08 | 可监控设计：如何利用eBPF来指导可监控设计？

2021-06-03 尉刚强

《性能优化高手课》 课程介绍 >

讲述：尉刚强
时长 14:30 大小 13.28M

你好，我是尉刚强。今天这节课，我们会从系统监控的角度，来聊聊如何有效提升软件性能。

在前面的课程中，我介绍的并行设计、缓存设计、IO 设计等设计方法，实际上都只是从软件设计架构的维度去优化软件性能。但软件的生命周期一般是比较长的，伴随着新业务需求的不断演进，你还需要持续不断地对软件系统进行调优，才能保证软件性能长期具备竞争力。而要对一个软件系统做持续的性能调优，则是需要建立在对它进行观测数据的基础上。

那么今天我们要学习的**系统可监控设计，就是有针对性地设计系统应该对外提供哪些观测数据，以及如何实现这些测量数据的获取过程。**

我们都知道，在软件系统中，打印日志、统计信息等都属于观测数据，但是对于一个高性能的软件系统而言，需要提供的观测手段可远远不止这些。事实上，对于高性能软件系统的可监控设计来说，我们面临的**最大挑战**就是，**如何能够在获取最有效的观测数据的前提下，又能尽量减少对正常业务流程所产生的额外开销。**

既然如此，有没有什么有效的手段可以帮助我们解决这个难题呢？这就是我今天要给你介绍的 eBPF 技术了。我认为，它是一项能够为软件系统的可监控设计与实现，带来巨大改变的技术。至于原因，我在课程中就会给你揭晓答案。

这节课呢，我会从一个对高性能软件进行监控和观测实现时，经常需要面对的问题出发，与你探讨如果使用 eBPF 会带来什么样的改变，在此过程中，你就能够明白为什么在做可监控设计时需要使用 eBPF。然后，我会带你理解 eBPF 的核心架构原理，帮助你更加准确地找到使用场景，以此更好地支撑对系统的可监控设计。

好，下面我们就先来看下 eBPF 技术所带来的改变是什么吧。

eBPF 对系统观测分析的改变

不过在开始介绍 eBPF 之前，我想先带你来看一下，在一个高性能的软件系统内，现在常用的一些监控与观测手段在实现过程中都会面临什么样的困境。这样你就可以更直观地发现，为什么使用 eBPF 技术之后能够更好地解决这些难题。

目前常用的监控与观测手段都存在哪些不足？

首先，在一众的监控与观测手段中，成本最低的应该是**各种计数器、统计变量**等。

在嵌入式系统中，这些计数器或统计变量会记录到内存当中；而对分布式服务架构来说，则是更习惯记录到一些 Key-Value 数据库中，如 Redis 等。但是使用计数器存在的典型问题就是，我们只能看到一个个孤立的数字，所以采用这种手段来支撑软件系统的观测分析能力，实际上是比较有限的。

然后，**日志打印**也是比较常用的一个手段。不过，这种观测手段对于软件运行所产生的开销要比计数器大，所以在高性能系统中使用就会受控，比如我曾经参与的嵌入式实时性系统项目，在分析性能时一般都无法使用。另外，它对于分布式高性能的场景来说，也会因为 IO 开销比较大，所以使用效果不是太好。

日志打印还有一个比较明显的局限性，就是在高性能软件系统中使用日志时，一般都默认只打开错误日志，当系统出现问题需要进行分析时，才可以短暂地打开 INFO 级别日志进行查看。但是，如果系统的业务吞吐量很高，打开 INFO 日志直接就会导致系统崩溃，所以它对性能分析的帮助其实不大。不仅如此，大量的日志打印逻辑与业务代码通常都会揽和在一起，也会导致我们阅读代码的体验非常糟糕。

其实，**动态跟踪和监控**才是高性能软件系统中比较重要的分析手段，可是这些机制需要与业务代码一起实现，只有当接收到某个监控任务时才会触发执行，所以有不少高性能软件系统并没有认真去设计实现这里的功能。

