

## 30-如何在系统中实现延迟监控？

你好，我是蒋德钧。

我们知道，Redis的一个显著特征就是**能提供低延迟的数据访问**。而如果Redis在运行过程中变慢了，我们就需要有方法能监控到哪些命令执行变慢了。更进一步的需求，就是我们需要有方法监控到，是Redis运行过程中的哪些事件导致Redis变慢了。这样一来，我们就既可以检查这些慢命令，分析它们的操作类型和访问的数据量，进而提出应对方法，也可以检查监控记录的事件，分析事件发生的原因并提出应对方法。

那么，为了满足这些需求，我们就需要有一套监控框架，一方面能监测导致Redis变慢的事件，另一方面，能监控并记录变慢的命令。其实，这也是我们在开发后端系统时，经常会面临的一个运维开发需求，也就是**如何监控后端系统的运行状态**。

今天这节课，我就来带你了解Redis的延迟监控框架和慢命令日志的设计与实现。掌握今天的课程内容后，一方面，你可以把这套监控框架应用到实际的业务，而另一方面，你也可以参考Redis的实现，给自己的系统增加延迟监控功能。

下面，我们就先来看下Redis实现的延迟监控框架。

### 延迟监控框架的实现

实际上，Redis在运行过程中，以下表格中给出的几类事件都会导致Redis变慢，我们通常也把这些事件称为**延迟事件**。你可以参考表格中的这些事件类型，以及它们在源码中对应的事件名称。

事件类型	源码中对应的具体事件名称
命令事件	command, fast-command
AOF事件	aof-write-pending-fsync, aof-write-active-child, aof-write-alone, aof-fstat, aof-rewrite-diff-write, aof-rename
fork事件	fork
过期key事件	expire-cycle
缓存替换事件	eviction-del, eviction-cycle



那么针对这些事件，Redis实现了延迟监控框架，通过采样的方式来记录它们的执行情况。当需要排查问题时，延迟监控框架提供了**latency history命令**，以便运维人员检查这些事件。

下面，我们就来看下记录事件执行情况的数据结构。因为延迟监控框架是在[latency.h](#)和[latency.c](#)文件中实现的，你也可以在这两个文件中找到相应的数据结构和函数。

### 记录事件执行情况的数据结构

首先，Redis是使用了**latencySample结构体**，来记录延迟事件的采样时间和事件的实际执行时长，这个结

构体的代码如下所示：

```
struct latencySample {
    int32_t time; //事件的采样时间
    uint32_t latency; //事件的执行时长（以毫秒为单位）
};
```

而在latencySample这个结构体基础上，Redis又设计了**latencyTimeSeries结构体**，这个结构体使用了一个latencySample类型的数组，记录了针对某一类事件的一系列采样结果，这样就可以为分析Redis变慢提供更多的事件信息。

```
struct latencyTimeSeries {
    int idx; //采样事件数组的写入位置
    uint32_t max; //当前事件的最大延迟
    struct latencySample samples[LATENCY_TS_LEN]; //采样事件数组，记录LATENCY_TS_LEN个采样结果，LATENCY_TS_LEN默
};
```

另外，也因为延迟监控框架要记录的延迟事件有很多种，所以Redis还进一步设计了一个**哈希表 latency\_events**，作为全局变量server的一个成员变量，用来记录不同事件的采样结果数组，如下所示：

```
struct redisServer {
    ...
    dict *latency_events;
    ...
}
```

这个哈希表是在Redis server启动初始化的函数initServer中，通过调用latencyMonitorInit函数来完成创建的，如下所示：

```
void initServer(void) {
    ...
    latencyMonitorInit();
}

void latencyMonitorInit(void) {
    server.latency_events = dictCreate(&latencyTimeSeriesDictType, NULL);
}
```

