目錄

为什么需要Tracing？

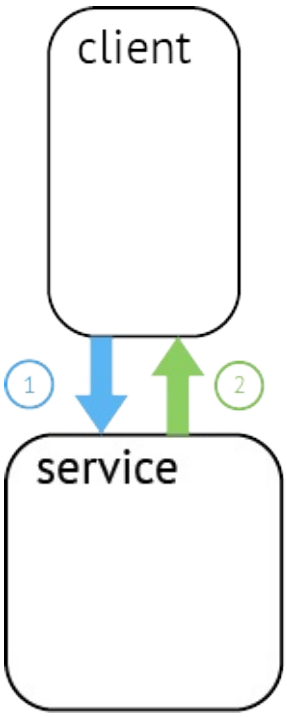
开发和工程团队因为系统组件水平扩展、开发团队小型化、敏捷开发、CD（持续集成）、解耦等各种需求，正在使用现代的微服务架构替换老旧的单片机系统。 也就是说，当一个生产系统面对真正的高并发，或者解耦成大量微服务时，以前很容易实现的重点任务变得困难了。过程中需要面临一系列问题：用户体验优化、后台真是错误原因分析，分布式系统内各组件的调用情况等。 当代分布式跟踪系统（例如，Zipkin, Dapper, HTrace, X-Trace等）旨在解决这些问题，但是他们使用不兼容的API来实现各自的应用需求。尽管这些分布式追踪系统有着相似的API语法，但各种语言的开发人员依然很难将他们各自的系统（使用不同的语言和技术）和特定的分布式追踪系统进行整合，

为什么需要OpenTracing？

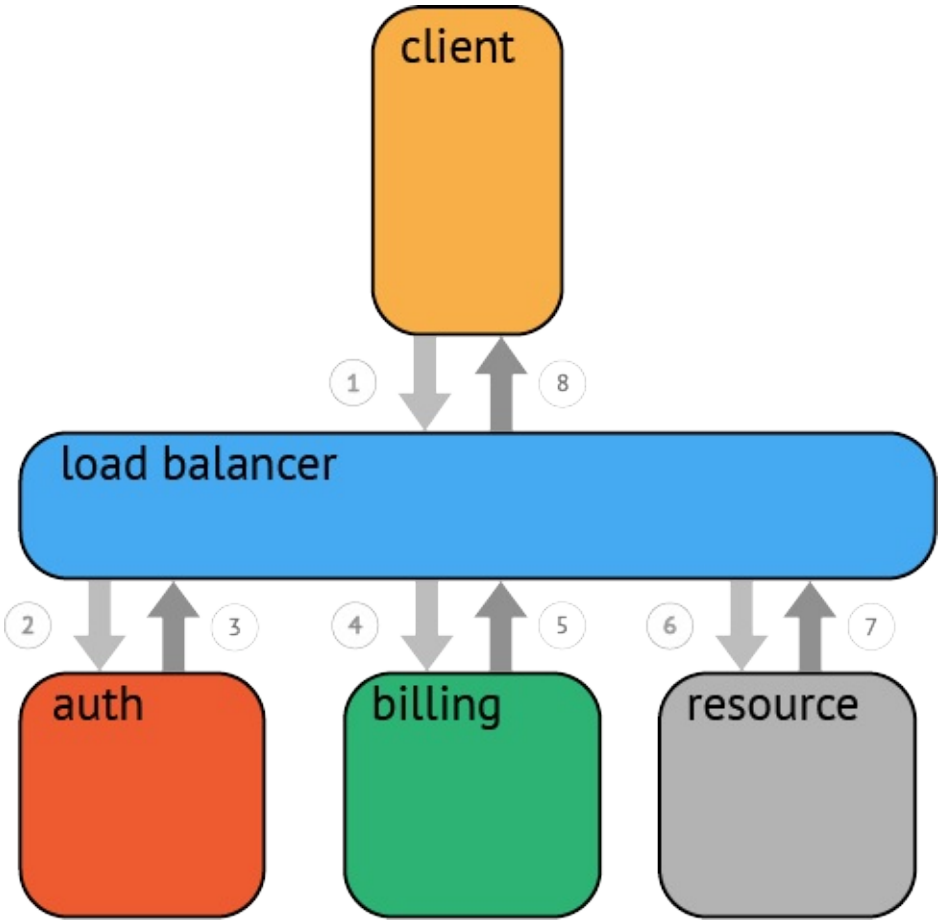
OpenTracing通过提供平台无关、厂商无关的API，使得开发人员能够方便的添加（或更换）追踪系统的实现。 OpenTracing提供了用于运营支撑系统的和针对特定平台的辅助程序库。程序库的具体信息请参考详细的规范。

什么是一个Trace?

在广义上，一个trace代表了一个事务或者流程在（分布式）系统中的执行过程。在OpenTracing标准中，trace是多个span组成的一个有向无环图（DAG），每一个span代表trace中被命名并计时的连续性的执行片段。

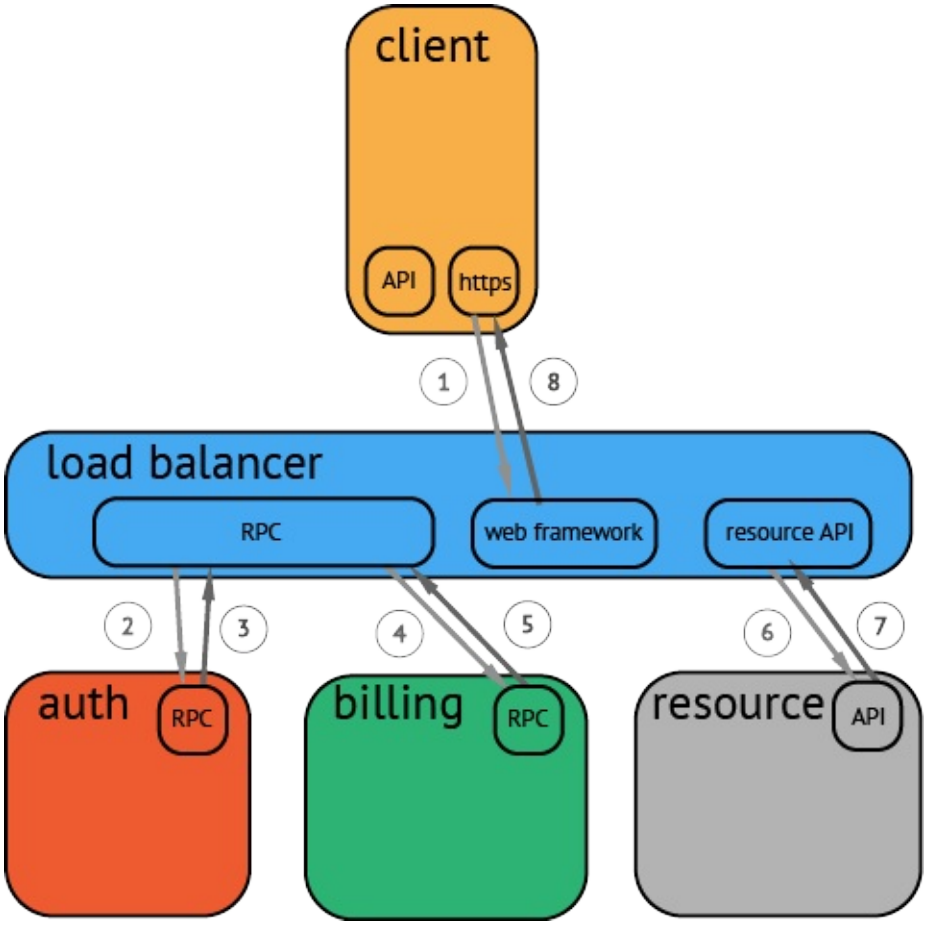


分布式追踪中的每个组件都包含自己的一个或者多个span。例如，在一个常规的RPC调用过程中，OpenTracing推荐在RPC的客户端和服务端，至少各有一个span，用于记录RPC调用的客户端和服务端信息。

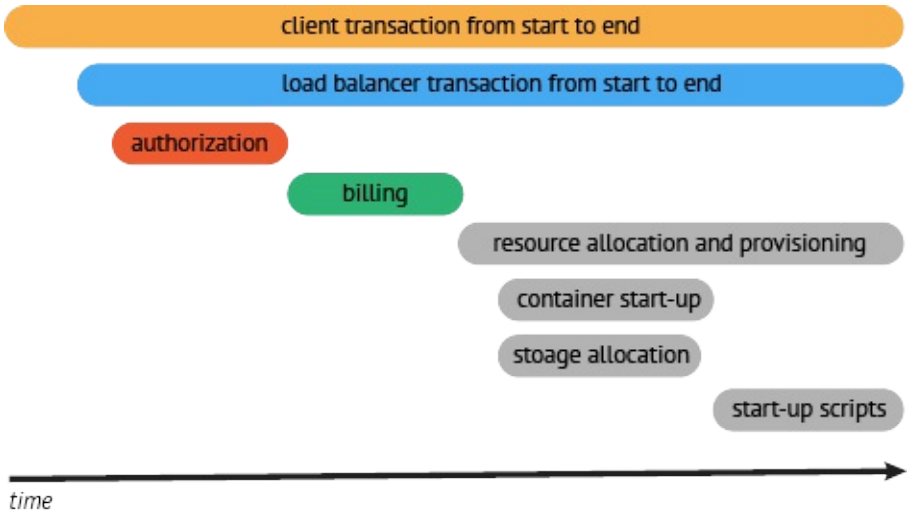


一个父级的span会显示的并行或者串行启动多个子span。在OpenTracing标准中，甚至允许一个子span有个多父span（例如：并行写入的缓存，可能通过一次刷新操作写入动作）。

一个典型的Trace案例



在一个分布式系统中，追踪一个事务或者调用流一般如上图所示。虽然这种图对于看清各组件的组合关系是很有用的，但是，它不能很好显示组件的调用时间，是串行调用还是并行调用，如果展现更复杂的调用关系，会更加复杂，甚至无法画出这样的图。另外，这种图也无法显示调用间的时间间隔以及是否通过定时调用来启动调用。一种更有效的展现一个典型的trace过程，如下图所示：



这种展现方式增加显示了执行时间的上下文，相关服务间的层次关系，进程或者任务的串行或并行调用关系。这样的视图有助于发现系统调用的关键路径。通过关注关键路径的执行过程，项目团队可能专注于优化路径中的关键位置，最大幅度的提升系统性能。例如：可以通过追踪一个资源定位的调用情况，明确底层的调用情况，发现哪些操作有阻塞的情况。

10分钟完成分布式追踪

原始发布地址 *OpenTracing blog*

随着并发和异步成为现代软件应用的必然特性，分布式追踪系统成为有效监控的一个必须的组成部分。尽管如此，监控并追踪一个系统的调用情况，至今仍是一个耗时而复杂的任务。随着系统的调用分布式程度（超过10个进程）和并发度越来越高，移动端与web端、客户端到服务端的调用关系越来越复杂，追踪调用关系带来的好处是显而易见的。但是选择和部署一个追踪系统的过程十分复杂。OpenTracing标准将改变这一点，OpenTracing尽力让监控一个分布式调用过程简单化。正如我下面视频演示的那样，你能在10分钟内快速配置一个监控系统。

本文描述的示例应用程序使用过程截图

试想一个简单的web网站。当用户访问你的首页时，web服务器发起两个HTTP调用，其中每个调用又访问了数据库。这个过程是否简单直白，我们可以不费什么力气就能发现请求缓慢的原因。如果你考虑到调用延迟，你可以为每个调用分布式唯一的ID，并通过HTTP头进行传递。如果请求耗时过长，你通过使用唯一ID来grep日志文件，发现问题出在哪里。现在，想想一下，你的web网站变得流行起来，你开始使用分布式架构，你的应用需要跨越多个机器，多个服务来工作。随着机器和服务数量的增长，日志文件能明确解决问题的机会越来越少。确定问题发生的原因将越来越困难。这时，你发现投入调用流程追踪能力是非常有价值的。

