

15 | 高可用架构案例（三）：如何打造一体化的监控系统？

2020-03-25 王庆友

架构实战案例解析

[进入课程 >](#)



讲述：王庆友

时长 16:24 大小 11.27M



你好，我是王庆友。

上一讲，我与你介绍了整体化监控系统的设计方案，今天我就带你深入它的内部设计，让你可以了解它具体是如何落地的。

这个监控系统主要分为 4 大部分：节点信息采集、节点接入、数据上报和前端展示。下面，我就来为你具体展开介绍。



节点信息采集

在上一讲中，我提到过，Agent 负责采集节点的健康数据，每隔 3s，主动访问一次；然后，Agent 会根据这些数据，结合相应的规则，来判断节点的健康状态。最终的健康状态有三种，分别是错误、警告和正常，这三种状态也对应了 Dashboard 中节点的红黄绿三种颜色。

节点分为 4 类：Web 应用、Redis、MQ 和数据库。下面我就来具体讲一下，系统是如何对它们进行监控的。

对于 Redis 节点，Agent 通过 Jredis API，尝试连接 Redis 实例并进行简单的读写。如果这个操作没有问题，就表明 Redis 当前的健康状态是正常的，否则就是错误的。


对于 MQ 节点，Agent 是通过 MQ API，来检测 MQ 节点是否有活跃的消费者在连接，同时检测队列积压的消息数量。如果没有活跃的消费者，或者未消费的消息超过了预设的值，就表明当前的 MQ 节点的健康状态是错误的，否则它就是正常的。

对于数据库节点，Agent 是通过 JDBC 去连接数据库，并对表进行简单的读写。如果操作成功，表明数据库的健康状态是正常的，否则就是错误的。

对于这三类节点，它们的健康状态只有正常和错误两种，没有警告状态。如果节点有问题，Agent 会同时给出具体的出错信息，比如节点连接错误、积压消息过多等等。

对于 Web 应用来说，Agent 采集的方式则稍微复杂一些，它会同时采集应用的功能和性能数据，具体包括最近 3s 的接口调用次数、接口平均响应时间、接口出错次数、节点的健康状态和错误消息。

这里我给你举一个 Web 节点请求和响应的例子，来帮助你直观地了解 Agent 是如何采集数据的。

 复制代码

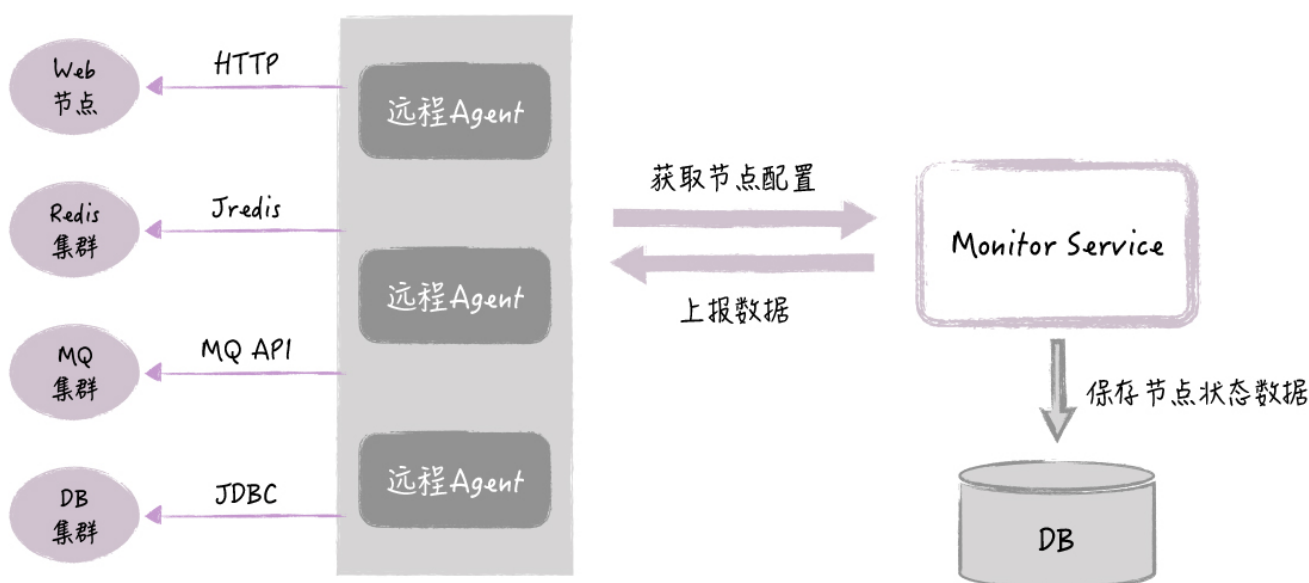
```
1 请求: http://10.10.1.1/agent/check
2 返回信息:
3   "status": "warning",
4   "avg_time": "583.0",
5   "call_count": "10",
6   "error_count": "0",
7   "error_info": " orderListGet: current average time= 583.0, total average time :
```