好，了解了记录延迟事件的数据结构和初始化操作后，我们再来看下事件采样是如何实现的。

## 如何实现延迟事件的采样？

延迟事件的**采样函数是latencyAddSample**，它的函数原型如下所示。它的参数中包含了要记录的事件名称，这实际是对应了latency\_events哈希表中的一个哈希项。此外，它的参数中还包括该事件的执行时长。

```
void latencyAddSample(char *event, mstime_t latency)
```

latencyAddSample函数的执行逻辑并不复杂，主要可以分成三步。

**首先**，它会根据传入的事件名称，在latency\_events哈希表中查找该事件。如果该事件对应的哈希项还不存在，它就会在哈希表中加入该事件，如下所示：

```
//查找事件对应的哈希项
struct latencyTimeSeries *ts = dictFetchValue(server.latency_events,event);
...
if (ts == NULL) { //如果哈希项为空，就新建哈希项
    ts = zmalloc(sizeof(*ts));
    ts->idx = 0;
    ts->max = 0;
    memset(ts->samples,0,sizeof(ts->samples));
    dictAdd(server.latency_events,zstrdup(event),ts); //在哈希表中插入哈希项
}
```

**然后**，latencyAddSample函数会根据传入的事件执行时间，更新当前记录的该类事件的最大执行时间，如下所示：

```
if (latency > ts->max) ts->max = latency;
```

**最后**，latencyAddSample函数会实际记录当前的采样结果。

不过在这一步，如果它发现当前的采样结果，和前一个采样结果是在同一秒中获得的，并且如果当前采样结果的事件执行时长，大于前一个采样结果的话，那么latencyAddSample函数就会直接更新前一个采样结果中记录的执行时长了，而不是新插入一个采样结果。

否则的话，latencyAddSample函数才会新插入一个采样结果。这样设计的目的，也是为了避免在同一秒中记录过多的采样结果。

下面的代码展示了latencyAddSample函数实际记录采样结果的逻辑，你可以看下。

```
//获得同类事件的前一个采样结果
prev = (ts->idx + LATENCY_TS_LEN - 1) % LATENCY_TS_LEN;
if (ts->samples[prev].time == now) { //如果当前和前一个采样结果在同一秒中
    if (latency > ts->samples[prev].latency) //如果当前采用结果的执行时长大于前一个采样结果
        ts->samples[prev].latency = latency; //直接更新前一个采样结果的执行时长
```

```
        return;
    }
    //否则，新插入当前的采样结果
    ts->samples[ts->idx].time = time(NULL);
    ts->samples[ts->idx].latency = latency;
```

而在这里，你也要注意一点，就是latencyAddSample函数在记录采样结果时，**会重复使用采样结果数组latencyTimeSeries**。所以，如果采样结果数量超过数组默认大小时，旧的采样结果是会被覆盖掉的。如果你要记录更多的采样结果，就需要扩大latencyTimeSeries数组的长度。

那么，latencyAddSample函数是在什么时候调用进行采样的呢？

其实，latencyAddSample函数是被封装在了**latencyAddSampleIfNeeded函数**中。在latencyAddSampleIfNeeded函数中，它只会在事件执行时长超过latency-monitor-threshold配置项的值时，才调用latencyAddSample函数记录采样结果。你可以看看下面给出的latencyAddSampleIfNeeded函数定义。

```
#define latencyAddSampleIfNeeded(event,var) \
    if (server.latency_monitor_threshold && \
        (var) >= server.latency_monitor_threshold) \
        latencyAddSample((event),(var));
```

而latencyAddSampleIfNeeded函数，实际上会在刚才介绍的延迟事件发生时被调用。这里我来给你举两个例子。

比如，当Redis命令通过call函数（在server.c文件中）执行时，call函数就会调用latencyAddSampleIfNeeded函数进行采样，如下所示：

```
if (flags & CMD_CALL_SLOWLOG && c->cmd->proc != execCommand) {
    //根据命令数据结构中flags的CMD_FAST标记，决定当前是fast-command事件还是command事件
    char *latency_event = (c->cmd->flags & CMD_FAST) ?
        "fast-command" : "command";
    latencyAddSampleIfNeeded(latency_event,duration/1000);
    ...
}
```

