01 | 服务注册与发现：AP和CP，你选哪个？

你好，我是大明。今天我们来聊一聊微服务架构下的服务注册与发现。

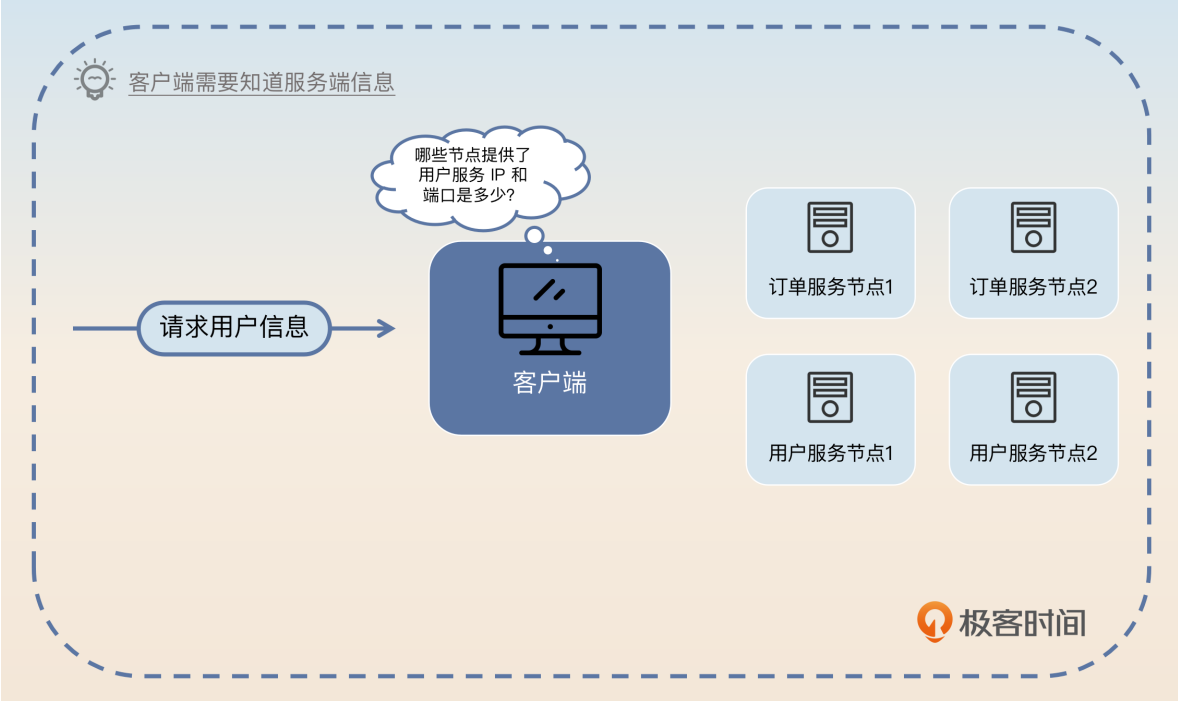
服务注册与发现在微服务架构中处于一个非常核心的地位，也是面试中的常见问题。不过因为微服务架构大行其道，现在我们多少都能回答出来一些服务注册与发现的内容，也因此不容易在面试中刷出亮点，拉开和其他面试者的差距。

所以这一节课我就要带着你深入剖析服务注册与发现，学习服务注册与发现的基本模型，然后在服务端崩溃检测、客户端容错和注册中心选型三个角度找到高可用微服务架构的亮点。

那么我们先来看看服务注册与发现的基本模型。

前置知识

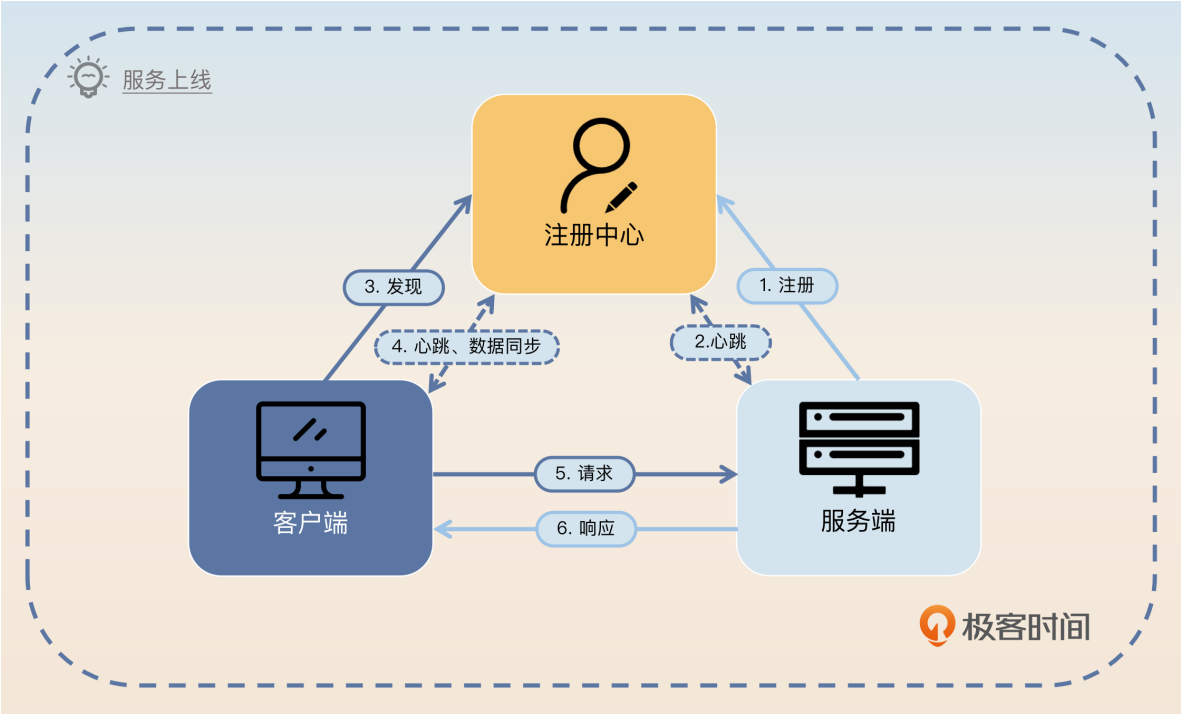
为什么我们会需要服务注册与发现呢？你设想这样一个场景，你的服务部署在不同的机房、不同的机器上，监听不同的端口。现在你的客户端收到了一个请求，要发送给服务端，那么你的客户端怎么知道哪些服务端能够处理这个请求呢？



