# zipkin简介 - xing苦独 - 博客园

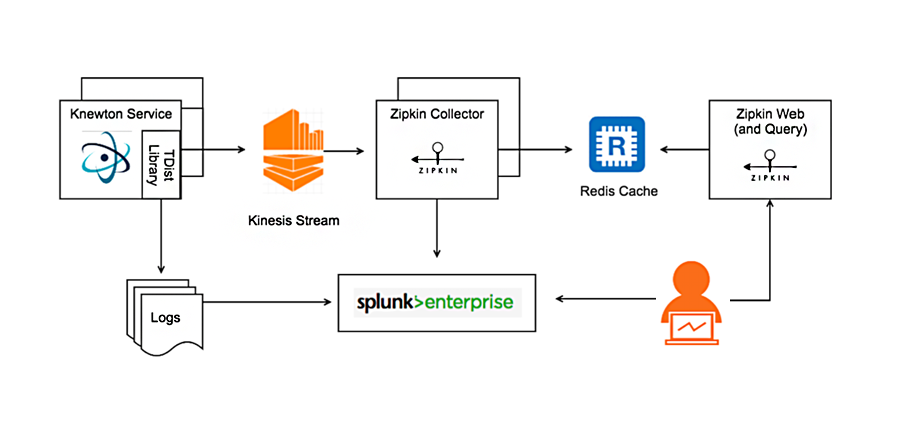
http://www.cnblogs.com/yuhuihong19941210/articles/6250495.html

文章翻译自英文原文：[Distributed Tracing: Design and Architecture](https://tech.knewton.com/blog/2016/04/distributed-tracing-design-architecture/)

[先前的博客公告](http://tech.knewton.com/blog/2016/04/distributed-tracing-motivation/)中讨论过为什么Knewton需要一个分布式追踪系统，并且数值可以被添加到一个公司中。这个章节将会更加深入探讨技术细节，我们如何实施分布式追踪系统的。

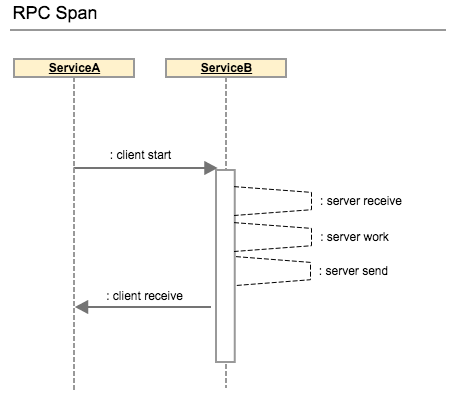
总体结构与追踪数据管理

　　我们的方案分为两大部分：所有服务集成到追踪库中，分配一个内存块来存储与查看追踪数据。我们选择[Zipkin](https://github.com/openzipkin/zipkin)，在Twitter开发的一个可扩展的开源追踪框架，用于存储与查看追踪数据。Zipkin通常以Finagle对的形式出现，但是，像上一节提及的一样，我们排除了与我们现有基础设施冲突的并发症。Knewton构建追踪库，称为TDist，从地面起，开始作为公司“黑客日”的实验。