再比如，当Redis调用flushAppendOnlyFile函数写AOF文件时，如果AOF文件刷盘的配置项是AOF\_FSYNC\_ALWAYS，那么flushAppendOnlyFile函数就会调用latencyAddSampleIfNeeded函数，记录aof-fsync-always延迟事件的采样结果，如下所示：

```
void flushAppendOnlyFile(int force) {
    ...
    if (server.aof_fsync == AOF_FSYNC_ALWAYS) {
        latencyStartMonitor(latency); //调用latencyStartMonitor函数开始计时
```

```
redis_fsync(server.aof_fd); //实际将数据写入磁盘
    latencyEndMonitor(latency); //调用latencyEndMonitor结束计时，并计算时长
    latencyAddSampleIfNeeded("aof-fsync-always", latency);
...}
}
```

那么在这里，你需要注意的是，Redis源码在调用latencyAddSampleIfNeeded函数记录采样结果时，经常会在延迟事件执行前，调用**latencyStartMonitor函数**开始计时，并在事件执行结束后，调用**latencyEndMonitor函数**结束计时和计算事件执行时长。

此外，你也可以在阅读Redis源码的工具中，比如sublime、sourceinsight等，通过查找函数关系调用，找到latencyAddSampleIfNeeded函数被调用的其他地方。

好了，到这里，Redis延迟监控框架就能通过latencyAddSampleIfNeeded函数，来记录延迟事件的采样结果了。而实际上，Redis延迟监控框架还实现了延迟分析，并能提供应对延迟变慢的建议，我们再来看下。

## 延迟分析和提供应对措施建议

首先，Redis是提供了latency doctor命令，来给出延迟分析结果和应对方法建议的。当我们执行这条命令的时候，Redis就会使用latencyCommand函数来处理。而在处理这个命令时，latencyCommand函数会调用**createLatencyReport函数**，来生成延迟分析报告和应对方法建议。

具体来说，createLatencyReport函数会针对latency\_events哈希表中记录的每一类事件，先调用analyzeLatencyForEvent函数，计算获得采样的延迟事件执行时长的均值、最大/最小值等统计结果。具体的统计计算过程，你可以仔细阅读下analyzeLatencyForEvent函数的源码。

然后，createLatencyReport函数会针对这类事件，结合Redis配置项等信息给出应对措施。

其实，在createLatencyReport函数中，**它定义了多个int变量，当这些变量的值为1时，就表示建议Redis使用者采用一种应对高延迟的措施**。我在下面的代码中展示了部分应对措施对应的变量，你可以看下。另外你也可以阅读createLatencyReport函数源码，去了解所有的措施。

```
sds createLatencyReport(void) {
...
int advise_slowlog_enabled = 0; //建议启用slowlog
int advise_slowlog_tuning = 0; //建议重新配置slowlog阈值
int advise_slowlog_inspect = 0; //建议检查slowlog结果
int advise_disk_contention = 0; //建议减少磁盘竞争
...
}
```

我们也来简单举个例子。比如说，针对command事件，createLatencyReport函数就会根据slowlog的设置情况，给出启用slowlog、调整slowlog阈值、检查slowlog日志结果和避免使用bigkey的应对建议。这部分代码如下所示：

```

if (!strcasecmp(event, "command")) {
    //如果没有启用slowlog, 则建议启用slowlog
    if (server.slowlog_log_slower_than < 0) {
        advise_slowlog_enabled = 1;
        advices++;
    } //如果slowlog使用的命令时长阈值太大, 建议调整slowlog阈值
    else if (server.slowlog_log_slower_than/1000 >
             server.latency_monitor_threshold)
    {
        advise_slowlog_tuning = 1;
        advices++;
    }
    advise_slowlog_inspect = 1; //建议检查slowlog结果
    advise_large_objects = 1; //建议避免使用bigkey
    advices += 2;
}

```

所以，像createLatencyReport函数这样在计算延迟统计结果的同时，也给出应对措施的设计就很不错，这也是从Redis开发者的角度给出的建议，它更具有针对性。

