20 | 如何通过监控快速发现问题?

前三讲基于“**防备上游、做好自己、怀疑下游**”的准则，讲解了如何通过系统设计、部署，以及代码编写的方式来构建一个更加高可用的后台系统。

基于上述三个准则提出的方案可以预防部分问题，但百密一疏，即使我们做了很多防护措施，仍无法保证绝对安全，避免问题发生。此时作为系统的负责人，你需要在第一时间，也就是用户感知前发现问题。

发现问题的方法便是**监控**，本讲将介绍如何设计微服务的监控，帮助你在日常系统维护时，更快地、自动地发现问题。

什么是监控

监控是指对被监控体的运行状态数据进行持续地审查，并设置运行状态数据不符合要求的阈值，对不符合阈值的运行状态主动报警的一种方式。被监控体的运行状态数据通常以如下图 1 中 XY 轴的格式进行展示。

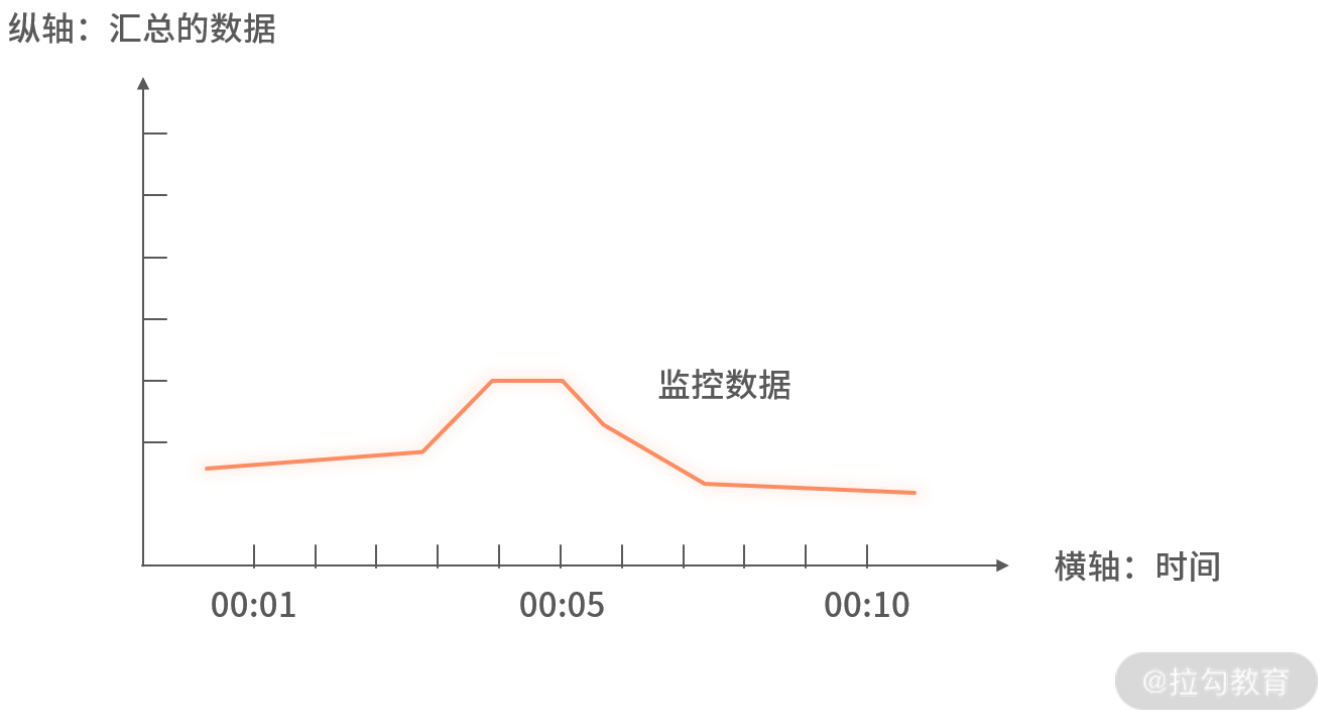

图 1：XY 轴格式的运行状态数据图

上图 1 的 X 轴表示时间，由固定的间隔组成，此间隔可以是秒级或分钟级。Y 轴表示在该时间间隔里的运行状态数据的汇聚，它可以是间隔数据的累加、平均值、最大值等方式。

