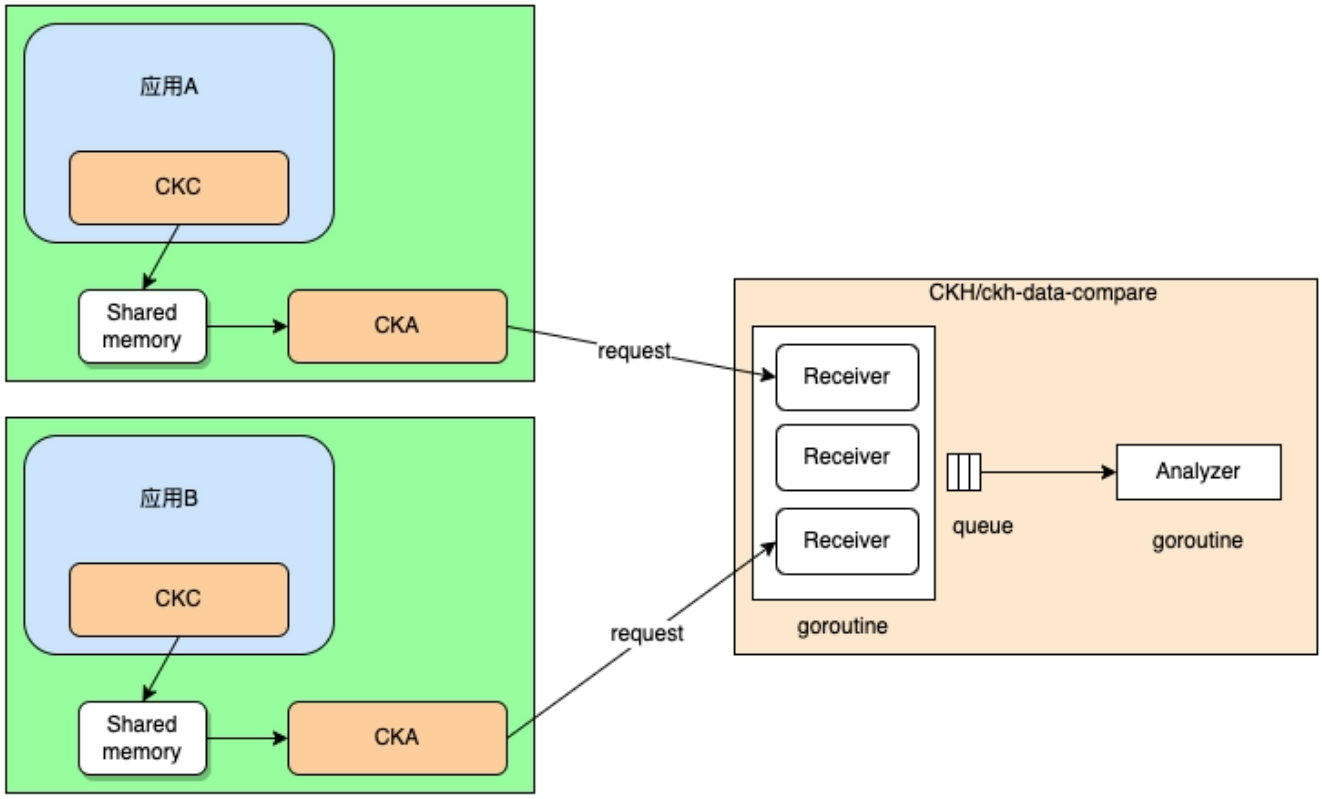
netem 与 tc:

netem 是 Linux 2.6 及以上内核版本提供的一个网络模拟功能模块。该功能模块可以用来模拟出复杂的互联网传输性能,诸如低带宽、传输延迟、丢包等等情况。使用 Linux 2.6 (或以上) 版本内核的很多发行版 Linux 都开启了该内核功能。