举一个例子，你去一个陌生的城市出差，下班了想去吃个火锅，还得是重庆火锅。那么你怎么知道这个城市哪里有重庆火锅？

你可能会说，我在 App 里面搜一下。那么 App 又怎么知道这里有一家重庆火锅店呢？你继续说，这肯定是商家去这个 App 注册过了呀！对，服务注册与发现模型就是这样。你扮演了客户端的角色，火锅店扮演了服务端的角色，而 App 则是扮演了我们常说的注册中心的角色。

那么我们现在就容易理解基本的服务注册与发现模型了。



你可以看一下我给出的这张图，你要牢牢记住这张图还有里面的具体步骤。

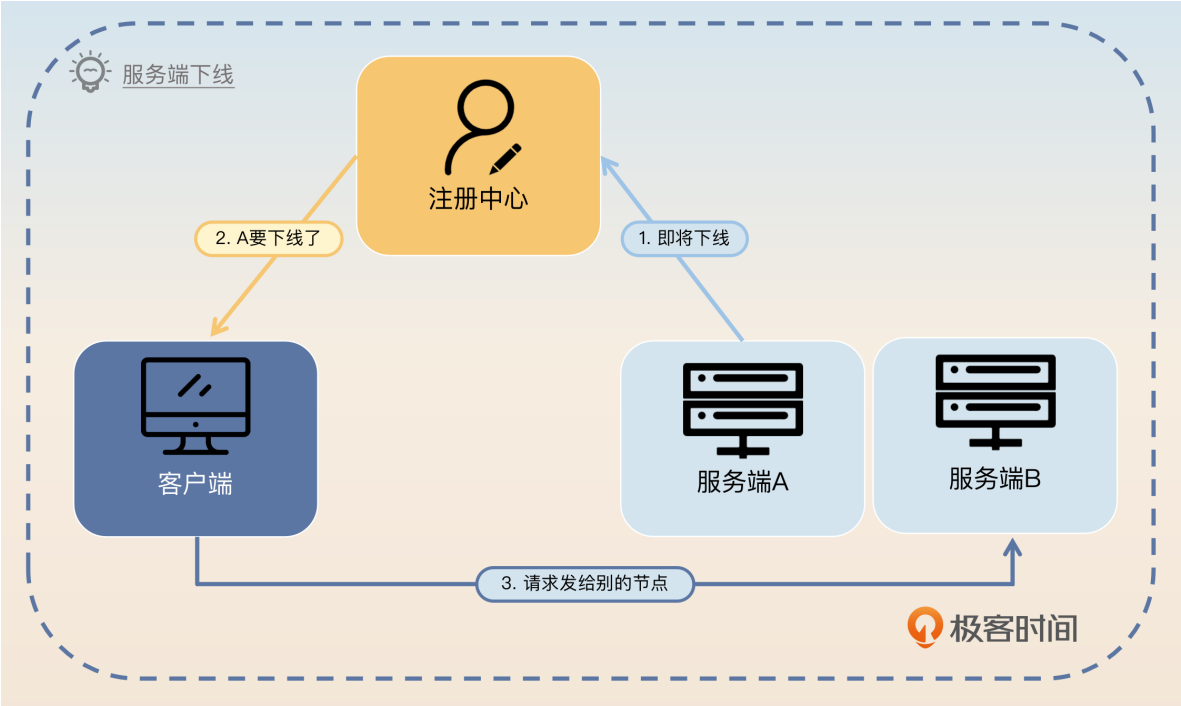
1. 服务端启动的时候，需要往注册中心里注册自身的信息，主要是定位信息。
2. 注册成功之后，注册中心和服务端要保持心跳。
3. 客户端第一次发起对某个服务的调用之前，要先找注册中心获得所有可用服务节点列表，随后客户端会在本地缓存每个服务对应的可用节点列表。
4. 客户端和注册中心要保持心跳和数据同步，后续服务端有任何变动，注册中心都会通知客户端，客户端会更新本地的可用节点列表。
5. 客户端发送请求。
6. 服务端返回响应。

上面的这个步骤你可以看作是一个“正向”的步骤，而对应的反向步骤则是指服务端下线的过程。

我们还是用前面的例子来描述，一家门店准备关张不再营业了，那么它需要做一些什么？显然它需要告诉 App 自己不再营业了，那么你在平台上也就再也搜索不到它了。

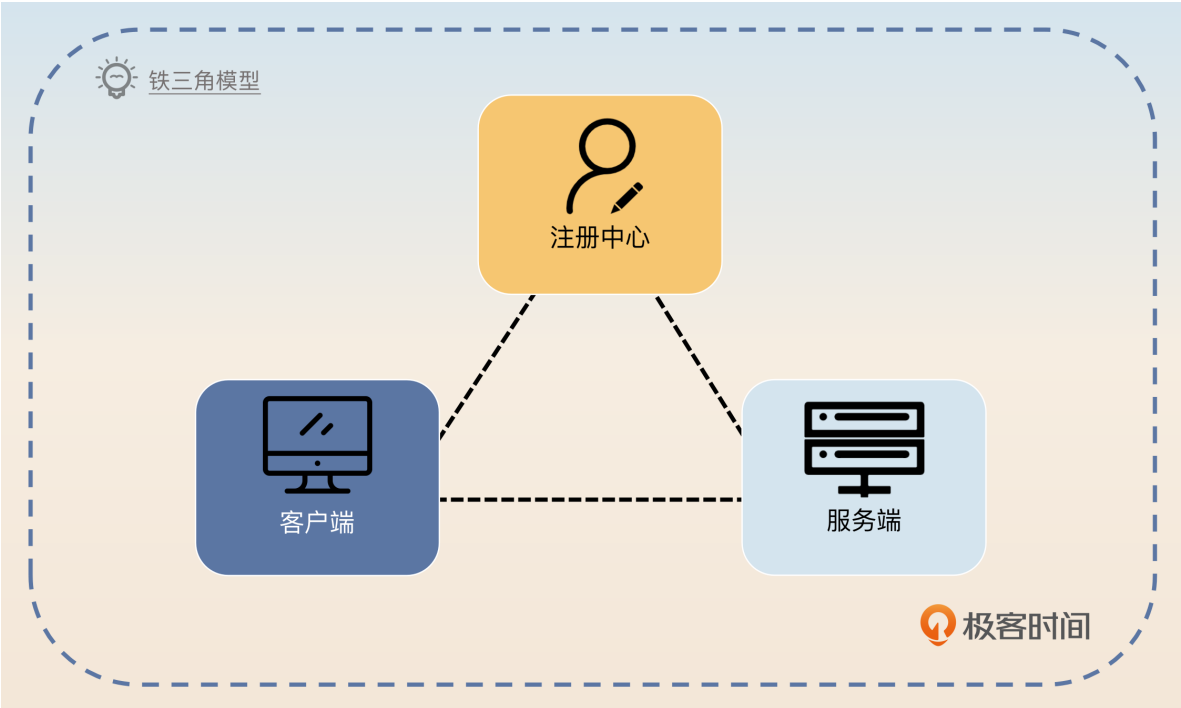
所以，服务端下线的过程可以总结为4步。

1. 服务端通知注册中心自己准备下线了。
2. 注册中心通知客户端某个服务端下线了。
3. 客户端收到通知之后，新来的请求就不会再给该服务端发过去。
4. 服务端等待一段时间之后，暂停服务并下线。



