声明式自愈系统——高可用分布式系统的设计之道

王昕

高级技术专家

GTLC
GLOBAL
TECH LEADERSHIP
CONFERENCE

全球技术领导力峰会

Geekbang. 极客邦科技 | TGO 鲲鹏會

500+高端科技领导者与你一起探讨
技术、管理与商业那些事儿

2019年6月14–15日 | 上海圣诺亚皇冠假日酒店

扫码了解更多信息

自我介绍

王昕，阿里中间件技术团队高级技术专家，阿里云开放云平台布道师。具有10多年软件系统开发和架构经验，在分布式系统领域经验丰富，长期参与高可用中间件系统、云平台基础管理系统和云原生自动运维系统的构建。在国内外有10多项授权和在审软件技术发明专利，并多次受邀出席技术会议，做技术专题分享。

目录

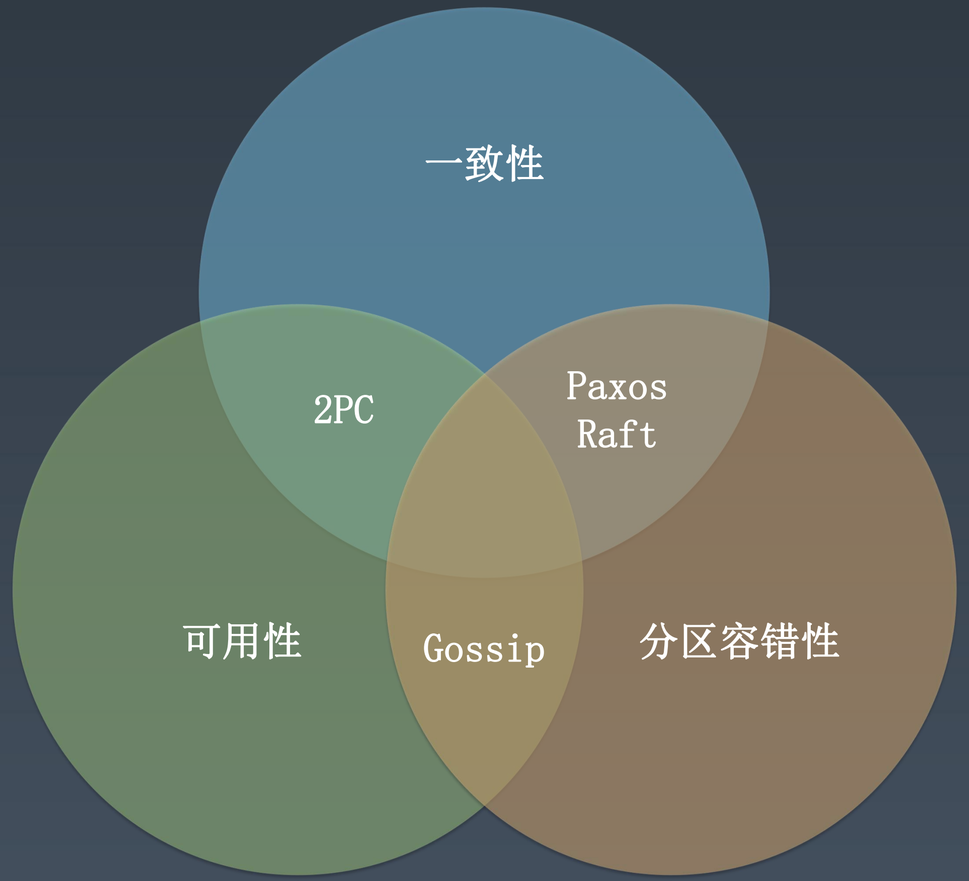
- 分布式系统面临的高可用问题
- 设计和验证高可用分布式系统的工具与方法
- 设计和验证高可用分布式系统的案例分享
- 高可用系统的最佳实践总结

无状态分布式系统的高可用问题

- 处理消息的服务节点可以随机选择
- 不必处理数据复制和同步的问题
- 系统容量和高可用能力可以同步提升
- 服务节点可以随意迁移，不必固定 IP 和存储

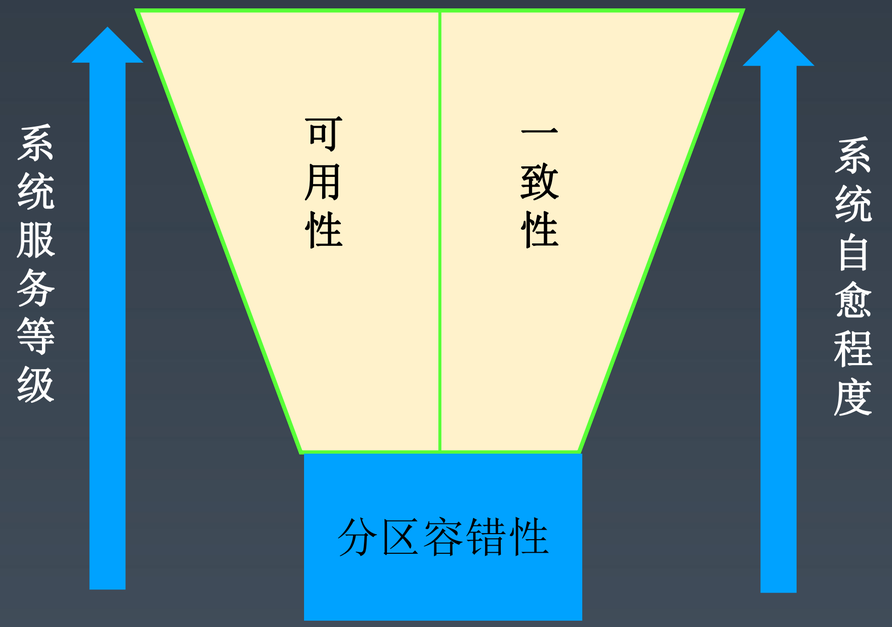
有状态分布式系统的高可用问题

- 处理请求需要特定节点
- 必须要考虑数据备份和同步的问题
- 容量扩展和高可用需要不同解决方案
- 服务节点不能随便迁移



CAP Is Not Simply 2 out of 3

- 没有分区时，可用性和一致性要兼得
- 经常要考虑的是可用性和一致性各有一部分
- 根据不同设计应用需求有不同的组合
- 重要的是系统如何恢复到“最佳状态”