Web 节点会预先提供一个 HTTP 接口，Agent 通过调用这个接口，返回当前 Web 实例最近 3s 的健康状态。

这里最主要的就是 **status 字段**，它表明了 Web 节点最近 3s 是否健康，如果是 “error” 或者 “warning”，返回的结果还会包含 **error_info 字段**，它负责给出具体的错误信息。

Agent 在获取了这 4 类节点的健康状态后，会调用 Monitor Service 进行数据上报，如果节点有问题，上报内容还包括具体的错误消息。

总体的架构如下图所示：



你要注意的是，Agent 本身是一个独立的应用，它不需要和节点部署在一起，如果节点数量少，我们部署一个 Agent 实例就可以；如果节点的数量比较多，我们可以部署多个 Agent 实例，比如给每类节点部署一个实例。总的要求就是，让 Agent 能够在 3s 内，完成所有节点的健康信息收集就可以了。

另外，节点的连接信息，事先是配置在数据库里的，比如数据库节点的 IP 端口、账号和密码等等，当 Agent 启动的时候，它会通过 Monitor Service 获取节点配置信息，Agent 在运行过程中也会定期刷新这个配置。

接入监控系统

好，说完了节点信息的采集，下面我们来看下，这些节点要接入监控系统，都需要做些什么。

对于 Redis、MQ、DB 这三类节点，接入监控系统只需要提供配置信息就可以了，无需额外的开发。


而对于 Web 应用接入监控，我们需要对应用代码做些改造：

1. 针对每次接口调用，应用程序需要在接口代码中记录本次调用的耗时以及出错状况；
2. 应用程序需要汇总最近 3 秒的接口调用情况，根据规则，给出节点的健康状态；
3. 应用程序提供一个对外的 HTTP 接口，供 Agent 来获取上一步给出的健康状态。

为了方便 Web 应用的接入，监控系统开发团队提供了 SDK，它内置了接口调用信息的统计和健康计算规则。应用程序借助 SDK，就可以给 Agent 提供最终的健康结果，也就是说 SDK 帮助应用完成了最复杂的第二步工作。


所以，对应用来说，它接入监控系统是非常简单的。

首先，在每个应用接口中，调用 SDK 提供的 **logHealthInfo** 方法，这个方法的输入包括了接口名字、本次接口调用耗时和错误信息，这和我们平常接入日志系统是很类似的。

 复制代码

```
1 try{
2     result = service.invoke(request)
3     HealthUtil.logHealthInfo("xxx_method",
4         (System.currentTimeMillis() - start), null);
5 }catch (Exception e){
6     HealthUtil.logHealthInfo("xxx_method",
7         (System.currentTimeMillis() - start),
8         e.getMessage());}
```

然后，应用提供一个额外的 HTTP 接口，在接口中直接调用 SDK 内置的 **healthCheck** 方法，给 Agent 提供最终的健康信息。这些就是应用接入监控系统要做的全部事情。

 复制代码

```
1 @RequestMapping(value = "/agent/check")
```

```
2 public String reportData(){
3     return HealthUtil.healthCheck();
4 }
```

