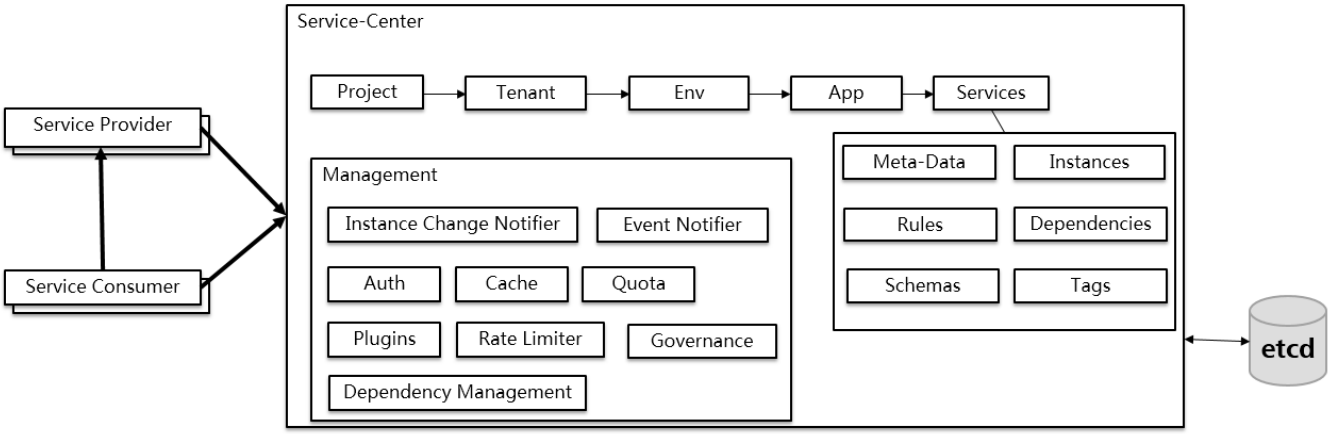
ServiceCenter 概述

ServiceCenter 是一个具有微服务实例注册/发现能力的微服务组件，提供一套标准的 RESTful API 对微服务元数据进行管理。ServiceComb 的微服务注册及动态发现能力也是依赖其实现的。



除了以上描述的微服务动态发现外，ServiceCenter 还具有以下特性：

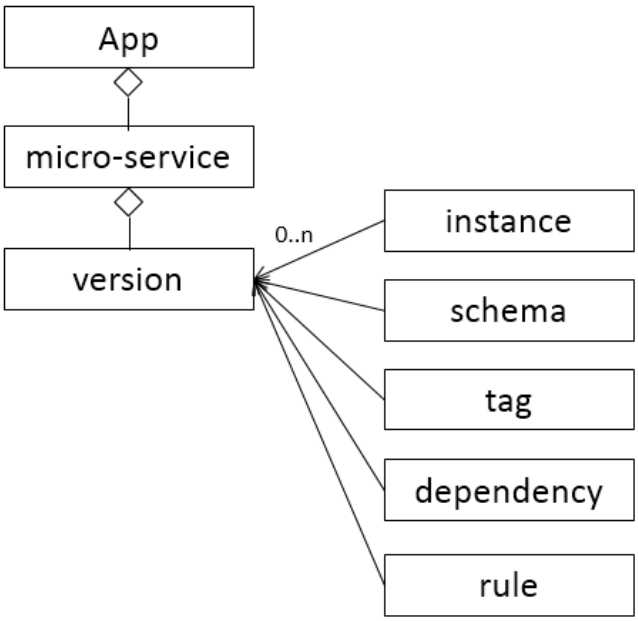
- 支持逻辑多租的微服务实例隔离管理
- 支持微服务黑白名单管理
- 支持长连接监听微服务实例状态变化
- 支持 OpenAPI 规范微服务契约管理
- 支持微服务依赖关系管理
- 提供 Web portal 展示微服务管理界面
- 高可用的故障处理模型（自我保护机制）
- 高性能接口和缓存数据设计

ServiceCenter 的作用

作为微服务系统中非常重要的组件。当前可选来做微服务注册中心的组件很多，例如

Zookeeper、Consul、Eureka、ETCD。然而这些服务注册中心组件的重点在于解决服务的注册和发现，也就是一个动态路由的问题。而在一个大型的微服务架构的系统中，除了这些动态路由信息外，还有一些微服务自身的元数据同样需要进行管理，所以我们定义出微服务静态元数据。通过 ServiceCenter 提供的接口可以很方便的检索微服务信息。