下面将对几种常见的被监控体，以及它们的运行状态数据、阈值设置等相关内容进行一一介绍。因为篇幅原因，关于其他类型的被监控体及对应的指标，本讲不再赘述，你可以参考拉勾教育关于“**监控主题**”的专栏，比如《31 讲带你搞懂 SkyWalking》、《分布式链路追踪实战》。

1. 次数监控

次数监控中被监控体就是次数，具体指微服务里各项代码逻辑运行的次数，可以是微服务对外提供接口的被调用次数、某一个方法被执行的次数。它的示意图如下图 2 所示，其中 Y 轴表示在指定间隔内，被监控体的总体运行次数。

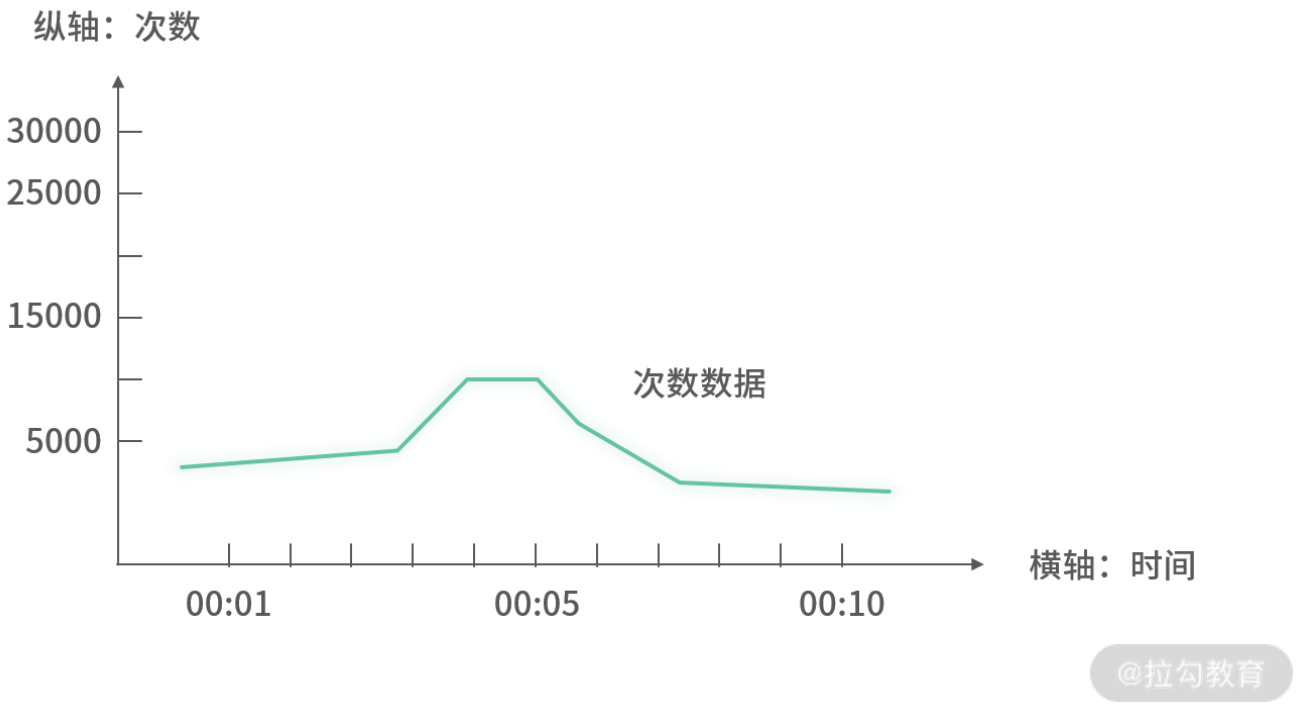

图 2：次数监控图

2. 性能监控

性能监控里的被监控体是性能，可以是微服务对外提供接口的性能、微服务依赖下游其他接口或存储的性能。

在性能监控里，有几个通用的运行状态数据：平均性能（AVG）、Max 以及TP999 。下面我们具体看一下它们的含义与区别。

（1）平均性能（AVG）指上述时间间隔里的代码总运行次数的耗时平均值。计算公式：间隔内所有调用的耗时累加/总次数。

（2）Max 性能则是直接显示上述时间间隔里的耗时最高的一次。假设在指定间隔内代码运行了三次，对应的性能分别为 10ms、300ms、50ms，那么最终显示的 Max 性能就是 300ms。

（3）TP999 性能表示对上述时间间隔里所有的耗时进行升序排序，处在第 99.9% 位置的性能耗时。打个比方，假如在指定间隔里共计发生了 1000 次代码执行，那么TP999 表示这 1000 次经过升序排序的请求的第 999 次的耗时。此指标反映了指定间隔里，99.9% 的请求的性能都低于 TP999 所代表的值。以此类推，还有 TP50、TP9999，分别表示满足 50% 和 99.99% 的请求性能所处的水平。

在实际工作中，上述三个性能指标常相互组合使用，而不是单看某一个指标。比如，某一个方法的 Max 性能很差，可能超过 1s，而 TP999 或 TP9999 却很好，控制在 50ms 以内。这说明它的大部分请求性能不错，只是每个时间间隔里，会有一次请求性能很差，此时你就需要注意某次请求的数据是否和其他请求存在差异，或者是否发生了其他故障。