需要注意的是，服务端必须要等待一段时间才能下线。因为从它通知注册中心自己要下线，到客户端收到通知，是有一段延时的，这段延时就是服务端要等待的最小时间。

如果你觉得这些步骤很复杂，那么我可以教你一个小技巧。你可以把整个模型看作是三角形，三个顶点分别是客户端、注册中心和服务端。三角形的三条边分别是客户端-注册中心，注册中心-服务端，客户端-服务端。而后面我们讨论的高可用方案，无非就是仔细思考三角形的任何一个顶点，或者任何一条边出问题了该怎么办。



面试准备

在面试前，如果你们公司确实是使用了注册中心，那么你要弄清楚一些数据和信息。

- 你们用了什么中间件作为注册中心以及该中间件的优缺点。确保自己在回答“你为什么用某个中间价作为注册中心”的时候，能够综合这些优缺点来回答。
- 注册中心的集群规模。
- 读写 QPS（每秒查询率）。

- 机器性能，如 CPU 和内存大小。
- 最好准备一个注册中心出故障之后你排查和后续优化的案例。在讨论使用注册中心的注意事项，或者遇到过什么 Bug 的时候可以用这个案例。

如果你所在的公司没有采用微服务架构，那么你可以在 ZooKeeper、Nacos 或者 etcd 里面选择一个大概学习一下它们的基本特性。在面试的时候你可以用它们来解释注册中心。这样就算你没接触过服务注册与发现，但是你对此也是有相当深入的理解的。

在面试过程中，你可以尝试从这些角度把话题引到服务注册与发现这个主题上。

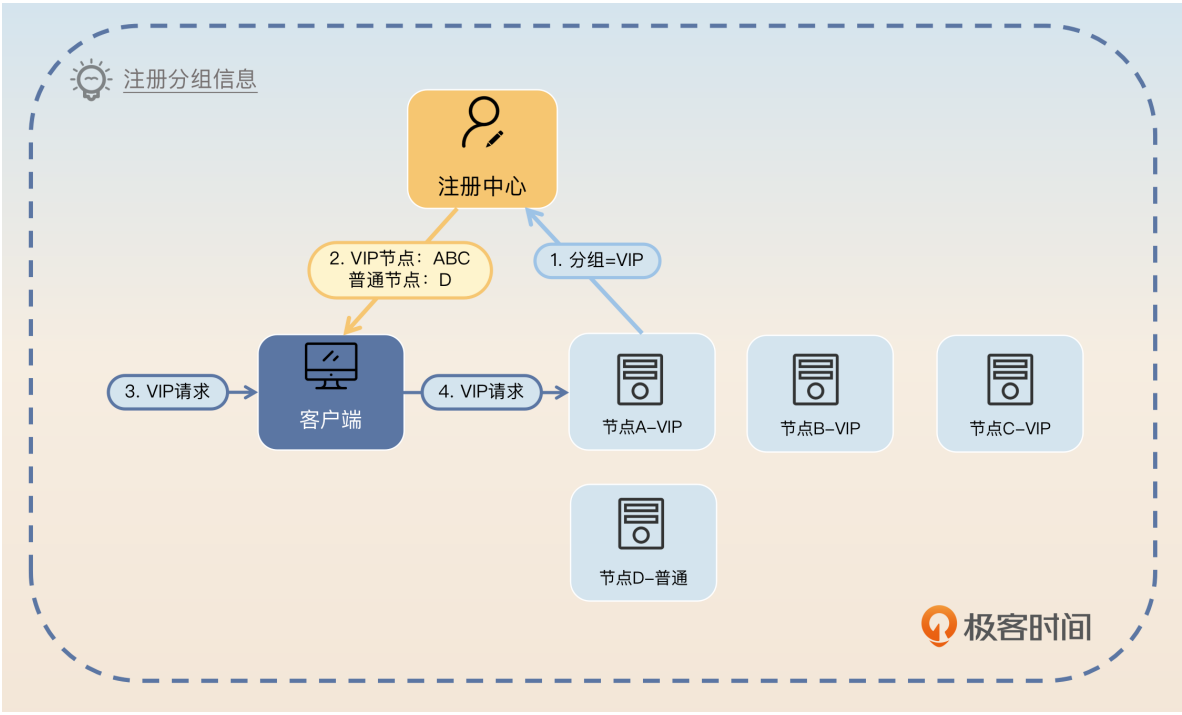
- 第一种情况， **面试官问到了某一个可以作为注册中心的中间件。**举例来说，如果你用 ZooKeeper 作为注册中心，那么如果面试官问到了 ZooKeeper，你可以主动提起你把它作为了注册中心；如果面试官问了 etcd，那么你可以主动提起 etcd 虽好，但是你用的是ZooKeeper。这个时候面试官很有可能会继续追问你，为什么最终选择 ZooKeeper 作为注册中心，这时候说一下它的优缺点就好了。
- 还有一种情况， **面试官问了你微服务高可用的问题**，那么你可以把高可用的服务注册与发现作为保证整个微服务架构高可用的一个环节来叙述。

基本模型

一般而言，在最开始的阶段，面试官会问你“你知道服务注册与发现吗？”或者“你知道注册中心吗？”等问题，其实都是希望你回答服务注册与发现的基本模型。

那么你可以回答前置知识里面的服务上线和服务下线这两个流程的具体步骤，而后可以简单描述一下你所在公司的注册中心，也就是罗列一下你准备的那些数据和信息。基本内容说完之后，你可以先浅刷一个亮点，关键词是 **注册数据。**

在说第一个步骤的时候，我提到“主要是定位信息”。既然用到了关键词“主要”，那自然有不那么主要的数据。非主要数据取决于微服务框架的功能特性。例如常见的分组功能，就是依赖于服务端在注册的时候同时注册自己的分组信息。



所以你可以用一个例子来解释，关键词是 **分组。**

> 服务端注册的数据除了定位信息是必需的以外，剩下需要什么数据都是根据微服务框架本身的功能和业务来设计的。比如说很多微服务框架支持分组功能，那么就可以让服务端在注册的时候同时注册自己的分组信息，比如说当前节点是 VIP 节点。那么客户端在收到 VIP 请求之后就会把请求发给 VIP 节点。