除了微服务本身信息属于静态元数据外，ServiceCenter 还扩展了微服务依赖关系、黑白名单、Tag 标签和契约信息等元数据，他们之间的关系如下图所示：



这些微服务元数据不仅把实例信息分组管理起来，同时也满足了用户对微服务管理的需要。比如说微服务的黑白名单，业界的服务注册中心大多仅实现了 consumer 服务依赖的服务实例发现，但没法对一些安全要求高的服务设置白名单访问控制，ServiceCenter 支持给这类 provider 服务设置黑白名单过滤规则，防止恶意服务发现并 DDoS 攻击。

如何保障分布式系统的高可用性

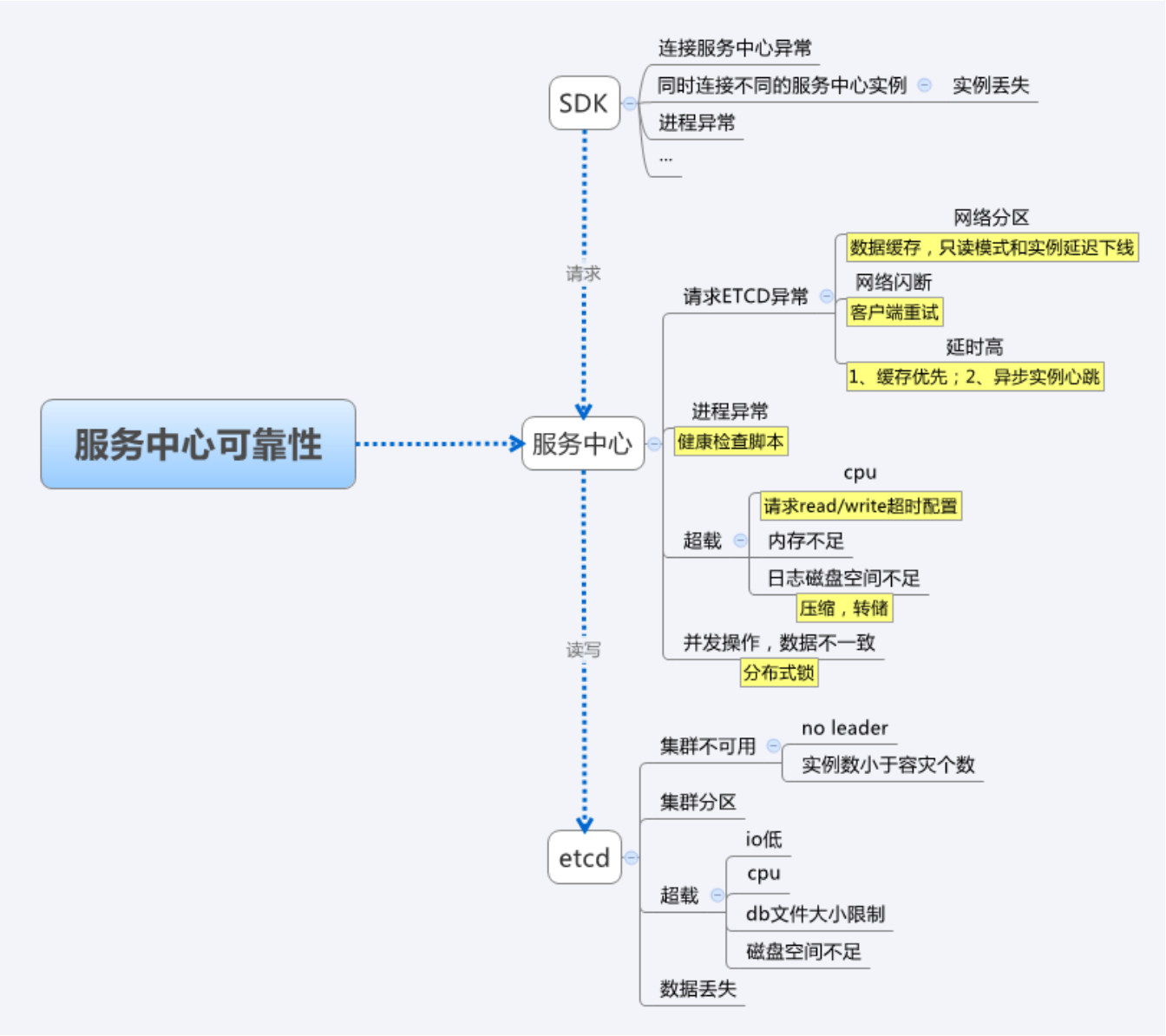
微服务及支撑它运行所需要的系统，从广义上来看，是一个典型的分布式系统。而分布式系统就一定不能逃脱 CAP 魔咒：