此外，上述三个性能监控指标也通常展示在一幅图里，如下图 3 所示：

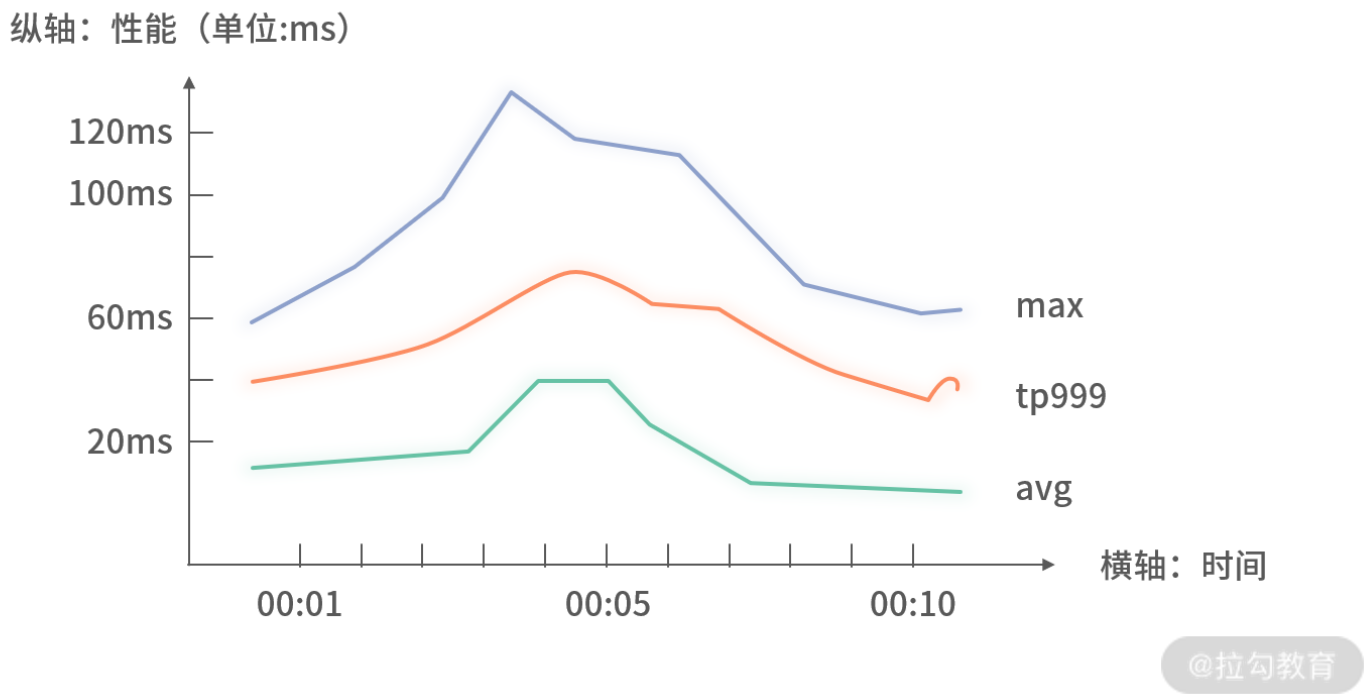

图 3：性能监控图

3. 可用率监控

可用率里的被监控体是在指定时间里，代码执行成功的占比。假设在指定时间间隔里，代码运行了 100 次，其中 99 次**经过判断**都为成功，那么在可用率监控图里 Y 轴显示的值即为 99%，具体格式可以参考下图 4 所示：