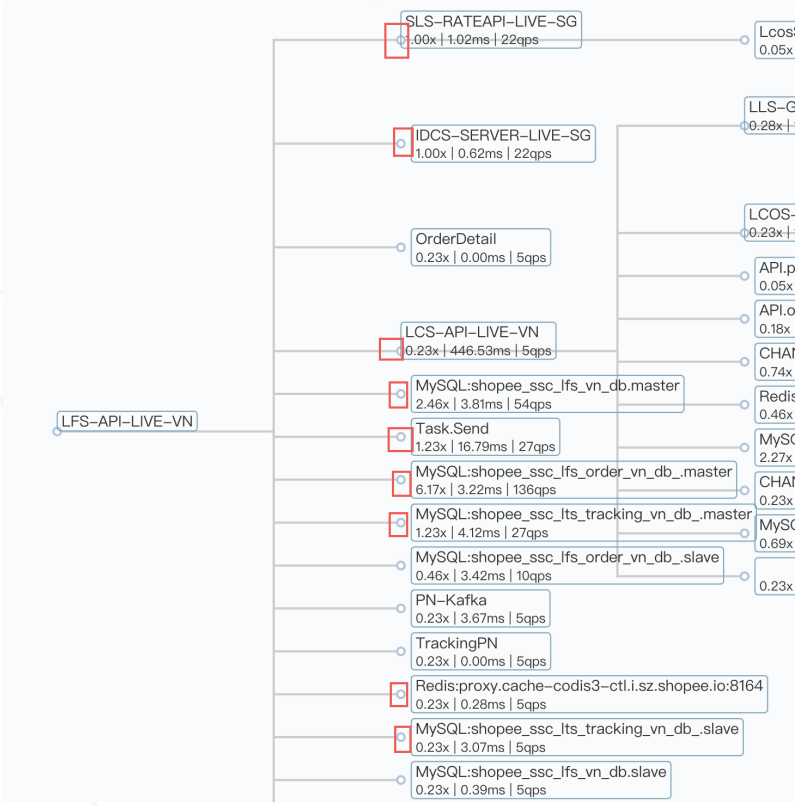
实验结果采集

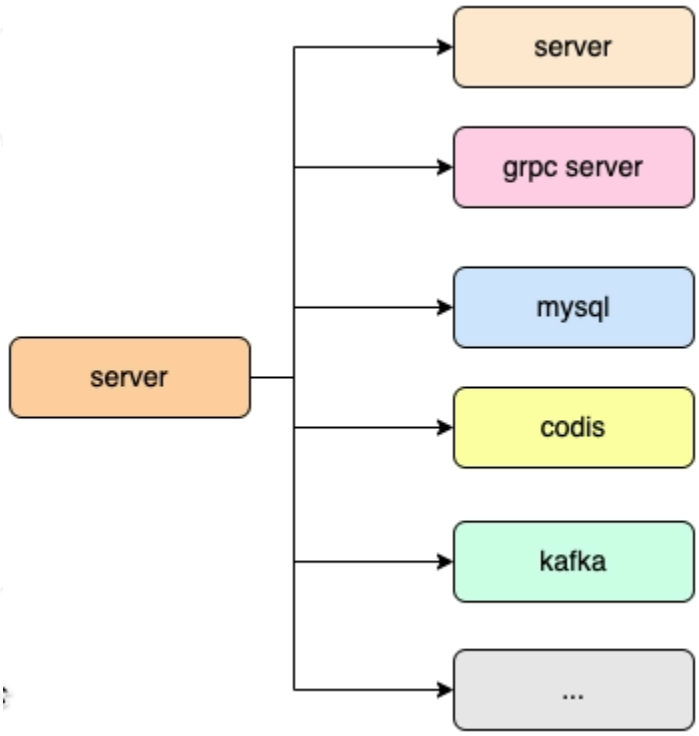
实验结果如果验证主要通过两个方面：1.业务性的指标，如接口产生的业务数据和返回结果；2.应用健康等监控指标，如CPU占用、接口延迟和接口错误率等。