Consistency(一致性)，在分布式系统的各点同时保持数据的一致。 Availability(可用性)，每个请求都能接受到一个响应，无论响应成功或失败。Partition tolerance(分区容错性)，当出现网络分区故障时系统的容错能力。C 和 A 是每个系统都在努力追求的目标，但是 P 又是不能完全避免的。想同时满足三者是不可能的任务。

此后应运而生的 BASE 理论便成了现代化，特别是互联网分布式系统设计的准则：BASE 理论，Basically Available（基本可用）、Soft state（软状态）和 Eventually consistent（最终一致性）；

ServcieCenter 本身作为实现微服务架构中服务注册发现的组件，对整体微服务系统起到至关重要的作用，并且其本身也是一个分布式的系统。因此后面我们通过对一个微服务系统中常见故障模式的剖析来介绍在 CSE 中对 BASE 的实现。

1. 常见的故障模式



这里列出了 ServiceComb 架构中常见的故障以及对应的自保护机制，由于后端存储使用 ETCD，本身也有一定的故障恢复能力，具体可以访问官网了解；现在简单了解下 ServiceComb SDK（简称：SDK）和 ServiceCenter 是如何实现高可用的。

- **SDK：**

1. 与 ServiceCenter 网络分区：SDK 提供实例缓存机制，保证业务继续可以用。
2. 业务实例不可达：SDK 提供客户端负载均衡能力，快速将请求路由到可用的业务实

例，同时也支持提供一系列的容错策略，控制流量继续流向异常的实例。

• **ServiceCenter：**

1. 进程异常：ServiceCenter 提供监听 127.0.0.1 的本地 HTTP 健康检查接口/health，当监控脚本调用该接口时，ServiceCenter 内部会进行 etcd 可用性检查、当前管理资源容量超限检查和平均延时等检查。当以上某个指标发生异常时，接口将返回错误，便于监控系统及时感知。

2. 环境约束：一般有 CPU 负载高、内存不足和磁盘空间不足等因素，ServiceCenter 针对环境不稳定或资源不足的场景，除了自动触发自保护机制以外，还有提供一些可定制策略尽量保证自身可服务在这类环境中。如：针对 CPU 负载高的环境，ServiceCenter 支持配置请求读写超时，最大限度保证请求可以正常处理完毕，针对磁盘空间资源不够场景，ServiceCenter 自身提供定时日志压缩转储能力，尽量减少磁盘空间占用。

3. 并发操作：针对并发向 ServiceCenter 发写请求场景，需要保证数据一致。为此 ServiceCenter 利用了 etcd 本身提供数据强一致性能力，保证每一次写操作是原子操作；对于复杂的分布式业务场景，ServiceCenter 内部还提供了分布式锁组件，防止分布式并发写数据不一致的问题。

4. 网络分区：微服务依赖常见问题，一般是依赖的服务不可用和相互之间网络故障问题。针对前者，松耦合设计，ServiceComb SDK（简称：SDK）和 ServiceCenter 均有提供缓存机制，保证运行态下，SDK 到 ServiceCenter、ServiceCenter 到 etcd 各段间互不干扰；针对后者，ServiceCenter 除了提供常见的客户端重试外，还提供异步心跳和自我保护机制。

可见 ServiceComb 从设计上已经考虑了高可用，下面分别从 SDK 到 ServiceCenter 和 ServiceCenter 到 etcd 两个层面来说明 ServiceComb 如何实现的。