这一段说完之后，你要稍微总结一下，引导面试官追问下去。

> 服务注册与发现的整个模型比较简单，不过要在实践中做到高可用还是很不容易的。

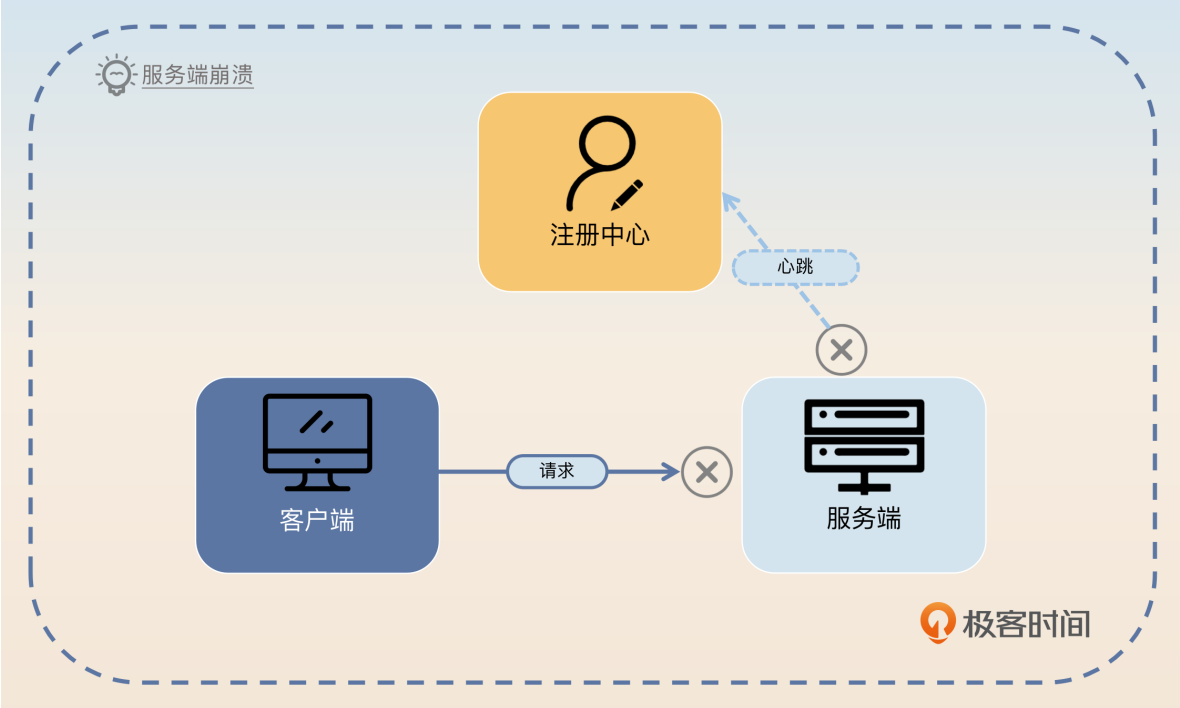
至于为什么不容易、怎么不容易你就等着面试官继续问。而 **高可用** 就是我们要刷的亮点。

高可用

不出所料的话，面试官就可能追问：“服务注册与发现怎么保证高可用呢？”，那么你就可以回答三个点，高可用的服务注册与发现要围绕 **注册服务端崩溃检测、客户端容错和注册中心选型** 三个方面进行。接下来我们一个点一个点地看。

服务端崩溃检测

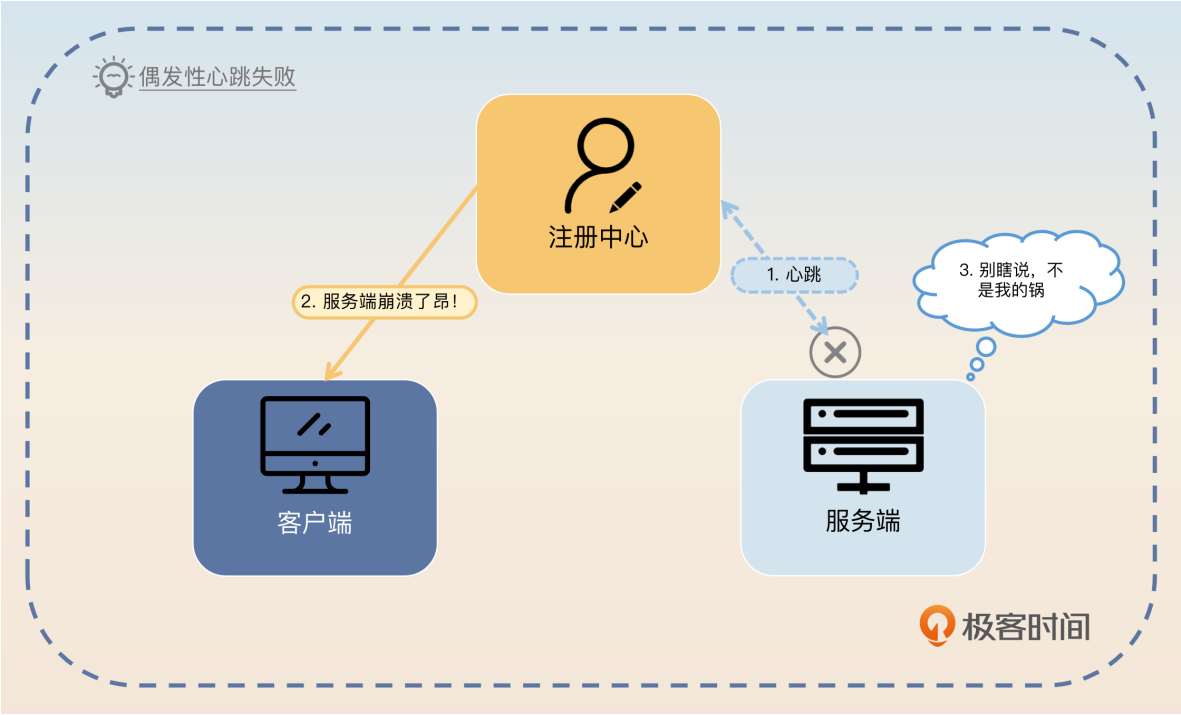
我在基本模型里面说到在正常情况下，服务端下线都需要通知注册中心。那么万一服务端宕机了呢？比如说运维大哥不小心一脚把服务器电源线踢掉了，服务器直接停电了。在这种情况下，服务端是没办法通知注册中心的，注册中心自然也就不会通知客户端。那么客户端就会继续把请求发送给服务端，而这些请求显然都会失败。



