# 18 | 如何设计微服务才能防止宕机?

在上一讲里，介绍了构建一个稳健的微服务的具体法则：**防备上游、做好自己、怀疑下游，** 并介绍了为什么要防备上游，以及一些防备上游的具体手段。

在本讲里，咱们一起来学习，做好微服务自身的设计和代码编写的常见手段。

## 微服务 CPU 被打满如何排查

在讲解具体有哪些手段可以用来构建一个更加稳固的微服务前，咱们先来看看如何高效、精准地定位问题。下面我将以设计不精良的微服务在线上最容易产生的问题：“**机器 CPU 被打满**”为例进行讲解。

这个问题也是面试中的高频话题，很多面试者能够回答出其中一二，但距离让面试官满意的答案还有一定距离，下面咱们就一起来看看详细的排查步骤。

一台机器上会部署一至多个进程，它们可能是一个或多个业务应用进程和多个其他工具类进程（比如日志收集进程、监控数据收集进程等）。大概率导致机器 CPU 飙升的是业务应用进程，但仍需准确定位才可得出结论。

1. 在 Linux 系统里，可以使用**top 命令进行定位，** top 命令可以按进程维度输出各个进程占用的 CPU 的资源并支持排序，同时还会显示对应的进程名称、进程 ID 等信息。
2. 根据排序，便可以确定占用 CPU 资源最高的进程，但此时仍然不知道是哪段代码导致的 CPU 飙升。所以可以在此进程基础之上，做进一步的定位。top 命令支持查看指定进程里的各线程的 CPU 占用量，命令格式为：**top -Hp 进程号**。通过此方式便可以获得指定进程中最消耗 CPU 资源的线程编号、线程名称等信息。
3. 假设导致 CPU 飙升的应用是基于 Java 开发的，此时，便可以通过 Java 里自带的 jstack 导出该进程的详细**线程栈**（包含每一个线程名、编号、当前代码运行位置等）信息，具体见下方示例代码：

```
"thread name" prio=0 tid=0x0 nid=0x0 runnable
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
at java.net.SocketInputStream.read(SocketInputStream.java:171)
```

通过第三步定位的**线程号**和此步骤生成的**线程栈**，便可以精准确定是哪行代码写的有 Bug，进而导致进程的 CPU 飙升。上述分析是以 **Java 应用**进行举例，关于其他语言的应用如何导出进程堆栈信息，你可以到对应官网查看。如果有疑问也可以写在留言区，我们一起交流。

## 如何预防故障

上述介绍了，当微服务的代码编写不优雅，导致 CPU 飙升时，如何快速、准确应对的方法。下面将从**微服务的部署和代码编写**两个层面介绍一些准则，以便构建一个更加稳固的微服务，前置减少出现故障的概率。