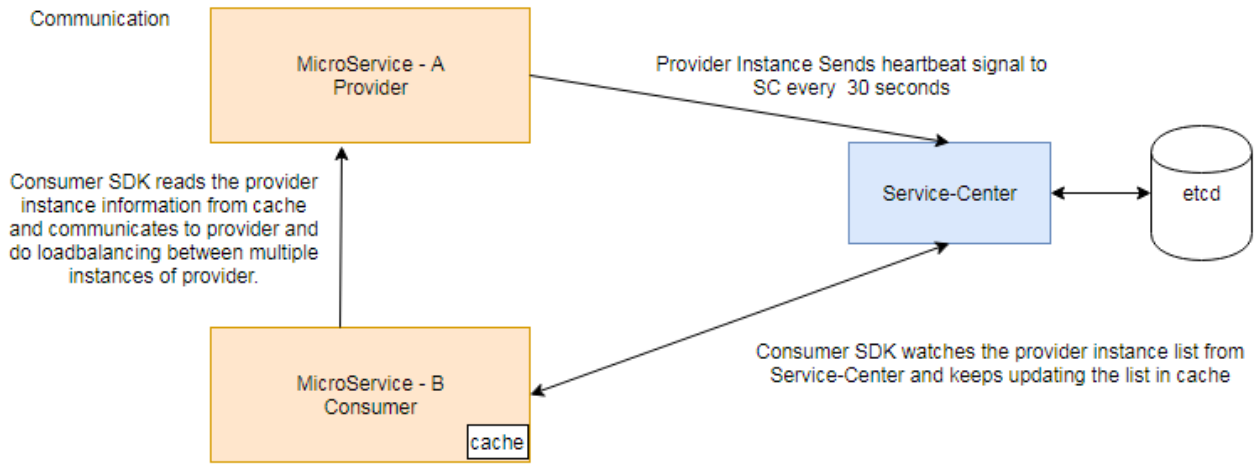
2. 保护机制

- **从 SDK 到 ServiceCenter**

1. 实例缓存机制

基于 SDK 开发的微服务，会在第一次消费 Provider 微服务时，会进行一次实例发现操作，此时内部会请求 ServiceCenter 拉取 Provider 当前存活的实例集合，并保存到内存缓存当中，后续消费请求就依据该缓存实例集合，按照自定义的路由逻辑发送到 Provider 的一个实例服务中。

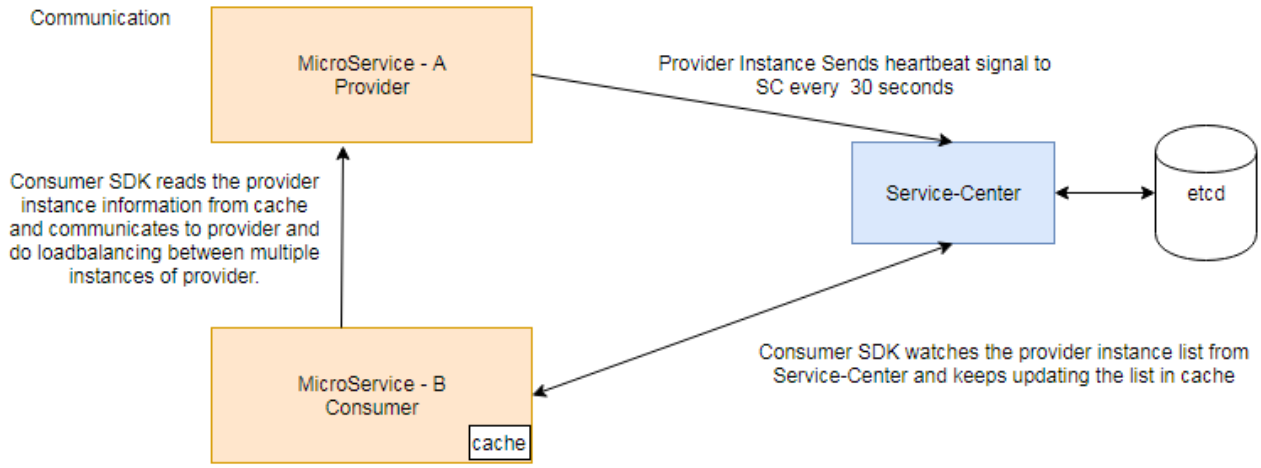
这样处理的好处是，已经运行态的 SDK 进程，始终保留一份实例缓存；虽然暂时无法感知实例变化及时刷新缓存，但当重新连上 ServiceCenter 后会触发一缓存刷新，保证实例缓存是最终有效的；在此过程中 SDK 保证了业务始终可用。