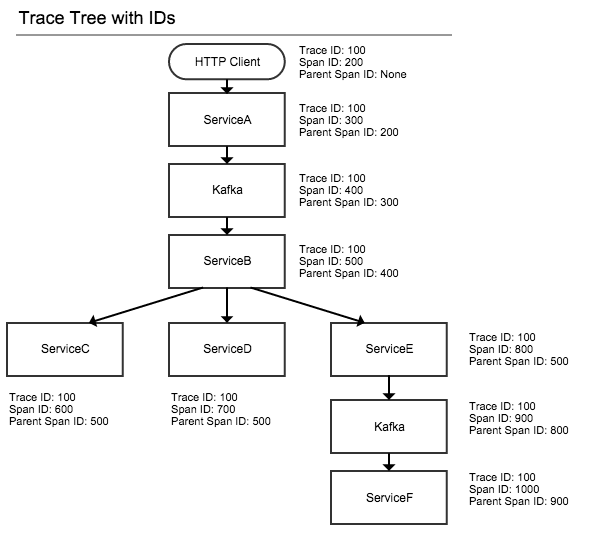


追踪数据模型

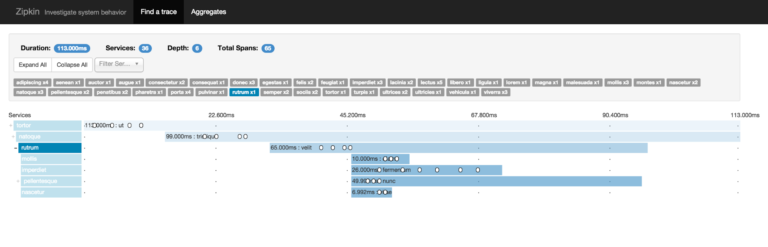
　　就我们的方案而言，我们选择使用Zipkin来匹配数据模型，轮流从[Dapper](http://static.googleusercontent.com/media/research.google.com/en/archive/papers/dapper-2010-1.pdf)大量借入。一个追踪树由一系列的跨度组成。跨度代表一个特殊的呼叫从服务器接收开始，到服务器发送，最后是客户端接收。举个例子，在服务器A和服务器B之间的呼叫与响应将会作为一个简单的跨度：



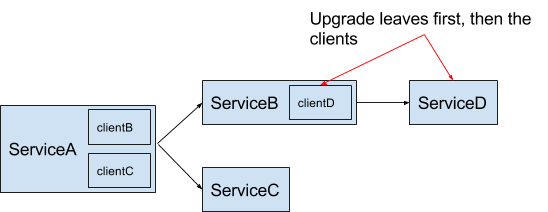
每一个跨度(span)有三个ID：  
• Trace ID：一个轨迹中所有的跨度(span)共享同一个Trace ID。  
• Span ID：用以标示不同的跨度(span)。Span ID与Trace ID不一定相同。  
• Parent Span ID: 只有子跨度持有这个ID，根跨度没有Parent Span ID。  
下面的图展示了在一个树结构的调用中，上面三个ID是如何应用的。注意在整个树结构中Trace ID是一致的。



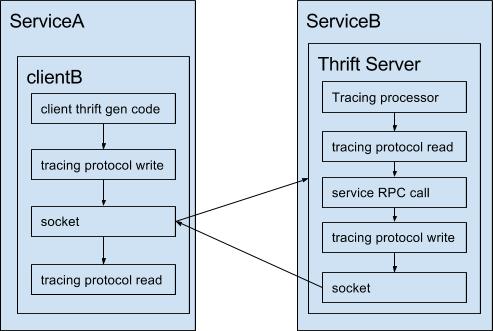
更多详情, 请参见 [Dapper paper](http://static.googleusercontent.com/media/research.google.com/en/archive/papers/dapper-2010-1.pdf).  
  
**TDist**  
TDist是Knewton开发的一个Java库。利用该库我们可以追踪所有的应用. TDist 目前支持[Thrift](https://thrift.apache.org/), HTTP, and Kafka, 还可以用来追踪使用了注解(参见[Guice](https://github.com/google/guice))的方法的调用。  
对每一个线程服务或者对另外一个服务发起的请求都分配一个跨度(span)，类库会在后台对跨度进行传播和更新。收到一个请求(或者即将发出一个请求)，追踪数据会被添加到一个内部队列中，DataManager将TraceID修改添加到处理请求的线程的名称中。工作线程消费队列，然后将追踪数据发布到追踪消息总线。Java ThreadLocal可以很方便的存储和读取线程范围内的全局变量，我们在DataManager中使用了这种方法。  
通常，线程将远程调用或者报告回父线程这样的实际的工作转嫁给其他线程来做。因此，我们也实现了线程factory和executor，这样就知道如何检索父线程的追踪数据，并将其分配给子线程，从而使得子线程也可以追踪。  
  
**Zipkin**  
追踪数据一旦经过TDist到达追踪消息总线，Zipkin基础设施会处理剩下的流程。多收集器实例，从消息总线中消费，存储追踪数据中每个记录。一个分离的查询集合与web服务，Zipkin的部分源代码，为了追踪依次查询数据库。我们为了使得事情变得简单，决定参与查询和web服务，并且也因为这种组合服务是内部的，并且有可预测的交通模式。但是，收集器是从查询与web服务中分离的，因为越多Knewton服务集成到收集器，越多追踪数据需要处理。  
  
**Zipkin UI**  
盒子外部，Zipkin在整个服务中提供一套简单的UI给视图追踪。当在所有服务中，非常容易地打印视图日志用于一个特殊的追踪ID号，Zipkin UI在每次调用的持续时长中提供一个总体视图，不需要查询数百个日志语句。在一个特定的周期时间内，它也是一个有效的方式来辨别最大的或者最慢的追踪。在发现这些异常值的情况下，允许我们标志出哪里重复调用其他服务，为了总体调用链而放慢我们的SLA。以下是Zipkin UI中的追踪截图：