### 部署层面

**首先，微服务及存储需要双机房部署**。双机房部署能够进一步提升服务的容灾能力。双机房部署的架构如下图 1 所示：

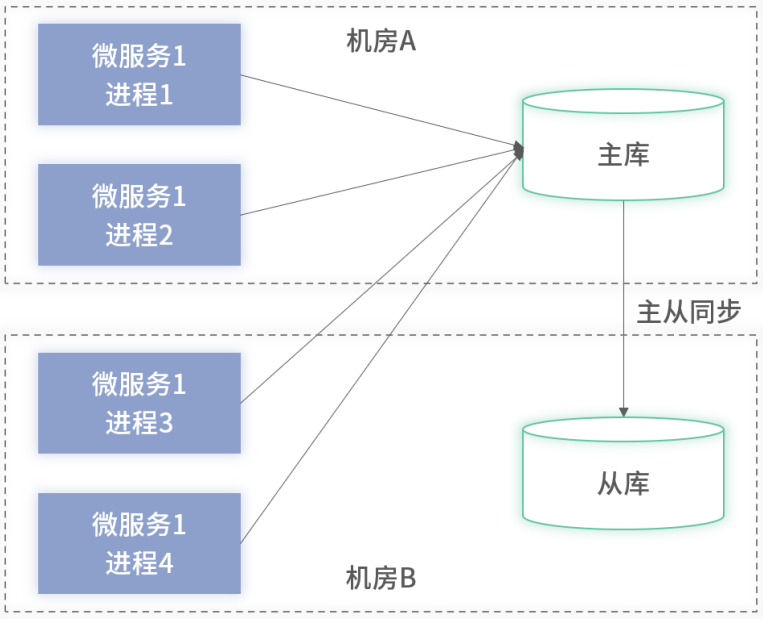




图 1：双机房部署架构图

上述部署里，同一个微服务分别在两个机房各部署了两台机器。在存储上，数据库的主从分别部署在两个机房里。当出现机房级别的故障，如网络不通时，可以直接将故障机房的机器从微服务的注册中心摘除。其次，如果故障发生在主库所在机房，就需要 DBA 进行协助，对主从数据库的数据对比、订正并进行数据库的主从切换。

双机房部署使得微服务具备了机房级别的容灾能力，当机房出现故障时，可以快速地进行切换，而不用耗费几个小时甚至更久的时间，在一个新的机房进行微服务和数据库的重新部署。但上述的部署里，数据库其实是单机房部署的。因为在实际运行时，只有主库承载读写流量，从库只是跨机房进行数据复制，作为灾备使用。

当真正出现机房故障时，整个微服务仍需停服一定时间，用来等待 DBA 进行主从切换，原则上只在秒级或者分钟级别。这在绝大部分场景里均可满足业务的需要，但有些用户使用高频的场景，如打车、即时通信等软件，需要业务尽可能 7*24 运行，减少或保障不出现业务停服的场景。对于此类需求，可以采用**存储按机房多地部署、且每个机房的存储均支持部分用户的数据读写的方案**进行升级，此方式在业界有个特有名词，叫作**单元化部署的架构**，具体架构如下图 2 所示：

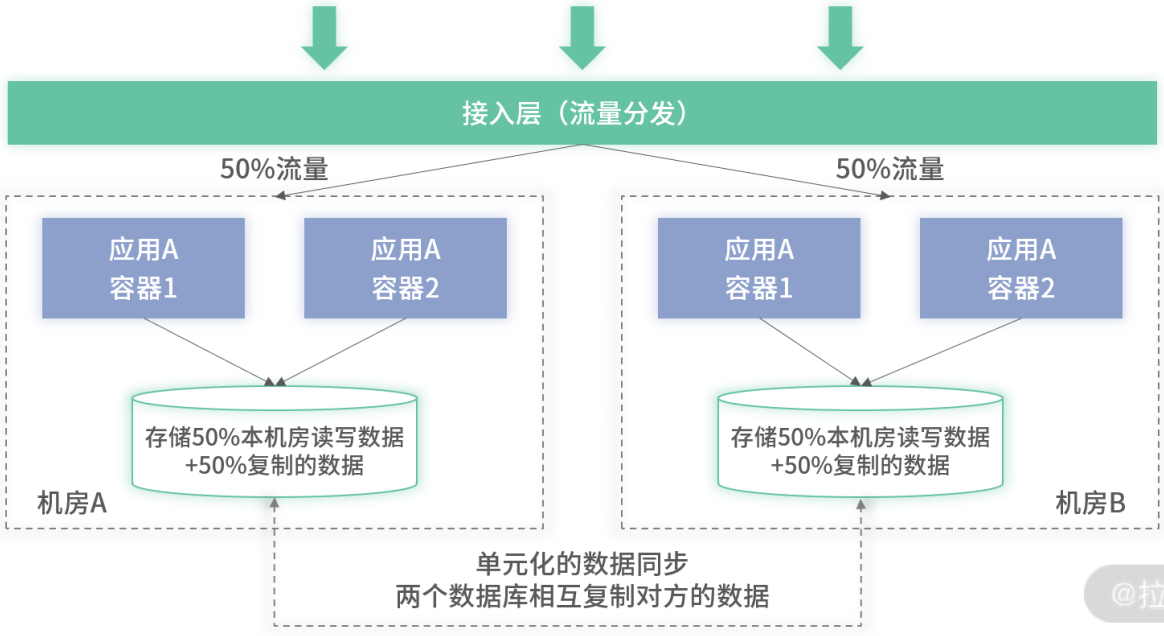




图 2：单元化架构图

在单元化架构里，两个机房里的数据库均为主库，它们都承载读写流量。此外，对于用户的请求流量，在网关层进行了转发，一部分转发至机房 A，另外一部分转发至机房 B。假设当机房 A 出现故障时，机房 B 所承载的流量是完全不受影响的，即路由至机房 B 的用户对于故障无感知。