Look Distributed System in another Way

Safety

- Something bad will never happen
- e.g. received message is lost

Liveness

- Something good will eventually happen
- e.g. is able to receive message

目录

- 分布式系统面临的高可用问题
- 设计和验证高可用分布式系统的工具与方法
- 设计和验证高可用分布式系统的案例分享
- 高可用系统的最佳实践总结

依据声明式自愈的理念设计系统

有一个统一的状态持久化接口，所有有状态模块通过统一的接口对应统一的对象模型

控制器模块对象包括Desired State 和 Realized State

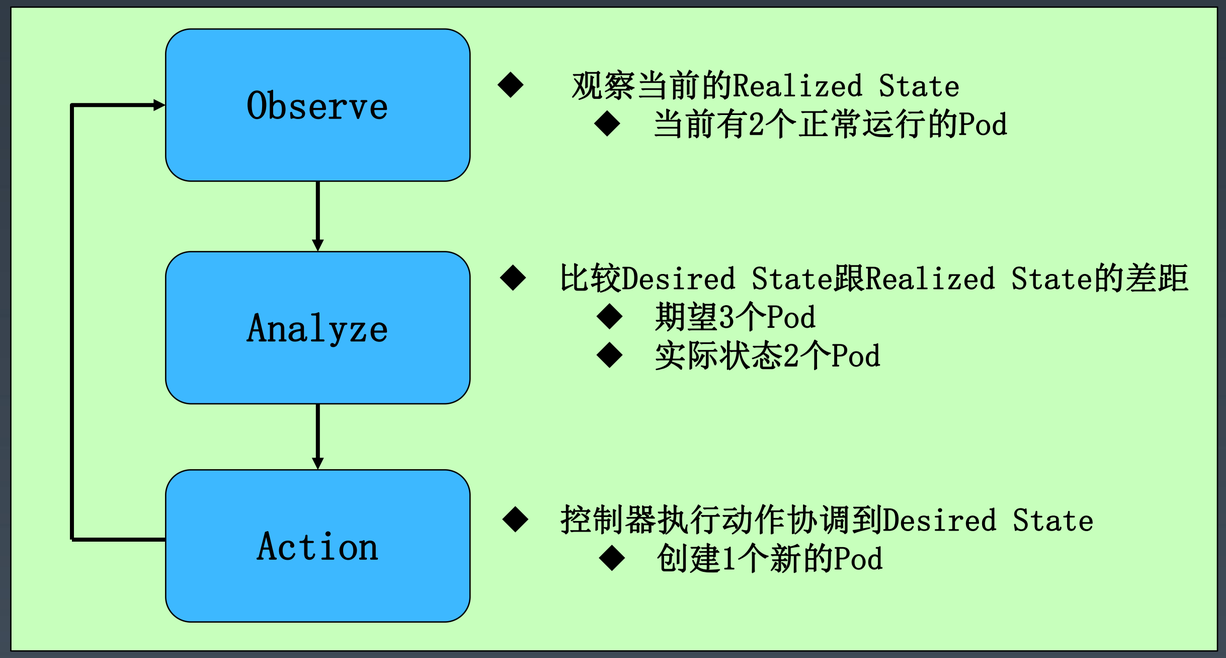
配置模块对象只需要包括 Desired State

改变状态的操作必须是幂等的声明式操作，没有新声明时各模块按照之前的声明继续工作

每个领域的控制器模块的逻辑保证自己领域独立自愈的能力

声明式自愈系统的控制器协调循环

- Controller观察特定领域的系统状态
- 协调Desired State跟Realized State之间的差距，维持最终一致性
- 定期处理集群中的事件
- 系统必须是幂等的



