OpenTelemetry
可观测性的未来

The Future of Observability with OpenTelemetry

目录

第 1 章

前言

软件开发的模式又一次改变了。开源软件和公有云供应商已经从根本上改变了我们构建和部署软件的方式。有了开源软件，我们的应用不再需要从头开始编码。而通过使用公有云供应商，我们不再需要配置服务器或连接网络设备。所有这些都意味着，你可以在短短几天甚至几小时内从头开始构建和部署一个应用程序。

但是，仅仅因为部署新的应用程序很容易，并不意味着操作和维护它们也变得更容易。随着应用程序变得更加复杂，更加异质，最重要的是，更加分布式，看清大局，以及准确地指出问题发生的地方，变得更加困难。

但有些事情并没有改变：作为开发者和运维，我们仍然需要能够从用户的角度理解应用程序的性能，我们仍然需要对用户的事务有一个端到端的看法。我们也仍然需要衡量和说明在处理这些事务的过程中资源是如何被消耗的。也就是说，我们仍然需要可观测性。

但是，尽管对可观测性的需求没有改变，我们实施可观测性解决方案的方式必须改变。本报告详细介绍了关于可观测性的传统思维方式对现代应用的不足：继续将可观测性作为一系列工具（尤其是作为“三大支柱”）来实施，几乎不可能以可靠的方式运行现代应用。

OpenTelemetry 通过提供一种综合的方法来收集有关应用程序行为和性能的数据，包括指标、日志和跟踪，来解决这些挑战。正如本报告所解释的，OpenTelemetry 是一种生成和收集这些数据的新方法，这种方法是为使用开源组件构建的云原生应用而设计的。

事实上，OpenTelemetry 是专门为这些开源组件设计的。与供应商特定的仪表不同，OpenTelemetry 可以直接嵌入到开放源代码中。这意味着开源库的作者可以利用他们的专业知识来添加高质量的仪表，而不需要在他们的项目中增加任何解决方案或供应商特定的代码。

OpenTelemetry 对应用程序所有者也有好处。在过去，遥测的产生方式与它的评估、存储和分析方式相联系。这意味着，选择使用哪种可观测性解决方案必须在开发过程的

早期完成，并且在之后很难改变。OpenTelemetry（就像它的前身 OpenTracing 和 OpenCensus 一样）将你的应用程序产生遥测的方式与遥测的分析方式相分离。

因此，当我们采用新技术时，我们往往没有考虑到该技术将如何被使用，特别是它将如何被不同角色的人使用。本报告描述了 OpenTelemetry 如何满足库作者、应用程序拥有者、运维和响应者的需求，以及它如何使这些角色中的每个人都能独立工作 —— 即自动做出关键决定并进行有效协作。

也许最重要的是，OpenTelemetry 为应用程序所有者和操作者提供了灵活性，使他们能够选择最适合其应用程序和组织需求的可观测性解决方案，包括需要多种工具的情况。它还包括这些需求随时间变化的情况，以及需要新的工具来满足这些需求的情况，因为随着技术的成熟和组织的发展，可能会出现这种情况。

虽然用户不需要致力于可观测性解决方案，但他们确实需要投资于生成遥测数据的一致方式。本报告展示了 OpenTelemetry 是如何被设计成范围狭窄、可扩展，而且最重要的是稳定的：如果你被要求进行这种投资，这正是你所期望的。

与以往任何时候相比，可观测性都不能成为事后的想法。没有一个有效的可观测性解决方案的风险，也就是说，无法了解你的应用程序中正在发生的事情的风险是非常高的，而对可观测性采取错误的方法的成本会造成你的组织多年来一直在偿还的债务。无论是长时间停机，还是开发团队被无休止的供应商迁移所困扰，你的组织的成功都取决于像 OpenTelemetry 这样的集成和开放的方法。

本报告提供了关于在你的组织中采用和管理 OpenTelemetry 的实用建议。它将使你开始走向成功的可观测性实践的道路，并释放出云原生和开源技术的许多真正的好处：你不仅能够快速建立和部署应用程序，而且能够可靠和自信地运行它们。

—— Daniel “Spoons” Spoonhower Lightstep

第 2 章

可观测性的历史

可观测性行业正处在巨变中。

传统上，我们观察系统时使用的是一套筒仓式的、独立的工具，其中大部分包含结构不良（或完全非结构化）的数据。这些独立的工具也是垂直整合的。工具、协议和数据格式都属于一个特定的后端或服务，不能互换。这意味着更换或采用新的工具需要耗费时间来更换整个工具链，而不仅仅是更换后端。

这种孤立的技术格局通常被称为可观测性的“三大支柱”：日志、度量和（几乎没有）追踪（见图 1-1）。

日志

记录构成事务的各个事件。

度量

记录构成一个事务的事件的集合。

追踪

测量操作的延迟和识别事务中的性能瓶颈，或者类似的东西。传统上，许多组织并不使用分布式追踪，许多开发人员也不熟悉它。

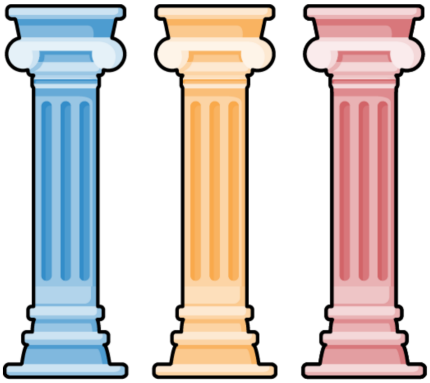

图 2-1: 可观测性的 ” 三大支柱”。

我们用这种方法工作了很久，以致于我们不常质疑它。但正如我们将看到的，“三大支柱”并不是一种正确的结构化的可观测性方法。事实上，这个术语只是描述了某些技术碰巧被实现的方式，它掩盖了关于如何实际使用我们的工具的几个基本事实。

2.1 什么是事务和资源?

在我们深入探讨不同可观测性范式的利弊之前，重要的是要定义我们所观察的是什么。我们最感兴趣的分布式系统是基于互联网的服务。这些系统可以被分解成两个基本组成部分：**事务**和**资源**。

事务（transaction）代表了分布式系统需要执行的所有动作，以便服务能够做一些有用的事情。加载一个网页，购买一个装满物品的购物车，订购一个共享汽车：这些都是事务的例子。重要的是要理解，事务**不仅仅是**数据库事务。对每个服务的每个请求都是事务的一部分。

例如，一个事务可能从一个浏览器客户端向一个网络代理发出 HTTP 请求开始。代理首先向认证系统发出请求以验证用户，然后将请求转发给前端应用服务器。应用服务器向各种数据库和后端系统发出若干请求。例如，一个消息系统：Kafka 或 AMQP，被用来排队等待异步处理的额外工作。所有这些工作都必须正确完成，以便向急切等待的用户提供结果。如果任何部分失败或耗时过长，其结果就是糟糕的体验，所以我们需要全面理解事务。

一路下来，所有这些事务都在消耗**资源（resource）**。网络服务只能处理这么多的并发请求，然后它们的性能就会下降并开始失效。这些服务可以扩大规模，但它们与可能调用锁的数据库互动，造成瓶颈。数据库读取记录的请求可能会阻碍更新该记录的请求，反之亦然。此外，所有这些资源都是要花钱的，在每个月的月底，你的基础设施供应商会给你发账单。你消耗的越多，你的账单就越长。

如何解决某个问题或提高服务质量？要么是开发人员修改事务，要么是运维人员修改可用资源。就这样了。这就是它的全部内容。当然，魔鬼就在细节中。

第四章将详细介绍可观测性工具表示事务和资源的理想方式。但是，让我们回到描述迄今为止我们一直在做的非理想的方式。

> **不是所有事情都是事务**
>
> 请注意，除了跨事务之外，还有其他的计算模型。例如，桌面和移动应用程序通常是基于**反应器（reactor）**模型，用户在很长一段时间内连续互动。照片编辑和视频游戏就是很好的例子，这些应用有很长的用户会话，很难描述为离散的事

> 务。在这些情况下，基于事件的可观测性工具，如**真实用户监控**（RUM），可以增强分布式追踪，以更好地描述这些长期运行的用户会话。
>
> 也就是说，几乎所有基于互联网的服务都是建立在事务模式上的。由于这些是我们关注的服务，观察事务是我们在本书中描述的内容。附录 B 中对 RUM 进行了更详细的描述。

2.2 可观测性的三大支柱

考虑到事务和资源，让我们来看看三大支柱模式。将这种关注点的分离标记为“三大支柱”，听起来是有意的，甚至是明智的。支柱听起来很严肃，就像古代雅典的帕特农神庙。但这种安排实际上是一个意外。计算机的可观测性由多个孤立的、垂直整合的工具链组成，其真正的原因只是一个平庸的故事。

这里有一个例子。假设有一个人想了解他们的程序正在执行的事务。所以他建立了一个日志工具：一个用于记录包含时间戳的消息的接口，一个用于将这些消息发送到某个地方的协议，以及一个用于存储和检索这些消息的数据系统。这足够简单。

另一个人想监测在任何特定时刻使用的所有资源；他想捕捉指标。那么，他不会为此使用日志系统，对吗？他想追踪一个数值是如何随时间变化的，并跨越一组有限的维度。很明显，一大堆非结构化的日志信息与这个问题没有什么关系。因此，一个新的、完全独立的系统被创造出来，解决了生成、传输和存储度量的具体问题。

另一个人想要识别性能瓶颈。同样，日志系统的非结构化性质使它变得无关紧要。识别性能瓶颈，比如一连串可以并行运行的操作，需要我们知道事务中每个操作的持续时间以及这些操作是如何联系在一起的。因此，我们建立了一个完全独立的追踪系统。由于新的追踪系统并不打算取代现有的日志系统，所以追踪系统被大量抽样，以限制在生产中运行第三个可观测系统的成本。

这种零敲碎打的可观测性方法是人类工程的一个完全自然和可理解的过程。然而，它有其局限性，不幸的是，这些局限性往往与我们在现实世界中使用（和管理）这些系统的方式相悖。

2.3 实际上我们如何观察系统?

让我们来看看运维人员在调查问题时所经历的实际的、不加修饰的过程。

调查包括两个步骤：注意到有事情发生，然后确定是什么原因导致了它的发生。

当我们执行这些任务时，我们使用可观测性工具。但是，重要的是，我们并不是孤立地使用每一个工具。我们把它们全部放在一起使用。而在整个过程中，这些工具的孤立性给操作者带来了巨大的认知负担。

通常情况下，当有人注意到一个重要的指标**变得歪七扭八时**，调查就开始了。在这种情况下，“毛刺” 是一个重要的术语，因为运维人员在这一点上所掌握的唯一信息是仪表盘上一条小线的形状，以及他们自己对该线的形状是否看起来 “正确” 的内部评估（图 1-2）。

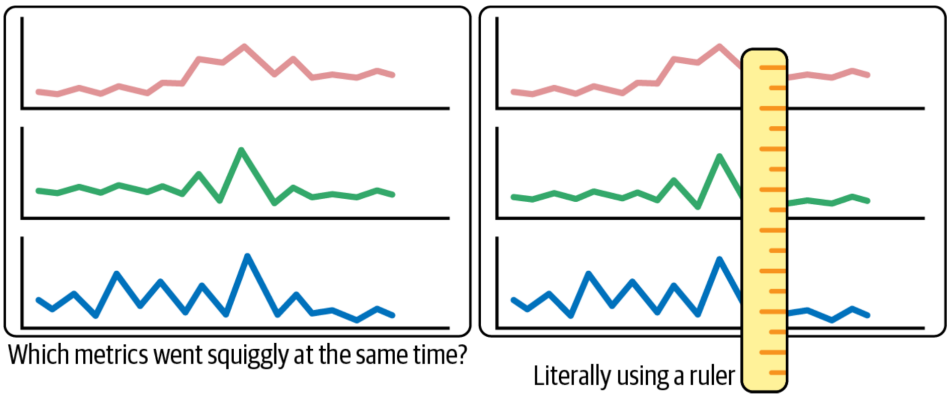

图 2-2: 寻找不同数据集之间的关联性是一种可怕的经历。

在确定了线的形状开始看起来 “不对劲” 的那一点后，运维人员将眯起眼睛，试图找到仪表板上同时出现 “毛刺” 的其他线。然而，由于这些指标是完全相互独立的，运维人员必须在他们的大脑中进行比较，而不需要计算机的帮助。