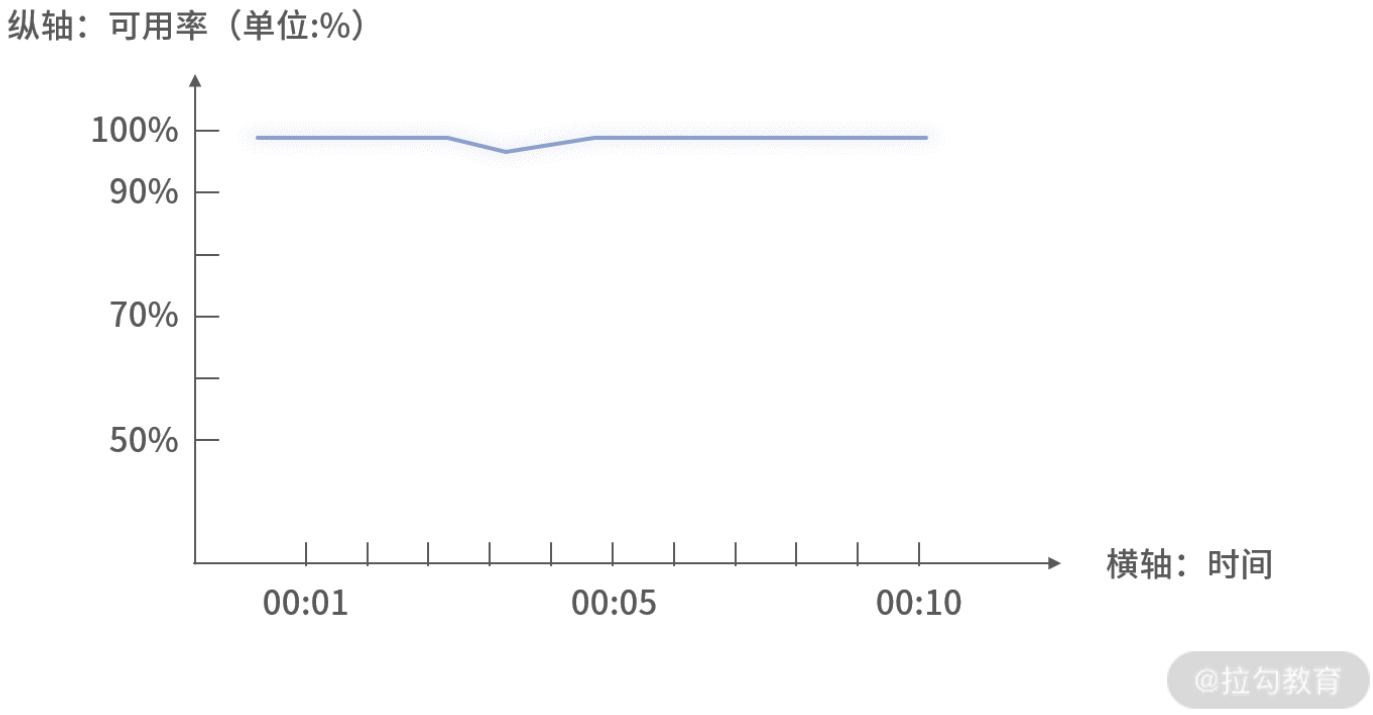

图 4：可用率监控图

讲述“确定某一方法在执行时是否成功”时，我使用了“经过判断”这一描述。这里使用相对抽象的描述，是因为在不同的场景下，判断某一个方法执行是否成功的准则不同。同样的执行结果，在某些场景里被认为是成功，在某些场景里则认为是失败。接下来，在可用率的小节将详细描述这些场景及对应的规则。

如何通过监控发现微服务的问题

这里我仍基于“第 17 讲”里介绍的微服务骨架图，如下图 5 所示，从微服务的入口、微服务自身及微服务的依赖这三个方面，讲解如何应用上述三种监控方式，以及对应的一些最佳实践准则，从而发现微服务里的各种潜在问题。

