任务跟踪系统开发记录

# 需求分析

## 上下文关系的重建能力

一个单线程的单机程序，所有的步骤都是有序进行的，日志的分析简单直观：

2021-02-25 10:32:13,081 - INFO: test\_main\_single\_machine\_single\_thread start

2021-02-25 10:32:13,081 - INFO: normal\_task\_with\_a\_loop start

2021-02-25 10:32:13,081 - INFO: normal\_task\_sleeps start

2021-02-25 10:32:15,082 - INFO: normal\_task\_sleeps end

2021-02-25 10:32:15,082 - INFO: normal\_task\_sleeps start

2021-02-25 10:32:17,084 - INFO: normal\_task\_sleeps end

2021-02-25 10:32:17,084 - INFO: normal\_task\_sleeps start

2021-02-25 10:32:19,084 - INFO: normal\_task\_sleeps end

2021-02-25 10:32:19,084 - INFO: normal\_task\_with\_a\_loop end

2021-02-25 10:32:19,085 - INFO: normal\_task\_throws\_exception start

2021-02-25 10:32:20,086 - INFO: normal\_task\_throws\_exception error

2021-02-25 10:32:20,087 - INFO: test\_main\_single\_machine\_single\_thread error

而多线程的单机程序，并发子线程都在往一个日志文件里面输出日志，过程调用的先后顺序和主从关系很难从日志的先后顺序中反映出来，增加了现场还原和问题定位的难度：

2021-02-25 10:50:48,435 - INFO: test\_main\_single\_machine\_multiple\_threads start  
2021-02-25 10:50:48,435 - INFO: normal\_task\_starts\_multiple\_threads start  
2021-02-25 10:50:48,436 - INFO: E start  
2021-02-25 10:50:48,437 - INFO: E start  
2021-02-25 10:50:48,437 - INFO: normal\_task\_throws\_exception start  
2021-02-25 10:50:48,437 - INFO: E error  
2021-02-25 10:50:48,437 - INFO: E start  
2021-02-25 10:50:48,438 - INFO: normal\_task\_throws\_exception start  
2021-02-25 10:50:49,438 - INFO: normal\_task\_throws\_exception error  
2021-02-25 10:50:49,438 - INFO: E error  
2021-02-25 10:50:49,439 - INFO: normal\_task\_throws\_exception error  
2021-02-25 10:50:49,439 - INFO: E error  
2021-02-25 10:50:49,440 - INFO: normal\_task\_starts\_multiple\_threads end  
2021-02-25 10:50:49,440 - INFO: normal\_task\_throws\_exception start  
2021-02-25 10:50:50,442 - INFO: normal\_task\_throws\_exception error  
2021-02-25 10:50:50,444 - INFO: test\_main\_single\_machine\_multiple\_threads error

多线程或者多进程的单机程序可以通过在日志中打印线程号和进程号的方式来重建上下文。

分布式系统则更加复杂：

* 日志不会输出到一起（elk）
* 日志与日志之间难以关联（除非对业务逻辑足够了解）
* 如果存在多级的远程调用，分析起来更加困难

导致的问题，如果有更深入的需求：

* 例如-希望在发生故障时，保存一些数据到磁盘，以便事后恢复线程重现bug；
* 例如-分析系统高故障率的模块，分析性能瓶颈。

现有系统在无法自动获取过程上下文的情况下，无法对这些繁琐操作做到去人工化

如何记录和汇总这个过程，并在分析过程中从日志中恢复这种上下文关系？

## 聚焦问题的能力

对原始的调用关系做一定程度的抽象，选取关注点，忽略不重要的细节

子系统在开发过程中依据各自的业务需求开发了自己日志输出模块，系统集成之后，又部署到集群，总的日志量会相当庞大，任务监控系统收集和分析所有的日志，可能会遇到性能瓶颈。

任务监控系统需要有可定制的筛选能力。

## 与子系统低耦合

* 低侵入性：子系统开发时不必关注跟踪系统的实现细节，子系统不必专门为跟踪系统做专门的源码适配。
* 高通用性：能够跟踪任何类型的分布式任务，跟踪系统不必专门为被监控系统做专门的适配。