而且它还存在一个问题，就是这种监控机制经常会和业务逻辑强耦合，实现起来会很复杂，同时我们在前期软件设计阶段，很难能考虑到所有的监控与观测场景，因而当后面出现性能问题时，只能靠不断定做特殊补丁软件版本来解决。

所以，根据以上对目前常用的监控与观测手段的介绍来看，我们其实能发现，对一个高性能的软件系统来说，设计和实现高效率的监控观测系统，确实是困难重重！

那么在使用 eBPF 之后，又会发生什么改变呢？

接下来，我们就通过一个具体的例子来了解一下，eBPF 为什么能够成为目前 Linux 生态中的热门技术，并真正改变和颠覆原有的软件系统监控和观测手段。

为什么 eBPF 可以帮助实现可监控设计？

这段代码是一个非常简单的 C 代码实现，函数在执行时可以不断输入数字，同时在函数中，还使用 DTRACE_PROBE1 定义了静态探针：

复制代码

```
#include <sys/sdt.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    while(1) {
        int age;
        scanf("%d", &age);
        DTRACE_PROBE1(sdt_group,sdt_age, age);  //定义一个静态探针。
        sleep(1);

```

```
    }
    return 0;
}
```

然后，在程序运行期间，你就可以使用下面这段 bpftrace 脚本，来跟踪打印上面程序中输入的值。

复制代码

```
sudo bpftrace \
    -e 'usdt:/home/**/**/sdt_group::sdt_age \
        { printf("%d\n", arg0); }' \
    -p $(pidof test-main)
12
32
44
```

看到这里，你是不是会觉得有些繁琐：**我直接在 main 函数中增加一个打印不香吗，为什么要搞得这么复杂呢？**

先别着急，下面我就带你来看看这样做都能带来哪些好处，然后你再去评判这样的做法是否值得。

第一个好处，就是使用这个 DTRACE_PROBE1 插入的静态跟踪点，在生成的二进制指令中只对应了一个 nop 指令。而这个 nop 指令，只需要消耗一个 CPU 指令周期，尤其在今天 CPU 超强的指令发射技术背景下，这个开销几乎可以忽略不计。所以，它就可以把这个监控实现对正常业务流程的开销，降低到了接近零开销。

第二个好处，就是你可以从应用程序的外部去选择开启或者关闭跟踪，完全不用修改业务程序。同时，你还可以选择性地去跟踪分析具体的某一个跟踪点，这样就避免了软件系统中，很多监控日志只能整体开启和关闭的尴尬。