而对于机房 A 里的用户，则可以在网关层进行前置再路由，将所有的请求全部转发至未故障的机房 B。在上图 2 中，有一条两个机房里的数据库主库互相同步的标识线，它是单元化里需要构建的数据同步模块。作用是发生故障时，减少机房 A 里的用户切换到机房 B 的时间。因为机房 A 里的用户可以切换到机房 B 的前置条件是，机房 A 里的数据已经全部同步至机房 B 里，实时的数据同步可以减少故障后 DBA 进行数据同步、对比和校准的时间。

可以看到，单元化架构并不是机房故障后，对于业务完全无损，而是保障一部分用户完全无损来提高高可用能力。

**其次，机房内至少部署两台及以上机器。** 这里再多啰唆一句，上述第一条要求至少双机房部署，并不是两个机房各部署一台机器即可，而是要在同一个机房里至少部署两台机器，保障机房内机器互相灾备。此方式可以防止当某一台机器故障后，出现整个机房全部失联，进而将调用方的所有的流量都打至另外一个机房，引起请求的性能和稳定性下降，因为跨机房的请求的网络传输时间更长。单机房部署单容器故障时导致的跨机房调用的架构如下图 3 所示，可以看到故障后，调用方的所有流量全部都路由至被调用方的单个机房里。

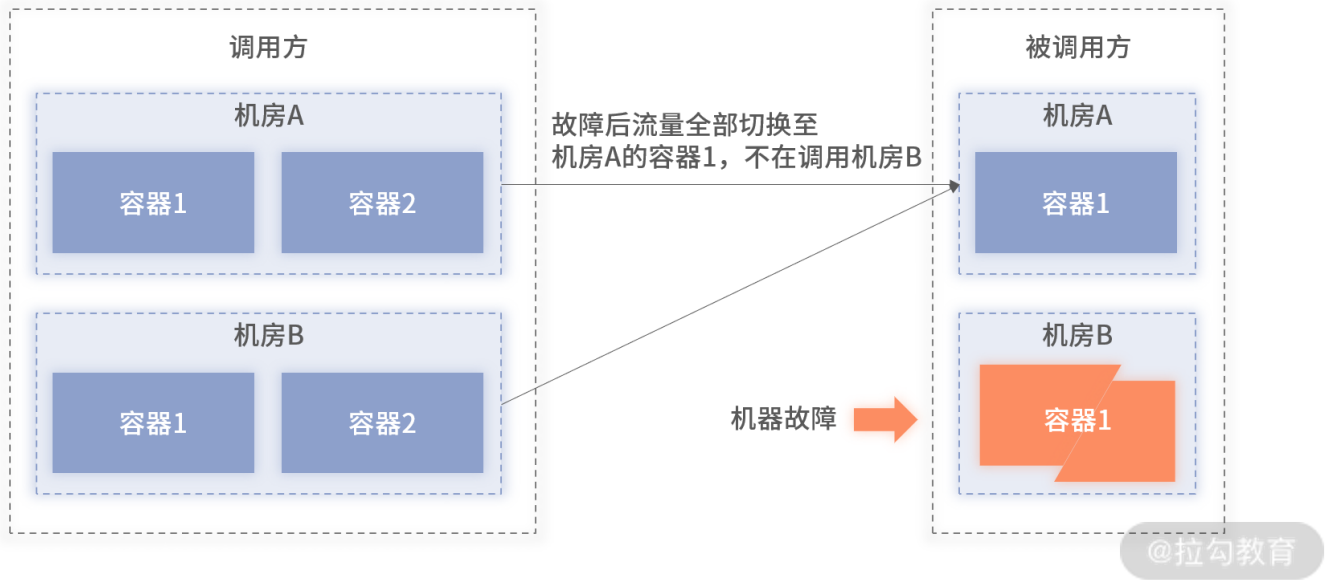


图 3：机器故障导致的跨机房架构图

**再者，不同类型的接口需要单独部署。** 在模块二和模块三里介绍过读服务和写服务的特点，这里再复习一下。读服务的特点是调用次数特别大，对于性能要求高。而写服务则是对于稳定性要求特别高，调用次数相比读服务会低很多。假设你在微服务拆分时，没有在垂直拆分时按读写分离的方式将读和写服务拆分开，而是将代码编写在同一个工程里。那么部署的时候，建议将二者的接口拆开部署，拆开后的结构如下图 4 所示：