正如我提到的，OpenTracing因为standardizes instrumentation, 监控标准化，会使得追踪过程变得容易。它意味着，你可以先进行追踪，再决定最终的实现方案。

以AppDash为例，你可以根据如下的步骤，从编译web项目到查看追踪信息。或者，你可以直接使用Appdash来完成追踪并查看追踪信息。

```
docker run --rm -ti -p 8080:8080 -p 8700:8700 bg451/opentracing-example
```

这将启动一个测试的本地的Appdash实例。点击查看源码

如果你想看到完成的实例，你可以根据下面的步骤，自己构建webapp，使用OpenTracing设置追踪，绑定到一个追踪系统（如AppDash），并最终查看调用情况。

创建一个web工程

在开始之前，先写几个简单的调用点:

```
// Acts as our index page
func indexHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte(`<a href="/home"> Click here to start a request </a>`))
}
func homeHandler(w http.ResponseWriter, r *http.Request) {
     w.Write([]byte("Request started"))
    go func() {
        http.Get("http://localhost:8080/async")
    }()
    http.Get("http://localhost:8080/service")
    time.Sleep(time.Duration(rand.Intn(200)) * time.Millisecond)
    w.Write([]byte("Request done!"))
}
// Mocks a service endpoint that makes a DB call
func serviceHandler(w http.ResponseWriter, r *http.Request) {
    // ...
    http.Get("http://localhost:8080/db")
    time.Sleep(time.Duration(rand.Intn(200)) * time.Millisecond)
    // ...
}
// Mocks a DB call
func dbHandler(w http.ResponseWriter, r *http.Request) {
    time.Sleep(time.Duration(rand.Intn(200)) * time.Millisecond)
    // here would be the actual call to a DB.
}
```

将这些调用点组合成一个server

```
func main() {
    port := 8080
    addr := fmt.Sprintf(":%d", port)
    mux := http.NewServeMux()
    mux.HandleFunc("/", indexHandler)
    mux.HandleFunc("/home", homeHandler)
    mux.HandleFunc("/async", serviceHandler)
    mux.HandleFunc("/service", serviceHandler)
    mux.HandleFunc("/db", dbHandler)
    fmt.Printf("Go to http://localhost:%d/home to start a request!\n", port)
    log.Fatal(http.ListenAndServe(addr, mux))
}
```

将这些放到 `main.go` 文件中，运行 `go run main.go` 。

监控应用程序

现在，你有了一个可以工作的web应用服务器，你可以开始监控它了。你可以开始像下面这样，在入口设置一个span：

```
func homeHandler(w http.ResponseWriter, r *http.Request) {
  span := opentracing.StartSpan("/home") // Start a span using the global, in this cas
e noop, tracer
  defer span.Finish()
  // ... the rest of the function
}
```

这个span记录**homeHandler**方法完成所需的时间，这只是可以记录的信息的冰山一角。OpenTracing允许你为每一个span设置tags和logs。例如：你可以通过**homeHandler**方法是否正确返回，决定是否记录方法调用的错误信息：

```
// The ext package provides a set of standardized tags available for use.
import "github.com/opentracing/opentracing-go/ext"

func homeHandler(w http.ResponseWriter, r *http.Request) {
// ...
// We record any errors now.
    _, err := http.Get("http://localhost:8080/service")
    if err != nil {
        ext.Error.Set(span, true) // Tag the span as errored
        span.LogEventWithPayload("GET service error", err) // Log the error
    }
// ...
}
```

你也可以添加其他事件信息，如：发生的重要事件，用户id，浏览器类型。

然而，这只是其中的一个功能。为了构建真正的端到端追踪，你需要包含调用HTTP请求的客户端的span信息。在我们的示例中，你需要在端到端过程中传递span的上下文信息，使得各端中的span可以合并到一个追踪过程中。这就是API中Inject/Extract的职责。**homeHandler**方法在第一次被调用时，创建一个根span，后续过程如下：

```
func homeHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Request started"))
    span := opentracing.StartSpan("/home")
    defer span.Finish()

    // Since we have to inject our span into the HTTP headers, we create a request
    asyncReq, _ := http.NewRequest("GET", "http://localhost:8080/async", nil)
    // Inject the span context into the header
    err := span.Tracer().Inject(span.Context(),
        opentracing.TextMap,
        opentracing.HTTPHeaderTextMapCarrier(asyncReq.Header))
    if err != nil {
        log.Fatalf("Could not inject span context into header: %v", err)
    }
    go func() {
        if _, err := http.DefaultClient.Do(asyncReq); err != nil {
            span.SetTag("error", true)
            span.LogEvent(fmt.Sprintf("GET /async error: %v", err))
        }
    }()
    // Repeat for the /service call.
    // ....
}
```

上述代码，在底层实际的执行逻辑是：将关于本地追踪调用的span的元信息，被设置到http的头上，并准备传递出去。下面会展示如何在**serviceHandler**服务中提取这个元数据信息。

```
func serviceHandler(w http.ResponseWriter, r *http.Request) {
    var sp opentracing.Span
    opName := r.URL.Path
    // Attempt to join a trace by getting trace context from the headers.
    wireContext, err := opentracing.GlobalTracer().Extract(
        opentracing.TextMap,
        opentracing.HTTPHeaderTextMapCarrier(r.Header))
    if err != nil {
        // If for whatever reason we can't join, go ahead an start a new root span.
        sp = opentracing.StartSpan(opName)
    } else {
        sp = opentracing.StartSpan(opName, opentracing.ChildOf(wireContext))
    }
  defer sp.Finish()
  // ... rest of the function
```

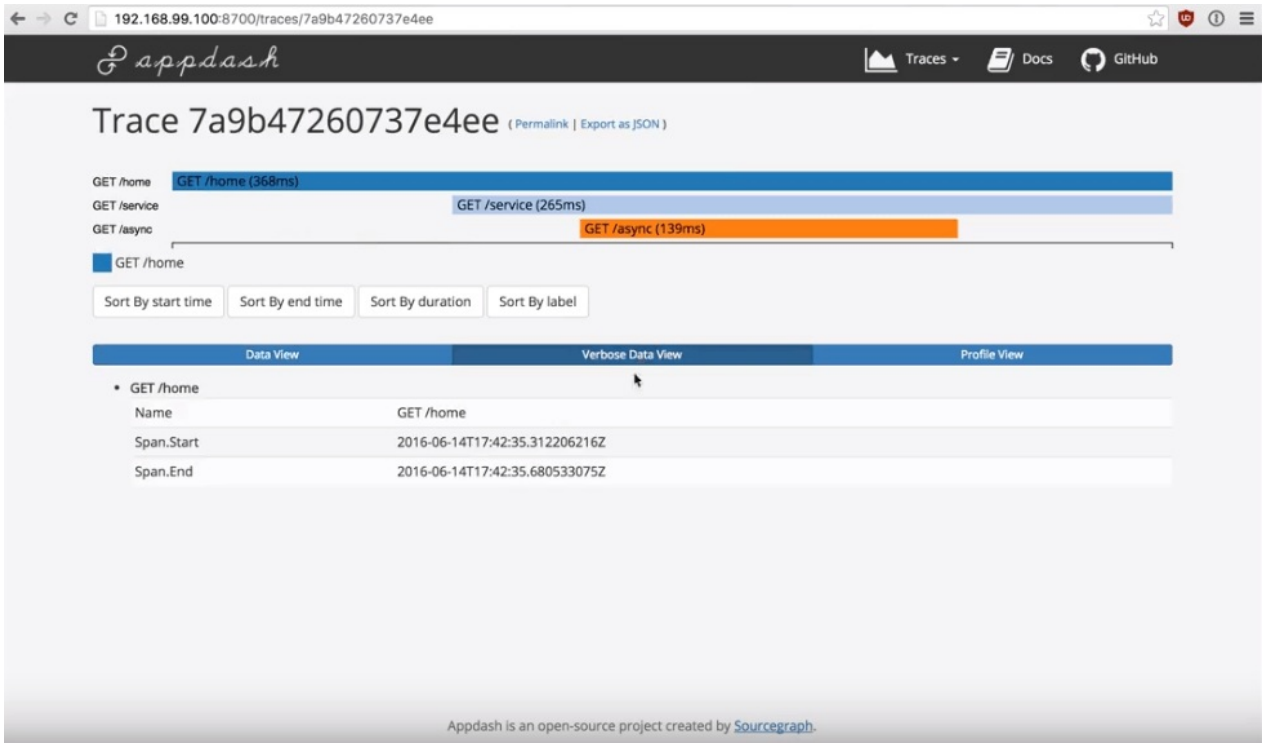
如上述程序所示，你可以通过http头获取元数据。你可以重复此步骤，为你需要追踪的调用进行设置，很快，你将可以监控整套系统。如何决定哪些调用需要被追踪呢？你可以考虑你的调用的关键路径。

连接到追踪系统

OpenTracing最重要的作用就是，当你的系统按照标准被监控之后，增加一个追踪系统将变得非常简单！在这个示例，你可以看到，我使用了一个叫做Appdash的开源追踪系统。你需要通过在main函数中增加一小段代码，来启动Appdash实例。但是，你不需要修改任何你关于监控的代码。在你的main函数中，加入如下内容：

```
import (
    "sourcegraph.com/sourcegraph/appdash"
    “sourcegraph.com/sourcegraph/appdash/traceapp”
    appdashot "sourcegraph.com/sourcegraph/appdash/opentracing"
)

func main() {
    // ...
      store := appdash.NewMemoryStore()

    // Listen on any available TCP port locally.
    l, err := net.ListenTCP("tcp", &net.TCPAddr{IP: net.IPv4(127, 0, 0, 1), Port: 0})
    if err != nil {
        log.Fatal(err)
    }
    collectorPort := l.Addr().(*net.TCPAddr).Port
    collectorAdd := fmt.Sprintf(":%d", collectorPort)

    // Start an Appdash collection server that will listen for spans and
    // annotations and add them to the local collector (stored in-memory).
    cs := appdash.NewServer(l, appdash.NewLocalCollector(store))
    go cs.Start()

    // Print the URL at which the web UI will be running.
    appdashPort := 8700
    appdashURLStr := fmt.Sprintf("http://localhost:%d", appdashPort)
    appdashURL, err := url.Parse(appdashURLStr)
    if err != nil {
        log.Fatalf("Error parsing %s: %s", appdashURLStr, err)
    }
    fmt.Printf("To see your traces, go to %s/traces\n", appdashURL)

    // Start the web UI in a separate goroutine.
    tapp, err := traceapp.New(nil, appdashURL)
    if err != nil {
         log.Fatal(err)
    }
    tapp.Store = store
    tapp.Queryer = store
    go func() {
    log.Fatal(http.ListenAndServe(fmt.Sprintf(":%d", appdashPort), tapp))
    }()

    tracer := appdashot.NewTracer(appdash.NewRemoteCollector(collectorPort))
    opentracing.InitGlobalTracer(tracer)
// ...
}
```