# 第一个方案

最开始的想法是，既然函数调用关系很难直接从日志中分析得到，那能不能手动预先定义，然后任务跟踪日志的记录和解析都依据这个预定义的调用关系图来进行呢？



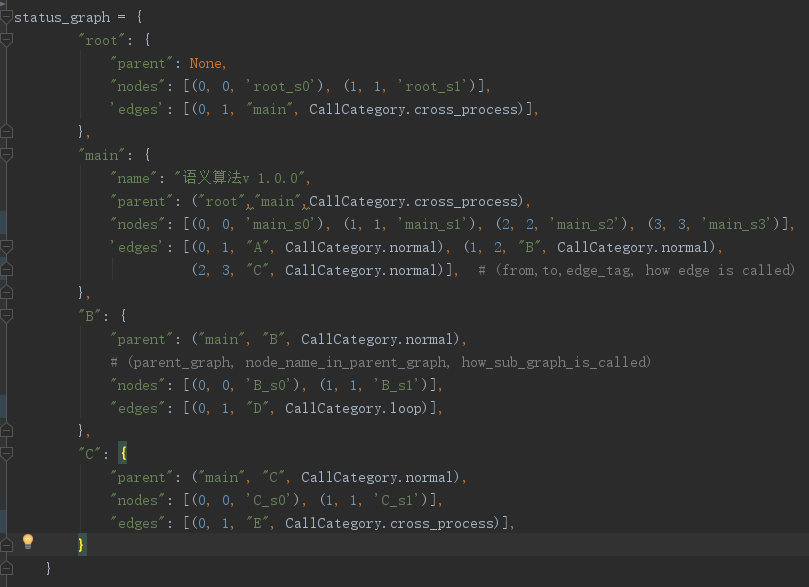
以下面的程序为例：



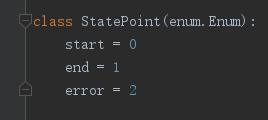
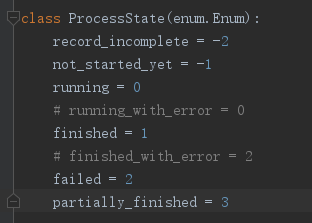


## 状态图表示

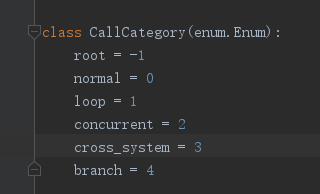
第一版的状态图设计目标是用系统的状态检查点为节点，状态之间的转移过程对应执行过程。记录状态点的结果上传到数据库，再分析执行过程的状态。



## 状态类型和过程结果类型

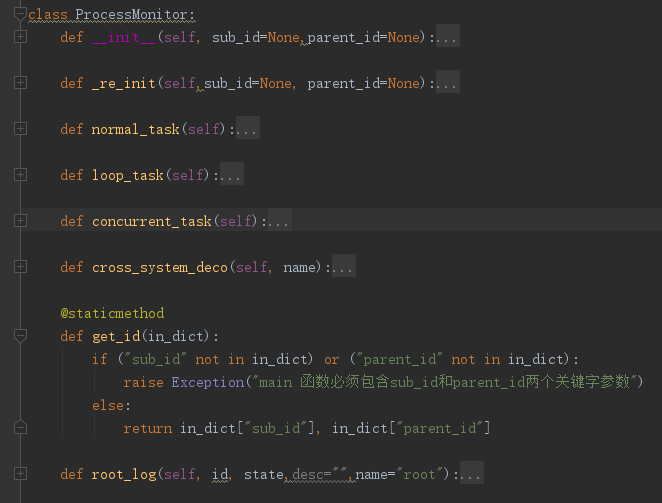
 

## 调用过程类型



## 状态上传装饰器及其使用

根据调用类型的不同写了不同的装饰器。



|  |  |
| --- | --- |
|  |  |
|  |

## 状态记录表结构



## 遇到的问题

* Json格式的状态转换图表达能力有限

算法开发者并不能预知一个子过程究竟会有多少次循环或者多少个并发子任务，因而只能从日志分析的结果中获取这部分信息。这导致原先设计的“预建状态图”的步骤不可实现。

而且，为了实现低侵入性，状态更新通过函数装饰器来完成，而不是在源码中加入检查点。这与方案的设计初衷——“用状态检查点做节点，过程为边”相违背，导致日志分析时需要额外做一些转换。