不用说，盯着图表，希望找到一个有用的关联，需要时间和脑力，更不用说会导致眼睛疲劳。

就个人而言，我会用一把尺子或一张纸，只看什么东西排成一排。在 “现代” 仪表盘中，标尺现在是作为用户界面的一部分而绘制的线。但这只是一个粗略的解决方案。识别相关性的真正工作仍然必须发生在操作者的头脑中，同样没有计算机的帮助。

在对问题有了初步的、粗略的猜测后，运维人员通常开始调查他们认为可能与问题有关的事务（日志）和资源（机器、进程、配置文件）（图 1-3）。

在这里，计算机也没有真正的帮助。日志存储在一个完全独立的系统中，不能与任何指标仪表板自动关联。配置文件和其他服务的具体信息通常**不在任何系统**中，运维人员必须通过 SSH 或其他方式访问运行中的机器来查看它们。

因此，运维人员再次被留下寻找相关性的工作，这次是在指标和相关日志之间。识别这些日志可能很困难；通常必须查阅源代码才能了解可能存在的日志。

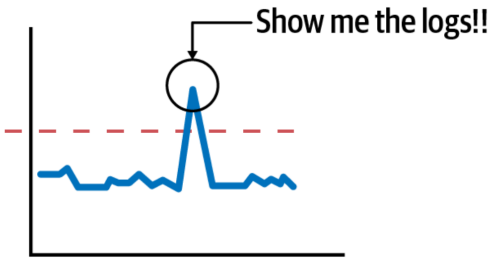

图 2-3: 图 1-3：找到与异常情况相关的日志也是一种可怕的经历。

当找到一个（可能是，希望是）相关的日志，下一步通常是确定导致这个日志产生的事件链。这意味着要找到同一事务中的其他日志。

缺乏关联性给操作者带来了巨大的负担。非结构化和半结构化的日志系统没有自动索引和按事务过滤日志的机制。尽管这是迄今为止运维人员最常见的日志工作流程，但他们不得不执行一系列特别的查询和过滤，将可用的日志筛选成一个子集，希望能代表事务的近似情况。为了成功，他们必须依靠应用程序开发人员来添加各种请求 ID 和记录，以便日后找到并拼接起来。

在一个小系统中，这种重建事务的过程是乏味的，但却是可能的。但是一旦系统发展到包括许多横向扩展的服务，重建事务所需的时间就开始严重限制了调查的范围。图 1-4 显示了一个涉及许多服务的复杂事务。你将如何收集所有的日志?

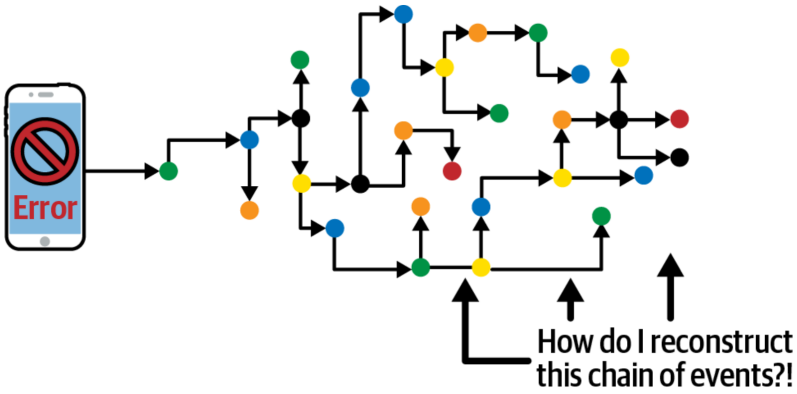

图 2-4: 在传统的日志记录中，找到构成一个特定事务的确切日志需要花费大量的精力。如果系统变得足够大，这几乎成为不可能。

分布式追踪是一个好的答案。它实际上拥有自动重建一个事务所需的所有 ID 和索引工具。不幸的是，追踪系统经常被看作是用于进行延迟分析的利基工具。因此，发送给它们的日志数据相对较少。而且，由于它们专注于延迟分析，追踪系统经常被大量采样，使得它们与这类调查无关。

2.4 不是三根柱子，而是一股绳子

不用说，这是一个无奈之举。上述工作流程确实代表了一种可怕的状态。但是，由于我们已经在这种技术体制下生活了这么久，我们往往没有认识到，与它可以做到的相比，它实际上是多么低效。

今天，为了了解系统是如何变化的，运维人员必须首先收集大量的数据。然后，他们必须根据仪表盘显示和日志扫描等视觉反馈，用他们的头脑来识别这些数据的相关性。这是一种紧张的脑力劳动。如果一个计算机程序能够自动扫描和关联这些数据，那么这些脑力劳动就是不必要的。如果运维人员能够专注于调查他们的系统是**如何**变化的，而不需要首先确定**什么**在变化，那么他们将节省大量宝贵的时间。

在编写一个能够准确执行这种变化分析的计算机程序之前，所有这些数据点都需要被连接起来。日志需要被连接在一起，以便识别事务。衡量标准需要与日志联系在一起，这样产生的统计数据就可以与它们所测量的事务联系起来。每个数据点都需要与底层系统资源 —— 软件、基础设施和配置细节相关联，以便所有事件都能与整个系统的拓扑结构相关联。

最终的结果是一个单一的、可遍历的图，包含了描述分布式系统状态所需的所有数据，这种类型的数据结构将给分析工具一个完整的系统视图。与其说是不相连的数据的“三根支柱”，不如说是相互连接的数据的一股绳子。

OpenTelemetry 因此诞生。如图 1-5 所示，OpenTelemetry 是一个新的遥测系统，它以一种综合的方式生成追踪、日志和指标。所有这些连接的数据点以相同的协议一起传输，然后可以输入计算机程序，以确定整个数据集的相关性。

图 2-5: OpenTelemetry 将所有这些孤立的信息，作为一个单一的、高度结构化的数据流连接起来。

这种统一的数据是什么样子的？在下一章，我们将把这三个支柱放在一边，从头开始建立一个新的模型。

第 3 章

结构化数据的价值

眯着眼睛看图表并不是寻找相关性的最佳方式。目前在运维人员头脑中进行的大量工作实际上是可以自动化的。这使运维人员可以在识别问题、提出假设和验证根本原因之间迅速行动。

为了建立更好的工具，我们需要更好的数据。遥测必须具备以下两个要求以支持高质量的自动分析：

- 所有的数据点都必须用适当的索引连接在一个图上。
- 所有代表常见操作的数据点必须有明确的键和值。

在这一章中，我们将从一个基本的构件开始浏览现代遥测数据模型：属性。

3.1 属性：定义键和值

最基本的数据结构是**属性（attribute）**，定义为一个键和一个值。OpenTelemetry 的每个数据结构都包含一个属性列表。分布式系统的每个组件（HTTP 请求、SQL 客户端、无服务器函数、Kubernetes Pod）在 OpenTelemetry 规范中都被定义为一组特定的属性。这些定义被称为 OpenTelemetry **语义约定**。表 2-1 显示了 HTTP 约定的部分列表。

属性	类型	描述	示例
http.method	string	HTTP 请求类型	GET; POST; HEAD
http.target	string	在 HTTP 请求行中传递的完整的请求目标或等价物	/path/12314/

属性	类型	描述	示例
http.host	string	HTTP host header 的值。当标头为空或不存在时，这个属性应该是相同的。	www.example.org
http.scheme	string	识别所使用协议的 URI 方案	http; https
http.status_code	int	HTTP 请求状态码	200

表 2-1：HTTP 规范的部分列表

有了这样一个标准模式，分析工具就可以对它们所监测的系统进行详细的表述，同时进行细微的分析，而在使用定义不明确或不一致的数据时，是不可能做到的。

3.2 事件：一切的基础

OpenTelemetry 中最基本的对象是**事件（event）**。事件只是一个时间戳和一组属性。使用一组属性而不是简单的消息 / 报文，可以使分析工具正确地索引事件，并使它们可以被搜索到。

有些属性对事件来说是独一无二的。时间戳、消息和异常细节都是特定事件的属性的例子。

然而，大多数属性对单个事件来说**并不**独特。相反，它们是一组事件所共有的。例如，`http.target` 属性与作为 HTTP 请求的一部分而记录的每个事件有关。如果在每个事件上反复记录这些属性，效率会很低。相反，我们把这些属性拉出到围绕事件的封装中，在那里它们可以被写入一次。我们把这些封装称为**上下文（context）**。

有两种类型的上下文：静态和动态（如图 2-1 所示）。**静态上下文**定义了一个事件发生的物理位置。在 OpenTelemetry 中，这些静态属性被称为**资源**。一旦程序启动，这些资源属性的值通常不会改变。

动态上下文定义了事件所参与的活动操作。这个操作层面的上下文被称为**跨度（span）**。每次操作执行时，这些属性的值都会改变。

不是所有的事件都有两种类型的上下文。只有资源的自由浮动事件，如程序启动时发出的事件，被称为**日志（log）**。作为分布式事务的一部分而发生的事件被称为**跨度事件（span event）**。

3.3 资源：观察服务和机器

资源（静态上下文）描述了一个程序正在消费的物理和虚拟信息结构。服务、容器、部署和区域都是资源。图 2-2 显示了一个典型的购物车结账事务中所涉及的资源。

系统运行中的大多数问题都源于资源争夺，许多并发的事务试图在同一时间利用相同的资源。通过将事件放在它们所使用的资源的上下文中，就有可能自动检测出许多类型的资源争夺。

像事件一样，资源可以被定义为一组属性。表 2-2 显示了一个服务资源的例子。

表 2-2：服务资源的例子

属性	类型	描述	示例
service.name	string	服务的逻辑名称	shopping cart
service.instance.id	string	服务实例的 ID	627cc493-f310-47de-96bd-71410b7dec09
service.version	string	服务 API 或者实现的版本号	2.0.0

除了识别机器所需的基本信息，配置设置也可以作为资源被记录下来。要访问一台正在运行的机器来了解它是如何配置的，这个负担太让人害怕了。相反，在配置文件中发现的任何重要信息也应该表示为一种资源。

3.4 跨度：观察事务

跨度（动态上下文）描述计算机操作。跨度有一个操作名称，一个开始时间，一个持续时间，以及一组属性。

标准操作是使用语义约定来描述的，比如上面描述的 HTTP 约定。但也有一些特定的应用属性，如 `ProjectID` 和 `AccountID`，可以由应用开发者添加。

跨度也是我们描述因果关系的方式。为了正确记录整个事务，我们需要知道哪些操作是由其他哪些操作触发的。为了做到这一点，我们需要给跨度增加三个属性：`TraceID`、`SpanID` 和 `ParentID`，如表 2-3 所示。

表 2-3：跨度的三个额外属性

属性	类型	描述	示例
traceid	16 字节数组	识别整个事务	4bf92f3577b34da6a3ce929d0e0e4736
spanid	8 字节数组	识别当前操作	00f067aa0ba902b7
parentid	8 字节数组	识别父操作	53ce929d0e0e4736

这三个属性是 OpenTelemetry 的基础。通过添加这些属性，我们所有的事件现在可以被组织成一个图，代表它们的因果关系。这个图现在可以以各种方式进行索引，我们稍后会讨论这个问题。

3.5 追踪：看似日志，胜过日志

我们现在已经从简单的事件变成了组织成与资源相关的操作图的事件。这种类型的图被称为**追踪（trace）**。图 2-3 显示了一种常见的可视化追踪方式，重点是识别操作的延迟。

从本质上讲，追踪只是用更好的索引来记录日志。当你把适当的上下文添加到适当的结构化的日志中时，可以得到追踪的定义。

想想你花了多少时间和精力通过搜索和过滤来收集这些日志；那是收集数据的时间，而不是分析数据的时间。而且，你要翻阅的日志越多，执行并发事务数量不断增加的机器堆积，就越难收集到真正相关的那一小部分日志。

然而，如果你有一个 TraceID，收集这些日志只是一个简单的查询。通过 TraceID 索引，

你的存储工具可以自动为你做这项工作；找到一个日志，你就有了该事务中的所有日志，不需要额外的工作。

既然如此，为什么你还会要那些没有 “追踪” ID 的 “日志”？我们已经习惯了传统的日志管理迫使我们做大量的工作来连接这些点。但这些工作实际上是不必要的；它是我们数据中缺乏结构的副产品。