因此为了提高可用性，需要让注册中心尽快发现服务端已经崩溃了，而后通知客户端。所以问题的关键就在于 **注册中心怎么判断服务端已经崩溃了。**

你可能在上面这张图片里注意到了，服务端崩溃之后注册中心和服务端之间的心跳就无法继续保持了。所以你得出一个简单的结论： **如果注册中心和服务端之间的心跳断了，就认为服务端已经崩溃了。**

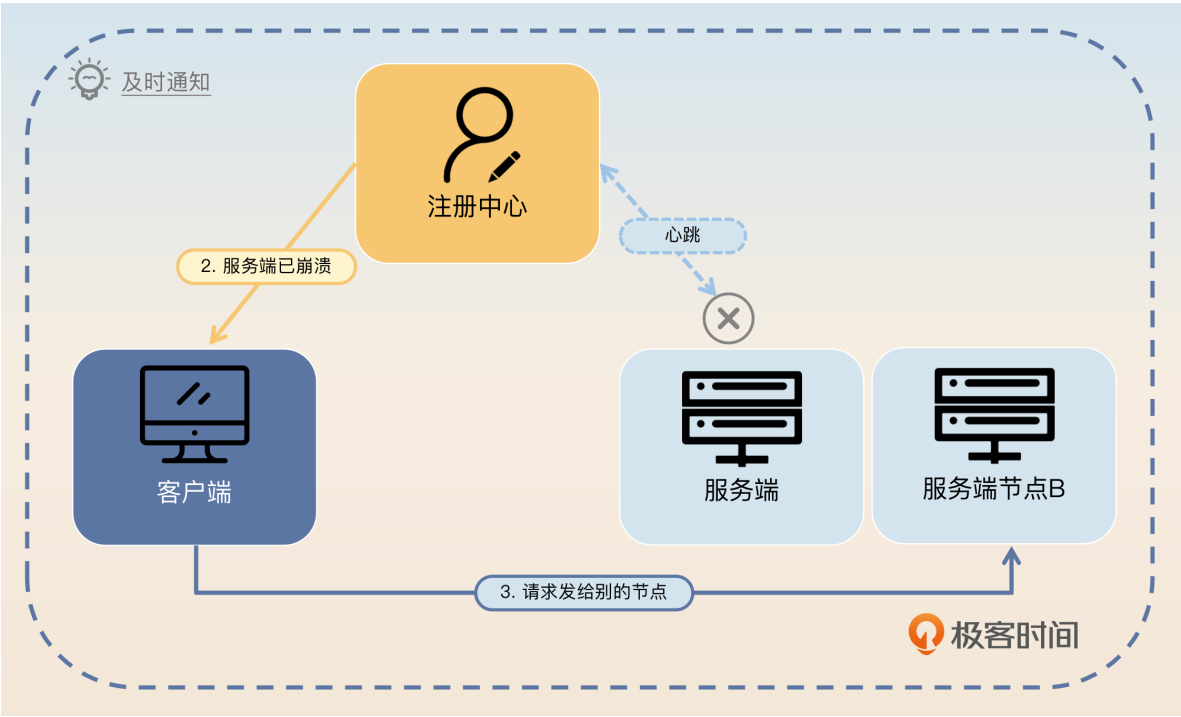
但是，如果注册中心和服务端之间的网络出现偶发性的抖动，那么心跳也会失败。此时服务端并没有真的崩溃，还活得好好的。



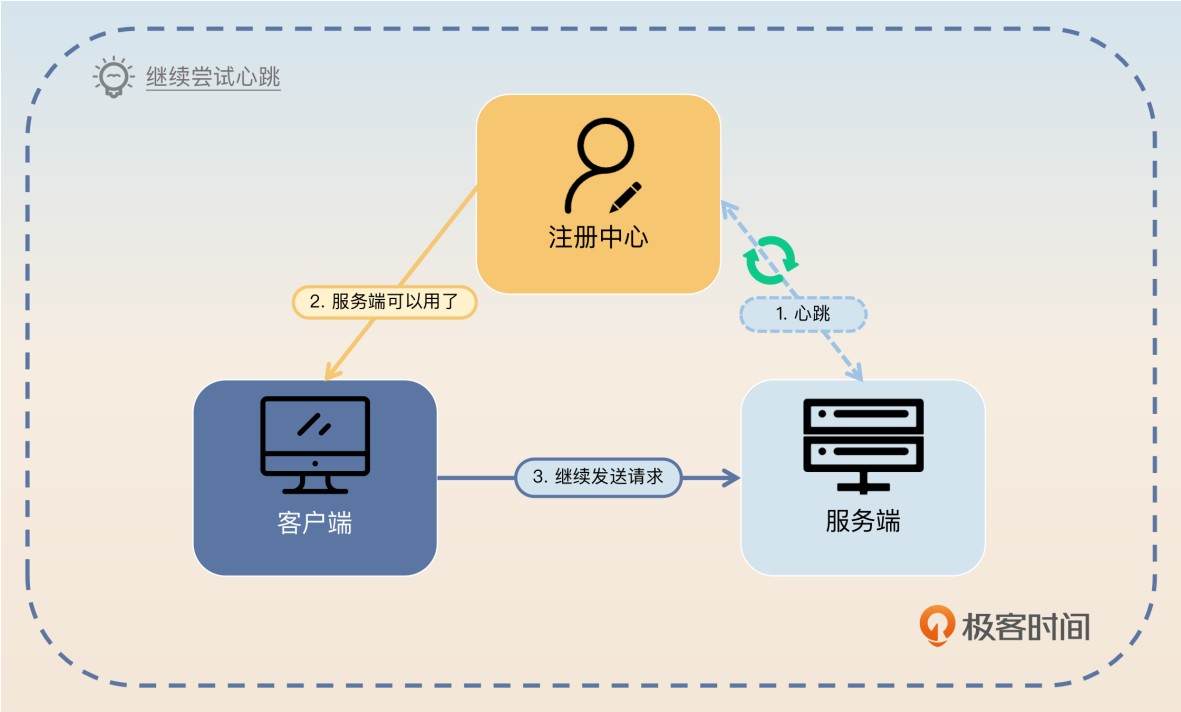
显然，心跳断了则服务端崩溃的判断并不能成立。这时候你可能会想到能不能多发几次心跳呢？答案是可以，但是次数越多，心跳间隔越长，注册中心断定服务端已经崩溃的时间就越长。而时间越长，就有越多请求发送给服务端。万一这个时候服务端真的崩溃了，这些请求都会失败。所以这就陷入两难境地了。要么是误以为服务端崩溃，要么是误以为服务端还活着。