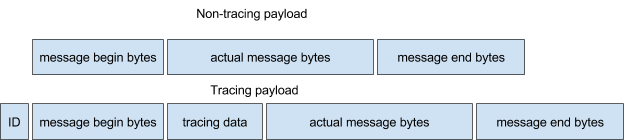
　　当然，UI并不会撤销。虽然它很容易看清楚个人痕迹，我们发现Zipkin UI 缺乏检查汇总数据。比如说，目前还没有方法获取总的时间信息或者总的数据，称之为端点，服务等等。  
在整个发展过程中，推出Zipkin基础设施，我们对Zipkin的开源做出了些许贡献，感谢它的活跃以及成长社区的支持。  
  
**Splunk**  
　　正如上面所提到，当前处理请求的线程名也会变动，其Trace ID会追加在上面。因为这样，我们在需要特定请求的时候，才能从所有启用追踪的服务上查询日志。这使得调试更加方便，同时事实证明其用于事后分析、日志聚合、独立问题的调试及解释平台的异常行为时也比较有用！  
  
**Thrift**  
　　Thrift是一个用于构建可拓展服务的跨平台的RPC框架。在Thrift中，用户可以定义一个服务、数据类型的规则，Thrift就会在许多不同的语言中编译其规则，这时用户就可以用想要的开发语言实现所生成的服务接口。Thrift同时自动生成客户端代码及用户为服务所定义的数据结构。  
在Knewton中Thrift是服务之间使用最普遍使用的RPC框架，我们服务的大多数通过使用此框架进行通信，所以在维护后端兼容性的时候支持它，对于此项目的成功性而言有着重大的影响。更准确的说，我们想让未启用追踪的服务能与启用追踪的服务通信。  
当我们开始研究给Thrift添加追踪支持的时候，我们与不同的两个方式进行实验。第一种方式涉及到一个修改过的Thrift编译器，而第二种涉及到修改后的序列协议及服务处理器。两种方法都有其优缺点。  
  
**自定义编译器**  
　　在这个方法中，我们体验修改C++简易编译器来生成额外的服务接口，可以传递追踪数据给用户。可能最著名例子就是Scrooge，修改简易的编译器并不罕见。修改过的编译器的其中一个优势是，客户端在它们代码中交换较少的类实现，由于在生成的代码中支持追踪。客户端也可以获得来自服务接口的追踪数据作为参考。  
虽然我们没有检测，我们也可以认为这个方法将会更快速地受到应用，由于只有较少的类调用追踪实现。但是，我们我么将会重编译我们所有的简易代码，偏离开源版本，使得它在将来更难升级。我们也将会认识到，允许用户访问追踪数据将缺乏渴望或者安全，并且数据管理可以更好地保证TDist的一致性。  
  
**自定义协议与服务处理器**  
　　最终我们应用这个方法到生产中。并不会维持一个自定义编译器来大量降低我们开发成本。自定义协议与服务接口的最大缺陷是，我们不得不升级来节俭0.9.0（从0.7.0），利用一些特征将会使得它更加容易插入我们自定义协议与处理器的追踪组件中。升级需要许多组织的协调。令人庆幸的是，更新的简易版本是向后兼容旧版本的，我们可以在TDist工作，当Knewton服务被更新到新版本时。但是，在我们可以开始用我们的分布式追踪方案来集成它们时，我们仍然不得不释放所有的Knewton服务。  
  
**升级 Thrift**  
　　一般来说通过依赖管理工具升级依赖库相对比较容易，但如果是那些类似Thrift的RPC框架，或者一些有很深调用链的SOA框架，问题就会复杂很多。一个典型的服务通常会同时包含服务端和客户端的代码，而服务端的代码往往会依赖其他的一些客户端的依赖库。所以，升级的时候需要从服务调用树的叶子节点开始向上逐级升级，以避免服务调用的兼容性问题。因为服务提供方可能并不知道调用方是否可以检测出那些附加的追踪数据。