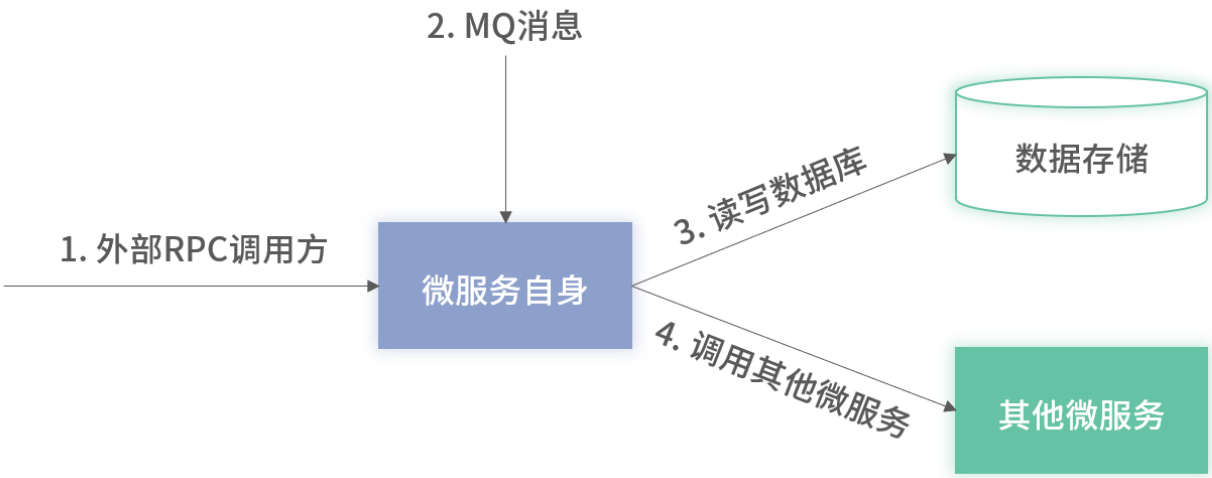

图 5：微服务骨架图

微服务入口

首先，**微服务的入口必须设置次数监控、可用率监控和性能监控**。因为微服务的入口，即对外供其他微服务调用的接口，是微服务自身能力对外提供的唯一通道，所以它的各项监控指标必须面面俱到。

微服务入口在调用次数监控上，有以下几点原则可以参考。

1. 需要设置调用次数报警，调用次数的报警阈值可以参考单机压测瓶颈值。

设置调用次数报警的目的是，通知你感知流量异常，并能够第一时间处理，比如进行扩容、排查异常流量等操作。当然，除了次数报警外，前置的设置限流不可少，因为有些时候是半夜收到报警，响应时间可能较长，通过限流可以规避因为处理报警不及时而导致的宕机问题。