这样你会增加一个嵌入式的Appdash实例，并对本地程序进行监控。



如果你想换一个监控系统的实现，如果他们都符合OpenTracing，你只需要进行一步操作。你只需要修改你的main函数，其他所有的监控代码，都可以保持不变。例如，如果你决定使用Zipkin，你只需要在main函数中进行如下修改：

```
import zipkin "github.com/openzipkin/zipkin-go-opentracing"

func main() {
  // ...
  // Replace Appdash tracer code with this
  collector, err := zipkin.NewKafkaCollector("ZIPKIN_ADDR")
  if err != nil {
    log.Fatal(err)
    return
  }

  tracer, err = zipkin.NewTracer(
    zipkin.NewRecorder(collector, false, "localhost:8000", "example"),
  )
  if err != nil {
    log.Fatal(err)
  }
  opentracing.InitGlobalTracer(tracer)
  // ...
}
```

到目前为止，你会发现，使用OpenTracing使得监控你的代码更简单。我推荐在启动一个新项目的研发过程中，就加入监控的代码。因为，即使你的应用很小，追踪数据也可以在你的应用演进，引入分布式的时候，提供数据支持。帮助你在这个过程中，构建一个可持续迭代的产品。

概念和术语

一个tracer过程中，各span的关系

```
        [Span A]  ←←←(the root span)
            |
     +------+------+
     |             |
 [Span B]      [Span C] ←←←(Span C 是 Span A 的孩子节点, ChildOf)
     |             |
 [Span D]      +---+-------+
               |           |
           [Span E]    [Span F] >>> [Span G] >>> [Span H]
                                         ↑
                                         ↑
                                         ↑
                         (Span G 在 Span F 后被调用, FollowsFrom)
```

上述tracer与span的时间轴关系

```
--|-------|-------|-------|-------|-------|-------|-------|-> time

 [Span A···················································]
   [Span B·····························································]
      [Span D··························································]
    [Span C·····························································]
         [Span E·····]        [Span F··] [Span G··] [Span H··]
```

Traces

一个trace代表一个潜在的，分布式的，存在并行数据或并行执行轨迹（潜在的分布式、并行）的系统。一个trace可以认为是多个span的有向无环图（DAG）。

Spans

一个span代表系统中具有开始时间和执行时长的逻辑运行单元。span之间通过嵌套或者顺序排列建立逻辑因果关系。

Operation Names

每一个span都有一个操作名称，这个名称简单，并具有可读性高。（例如：一个RPC方法的名称，一个函数名，或者一个大型计算过程中的子任务或阶段）。span的操作名应该是一个抽象、通用的标识，能够明确的、具有统计意义的名称；更具体的子类型的描述，请使用Tags

例如，假设一个获取账户信息的span会有如下可能的名称：

操作名	指导意见
`get`	太抽象
`get_account/792`	太明确
`get_account`	正确的操作名，关于 `account_id=792` 的信息应该使用Tag操作

Inter-Span References

一个span可以和一个或者多个span间存在因果关系。OpenTracing定义了两种关系：`ChildOf` 和 `FollowsFrom` 。这两种引用类型代表了子节点和父节点间的直接因果关系。未来，OpenTracing将支持非因果关系的span引用关系。（例如：多个span被批量处理，span在同一个队列中，等等）

`ChildOf` 引用：一个span可能是一个父级span的孩子，即"ChildOf"关系。在"ChildOf"引用关系下，父级span某种程度上取决于子span。下面这些情况会构成"ChildOf"关系：

- 一个RPC调用的服务端的span，和RPC服务客户端的span构成ChildOf关系
- 一个sql insert操作的span，和ORM的save方法的span构成ChildOf关系
- 很多span可以并行工作（或者分布式工作）都可能是一个父级的span的子项，他会合并所有子span的执行结果，并在指定期限内返回

下面都是合理的表述一个"ChildOf"关系的父子节点关系的时序图。

```
    [-Parent Span---------]
         [-Child Span----]

    [-Parent Span--------------]
         [-Child Span A----]
          [-Child Span B----]
        [-Child Span C----]
         [-Child Span D---------------]
         [-Child Span E----]
```

`FollowsFrom` 引用：一些父级节点不以任何方式依然他们子节点的执行结果，这种情况下，我们说这些子span和父span之间是"FollowsFrom"的因果关系。"FollowsFrom"关系可以被分为很多不同的子类型，未来版本的OpenTracing中将正式的区分这些类型

下面都是合理的表述一个"FollowFrom"关系的父子节点关系的时序图。

```
    [-Parent Span-]  [-Child Span-]


    [-Parent Span--]
     [-Child Span-]


    [-Parent Span-]
               [-Child Span-]
```

Logs

每个span可以进行多次**Logs**操作，每一次**Logs**操作，都需要一个带时间戳的时间名称，以及可选的任意大小的存储结构。

标准中定义了一些日志（logging）操作的一些常见用例和相关的log事件的键值，可参考Data Conventions Guidelines 数据约定指南。

Tags

每个span可以有多个键值对（key:value）形式的**Tags**，**Tags**是没有时间戳的，支持简单的对span进行注解和补充。

和使用**Logs**的场景一样，对于应用程序特定场景已知的键值对**Tags**，tracer可以对他们特别关注一下。更多信息，可参考Data Conventions Guidelines 数据约定指南。

SpanContext

每个span必须提供方法访问**SpanContext**。SpanContext代表跨越进程边界，传递到下级span的状态。(例如，包含 `<trace_id, span_id, sampled>` 元组)，并用于封装**Baggage** (关于Baggage的解释，请参考下文)。SpanContext在跨越进程边界，和在追踪图中创建边界的时候会使用。(ChildOf关系或者其他关系，参考Span间关系)。

Baggage

Baggage是存储在SpanContext中的一个键值对(SpanContext)集合。它会在一条追踪链路上的所有span内全局传输，包含这些span对应的SpanContexts。在这种情况下，"Baggage"会随着trace一同传播，他因此得名（Baggage可理解为随着trace运行过程传送的行李）。鉴于全栈OpenTracing集成的需要，Baggage通过透明化的传输任意应用程序的数据，实现强大的功能。例如：可以在最终用户的手机端添加一个Baggage元素，并通过分布式追踪系统传递到存储层，然后再通过反向构建调用栈，定位过程中消耗很大的SQL查询语句。

Baggage拥有强大功能，也会有很大的消耗。由于Baggage的全局传输，如果包含的数量量太大，或者元素太多，它将降低系统的吞吐量或增加RPC的延迟。

Baggage vs. Span Tags

- Baggage在全局范围内，（伴随业务系统的调用）跨进程传输数据。Span的tag不会进行传输，因为他们不会被子级的span继承。
- span的tag可以用来记录业务相关的数据，并存储于追踪系统中。实现OpenTracing时，可以选择是否存储Baggage中的非业务数据，OpenTracing标准不强制要求实现此特性。

Inject and Extract

SpanContexts可以通过**Injected**操作向**Carrier**增加，或者通过**Extracted**从**Carrier**中获取，跨进程通讯数据（例如：HTTP头）。通过这种方式，SpanContexts可以跨越进程边界，并提供足够的信息来建立跨进程的span间关系（因此可以实现跨进程连续追踪）。

平台无关的API语义

OpenTracing支持了很多不同的平台，当然，每个平台的API试图保持各平台和语言的习惯和管理，尽量做到入乡随俗。也就是说，每个平台的API，都需要根据上述的核心tracing概念来建模实现。在这一章中，我们试图描述这些概念和语义，尽量减少语言和平台的影响。

The `Span` Interface

`Span`接口必须实现以下的功能：

- **Get the `Span`'s `SpanContext`**，通过span获取SpanContext（即使`span`已经结束，或者即将结束）
- **Finish**，完成已经开始的`Span`。处理获取`SpanContext`之外，Finish必须是span实例的最后一个被调用的方法。**(py: `finish`, go: `Finish`)**。一些的语言实现方法会在`Span`结束之前记录相关信息，因为Finish方法可能不会被调用，因为主线程处理失败或者其他程序错误。在这种情况下，实现应该明确的记录`Span`，保证数据的持久化。
- **Set a key:value tag on the `Span`.**，为`Span`设置tag。tag的key必须是`string`类型，value必须是`string`，`boolean`或数字类型。tag其他类型的value是没有定义的。如果多个value对应同一个key（例如被设置多次），实现方式是没有被定义的。**(py: `set_tag`, go: `SetTag`)**
- **Add a new log event**，为`Span`增加一个log事件。事件名称是`string`类型，参数值可

以是任何类型，任何大小。tracer的实现者不一定保存所有的参数值（设置可以所有参数值都不保存）。其中的时间戳参数，可以设置当前时间之前的时间戳。**(py: `log`, go: `Log`)**

- **Set a Baggage item**, 设置一个string:string类型的键值对。注意，新设置的Baggage元素，只保证传递到未来的子级的 `Span` 。参考下图所示。**(py: `set_baggage_item`, go: `SetBaggageItem`)**
- **Get a Baggage item**， 通过key获取Baggage中的元素。**(py: `get_baggage_item`, go: `BaggageItem`)**

```
        [Span A]
            |
     +------+------+
     |             |
 [Span B]      [Span C] ←←← (1) BAGGAGE ITEM "X" IS SET ON SPAN C,BUT AFTER SPAN E ALR
EADY STARTED.
     |             |            为SPAN C设置BAGGAGE元素，值为X，时间点为SPAN E已经开始运行
 [Span D]      +---+-----+
               |         |
           [Span E]  [Span F] >>> [Span G] >>> [Span H]
                                                 ↑
                                                 ↑
                                                 ↑
             (2) BAGGAGE ITEM "X" IS AVAILABLE FOR RETRIEVAL BY SPAN H (A CHILD OF SPA
N C), AS WELL AS SPANS F AND G.
                 SPAN C元素的F\G\H子级span可以读取到BAGGAGE的元素X
```

The `SpanContext` Interface

`SpanContext` 接口必须实现以下功能。用户可以通过 `Span` 实例或者 `Tracer` 的Extract能力获取 `SpanContext` 接口实例。

- **Iterate over all Baggage items** 是一个只读特性。**(py: `baggage`, go: `ForeachBaggageItem`)**
- 虽然以前 `SpanContext` 是 `Tracer` 接口的一部分，但是 `SpanContext` 对于Inject and Extract是必不可少的。

The `Tracer` Interface

`Tracer` 接口必须实现以下功能：

- **Start a new `Span`**, 创建一个新的Span。调用者可以指定一个或多个 `SpanContext` 关系(例如 `FollowsFrom` 或 `ChildOf` 关系)，显示声明一个开始的时间戳（除"now"之外），并设置处理化的 `Span` 的tags数据集。**(py: `start_span`, go: `StartSpan`)**

- **Inject a `SpanContext`** ，将 `SpanContext` 注入到 `SpanContext` 对象中，用于进行跨进程的传输。"carrier"类型需要反射或者明确指定的方式来确定。查看end-to-end propagation example 端到端传递示例获取更多消息。
- **Extract a `SpanContext`** ，通过"carrier"跨进程获取 `SpanContext` 信息。Extract会检查 `carrier` ，尝试获取先前通过 `Inject` 放入的数据，并重建 `SpanContext` 实例。除非有错误发生，Extract返回一个包含 `SpanContext` 实例，此实例可以用来创建一个新的子级Span。（注意：一些OpenTracing实现方式，认为 `Span` 在RPC的两端应该具有相同的ID，而另一些考虑客户端是父级span，服务端是子级span）。"carrier"类型需要反射或者明确指定的方式来确定。查看end-to-end propagation example 端到端传递示例获取更多消息。

Global and No-op Tracers

每一个平台的OpenTracing API库 (例如 opentracing-go, opentracing-java，等等；不包含OpenTracing `Tracer` 接口的实现）必须提供一个no-op Tracer（不具有任何操作的tracer）作为接口的一部分。No-op Tracer的实现必须不会出错，并且不会有任何副作用，包括baggage的传递时，也不会出现任何问题。同样，Tracer的实现也必须提供no-op Span实现；通过这种方法，监控代码不依赖于Tracer关于Span的返回值，针对no-op实现，不需要修改任何源代码。No-op Tracer的Inject方法永远返回成功，Extract返回的效果，和"carrier"中没有找到 `SpanContext` 时返回的结果一样。