2. 心跳保活机制

ServiceCenter 的微服务实例设计上是存在老化时间的，SDK 通过进程上报实例心跳的方式保活，当 Provider 端与 ServiceCenter 之间心跳上报无法保持时，实例自动老化，此时 ServiceCenter 会通知所有 Consumer 实例下线通知，这样 Consumer 刷新本地缓存，后续请求就不会请求到该实例，也就微服务的实现动态发现能力。

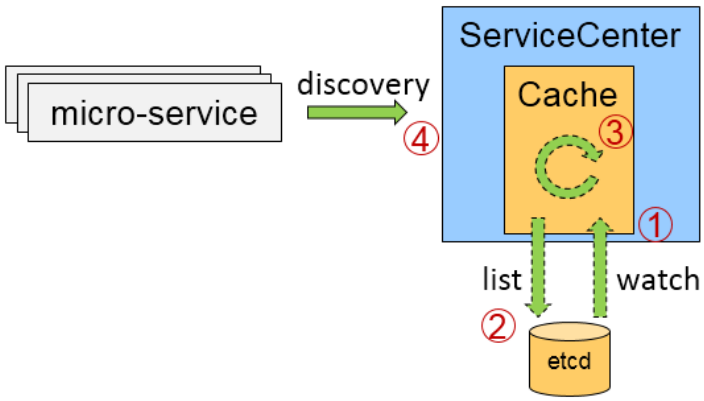
这样做的好处是，解决了业务某个实例进程下线或异常时，依赖其业务的微服务实例，会在可容忍时间内，切换调用的目标实例，相关业务依然可用。



- **从 ServiceCenter 到 etcd**

1. 异步缓存机制

在 ServiceCenter 内部，因为本身不存储数据，如果设计上单纯的仅作为一个 Proxy 服务转发外部请求到 etcd，这样的设计可以说是不可靠的，原因是因为一旦后端服务出现故障或网络访问故障，必将导致 ServiceCenter 服务不可用，从而引起客户端实例信息无法正确拉取和刷新的问题。所以在设计之初，ServiceCenter 引入了缓存机制。



1) 启动之初，ServiceCenter 会与 etcd 建立长连接（watch），并实时监听资源的变化。

2) 每次 watch 前，为防止建立连接时间窗内发生资源变化，ServiceCenter 无法监听到这些事件，会进行一次全量 list 查询资源操作。

3) 运行过程中，List & watch 所得到的资源变化会与本地缓存比对，并刷新本地缓存。

4) 微服务的实例发现或静态数据查询均使用本地缓存优先的机制。

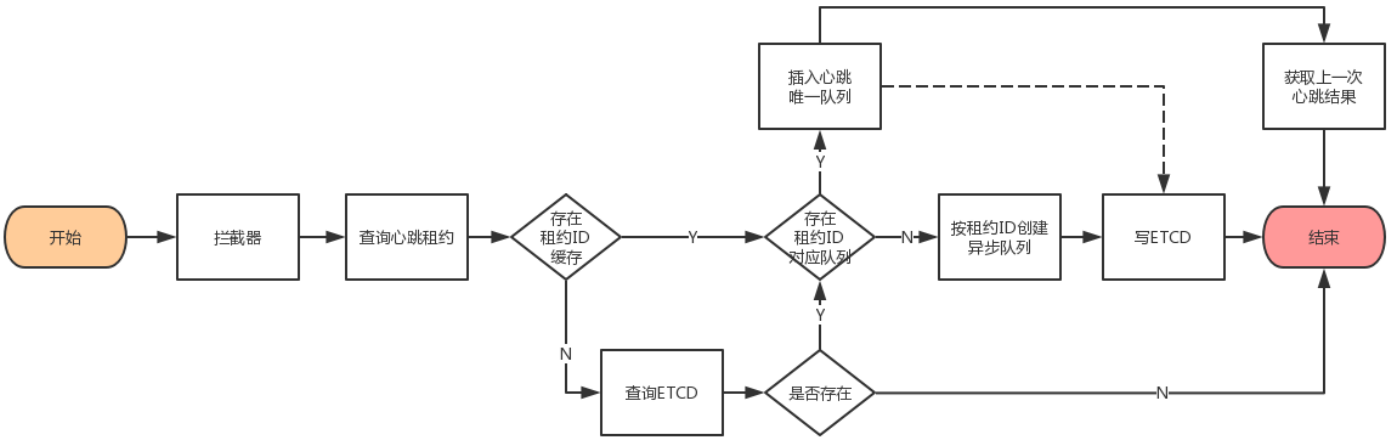
异步刷新缓存机制，可以让 ServiceCenter 与 etcd 的缓存同步是异步的，微服务与 ServiceCenter 间的读请求，基本上是不会因为 etcd 不可用而阻塞的；虽然在资源刷新到 ServiceCenter watch 到事件这段时间内，会存在一定的对外呈现资源数据更新延时，但这是可容忍范围内的，且最终呈现数据一致；这样的设计即大大提升了 ServiceCenter 的吞吐量，同时保证了自身高可用。