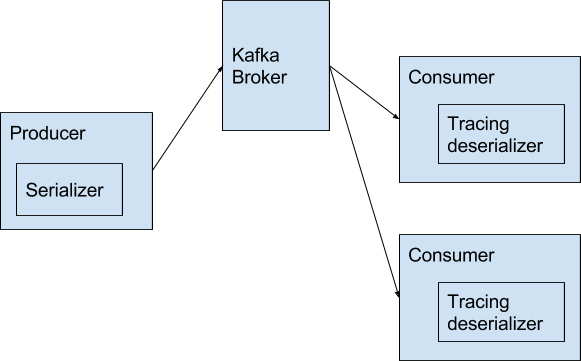
　　另外一个障碍是，一些服务会依赖Thrift 0.7.0（译者注：上文谈到需要升级为0.9.0），比如Cassandra客户端依赖[Astyanax](https://github.com/Netflix/astyanax)，[Astyanax](https://github.com/Netflix/astyanax)依赖的一些第三方依赖库会反过来依赖Thrift 0.7.0。对于Astyanax，我们不得不通过[Maven](https://maven.apache.org/)将依赖的JAR包屏蔽（shade ）并且修改包名来避免新旧版本Thrift库之间的冲突。整个升级过程必须迅速，并且没有停机时间。为了不让升级过程给Knewton的其他团队带来额外成本，分布式追踪小组不得不实施并推动整个变更过程。  
  
追踪简约组件：他是如何工作的  
　　我们的简约方案由自定义、向后兼容的协议与自定义服务处理器组成，提取跟踪数据，放在路由到相应的RPC调用。我们的协议基本上在每个消息头写入追踪数据。当RPC调用到达服务器，处理器将会识别并且标记呼入调用是否有追踪数据，因此它可以恰到好处地响应。追踪数据的调用也从中获取响应，来自非集成服务的请求不会携带没有响应的追踪数据。这使得追踪协议向后兼容，因为服务器传出的协议不会写追踪数据，如果指示不是出自处理器所服务的请求。一个追踪协议可以检测有效载荷是否包含追踪数据基于前面几个字节。简约追加协议ID到协议头，假如读取协议时发现前面几个字节不显示追踪数据，缓存与有效载荷的存在作为一个非追踪载荷来重读。当正在读取一个消息时，协议将会提取追踪数据，并且使用数据管理器来保存它们至本地线程，用于RPC呼入调用的线程服务。假如该线程额外调用于其他服务的下游，追踪数据将通过TDist从数据管理器自动地被提取出来，并且添加到外部消息。

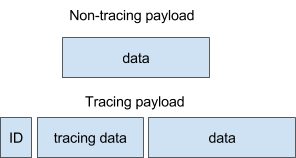


下面是一个图解，展示如何修改载荷来添加追踪数据：



Kafka消息追踪支持  
　　当我们为kafka提供消息追踪支持时，我们希望把Kafka服务（也被称为brokers），做为一个黑箱看待。换言之，我们希望brokers不需要知道消息是否被消费（即brokers不需要知道消息是否通过它发送给消费者）因此我们不需要修改Kafka的源码。我们采用与RPC 服务类似的处理方式，在升级生产者之前先升级消费者。消费者反向兼容并能检查到一个包含追踪信息的消息，以之前Thrift 协议描述的方式反序列化内容。为了能够实现上述方式，我们需要客户端封装他们在追踪系统中使用的序列化/反序列化实现，用于不包含追踪信息的消息和包含追踪信息的消息之间的读写转换。





HTTP 请求追踪支持  
　　在Knewton内部的一些基础构件中所有对外开放的节点都是基于HTTP的，我们需要以一种简便的方式在HTTP请求中插入需要携带的追踪信息。  
这个实现起来很简单，因为HTTP请求支持在消息头中放入任意数据。根据[rfc2047](http://www.ietf.org/rfc/rfc2047.txt)的第5章节中的内容，唯一的参考是在放置自定义的消息头需要为他们加入前缀'X-'。  
我们保持Zipkin传统并且使用一下标题传播信息：  
• X-B3-TraceId  
• X-B3-SpanId  
• X-B3-ParentSpanId  
  
　　Knewton 的服务主要使用[Jetty HTTP Server](http://eclipse.org/jetty/)和 [Apache HTTP Client](https://httpd.apache.org/)。所有依据这两种中间件构建的项目都能以便捷的方式实现对HTTP消息头的操作。  
Jetty 服务请求被路由到一个 Servlet。作为这个路由的一部分， Jetty 允许请求和回应通过一系列过滤。我们觉得处理追踪数据，这是最理想的。当任何传入请求附带跟踪数据头时，我们构造的跨数据会提交到 DataManager 。  
与 Apache HTTP 客户端一起，我们使用一个 [HttpRequestInterceptor](https://hc.apache.org/httpcomponents-core-ga/httpcore/apidocs/org/apache/http/HttpRequestInterceptor.html) 和 [HttpResponseInterceptor](https://hc.apache.org/httpcomponents-core-ga/httpcore/apidocs/org/apache/http/HttpResponseInterceptor.html)，它被设计成与头部内容能交互，并能修改他们。这些拦截器使用 DataManager， 并能从头中读出追踪数据，反之亦然。  
  