每一个平台的OpenTracing API库 可能 支持配置(Go: `InitGlobalTracer()` , py: `opentracing.tracer = myTracer`)和获取单例的全局Tracer实例(Go: `GlobalTracer()` , py: `opentracing.tracer`)。如果支持全局的Tracer，默认返回的必须是no-op Tracer。

OpenTracing APIs

- OpenTracing Language Support, OpenTracing多语言支持
- Data Conventions, 数据约定
- Cross-Process Tracing, 跨进程追踪

OpenTracing多语言支持

OpenTracing API支持以下平台：

- Go - https://github.com/opentracing/opentracing-go
- Python - https://github.com/opentracing/opentracing-python
- Javascript - https://github.com/opentracing/opentracing-javascript
- Objective-C - https://github.com/opentracing/opentracing-objc
- Java - https://github.com/opentracing/opentracing-java
- C++ - https://github.com/opentracing/opentracing-cpp

OpenTracing API以下平台实现正在研发中：

- PHP - link forthcoming
- Ruby - link forthcoming

有关各平台的使用示例，请参考查阅各平台仓库的README文档。

欢迎社区贡献其他语言的实现。

Data Conventions 数据约定

介绍

OpenTracing通过定义的API，可实现将监控数据记录到一个可插拔的tracer上。总体上来说，OpenTracing不能保证底层追踪系统的实现方式。那么API层应该提供什么类型的数据来保证这些底层追踪系统实现的兼容性呢？

监控软件和追踪软件开发者在高层次的共识，将产生巨大的价值：如果在一些通用的应用场景下，都使用某些已知的tag的键值对，tracer程序可以选择对他们进行特别的关注。被 `log` 的事件，span的结构也是如此。

例如，考虑基于HTTP的应用服务器。应用系统处理的请求中的URL、HTTP动作（get/post等）、返回码，对于应用系统的诊断是非常有帮助的。监控者可以选择使用tag方法标记这个参数，命名为 `URL` 或 `http.url` ，从纯API技术角度来说是有效的。但是，如果一个tracer需要增加一些高级功能，例如根据URL的值建立索引，或者针对特定来源的请求进行采样，你必须知道数据的格式。换句话说，tag的名字和监控程序的提供方的要求必须是一致的，这样追踪程序才能在收到数据后，提供更加智能的分析结果。

本文档对追踪软件开发和探针软件开发都有通用指导意义。追踪系统的开发者不必严格遵守指南，但是强烈推荐大家这么做。

Spans

Span Naming, Span命名

Span 可以包含很多的tags、logs和baggage，但是始终需要一个高度概括的**operation name**。这些应该是一个简单的字符串，代表span中进行的工作类型。这个字符串应该是工作类型的逻辑名称，例如代表一个RPC或者一次HTTP的调用的端点，亦或对于代表SQL的span，使用 `SELECT` or `INSERT` 作为逻辑名，等等。

其次，span可以存在一个可选的tag，叫做**component**，他的值可以典型的代表一个进程、框架、类库或者模块名称。这个tag会很有价值。

Span Structure, Span的结构

Span的结构也是非常重要的：span代表了什么，span和span的上下级是什么关系？这些内容在章节Concepts and Terminology, 概念与术语中描述。

Span Tag Use-Cases, Span Tag操作用例

监控软件开发者，如果试图标注如下特定类型的数据，请使用下面推荐的tags。tag名称遵循命名空间的通用结构（即：java包名的结构）

下面推荐的tag，在 `ext` 模块中，都会为每一个实现制定一个 `const` 常量值。这些 `ext` 的值应该用来代表下面的字符串，不同的追踪系统，可以为这些通用概念选择不同的底层实现。在每种实现中，这些值的实现方式是十分相似的。(例如：Go, Python)

下面提供的一下tags可能包含一些象征大小的值。如何处理这些值是依赖于实现的：追踪系统会需要适当选择，是否要使用、删除或者清空这些tags标记。然而，不仅仅追踪程序才需要关注这些值，给追踪系统生成、传递这些值，也可能对应用系统造成不良影响。

监控系统可以只支持其中的部分tags。

Errors

一个span实例的错误状态，通过一个tag来标注。

- `error` - bool
  - `true` 代表这个span是错误状态
  - `false` 或没有 `error` tag ，代表span没有发生错误

Component Identification, 框架定义

对于任何一个span，被监控的组件，指定组件的类型是十分有帮助的。十分推荐库或者模块为监控程序提供组件的定义，最终用户可能会拥有一个由框架和第三方混合提供的监控。

- `component` - string
  - 需要被检测/监控的类库、模块、包的基本名称。
  - Examples:
    - `httplib` 代表Python内建的httplib函数功能
    - `JDBC` 代表JDBC数据库连接
    - `mongoose` 代表Ruby的MongoDB客户端连接
- `span.kind` - string
  - `client` 或 `server` , 指定这个span代表一个客户端还是服务端

HTTP Server Tags

这些tag作用于基于HTTP的服务入口的span。

- `http.url` - string
  - URL 分布式追踪中，这一阶段的调用的URL地址, 参考 standard URI format.

    - Protocol 协议，可选
    - Examples:
        - `https://domain.net/path/to?resource=here`
        - `domain.net/path/to/resource`
        - `http://user:pass@domain.org:8888`
- `http.method` - string
    - HTTP 请求被处理的方法.
    - Case-insensitive 大小写敏感
    - Examples:
        - `GET`, `POST`, `HEAD`
- `http.status_code` - integer
    - HTTP 返回值
    - Examples:
        - `200`, `503`
- `span.kind` - string
    - `server` 定义这是服务端类型的span (see "Component Identification, 框架定义")

Peer Tags

这些tag可以被客户端或者服务端提供，用于描述远程请求过程中，请求调用的方向。（客户端记录下行访问，服务端记录上行访问）

- `peer.hostname` - string
    - 目标 hostname
- `peer.ipv4` - string
    - 目标 IP v4 地址
- `peer.ipv6` - string
    - 目标 IP v6 地址
- `peer.port` - integer
    - 目标 port
- `peer.service` - string
    - 目标服务名称

Sampling, 采样

OpenTracing API不强调采样的概念，但是大多数追踪系统通过不同方式实现采样。有些情况下，应用系统需要通知追踪程序，这条特定的调用需要被记录，即使根据默认采样规则，它不需要被记录。`sampling.priority` tag 提供这样的方式。追踪系统不保证一定采纳这个参数，但是会尽可能的保留这条调用。

- `sampling.priority` - integer

- 如果大于 0, 追踪系统尽可能保存这条调用链
- 等于 0, 追踪系统不保存这条调用链
- 如果此tag没有提供，追踪系统使用自己的默认采样规则

Logs

Common fields

OpenTracing中的每一次日志记录都会包含一个时间戳，并至少包含一个基于键值对的域。以下是一些标准化的域定义 Every Log record in OpenTracing has a timestamp and at least one key:value "field". The following fields are standardized:

- `event` - string
  - 事件域代表span生命周期内某些关键时间点的标识。例如，在浏览器页面加载过程中，获得或释放一个互斥锁就是一个特定的事件域，可参考 Performance.timing 标准.

每一个对Inject和Extract存在疑惑，又羞于启齿的人

开发者为应用程序增加跨进程追踪能力时，必须理解the OpenTracing specification中定义的 `Tracer.Inject(...)` 和 `Tracer.Extract(...)` 的能力。这两个方法在概念上十分强大，他允许开发人员正确并抽象的完成跨进程传输的代码，而不需要绑定特定的**OpenTracing**的实现；也就是说，强大的能力带来了巨大的困惑:)

这篇文档，针对 `Inject` 和 `Extract` 设计和用法，提供一个简要的总结，而不考虑特定的OpenTracingg规范各语言的实现和基于OpenTracing标准的追踪系统。

显示的trace传播的“重大作用”

分布式追踪系统最困难的部分就是在分布式的应用环境下保持追踪的正常工作。任何一个追踪系统，都需要理解多个跨进程调用间的因果关系，无论他们是通过RPC框架、发布-订阅机制、通用消息队列、HTTP请求调用、UDP传输或者其他传输模式。

一些分布式追踪系统（例如，2003年的Project5，2006年的WAP5，2014年的The Mystery Machine）会推断跨进程间的因果关系。当然，这些系统，都需要在基于黑盒的因果关系推断与 追踪结果的整合、实时准确展现上，进行处理折衷。 处于对准确展现的关注，OpenTracing是一个明确的分布式追踪系统标准，它更倾向于如果产品的处理方式：2007年的X-Trace，2010年的Dapper，以及很多开源的追踪系统，如：Zipkin，Appdash 等等

`Inject` 和 **`Extract`** 允许开发者进行跨进程追踪时，不用和特定的**OpenTracing**实现进行紧耦合

OpenTracing跨进程传播需求

为了使 `Inject` 和 `Extract` 有效，必须遵守如下要求：

- 如上文所述，OpenTracing 用户 处理跨进程的追踪传输时，必须不需要使用OpenTracing使用中的特定代码
- 基于OpenTracing的追踪系统，必须不需要针对每一种已知的跨进程通讯机制都进行处理：这其中包含太多的工作，很多还没有明确的定义
- 也就是说，这套传播机制必须是最利于扩展的

基本方法：Inject, Extract, 和 Carriers

追踪过程中的任何一个SpanContext可以被**Injected**（注入）到一个Carrier中。**Carrier**可以是一个接口或者一个数据载体，他对于跨进程通讯（IPC）是十分有帮助的。**Carrier**负责将追踪状态从一个进程"carries"（携带，传递）到另一个进程。OpenTracing标准包含两种必须的Carrier 格式，尽管，自定义的 Carrier 格式 也是可能的。

同样的，对于一个Carrier，如果已经被**Injected**，那么它也可以被**Extracted**（提取），从而得到一个SpanContext实例。这个SpanContext代表着被**Injected**到Carrier的信息。

Inject伪代码示例

```
span_context = ...
outbound_request = ...

# 我们将使用（内建的）基于HTTP_HEADERS的carrier格式。
# 我们在调用`tracer.inject之前，先将一个空的map作为一个carrier
carrier = {}
tracer.inject(span_context, opentracing.Format.HTTP_HEADERS, carrier)

# `carrier` 现在（隐形）包含我们通过网络传输的键值对。
for key, value in carrier:
    outbound_request.headers[key] = escape(value)
```

Extract 伪代码示例

```
inbound_request = ...

# 我们将再次使用基于（内建的）HTTP_HEADERS carrier格式。
# 按照HTTP_HEADERS的文档，我们可以使用一个map来存储外来的值，
# 允许OpenTracing实现者来根据需要，
# 来查找集合内部的键值对。
#
# 也就是说，我们直接使用基于键值对的`inbound_request.headers`作为carrier。
carrier = inbound_request.headers
span_context = tracer.extract(opentracing.Format.HTTP_HEADERS, carrier)
# Continue the trace given span_context. E.g.,
span = tracer.start_span("...", child_of=span_context)

# (如果 `carrier` 保存着trace的数据， 则现在可以创建`span`了。)
```

Carrier格式

所有的Carrier都有自己的格式。在一些语言的OpenTracing实现中，格式必须必须作为一个常量或者字符串来指定； 另一些，则通过Carrier的静态类型来指定。

