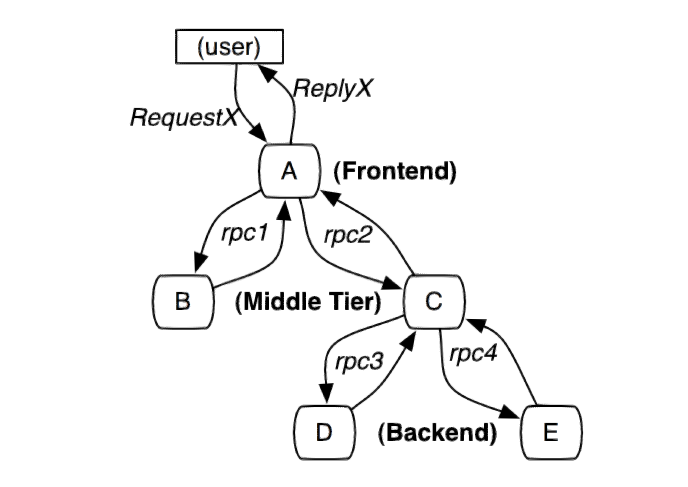
分布式跟踪系统

当前互联网服务通常由复杂、大规模分布式集群来实现，其应用构建在不同软件组件模块上，这些模块可能由不同的团队、使用不同的编程语言来实现，有可能分布在几千台服务器上，横跨多个不同的数据中心，因此，需要一些工具来理解系统行为及分析性能问题。Google在2010年发布内部使用的分布式跟踪系统Dapper，目前社区有多个比较成熟的实现，例如Zipkin、Pinpoint、HTrace等，国内的分布式跟踪系统有美团点评的CAT、阿里的鹰眼Tracing等。

分布式跟踪系统收集分布式系统复杂的行为信息，包括服务的调用链路，例如前端服务可能对上百台查询服务器发起一个Web查询，这个查询可能发送到多个子系统，分别处理广告、拼写检查、查询图片新闻等特殊结果，最后将结构进行汇总到页面上，这个全局搜索会调用上千台服务器，涉及各种服务。用户对搜索的耗时是敏感的，任何一个子系统的低效都会导致最终搜索延迟，如果没有分布式跟踪系统，工程师无法准确快速定位到延迟由哪些服务造成的。首先，这个工程师可能无法准确的定位到这次全局搜索是调用了哪些服务，因为新的服务、乃至服务上的某个片段，都有可能在任何时间上过线或修改过，有可能是面向用户功能，也有可能是一些例如针对性能或安全认证方面的功能改进。其次，不能苛求这个工程师对所有参与这次全局搜索的服务都了如指掌，每一个服务都有可能是由不同的团队开发或维护的。再次，这些暴露出来的服务或服务器有可能同时还被其他客户端使用着，所以这次全局搜索的性能问题甚至有可能是由其他应用造成的。举个例子，一个后台服务可能要应付各种各样的请求类型，而一个使用效率很高的存储系统，比如Bigtable，有可能正被反复读写着，因为上面跑着各种各样的应用。对于分布式跟踪系统要实现全方位、持续的监控，因此要有以下三个具体的设计目标：

* 低消耗、高稳定，跟踪系统对在线服务的影响要足够小，不影响线上服务器的性能，不能有复杂的逻辑和外部依赖
* 对应用透明、低侵入，系统不需要改造就可以接入分布式跟踪系统
* 可扩展，随着分布式系统的增多，压力不断增长，分布式跟踪系统需要动态扩展来支撑不断接入的业务系统

下图是上例中的简单描述：



路径由用户的X请求发起，穿过一个简单的服务器系统（前端A，中间层B/C及两个端点D/E），用字母标识的节点代表分布式系统中的不同处理过程。用户发起请求时，首先到达前端，然后发送两个RPC到服务器B和C，B马上返回结果，但是C需要和后端D/E交互之后再返回给A，由A来响应最初的请求。

分布式跟踪系统需要记录在一次特定的请求后系统中完成的所有工作的信息，对于服务器上的每次请求都可以用跟踪标识符（Message Identifiers）和时间戳（Timestamped Events）来抽象。在生产环境中，所有的应用程序都使用相同的线程模型、控制流和RPC系统，因此通过将代码植入到通用组件库中，可以实现监控系统对开发人员有效透明。