控制器的设计理念

控制逻辑应该只依赖于当前状态

假设任何错误的可能，并做容错处理

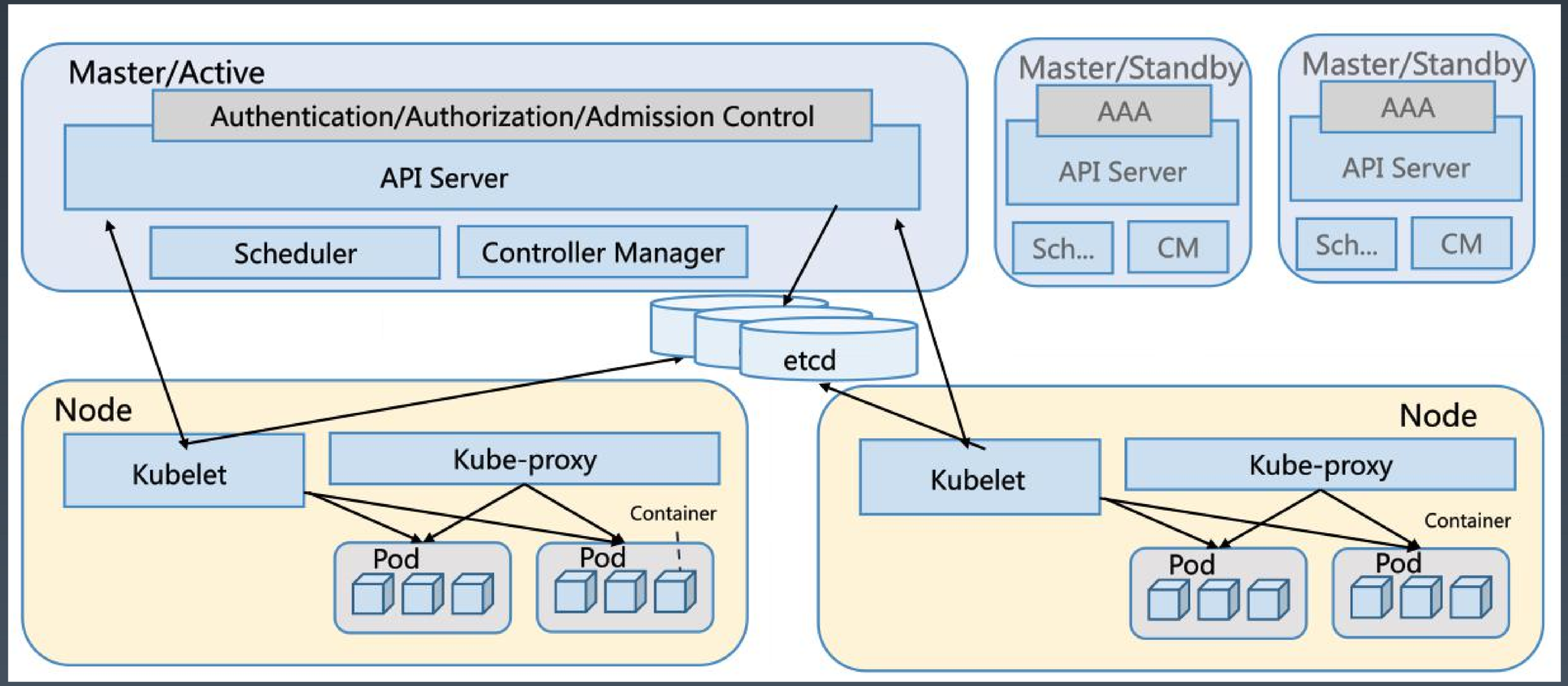
尽量避免复杂状态机，逻辑不要依赖无法监控的内部状态

每个模块都可以在必要时优雅地降级服务

每个模块都可以在出错后自动恢复

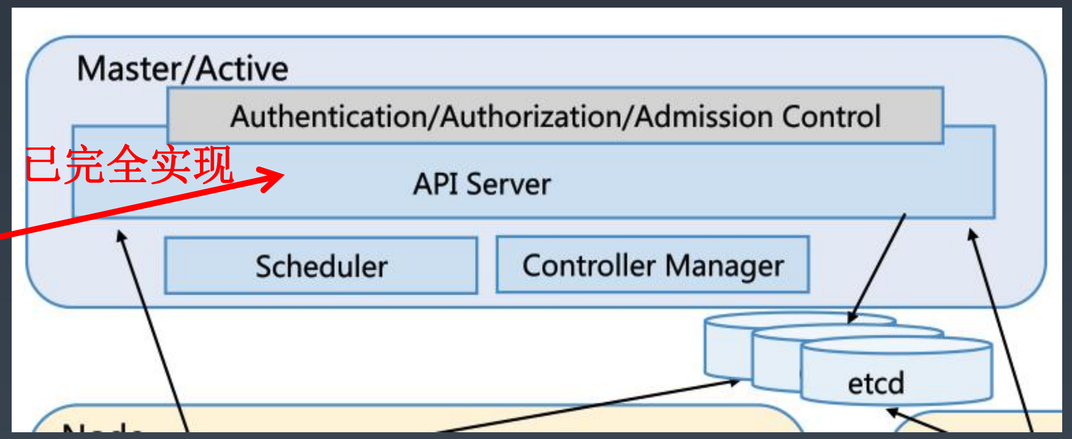
假设任何命令都可能被任何调用对象拒绝，甚至返回错误结果

声明式自愈系统的现有框架——Kubernetes



声明式自愈系统设计理念的回顾

统一接口和对象模型



幂等操作和状态机

要在领域内自己实现

```
// ReplicationController represents the configuration
type ReplicationController struct {
	metav1.TypeMeta
	// +optional
	metav1.ObjectMeta

	// Spec defines the desired behavior of this rep
	// +optional
	Spec ReplicationControllerSpec

	// Status is the current status of this replication co
	// out of date by some window of time.
	// +optional
	Status ReplicationControllerStatus
}
```

DS 和 RS

可重用部分

```
// Policy d
type Policy
	metav1.Type

	// Spec describes the po
	Spec PolicySpec
}
```

Desired State

自愈能力

如何设计好状态机和自愈协议？

Writing Correct Software Is Hard!

Math and Thinking Can Help Us!

使用 TLA+ 定义和验证系统设计

- TLA+ 是用来给（软件或硬件）系统建模的语言
- TLA+ 强调排除特定编程语言（软件或硬件）的影响验证系统设计
- TLA+ 由 Paxos 协议的发明人 Leslie Lamport 发明

Why TLA+

Language vs. Concept

For quite a while, I've been disturbed by the emphasis on language in computer science. One result of that emphasis is programmers who are C++ experts but can't write programs that do what they're supposed to ...

I believe that the best way to get better programs is to teach programmers how to think better. Thinking is not the ability to manipulate language; it's the ability to manipulate concepts.

Leslie Lamport

Whorfian syndrome
沃尔夫综合征
Language vs. Reality

Computer scientists collectively suffer from what I call the Whorfian syndrome1—the confusion of language with reality. Since these devices are described in different languages, they must all be different. In fact, they are all naturally described as state machines.

Leslie Lamport

Functions vs. State Machines

Yet computer scientists are so focused on the languages used to describe computation that they are largely unaware that those languages are all describing state machines.