分布式追踪不仅仅是一个测量延迟的工具；它是一个定义上下文和因果关系的数据结构。它是把所有东西联系在一起的胶水。正如我们将看到的，这种胶水包括最后一个支柱 —— 指标。

3.6 指标：观察事件的总体情况

现在我们已经确定了什么是事件，让我们来谈谈事件的聚合。在一个活跃的系统中，同样的事件会不断发生，我们以聚合的方式查看它们的属性来寻找模式。属性的值可能出现得太频繁，或者不够频繁，在这种情况下，我们要计算这些值出现的频率。或者该值可能超过某个阈值，在这种情况下，我们想衡量该值是如何随时间变化的。或者我们可能想以直方图的形式来观察数值的分布。

这些聚合事件被称为**度量**。就像普通的事件一样，度量有一组属性和一组语义上的便利条件来描述普通概念。表 2-4 显示了一些系统内存的例子属性。

表 2-4：系统内存的属性

属性	值类型	属性值
system.memory.usage	int64	used, free, cached, other
system.memory.utilization	double	used, free, cached, other

3.7 与事件相关的指标：统一的系统

传统上，我们认为指标是与日志完全分开的。但实际上它们是紧密相连的。例如，假设一个 API 有一个衡量每分钟错误数量的指标。那是一个统计数字。然而，每一个错误都是由一个特定的事务行为产生的，使用特定的资源。这些细节在我们每次递增该计数器时都会出现，我们想知道这些细节。

当运维人员被提醒发现错误突然激增时，他们会想到的第一个问题自然是：“是什么导致了这个激增?” 看一下例子的追踪可以回答这个问题。在失败的事务中较早发生的事件（或未能发生的事件）可能是错误的来源。

在 OpenTelemetry 中，当指标事件在跨度的范围内发生时，这些追踪的样本会自动与指标相关联，作为**追踪的范例（trace examplar）**。这意味着不需要猜测或寻找日志。OpenTelemetry 明确地将追踪和度量联系在一起。一个建立在 OpenTelemetry 上的分析工具可以让你从仪表盘上直接看到追踪，只需一次点击。如果有一个模式 —— 例如，一个特定的属性值与导致一个特定错误的追踪密切相关 —— 这个模式可以被自动识别。

3.8 自动分析和编织

事件、资源、跨度、指标和追踪：这些都被 OpenTelemetry 连接在一个图中，并且它们都被发送到同一个数据库，作为一个整体进行分析。这就是下一代的可观测性工具。

现代可观测性将建立在使用结构的数据上，这些结构允许分析工具在所有类型的事件和总量之间进行关联，这些关联将对我们如何实践可观测性产生深远影响。

向全面观察我们的系统过渡将有许多好处。但我相信，这些新工具提供的主要省时功能将是各种形式的**自动关联检测**。在寻找根本原因时，注意到相关关系可以产生大量的洞察力。如图 2-4 所示，相关性往往是产生根本原因假设的关键因素，然后可以进一步调查。

平均表现是什么样子的？异常值是什么样子的？哪些趋势在一起变化，它们的共同点是什么？相关性可能发生在许多地方：跨度中的属性之间、追踪中的跨度之间、追踪和资源之间、指标内部，以及所有这些地方都有。当所有这些数据被连接到一个图中时，这些相关性就可以被发现了。

这就是为什么统一的数据编织是如此关键。任何自动匹配的分析的价值完全取决于被分析的数据的结构和质量。机器遍历数据的图表；它们不会进行逻辑的飞跃。准确的统计分析需要一个有意设计的遥测系统来支持它。

3.9 重点：自动分析为您节省时间

为什么我们关心相关性分析的自动化？因为时间和复杂性对我们不利。随着系统规模的扩大，它们最终变得太复杂了，任何操作者都无法完全掌握系统的情况，而且在建立一个假设时，永远没有足够的时间来调查每一个可能的联系。

问题是，选择调查什么需要直觉，而直觉往往需要对组成分布式系统的每个组件有深刻

的了解。随着企业系统的增长和工程人员的相应增加，任何一个工程师对每个系统深入了解的部分自然会缩减到整个系统的一小部分。直觉并不能很好地扩展。

直觉也极易被误导；问题经常出现在意想不到的地方。根据定义，可以预见的问题几乎不会经常发生。剩下的就是所有未曾预料到的问题了，这些问题已经超出了我们的直觉。

这就是自动关联检测的作用。有了正确的数据，机器可以更有效地检测出相关的关联。这使得运维人员能够快速行动，反复测试各种假设，直到他们知道足够的信息来制定解决方案。

第 4 章

自动分析的局限性

自动化开始听起来很神奇，但我们要面对现实：计算机分析不能每天告诉你系统有什么问题或为你修复它。它只能为你节省时间。

至此，我想停止夸赞，我必须承认自动化的一些局限性。我这样做是因为围绕结合人工智能和可观测性会有相当多的炒作。这种炒作导致了惊人的论断，大意是：“人工智能异常检测和根源分析可以完全诊断问题!”和“人工智能操作将完全管理你的系统!”

4.1 谨防炒作

为了摆脱此类炒作，我想明确的是，这类梦幻般的营销主张**并不是**我所宣称的现代可观测性将提供的。事实上，我预测许多与人工智能问题解决有关的主张大多是夸大其词。

为什么人工智能不能解决我们的问题？一般来说，机器无法识别软件中的“问题”，因为定义什么是“问题”需要一种主观的分析方式。而我们所讨论的那种现实世界的人工智能不能以任何程度的准确性进行主观决策。机器将始终缺乏足够的背景。

例如，我们说该版本降低了性能，这里面也隐含了一个期望，就是这个版本包含了每个用户都期望的新功能。对于这个版本，性能退步是一个特征，而不是一个错误。

虽然它们可能看起来像类似的活动，但在**确定相关关系**和**确定根本原因**之间存在着巨大的鸿沟。当你有正确的数据结构时，相关性是一种**客观的分析** —— 你只需要计算数字。哪些相关关系是相关的，并表明真正问题的来源，总是需要**主观的分析**，对数据的解释。所有这些相关关系意味着什么？正如杰弗里·李波斯基（Jeffrey Lebowski）所说：“嗯，你知道，这只是你的观点，伙计。”

4.2 神奇的 AIOps

在调查一个系统时，有两种类型的分析起作用：

- 客观的分析，基于事实的、可衡量的、可观测的。

- 主观分析，基于解释、观点和判断的。

这种二分法 —— 客观与主观，与可计算性理论中一个重要的问题有关，即 **停机问题（halting problem）**。停机问题的定义是，在给定任意计算机程序及其输入的描述的情况下，是否可以编写一个计算机程序来确定任意程序是否会结束运行或永远继续运行。简而言之，在 1936 年，艾伦·图灵（Alan Turning）证明了解决停机问题的一般算法是不存在的，这个证明的延伸可以应用于计算机软件中许多形式的识别 “问题”。

阿兰·图灵的意思是，我们没有办法拥有神奇的 AIOps（IT 运维的人工智能）。寻找那些承诺将繁琐的客观分析自动化的工具 —— 计算数字是机器的强项！但要小心那些声称能找到问题根源并自动修复的工具。它们很可能会让你失望。

这就是为什么我们可以确定相关关系，但不能确定因果关系：想象一下，有一台机器可以确定任意计算机程序中的问题行为是什么，并确定该行为的根本原因。如果我们真的造出了这样一台机器，那就是开香槟的时候了，因为这意味着我们终于解决了停机问题！但是，目前还没有迹象表明，机器可以解决这个问题。然而，没有迹象表明机器学习已经超越了艾伦·图灵的统一计算模型；你可以相信，这不会发生。

做出正确的决定和修复的工作还是要靠你自己。

4.3 时间是最宝贵的资源

然而，我们不需要神奇的 AIOps 来看到我们工作流程的巨大改善。识别相关性，同时获取相关信息，以便你能有效地浏览这些信息，这是计算机**绝对可以**做到的事情！这将为你节省时间。大量的时间。这么多的时间，它将从根本上改变你调查系统的方式。

减少浪费的时间是实践现代可观测性的核心。即使是在简单的系统中，通过分析数字来识别相关性也是很困难的，而在大规模的系统中，这几乎是不可能的。通过将认知负担转移到机器上，运维人员能够有效地管理那些已经超出人类头脑所能容纳的系统。

但是，我们分析遥测方式的这种转变并不是可观测性世界中即将发生的唯一重大变化。我们需要的大部分遥测数据来自于我们没有编写的软件：我们所依赖的开源库、数据库和管理服务。这些系统在传统上一直在为产生遥测数据而奋斗。我们可以获得哪些数据，以及这些数据来自哪里，也将发生根本性的变化。

第 5 章

支持开源和原生监测

到目前为止，我们已经从数据的角度讨论了现代可观测性。但是，现代可观测性还有另一个方面，从长远来看，它可能被证明同样重要：如何对待产生数据的仪表（instrumention）。

大多数软件系统都是用现成的部件构建的：网络框架、数据库、HTTP 客户端、代理服务器、编程语言。在大多数组织中，很少有这种软件基础设施是在内部编写的。相反，这些组件是在许多组织中共享的。最常见的是，这些共享组件是以开放源码（OSS）库的形式出现的，并具有许可权。

由于这些开放源码软件库几乎囊括了一般系统中的所有关键功能，因此获得这些库的高质量说明对大多数可观测性系统来说至关重要。

传统上，仪表是“单独出售”的。这意味着，软件库不包括产生追踪、日志或度量的仪表。相反，特定解决方案的仪表是在事后添加的，作为部署可观测系统的一部分。

> **什么是特定解决方案仪表？**
>
> 在本章中，术语“**特定解决方案仪表**”是指任何旨在与特定的可观测系统一起工作的仪表，使用的是作为该特定系统的数据存储系统的产物而开发的客户端。在这些系统中，客户端和存储系统常常深深地融合在一起。因此，如果一个应用要从一个观测系统切换到另一个观测系统，通常需要进行全面的重新布设。

针对解决方案的仪表是“三大支柱”中固有的垂直整合的遗留问题。每个后端都摄取特定类型的专有数据；因此，这些后端的创建者也必须提供产生这些数据的仪表。

这种工具化的方法给参与软件开发的每个人都带来了麻烦：供应商、用户和开放源码库的作者。

5.1 可观测性被淹没在特定解决方案的仪表中

从可观测性系统的角度来看，仪表化代表了巨大的开销。

在过去，互联网应用是相当同质化的，可以围绕一个特定的网络框架来建立可观测性系统。Java Spring、Ruby on Rails 或 .NET。但随着时间的推移，软件的多样性已经爆炸性增长。现在为每一个流行的网络框架和数据库客户端维护仪表是一项巨大的负担。

这导致的重复劳动难以估量。传统上，供应商将他们在仪表上的投资作为销售点和把关的一种形式。但是，日益增长的软件开发速度已经开始使这种做法无法维持了。对于一个合理规模的仪表设备团队来说，覆盖面实在是太大了，无法跟上。

这种负担对于新的、新颖的观测系统，特别是开放源码软件的观测项目来说尤其严峻。如果一个新的系统在编写了大量的仪表之前无法在生产中部署，而一个开放源码软件项目在广泛部署之前也无法吸引开发者的兴趣，那么科学进步就会陷入僵局。这对于基于追踪的系统来说尤其如此，它需要端到端的仪表来提供最大的价值。

5.2 应用程序被锁定在特定解决方案的仪表中

从应用开发者的角度来看，特定解决方案的仪表代表了一种有害的锁定形式。

可观测性是一个交叉性的问题。要彻底追踪、记录或度量一个大型的应用程序，意味着成千上万的仪表 API 调用将遍布整个代码库。改变可观测性系统需要把所有这些工具去掉，用新系统提供的不同工具来代替。

替换仪表是一项重大的前期投资，即使只是为了尝试一个新的系统。更糟的是，大多数系统都太大了以致于在所有服务中同时更换所有仪表是不可行的。大多数系统需要逐步推出新的仪表设备，但这样的上线可能很难设计。

被特定解决方案的仪表所“困住”是非常令人沮丧的。在可观测性供应商开始努力提供自己的仪表的同时，用户也开始拒绝采用这种仪表。由于了解到重新安装仪表的工作量，许多用户强烈希望他们正在考虑的任何新的观测系统能与他们目前使用的仪表一起工作。

为了支持这一要求，许多可观测性系统试图与其他几个系统提供的仪表一起工作。但这种拼凑的方式降低了每个系统所摄取的数据的质量。从许多来源摄取数据意味着对输入的数据不再有明确的定义，当预期的数据不均衡且定义模糊时，分析工具就很难完成它们的工作。

5.3 针对开源软件的特定解决方案的仪表基本上是不可能的

从一个开放源码库作者的角度来看，特定解决方案的仪表化是一个悲剧。

来自开放源码软件库的遥测数据对于操作建立在它们之上的应用程序来说至关重要。最了解哪些数据对操作至关重要的人，以及操作者应该如何利用这些数据来补救问题的人，就是实际编写软件的开源库的开发者。

但是，库的作者却陷入了困境。正如我们将看到的，没有任何一个特定解决方案的仪表化 API，无论写得多么好，都无法作为开放源码库可以接受的选择。

5.4 如何挑选一个日志库?