2. 按调用方设置调用次数监控。

正常情况下，一个微服务的调用方不止一个。当微服务的调用量在某些时刻突然暴涨时，你需要定位到是哪个调用方导致的。通过设置按调用方的次数监控，便可以排查到具体是哪个调用方导致的流量增加。

3. 关于调用次数的阈值，需要设置调用次数的同环比监控。

在上述流量飙升的案例里，如果有调用次数的同环比监控，不通过排查，基于监控就可以自动知道具体是哪个调用方的流量异常，导致整体流量飙升。

微服务入口在性能监控上，有几点可以参考。

1. 并不是每一个性能指标都需要设置报警。

比如 TP99、TP999 或 TP9999，只需要设置其中之一即可。但平均性能、TP9999 （或 TP999）以及 Max 这三个指标均需要设置报警阈值。值得注意的是，报警阈值需要根据接口的 SLA 来设置，而不是“拍脑袋决策”。

2. 和上述调用次数的第二点类似，需要配置按调用方的性能监控，用来观察是否存在不同的调用方，因为使用方式的差异，进而导致性能上的差异。

3. 基于入参数据进行监控。

假设一个接口支持调用方指定批量大小来查询用户信息。那么此时，可以按调用方传入的批量大小进行性能监控。比如将批量大小处在 0~10 之间设置一个监控点、处在 10~20 之间的设置另一个监控点，以此类推。

在添加上述的监控埋点后，你可能会发现：批量大小处于 50 以内的性能都差不多，而批量大小处于 50 以后的性能会有较明显的增加。此时，通过监控，你就可以把接口的最大批量大小设置为 50。如果调用方有一次查询需要超过 50 个数量的需求，他可以通过并发的方式查询，将单次查询需要超过 50 个的数量一分为二即可，此时的性能将比一次查询超过 50 个的性能更好一些。

微服务入口在可用率监控上，可参考的原则如下。

1. 需要设置接口的可用率告警和按调用方设置报警（与“性能监控”第一条类似）。

2. 我们讲解可用率时曾提到，判断是否需要降低可用率的方法由当次请求是业务异常还是非业务异常决定。

其中，业务异常不能判断为方法执行失败，还是成功，即业务异常不能降低可用率。而非业务异常（如网络故障、连接失败、机器宕机等导致的异常），需要降低可用率。

业务异常指的是用户没有按规定的要求输入数据，比如用户输入的参数如手机号码、邮箱地址不合法等场景。对于这类操作，不需要降低可用率，只需要提示客户按指定格式重新输入即可。

而非业务异常指的是网络故障、连接失败、机器宕机、代码执行出现空指针、调用下游超时等现象，是需要降低可用率的。出现上述异常情况，可能是因为你的代码未按预期执行、网络环境未按预计运行，此时，通过降低可用率进而报警，可以让研发人员排查导致问题的原因。

3. 可用率的阈值需要按接口的等级差异化设置。

在线上环境里，网络并不是绝对稳定的，可能会产生偶发的抖动，进而导致接口调用出现几秒或几分钟的部分失败，产生可用率下降的现象。 为了规避网络抖动导致的可用率报警的情况，你可能会将可用率报警的阈值设置为低于 95% 才报警。虽然此种情况可以屏蔽误报警，但也有可能屏蔽掉真正的报警。因此你需要根据接口的等级设置报警，如果是提单接口，你就需要设置可用率低于 100% 时报警。虽然偶尔会收到一些误报警，但相对错过提单真正的报警而言，这样做还是值得的。