第三个好处，就是在这种场景下，你可以使用 eBPF 在软件系统的外部，去注入和修改打印与跟踪逻辑，而不用担心影响到业务运行逻辑，同时还能让业务代码更加清爽！

所以不知你发现了没，把这些好处总结到一起，其实就是使用 eBPF 这个技术手段的优势之一。**它不仅可以提升系统的性能，同时还会让软件设计更加灵活。**

那么 eBPF 还有什么其他的使用优势呢？

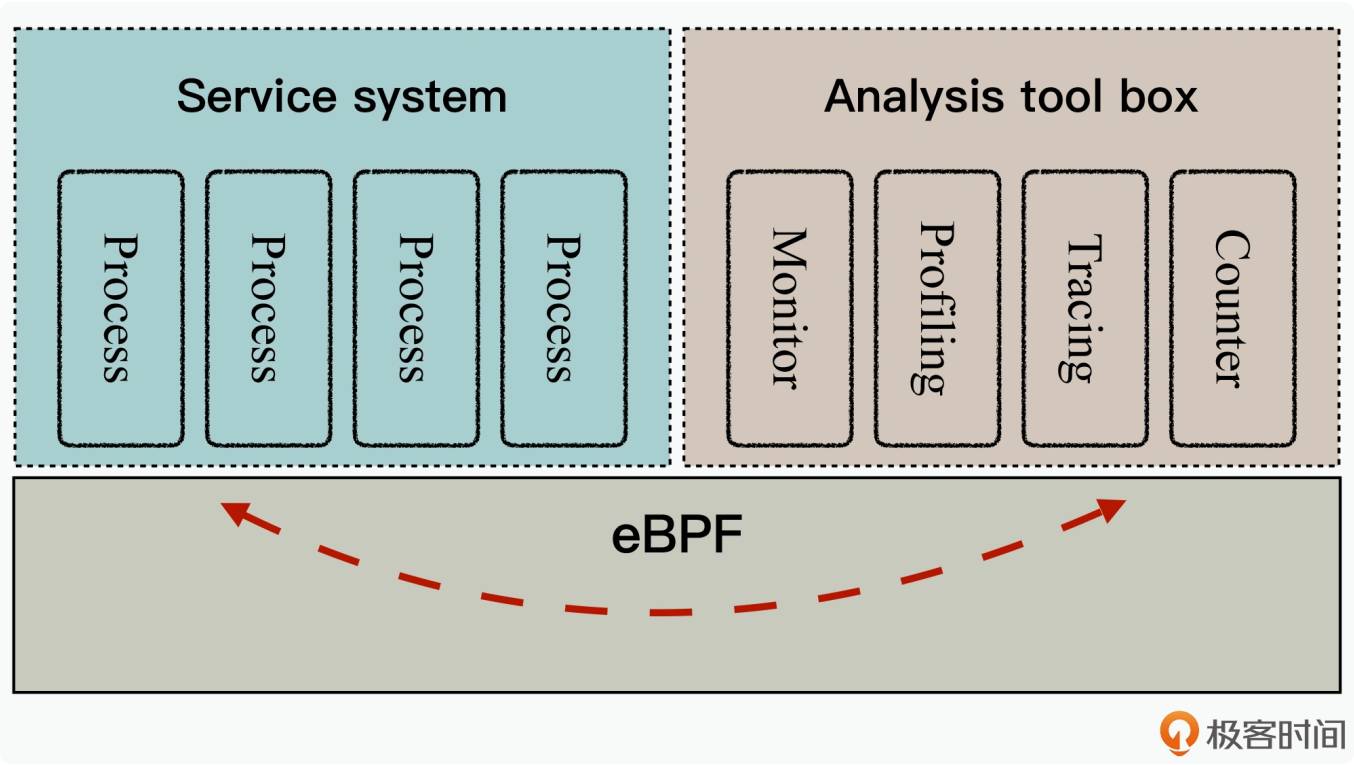
从前面的代码示例中，你可能会观察到在 bpftrace 脚本里，包含了一个 USDT（User Statically Defined Tracing）命令，这是一个接入到用户业务代码中进行静态定义的跟踪点，然后会获取跟踪信息并打印，这里使用的 USDT 技术其实只是 eBPF 中的一个技术而已。

实际上，现在不同的编程语言都在尝试着引入 USDT 技术，以此支持对开发软件的监控分析工作。比如说，Rust、Go、Python 使用的USDT，以及 Node.js、JVM 使用的USDT。

而除了 USDT 技术，在 eBPF 中还可以针对内核动态插桩（基于 kprobes 探针），针对用户态程序动态插桩（基于 uprobes 探针），以及针对内核和应用程序的静态插桩等各种能力。所以，基于 eBPF 技术，你就可以对监控、观测应用程序和系统的运行状态，具有了前所未有的可见性和灵活性。

正是因为 eBPF 技术这种强大的监控和观测能力，你在对高性能软件系统进行可监控设计时，就可以**把监控测量作为一个独立的侧面来设计**。

这又是什么意思呢？下面，为了让你更好地理解这个要点，我们一起来看一张示意图，这是一张软件业务与软件监控的设计解耦示意图：



可以看到，在这个图中主要分为三个大块，其中 Service system 代表了我们的业务系统，Analysis tool box 代表对业务系统的监控与分析工具箱，它们之间可以基于 eBPF 技术这个桥梁，实现高效互通。