假设你正在编写世界上最伟大的开源网络框架。在生产过程中，很多事情都会出错，你自然希望把错误、调试和性能信息传达给你的用户。你使用哪个日志库?

有很多体面的日志库。事实上，有很多，无论你选择哪个库，你都会有很多用户希望你选择一个不同的库。如果你的 Web 框架选择了一个日志库，而数据库客户端库选择了另一个，怎么办？如果这两个都不是用户想要使用的呢？如果他们选择了同一个库的不兼容的版本呢?

没有一个完美的方法可以将多个特定解决方案的日志库组合成一个连贯的系统。虽然日志足够简单，不同解决方案的大杂烩可能是可行的，但对于特定解决方案的指标和追踪来说，情况并非如此。

因此，开放源码软件库通常没有内置的日志、度量或追踪功能。取而代之的是，库提供了“可观测性钩子”，这需要用户编写和维护一堆适配器，将使用的库连接到他们的可观测性系统上。

作者们有大量的知识，他们想通信关于他们的系统应该如何运行的知识，但他们没有明确的方法去做。如果你问任何写过大量开源软件的人，他们会告诉你。这种情况是痛苦的！而且是不幸的！一些库的作者**确实**试图选择一个日志库，但却发现他们无意中为一些用户造成了版本冲突，同时迫使其他用户编写日志适配器来捕捉使用其**实际**日志库的数据。

但是对于大多数库来说，可观测性只是一个事后的想法。虽然库的作者经常编写大量的测试套件，但他们很少花时间去考虑运行时的可观测性。考虑到库有大量的测试工具，但可观测性工具为零，这种结果并不令人惊讶。

正如我们在接下来的几章中所看到的，现代可观测性的设计是为了使在可观测性管道中发挥作用的每个人的代理权最大化。但受益最大的是库的作者；对于特定解决方案的仪表，他们目前根本没有选择。

5.5 分解问题

我们可以通过设计一个可观测性系统来解决上面列出的所有问题，以明确地解决每个人的需求。在本章的其余部分，我们将把现代观测系统的设计分解为基本要求。这些要求将为第五章中描述的 OpenTelemetry 的结构提供动力。

5.6 要求：独立的仪表、遥测和分析

归根结底，计算机系统实际上就是人类系统。像可观测性这样的跨领域问题，几乎与每一个软件组件都有互动。同时，传输和处理遥测数据可能是一个大批量的活动，以至于一个大规模的观测系统会产生自己的操作问题。这意味着，许多不同的人，以不同的身份，需要与观测系统的不同方面交互。为了很好地服务于他们，这个系统必须确保每个参与其中的人都有他们所需要的代理权，以便快速和独立地执行任务。提供代理权是设计一个有效的观测系统的基本要求。

让我们首先确定与运行中的软件系统有关的每个角色的责任：库的作者、应用程序的所有者、操作者和响应者。

库的作者了解他们软件的情况

对于封装了关键功能的软件库，如网络和请求管理，库的作者也必须管理追踪系统的各个方面：注入、提取和上下文传播。

应用程序拥有者组织软件并管理依赖关系

应用程序所有者选择构成其应用程序的组件，并确保它们编译成一个连贯的、有功能的系统。应用程序所有者还编写应用程序级别的工具，它必须与库作者提供的指令（和上下文传播）进行正确的交互。

运维人员管理遥测的生产和传输

运维管理从应用到响应者的可观测性数据的传输。他们必须能够选择数据的格式以及数据的发送地点。当数据在产生时，他们必须操作传输系统：管理缓冲、处理和传送数据所需的所有资源。

响应者消费遥测数据并产生有用的见解

要做到这一点，应对者必须了解数据结构及其内在意义（结构和意义将在第三章中详细描述）。当新的和改进的分析工具出现时，反应者还需要将其添加到他们的工具箱中。

这些角色代表不同的决策点：

- 库的作者只能通过发布其代码的新版本来进行修改。
- 应用程序所有者只能通过部署其可执行文件的新版本来进行更改。
- 运维人员只能通过管理可执行文件的拓扑结构和配置来进行改变。
- 响应者只能根据他们收到的数据做出改变。

传统的三大支柱方法扰乱了所有这些角色。垂直整合的一个副作用是，几乎所有的数据变化都需要进行代码修改。几乎任何对可观测性系统的非微不足道的改变都需要应用程序所有者进行代码修改。要求其他人进行你所关心的改变，这样会有很大的阻力，并可能导致压力、冲突和不作为。

显然，一个设计良好的可观测性系统应该侧重于允许每个人尽可能多的代理和直接控制，它应该避免将开发者变成意外的看门人。

5.7 要求：零依赖性

应用程序是由依赖关系（网络框架、数据库客户端），加上依赖关系的依赖关系（OpenTelemetry 或其他仪表库），加上它们的依赖关系的依赖关系的依赖关系（无论这些仪表库依赖什么）组成。这些都被称为反式的依赖关系。

如果任何两个依赖关系之间有冲突，应用程序就无法运行。例如，两个库可能分别需要一个不同的（不兼容的）底层网络库的版本，如 gRPC。这可能会导致一些不好的情况。例如，一个新版本的库可能包括一个需要的安全补丁，但也包括一个升级的依赖关系，这就产生了依赖关系冲突。

诸如此类的过渡性依赖冲突给应用程序所有者带来了很大的麻烦，因为这些冲突无法独立解决。相反，应用程序所有者必须联系库的作者，要求他们提供一个解决方案，这最终需要时间（假设库的作者回应了这个请求）。

为了使现代可观测性发挥作用，库必须能够嵌入仪表，而不必担心当他们的库被用于组成应用程序时而导致问题。因此，可观测性系统必须提供不包含可能无意中引发横向依赖冲突的依赖性的仪表。

5.8 要求：严格的后向兼容和长期支持

当一个仪表化的 API 破坏了向后的兼容性，坏事就会发生。一个精心设计的应用程序最终可能会有成千上万的仪表调用站点。由于 API 的改变而不得不更新数以千计的调用站点是一个相当大的工作量。

这就产生了一种特别糟糕的依赖性冲突，即一个库中的仪表化不再与另一个库中的仪表化兼容。

因此，仪表化 API 必须在很长的时间范围内具有严格的向后兼容能力。理想的情况是，仪表化 API 一旦变得稳定，就永远不会破坏向后兼容。新的、实验性的 API 功能的开发方式必须保证它们的存在不会在包含稳定仪表的库之间产生冲突。

5.9 分离关注点是良好设计的基础

在下一章中，我们将深入研究 OpenTelemetry 的架构，看看它是如何满足上面提出的要求的。但在这之前，我想说的是一个重要的问题。

如果你分析这些需求，你可能会注意到一些奇特的现象：它们中几乎没有任何专门针对可观测性的内容。相反，重点是尽量减少依赖性，保持向后的兼容性，并确保不同的用户可以在没有无谓干扰的情况下发挥作用。

每一个要求都指出了关注点分离是一个关键的设计特征。但是这些特性并不是 OpenTelemetry 所独有的。任何寻求广泛采用的软件库都会很好地包括它们。在这个意义上，OpenTelemetry 的设计也可以作为设计一般的开源软件的指南。下次当你开始一个新的开放源码软件项目时，请预先考虑这些要求，相应地设计你的库及你的用户会感谢你的。

第 6 章

OpenTelemetry 架构概述

第 2 章描述了实现自动分析所需的数据模型，第 4 章描述了支持原生 开源仪表的额外要求，并赋予各角色（应用程序所有者、运维和响应者）自主权。这就是我们对现代可观测性的概念模型。

在本报告的其余部分，我们描述了这个新模型的一个事实实现，即 OpenTelemetry。这一章描述了构成 OpenTelemetry 遥测管道的所有组件。后面的章节将描述稳定性保证、建议的设置以及 OpenTelemetry 现实中的部署策略。有关该项目的更多细节可以在附录中找到。

6.1 信号

OpenTelemetry 规范被组织成不同类型的遥测，我们称之为**信号（signal）**。主要的信号是追踪。日志和度量是其他例子。信号是 OpenTelemetry 中最基本的设计单位。

每一个额外的信号首先是独立开发的，然后与追踪和其他相关信号整合。这种分离允许开发新的、实验性的信号，而不影响已经变得稳定的信号的兼容性保证。

OpenTelemetry 是一个**跨领域的关注点（cross-cutting concern）**，它在事务通过每个库和服务时追踪其执行。为了达到这个目的，所有的信号都建立在低级别的上下文传播系统之上，该系统为信号提供了一个地方来存储它们需要与当前正在执行的代码相关联的任何事务级数据。因为上下文传播系统与追踪系统是完全分开的，其他跨领域的问题也可以利用它。图 5-1 说明了这个分层结构。

6.2 上下文（Context）

上下文对象是一个与执行上下文相关联的键值存储，例如线程或循环程序。如何实现这一点取决于语言，但 OpenTelemetry 在每种语言中都提供一个上下文对象。

信号在上下文对象中存储它们的数据。因为 OpenTelemetry 的 API 调用总是可以访问整个上下文对象，所以信号有可能成为集成的，并在上下文共享数据，而不需要改变

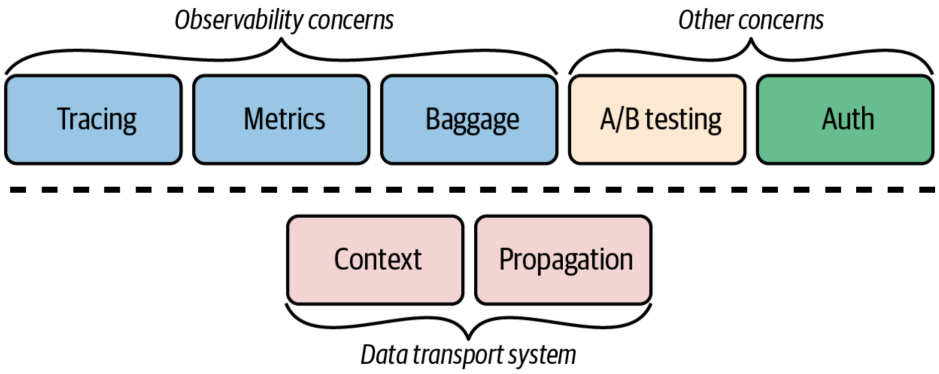

图 6-1: 所有 OpenTelemetry 信号都建立在一个共享的上下文传播系统之上。其他的，非可观测性的交叉关注也可以使用上下文传播机制来通过分布式系统传输他们的数据。

API。例如，如果追踪和度量信号都被启用，记录一个度量可以自动创建一个追踪范例。日志也是如此：如果有的话，日志会自动绑定到当前的追踪。

6.3 传播器（Propagator）

为了使分布式追踪发挥作用，追踪上下文必须被参与事务的每个服务所共享。传播器通过序列化和反序列化上下文对象来实现这一点，允许信号在网络工作请求中追踪其事务。

6.4 追踪（Tracing）

OpenTelemetry 追踪系统是基于 OpenTracing 和 OpenCensus。这两个系统，以及流行的 Zipkin 和 Jaeger 项目，都是基于谷歌开发的 Dapper 追踪系统。OpenTelemetry 试图与所有这些基于 Dapper 的系统兼容。