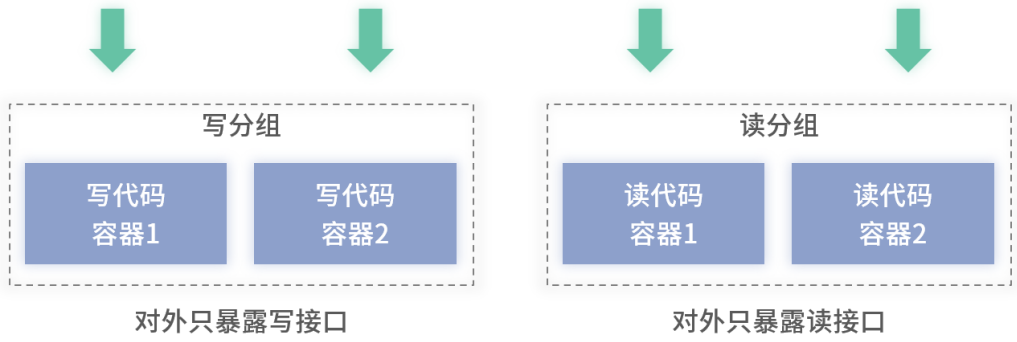

图 4：读写分离的部署架构

隔离开单独部署主要有以下几点考虑。

1. 写服务对于稳定性要求较高。隔离后，读服务里因为代码 Bug 等因素导致的机器 CPU 飙升、内存占满等问题不会影响到写服务的性能和稳定性。
2. 其次，读服务调用量较高，对于机器 CPU、内存、网络等占用也较高。隔离后，写服务将独享机器资源，性能和稳定性也较好。
3. 最后，微服务的执行线程是根据机器的 CPU 提前设置好的，大小是固定的。读写混合部署时，读请求很容易将微服务框架的执行线程沾满，导致线程枯竭，进而导致写请求得不到执行。此时，通过隔离部署也可以解决此问题。

**最后，至少要线程池隔离。** 在某些时候，可能读服务的调用次数并不是特别大或机器资源有限，实现不了上述的纯机器隔离。此时，可以实现一个简版的隔离，即微服务框架的执行线程池隔离。现在主流的微服务框架都支持对于接口单独配置一个执行线程，这样在执行时，就可以做到线程池资源隔离，互不影响，具体架构见如下图 5 所示。在某些无法完成机器隔离的场景里，可以使用此方式实现一定程度的资源隔离。