1. 业务性指标需要到服务内埋点收集，和catclient 埋点收集transaction类似，在链路节点的数据进出口处将参数和结果收集起来。CKC和CKA通信通过共享内存实现的一个队列，主要传输CKC埋点在服务上收集的信息，架构图如下：



埋点位置主要在链路的各服务交互处，如下图：





对应到代码中则是在调用其他服务请求、读写数据库等操作时进行拦截添加埋点(当前各服务已实现埋点进行链路信息上报):

```
func Post(ctx context.Context, url string, data []byte, timeout time.Duration, opts ...OptionsFunc) (fastResponse
    if httpInterceptor != nil : httpInterceptor(ctx, httpMethodPost, url, nil, data, timeout, request, opts...)
    fastResponse, err = requestWithMonitor(ctx, httpMethodPost, url, params: nil, data, timeout, opts...)
    if fastResponse == nil && err == nil : nil, fmt.Errorf("%s:%s, response is empty", "POST", url)
    return fastResponse, err
}

// Get querys : query parameters
func Get(ctx context.Context, url string, querys []byte, timeout time.Duration, opts ...OptionsFunc) (fastResponse
    if httpInterceptor != nil : httpInterceptor(ctx, httpMethodGet, url, querys, nil, timeout, request, opts...)
    fastResponse, err = requestWithMonitor(ctx, httpMethodGet, url, querys, data: nil, timeout, opts...)
    if fastResponse == nil && err == nil : nil, fmt.Errorf("%s:%s, response is empty", "GET", url)
    return fastResponse, err
}
```

```
func requestWithMonitor(ctx context.Context, method []byte, url string, params []byte, data []byte, timeout time.Duration, o
    monCtx := monitor.AwesomeReportTransactionStart(ctx)
    var fastResponse *FastResponse
    var err error
    rootMsgId, parentMsgId, childMsgId := getCatTracingIds(monCtx)
    defer func() {
        moduleName, interfaceName := getModuleAndInterfaceName(ctx, url)
        msg := getTrackInfo(method, params, data, fastResponse, err)
        if r := recover(); r != nil {
            stack := debug.Stack()
            monitor.AwesomeReportTransactionEnd(monCtx, moduleName, interfaceName, StatusPanic, fmt.Sprintf("recover: #{r} |
            return
        } else if err != nil {
            monitor.AwesomeReportTransactionEnd(monCtx, moduleName, interfaceName, StatusTimeout, msg)
            return
        }
        _ = monitor.AwesomeReportEvent(monCtx, apm.RemoteCall, interfaceName: "", status: "0", childMsgId)
        status := getMonitorStatus(ctx, fastResponse)
        monitor.AwesomeReportTransactionEnd(monCtx, moduleName, interfaceName, status, msg)
    }()
    if len(opts) == 0 {
        opts = make([]OptionsFunc, 0, 3)
    }
    opts = append(opts,
```

2. 应用健康监控指标可以通过grafana获取。

实验结果分析

实验结果分析根据收集的两种指标有不同的分析逻辑。

1.对于应用健康监控指标，将故障期间峰值(谷值)与阈值相比较即可得出指标结果。该类指标在实验前必需明确指标获取方式，指标阈值，和安全范围(阈值之上还是阈值之下)。下面是一些常见应用健康监控指标：

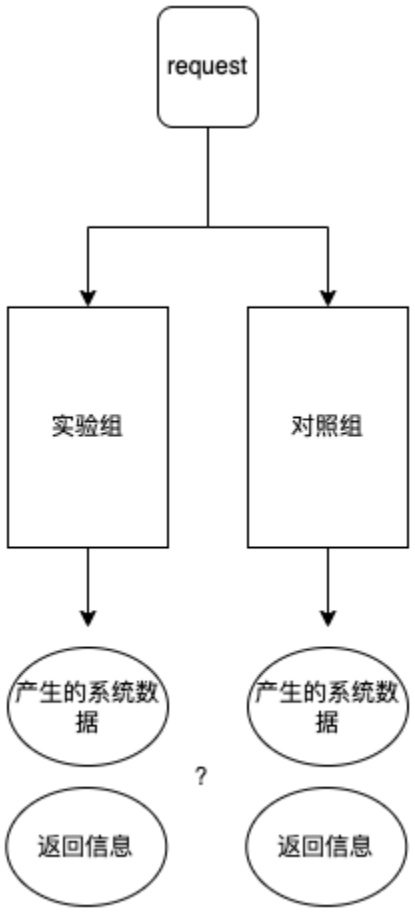
类型	指标	安全范围
应用	接口错误率	阈值之下
	接口延迟	阈值之下
	cpu负载	阈值之下
	内存占用	阈值之下