Guice  
　　大多数人不熟悉 [Guice](https://github.com/google/guice) ，这是一个谷歌开发的依赖关系管理框架。TDist 集成到我们现有的服务模块，那么我们的客户端将更简洁、更少出错，我们依靠的就是 Guice ，并且实现了多个模块可以让我们的客户端更易于安装。Guice 处理依赖于在对象实例化期间注入，在交换接口也能更简洁。如果通过这篇文章你已经开始思考集成 TDist ，那么听起来很复杂。很多时候，我们的客户端需要安装附加的 Guice 模块，这些模块将绑定到我们的追踪实现上，实现现有的 Thrift 接口。这也意味着，我们的客户端不能实例化任何我们追踪启动的构造。无论什么时候，一个 TDist 客户端忽略绑定一些东西，Guice 都会在编译时通知我们的客户端。我们把很多心思放在如何制定我们的 Guice 模块层次结构上，因此 TDist 不会与我们的客户端冲突，我们都很小心，无论什么时候，我们都必须暴露元素到外部世界。  
  
追踪消息总线  
　　所有我们的客户服务在使用之前都会放置追踪数据，追踪消息总线是通过 Zipkin 收集器来持续收集的。 我们的两个选项是 Kafka 和 [Kinesis](https://aws.amazon.com/kinesis/)，不过最终我们选择了 Kinesis 。我们考虑到 Kafka 是因为 Knewton 已经稳定部署 Kafka 很多年。在那时，我们的 Kafka 集群一直使用我们的子事件总线，在生产环境中每秒产生超过300条消息。我们初步估计将超过 400,000 条的跟踪信息，每秒只有部分进行集成。生产系统与仪表数据使我们紧张。Kinesis 似乎是一个有吸引力的替代，它将我们从 Kafka 服务器分离，这只是生产数据，而不是仪表上的数据。在实现的时候，Kinesis 是一个新的 AWS 服务，我们对它很熟悉。它的价格，吞吐能力，不用太多维护，这些促成了我们达成一致。总的来说，我们已经满意它的性能和稳定性。它没有 Kafka 那么快，但是 Kafka 数据产生的性质，它从产生到摄入 SLA 甚至要几分钟。自从我们部署追踪消息总线到生产环境，我们也很能容易扩展大量 Kinesis， 且不会引起任何宕机。  
  
追踪数据的存储  
　　所有我们追踪的数据都会被放在追踪数据存储。数据放在那里是有一个配置时间的，并且由 Zipkin 查询服务显示在 UI 上。Zipkin 提供了大量开箱即用的数据存储，包括 Cassandra， Redis，MongoDB， Postgres 和 MySQL。我们对 Cassandra 和 DynamoDB 做过实验，这主要是因为我们在 Knewton 中获得的习以为常的知识，最终我们还是选择了亚马逊的 [Elasticache Redis](https://aws.amazon.com/elasticache/) 。下面这些是我们做出这个决定的最重要原因。  
• 花在生产上的时间，我们还没铺开就要交付了，并且，我们还要维护一个新的集群  
• 成本  
• 与 Zipkin 集成更简单，代码更少  
• 在数据上支持TTLs  
  
  
**结 论**  
　　现在在 Knewton， 我们的追踪解决方案在整个环境中已经运行好多个月了。目前它已经被证明是非常有价值的。我们只有在开始擦除那个面的时候，我们才可以追踪并且收集时间数据。我们有很多有趣的实现，并且在 Knewton 交付，之后，我们就理解了这个数据的价值。

分类: [分布式追踪](http://www.cnblogs.com/yuhuihong19941210/category/835774.html)

标签: [Trace](http://www.cnblogs.com/yuhuihong19941210/tag/Trace/), [Twitter Zipkin](http://www.cnblogs.com/yuhuihong19941210/tag/Twitter%20Zipkin/), [分布式服务](http://www.cnblogs.com/yuhuihong19941210/tag/%E5%88%86%E5%B8%83%E5%BC%8F%E6%9C%8D%E5%8A%A1/), [调用链](http://www.cnblogs.com/yuhuihong19941210/tag/%E8%B0%83%E7%94%A8%E9%93%BE/), [追踪](http://www.cnblogs.com/yuhuihong19941210/tag/%E8%BF%BD%E8%B8%AA/), [官方](http://www.cnblogs.com/yuhuihong19941210/tag/%E5%AE%98%E6%96%B9/)