那么怎么走出这个窘境呢？

一方面，注册中心在和服务端进行心跳的时候失败了，就要 **立刻通知客户端** 该服务端已经不可用了，那么客户端就不会再发请求过来。



另外一方面， **注册中心还要继续往服务端发心跳。**如果只是偶发性的心跳失败，那么注册中心后面心跳是肯定能够连上的，这时候注册中心再通知客户端这个服务端是可用的。



不过注册中心并不是无限制发心跳直到连接上，而是发了一段时间之后发现心跳还是失败就不再发了，这意味着注册中心认定服务端彻底崩溃了。在彻底崩溃的场景下，注册中心不需要再次通知客户端，因为在之前注册中心就已经通知过了。

所以关键词就是 **心跳**，你可以这样回答。

> 影响到可用性的另一个关键点是注册中心需要尽快发现服务端宕机。在基本模型里面，如果服务端突然宕机，那么服务端是来不及通知注册中心的。所以注册中心需要有一种检测机制，判断服务端有没有崩溃。在服务端崩溃的情况下，要及时通知客户端，不然客户端就会继续把请求发送到已经崩溃的节点上。

> 这种检测就是利用心跳来进行的。当注册中心发现和服务端的心跳失败了，那么它就应该认为服务端可能已经崩溃了，就立刻通知客户端停止使用该服务端。但是这种失败可能是偶发性的失败，比如说因为网络偶尔不稳定造成的。所以注册中心要继续保持心跳。如果几次心跳都失败了，那么就可以认为服务端已经彻底不可用了。但是如果心跳再次恢复了，那么注册中心就要再次告诉客户端这个服务端是可用的。

回答到这里，亮点已经有了，不过你还可以继续钓鱼，稍微升华一下。

> 实际上，在所有有心跳机制的分布式系统里面判断节点是否崩溃都是一个棘手的问题。比如说心跳失败了要不要继续重试，是立刻重试还是间隔重试，重试的话试几次？

> 理论上来说，在心跳失败之后如果不进行重试就直接判定服务端崩溃，那么就难以处理偶发性网络不通的问题。而如果要重试，比如说在注册中心和服务端的模型里面，重试三次，而且重试间隔是十秒钟，那么注册中心确定服务端崩溃就需要三十秒。在这三十秒内，客户端估计有成千上万的请求尝试发到崩溃的服务端，结果都失败了。

这时候，面试官很自然地就会觉得不要搞重试间隔，而是直接发起连续几次重试，这时候你就要无情地击碎这种幻想。

> 如果不考虑重试间隔的话，就难以避开偶发性的失败。比如说注册中心和服务端之间网络抖动，那么第一次心跳失败之后，你立刻重试多半也是失败的，因为此时网络很可能还是不稳定。

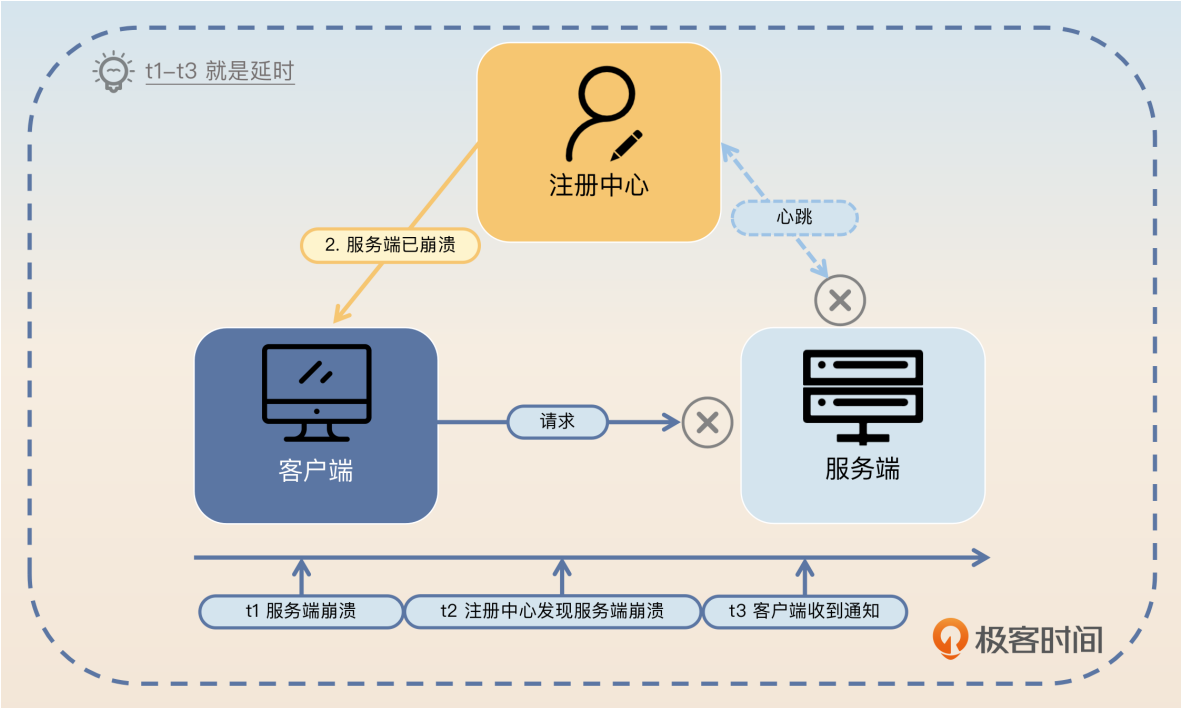
> 所以比较好的策略是立刻重试几次，如果都失败了就再间隔一段时间继续重试。所有的重试机制实际上也是要谨慎考虑重试次数和重试间隔的，确保在业务可以接受的范围内重试成功。不过再怎么样，从服务端崩溃到客户端知道，中间总是存在一个时间误差的，这时候就需要客户端来做容错了。