	...	...
消息队列	队列消息堆积	阈值之下
	内存占用	阈值之下
	磁盘空间	阈值之下
	...	
mysql/codis	读QPS	阈值之下
	写QPS	阈值之下
	读延迟	阈值之下
	读错误率	阈值之下
	cpu负载	阈值之下
	连接数	阈值之下
	...	

综上所示，几乎所有的安全范围都是阈值之下，即使一些指标的安全范围是阈值之上也可以通过反向描述将其变为阈值之下，所以默认所有指标的安全范围都是阈值之下，无需指明。

2.对于业务性指标，要分析则复杂很多。

理想状态需要一个实验组和一个对照组，将相同的请求分别发到故障环境和正常环境，最后比较两个环境的产生数据和返回信息：



只比较返回信息是不足以证明问题的，产生的系统数据比较也是相当重要的，可以发现很多隐性缺陷。

产生的系统数据包括有：写入/修改磁盘的文件、写入消息队列的消息、写入/修改数据库的信息等，如果实验时设置有服务边界，发送给其他服务的请求数据也可以算(因为其他服务也有可能根据请求数据产生系统数据)。

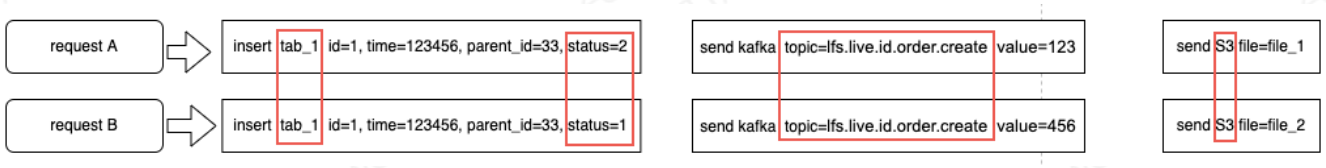
但是毕竟理想状态，很多条件都没法实现。比如资源问题，不可能拿两套环境做对比实验；如果拿一套环境又做实验组又做对照组，那么对于相同的写请求就有可能无法执行两次，除非每次写完要做数据清理，但面对复杂的业务场景，数据清理也是很庞大的工作量，而且这不是混沌工程的主要工作。所以需要接近理想状态的既轻量又便捷的方案。

把读接口和写接口分别分析：

读接口不会产生系统数据，所以可以只关注返回信息，并且可以相同请求重复实验。

写接口会产生系统数据，我们可以在接口的调用链上对产生数据的节点前进行拦截，获取执行动作的参数和执行结果。比如接口往数据表A insert 一条记录，那么可以拦截到insert 的 sql 和 执行结果，这样可以认为和直接查询数据记录是相同的效果。