Inject/Extract Carrier 所必须的格式

至少，OpenTracing标准所有平台的实现者支持两种Carrier格式：基于"text map"（基于字符串的map）的格式和基于"binary"（二进制）的格式。

- *text map* 格式的 Carrier是一个平台惯用的map格式，基于unicode编码的 `字符串` 对 `字符串` 键值对
- *binary* 格式的 Carrier 是一个不透明的二进制数组（可能更紧凑和有效）

OpenTracing的实现者选择如何将数据存储到Carrier中，OpenTracing标准没有正式定义，但是，可以推测的是，他们会通过一种方式编码“追踪状态”，来传递 `SpanContext` （例如，Dapper会包含 `trace_id` ， `span_id` ，以及一位掩码标识这个trace的采样状态）和Baggage中的其他键值对数据。

各种**OpenTracing**实现者，实现 跨进程边界 方式的互操作性

不能期待不同的OpenTracing实现， `Inject` 和 `Extract` SpanContexts采用相互兼容的方式。虽然OpenTracing对于实现 跨整个分布式系统 的追踪系统是无从得知的，为了成功实现跨进程的追踪的我收过程，跨进程追踪的两端应该使用相同的追踪系统实现。（即远程调用的两段，使用同一套tracer）。

自定义的 Inject/Extract Carrier 格式

任何的基于网络传输的子系统（RPC库，消息队列等）可能选择引入他们自定义的Inject/Extract的Carrier格式；根据需要自定义格式，但最终要求返回符合**OpenTracing**格式的结果。这样允许OpenTracing的实现者可以优化他们自己的自定义格式，而不需要实现者支持这些子系统的自定义格式。

一些伪代码将可能更明确的说明这个问题。假设我们是**ArrrPC pirate RPC subsystem**的作者，我们希望增加OpenTracing的数据在RPC请求过程中传输。不考虑异常处理，我们的伪代码可能如下所示：

```
span_context = ...
outbound_request = ...

# 首先，我们使用我们自定义的Carrier：outbound_request
# 如果我们优先支持OpenTracing的实现，这样会更加高效的处理。
# 但是，这不是一个必须的格式要求，我们不能指望基于OpenTracing的
# 追踪程序支持arrrpc.ARRRPC_OT_CARRIER参数
try:
    tracer.inject(span_context, arrrpc.ARRRPC_OT_CARRIER, outbound_request)

except opentracing.UnsupportedFormatException:
    # If unsupported, fall back on a required OpenTracing format.
    # 如果不支持，则使用OpenTracing支持的格式
    carrier = {}
    tracer.inject(span_context, opentracing.Format.HTTP_HEADERS, carrier)
    # `carrier` 现在包含键值对，我们可以使用任何网络协议，来传输这个键值对即可
    for key, value in carrier:
    outbound_request.headers[key] = escape(value)
```

关于**Carrier**自定义格式的更多内容

"Carrier的格式"在不同平台可能是不一样的，但在所有的场景下，他们都会使用一个全局的命名空间。支持一个全新的自定义格式的carrier不必修改OpenTracing核心平台的API，尽管每一个实现OpenTracing平台API时，必须定义符合OpenTracing标准要求的carrier格式（比如：基于字符串的map和二进制块）。例如，ArrrPC RPC的维护团队定义了一个叫做"ArrrPC"的Inject/Extract格式，他们不需要向OpenTracing团队提交PR（当然OpenTracing的实现者不要求一定支持"ArrrPC"格式）。an end-to-end injector and extractor example below，一个端到端的injector和extractor示例 将更具体的描述这个问题。

一个端到端的**injector**和**extractor**示例

为了让描述更具体，考虑如下的流程：

1. 一个客户端进程有一个 `SpanContext` 实例，并准备进行通过自制的HTTP协议，进行一次RPC调用
2. 客户端进程调用 `Tracer.Inject(...)` ，传入 `SpanContext` 实例，支持基于字符格式的map的标识符，以及支持基于字符map的Carrier，三个参数
3. Carrier将在Carrier对基于字符的map（参数2）填充必要的数据；客户端应用程序会在自制的HTTP协议中，对这个map进行编码（如：添加到HTTP头中）
4. 进行*HTTP*调用，将数据进行跨进程传输
5. 现在，在服务端进程进行处理。应用程序从自制的HTTP协议中解码出上文所述的map（参数2），并通过他，初始化基于字符map的Carrier
6. 服务端进程调用 `Tracer.Extract(...)` ，传入需要的操作名（operation name），支持就

要字符格式的map的标识符，以及上面构建的Carrier

7. 不考虑数据丢失，和其他错误，服务端现在有了和客户端追踪上下文中，一样的 `SpanContext` 。

在OpenTracing use cases, OpenTracing常见用例 文档中，可以找到其他使用案例。

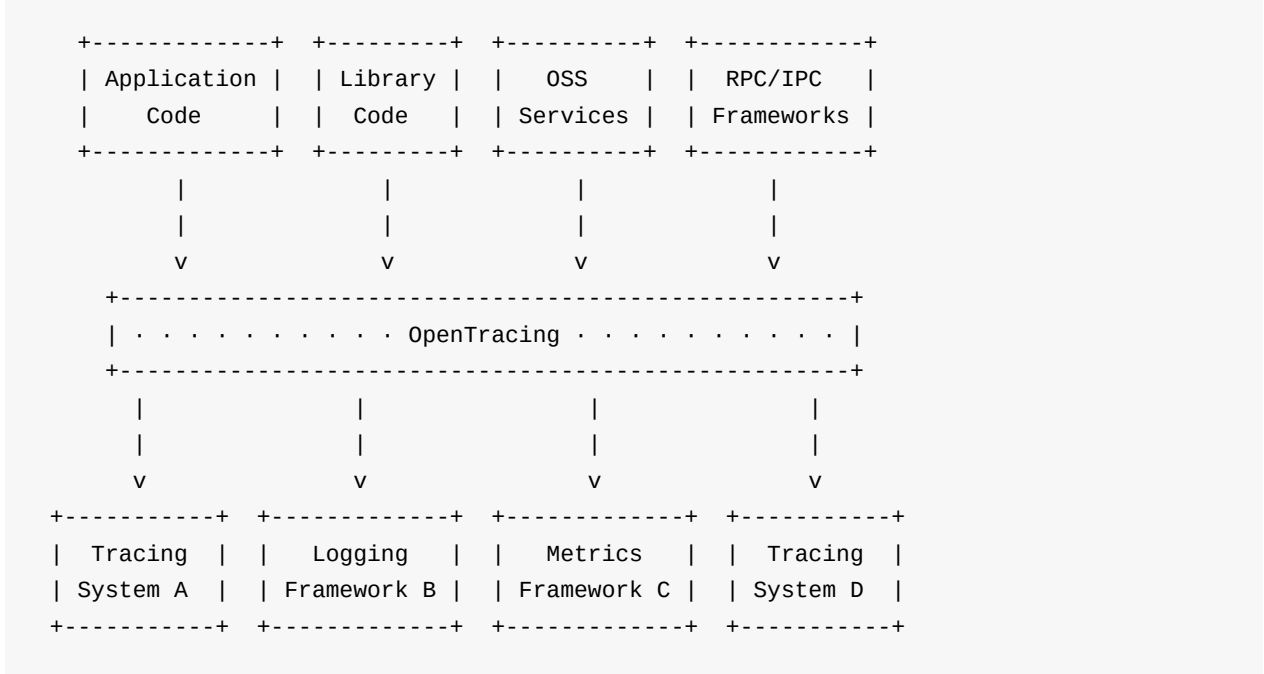
Best Practices

常见用例

本章的主要目的是，针对通过使用OpenTracing API来监控应用程序或类库的开发者，提供示例说明。

回到伊始：**OpenTracing**是为了哪些人建立的？

OpenTracing是一个轻量级的标准化层，它位于应用程序/类库 和 追踪或日志分析程序 之间。



Application Code, 应用程序代码： 开发者在开发业务代码时，可以通过OpenTracing来描述追踪数据间的因果关系，控制流程，增加细粒度的日志信息。

Library Code, 类库代码：类似的，类库程序作为请求控制的中介媒介，也可以通过OpenTracing来描述追踪数据间的因果关系，控制流程，增加细粒度的日志信息。例如：一个web中间件类库，可以使用OpenTracing，在请求被处理时新增span；或者，一个ORM类库，可以使用OpenTracing来描述高级别的ORM语义和特定SQL查询间的关系。

OSS Services, OSS服务（运营支持服务）：除嵌入式类库以外，整个OSS服务可以采取OpenTracing标准来，集成分布式追踪系统来处理一个大型的分布式系统中的复杂调用关系。例如，一个HTTP的负载均衡器可以使用OpenTracing标准来设置请求（如：设置请求图），或者一个基于键值对的存储系统使用OpenTracing来解读系统的读写性能。

RPC/IPC Frameworks，**RPC/IPC**框架（远程调用框架）：任何一个跨进程的子任务，都可以通过使用OpenTracing，来标准化追踪数据注入到传输协议中的格式。

所有上面这些，都应该使用OpenTracing来描述和传递分布式追踪数据，而不需要了解**OpenTracing**的实现。

OpenTracing 优先级

由于OpenTracing层的 上层 有更多的应用程序和开发者（而不是下层），API和用例的易用性也倾向于他们。这篇文档中的用例将面向OpenTracing API调用者（而非被调者），帮助他们在建立辅助的类库和各种抽象模型，最终有利于为OpenTracing实现者节省时间和精力。

让我们直接进入主题：

用例

追踪Function（函数）

```
def top_level_function():
    span1 = tracer.start_span('top_level_function')
    try:
        . . . # business logic，业务逻辑
    finally:
        span1.finish()
```

后续，作为业务逻辑的一部分，我们调用了 `function2` 方法，也想被追踪。为了让这个追踪附着在正在进行的追踪上（和上述的追踪形成一根调用链）。我们将在后面的t章节讨论如何实现，现在，我们假设一个 `get_current_span` 函数可以完成这个功能：

```
def function2():
    span2 = get_current_span().start_child('function2') \
        if get_current_span() else None
    try:
        . . . # business logic
    finally:
        if span2:
            span2.finish()
```

我们假设，如果这个追踪还未被启动，无论什么原因，开发者都不想在这个函数内启动一个新的追踪，所以我们考虑到 `get_current_span` 函数可能返回 `None` 。

这两个例子都非常的简单。 通常情况下，应用程序不希望追踪代码和业务代码混在一起，而使用其他方式，例如：标注等，参考function decorator in Python:

```
@traced_function
def top_level_function():
    ... # business logic
```

服务端追踪

当一个应用服务器要追踪一个请求的执行情况，他一般需要以下几步：

1. 试图从请求中获取传输过来的SpanContext（防止调用链在客户端已经开启），如果无法获取SpanContext，则新开启一个追踪。
2. 在*request context*中存储最新d创建的span，*request context*会通过应用程序代码或者RPC框架进行传输
3. 最终，当服务端完成请求处理后，使用 `span.finish()` 关闭span。

从请求中获取（Extracting）SpanContext

假设，我们有一个HTTP服务器，SpanContext通过HTTP头从客户端传递到服务端，可通过 `request.headers` 访问到：

```
extracted_context = tracer.extract(
    format=opentracing.HTTP_HEADER_FORMAT,
    carrier=request.headers
)
```

这里，我们使用 `headers` 中的map作为carrier。追踪程序知道需要hearder的哪些内容，用来重新构建tracer的状态和Baggage。

从请求中获取一个已经存在的追踪，或者开启一个新的追踪

如果无法在请求的相关的头信息中获取所需的值，上文中的 `extracted_context` 可能为 `None`：此时我们假设客户端没有发送他们。在这种情况下，服务端需要新创建一个追踪（新调用链）。

```
extracted_context = tracer.extract(
    format=opentracing.HTTP_HEADER_FORMAT,
    carrier=request.headers
)
if extracted_context is None:
    span = tracer.start_span(operation_name=operation)
else:
    span = tracer.start_span(operation_name=operation, child_of=extracted_context)
span.set_tag('http.method', request.method)
span.set_tag('http.url', request.full_url)
```