这个回答里面，最后的一句话，就是为了引出下面这个亮点：客户端容错。

客户端容错

客户端容错是指 **尽量在注册中心或者服务端节点出现问题的时候，依旧保证请求能够发送到正确的服务端节点上。**

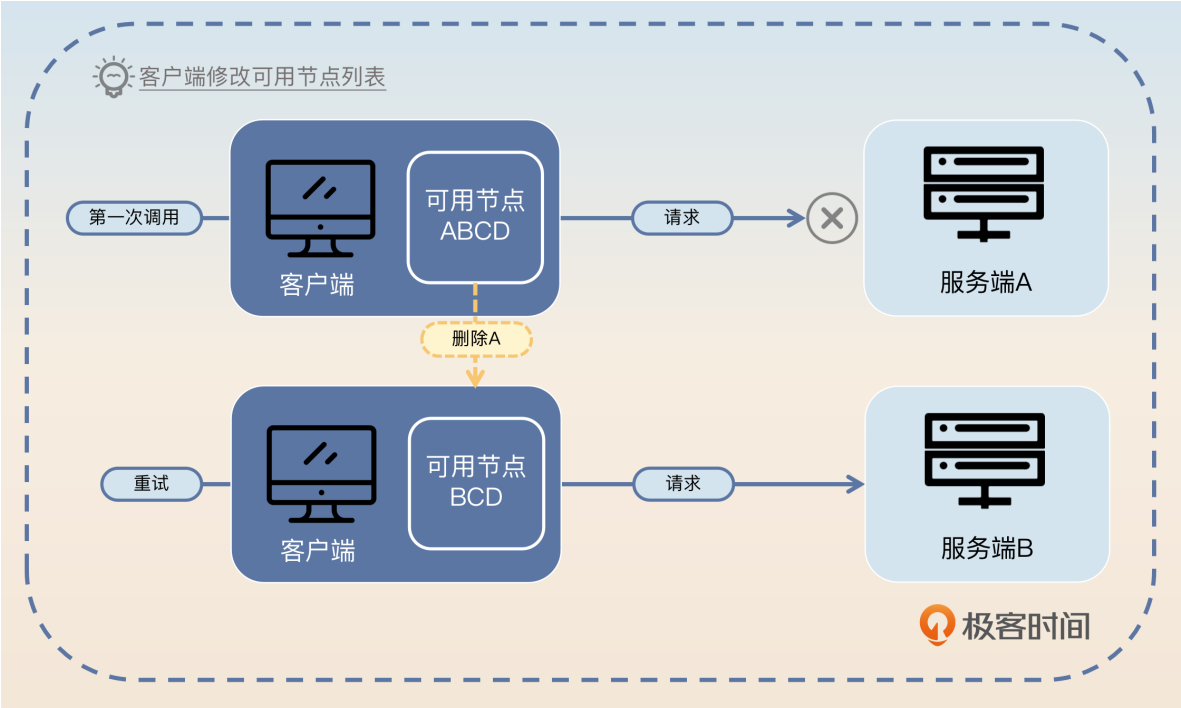
在前一个亮点里面，你已经知道从服务端崩溃到客户端最终知道是有一段延时的。在这段延时内，客户端还是会把请求发送到已经崩溃的服务端节点上。



所以你要紧接着前面刷的亮点继续回答，关键词是 **换节点**，也就是所谓的 **failover**。

> 客户端容错第一个要考虑的是如果某个服务端节点崩溃了该怎么办。在服务端节点崩溃之后，到注册中心发现，再到客户端收到通知，是存在一段延时的，这个延时是能算出来的。在这段延时内，客户端发送请求给这个服务端节点都会失败。

> 这个时候需要客户端来做一些容错。一般的策略是客户端在发现调不通之后，应该尝试换另外一个节点进行重试。如果客户端上的服务发现组件或者负载均衡器能够根据调用结果来做一些容错的话，那么它们应该要尝试将这个节点挪出可用节点列表，在短时间内不要再使用这个节点了。后面再考虑将这个节点挪回去。