* 日志解析困难

日志根据被调用过程的类型不同，采用不同的方式展开。在日志分析阶段，节点的多样性和关联关系的多样性使得调用树的构造代码十分复杂且容易出错。

* 无法支持更深层次的调用关系

batch\_id和task\_id两个字段被用来将记录对应到一次子任务，index被用来区分这次子任务中的并发子任务，index的设计具有局部性，而batch \_id和task\_id这两个字段只能表达一级跨系统调用。这意味着如果子任务又有子任务，或者并发子任务又有并发子任务，就会没有合适的字段用来做关联。

# 第二个方案

因为以上方案的实施遇到了困难，所以重新设计了第二个方案。

## 表结构变更

|  |  |
| --- | --- |
|  | parent\_id 和sub\_id可以无限递归扩展。Sub\_id指当前过程的唯一性标识，不再是全局唯一的。Parent\_id是调用当前过程的过程的id。  parent\_id 和sub\_id的关联关系可以无限延申，递归地关联一颗调用树上任意两个存在调用关系的过程。  Batch\_id是根任务的批次。虽然上面两个id足以将一次任务中所有相关的日志都关联到一起了，但是还是需要这个字段来减少数据库查询的次数（一次查询一个任务的所有记录到内存中） |

## 记录过程上下文的数据结构

用树来表示调用关系明显具有更大的优势。



因为已经实现了过程id的全局唯一性，不再需要根据四个字段（批任务id，子任务id，索引号，过程标签）来定位一个子过程。

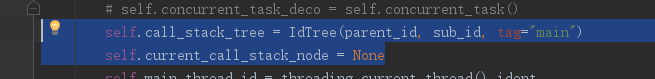
同时，随着调用关系的深入，树也可以无限生长，只要parent\_id能够传递到子过程，就不会丢失这种关联关系，调用上下文的记录和恢复就得到了保障。