对于不同写请求，在产生的数据可能不一样，但基本认为产生数据的数量是要一致，并且一条数据中一些状态字段的值也是要一致的。



如上图比较一些和请求无关的关键字段值发现 status 的值不一致，可以认为该故障影响了 tab_1插入数据这部分的逻辑。

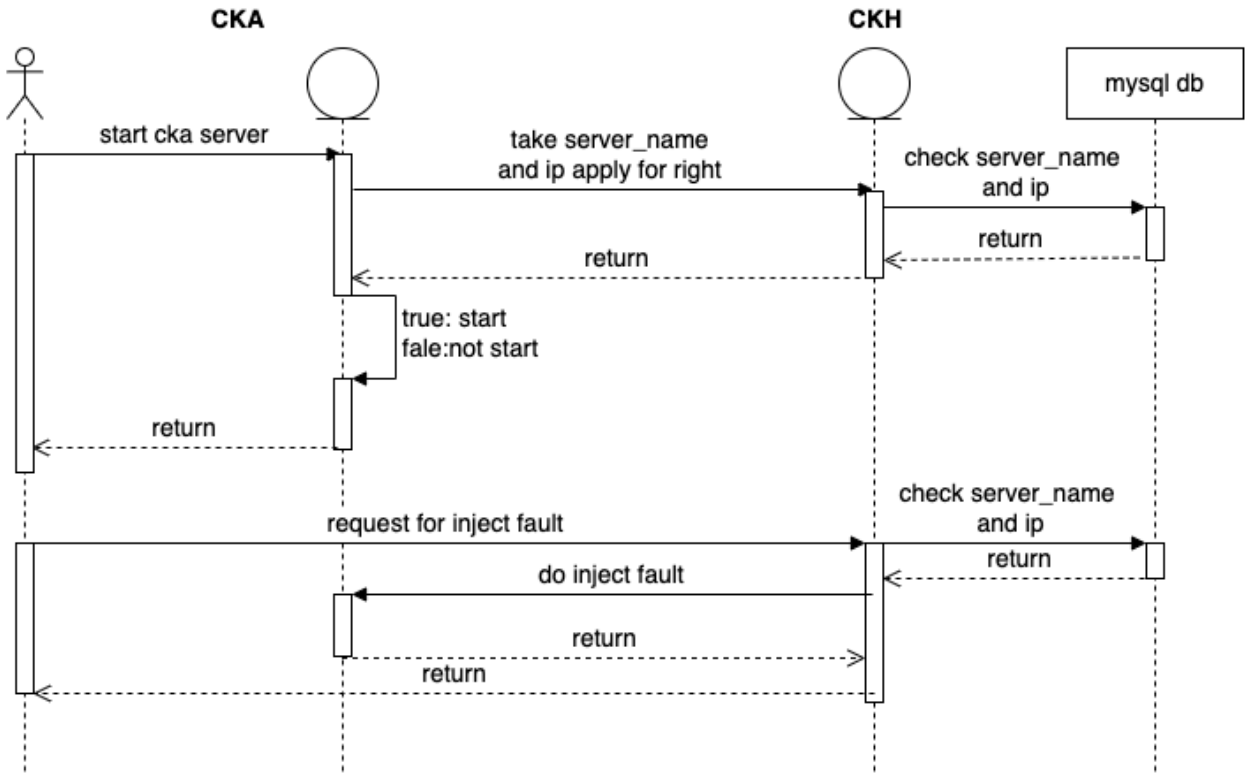
lfs履约下单场景混沌实验故障点和实验结果分析

lfs履约下单场景梳理：链接

场景接口	故障目标类型	故障	应用健康指标	业务指标
/api/sls/order/create	SERVER	RATE-API 应用下线 /api/sls-basic/freight/calc_estimated_shipping_fee/ 接口返回500	1. LFS-API 接口 /api/sls/order/create 错误率，阈值=0 2. LFS-API 接口 /api/sls/order/create 延迟，阈值=500ms 3. LFS-API cpu占用，阈值=60% 4. LFS-TASK cpu占用，阈值=60% 5. LFS-API 内存占用，阈值=80% 6. LFS-TASK 内存占用，阈值=80% 7. kafka-ssc-live-01.ctl.i.sz.shopee.io:9092;kafka-ssc-live-02.ctl.i.sz.shopee.io:9092;kafka-ssc-live-03.ctl.i.sz.shopee.io:9092 队列的topic=lfs.live.${cid}.order.create 队列消息堆积，阈值=1000 8. db-(backendslave\|master)-ssc-lfs-cross-sg1-live.shopeemobile.com:6606 数据库读qps, 阈值=10000 9. db-(backendslave\|master)-ssc-lfs-${cid}-sg1-live.shopeemobile.com:6606 数据库读qps, 阈值=10000 10. db-(backendslave|master)-ssc-lfs-order-${CID}-%09d-sg1-live.shopeemobile.com:6606 数据库读qps, 阈值=10000 11. db-(backendslave\|master)-ssc-sls-${cid}-sg1-live.shopeemobile.com:6606 数据库读qps, 阈值=10000 12. db-backendslave-sls-basic-service-sg1-live.shopeemobile.com:6606 数据库读qps, 阈值=10000 13. proxy.cache-codis3-ctl.i.sz.shopee.io:8164 读qps，阈值=80000	1. RATE-API 接口 /api/sls-basic/freight/calc_estimated_shipping_fee/ 请求参数： 返回值： 2. IDCS 接口 /segmentpb.SegmentIdService/Request 请求参数： 返回值： 3. LFS-API 向 kafka-ssc-live-01.ctl.i.sz.shopee.io:9092;kafka-ssc-live-02.ctl.i.sz.shopee.io:9092;kafka-ssc-live-03.ctl.i.sz.shopee.io:9092 topic=lfs.live.${cid}.order.create 发送 sloid 4. LCS-API 接口 /api/lls/create_line_waybill 请求参数： 返回值： 5. db-(backendslave\|master)-ssc-lfs-order-${CID}-%09d-sg1-live.shopeemobile.com:6606/shopee_ssc_lfs_order_${cid}_db_%09d/logistic_order_tab_%09d 写数据 6. db-(backendslave\|master)-ssc-lfs-order-${CID}-%09d-sg1-live.shopeemobile.com:6606/shopee_ssc_lfs_order_${cid}_db_%09d/fulfillment_action_tab_%09d 