# Span和跟踪树

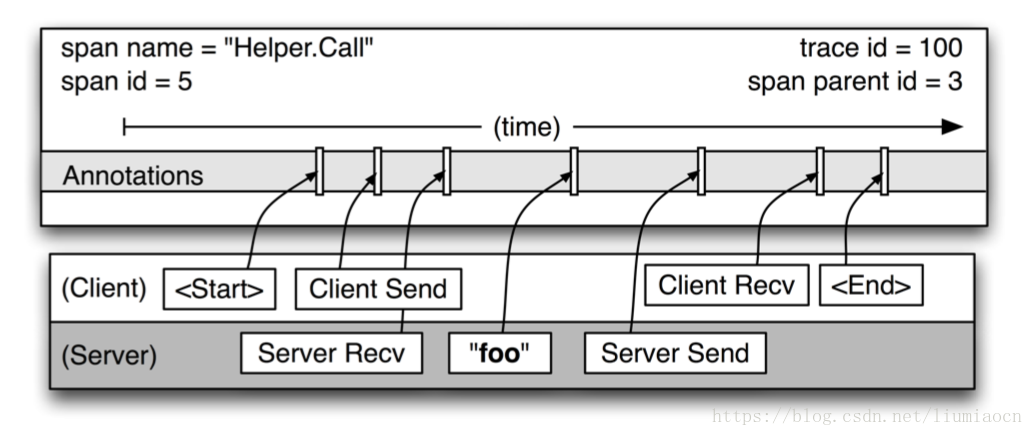
在Dapper中，将跟踪模型抽象成树形结构、Span和Annotation。Span是分布式跟踪的基本工作单元，一次链路调用（Rpc、DB连接等）可以对应Span，可以包括：

* 名称，Span名称
* SpanId，记录Span的id，全局唯一的64位整数标识
* 父SpanId，父Span的Id，用于描述跟踪树结构
* 事件信息，cs/cr/sr/ss四种事件类型，不同事件类型对应不同时间戳，根据时间戳可以计算出不同阶段的耗时信息

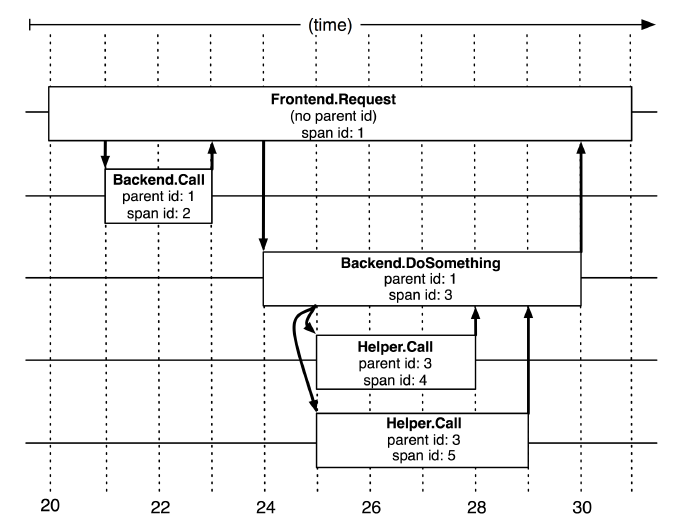
|  |  |  |
| --- | --- | --- |
| 缩写 | 全称 | 说明 |
| CS | Client send | 客户端/消费者发起请求 |
| CR | Client receive | 客户端/消费者接收到应答 |
| SR | Server Receive | 服务端/生产者接收到请求 |
| SS | Server Send | 服务端/生产者发送应答 |

* Annotation，事件信息在annotation中存放，自定义信息也可以与之关联

Span的概念如下图所示：



在分布式跟踪系统中，一个请求可和服务调用关联，每次服务调用和Span进行关联，而Span之间通过spanid进行连接，这样组成一个树形结构，即所谓的跟踪树。跟踪树体现了某一请求的服务调用链的状况，如下图所示：



在树形结构中，树节点是整个架构的基本单元，节点之间的连线表示Span和父Span直接的关系。虽然Span在日志文件中只是简单的代表Span的开始和结束时间，但是在树形结构中相对独立。上例中的请求树结构，可以表示成：

|  |  |  |  |
| --- | --- | --- | --- |
| 层次 | 服务名称 | 父Span | 调用顺序 |
| 前端 | Frontend：A | 无 | 1 |
| 中间 | MiddleTier：B | A | 2 |
| 中间 | MiddleTier：C | A | 3 |
| 后端 | Backend：D | C | 4 |
| 后端 | Backend：E | C | 5 |