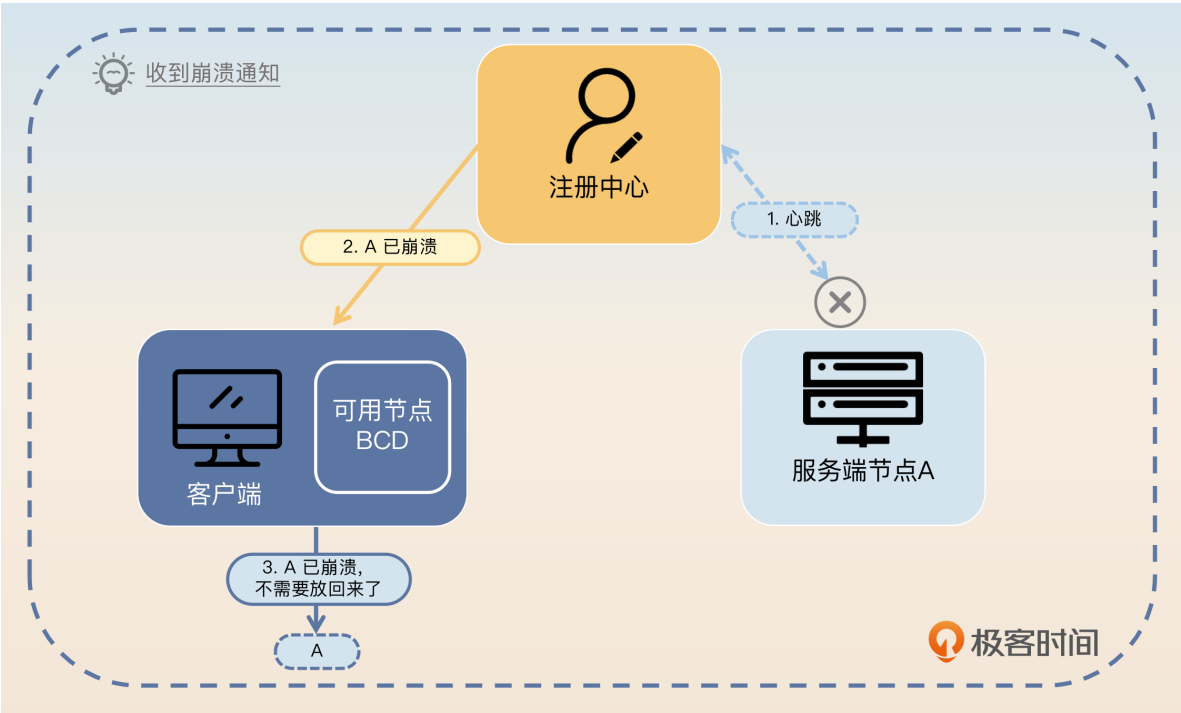


在上面那段话中，我留了两个口子。第一个是延时怎么计算，非常简单，你从图里面就能看出来。

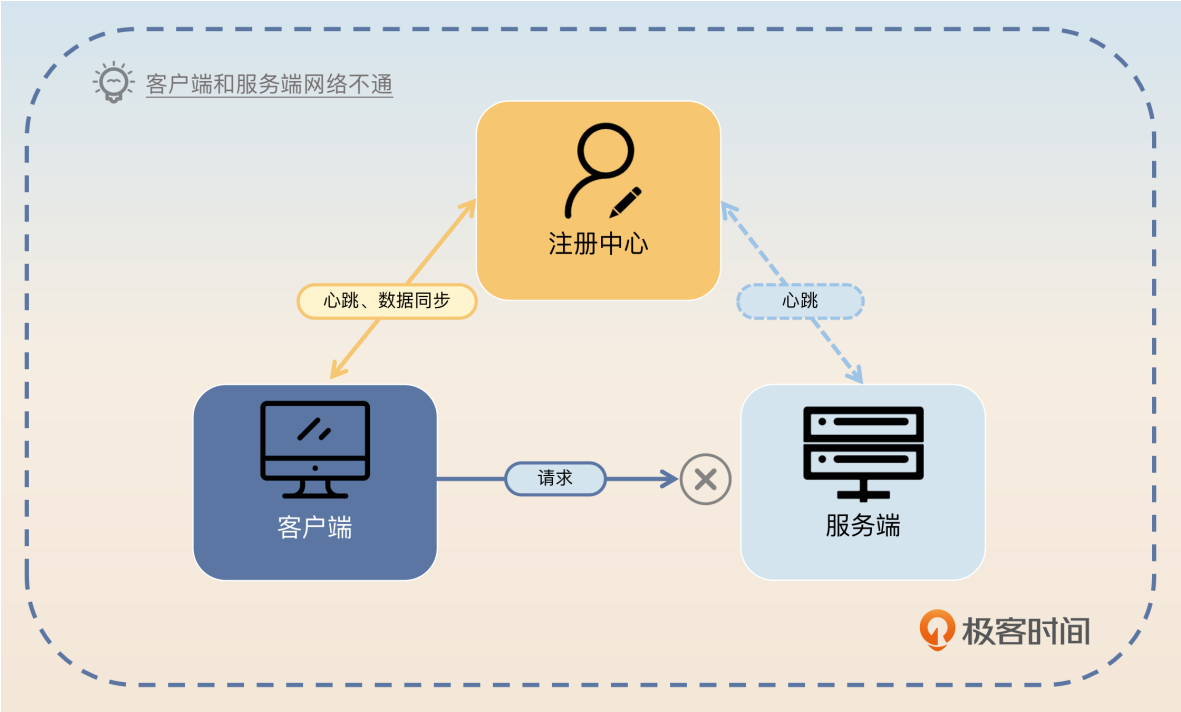
> 最坏的情况下，延时等于服务端和注册中心心跳间隔加上注册中心通知客户端的时间。大多数时候，注册中心通知客户端都是很快的，在毫秒级以内。因此可以认为服务端和注册中心的心跳间隔就是这个延时。

第二个点就是什么时候再将这个节点挪回可用列表，在上图中就是 A 什么时候会被重新放回可用列表。

显然，如果注册中心最终发现服务端崩溃，然后通知了客户端，那么客户端就不用放回去了。等到注册中心发现服务端再次恢复了，那么注册中心会通知客户端，此时客户端更新可用节点列表就可以了。



但是有一种情况是需要客户端主动检测的。这种情况就是服务端节点还活着，注册中心也还活着，唯独**客户端和服务端之间的网络有问题**，导致客户端调用不通。



在这种情况下，类似于注册中心和服务端心跳失败，客户端也要朝着那个疑似崩溃的服务端节点继续发送心跳。如果心跳成功了，就将节点放回可用列表。如果连续几次心跳都没有成功，那么就不用放回去了，直接认为这个节点已经崩溃了。

这个分析也适用于客户端和注册中心心跳失败的场景。很显然在这种情况下，客户端可以直接使用本地缓存的可用节点列表，而后如果调不通了则处理方式完全一样。但是不同的是，如果客户端长期连不上注册中心，那么客户端本身应该考虑整个退出。

注册中心选型

注册中心选型类似于其他中间件选型，要考虑的因素非常多。比如说中间件成熟度、社区活跃度、性能等因素。相比之下，注册中心更加关注 CAP 中选 CP 还是选 AP 的问题。

> C：Consistency，数据一致性
>
> A：Availability，服务可用性
>
> P：Partition-tolerance，分区容错性
>
> CAP 理论告诉我们，一个分布式系统不可能同时满足数据一致性、服务可用性和分区容错性这三个基本需求，最多只能同时满足其中的两个。——来自 《深入浅出分布式原理》

简单来说，选择 CP 就是选了一致性和分区容错性，而选择 AP 就相当于选了可用性和分区容错性。

看上去 P 分区容错性是肯定要选的，那么剩下的就是选 C（一致性） 还是选 A（可用性） 了。那么你要先理解在注册中心选型里面，一致性和可用性究竟哪个更加重要？标准答案是可用性，也就意味着 CP 和 AP 你应该选 AP。

前面我们讨论了客户端容错，那么显然在选择 AP 的情况下，客户端就可能拿到错误的可用节点列表。如果客户端将请求发到错误的可用节点上，就会出现错误，此时客户端自然可以执行容错，换一个可用节点重试。

所以我们要抓住关键词 **客户端容错** 进行回答。

> 在注册中心选型上，重要的是 CAP 原理中应该选择 AP，比如说 Eureka，又或者 Nacos 启用 AP 模式。

万一你公司并没有使用 AP 模型的注册中心，比如说用了 CP 模型的 ZooKeeper，那么你就可以进一步解释，关键词是 **体量小**。

> 我司之所以用 ZooKeeper，主要是因为我司体量小，集群规模也不大，ZooKeeper 虽然不是 AP 的，但是在这种体量下也够用了。不过我也尝试在公司内部推动看能否换一个中间件，比如说用 Nacos 的 AP 模式。

面试思路总结

最后，让我们来总结一下这节课的重点内容。

这节课我们主要解决的是服务注册与发现的问题。我给出了基本的服务注册与发现模型，然后从服务端崩溃检测、客户端容错、注册中心选型三个角度来保证了服务注册与发现的高可用。其中我提到了几个关键词，分别是 **注册数据、分组、心跳、换节点、客户端容错、体量小。** 你可以从这几个关键词出发，根据自己的项目经验，梳理思路。

最后我再提醒一下，如果你觉得服务注册与发现实在难以记忆，可以 **把整个模型想成是一个三角形，而解决高可用问题的关键就是这个三角形任何一条边出问题了该怎么办。**我非常建议你画一画这个三角形，并且手写一下你能想到的各种容错措施。

思考题

最后你来思考2个问题。

1. 我在客户端容错里提到这个分析也适用于注册中心崩溃，你能组织一下语言尝试回答“如果注册中心崩溃，你的系统会怎样？”这个问题吗？
2. 你可以再举出一个心跳频率、心跳重试机制对系统可用性影响的例子吗？

欢迎你把你的答案分享在评论区，也欢迎你把这节课的内容分享给需要的朋友，我们下节课再见！