Leslie Lamport

TLA+试图用状态机而不是计算过程描述系统

- 更加关注系统的非正常输入
- 更加关注系统的稳定性可用性等特性而不是系统输出
- 分布式系统环境下没有单一的输入输出

TLA+是一种适合定义状态机的语言

定义一个状态机需要：

1. 定义所有可能的初始状态
2. 定义在特定状态下可能有哪些状态转换

```
CONSTANTS NUMBER_RANGE
VARIABLES x, pc

vars == << x, pc >>

Init == (* Global variables *)
        /\ x \in NUMBER_RANGE
        /\ pc = "Divide"

Divide == /\ pc = "Divide"
          /\ x' = 100 \div x
          /\ pc' = "Done"

Next == Divide
           \/ (* Disjunct to prevent deadlock on termination *)
              (pc = "Done" /\ UNCHANGED vars)

Spec == Init /\ [][Next]_vars

Termination == <>(pc = "Done")
```

一个程序可以用状态机描述

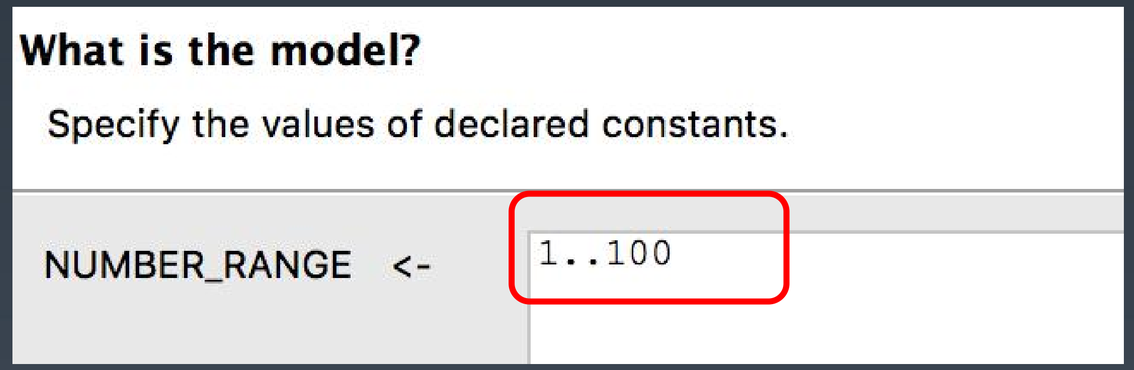
计算 100/x 的程序：
x = 100 / x;

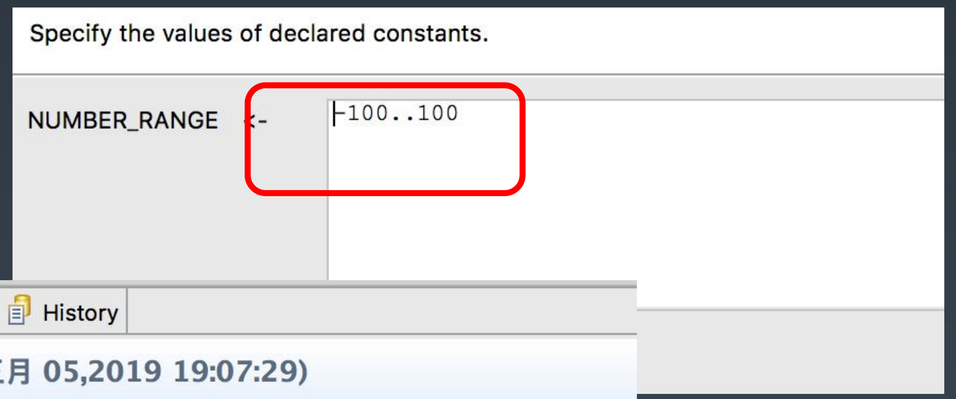
定义一个状态机需要：
1. 定义变量
2. 定义所有可能的初始状态
3. 定义在特定状态下可能有哪些状态转换

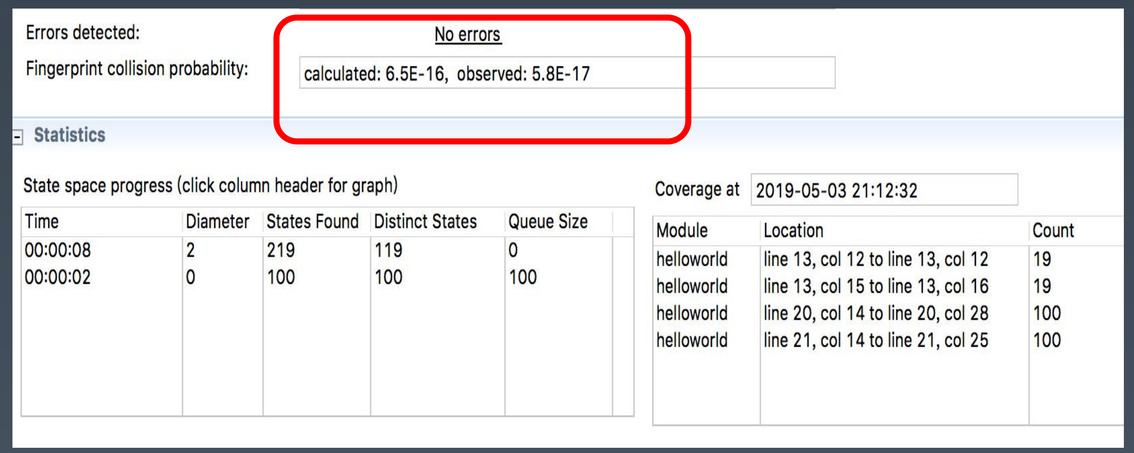
```
CONSTANTS NUMBER_RANGE
VARIABLES x, pc

vars == << x, pc >>

Init == (* Global variables *)
        /\ x \in NUMBER_RANGE
        /\ pc = "Divide"

Divide == /\ pc = "Divide"
          /\ x' = 100 \div x
          /\ pc' = "Done"

Next == Divide
           \/ (* Disjunct to prevent deadlock on termination *)
              (pc = "Done" /\ UNCHANGED vars)

Spec == Init /\ [][Next]_vars

Termination == <>(pc = "Done")
```