## 调用树的构造

调用树的记录需要在调用过程中传递parent\_id，为被调用的过程生成全局唯一id。这个过程由针对不同场景的装饰器来实现。

* 单机同线程函数调用：

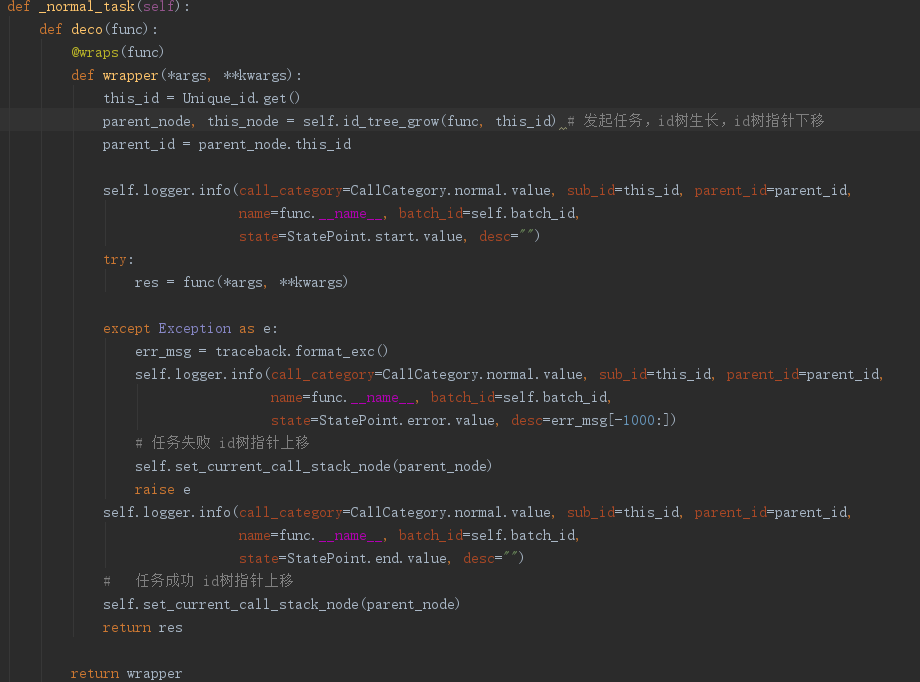
在装饰器的全局变量中维护一个调用树，加上一个指向当前调用树节点的指针；



每次发起一次调用时，在当前节点上创建一个子节点，同时指针指向这个节点；

完成这次调用之后，指针退回到当前节点。





以上是单机单线程程序的任务跟踪模型。分布式系统的调用关系中存在跨进程，跨线程的情况，无法让每个子过程都共享到这棵全局树。

但调用树模型没有问题，以此为基础，想到一些方法来屏蔽差异性，让分布式系统仿佛运行在单机版上一样。

* 跨机器（celery，微服务）、跨进程（进程池，subprocess启动脚本）

这些统统被处理为跨进程调用，需添加（cross\_process）装饰器；

装饰器会检查被装饰的函数是否收到了batch\_id和parent\_id两个参数，否则会抛异常；

这两个参数可以保证新进程中的调用树能够作为整个系统的调用树的一棵子树被创建，且能够被融合到系统调用树里面去。

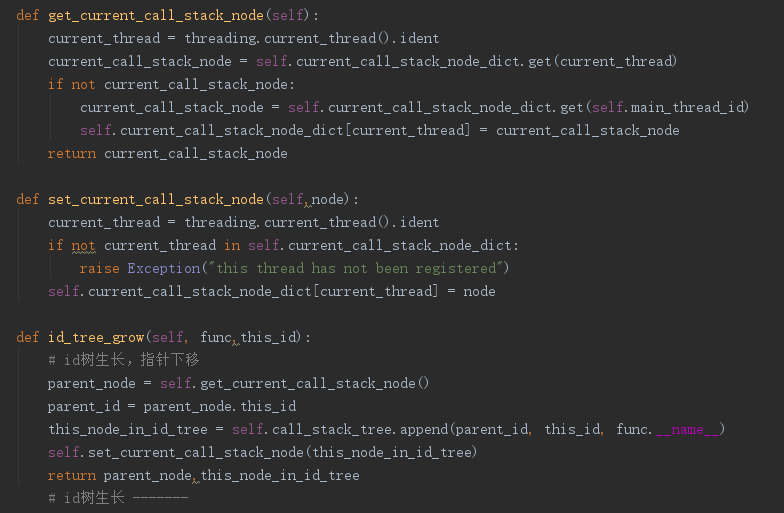
* 跨线程调用（线程池并发，手动多线程）

跨线程调用的所有子过程都能够共享到全局调用树，但是没法通过同一个调用树指针去更新自己的子节点。

于是为每个线程都创建一个独有的调用树指针，保存在一个全局的key=线程标识的字典中。



子线程通过这个字典来访问和维护各自的调用树指针。整棵树会在python解释器锁的保护下无竞争地在各个子线程的子树上生长。



## 日志上传

通过装饰器中的post请求来完成。Monitor后端有一个接口接收日志上传请求并存入postgres中。

## 日志分析

* 调用树重建+状态抽取

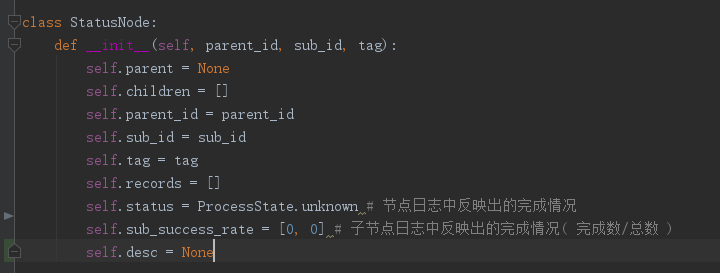
根据parent\_id和sub\_id，可以很容易地通过日志将调用树重建出来，不再需要子系统开发者手动编写状态转移图了。

能查到对应每一个树节点的任务记录，状态的更新也很简单了。

挂载了分析结果的调用树，后面称之为状态树。

* 任务状态描述

一个任务的状态由两个变量描述：



status表示这个节点是否正常执行完成，sub\_success\_rate则表示子任务完成情况：

* 若节点失败，则标记为 failed+红色；
* 若节点完成，且子任务完成率100%，则标记为 succeeded+绿色；
* 若节点完成，但是子任务完成率未达到100%，则标记为 succeeded+黄色；
* 其他的状态（执行中）都被标记为黄色。

## 前端分级展示状态树

根据焦点的不同，后端可以在不同的层级上展示状态树。





## 数据缓存

从数据库查询多条记录的速度很慢。

为了加快响应速度，后端设置了定时器，会定时取较新的若干batch的记录，保存在本地。

后端在响应前端查看任务状态的请求时，会直接从本地的文件中读取记录。