其调用顺序则是：A->B->C->D->E。

# 植入点

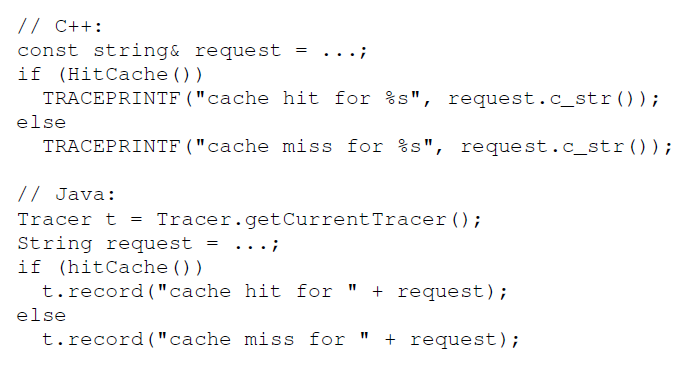
Dapper可以对应用近乎零成本的方式侵入分布式系统进行跟踪，其完全依赖于基于少量通用组件库的改造：

* 当一个线程在处理跟踪控制路径的过程中，Dapper把这次跟踪的上下文在ThreadLocal中存储。追踪上下文是小而且容易复制的容器，其中承载了Scan属性，比如跟踪ID和Span ID
* 计算过程是异步的，大多是Google开发者通过线程池或者其他执行器，使用通用控制流库来回调。Dapper确保这样的回调可以存储这次跟踪的上下文，而当回调函数被触发时，这次跟踪的上下文会与适当的线程管理。这种方式下，Dapper可以使用trace ID和Span ID来辅助构建异步调用的路径
* 几乎所有的Google进程间通信是建立在C++和JAVA开发的RPC上，把跟踪植入这些框架来定义RPC中所有Span。Span的ID和跟踪ID会从客户端发送给服务端。

# Annotation

上述植入点可以推导出复杂的分布式系统的跟踪细节，使得Dapper核心功能在不改变引用的情况下可用。然后Dapper还允许应用程序开发人员使用Dapper跟踪的过程中添加额外的信息，以监控更高级别的系统行为，或者帮助调试问题。

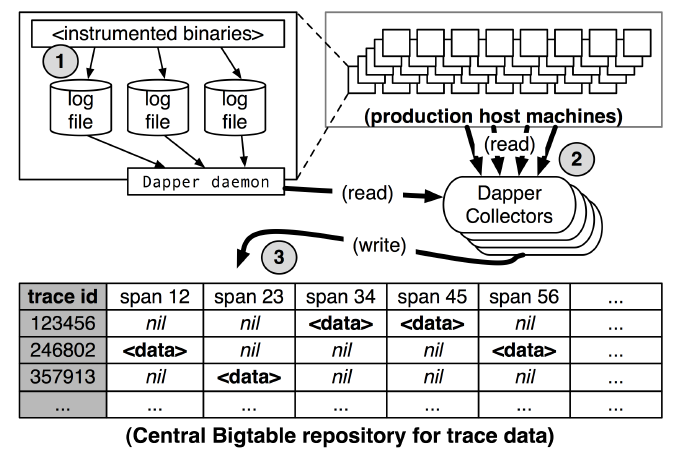
Dapper允许用户通过一个简单的API来定义时间戳的Annotation，核心的实例代码植入如下所示：



在Annotation中可以添加任意内容，但是应用程序级别的Annotation不能替代用户表示Span结构的信息和记录RPC相关信息，这提供给开发人员更强的跟踪能力，如持续的计数器、二进制消息记录和一个进程上跑着任意的用户数据。Dapper支持KV映射的Annotation，在追踪上下文中定义某个特定应用程序的相关类型。

# 采样

低损耗是Dapper的一个关键设计目标，某些类型的服务对植入带来的性能损耗比较敏感，因此Dapper的收集工作要对基本组件的性能损耗限制尽可能小，可以通过采样率来实现，如下：



在上图中，Dapper将跟踪记录和收集的过程分为三个阶段:

1. 将Span数据写入到本地文件中
2. Dapper守护进程和收集组件将这些数据从生产环境的主机中拉出来
3. 将数据写到Dapper的Bigtable

Dapper还提供一个API来简化对仓库中跟踪数据的访问。

https://blog.csdn.net/apei830/article/details/78722168

https://blog.csdn.net/manzhizhen/article/details/52811600

http://bigbully.github.io/Dapper-translation/