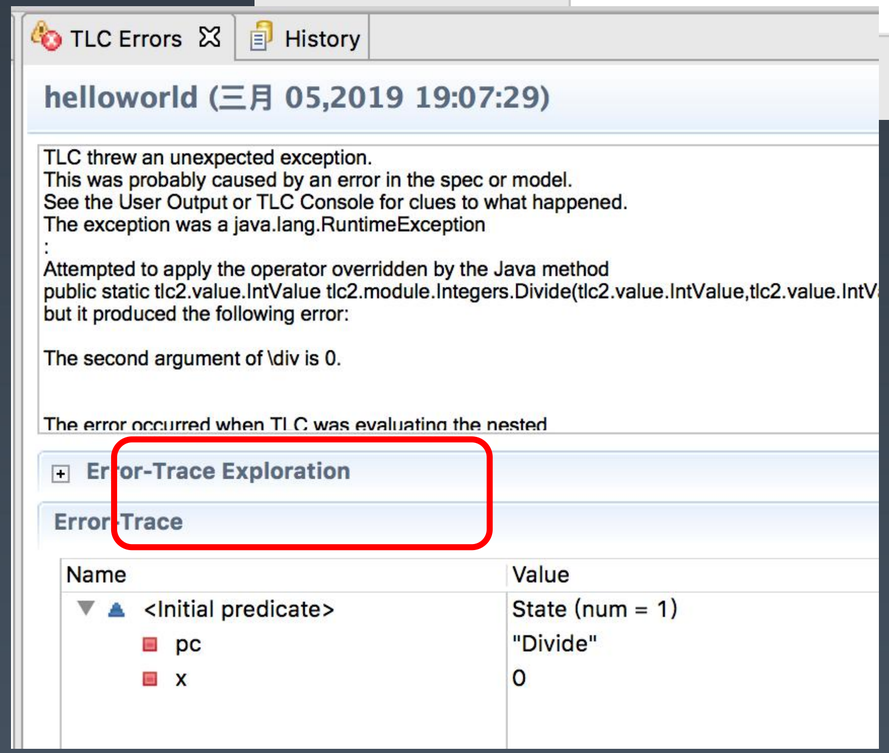
TLA+的应用场景——验证系统设计的正确性

TLA+工具会遍历所有可能的状态验证系统的正确性



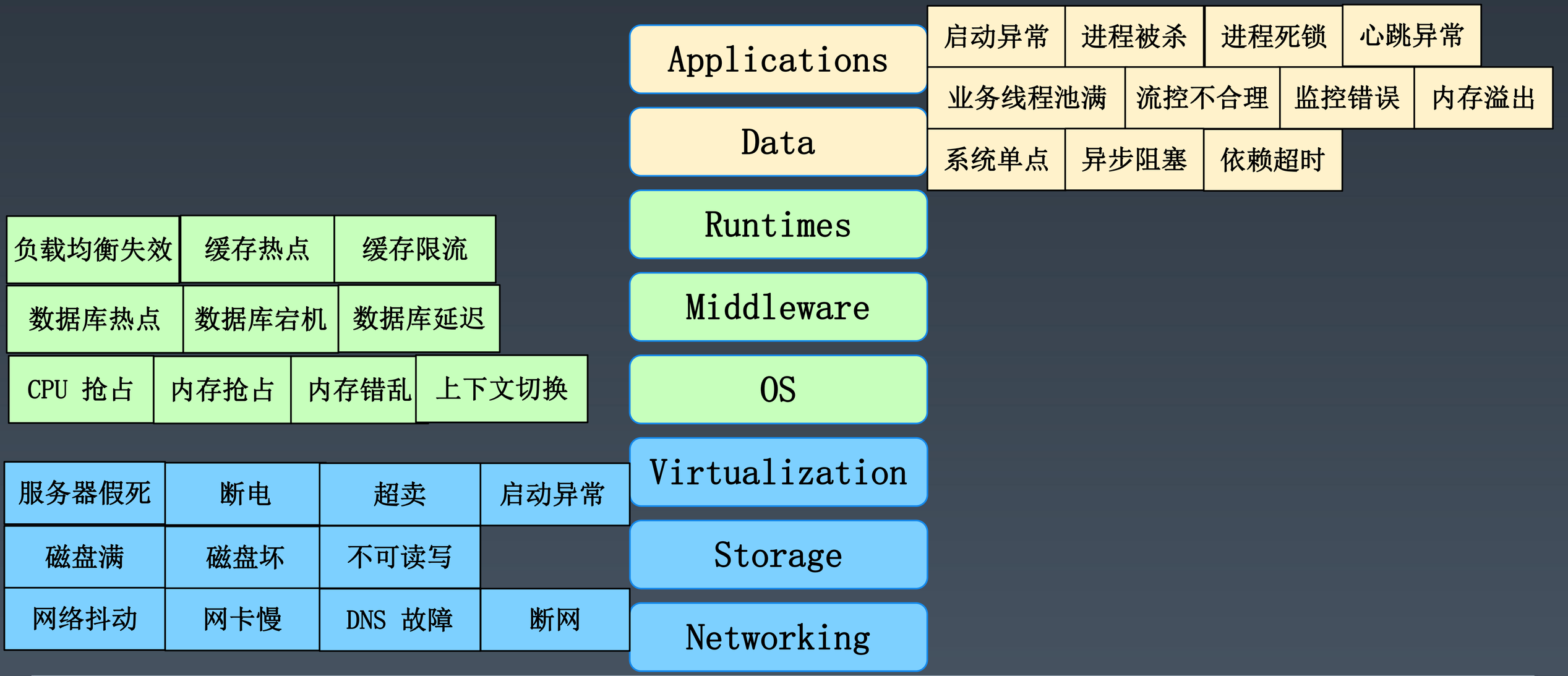






分布式系统中有哪些异常情况需要模拟？

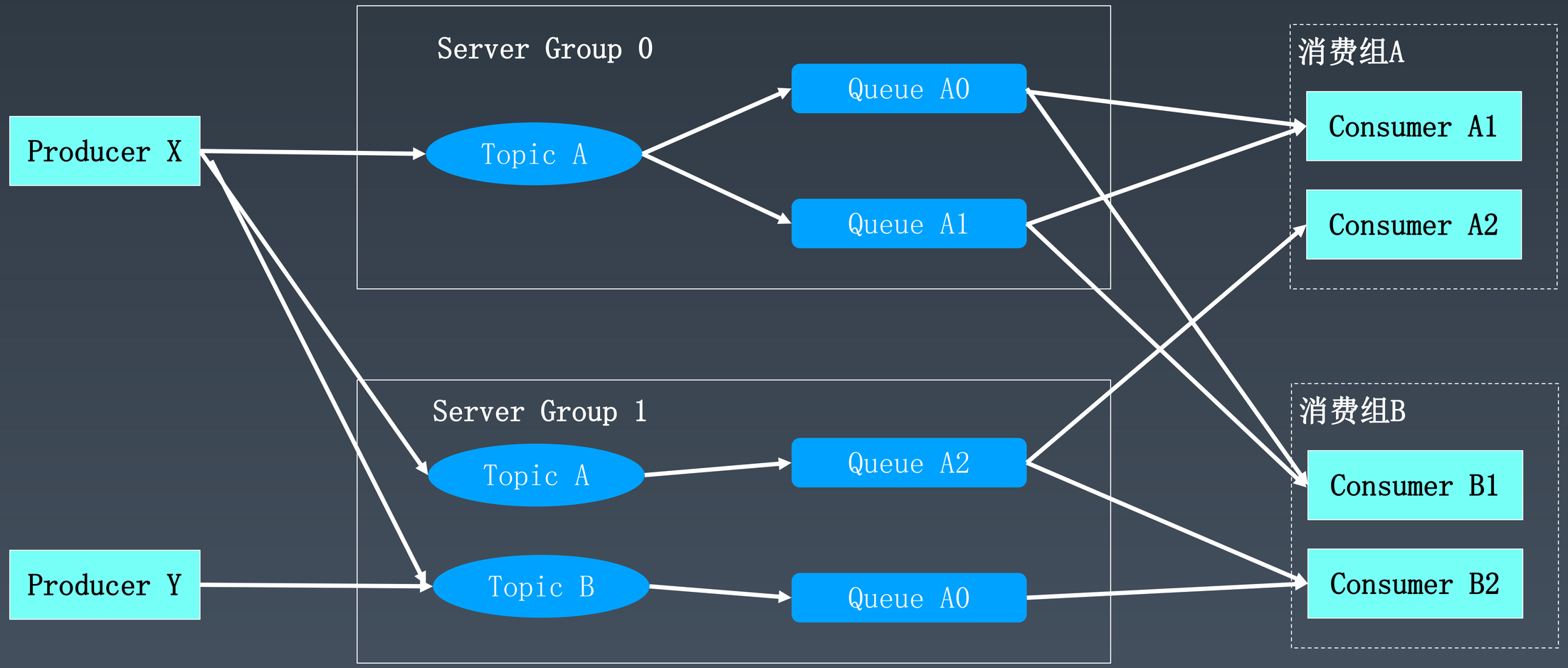
运行时可能出现的异常


Applications
启动异常
进程被杀
进程死锁
心跳异常
业务线程池满
流控不合理
监控错误
内存溢出
Data
系统单点
异步阻塞
依赖超时
Runtimes
负载均衡失效
缓存热点
缓存限流
Middleware
数据库热点
数据库宕机
数据库延迟
OS
CPU 抢占
内存抢占
内存错乱
上下文切换
Virtualization
服务器假死
断电
超卖
启动异常
Storage
磁盘满
磁盘坏
不可读写
Networking
网络抖动
网卡慢
DNS 故障
断网