2. 异步心跳机制

中心化设计，另一个难题是，性能问题，基于心跳保活机制，随着实例数量增加，ServiceCenter 处理心跳的压力就会越大；要优化心跳处理流程， 首先了解下 ServiceCenter 怎么实现实例心跳的。

实例心跳，主要是为了让 ServiceCenter 延长实例的下线时间，在 ServiceCenter 内部，利用了 etcd 的 KeyValue 的 lease 机制来计算实例是否过期。lease 机制的运作方式是，需保证在定义的 TTL 时间内，发送 keepalive 请求进行保持存储的数据不被 etcd 删除。基于这个机制，ServiceCenter 会给每个实例数据设置下线时间，外部通过调用心跳接口出发 ServiceCenter 对 etcd 的 keepalive 操作，保持实例不下线。

可以知道，这样设计会导致上报心跳方因 etcd 延时大而连接阻塞，上报频率越高，ServiceCenter 等待处理连接队列越长，最终导致 ServiceCenter 拒绝服务。



在这里 ServiceCenter 做了一点优化，异步心跳机制。ServiceCenter 会根据上报实例信息中的心跳频率和可容忍失败次数来推算出最终实例下线时间，比如：某实例定义了 30s 一次心跳频率，最大可容忍失败是 3 次，在 ServiceCenter 侧会将实例定义为 120s 后下线。另外，在实例上报第一次心跳之后，ServiceCenter 会马上产生一个长度为 1 的异步队列，进行请求 etcd，后续上报的心跳请求都会在放进队列后马上返回，返回结果是最近一次请求 etcd 刷新 lease 的结果，而如果队列已有在执行的 etcd 请求，新加入的请求将会丢弃。

这样处理的好处是持续的实例心跳到 ServiceCenter 不会因 etcd 延时而阻塞，使得 ServiceCenter 可处理心跳的能力大幅度的提高；虽然在延时范围内返回的心跳结果不是最新，但最终会更新到一致的结果。