微服务自身

微服务自身执行的各个方法，可以根据需求选择使用上述三个监控指标：性能、可用率和次数监控并配置对应的预期阈值。微服务内的方法是否使用上述监控有几点准则。

1. 并不是监控点越多越好。虽然记录监控数据对机器的性能损耗很小，但一个方法设置几十、上百个监控点仍会有一定影响。
2. 太多的监控点会导致系统维护人员产生麻木。首先监控点太多，不知道从哪里看起。其次，当出现网络抖动、机器故障等异常时，所有的监控点都在告警，研发同学在排查时，无从下手。
3. 建议对核心方法、怀疑性能较差的方法增加监控，这样可以快速发现和排查到核心方法和性能差方法存在的问题。其他方法的监控数据，通过微服务入口的监控即可查看到。

除了微服务自身的各类方法需要监控，微服务所属的进程，以及它部署的机器也有很多被监控体可以监控。下面我们详细介绍。

1. 微服务使用的 RPC 框架的剩余线程池数量。

当微服务框架的线程池变少或为零后，调用方新的请求都会被拒绝。因此，当监控到剩余的线程池快耗尽，就需要快速处理，如调整线程池的大小、扩容新的机器等。

2. 如果是基于 JVM 的各类语言应用，对于 JVM 相关数据也需要监控。比如 Young GC、Full GC 的频率、每次 GC 的时间，以及堆内存的使用量。
3. 微服务的进程存活监控。

每一个微服务的进程都有一个进程号，此外对外提供接口服务的进程还会有端口号。可以使用 ping 或者 ps 命令，每一个时间间隔检查一次，如果监听到进程的进程号或端口不存在，便进行报警。

4. 监控微服务所在机器的内存的使用率。

类似于可用率，使用率也是一个比例值。它表示机器已使用内存占机器总内存的比值。如果机器的内存使用率很高，操作系统可能会主动将占用较高的进程关闭。因此需要监控此值，当使用率飙升，去排查具体对应的原因。

5. 机器的 CPU 的使用率和内存使用率类似。

太高的 CPU 使用率会导致微服务卡顿或者微服务不可用。因此，需要主动监控并配置告警，以便提前去处理。

6. 机器负载（CPU Load）。

这一监控容易被遗漏，它表示当前机器里有多少进程处在“正在执行”和“等待执行”这两个状态里。假设机器的 CPU 只有 4 核，而机器负载在多个监控间隔里都远超于 4，比如在 10 以上，那么说明当前机器负载过高，这些进程排队等待执行的时间较长，性能可能较差。此时，可以适当减少机器上部署的进程数。