我们可以看到，**SDK 通过在接口方法中进行埋点，可以收集每次接口的调用情况，那它最终是怎么计算出当前节点的健康状况呢？**

SDK 的内部，实际上是一个 HashMap 结构，它的 key 就是 Web 应用的各个接口名字，它的 value 是一个简单的对象，包含这个接口最近 3s 总的调用数量、总的出错次数和总的耗时等。当每次 Web 应用有接口调用时，我们在 HashMap 内部根据接口名字，找到对应的 value，然后增加这三个数值，就完成了接口调用数据的收集。

当 Agent 调用 HTTP 接口，拉取节点健康数据时，SDK 会计算节点的健康状况，具体规则如下：

如果最近 3s，接口调用没有发生错误，节点的健康结果就是正常；如果出错次数在 1 到 5 之间，健康结果就是警告；如果大于 5，健康结果就是错误。

如果最近 3s，接口响应时间超过正常值的 10 倍，健康结果就是错误；如果在 5 倍到 10 倍之间，健康结果就是警告，否则结果就是正常。

这里有个问题，**接口调用响应时间的正常值是怎么来的呢？**这个值不是预先设置的，我们知道，如果预先设置的话，这个数字很难确定。这里的正常值其实是 SDK 自动计算的，SDK 会记录应用从启动开始到目前为止，接口的总耗时和总调用次数，然后得出平均的响应时间，作为接口调用的正常耗时（总调用次数和总耗时也记录在 HashMap 的 value 里）。

你可以看到，Web 应用的健康状态判断是结合了应用的功能和性能的，两者是“或”的逻辑关系，只要某一项有问题，健康结果就是有问题。比如说，最近 3s 接口功能没出错，但耗时是正常的 10 倍以上，SDK 就会认为节点的健康状态是错误的。

值得注意的是，SDK 会针对每个接口进行分别计算，最后取最差接口的结果。比如说，应用有 10 个接口，如果其中 8 个接口是正常状态，1 个接口是警告状态，1 个接口是错误状态，那么该应用的健康结果就是错误状态。

还有一点，SDK 在 HashMap 内部，不会记录每个接口调用的详细日志，而是只维护几个简单的总数值，因此 SDK 对应用的内存和 CPU 影响，都可以忽略不计。

前端信息展示

现在，监控数据已经通过 Agent 和 Monitor Service 保存到数据库了，前端的 Dashboard 通过调用 Monitor Service 接口，就可以获取所有节点的最新健康状态（Dashboard 也是每 3s 刷新一次页面）。接下来我们就要考虑，**如何在 Dashboard 里展示节点健康状态，这影响到我们能否直观地定位系统的问题。**

首先，一个应用一般有多个实例，比如 Web 应用很可能部署了多个实例；

然后，应用之间有上下游依赖关系，比如 Web 应用依赖 Redis 和数据库。

我们在页面中，就需要把所有这些信息直观地体现出来，这对我们判断问题的源头很有帮助。

这里的页面显示有两种实现方式。

一种是**页面定制**的方式，我们把应用有哪些节点，以及应用的上下游依赖关系，在前端代码里固定死。但问题是，如果系统的部署有变动，页面就要重新调整。在我们的监控实践中，我们要监控很多套系统，这样我们就需要为每个系统定制页面，初始的工作量就很大，更加不用说后续的调整了。

所以，在实践中，我们采取了一种更加灵活的**前端展现**方式，能够通过一套前端代码，灵活地展示系统的节点以及依赖关系，效果上也非常直观。

它的具体实现方式是：我们把页面的展示内容分为三个层次，分组、应用和节点。一个页面代表一个系统，它包含多个分组，一个分组包含多个应用，一个应用包含多个节点（节点代表了一个具体的实例，有独立 IP）。

这里的分组实际上是对应用进行归类，比如说，共享服务是一个分组，它内部包含多个服务，这些服务是并列的关系。这样，我们通过分组在页面里的位置关系，来体现应用之间的上下游依赖关系。

如下图所示，红色圈里的是各个分组，蓝色圈里是各个应用。我们可以很清晰地看到，“应用层”分组里的会员应用，会调用“依赖服务”分组里的四个服务。



这里，你可以发现，“应用层”分组里只有 1 个应用，它采取了 1 行 1 列的布局，而“依赖服务”分组里有四个服务，它采用的是 2 行 2 列的布局。**那么这个布局是怎么实现的呢？**