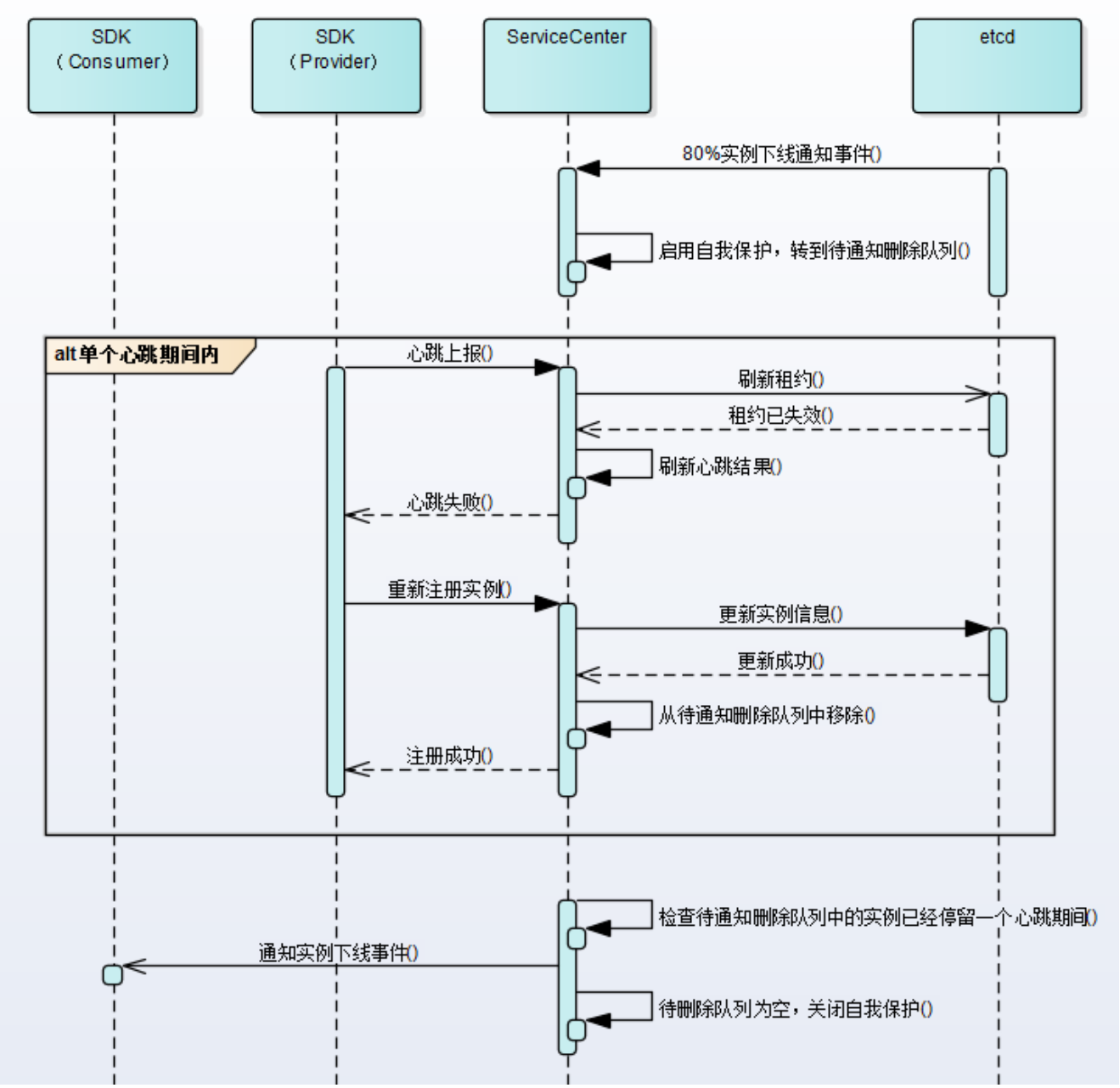
3. 自我保护机制

前面提到的缓存机制，保证了 ServiceCenter 在 etcd 出现网络分区故障时依然保持可读状态，ServiceCenter 的自我保护(Self-preservation)机制保证了 Provider 端与 ServiceCenter 在出现网络分区故障时依然保持业务可用。

现在可以假设这样的场景：全网大部分的 Provider 与 ServiceCenter 之间网络由于某种原因出现分区，Provider 心跳无法成功上报心跳。这样的情况下，在 ServiceCenter 中会出现大量的 Provider 实例信息老化下线消息，ServiceCenter 将 Provider 实例下线事件推送到全网大

部分的 Consumer 端，最终导致一个结果，用户业务瘫痪。可想而知对于 ServiceCenter 乃至于整个微服务框架是灾难性的。

为了解决这一问题，ServiceCenter 需要有一个自我保护机制（Self-preservation）：



1) ServiceCenter 在一个时间窗内监听到 etcd 有 80%的实例下线事件，会立即启动自我保护机制。

2) 保护期间内，所有下线事件均保存在待通知队列中。

3) 保护期间内，ServiceCenter 收到队列中实例上报注册信息则将其从队列中移除，否则

当实例租期到期，则推送实例下线通知事件到 consumer 服务。

4) 队列为空，则关闭自我保护机制。

有了自我保护机制后，即使 etcd 存储的数据全部丢失，这种极端场景下，SDK 与 ServiceCenter 之间可在不影响业务的前提下，做到数据自动恢复。虽然这个恢复是有损的，但在这种灾难场景下还能保持业务基本可用。

了解更多信息，请访问 [华为云微服务引擎 CSE 主页]