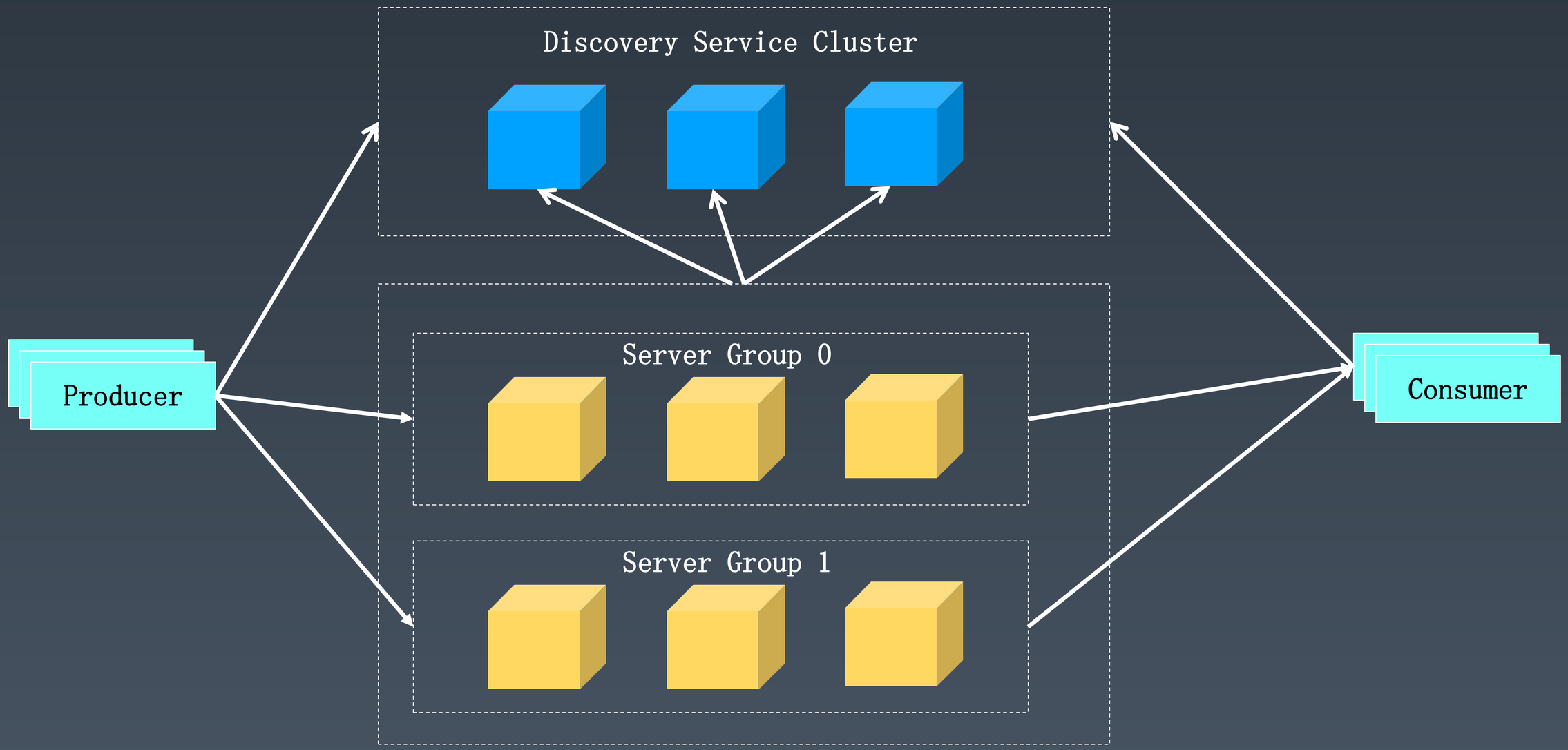
目录

- 分布式系统面临的高可用问题
- 设计和验证高可用分布式系统的工具与方法
- 设计和验证高可用分布式系统的案例分享
- 高可用系统的最佳实践总结

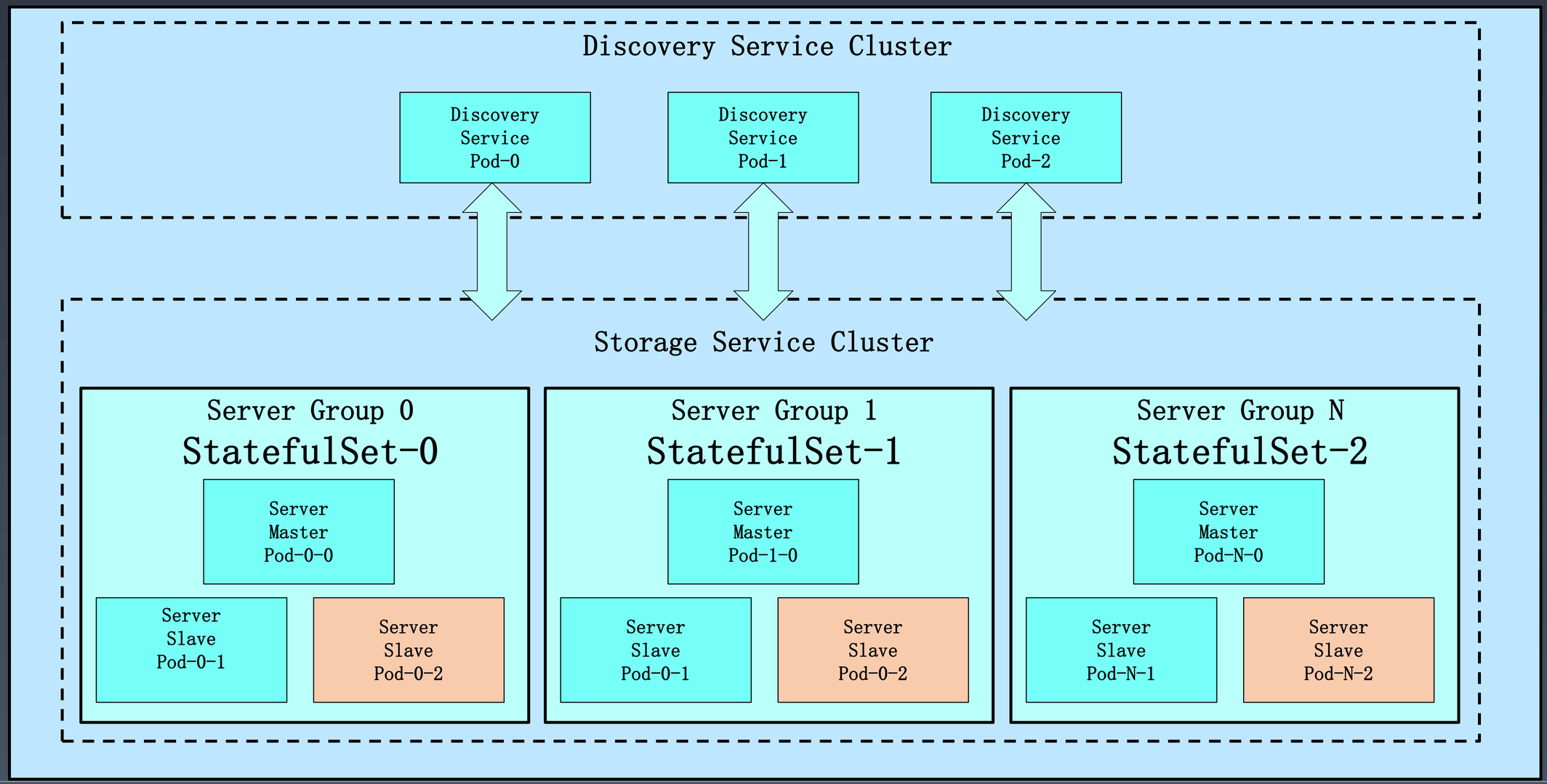
一个分布式消息系统的概念模型



分布式消息系统的部署模型



分布式消息系统在 Kubernetes 下的部署运维



使用 TLA+ 验证系统自愈特性——目标状态

```
type BrokerCluster struct {
  metav1.TypeMeta   `json:",inline"`
  metav1.ObjectMeta `json:"metadata"`
  Spec              BrokerClusterSpec   `json:"spec"`
  Status            BrokerClusterStatus `json:"status"`
}
```

```
type BrokerClusterSpec struct {
  BrokerImage         string                          `json:"br
  NameServers         string                          `json:"na
  StorageClass        string                          `json:"st
  ReplicationMode     string                          `json:"re
  GroupReplicas       int32                           `json:"gr
  MembersPerGroup     int32                           `json:"me
  Properties          map[string]string               `json:"pr
  NodeSelector        map[string]string               `json:"no
  Affinity            *corev1.Affinity                `json:"af
  VolumeClaimTemplate *corev1.PersistentVolumeClaim   `json:"vo
  Config              *corev1.LocalObjectReference    `json:"co
}
```