写数据 7. db-(backendslave\|master)-ssc-lfs-${cid}-sg1-live.shopeemobile.com:6606/shopee_ssc_lfs_${cid}_db/logistic_order_index_map_tab_%08d 写数据 8. db-(backendslave\|master)-ssc-lfs-order-${CID}-%09d-sg1-live.shopeemobile.com:6606/shopee_ssc_lfs_order_${cid}_db_%09d/logistic_order_tab_%09d 写数据 9. db-(backendslave\|master)-ssc-lfs-order-${CID}-%09d-sg1-live.shopeemobile.com:6606/shopee_ssc_lfs_order_${cid}_db_%09d/logistic_order_data_tab_%09d 写数据 10. db-(backendslave\|master)-ssc-lfs-order-${CID}-%09d-sg1-live.shopeemobile.com:6606/shopee_ssc_lfs_order_${cid}_db_%09d/actual_shipping_fee_tab_%09d 写数据 11. proxy.cache-codis3-ctl.i.sz.shopee.io:8164 /new_status_order-${slo_id} 写数据 12. LFS-API 接口 /api/sls/order/create 返回值
		RATE-API 应用 /api/sls-basic/freight/calc_estimated_shipping_fee/ 接口延迟至1s后返回成功		
		IDCS 应用下线 /segmentpb.SegmentIdService/Request 接口返回500		
		IDCS 应用 /segmentpb.SegmentIdService/Request 接口延迟到1s后返回成功		
		LCS-API 应用下线 /api/lls/create_line_waybill 接口返回500		
		LCS-API 应用 /api/lls/create_line_waybill 接口延迟1s后返回成功		
		LFS-API 内存占用80%		
		LFS-TASK 内存占用80%		
		LFS-API cpu占用60%		
		LFS-TASK cpu占用60%		
		...		
	MYSQL	db-(backendslave|master)-ssc-lfs-cross-sg1-live.shopeemobile.com:6606 数据库下线，请求返回失败		
		db-(backendslave|master)-ssc-lfs-${cid}-sg1-live.shopeemobile.com:6606 数据库下线，请求返回失败		
		db-(backendslave|master)-ssc-lfs-order-${CID}-%09d-sg1-live.shopeemobile.com:6606 数据库下线，请求返回失败		
		db-(backendslave|master)-ssc-sls-${cid}-sg1-live.shopeemobile.com:6606 数据库下线，请求返回失败		
		db-backendslave-sls-basic-service-sg1-live.shopeemobile.com:6606 数据库下线，请求返回失败		
		...		
	REDIS	proxy.cache-codis3-ctl.i.sz.shopee.io:8164 数据库下线，请求返回失败		
		...		
	KAFKA	kafka-ssc-live-01.ctl.i.sz.shopee.io:9092;kafka-ssc-live-02.ctl.i.sz.shopee.io:9092;kafka-ssc-live-03.ctl.i.sz.shopee.io:9092 队列的topic=lfs.live.${cid}.order.create 发生消息堆积		