OpenTelemetry 追踪包括一个叫做 **链接（link）** 的概念，它允许单独的追踪被组合成一个更大的图。这被用来连接事务和后台处理，以及观察大型异步系统，如 Kafka 和 AMQP。

6.5 指标（Metric）

度量指标（metric）是一个很大的话题，包含各种各样的方法和实现。OpenTelemetry 度量信号被设计成与 Prometheus 和 StatsD 完全兼容。

指标包括追踪样本，自动将指标与产生它们的追踪样本联系起来。手工将指标和追踪联

系起来往往是一项繁琐且容易出错的任务，自动执行这项任务将为运维人员节省大量的时间。

6.6 日志（Log）

OpenTelemetry 结合了高度结构化的日志 API 和高速日志处理系统。现有的日志 API 可以连接到 OpenTelemetry，避免了对应用程序的重新测量。

每当它出现的时候，日志就会自动附加到当前的追踪中。这使得事务日志很容易找到，并允许自动分析，以找到同一追踪中的日志之间的准确关联。

6.7 Baggage

OpenTelemetry Baggage 是一个简单但通用的键值系统。一旦数据被添加为 Baggage（包袱）它就可以被所有下游服务访问。这允许有用的信息，如账户和项目 ID，在事务的后期变得可用，而不需要从数据库中重新获取它们。例如，一个使用项目 ID 作为索引的前端服务可以将其作为 Baggage 添加，允许后端服务也通过项目 ID 对其跨度和指标进行索引。

你可以将 Baggage 看做是一种**分布式文本的**形式。直接放入上下文对象的项目只能在当前服务中访问。与追踪上下文一样，作为 Baggage 添加的项目被作为 header 注入网络请求，允许下游服务提取它们。

与上下文对象一样，Baggage 本身不是一个可观测性工具。它更像是一个通用的数据存储和传输系统。除了可观测性之外，其他跨领域的工具，例如，功能标记、A/B 测试和认证，可以使用 Baggage 来存储他们需要追踪当前事务的任何状态。

然而，Baggage 是有代价的。因为每增加一个项目都必须被编码为一个头，每增加一个项目都会增加事务中每一个后续网络请求的大小。这就是为什么我们称它为 Baggage。我建议，Baggage 要少用，作为交叉关注的一部分。Baggage 不应该被用作明确定义的服务 API 的 “方便” 替代品，以明确地向下游应用程序发送参数。

6.8 OpenTelemetry 客户端架构

应用程序通过安装一系列的软件库来检测 OpenTelemetry：API、SDK（软件开发工具包）、SDK 插件和库检测。这套库被称为 OpenTelemetry 客户端。图 5-2 显示了这些组件之间的关系。

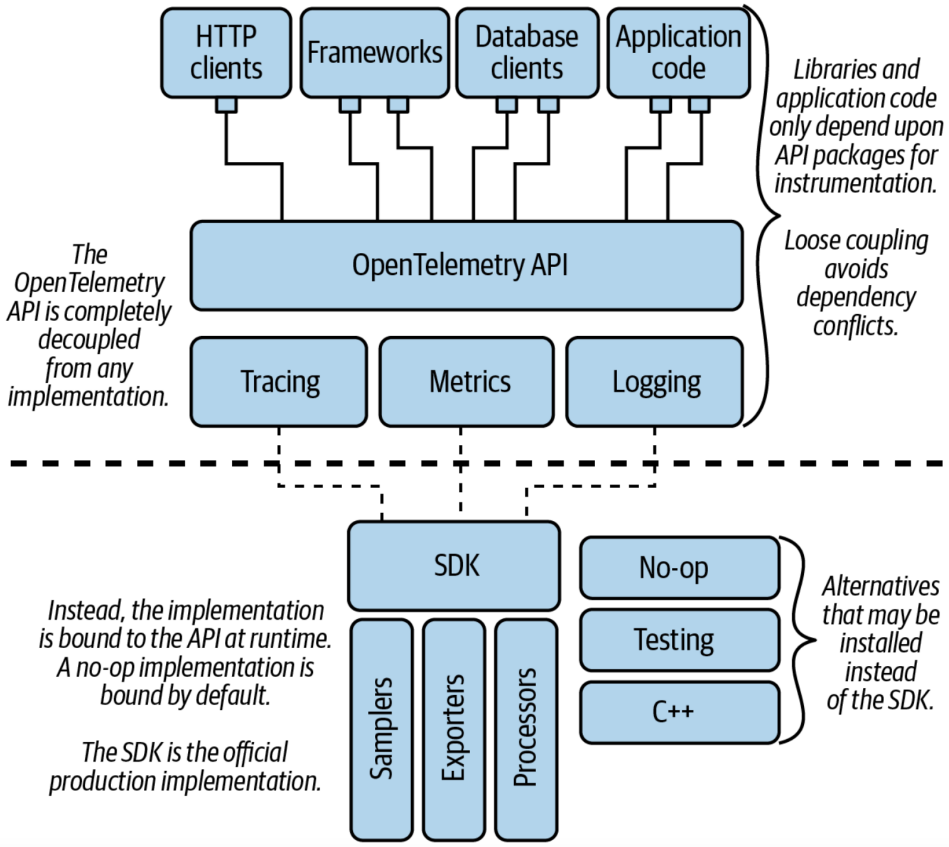

图 6-2: OpenTelemetry 客户端架构。为了帮助管理依赖性，OpenTelemetry 将实现与仪表使用的 API 分开。

在许多语言中，OpenTelemetry 提供安装程序，这有助于自动安装和设置 OpenTelemetry 客户端。然而，可用的自动化程度取决于语言。在 Java 中，OpenTelemetry 提供了一个 Java 代理，它通过动态地注入所有必要的组件来实现安装的完全自动化。在 Go 中，OpenTelemetry 包必须通过编写代码来安装和初始化，就像任何其他 Go 包一样。Python、Ruby 和 NodeJS 介于两者之间，提供不同程度的自动化。

在学习 OpenTelemetry 时，了解在你使用的语言中如何设置是很重要的。特别是，一定要学习如何安装仪表，因为不同的语言有很大的不同。

请查看客户端文档，了解更多的入门细节。

6.9 客户端架构：仪表 API

OpenTelemetry API 是指用于编写仪表的一组组件。该 API 被设计成可以直接嵌入到开放源码软件库以及应用程序中。这是 OpenTelemetry 的唯一部分，共享库和应用逻

辑应该直接依赖它。

6.10 提供者（Provider）

API 与任何实现完全分开。当一个应用程序启动时，可以通过为每个信号注册一个提供者来加载一个实现。提供者成为所有 API 调用的接收者。

当没有加载提供者时，API 默认为无操作提供者。这使得 OpenTelemetry 仪表化可以安全地包含在共享库中。如果应用程序不使用 OpenTelemetry，API 调用就会变成 no-ops，不会产生任何开销。

对于生产使用，我们建议使用官方的 OpenTelemetry 提供商，我们称之为 **OpenTelemetry SDK**。

> **为什么有多个实现方案?**
>
> API 和实现的分离有很多好处。但是，如果用户被迫总是安装官方的 OpenTelemetry SDK，这又有什么意义？是否有必要安装另一个实现？SDK 已经具有很强的扩展性。
>
> 我们相信是有的。虽然我们希望 OpenTelemetry 仪表是通用的，但建立一个对所有用例都理想的单一实现是不可能的。尽管我们相信 OpenTelemetry SDK 很好，但也应该有一个选择，那就是使用另一种实现。实现的灵活性是提供通用仪表 API 的一个关键特征。
>
> 首先，这种分离保证了 OpenTelemetry 不会产生无法克服的依赖冲突。我们总是可以选择加载一个包括不同依赖链的实现。
>
> 另一个原因是性能。OpenTelemetry SDK 是一个可扩展的、通用的框架。虽然 SDK 的设计是为了尽可能地提高性能，但扩展性和性能总是要权衡一下的。例如，通过外来函数接口创建与 OpenTelemetry C++ SDK 的绑定，有可能成为 Ruby、Python 和 Node.js 等动态脚本语言的一个非常有效的选择。

还有一些流媒体架构显示了有希望的性能提升。在许多这样的优化解决方案中，编写插件和生命周期钩子的能力将受到严重限制；支持这些类型的功能所需的数据结构在这些优化解决方案中是不存在的。归根结底，没有 “完美的实现”；只有权衡。

API/SDK 的分离是一个关键的设计选择，该项目大量使用了这一点。例如，除了 SDK 之外，每一种语言都有一个 no-op 的实现，它是默认安装的。还有一个 Fake/Mock 实

现，我们用它来测试。而且，还有可能实现更多创造性的实现。例如，为分布式系统建立开发者工具，如一个实时调试器，它可以跨越网络边界工作。

6.11 客户端架构：SDK

OpenTelemetry 项目为 OpenTelemetry API 提供了一个官方实现，我们称之为 OpenTelemetry SDK。该 SDK 通过提供一个插件框架来实现 OpenTelemetry API。下面将介绍追踪 SDK；类似的架构也适用于度量和日志。

基本数据结构是一个无锁的 SpanData 对象。当用户开始一个跨度时，SpanData 对象被创建，当用户添加属性和事件时，它被自动建立起来。一旦一个跨度结束，SpanData 对象将不再被更新，可以安全地传递给后台线程。

SDK 的插件架构被组织成一个流水线。对于追踪来说，该管道由一连串的 SpanProcessors 组成。每个处理器对 SpanData 对象进行两次同步访问：一次是在跨度开始时，另一次是在跨度结束后。采样器、日志附加器和数据清洗器是 SpanProcessors 的例子。链中的最后一个处理器通常是一个 BatchSpanProcessor，它管理着一个已完成的跨度的缓冲区。输出器可以连接到 BatchSpanProcessor，通过网络将成批的跨度传递到遥测管道中的下一个服务，通常将它们发送到收集器或直接发送到追踪后端。一旦跨度被导出，管道就完成了，SpanData 对象也被释放。

6.12 采样器（Sampler）

OpenTelemetry 提供了几种常见的采样算法，包括前期采样和基于优先级的采样。采样可以帮助控制成本，但它是有代价的：你将会错过数据。在启用任何种类的采样算法之前，重要的是要检查你计划使用的分析工具支持哪些类型的采用。意外的采样可能会破坏某些形式的分析。一些工具需要他们自己的采样插件。例如，AWS X-Ray 使用它自己的采样算法，它可以作为 AWS 特定的采样插件使用。

6.13 导出器（Exporter）

OpenTelemetry 为 OTLP（OpenTelemetry Protocol）、Jaeger、Zipkin、Prometheus 和 StatsD 提供导出器。由第三方维护的其他导出器可以在每种语言的 OpenTelemetry-Contrib 资源库中找到。使用 OpenTelemetry 注册表来了解目前有哪些插件可用。

6.14 客户端架构：库仪表化

为了正常工作，OpenTelemetry 需要端到端的工具。这不是可有可无的：如果关键的库不包括仪表，上下文传播将被破坏。

一般来说，必须检测的库包括 HTTP 客户端、HTTP 服务器、应用框架、消息传递 / 队列系统和数据库客户端。这些库经常在上下文传播中起作用。

- HTTP 客户端必须创建一个**客户端 span** 来记录请求。客户端还必须使用一个传播器，将当前的上下文作为一组 HTTP 头信息注入到请求中。
- HTTP 服务器（应用框架）必须使用一个传播器来从 HTTP 头信息中提取上下文。提取的上下文被用来创建一个**服务器跨度**，该跨度被设置为当前活动的跨度，它封装了所有的应用程序代码。
- 同样，消息 / 队列系统中的发送者必须使用传播器将上下文注入消息中，这样就可以通过在接收者身上提取上下文来继续追踪。
- 数据库客户必须创建一个**数据库跨度**来记录数据基础事务。一旦数据库服务器也使用 OpenTelemetry 工具，数据库客户端也必须将上下文注入数据库请求中。

这一要求是我们希望看到开放源码软件库能带有本地仪表的主要原因之一。同时，仪表插件是由 OpenTelemetry 项目或第三方提供的 **contrib 包**。