Desired State

```
\* Allowed replica number
CONSTANTS REPLICAS_NUMBER

VARIABLE groupReplicas
VARIABLE readyGroupRecplicas

controllerVars == <<groupReplicas, readyGroupRecplicas>>

\* All variables; used for stuttering (asserting state hasn't changed).
vars == <<controllerVars>>

\* Define initial values for all variables

InitControllerVars == /\ groupReplicas = 0
                      /\ readyGroupRecplicas  = 0

Init == /\ InitControllerVars

\* It loses everything but its currentTerm, votedFor, and log.
SetDesiredReplicas(i) ==
    /\ groupReplicas' = i
    /\ UNCHANGED <<readyGroupRecplicas>>
```

使用 TLA+ 验证系统自愈特性——实际状态

```
type BrokerCluster struct {
  metav1.TypeMeta   `json:",inline"`
  metav1.ObjectMeta `json:"metadata"`
  Spec              BrokerClusterSpec   `json:"spec"`
  Status            BrokerClusterStatus `json:"status"`
}
```

Realized State

```
type BrokerClusterStatus struct {
  ReadyGroupReplicas int32 `json:"readyGroupRepl
  Groups             []BrokerGroupStatus
  Conditions         []BrokerClusterCondition
}
```

```
\* Allowed replica number
CONSTANTS REPLICAS_NUMBER

VARIABLE groupReplicas
VARIABLE readyGroupRecplicas

controllerVars == <<groupReplicas, readyGroupRecplicas>>

\* All variables; used for stuttering (asserting state hasn't changed).
vars == <<controllerVars>>

\* Define initial values for all variables

InitControllerVars == /\ groupReplicas = 0
                      /\ readyGroupRecplicas  = 0

Init == /\ InitControllerVars

\* It loses everything but its currentTerm, votedFor, and log.
SetDesiredReplicas(i) ==
    /\ groupReplicas' = i
    /\ UNCHANGED <<readyGroupRecplicas>>
```

设计实现声明式自愈系统的工具与方法

架构选型

- 选择基础状态持久化框架
- 参考系统：Kubernetes

高可用系统设计

- 设计并验证高可用分布式系统
- 参考工具：TLA+ Toolkit

系统实现

- 实现高可用系统的控制模块
- 参考模式：Kubernetes Operator

系统测试

- 测试分布式高可用系统的自愈能力
- 参考工具：Jepsen

目录

- 分布式系统面临的高可用问题
- 设计和验证高可用分布式系统的工具与方法
- 设计和验证高可用分布式系统的案例分享
- 高可用系统的最佳实践总结

最佳实践分享

有关 TLA+ 的使用

- 分布式系统设计 80% 的重点工作在与设计安全性原则
- 目前 TLA+ 工具已经有云服务上线，但只支持检查安全性
- 单机版的 TLA+ 工具支持系统活性的检查，但是性能比较差
- 活性检查的性能瓶颈在于系统状态图中强连通图算法的实现
- TLA+ 中实现的卡壳（Stutter）等价能力，即对所有状态保持不变也是合法状态

最佳实践分享

有关分布式系统统一 API 的设计

- 所有API应该是声明式的
- 高层API以操作意图为基础设计
- 低层 API 根据高层 API 的控制需要设计
- API 对象是彼此互补而且可组合的
- 尽量避免简单封装,不要有隐藏的内部 API
- API 操作复杂度与对象数量成正比
- API 对象状态不依赖于连接状态
- 针对全局状态设计自愈容错机制

最佳实践分享

有关系统设计和运营

- 分开设计理想状态和实际状态 Desired State Realized State
- 利用队列解耦系统
- 尽量不要依赖 DNS 做高可用
- 监控负载不均的情况
- 避免Self DoS

最佳实践分享

有关微服务架构的典型模式

- 限流 Throttling
- 超时 Timeouts
- 熔断器 Circuit Breaker
- 壁舱 Bulkheads
- 快速失败 Fail Fast
- 背压模式 Backpressure

最佳实践分享

有关遵循快速回复的原则

- 快速报错 Fail Fast
- 复杂操作拆分成简单原子操作，在界面和 CLI 做组合
- 注意设置超时
- 尽量避免用全局事物
- 注意慢操作可能造成的客户端冲突问题

最佳实践分享

有关缓存的使用

- 缓存的内容：服务发现应答，DNS 结果，元数据，数据，之前的请求
- 逻辑正确性不能依赖缓存，写操作服务端必须有校验而且幂等，没有缓存情况下系统仍可服务
- 错误回复缓存，过期时间不能太长，而且有清晰的修复建议
- 数据库更新与缓存失效的策略

最佳实践分享

有关配置文件

- 集群使用统一的配置来源
- 定义正常的默认配置，满足读取不到配置的正常运行
- 支持可扩展的配置命令格式
- 尽量支持更改配置不需要重启服务
- 注意配置项之间的关联性

欢迎与我交流

王昕个人微信

极客时间 | 企业服务

这是一封正经“福利”贴

想做团队的领跑者 需要迈过这些“槛”

成长型企业，易忽视人才体系化培养 企业转型加快，团队能力又跟不上	VS	从基础到进阶，超100+一线实战 技术专家带你系统化学习成长
团队成员技能水平不一， 难以一“敌”百人需求	VS	解决从小白到资深技术人所遇到 80%的问题
寻求外部培训，奈何价更高且 集中式学习	VS	多样、灵活的学习方式，包括 音频、图文 和视频
学习效果难以统计，产生不良循环	VS	获取员工学习报告，查看学习 进度，形成闭环

课程顾问「橘子」

回复「QCon」
免费获取
学习解决方案

 # 极客时间企业账号 # 解决技术人成长路上的学习问题

THANKS!

QCon 10th