		kafka-ssc-live-01.ctl.i.sz.shopee.io:9092;kafka-ssc-live-02.ctl.i.sz.shopee.io:9092;kafka-ssc-live-03.ctl.i.sz.shopee.io:9092 队列的topic=lfs.live.${cid}.order.create 发生消息丢失		
		kafka-ssc-live-01.ctl.i.sz.shopee.io:9092;kafka-ssc-live-02.ctl.i.sz.shopee.io:9092;kafka-ssc-live-03.ctl.i.sz.shopee.io:9092 队列的topic=lfs.live.${cid}.order.create 发生消息重复		

权限管理



因为CKA是一个能破坏系统的服务，所以需要对其进行严格的权限管理。参考上图，具体的权限管理如下：

1. CKA启动里需要权限：CKA带着目标服务名和机器的IP地址，向CKH请求权限；CKH通过对比权限配置的白名单决定是否允许申请的CKA启动。
2. 故障注入阶段需要权限：每次通过CKH故障注入时，都需要检查目标服务是否有权限。这种是为了防止白名单在服务启动后紧急添加的情况。

非功能特性设计

可靠性

描述系统在可靠性上的分析和设计，系统在容错性上是如何处理的，故障恢复、服务降级、熔断等的处理机制，以及在数据可靠性方面的考虑等。

可运维

描述在提升系统运维方面的分析及设计，好的系统应该是尽可能自动化的完成业务，自身应该具备相当能力的容错处理，不需要运维人员介入处理；同时也尽可能是标准化的，方便运维人员部署，或者提供可视化的运维操作。

1. CKH 嵌入在大促系统内，随大促系统部署运维
2. CKC、CKA 随应用服务部署运维

安全性

配合安全要求，阐述系统安全方面（例如XSS，SQL注入，DDOS，数据安全等）的设计。

1. 采用xx环境来模拟live环境，不影响生产
2. 混沌工具agent运行增加权限，指定授权环境方可运行，避免误部署到其他环境制造破坏

3. 每个注入的故障都有相应的终止操作
4. 同步live数据做为实验请求数据，给数据添加xx的标识，以防数据意外发送其他live服务无法识别

可测试性

描述系统各个业务在可测试性上的分析，发布应该满足灰度要求，系统应该具备在生产环境的可测试性。必须等到某个时间点、修改操作系统时间，或者直接不可测，都不应该发生。

1. 每一项故障注入都可实现
2. 故障注入效果可监测，故障下发情况可跟踪到效果
3. 每一项稳态指标都有监控或告警
4. 实验环境选择在xx环境，apollo中的服务配置信息需要同步live环境的配置。测试数据同步live 的数据，请求数据通过收集live 请求来回放实现，SLS以外服务使用mock实现

监控

1. 因为实验结果收集的模块是通过改造cat client 实现，在进行混沌工程实验时服务cat 上报信息到cat 服务的功能将被替换，所以cat 类的监控无法使用。实验结果验证的监控主要以grafana 为主，辅以实验数据对比来进行实验结果分析。
2. 实验进行之前需要在预估影响范围内配置相关的监控。

其他