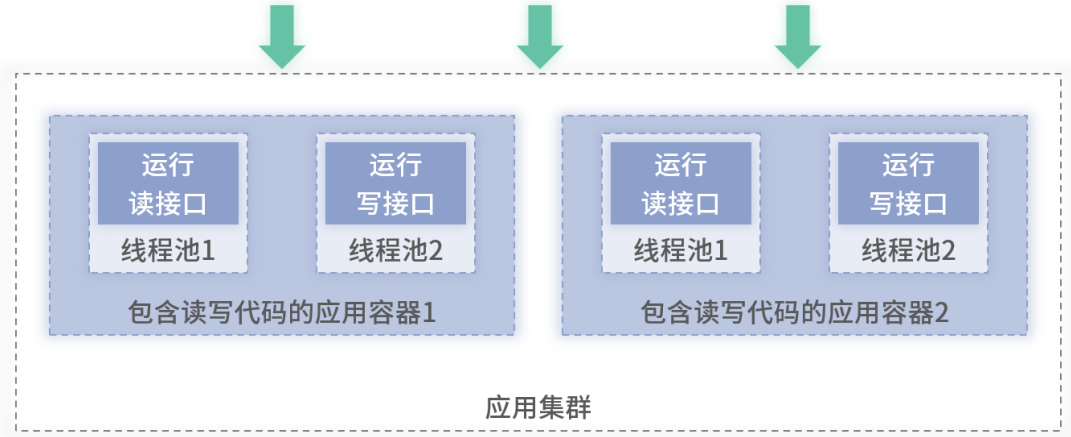



图 5：线程池隔离架构图

## 代码层面

有很多编写优雅、易阅读、易维护代码的技巧，因为篇幅和专栏定位原因，此处并不会一一介绍，此小节主要聚焦**如何编写避免系统故障的编码准则**。主要包含以下几点。

**第一，不要基于 JSON 格式打印太多、太复杂的日志。**

假设有一个特别复杂的类，其中包含了几十上百个字段，同时某些字段也是对象类型，该字段又嵌套了很多对象字段。如果在日志输出时，直接将该类通过 JSON 进行序列化，并进行日志输出，伪代码格式如下：

```
复杂Object obj=new 复杂Object();
logger.info(JSON.toJson(obj));
```

如果每一次请求，微服务的代码都会按上述格式打印日志，那么当调用量稍微上升时，很容易将微服务的 CPU 占满，进而导致服务宕机。导致上述现象的主要原因是：**复杂的对象在序列化时非常消耗 CPU 资源**。建议在打印日志时，按需输出。采用 toJson 方式序列化大对象，很多时候因为简单、粗暴，不需要太多开发量，所以就被研发同学大量、广泛地使用，与此同时也带来了系统宕机的风险。两害相较取其轻，建议采用如下按需的方式输出日志，规避宕机风险。

```
logger.info("内容1:{},内容2:{},内容3:{}",具体内容1,具体内容2,具体内容3);
```

**第二，需要具有日志级别的动态降级功能。**

假如上述按需输出日志的方式还没有被大家广泛接受，你还是习惯使用 toJson 的方式输出日志。那么为了防止打印日志导致机器宕机，需要在日志输出前进行级别判断，使得当日志打印导致机器出现问题时，通过此方式可以将日志进行关闭。具体写法如下：

```
if(logger.isInfoEnabled()){
  复杂Object obj=new 复杂Object();
  logger.info(JSON.toJson(obj));
}
```

当 toJson 的日志打印把 CPU 占满之后，可以将日志级别调整为更高等级，比如 error 级别，禁止日志输出即可规避问题。此外，更进一步的是，此日志级别调整可以开发动态功能，结合配置中心，动态的修改日志级别，可以实现不重启应用即可生效日志级别修改的功能。

**第三，for 循环不要嵌套太深，原则上不要超过三层嵌套。**

实践中，for 循环迭代的数据是从数据库或远程 RPC 获取的，获取到的数据量是动态的，可多可少，极端情况下可能多达上千条。此时，三层嵌套下的时间复杂度则为：$O(1000^3)$=10 亿。上亿次的代码执行，分分钟就会把微服务打挂，建议在代码编写时，规避此种写法。

**第四，多层嵌套需要有动态跳出的降级手段。**

假设业务上无法规避上述的多层嵌套，在实现时，需要在嵌套内部开发主动跳出的降级开关。当上述数据量增多时，此方式可以通过开关主动地跳出嵌套，防止机器宕机。

**第五，如果使用应用内的本地缓存，需要配置容量上限。**

如果不显式地配置本地缓存的容量上限，有可能因为容量暴涨，导致进程 OOM。因此，需要根据机器的内存大小，显式地配置本地缓存的容量上线。

## 总结

本讲介绍了设计不规范的微服务会产生的典型线上问题：机器 CPU 被打满的详细处理过程。后续遇到此类线上问题时，你可以参考上述过程进行处理。然后通过部署和代码两个层面讲解了可以规避服务器宕机的一些设计和编码技巧。后续你在设计和开发中，可以进行参考使用。

上述介绍的应用部署和代码编写的原则，都是为了防止微服务出现故障的手段。这些手段，换种说法就是做好自己，防止出现问题。

**除了上述介绍的一些原则，你们团队有哪些设计和编码规范呢？欢迎留言，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 19 |**如何做好微服务间依赖的治理和分布式事务？** 再见。