6.15 收集器（Collector）

除了上述的客户端外，OpenTelemetry 还提供了一个独立的服务，称为收集器。收集器是一个灵活、可配置的遥测处理系统。其基本结构如图 5-3 所示。

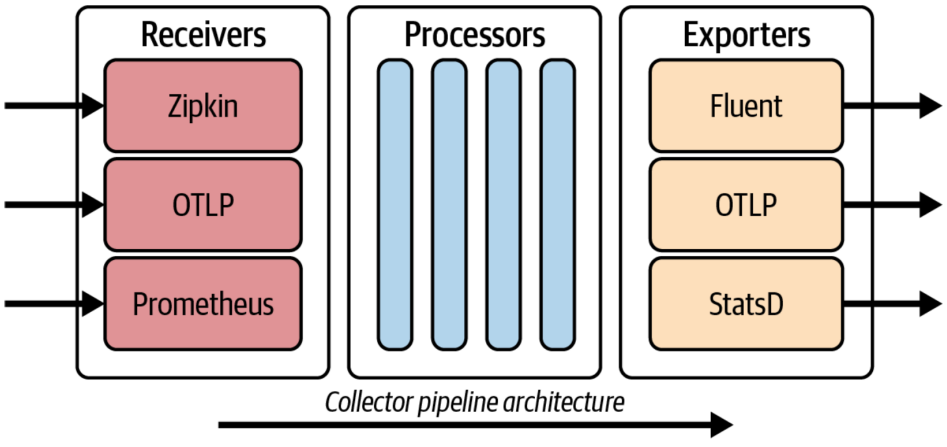

图 6-3: 收集器有一个可配置的处理管道，可以导入和导出许多常见格式的数据。

收集器管道可以提供以下服务：

- 配置，如路由和数据导出格式。OpenTelemetry 客户端可用的几乎所有配置选项都可以在收集器中进行管理。
- 数据处理，如刷新、格式转换和向多个目的地发送。
- 缓冲，帮助管理网络。
- 机器级环境的资源检测。可以发现主机、Kubernetes 和云提供商的细节，并将其附加到收集器收到的所有数据上。
- 收集主机指标，如 RAM、CPU 和存储容量。

运维人员可以使用收集器来管理与可观测系统相关的所有部署细节，而不需要与应用程序本身进行交互。由于大多数配置选项是特定于部署的，而且是由运维而不是应用程序开发人员管理的，因此，将遥测配置从应用程序转移到收集器，可以干净地分离关注点。

如果所有的路由和数据处理任务都转移到收集器上，OpenTelemetry SDK 就可以以更简单的配置运行。默认情况下，SDK 将发送未处理的 OTLP 数据到一个预定的本地端口，在那里它可以由本地收集器接收。

6.16 收集器架构：接收器（Receiver）

收集器可以被配置为从各种来源接收各种格式的遥测数据。目前，收集器支持超过四十种不同类型的接收器！一旦接收到，所有这些数据都会被转换为 OTLP。OpenTelemetry 同时支持基于推和拉的接收器。

6.17 收集器架构：处理器（Processor）

一旦接收器将遥测数据转换为 OTLP，就会有各种处理器可用。处理器可以被配置为执行各种任务。

- 清洗数据以删除敏感数据，如 PII（个人身份信息）。
- 数据规范化，例如将数据源的旧版本转换为与当前后台使用的仪表盘和查询相匹配的版本。
- 根据某些属性将数据路由到特定的后端。例如，将与欧盟用户有关的数据存储在欧盟境内托管的存储系统上。

- 基于尾部的采样，以帮助确保错误和异常值更有可能被捕获，同时对嘈杂和无趣的信息进行速率限制。

6.18 收集器架构：导出器（Exporter）

一旦遥测数据被处理，它可以被输出到各种后端。在未来，我们希望越来越多的后端能够原生支持 OTLP。同时，OTLP 可以被转换为目前流行的系统所支持的许多格式。请查看 OpenTelemetry 供应商页面，找到目前支持 OpenTelemetry 的商业供应商列表。

除了将遥测数据转换为单一格式外，还可以安装多个导出器。遥测数据可以按类型分开，并发送到不同的后端。例如，将追踪数据发送到 Jaeger，将度量数据发送到 Prometheus。

重复的遥测数据也可以被同步发送到多个后端。这使得运维人员可以从一个后端无缝切换到另一个后端，而不会有任何服务上的损失。它还允许特殊的分析工具与通用的观测平台一起接收数据。

6.19 收集器架构：管道（Pipeline）

收集器允许接收器、处理器和导出器组合成复杂的管道（pipeline），可以同时运行。管道是通过 YAML 配置文件设计和管理的。

这种配置语言是非常强大的。通过在每台机器上部署一个本地收集器，并将它们连接到配置为执行专门处理任务的几层收集器部署上，收集器可以用来开发一个大规模的、强大的遥测系统。

第 7 章

稳定和长期支持

OpenTelemetry 被设计成允许长期稳定性和不确定性并存的局面。在 OpenTelemetry 中，稳定性保证是在每个信号的基础上提供的。与其看版本号，不如检查你想使用的信号的稳定性等级。

7.1 信号生命周期

图 6-1 显示了新信号是如何被添加到 OpenTelemetry 的。**实验性**信号仍在开发中。它们可能在任何时候改变并破坏兼容性。实验性信号的开发是以规范提案的形式开始的，它是与一组原型一起开发的。一旦实验性信号准备好在生产中使用，信号的特性就会被冻结，新信号的测试版就会以多种语言创建。测试版可能不是完整的功能，它们可能会有一些突破性的变化，但它们被认为是为早期采用者的产品反馈做好准备。一旦一个信号被认为可以被宣布为**稳定**版本，就会发布一个候选版本。如果候选版本能够在一段时间内保持稳定，没有问题，那么该信号的规范和测试版都被宣布为稳定。

一旦一个信号变得稳定，它就属于 OpenTelemetry 的长期支持保障范围。
OpenTelemetry 非常重视向后兼容和无缝升级。详情见以下章节。

如果 OpenTelemetry 信号的某个组件需要退役，该组件将被标记为**废弃的**。被废弃的组件不再获得新的功能，但它们仍然被 OpenTelemetry 的长期支持保证所覆盖。如果可能的话，该组件将永远不会被删除，并将继续发挥作用。如果一个组件必须被删除，将提前宣布删除日期。

实验性的功能总是与稳定性的功能保持在不同的包中，稳定的功能永远不能引用实验性的功能。这确保了新的开发不会影响现有特性的稳定性。只要库只依赖于稳定的特性，它们就不会经历破坏性的 API 变化。

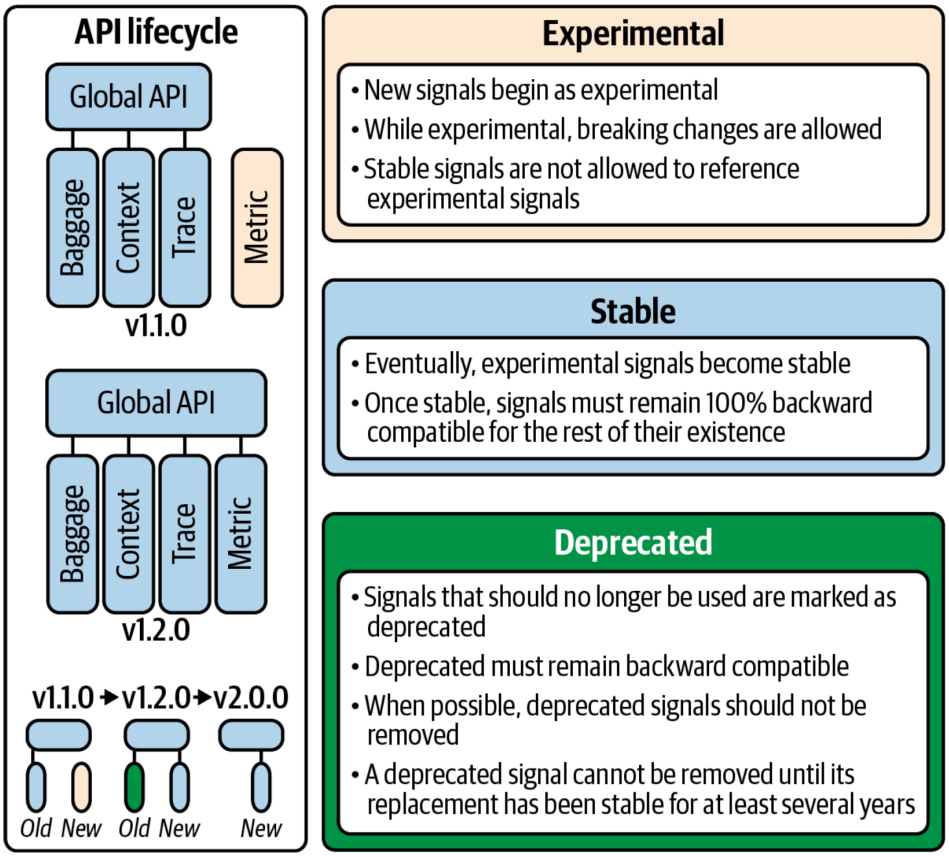

图 7-1: 图 6-1：OpenTelemetry 中的每个主要功能都被赋予了一个稳定性等级，并遵循相同的生命周期。

7.2 API 的稳定性

OpenTelemetry API 预计将被数以千计的库所依赖，有数以百万计的调用站点。因此，API 的稳定部分决不能破坏向后的兼容性。应用程序的所有者和库的开发者不应该为了升级到一个新版本的 API 而重新测量他们的应用程序。

如果一个 OpenTelemetry API 被废弃（这不太可能），被废弃的 API 仍将保持稳定并发挥功能。

> **原生工具是稳定的工具**
>
> 携带原生 OpenTelemetry 仪表的 OSS 库应该只使用稳定的 API，因为实验性功能的改变可能会造成依赖性冲突。
>
> 也就是说，我们鼓励进行测试。如果一个库愿意为实验性的 OpenTelemetry 功能提供支持，这是一个给项目提供反馈和参与新功能设计的好方法。然而，我们

> 建议将与实验性 OpenTelemetry 功能的集成作为可选的插件提供，终端用户必须单独安装才能启用。
>
> 一旦功能变得稳定，就没有必要把它们作为一个单独的插件。事实上，最好是将 OpenTelemetry 原生集成，因为用户可能会忘记安装插件。这样一来，如果应用程序所有者安装了 OpenTelemetry SDK，他们就会自动开始接收来自每个库的数据。如果没有安装 SDK，API 的调用就没有意义了。原生仪表是我们希望 OpenTelemetry 能够简化应用程序所有者的观察能力的一种方式 —— 它已经存在于每一个库中，只要它需要，就可以随时使用。

7.3 SDK 和收集器的稳定性

SDK 的稳定性集中在两个方面：插件接口和资源使用。SDK 可能偶尔会废止一个插件接口。为了确保应用程序的所有者能够干净利落地进行升级，必须在废弃的接口被删除之前添加一个替代接口，而使用废弃接口的流行插件必须被迁移到新的接口上。通过以这种方式安全地迁移插件生态系统，可以避免应用程序所有者陷入这样的境地：他们想要升级，但却被一个无法使用的插件所阻挡。在废弃和移除一个插件接口之间必须有至少 6 个月的时间，而且废弃的接口只有在维护它们会造成性能问题时才会被移除。否则，我们将无限期地保留这些被废弃的接口。

说到性能，OpenTelemetry SDK 的稳定部分必须避免性能倒退，以确保 SDK 的较新版本在升级时不会引起资源争夺。很明显，启用新版本中增加的功能可能需要额外的资源。但是，简单地升级 SDK 不应该导致性能退步。

与 SDK 一样，收集器试图避免性能退步，并为收集器插件生态系统提供一个渐进的升级路径。

7.4 升级 OpenTelemetry 客户端

在运行 OpenTelemetry 时，我们希望用户能保持最新的 SDK 版本。有两个事件可能会迫使用户升级 SDK：一个库将其仪表升级到新版本的 API，或者 OpenTelemetry 发布一个重要的安全补丁。

上面列出的稳定性保证确保了这种升级路径始终是可行的。只要一个应用程序只依赖于稳定的信号，升级应该只涉及依赖性的提升。教程不需要重写，插件也不会突然变得不受支持。

OpenTelemetry 致力于向后兼容的一个很好的例子是它对其前身 OpenTracing 的支

持。OpenTelemetry 追踪信号与 OpenTracing API 完全兼容，OpenTelemetry 和 OpenTracing API 调用可以混合到同一个应用程序中。OpenTracing 用户可以升级到 OpenTelemetry 而不需要重写现有的仪表。

第 8 章

建议的设置和遥测管道

现在我们了解了组成 OpenTelemetry 的各个构件，我们应该如何将它们组合成一个强大的生产管道?