微服务依赖

在介绍微服务的依赖监控前，我们再回顾下微服务依赖的架构图，如下图 6 所示：

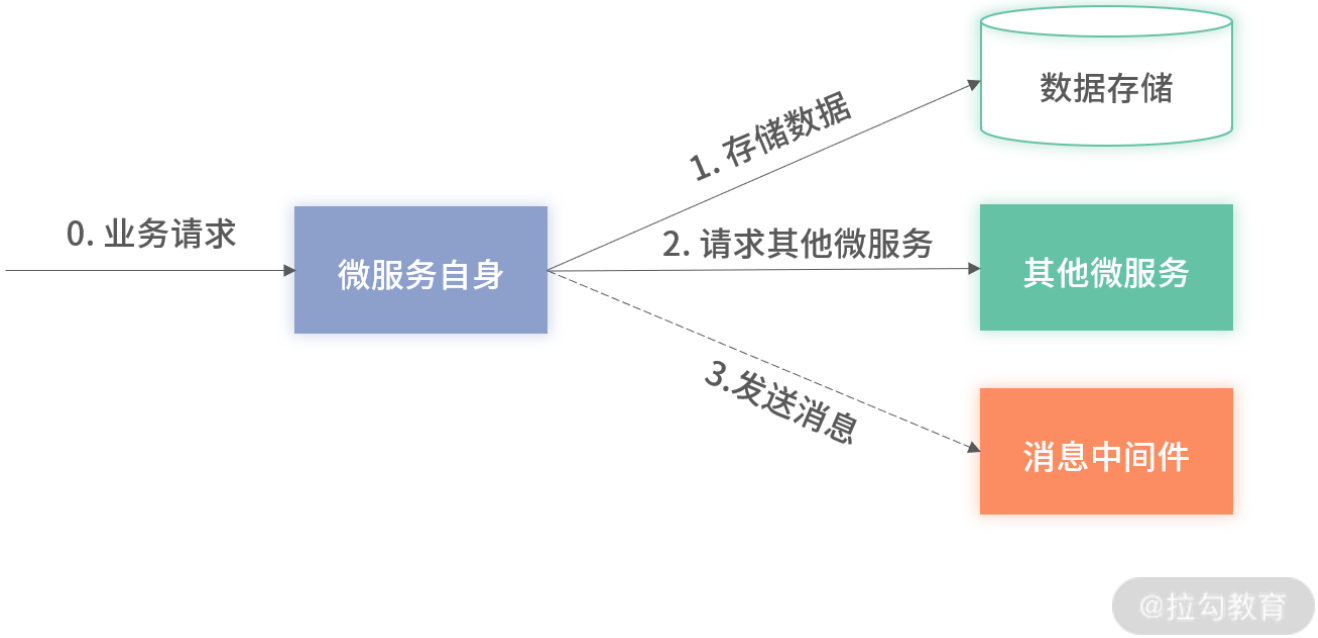

图 6：微服务依赖的架构图

对微服务每一个依赖的调用，类似上述“微服务自身的监控”小节里提到的对内部方法增加监控——并不是微服务内部的所有方法都要加监控，而是要挑选重点和可能存在问题的方法。

而对于微服务的所有依赖都需要统一增加监控，你可以从“第 17 讲”提到的原则“怀疑下游”找到原因——因为依赖的下游随时可能出现问题，为了快速定位问题，所以所有的外部依赖都需要增加监控。

如果是 Java 应用，监控的方式可以采用统一的 AOP 切面来实现。此外，也可以借助一些框架的功能统一拦截并进行监控，比如 MyBatis 里就提供用 Interceptor 拦截所有 SQL 的执行，在此处就可以添加统一的监控。

除了依赖要增加监控，判断依赖的其他微服务的接口在执行上是否成功，也需要格外注意。有些依赖的微服务执行异常时，并不会抛出异常，而是返回一个经过包装的结果对象，比如 RPCResult，并将错误信息包装在其中。此时，如果你在对外部依赖的可用率监控中没有判断 RPCResult 中的值，有可能遗漏应该告警的结果，导致问题没有被发现，进而影响线上的业务。

监控时间间隔

在上述的讲解里，一直使用了时间间隔来表示监控图的横轴，但没有指出这个时间间隔是多少，是分钟级还是秒级的?

理论上这个间隔是越小越好，最好是秒级。但很多监控系统都不提供秒级的间隔，原因是时间间隔越小，需要存储的数据量就越多。

以可用率监控为例，间隔为 1min 时，表示 1min 里所有成功的次数和总的调用次数比例。而间隔为 1s 时，表示 1s 里所有的成功与总的调用次数的比例。秒级产生的可用率数据量是分钟级的 60 倍，所以因为存储容量的限制，很多监控系统只提供分钟级别的监控。

但如果监控系统既提供分钟级又提供秒级监控，那么优先选择秒级。因为秒级监控发现问题的速度更快。以可用率监控为例，秒级监控在 1s 间隔达到后，即可算出可用率，而分钟监控要在 1min 间隔到达后，才可算出可用率，两者相差了 59s。

总结

在本讲里，介绍了几种常见的被监控指标，可用率、调用次数、性能监控，这三个监控指标是每一个微服务应用都必须配置的。其次，从微服务入口、微服务自身、微服务依赖这三个角度梳理了如何落地上述三个监控，以及一些附加的监控，如机器监控、内存监控等。

最后，留给你一个任务，根据本讲介绍的内容和原则，检查你所负责的微服务是否存在监控遗漏，可以进行一次查漏补缺。如果遇到什么问题，也可以写在留言区，我们一起讨论。

这一讲就到这里，感谢你学习本次课程，接下来我们将学习21 | 如何进行高保真压测和服务扩容?