好了，到这里，我们就了解了延迟监控框架的实现。接下来，我们再来学习下Redis中慢命令日志的实现。

## 慢命令日志的实现

Redis是使用了一个较为简单的方法来记录慢命令日志，也就是用一个列表，把执行时间超出慢命令日志执行时间阈值的命令记录下来。

在Redis全局变量server对应的数据结构redisServer中，有一个list类型的成员变量**slowlog**，它就是用来记录慢命令日志的列表的，如下所示：

```

struct redisServer {
    ...
    list *slowlog;
    ...}

```

而实现慢命令日志记录功能的代码是在[slowlog.c](#)文件中。这里的主要函数是**slowlogPushEntryIfNeeded**，它的原型如下所示：

```

void slowlogPushEntryIfNeeded(client *c, robj **argv, int argc, long long duration)

```

从代码中你可以看到，这个函数的参数包含了当前执行命令及其参数argv，以及当前命令的执行时长duration。

这个函数的逻辑也不复杂，它会判断当前命令的执行时长duration，是否大于redis.conf配置文件中的慢命

令日志阈值slowlog-log-slower-than。如果大于的话，它就会调用slowlogCreateEntry函数，为这条命令创建一条慢命令日志项，并调用listAddNodeHeader函数，把这条日志项加入到日志列表头，如下所示：

```
//当前命令的执行时长是否大于配置项
if (duration >= server.slowlog_log_slower_than)
    listAddNodeHead(server.slowlog, slowlogCreateEntry(c,argv,argc,duration));
```

当然，如果日志列表中记录了太多日志项，它消耗的内存资源也会增加。所以slowlogPushEntryIfNeeded函数在添加日志项时，会判断整个日志列表的长度是否超过配置项slowlog-max-len。一旦超过了，它就会把列表末尾的日志项删除，如下所示：

```
//如果日志列表超过阈值长度，就删除列表末尾的日志项
while (listLength(server.slowlog) > server.slowlog_max_len)
    listDelNode(server.slowlog,listLast(server.slowlog));
```

现在，我们也就了解了记录慢命令日志项的主要函数，slowlogPushEntryIfNeeded的基本逻辑了。然后我们再来看下，它在记录日志项时调用的**slowlogCreateEntry函数**。

这个函数是用来创建一个慢命令日志项。慢命令日志项的数据结构是slowlogEntry，如下所示：

```
typedef struct slowlogEntry {
    robj **argv;      //日志项对应的命令及参数
    int argc;         //日志项对应的命令及参数个数
    long long id;      //日志项的唯一ID
    long long duration; //日志项对应命令的执行时长（以微秒为单位）
    time_t time;       //日志项对应命令的执行时间戳
    sds cname;         //日志项对应命令的发送客户端名称
    sds peerid;        //日志项对应命令的发送客户端网络地址
} slowlogEntry;
```

从slowLogEntry的定义中，你可以看到，它会把慢命令及其参数，以及发送命令的客户端网络地址记录下来。**这样设计的好处是**，当我们分析慢命令日志时，就可以直接看到慢命令本身及其参数了，而且可以知道发送命令的客户端信息。而这些信息，就有利于我们排查慢命令的起因和来源。

比如说，如果我们发现日志中记录的命令参数非常多，那么它就可能是一条操作bigkey的命令。

当然，考虑到内存资源有限，slowlogCreateEntry函数在创建慢命令日志项时，也会判断命令参数个数。如果命令参数个数，超出了阈值SLOWLOG\_ENTRY\_MAX\_ARGC这个宏定义的大小（默认32）时，它就不会记录超出阈值的参数了，而是记录下剩余的参数个数。这样一来，慢命令日志项中就既记录了部分命令参数，有助于排查问题，也避免了记录过多参数，占用过多内存。