可以通过调用 `set_tag` ，在Span中记录请求的附加信息。

上面提到的 `operation` 是通过提供的服务名指定Span的名称。例如,如果HTTP请求到 `/save_user/123` ，那么 `operation` 名称应该被设置为 `post:/save_user/` 。OpenTracing API不会强制要求应用程序如何给span命名。

进程内请求上下文传输

请求的上下文传输是指，对于一个请求，所有处理这个请求的层都需要可以访问到同一个*context*（上下文）。可以通过特定值，例如：用户id、token、请求的截止时间等，获取到这个*context*（上下文）。也可以通过这种方法获取正在追踪的Span。

请求*context*（上下文）的传输不属于OpenTracing API的范围，但是，这里提到他，是为了让大家更好的理解后面的章节。下面有两种常用的上下文传输技术：

隐式传输

隐式传输技术要求*context*（上下文）需要被存储到平台特定的位置，允许从应用程序的任何地方获取这个值。常用的RPC框架会利用thread-local 或 continuation-local存储机制，或者全局变量（如果是单线程处理）。

这种方式的缺点在于，有明显的性能损耗，有些平台比如Go不知道基于thread-local的存储，隐式传输将几乎不可能实现。

显示传输

显示传输技术要求应用程序代码，包装并传递*context*（上下文）对象：

```
func HandleHttp(w http.ResponseWriter, req *http.Request) {
    ctx := context.Background()
    ...
    BusinessFunction1(ctx, arg1, ...)
}

func BusinessFunction1(ctx context.Context, arg1...) {
    ...
    BusinessFunction2(ctx, arg1, ...)
}

func BusinessFunction2(ctx context.Context, arg1...) {
    parentSpan := opentracing.SpanFromContext(ctx)
    childSpan := opentracing.StartSpan(
        "...", opentracing.ChildOf(parentSpan.Context()), ...)
    ...
}
```

显示传输的缺点在于，它向应用程序代码，暴露了底层的实现。Go blog post这边文章提供了这种方式的深层次的解析。

追踪客户端调用

当一个应用程序作为一个RPC客户端时，它可能希望在发起调用之前，启动一个新的追踪的span，并将这个心的span随请求一起传输。下面，通过一个HTTP请求的实例，展现如何做到这点。

```
def traced_request(request, operation, http_client):
    # retrieve current span from propagated request context
    parent_span = get_current_span()

    # start a new span to represent the RPC
    span = tracer.start_span(
        operation_name=operation,
        child_of=parent_span.context,
        tags={'http.url': request.full_url}
    )

    # propagate the Span via HTTP request headers
    tracer.inject(
        span.context,
        format=opentracing.HTTP_HEADER_FORMAT,
        carrier=request.headers)

    # define a callback where we can finish the span
    def on_done(future):
        if future.exception():
            span.log(event='rpc exception', payload=exception)
        span.set_tag('http.status_code', future.result().status_code)
        span.finish()

    try:
        future = http_client.execute(request)
        future.add_done_callback(on_done)
        return future
    except Exception e:
        span.log(event='general exception', payload=e)
        span.finish()
        raise
```

- `get_current_span()` 函数不是OpenTracing API的一部分。它仅仅代表一个工具类的方法，通过当前的请求上下文获取当前的span。（在Python一般会这样用）。
- 我们假定HTTP请求是异步的，所以他会返回一个Future。我们为这次调用增加的成功回调函数，在回调函数内部完成当前的span。
- 如果HTTP客户端返回一个异常，则通过log方法将异常记录到span中。
- 因为HTTP请求可以在返回Future后发生异常，我们使用try/catch块，在任何情况下都会

完成span，保证这个span会被上报，并避免内存溢出。

使用 **Baggage /** 分布式上下文传输

上面通过网络在客户端和服务端间传输的Span和Trace，包含了任意的Baggage。客户端可以使用Baggage将一些额外的数据传递到服务端，以及这个服务端的下游其他服务器。

```
# client side
span.context.set_baggage_item('auth-token', '.....')

# server side (one or more levels down from the client)
token = span.context.get_baggage_item('auth-token')
```

Logging事件

我们在客户端span的示例代码中，已经使用过 `log` 。事件被记录不会有额外的负载，也不一定必须在span创建或完成时进行操作。例如，应用通过可以在执行过程中，通过获取当前请求的当前span,记录一个缓存未命中事件：

```
span = get_current_span()
span.log(event='cache-miss')
```

tracer会为事件自动增加一个时间戳，这点和Span的tag操作时不同的。也可以将外部的时间戳和事件相关联，例如，Log (Go)。

使用外部的时间戳，记录**Span**

因为多种多样的原因，有些场景下，会将OpenTracing兼容的tracer集成到一个服务中。例如，一个用户有一个日志文件，其中包含大量的来自黑盒进程（如：HAProxy）产生的span。为了让这些数据接入OpenTracing兼容的系统，API需要提供一种方法通过外部的时间戳记录span的信息。

```
explicit_span = tracer.start_span(
    operation_name=external_format.operation,
    start_time=external_format.start,
    tags=external_format.tags
)
explicit_span.finish(
    finish_time=external_format.finish,
    bulk_logs=map(..., external_format.logs)
)
```

在追踪开始之前，设置采样优先级

很多分布式追踪系统，通过采样来降低追踪数据的数量。有时，开发者想有一种方式，确保这条trace一定会被记录（采样），例如：HTTP请求中包含特定的参数，如 `debug=true` 。OpenTracing API标准化了一些有用的tag，其中一个被叫做"sampling priority"（采样优先级）：精确的语义是由追踪系统的实现者决定的，但是任何值大于0（默认）代表一条trace的高优先级。为了将 `debug` 属性传递给追踪系统，需要在追踪前进行预处理，如下面所写的这样：

```
if request.get('debug'):
    span = tracer.start_span(
        operation_name=operation,
        tags={tags.SAMPLING_PRIORITY: 1}
    )
```

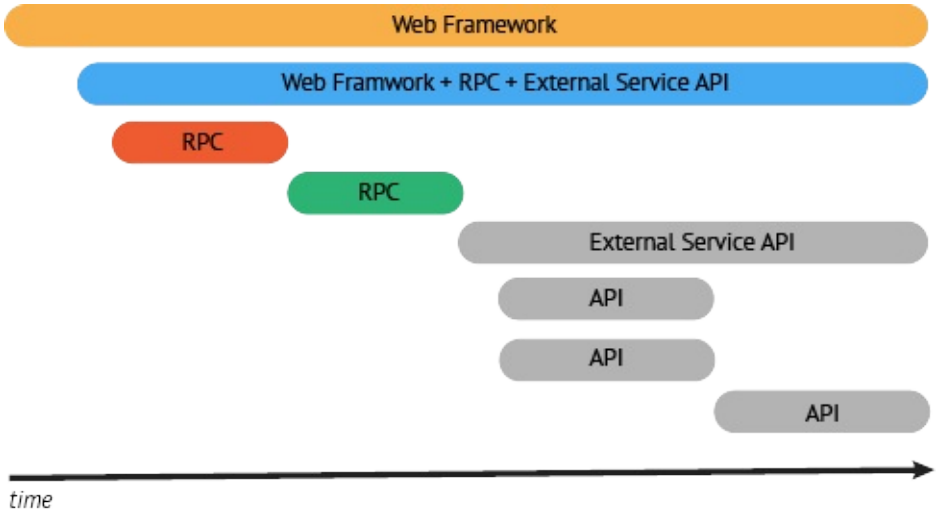
如何追踪大规模分布式系统

在阅读如何使用*OpenTracing*标准，监控大规模分布式系统之前，确保你已经阅读过*Specification overview*章节

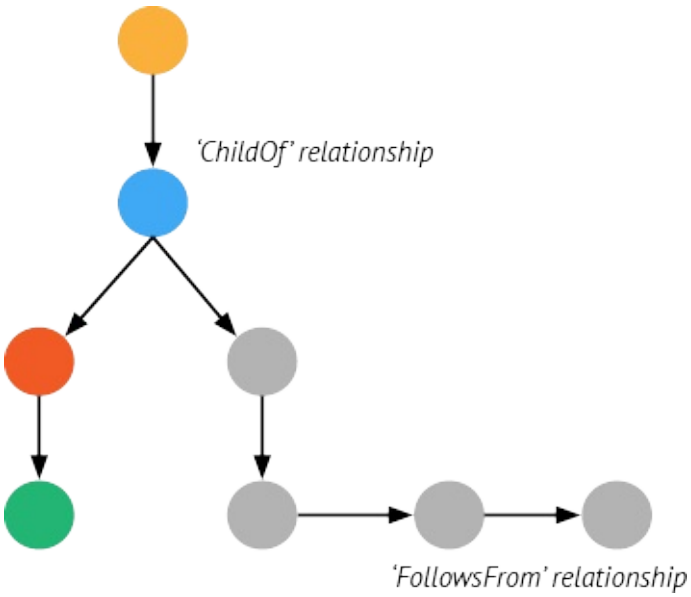
Spans 和它们之间的关系

实现OpenTracing完成分布式追踪的两个基本概念就是*Spans*和*Relationships*(span间关系)：

- ***Span*** 是系统中的一个逻辑工作单元，包含这个工作单元启动时间和执行时间。在一条追踪链路中，各个span与系统中的不同组件有关，并体现这些组件的执行路径。



- ***Relationships*** 是span间的连接关系。一个span可以和0-n个组件存在因果关系。这种关系是的各个span被串接起来，并用来帮助定位追踪链路的关键路径。



你所期待的结束状态，是获取你所有组件的span，以及它们之间的关系。当开始建立你的分布式追踪系统时，最好的方法是从服务框架（如：RPC层）或者其他和复杂执行路径有关的组件开始。

你可以从使用支持OpenTracing标准的服务框架开始 (如：gRPC）。但是，如果你不支持OpenTracing的框架，你可以阅读IPC/RPC Framework Guide章节。

专注高价值区域

如上面提到的，从RPC层和你的web框架开始构建追踪，是一个好方法。这两部分将包含事务路径中的大部分内容。

下一步，你应该着手在没有被服务框架覆盖的事务路径上。为足够多的组件增加监控，为高价值的事务创建一条关键链路的追踪轨迹。

你监控的首要目标，是基于关键路径上的span，寻找最耗时的操作，为可量化的优化操作提供最重要的数据支持。例如，对于只占用事务时间1%的操作（一个大粒度的span）增加更细粒度的监控，对于你理解端到端的延迟（性能问题）不会有太大意义。

先走再跑，逐步提高

如果你正在构建你的跨应用追踪系统实现，使用这套系统建立高价值的关键事务与平衡关键事务和代码覆盖率的概念。最大的价值，在于为关键事务生成端到端的追踪。可视化展现你的追踪结果是非常重要的。它可能帮助你确定那块区域（代码块/系统模块）需要更细粒度的追踪。

一旦你有了端到端的监控，你很容易评估在哪些区域增加投入，进行更细粒度的追踪，并能确定事情的优先级。如果你开始深入处理监控问题，可以考虑哪些部分能够复用。通过这些复用建立一套可以在多个服务间服用的监控类库。