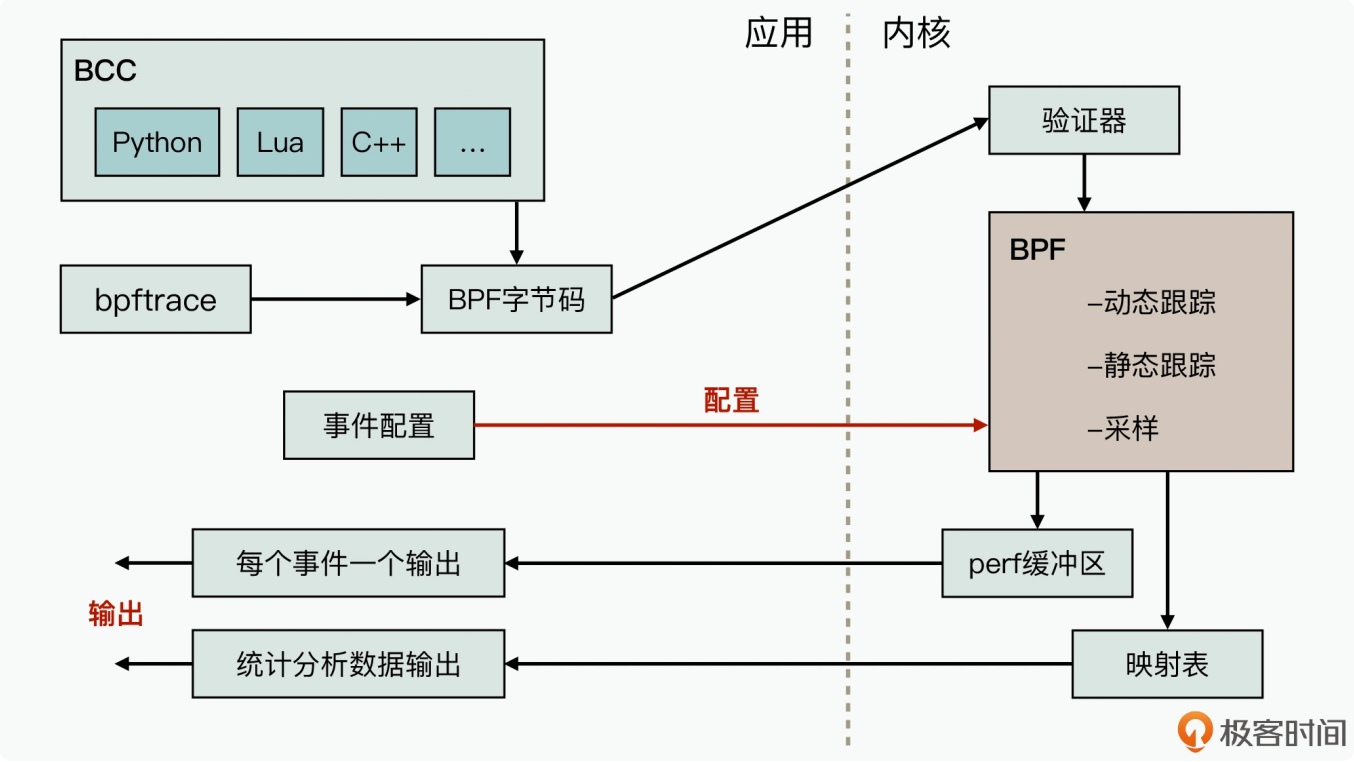
那么这也就是说，**当我们在对一个高性能系统进行监控设计时，就可以将监控观测设计与实现，从原来的业务逻辑中单独剥离出来进行。**