首先，布局是在后台定义的，保存在数据库里。我们为每个系统预先设定好布局，类似 HTML 里的 Table 布局语法，行用 TR 表示，列用 TD 表示。我们根据页面显示要求，提前确定好分组和应用会占用多少行，多少列。前端通过 Monitor Service 的接口获取页面的布局信息，然后根据布局信息进行动态展示，如果系统的部署有变化，我们在管理后台调整布局就可以了，非常灵活。

这样，我们通过类似 Table 方式的布局，前端通过一套代码，就可以满足所有系统的节点展示需求，并且能够比较好地体现应用之间的上下游依赖关系，当系统有问题时，我们就可以很直观地判断出问题的根源在哪里。

在前面，我说的是一个页面代表一个系统，其实我们也可以对所有系统的节点做一个整体的**大盘监控**，这样我们只需要看一个大盘页面，就可以监控所有的节点，如下图所示：



大盘监控具体的实现方式是这样的：

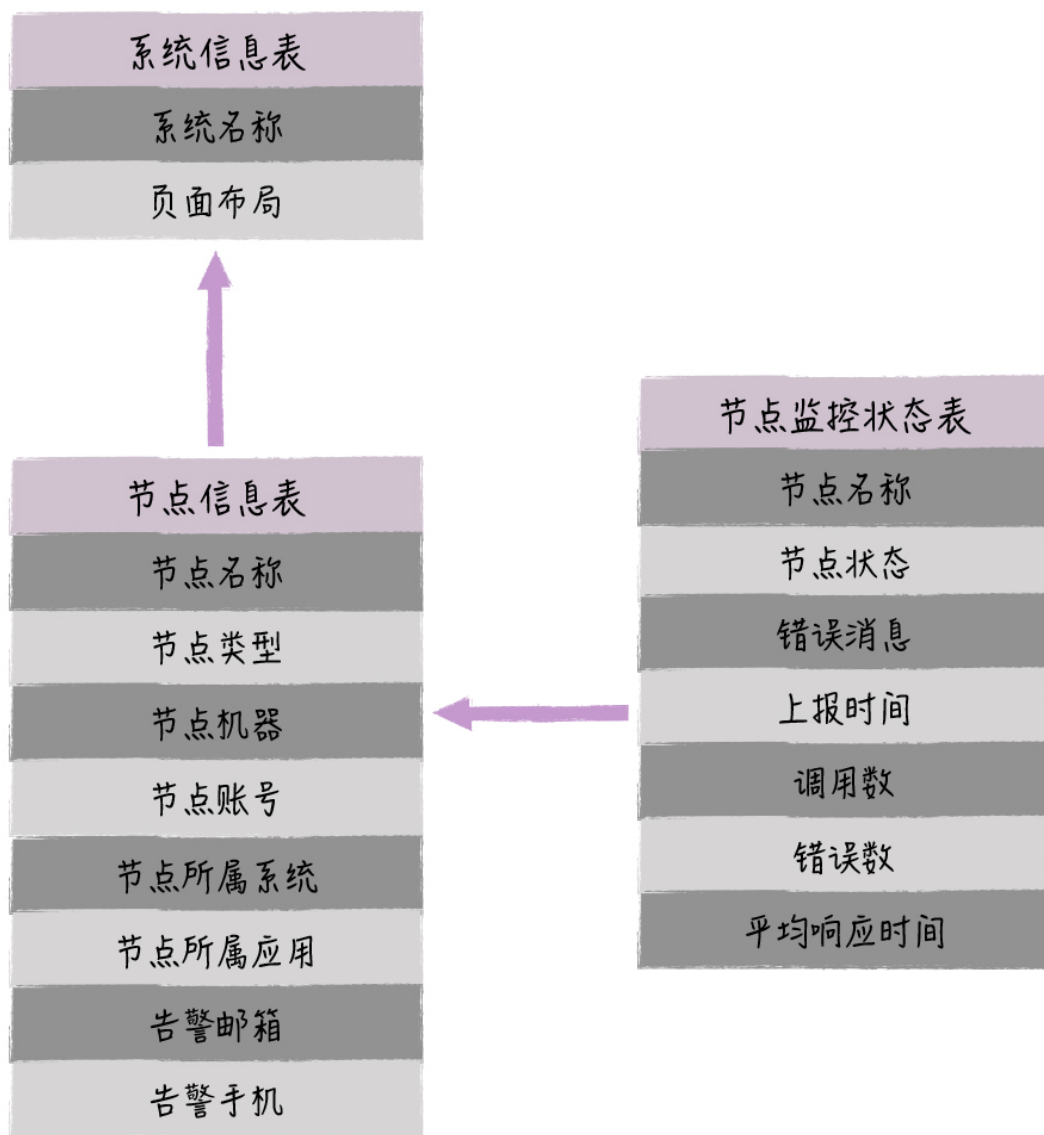
- 首先，前端页面读取所有节点的健康状态，按照节点分类展示有问题的节点，并标识出相应的颜色；
- 然后，节点的具体出错信息也可以在大盘中展示；
- 最后，我们根据每个系统内部节点的健康状况，按照一定的规则，算出各个系统的总体健康状态，在页面展示系统的健康状态。