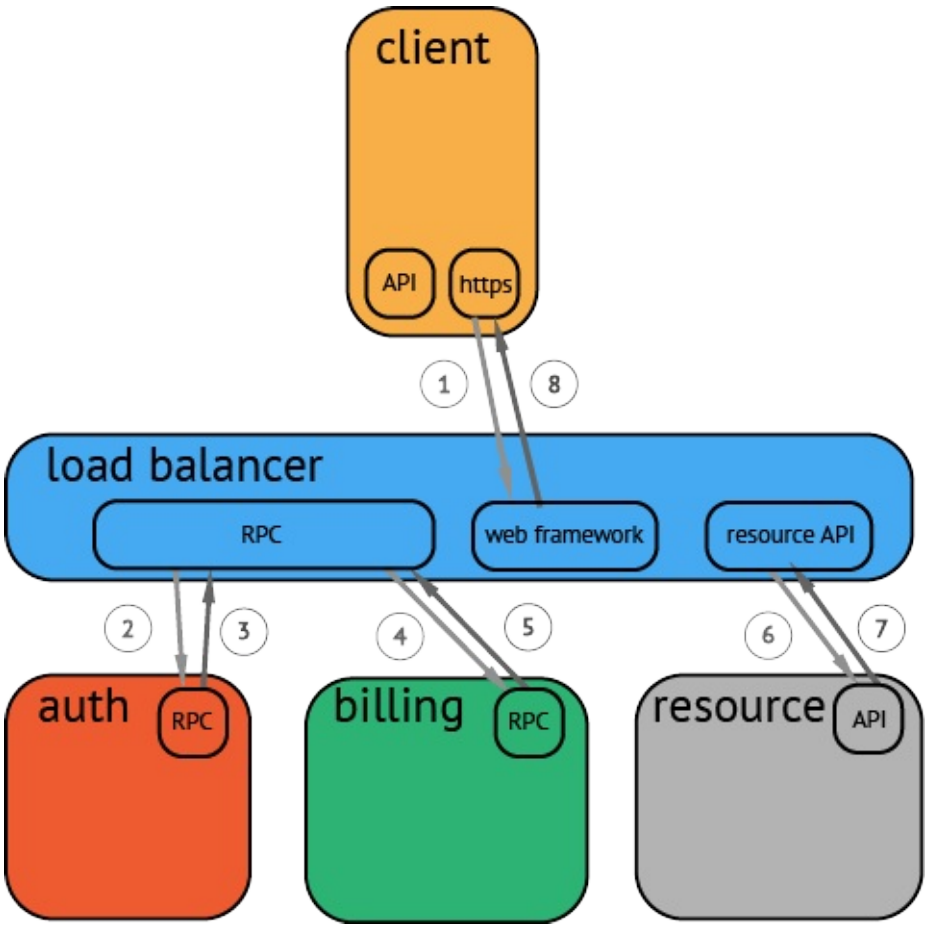
这种方法可以提供广泛的覆盖（如：RPC，web框架等），也能为关键业务的事务增加高价值的埋点。即使有些埋点（生成span）的代码是一次性工作，也能通过这种模式发现未来工作的优先级，优化工作效率。

示例实例

下面的例子让上述的概念更具体一些：

在这个例子中，我们想追踪一个，由手机端发起，调用了多个服务的调用链。

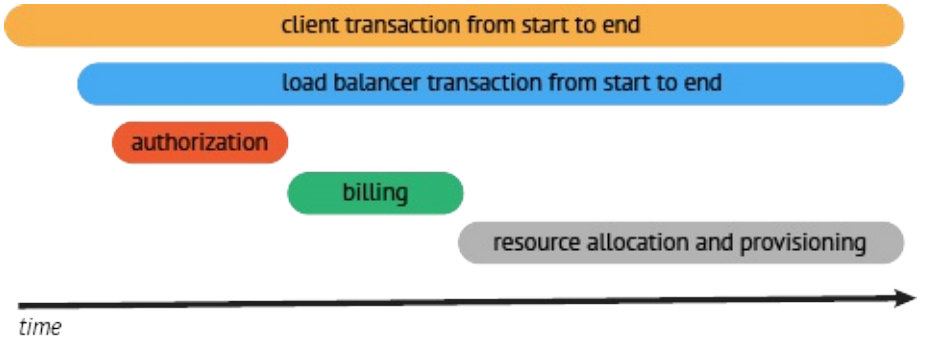
1. 首先，我们必须说明这个事务的大体情况。在我们的例子中，事务如下所示：



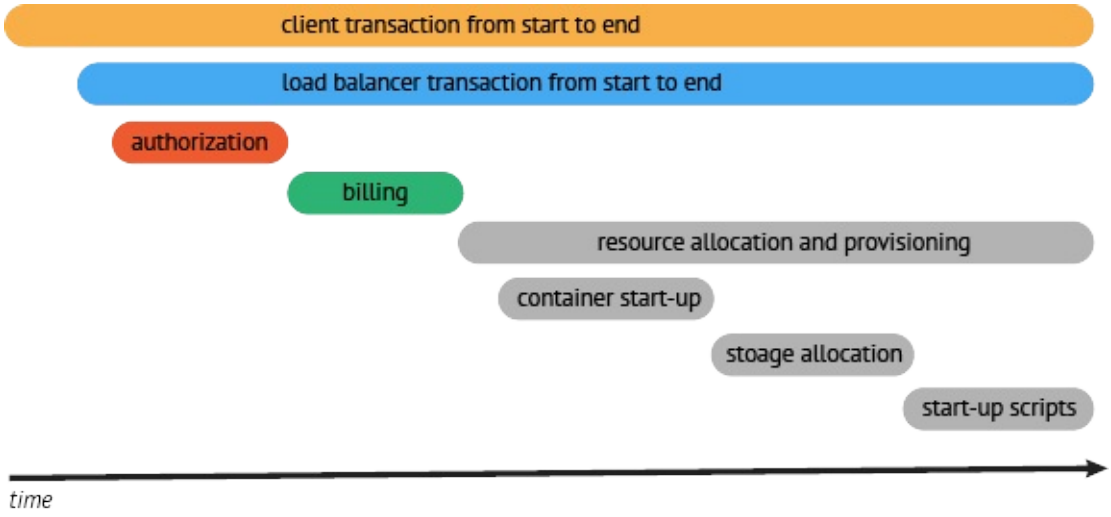
一个客户通过手机客户端向***web***发起了一个***HTTP***请求，产生一个复杂的调用流程：***mobile client (HTTP) → web tier (RPC) → auth service (RPC) → billing service (RPC) → resource request (API) → response to web tier (API) → response to client (HTTP)***

2. 现在，我们对事务的大概情况了解，我们去监控一些通用的协议和框架。最好的选择是从RPC服务框架开始，这将是收集web请求背后发生的调用情况的最好方式。（或者说，任何在分布式过程中发生的问题，都会在直接体现在RPC服务中）

3. 下一个重点监控的组件应该是web框架。通过增加web框架的监控，我们能够得到一个端到端的追踪链路。虽然这点追踪链路有点粗，但是至少，我们的追踪系统获取到了完整的调用栈。

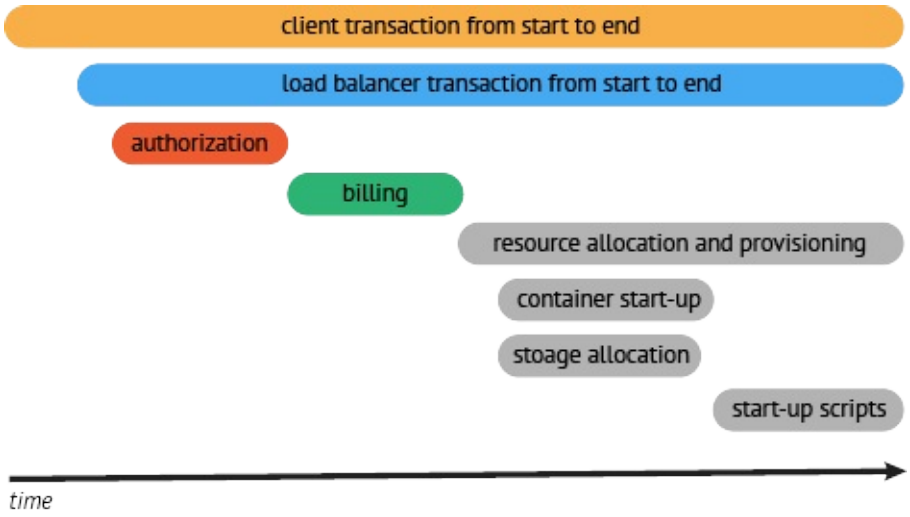


4. 通过上面的工作，我们可以看到所需的调用链，并评估我们细化哪一块的追踪。在我们的例子中，我们可以看到，请求中最耗时的操作时获取资源的操作。所以，我们应该细化这块的监控粒度，监控资源定位内部的组件。一旦我们完成资源请求的监控，我们可以看到资源请求被分解成下图所示的情况：



resource request (API) → container startup (API) → storage allocation (API) → startup scripts (API) → resource ready response (API)

5. 一旦我们完整资源组件的追踪，我们可以看到大量的时间消耗在提供上，下一步，我们深入分析，如果可能，我们优化资源获取程序，使用并行处理替代串行处理。



6. 现在我们有了一条基于端到端调用流程的可视化展现以及基线，我们可以为这个服务建立明确的SLO。另外，为内部服务建立SLO，可以成为对服务正常和错误运行的时间的讨论的基础。

7. 下一次跌倒，我们回到最顶层的追踪，去寻找下一个长耗时的任务，但是没有明细展现，这时需要更细粒度的追踪。如果展现的粒度已经足够，我们可以进行下一个关键事务的追踪和调优处理了。

8. 重复上述步骤.

监控框架

追踪所有的事情!

谁应该阅读本章节？

这篇指导文档，面向希望将OpenTracing将入到web、RPC或其他框架的监控当中的开发者。增加监控能力可以使框架能够和端到端的分布式追踪进行整合。

当一个请求跨越一套分布式系统时，分布式追踪可以提供请求在分布式系统内的运行情况。OpenTracing是一个开源的API标准，致力于分布式请求的追踪，保证能够追踪用户从web或移动端到后台应用，以及最终的数据存储。一旦OpenTracing完成跨应用栈（跨进程）的整合，在一个分布式系统中进行追踪将更容易。这将可以满足，开发者和运维人员对于产品服务优化和加强健壮性的要求。

在开始之前，请确保有你的平台（编程语言）有对应的OpenTracing API的实现。查看这里。

总览

总体来说，集成OpenTracing，你需要做下面两件事：

服务端框架修改需求：

- 过滤器、拦截器、中间件或其他处理输入请求的组件
- span的存储，存储一个request context或者request到span的映射表
- 通过某种方式对tracer进行配置

客户端框架修改需求：

- 过滤器、拦截器、中间件或其他处理对外调用的请求的组件
- 通过某种方式对tracer进行配置

重要提醒：

在我们专注于实现之前，有几个重要的概念和特性需要框架开发者所熟悉。

Operation Names，操作名

你会注意到operation_name（操作名）这个变量出现在这篇文章的各处。每一个span都需要通过一个operation_name创建，operation_name需要遵守规范的要求，点击查看。每一个span都需要一个默认的operation_name，并提供一种可以由用户命名的方式。

默认operation_name示例：

- request handler的方法名
- web请求路径
- RPC的服务名+方法名

确定需要追踪的请求

有些用户希望追踪所有的请求，同时，有些用户只需要追踪特定的请求。你应该允许用户去设置是否需要追踪，以满足这两种场景。例如，你可以提供@Trace标注，被标注的方法会被追踪。你也可以提供一种配置，允许用户去设置他们是否使用标准，所有的请求是不是应该被追踪。