好了，前面我只是给你介绍了 eBPF 都有什么功能，以及它对可监控设计都会带来哪些帮助，现在你可能还是不清楚，为什么 eBPF 可以具备如此强大的能力。

所以接下来，我就给你详细介绍下 eBPF 的核心架构，这样当你拥有了对 eBPF 的全局认识之后，就可以更好地支撑你的系统可监控设计了。

eBPF 的实现原理和使用场景

首先我们来看一张示意图，这是 eBPF 的核心架构原理图，它可以帮助你更好地理解 eBPF 的机制和性能优势：



你可以发现，这张图被中间的一条线给分开了，左半部分代表的就是应用态软件，右边则是代表操作系统内核。其中，内核是一个操作系统中最核心的模块，我们经常说的进程调

度、设备管理、文件系统等，都属于内核，它是计算机上运行软件的大脑，控制着操作系统中软件的资源分配和调度。

现在，eBPF 的核心能力都已经集成到了内核中，而 eBPF 的功能则主要是由**处于应用态的前端和内核态的执行引擎**部分协同完成的。

其中，**BBC 和 bpftrace 是 eBPF 的前端**，你可以基于这两款工具来开发各种监控分析程序，这样开发出来的监控程序会编译成标准的 BPF 字节码，然后内核态的校验器执行完安全校验后，再交给内核态 eBPF 执行引擎来处理。

如此一来，通过这种方式，**内核态的 eBPF 引擎**提供的动态探知、静态探针、采样等丰富的观测监控手段，就可以很方便地被业务软件触发执行和控制。

另外，在软件系统运行过程中，**我们还可以不断地通过动态配置的方式来修改观测行为**。同时，内核态的 eBPF 引擎在获取到各种跟踪观测数据后，可以直接对数据进行加工处理，然后通过 perf 缓冲区和映射表数据结构，将处理后的数据再传输给应用态的监控工具。

也正是因为 eBPF 内核态的数据加工能力，就有效减少了内核态与应用态程序之间的内存拷贝数据量，进一步提升了监控性能。

而且，为了支持 eBPF 的处理性能，现在很多通用的 CPU 硬件上，都已经集成了专门用于 eBPF 程序的寄存器、函数栈等，所以说 eBPF 的运行性能优势会更大。

实际上，目前 eBPF 技术发展得非常快，不过在我们的课程设计中并不会系统讲解 eBPF，所以如果你还想更深入地了解它，我推荐你可以参考这个学习地址，其中包含了 eBPF 的理论、生态工具、丰富的开放样例等。

但是在今天的课程中，你需要明确理解和掌握 eBPF 这项技术是什么，以及它为高性能软件监控设计的解决思路所带来的变化是什么，然后你也可以根据这项技术的实现原理继续去思考，如何才能更好地实现高性能系统的可监控设计。

小结

其实，eBPF 对性能的改变并不限于监控观测分析，因为 eBPF 是标准化的，同时它还可以实现动态地改变内核态在一些场景下的行为，从而可以支撑 Linux 协议栈层上的性能优化，比如 bypass 优化等。

但这些已经都属于专有领域的范畴了，今天的课程上，我是站在一个高性能软件系统的可监控设计的视角，分析了 eBPF 技术可以帮助你解决哪些问题，并带你理解了 eBPF 的原理架构。学完今天的课程后，你就会更加清楚 eBPF 在性能上的优势，以及它应该在什么样的业务场景下被使用，这样你在自己的软件系统监控设计中，才会取得更好的效果。

思考题

使用 eBPF 技术开发出来的一般是比较小的工具，是否可以将这些工具和监控运维系统集成到一起呢？

欢迎给我留言，分享你的思考和看法。如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

上一篇 07 | 数据库选型：如何基于性能需求选择合适的数据库？

下一篇 09 | 性能模式（上）：如何有效提升性能指标？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。