比如说一个系统，如果它下面有一个节点是错误状态，对应的系统状态就是红色的；超过两个节点是警告状态，对应系统状态就是黄色的。如果我们点击相应的系统节点，就会跳转到具体系统的监控页面中，我们可以进一步了解该系统内部各个节点的详细状态信息。

通过这个大盘监控，我们就能在一个页面里，知道当前哪些节点有问题、哪些系统有问题、具体出错信息是什么，我们平常监控这一个页面就可以了。

库表设计

最后，我简单介绍下监控系统的数据库表设计，主要的表有 3 张：



1. **系统信息表**，用来定义监控体系里有哪些系统，其中 Layout（布局）定义了该系统前端的布局方式。
2. **节点信息表**，用来定义节点的配置信息，其中节点类型可选的值有 Web 应用、Redis、MQ、DB 等等，节点类型决定了节点健康信息的获取方式。其他字段用于 Agent 如何去连接节点，还有邮箱和手机用于节点出错时，相应的人可以接收报警信息。
3. **节点监控状态表**，用来记录每个节点的最新健康状态，用于 Dashboard 显示。

到这里为止，我给你介绍完了整个系统的核心设计。从监控的层次来看，这个监控系统可以分为大盘级别监控 -> 系统级别监控 -> 节点级别监控，你甚至还可以快速关联到每个节点的专门监控系统，比如 Zabbix 的硬件监控、CAT 的应用监控、ELK 的日志监控等等，实现最粗粒度到最细粒度监控的一体化。

相比较各个专门的监控系统，我们这里不求对各类节点的监控做得多深入，而是大致上能反映节点的健康状况即可（如果我们要对组件做更深入的监控，组件的 API 也可以为我们提供非常详细的信息）。我们更强调的是要把系统的所有节点串起来，直观地反映它们的健康状况，避免监控系统的碎片化和专业化。

总而言之，这个监控系统就相当于是一个全身体检，不同于对某个器官的深入检查，它是把系统的各个部位都做了初步检查，并且给出了一个很容易阅读的结果报告。这个系统实现起来很简单，但非常实用，我们相当于用 20% 的成本，实现了 80% 的监控效果。

总结

今天，我与你分享了一体化监控系统具体的设计细节，相信你已经非常清楚了它的内部实现机制，如果有需要，你也可以在实践中尝试落地类似的监控系统。

这里，我讲得比较细，不仅仅是为了让你理解这个监控系统是怎么设计的，而是想和你分享做架构设计时，我们要做全面深入的考虑，要简化开发的对接工作，要简化用户的使用，这样的架构设计才能顺利落地，实现预期的价值。

比如在这里，我们为 Web 应用提供了 SDK，这降低了开发者的接入成本；我们通过页面的动态布局设计，避免了前端开发工作的定制化；我们通过大盘监控以及和现有监控系统进行打通，进一步方便了用户的使用，全面提升监控系统的价值。

最后，给你留一道思考题：你觉得在做架构设计时，最大的挑战是什么？

欢迎你在留言区与大家分享你的答案，如果你在学习和实践的过程中，有什么问题或者思考，也欢迎给我留言，我们一起讨论。感谢阅读，我们下期再见。

点击参与 

20年架构老兵邀你一起 打卡，带你进阶资深架构师



扫一扫参与小程序话题



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 14 | 高可用架构案例（二）：如何第一时间知道系统哪里有问题？

下一篇 16 | 高性能和可伸缩架构：业务增长，能不能加台机器就搞定？

精选留言 (12)

 写留言



正在减肥的胖籽。

2020-03-25

老师您好。有参考的代码吗？我主要的困境是代码如何落地。



👍 2



Geek_kevin

2020-03-30

请教老师,那个页面的动态布局设计,避免了前端的定制化,这里有没有再具体一点的信息可供参考的?