追踪请求的属性

用户可能需要追踪关于请求的一些信息，而不希望去操作span或者为span设置tag。为用户提供一种方式设置需要追踪的请求的属性，并自动追踪这些属性值，是十分有帮助的。概念上，这和gRPC中的span的Decorator函数十分类似：

```
// SpanDecorator binds a function that decorates gRPC Spans.
func SpanDecorator(decorator SpanDecoratorFunc) Option {
    return func(o *options) {
        o.decorator = decorator
    }
}
```

另一种方式，是设置 `TRACED_REQUEST_ATTRIBUTES` ，允许用户传递一个列表（例如： `URL` , `METHOD` , `HEADERS` ），然后你会在追踪过滤器中，包含这些属性：

```
for attr in settings.TRACED_REQUEST_ATTRIBUTES:
    if hasattr(request, attr):
        payload = str(getattr(request, attr))
        span.set_tag(attr, payload)
```

服务端追踪

服务端追踪的目的是追踪请求在这个服务器内部的全生命周期的情况，并保证能够和前置的客户端追踪信息连接起来。你可以在服务器收到请求时，创建span，并在服务器完成请求处理后，关闭这些span。追踪一个服务端请求的流程如下：

- 服务器接收到请求
  - 从网络请求（跨进程的调用：HTTP等）获取当前的追踪链状态
  - 创建一个新的span
  - 保存当前的追踪状态
- 服务器完成请求处理 / 返回响应
  - 结束上面创建的span

由于调用流程决定于请求的处理情况，所以你需要知道如果修改框架的请求和响应处理——是否需要通过修改过滤器、中间件、配置栈或者其他机制。

获取当前的追踪链状态

为了在分布式系统中，跨进程边界追踪调用情况，RPC服务需要能够衔接每一个服务请求的服务端和客户端。OpenTracing允许通过inject和extract方法，将span的上下文信息编码到carrier中。（编码规范留给开发者确定，所以你需要担心这个问题。）

如果客户端发起一个请求时，span的上下文就已经被加到了请求内容中。你的工作是使用io.opentracing.Tracer.extract方法，从请求中获取span的上下文。carrier通过你使用哪种服务，决定是否哪种方法从请求中获取上下文；例如，web服务通过HTTP头作为carrier，从HTTP请求中获取span上下文（如下所示）：

Python:

```
span_ctx = tracer.extract(opentracing.Format.HTTP_HEADERS, request.headers)
```

Java:

```
import io.opentracing.propagation.Format;
import io.opentracing.propagation.TextMap;

Map<String, String> headers = request.getHeaders();
SpanContext parentSpan = tracer.getTracer().extract(Format.Builtin.HTTP_HEADERS,
    new TextMapExtractAdapter(headers));
```

OpenTracing当提取失败时，可以选择抛出异常，所以确保会捕获异常，防止异常造成服务器宕机。这种情况通常意味着请求来自于第三方应用（没有被追踪的应用），此时应该开启一个新的追踪。

启动一个**span**

一旦你接收到一个请求，并且获取到了span的上下文，你应该立即为这次请求创建一个span，代表这次请求的全生命周期。如果存在被提取出来的上下文，则新的服务端server应该是被提取出的span的孩子节点（ChildOf关系），代表客户端和服务端之间的调用关系。如果没有被注入的span，你需要启动一个新的span（没有上下级关系）。

Python:

```
if(extracted_span_ctx):
    span = tracer.start_span(operation_name=operation_name,
        child_of=extracted_span_ctx)
else:
    span = tracer.start_span(operation_name=operation_name)
```

Java:

```
if(parentSpan == null){
    span = tracer.buildSpan(operationName).start();
} else {
    span = tracer.buildSpan(operationName).asChildOf(parentSpan).start();
}
```

保存当前的span上下文

在处理请求期间，让用户可以访问span上下文是十分重要的。只有获取上下文，才能为服务端，进行自定义的tag设置，记录事件(log event)，创建子级的span，用于最终展现服务内部的工作情况。为了满足这个目标，你必须决定如何让用户访问当前的span。这将由框架的架构决定。这里有两个常见用例：

1. 使用请求上下文：如果你的框架有一个请求上下文，上下文可以存储任意值，这样你可以在请求处理过程中，一直把现在的span存储到上下文中。如果你的框架中有过滤器（Filter），这种实现方式是一种很好的方式。例如你有一个请求上下文叫做ctx，那么你可以这样实现一个过滤器（Filter）：

```
def filter(request):
    span = # extract / start span from request
    with (ctx.active_span = span):
        process_request(request)
    span.finish()
```

1. 现在，在请求处理的任何时候，用户都可以通过 `ctx.active_span` 获取当前的span。注意，一旦请求被处理， `ctx.active_span` 的值就不应该被改变。

2. 建立请求和span的映射关系：如果存在这种情况：如有可能没有一个可用的请求上下文，或者你针对请求的预处理和后处理有不同的过滤器方法， 你可以选择建立一个请求和span的映射表。其中一种实现方式是创建一个框架特有的tracer的包装器（tracer wrapper），存储这个映射表，例如：

```
class MyFrameworkTracer:
    def __init__(opentracing_tracer):
        self.internal_tracer = opentracing_tracer
        self.active_spans = {}
    def add_span(request, span):
        self.active_spans[request] = span
    def get_span(request):
        return self.active_spans[request]
    def finish_span(request):
        span = self.active_spans[request]
        span.finish()
        del self.active_spans[request]
```

1. 如果你的服务器可以并行的处理请求，请确保你的span的映射表是线程安全的。

2. 过滤器处理示例代码如下：

```
def process_request(request):
    span = # extract / start span from request
    tracer.add_span(request, span)
def process_response(request, response):
    tracer.finish_span(request)
```

1. 注意：用户在处理reponse时，调用 `tracer.get_span(request)` 获取当前的span，请确保用户依然能获取request实例。（也可以不使用request对象，而使用其他可以标识当前请求的参数）

客户端追踪

当框架有一个客户端组件的时候，需要在初始化request的时候，开启客户端的追踪。这样做是为了将生成的span放到请求头中，这样span才能请求随着请求，传递到服务端。类似于服务端追踪，你需要知道如何修改你的客户端代码，来发送请求，和接收相应。当客户端完成修改，就可以完成端到端的追踪了。

追踪一个客户端请求的流程如下：

- 准备请求对象
  - 读取现在的追踪状态
  - 新建一个span

    - 将span注入(Inject)到请求中
- 发送请求
- 接收响应
    - 完成并关闭span

读取现在的追踪状态 / 新建一个span

正如服务端一样，我们必须知道是应该开启一个新的追踪或者和一个已有的追踪连接上。例如,一个基于微服务架构分布式架构中，一个应用可能即是服务端又是客户端。一个服务的提供方同时又是另一个服务的发起方，这个东西需要被联系起来。如果存在一个活跃的调用链，你需要帮他的活跃span作为父级span，并在客户端请求出开启一个新的span。否则，你需要新建没有没有父级节点的span。

如何判断是否存在一个活跃的追踪，取决于你如何存储的活跃的span。如果你使用一个请求上下文，你可以这样处理：

```
if hasattr(ctx, active_span):
    parent_span = getattr(ctx, active_span)
    span = tracer.start_span(operation_name=operation_name,
        child_of=parent_span)
else:
    span = tracer.start_span(operation_name=operation_name)
```

如果你使用request到span的映射机制，你可以这样处理：

```
parent_span = tracer.get_span(request)
span = tracer.start_span(
    operation_name=operation_name,
    child_of=parent_span)
```

gRPC 和 JDBI 的处理实例。

注入(Inject) Span

注入span的时候，你会把当前追踪的上下文信息放到客户端的请求中，这样当调用发生时，追踪可以在服务端被还原，并继续进行。如果是使用HTTP请求，你可以使用HTTP头作为上下文数据的carrier(载体)。

span = # 从请求头中获取当前的追踪状态 `tracer.inject(span, opentracing.Format.HTTP_HEADERS, request.headers)`

完成并关闭span

当你收到相应后，你完成并关闭span，标志着客户端调用结束。和服务端一样，如果完成这个操作取决于你在客户端如何处理请求和响应。如果你存在过滤器(filter)，你可以这样处理：

```
def filter(request, response):
    span = # start span from the current trace state
    tracer.inject(span, opentracing.Format.HTTP_HEADERS, request.headers)
    response = send_request(request)
    if response.error:
       span.set_tag(opentracing., true)
    span.finish()
```

否则，如果你的请求和相应是分开处理的，你可能需要扩展你的tracer，包含请求和span的映射关系。参考实现如下：

```
def process_request(request):
    span = # start span from the current trace state
    tracer.inject(span. opentracing.Format.HTTP_HEADERS, request.headers)
    tracer.add_client_span(request, span)
def process_response(request, response):
    tracer.finish_client_span(request)
```

Closing Remarks

如果你想突显你的项目是监控OpenTracing标准的，你可以在你的所需的地方，使用我们的GitHub图标。也可以将你的项目链接，添加到OpenTracing官方网站上。

```
 [![OpenTracing Badge]
(https://github.com/opentracing/contrib/blob/master/badge/OpenTracing-enabled-blue.png)]
(http://opentracing.io)
```

一旦你发布你的实现，请发邮件到community@opentracing.io，并说明你的实现细节(platform, description, github username 平台、描述和github账号)，我们将在opentracing-contrib下给你开一个子项目，其他人可以在这个子项目里面发现并使用你的集成方案。你也可以在这里，找到多种开源项目集成OpenTracing的实例。

如果你想了解更多关于OpenTracing的信息，欢迎加入mailing list 或 Gitter。

作者和贡献者

(按照字母表排序)

- @adriancole (Adrian Cole)
- @bcronin (Ben Cronin)
- @bensigelman (Ben Sigelman)
- @bg451 (Brandon Gonzalez)
- @dkuebric (Dan Kuebrich)
- @michaelsembwever (mck)
- @pritianka (Priyanka Sharma)
- @slimsag (Stephen Gutekanst)
- @tschottdorf (Tobias Schottdorf)
- @wu-sheng (Wu Sheng)
- @yurishkuro (Yuri Shkuro)

Supported Tracer Implementations, 符合标准的项目

Zipkin and Jaeger

Zipkin 和 Jaeger 在多种语言环境中支持OpenTracing. 另外，还有一个实验性的项目bridge from Brave (Zipkin Java) instrumentation to OpenTracing。 其他相关链接zipkin-go-opentracing, jaeger-client-java, jaeger-client-go, jaeger-client-python, and jaeger-client-node。

Appdash

Appdash (background reading) 是一个来源自sourcegraph的轻量级，基于Golang的分布式追踪系统。有一个兼容OpenTracing的追踪系统实现，使用Appdash作为后端，通过绑定Appdash到OpenTracing监控上，应用系统就可以轻松实现监控：

```
import (
    "github.com/sourcegraph/appdash"
    appdashtracer "github.com/sourcegraph/appdash/opentracing"
)

func main() {
    // Initialization with a local collector:
    collector := appdash.NewLocalCollector(myAppdashStore)
    chunkedCollector := appdash.NewChunkedCollector(collector)
    tracer := appdashtracer.NewTracer(chunkedCollector)

    // Initialization with a remote collector:
    collector := appdash.NewRemoteCollector("localhost:8700")
    tracer := appdashtracer.NewTracer(collector)
}
```

查看更多信息，可以查阅 the godocs.

LightStep

LightStep 用来在生产环境运行，使用本地化，符合OpenTracing标准的tracer运行一个私有的测试版本。兼容OpenTracing的LightStep Tracers所支持的语言包括Go, Python, Javasrcipt, Objective-C, Java, PHP, Ruby, and C++.