答案取决于你的出发点是什么。OpenTelemetry 是模块化的，设计成可以在各种不同的规模下工作。你只需要使用相关的部分。这就是说，我们已经创建了一个建议的路线图供你遵循。

8.1 安装 OpenTelemetry 客户端

可以单独使用 OpenTelemetry 客户端而不部署收集器。这种基本设置通常是绿地部署的充分起点，无论是测试还是初始生产。OpenTelemetry SDK 可以被配置为直接向大多数可观测性服务传输遥测数据。

8.2 挑选一个导出器

默认情况下，OpenTelemetry 使用 OTLP 导出数据。该 SDK 提供了几种常见格式的导出器。Zipkin、Prometheus、StatsD 等。如果你使用的可观测性后端没有原生支持 OTLP，那么这些其他格式中的一种很可能会被支持。安装正确的导出器并将数据直接发送到你的后端系统。

8.3 安装库仪表

除了 SDK，OpenTelemetry 仪表必须安装在所有 HTTP 客户端、Web 框架、数据库和应用程序的消息队列中。如果这些库中有一个缺少仪表，上下文传播就会中断，导致不完整的追踪和混乱的数据。

在某些语言中，如 Java，仪表可以自动安装，这就更容易了。请确保了解 OpenTelemetry 如何在你使用的编程语言中管理仪表，并仔细检查仪表是否正确安装

在你的应用程序中。

8.4 选择传播器

仔细检查你的系统需要哪些传播器也很重要。默认情况下，OpenTelemetry 使用 W3C 的追踪上下文和 Baggage 传播器。然而，如果你的应用程序需要与使用不同的追踪传播器的服务进行通信，如 Zipkin 的 B3 或 AWS 的 X-Amzn，那么改变 `OTEL_PROPAGATORS` 配置以包括这个额外的传播器。

如果 OpenTelemetry 最终要取代这些其他的追踪系统，我建议同时运行 trace-context 和额外的追踪传播器。这将使你在部署中逐步取代旧系统时，能够无缝地过渡到 W3C 标准。

8.5 部署本地收集器

虽然有些系统有可能只使用客户端，但通过在你的应用程序所运行的机器上添加一个本地收集器，可以改善你的操作体验。

运行一个本地收集器有许多好处，如图 7-1 所示。收集器可以生成机器指标（CPU、RAM 等），这是遥测的一个重要部分。收集器还可以完成任何需要的数据处理任务，如从追踪和日志数据中清除 PII。

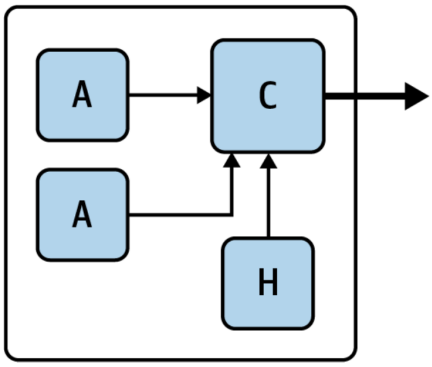

图 8-1: 一个本地收集器（C）从本地应用程序（A）接收遥测数据，同时收集主机指标（H）。收集器将合并的遥测数据输出到管道的下一个阶段。

运行收集器后就可以将大多数遥测配置从你的应用程序中移出。遥测配置通常是特定的部署，而不是特定的应用。SDK 可以简单地设置为使用默认配置，总是将 OTLP 数据导出到预定义的本地端口。通过管理本地收集器，运维可以在不需要与应用程序开发人员协调或重新启动应用程序的情况下进行配置更改。在通过复杂的 CI/CD（持续集成 / 持

续交付）管道移动应用程序时，这尤其有帮助，因为在不同的暂存和负载测试环境中，遥测需要不同的处理方式。

快速发送遥测数据到本地收集器，可以作为一个缓冲器来处理负载，并确保在应用程序崩溃时，缓冲的遥测数据不会丢失。

8.6 部署收集器处理器池

如果你的本地收集器开始执行大量的缓冲和数据处理，它就会从你的应用程序中窃取资源。这可以通过部署一个只运行收集器的机器池来解决，这些机器位于负载均衡器后面，如图 7-2 所示。现在可以根据数据吞吐量来管理收集器池的大小。

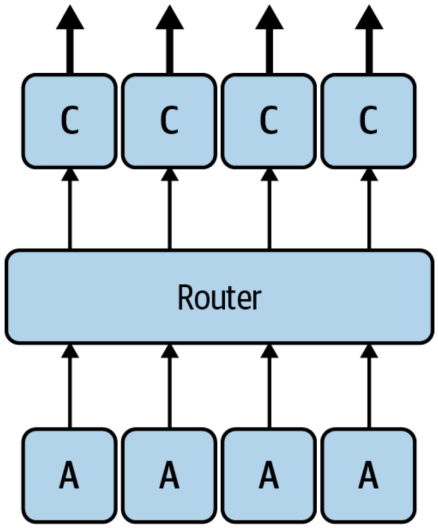

图 8-2: 应用程序 (A) 可以通过使用路由器或负载均衡器向收集器 (C) 池发送遥测信息。

本地收集器现在可以关闭其处理器以释放资源。它们继续收集机器级遥测数据，作为来自本地应用程序的 OTLP 的转发机制。

8.7 添加额外的处理池

有时，单个收集器池是不够的。一些任务可能需要以不同的速度扩展。将收集器池分割成一个更专门的池的管道，可能允许更有效和可管理的扩展策略，因为每个专门的收集器池的工作负载变得更可预测。

一旦你达到了这个规模，就没有什么部署的问题了。大规模系统的专门需求往往是独特的，这些需求将驱动你的可观测性管道的拓扑结构。利用收集器提供的灵活性，根据你的需求来定制每一件事情。我建议对每个收集器配置的资源消耗进行基准测试，并使用

这些信息来创建弹性的、自动扩展的收集器池。

8.8 用收集器管理现有的遥测数据

上面描述的路线图适用于上线 OpenTelemetry。但你应该如何处理现有的遥测？大多数运行中的系统已经有了某种形式的指标、日志和（可能有）追踪。而大型的、长期运行的系统往往最终会有多个遥测解决方案的补丁。不同的组件可能是在不同的时代建立的，有些组件可能是从外部继承的，比如收购。从可观测性的角度来看，这可能会导致混乱的局面。

即使在这样复杂的遗留情况下，仍然有可能过渡到 OpenTelemetry，而不需要停机或一次重写所有的服务。秘诀是首先部署一个收集器，作为一个透明的代理。

在收集器中，为接收你的系统目前产生的每一种类型的遥测设置接收器，并与以完全相同的格式发送遥测的导出器相连。一个 StatsD 接收器连接到一个 StatsD 导出器，一个 Zipkin 接收器连接到一个 Zipkin 导出器，以此类推。这种透明的代理可以逐步推出，而不会造成干扰。一旦所有的远程测量都由这些收集器来调解，就可以引入额外的处理。甚至在你把你的仪表切换到 OpenTelemetry 之前，你可能会发现这些收集器是管理和组织你当前拼凑的遥测系统的一个有用的方法。图 7-3 显示了一个收集器处理来自各种来源的数据。

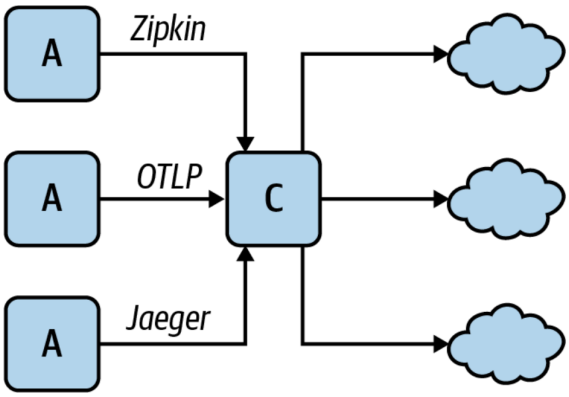

图 8-3: 收集器可以帮助管理复杂的、拼凑的可观测性系统，这些系统以各种格式向各种存储系统发送数据。

为了开始将服务切换到 OpenTelemetry，可以在收集器上添加一个 OTLP 接收器，与现有的导出器相连。随着服务转向使用 OpenTelemetry 客户端，它们将 OTLP 发送到收集器，收集器将把 OTLP 翻译成这些系统以前产生的相同数据。这使得 OpenTelemetry 可以由不同的应用团队逐步上线，而不会出现中断。

8.9 转移供应商

一旦所有的遥测流量都通过收集器发送，切换到一个新的可观测性后端就变得很容易了：只需在收集器中添加一个导出器，将数据发送到你想尝试的新系统，并将遥测数据同时发送到旧系统和新系统。通过向两个系统发送数据，你创造了一个重叠的覆盖范围。如果你喜欢新系统，你可以在一段时间后让旧系统退役，以避免在可视性方面产生差距。图 7-4 说明了这个过程。

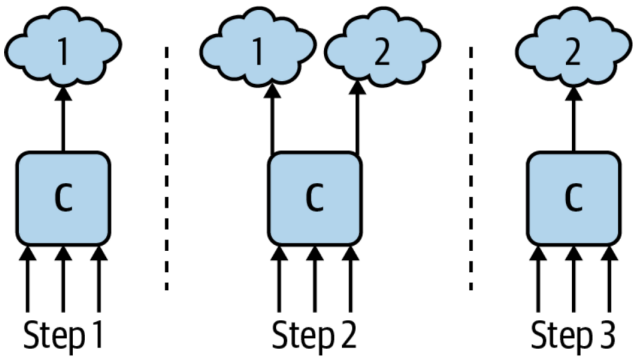

图 8-4: 使用收集器在可观测性后端之间迁移而不中断服务。

也可以使用 OpenTelemetry 在多个供应商之间进行测验。你可以同时向多个系统发送遥测信息，并直接比较系统，看哪一个最适合你的需要。

第 9 章

如何在组织中推广 OpenTelemetry

上线一个新的遥测系统可能是一项复杂的工作。它需要整个工程组织的支持，不能一蹴而就。不要低估这可能会产生的问题！

在大型组织中，通常有许多服务团队负责系统的不同部分。通常情况下，每个团队都需要付出一定的努力来使他们所管理的服务得到充分的工具化。而这些团队都有自己积压的工作，他们当然希望能够优先处理这些工作。

不幸的是，可观测性计划在开始提供价值和证明其价值之前就会耗尽人的耐心。但通过仔细的计划和协调，这种情况是可以避免的。

9.1 主要目标

在推广 OpenTelemetry 时，重要的是要记住，任何基于分布式追踪的可观测性系统都需要对参与事务的每个服务进行检测，以提供最大价值。如果只有部分服务被检测到，那么追踪就会被分割成小的、不相连的部分。

这种散乱的仪表的结果是不可取的。这种情况 —— 不一致的仪表和断裂的追踪是你想要避免的主要事情。如果追踪是断开的，运维人员仍然需要在他们的头脑中把所有的东西拼凑起来，以获得他们系统的情况。更糟糕的是，自动分析工具可以使用的数据非常有限。与人类运维人员不同，他们可以运用直觉，跳出框框来思考问题，而分析工具却只能使用他们得到的数据。由于数据有限，他们提供有用的见解的机会也将是有限的。

为了避免这个陷阱，集中精力对一个工作流程进行检测，在进入下一个工作流程之前对其进行完整的追踪。大多数工作流程并不涉及大系统的每一个部分，所以这种方法将最大限度地减少分析开始和价值实现之前所需的工作量。

9.2 选择一个高价值的目标

谈到实现价值，在开始进行仪表测量工作时，重要的是要有一个有吸引力的目标！这一点很重要。

最有可能的是，有一个特别的问题促使人们去部署 OpenTelemetry。如果是这样的话，就把重点放在解决该问题所需的最小的推广上。否则，想一想那些众所周知的问题，解决它们就值得公布了。