下面的代码展示了slowlogCreateEntry的基本执行逻辑，你可以看下。



```

slowlogEntry *slowlogCreateEntry(client *c, robj **argv, int argc, long long duration) {
    slowlogEntry *se = zmalloc(sizeof(*se)); //分配日志项空间
    int j, slargc = argc; //待记录的参数个数，默认为当前命令的参数个数

    //如果当前命令参数个数超出阈值，则只记录阈值个数的参数
    if (slargc > SLOWLOG_ENTRY_MAX_ARGC) slargc = SLOWLOG_ENTRY_MAX_ARGC;
    se->argc = slargc;
    ...
    for (j = 0; j < slargc; j++) { //逐一记录命令及参数
        if (slargc != argc && j == slargc-1) { //如果命令参数个数超出阈值，使用最后一个参数记录当前命令实际剩余的参数个数
            se->argv[j] = createObject(OBJ_STRING,
                sdscatprintf(sdsempty(), "... (%d more arguments)",
                    argc-slargc+1));
        } else {
            ... //将命令参数填充到日志项中
        }
    }
    ... //将命令执行时长、客户端地址等信息填充到日志项中
}

```

好了，到这里，你就了解了慢命令日志的实现。最后，你也要注意，**慢命令日志只会记录超出执行时长阈值的命令信息**，而不会像延迟监控框架那样记录多种事件。所以，记录日志的函数 `slowlogPushEntryIfNeeded`，只会在命令执行函数 `call`（在 `server.c` 文件中）中被调用，如下所示：

```

void call(client *c, int flags) {
    ...
    start = server.ustime; //命令执行前计时
    c->cmd->proc(c); //命令实际执行
    duration = ustime()-start; //命令执行完成计算耗时
    ...
    if (flags & CMD_CALL_SLOWLOG && c->cmd->proc != execCommand) {
        ...
        //调用slowlogPushEntryIfNeeded函数记录慢命令
        slowlogPushEntryIfNeeded(c, c->argv, c->argc, duration);
    }
    ...}

```

## 小结

今天这节课，我给你介绍了Redis实现的延迟监控框架和慢命令日志。你要知道，Redis源码会针对可能导致Redis运行变慢的五类事件，在它们执行时进行**采样**。而一旦这些事件的执行时长超过阈值时，监控框架就会将采样结果记录下来，以便后续分析使用。这种针对延迟事件进行采样记录的监控方法，其实是很值得我们学习的。

而慢命令日志的实现则较为简单，就是针对运行时长超出阈值的命令，使用一个**列表**把它们记录下来，这里面包括了命令及参数，以及发送命令的客户端信息，这样可以方便运维人员查看分析。

当然，Redis源码中实现的延迟监控框架主要是关注导致延迟增加的事件，它记录的延迟事件，也是和Redis运行过程中可能会导致运行变慢的操作**紧耦合**的。此外，Redis的INFO命令也提供了Redis运行时的监控信息，不过你要知道，INFO命令的实现，主要是在全局变量 `server` 的成员变量中，用来记录Redis实例的实时运行状态或是资源使用情况的。



## 每课一问

Redis在命令执行的call函数中，为什么不会针对EXEC命令，调用slowlogPushEntryIfNeeded函数记录慢命令呢？

## 精选留言：

- 斩弦 2021-10-19 17:41:48

蒋老师好，今天很巧正好在思考延时的问题，同时看到了更新了这节课，

我的问题(1) ./redis-cli --intrinsic-latency xxx这个命令能够看到内核本身的延时，因为我们的redis环境是部署在虚拟化环境的，时不时会有延时增大的现象发生，也有过发生问题后排查到集中发生在某个虚机所在的宿主机上，问题处理就是将这个虚机迁移到了其他的宿主机上，那么能否围绕./redis-cli --intrinsic-latency这个命令，我们可以自行开发一个逻辑，每10s或者15s采样一次，然后作为CPU内核延时的指标值，放到我们的监控逻辑中，这个耗时可以作为问题出现时的一个参考，作为问题出现时的一个重要的延时的成分；

另外还有一个问题(2)latency monitor这个功能对系统的性能影响是否明显呢，其实看到了这个功能后，有想要尝试使用这个功能以扩展延时问题排查时的手段