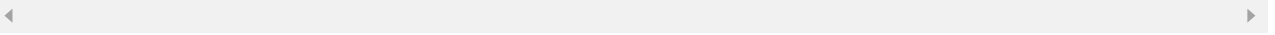
作者回复: 这个说起来有点话长，你可以参考下HTML的table语法。在table里的每个cell单元，它的rowspan表明跨多少行，colspan表明跨多少列。

每个cell里放一个应用，包括它的各个app节点，redis节点，db节点等。

如果两个应用是并列关系，我们把它放同一水平位置（或者说同一行），如果是上下游调用关系，把应用上下放置。

这里，布局在后台指定，前端负责统一解释布局，并按照cell的定义，把应用放在table里的相应位置。

如果你想落类似的监控系统，前端一开始可以定制化。



💬 1

👍 1



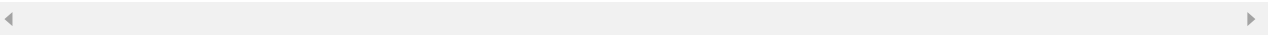
夜空中最亮的星（华仔...

2020-03-25

这个是自己开发的？有开源软件的参考吗？

展开 ▾

作者回复: 自己开发



💬

👍 1



Minasix

2020-03-29

以前总是很局限的看项目，业务功能为主，看了老师的课程，发现架构整体层面和监控如此重要，学到了，感谢

展开 ▾

💬

👍

孙同学

孙同学

2020-03-28

<https://www.processon.com/view/link/5e51378ce4b0c037b5f9d1e3> 整理学习更新
感觉架构设计要对整体把握性很强，设计缩放自如，发现自己道行很浅，需要加油补充各方面知识呀

💬

👍



蓝天

2020-03-26

1，最大的挑战是过度设计，总觉得流量要大到不行，其实并没有
2，我们还在使用rmi调用，虽然性能够用，但扩展性差，个人感觉不适合微服务治理等，但在公司使用较长，有些根深蒂固的，架构师们也不推，不知道是否我的看法不对，老师有啥好的建议呢？

展开 ▾

作者回复: rmi确实有点老, netty,dubbo,spring cloud都可以试试, 找1-2个非核心系统试试, 感觉早晚要升级



starj

2020-03-25

感觉这个系统很复杂啊, 对于我们人员缺少的小公司很难花时间去实现



Better me

2020-03-25

老师这里对应用的监控是通过agent每3s请求接口得到相关信息, 然后取最差接口, 这里是否会影响到接口的性能, 应该如何平衡

展开 ∨

作者回复: 性能不会有问题, 3秒调用一次, 每次耗时不到1ms



Alex

2020-03-25

监控系统有两条线一条是预警一条是排障。预警就是针对关键指标设阈值主动提醒。排障以业务异常为主线, 实现统一日志, 纵通过以traceid 看调用链信息, 横向看错误时间点附近资源是否异常。我原来按这个思路做的监控。

展开 ∨



Jxin

2020-03-25

感觉这个监控应该要包含全链路监控的职能。这样子全链路监控除了定时采样外, 还能记录 上述监控异常场景时的调用链路信息。便于定位问题。

展开 ∨



tt

2020-03-25

我觉得难点就在于这一句话 “用 20% 的成本, 实现了 80% 的效果” 。

最近设计的一个系统，目标就是满足业务用最小的代价能最快速地上线新的业务，这样，即使错了，也没有什么损失。

...

展开 ▾



老姜

2020-03-25

利用现有的监控的API可以实现吗？不是开源的开发成本不小，还有集成的成本。