目前有哪些痛苦的、长期的问题在困扰着运维？减少系统延迟在哪里可以直接转化为商业价值？当人们问 “为什么这么慢?” 时，他们说的是系统的哪一部分？在选择第一个工作流程时，请以这些信息为指导。

识别一个有吸引力的目标有两个好处。首先，它可以更有利于说服众人，因为有一个具体的理由来做这项工作。这使得它更容易说服有关团队优先考虑增加指导，并以协调的方式进行。

第二，速战速决给你的新的可观测性系统一个闪亮的机会。第一次通过分布式追踪的视角来分析一个软件系统时，几乎总是会产生有用的见解。证明可观测性的价值可以引起很多人的兴趣，并有助于降低采用时任何挥之不去的障碍。

如果直接进入生产是困难的，“生产支持” 系统也是一个好的开始。可以对 CI/CD 系统进行检测，以帮助了解构建和部署的性能。在这里，整个组织都会感受到性能的大幅提升，并可以为将 OpenTelemetry 转移到生产中提供良好的理由。

9.3 集中遥测管理

展开和管理遥测系统从集中化中获益良多。在一些组织中，会有一个平台或信息结构团队可以接触到每一项服务。像这样的团队是集中管理遥测的一个好地方，这可以提供巨大的帮助。遥测管道最好被认为是它自己的系统；允许一个团队操作整个遥测管道，往往比要求许多团队各自拥有系统的一部分要好。

在软件层面，将 OpenTelemetry 设置与已经广泛部署的代码管理工具 —— 例如共享的启动脚本和应用框架整合起来，减少了每个团队需要管理的代码量。这有助于确保服务与最新的版本和配置保持同步，并使采用更加容易。

另一个关键工具是一个集中的知识库。OpenTelemetry 有文档，但它是通用的。创建特定于在你的组织内部署、管理和使用 OpenTelemetry 的文档。大多数工程师都是第一次接触 OpenTelemetry 和分布式追踪，这对他们的帮助怎么强调都不为过。

9.4 先广度后深度

还有一个关于合理分配精力的说明。当对一个工作流程进行端对端检测时，通常只需安装 OpenTelemetry 附带的仪表，加上你的组织所使用的任何内部或自创框架的仪表即

可。没有必要深入检测应用程序代码，至少在开始时没有必要。OpenTelemetry 附带的标准跨度和指标足以让你识别大多数问题；必要时可以有选择地增加更深层次的检测。在添加这种细节之前，请确保你已经建立并运行了端到端的追踪。

这就是说，有一个快速的方法可以为你的追踪增加很多细节，那就是把任何现有的日志转换成追踪事件。这可以通过创建一个简单的日志附加器来实现，它可以抓取当前的跨度，并将日志作为一个事件附加到它上面。现在，当你查找追踪时，你所有的应用日志都可以得到，这比在传统的日志工具中寻找它们要容易得多。OpenTelemetry 确实为一些常见的日志系统提供了日志附加程序，但它们也很容易编写。

9.5 与管理层合作

如果你是一个工程师在读这篇文章，我有一个补充说明。推广一个新的遥测系统可能需要组织很多人。幸运的是，有些人已经在做这种组织工作了 —— 经理们！这就是我们的工作。

但是，如果你想启动其中的一个项目，说服工程或项目经理来帮助你是很好的第一步。他们将对如何完成项目有宝贵的见解，并能在你可能不参加的会议上推销该项目。有时候，组织人比组织代码更难，所以不要害怕寻求帮助！

9.6 加入社区

最后，在个人和组织层面上，考虑加入 OpenTelemetry 社区！维护者和项目负责人都很友好，非常平易近人。社区是一个很好的获取援助和专业知识的资源的地方；我们总是很乐意帮助新的用户得到指导。还有一种汗水文化：如果有你想看到的 OpenTelemetry 功能，加入一个工作组并提供帮助是使它们得到优先考虑的一个好办法。

至少，一定要给我们反馈。我们从用户那里听到的越多，我们就越能专注于最重要的问题。我们的目标是建立一个推动下一代可观测性的标准。没有你，我们做不到，我们的大门永远是敞开的。

9.7 谢谢你的阅读

我希望你喜欢这份关于 OpenTelemetry 的可观测性的未来的报告。如果你有任何问题、评论、反馈或基于你所读的内容的灵感，请随时在 Twitter 上与我联系，我是 @tedsuo。

要想获得 OpenTelemetry 的帮助，请加入云原生计算基金会（CNCF）Slack 上的 #OpenTelemetry 频道。我希望能在那里见到你！

第 10 章

附录 A：OpenTelemetry 项目组织

OpenTelemetry 是一个大型项目。OpenTelemetry 项目的工作被划分为**特殊兴趣小组**（SIG）。虽然所有的项目决策最终都是通过 GitHub issue 和 pull request 做出的，但 SIG 成员经常通过 CNCF 的官方 Slack 保持联系，而且大多数 SIG 每周都会在 Zoom 上会面一次。

任何人都可以加入一个 SIG。要想了解更多关于当前 SIG、项目成员和项目章程的细节，请查看 GitHub 上的 OpenTelemetry 社区档案库。

10.1 规范

OpenTelemetry 是一个规范驱动的项目。OpenTelemetry 技术委员会负责维护该规范，并通过管理规范的 backlog 来指导项目的发展。

小的改动可以以 GitHub issue 的方式提出，随后的 pull request 直接提交给规范。但是，对规范的重大修改是通过名为 OpenTelemetry Enhancement Proposals（OTEPs）的征求意见程序进行的。

任何人都可以提交 OTEP。OTEP 由技术委员会指定的具体审批人进行审查，这些审批人根据其专业领域进行分组。OTEP 至少需要四个方面的批准才能被接受。在进行任何批准之前，通常需要详细的设计，以及至少两种语言的原型。我们希望 OTEP 的作者能够认真对待其他社区成员的要求和关注。我们的目标是确保 OpenTelemetry 适合尽可能多的受众的需求。

一旦被接受，将根据 OTEP 起草规范变更。由于大多数问题已经在 OTEP 过程中得到了解决，因此规范变更只需要两次批准。

10.2 项目治理

管理 OpenTelemetry 项目如何运作的规则和组织结构由 OpenTelemetry 治理委员会定义和维护，其成员经选举产生，任期两年。

治理成员应以个人身份参与，而不是公司代表。但是，为同一雇主工作的委员会成员的数量有一个上限。如果因为委员会成员换了工作而超过了这个上限，委员会成员必须辞职，直到雇主代表的人数降到这个上限以下。

10.3 发行版

OpenTelemetry 有一个基于插件的架构，因为有些观察能力系统需要一套插件和配置才能正常运行。

发行版（**distros**）被定义为广泛使用的 OpenTelemetry 插件的集合，加上一组脚本或辅助功能，可能使 OpenTelemetry 与特定的后端连接更简单，或在特定环境中运行 OpenTelemetry。

需要澄清的是，如果一个可观测性系统声称它与 OpenTelemetry 兼容，那么它应该总是可以使用 OpenTelemetry 而不需要使用某个特定的发行版。如果一个项目没有经过规范过程就扩展了 OpenTelemetry 的核心功能，或者包括任何导致它与上游 OpenTelemetry 仓库不兼容的变化，那么这个项目就是一个分叉，而不是一个发行版。

10.4 注册表

为了便于发现目前有哪些语言、插件和说明，OpenTelemetry 提供了一个注册表。任何人都可以向 OpenTelemetry 注册表提交插件；在 OpenTelemetry 的 GitHub 组织内托管插件不是必须的。

第 11 章

附录 B：OpenTelemetry 项目路线图

路线图很快就会过时。有关最新的路线图，请参见 OpenTelemetry 状态页面。也就是说，以下是截至目前 OpenTelemetry 的状态。

11.1 核心组件

目前，OpenTelemetry 追踪信号已被宣布为稳定的，并且在许多语言中都有稳定的实现。

度量信号刚刚被宣布稳定，测试版的实施将在 2022 年第一季度广泛使用。

日志信号预计将在 2022 年第一季度宣布稳定，测试版实现预计将在 2022 年第二季度广泛使用。2021 年，Stanza 项目被捐赠给 OpenTelemetry，为 OpenTelemetry 收集器增加了高效的日志处理能力。

用于 HTTP/RPC、数据库和消息系统的语义公约预计将在 2022 年第一季度宣布稳定。

11.2 未来

完成上述路线图就完成了对 OpenTelemetry 核心功能的稳定性要求。这是一个巨大的里程碑，因为它打开了进一步采用 OpenTelemetry 的大门，包括与数据库、管理服务和 OSS 库的原生集成。这些集成工作大部分已经以原型和测试版支持的形式在进行。随着 OpenTelemetry 的稳定性在 2022 年上半年完成，我预计在 2022 年下半年将出现支持 OpenTelemetry 的宣言，使 2022 年成为 “OpenTelemetry 之年”。

但我们并没有就此止步。下一步是什么？

11.3 eBPF

Extended Berkeley Packet Filter（eBPF）是一种在 Windows、Linux 和其他类似 Unix 的操作系统上提供底层网络访问的机制。利用 eBPF 将给 OpenTelemetry 提供一种极其有效的网络监控形式，不需要任何开发人员的工具。

2021 年，Flowmill 项目被捐赠给了 OpenTelemetry。Flowmill 是一个基于 eBPF 的可观测性解决方案，专门设计用于观测分布式系统。Flowmill 的开发者与 Pixie 项目的开发者一起，正在努力为 OpenTelemetry Collector 增加 eBPF 支持。Pixie 是专门为 Kubernetes 设计的基于 eBPF 的观测工具，是 CNCF 旗下 OpenTelemetry 的一个姊妹项目。我们还在一起研究如何将底层（第 2 层）eBPF 数据与高层（第 7 层）分布式追踪数据进一步关联起来，这在业界尚属首次。

11.4 RUM

真实用户监控（RUM）是一种可观测性工具，用于描述用户在长期运行的用户会话中如何与移动、网络和桌面客户端进行交互。RUM 与分布式追踪不同，因为图形用户界面（GUI）往往是基于反应器的系统，与数据库和网络服务器等基于事务的系统有根本的架构差异。长话短说：你不能把一个用户会话建模为一个追踪，然后就收工了。

Client Instrumentation SIG 目前正在开发一个新的 RUM 设计，它将扩展并与 OpenTelemetry 现有的分布式追踪、指标和日志信号完全整合。

11.5 OpenTelemetry 控制平面

目前，OpenTelemetry 收集器和 SDK 是作为独立的单元来管理的，必须重新启动才能改变它们的配置。目前，Agent Management SIG 正在开发一个控制平面，它可以报告这些组件的当前状态，并允许实时改变配置。它将允许运维人员或自动服务动态地控制整个遥测管道的处理。

动态配置将允许对采样、日志水平和其他形式的资源管理进行细粒度、反应灵敏的控制，从而减少成本。最终，这个控制平面可以实现更先进的基于尾部的采样形式，即需要在整个部署中协调的采样技术。

11.6 列式编码的 OTLP

目前的 OpenTelemetry 协议是对 OpenTelemetry 数据模型的一种有效而直接的编码。跨度、度量和日志被编码为跨度、度量和日志。这可以被认为是一个**基于行的模型**，每个跨度、度量和日志都被编码为元素列表中的一个离散元素。基于列的模型则将每个元素编码为一个单一的表格，其中所有元素共享相同的列，用于重叠的概念，如时间戳和属性。

这种列式表示法有望优化数据批次的创建、大小和处理。这种方法的主要好处是：

- 更好的数据压缩率（一组相似的数据）。
- 更快的数据处理（更好的数据定位意味着更好地使用 CPU 缓存线）。
- 更快的序列化和反序列化（更少的对象需要处理）。
- 更快的批量创建（更少的内存分配）。
- 更好的 I/O 效率（传输的数据更少）。

这种方法的好处与批次的大小成比例地增加。使用现有的 “面向行” 的表示方法很适合小批量的情况。因此，列式编码将**扩展**当前的协议。目前提出的实施方案是基于 Apache Arrow，这是一种成熟的列式内存格式。

这种优化的重点是允许高容量的数据源，如 CDN 和大型多租户系统，像常规服务一样参与可观测性。列式编码也将减少跨网络边界的遥测出口的成本。

扫码关注「几米宋」
了解更多

jimmysong.io