InfluxDB 使用和注意事项

1 安装事项:

1.1 前提条件:

安装 InfluxDB 包需要 root 或是有管理员权限才可以

1.2 网络

TCP 端口 8086 用作 InfluxDB 的客户端和服务端的 http api 通信;

TCP 端口 8088 给备份和恢复数据的 RPC 服务使用;

InfluxDB 也提供了多个可能需要自定义端口的插件，所以的端口映射都可以通过配置文件修 改，对于默认安装的 InfluxDB，这个配置文件位于 /etc/influxdb/influxdb.conf。

1.3 时间同步 NTP

InfluxDB 使用服务器本地时间给数据加时间戳，而且是 UTC 时区的。并使用 NTP 来同步服务器之间的时间，如果服务器的时钟没有通过 NTP 同 步，那么写入 InfluxDB 的数据的时间戳就可能不准确。

2 硬件要求:

2.1 读写磁盘:

建议使用两块 SSD 卷，一个是为了 influxdb/wal，一个是为了 influxdb/data；根据您的负载量，每个卷应具有大约 1k-3k 的 IOPS。influxdb/data 卷应该有更多的磁盘空间和较低的 IOPS，而 influxdb/wal 卷则相反有较少的磁盘空间但是较高的 IOPS。

2.2 内存要求:

每台机器应该有不少于 8G 的内存。

3 入门使用:

3.1 使用方式与基本概念

命令行或者直接发送裸的 HTTP 请求来操作数据库(例如 curl)

基本概念：

Database: 数据库名，在 InfluxDB 中可以创建多个数据库，不同数据库中的数据文件是隔离存放的，存放在磁盘上的不同目录。

Retention Policy: 存储策略，用于设置数据保留的时间，每个数据库刚开始会自动创建一个默认的存储策略 autogen，数据保留时间为永久，之后用户可以自己设置，例如保留最近 2 小时的数据。插入和查询数据时如果不指定存储策略，则使用默认存储策略，且默认存储策略可以修改。InfluxDB 会定期清除过期的数据。

Measurement: 对于传统数据库的表，例如 cpu_usage 表示 cpu 的使用率。

Tag sets: tags 在 InfluxDB 中会被建立索引，且放在内存中。如果某种数据经常用来被作为查询条件，可以考虑设为 Tag

Field: 记录值，是查询的主要对象，例如 value 值等

Point:代表一条记录

Series:tag key 与 tag value 的唯一组合

Timestamp: 每一条数据都需要指定一个时间戳，在 TSM 存储引擎中会特殊对待，以为了优化后续的查询操作。

3.2 读写数据

```
#插入数据
$INSERT cpu,host=serverA,region=us_west value=0.64
$INSERT temperature,machine=unit42,type=assembly external=25,internal=37
#查询数据
$SELECT "host", "region", "value" FROM "cpu"
#InfluxQL 还有很多特性和用法，包括支持 golang 样式的正则，例如：
$SELECT * FROM /.*/ LIMIT 1
```

3.3 HTTP 操作数据库

```
#使用 HTTP 接口创建数据库,使用 POST 方式发送到 URL 的/query 路径
$curl -i -XPOST http://localhost:8086/query --data-urlencode "q=CREATE DATABASE mydb"
#通过 HTTP 接口 POST 数据到/write 路径是我们往 InfluxDB 写数据的主要方式
$curl -i -XPOST 'http://localhost:8086/write?db=mydb' --data-binary
'cpu_load_short,host=server01,region=us-west value=0.64 1434055562000000000'
#同时写入多个点,可以用新的行来分开这些数据点。这种批量发送的方式可以获得更高的性能。
$curl -i -XPOST 'http://localhost:8086/write?db=mydb' --data-binary
'cpu_load_short,host=server02 value=0.67
cpu_load_short,host=server02,region=us-west value=0.55 1422568543702900257
cpu_load_short,direction=in,host=server01,region=us-west value=2.0
1422568543702900257'
#写入文件中的数据,给一个正确的文件(cpu_data.txt)的例子：
cpu_load_short,host=server02 value=0.67
cpu_load_short,host=server02,region=us-west value=0.55 1422568543702900257
cpu_load_short,direction=in,host=server01,region=us-west value=2.0
1422568543702900257

#看我们如何把 cpu_data.txt 里的数据写入 mydb 数据库：
$curl -i -XPOST 'http://localhost:8086/write?db=mydb' --data-binary @cpu_data.txt
无模式设计
InfluxDB 是一个无模式(schemaless)的数据库，你可以在任意时间添加 measurement，tags 和 fields。注意：如果你试图写入一个和之前的类型不一样的数据(例如，filed 字段之前接收的是数字类型，现在写了个字符串进去)，那么 InfluxDB 会拒绝这个数据。

#HTTP 返回值概要
```

```
2xx：如果你写了数据后收到 HTTP 204 No Content，说明写入成功了！
4xx：表示 InfluxDB 不知道你发的是什么鬼。
5xx：系统过载或是应用受损。
```

4 使用注意点:

4.1 操作域

由于 Tag 与 Field 的不同特性，在编写 SQL 进行查询时，Tag 与 Field 支持不同的操作，总结如下:

Tag:

只能使用 Tag 进行 Group

只能使用 Tag 进行正则表达式操作

Field:

只能使用 Field 进行函数操作，例如 sum()

只能使用 Field 进行比较操作

如果需要使用 int,float,boolean 类型进行存储,只能使用 Field

4.2 Schema 设计总结

4.2.1 不要把数据放到 measurement 名称中。

例如 不要让 measurement 名称看起来是这样的：

```
cpu.server1.us_west
```

应该改成

```
cpu,host=server1,region=us_west
```

4.2.2 不要把数据放到 Tag value 中

例如 不要让 measurement 名称看起来是这样的：

```
cpu,host=server1.us_west
```

应该改成

```
cpu,host=server1,region=us_west
```

4.2.3 不要使用取值范围很广的数据作为 tag,例如 uuid,hash 等等

如果实在有这方面的需求，考虑一下几点建议

1 切成多个 shard,并分到多个实例上

2 使用 tag 前缀进行区分

3 使用 field

4 使用集群

4.2.4 Tag Key 不要与 Field 的名称相同

4.2.5 Tags 的数量不要太少

4.2.6 database 的数量不要太多

当 database 的数量达到千万级别时，会出现打开文件过多，占用内存过多等问题。

5 优化方案:

5.1 控制 series 的数量

Series 会被索引且存在内存中，如果量太大会对资源造成过多损耗，且查询效率也得不到保障。

可以通过以下方式查询 series 的数量：

```
influx -database 'cloudportal' -execute 'show series' -format 'csv'|wc -l
```

通过以下方式查询 tag values 的数量：

```
influx -database 'cloudportal' -execute 'SHOW TAG VALUES FROM six_months.collapsar_flow WITH KEY = dip' -format 'csv'|wc -l
```

数量是否合适可以参考以下标准：

- 机器配置

类型	CPU	RAM	IOPS
Low	2-4 cores	2-4G	500
Moderate	4-6 cores	8-32G	500-1000
High	8+ cores	32+G	1000+

- 配置对应的 Series 数量

机器配置	每秒写 Field	每秒查询数量	Series 数量
Low	<5 K	<5	<100k
Moderate	< 250 K	<25	<1million
High	>250 K	>25	>1million

5.2 控制 Tag Key,与 Tag Value 值的大小

5.3 使用批量写如果使用 HTTP 一次写一条记录，或许还没有太大的负担，但是如果用 HTTPS 的进行一条一条的写，在加密/解密上的资源损耗会非常的大。如果不能使用 HTTP,则推荐使用 UDP 协议

5.4 使用 Continuous Queries 进行数据汇聚对于查询时间范围较大且数据粒度要求不是非常高的数据，可以考虑使用 CQ 进行数据汇总，并对汇总结果进行查询

5.5 使用恰当的时间粒度在数据存储的时候默认使用纳秒。而对于很多业务操作而言，可能只需要精确到秒级别。这种情况对于存储资源以及查询性能都会有一定的影响。想法如果业务需要毫秒级别的精确程度，而存的时候使用了秒级别的数据，此时查询又会出现数据的丢失

5.6 存储的时候尽量对 Tag 进行排序

5.7 无关的数据写不同的 database

5.8 根据数据情况，调整 shard 的 duration

默认 7 天创建一个，如果查询的时间范围较大，会打开多个 shard 文件，对于数据量不大，且查询范围可能较大的数据，可以将 shard duration 时间设置的长一点

5.9 存储分离将 WAL 目录与 data 目录分别映射到不同的磁盘上，以减少读写操作的相互影响

内存问题:

InfluxDB TCP 连接数过多问题：

描述:发现与 InfluxDB 使用的 8086 端口相关的 TCP 连接数竟然多大 6K+，有时候甚至会逼近 1w ,这个数量对于一个只是在内部使用的监控系统来说, 无论如何都是无法接受的。

因为业务的需要, client 对 InfluxDB 做 query 时， 会经过 server 的一个 proxy，再由 proxy 发起到 Influxdb 的 query (http) 请求. 在我们的部署架构中, proxy 与 InfluxDB 是部署在同一个 server 上的. 使用命令

```
$netstat -apn | grep 8086
```

可以看到大量处于 TIME_WAIT 状态的 tcp 连接

```
...
tcp6 0 0 127.0.0.1:8086 127.0.0.1:58874 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59454 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59084 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59023 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59602 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59027 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59383 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59053 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:58828 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:58741 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59229 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:58985 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59289 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59192 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59161 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59292 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59242 TIME_WAIT -
tcp6 0 0 127.0.0.1:8086 127.0.0.1:59430 TIME_WAIT -
...
```

使用命令

```
$netstat -apn | grep 8086 | grep TIME_WAIT | wc -l
```

进行计数，会发现连接数会不断增加，经过多次测试，在公司环境中连接数至少都会达到 6k+. 这个问题必须要解决，一方面是因为每条 tcp 连接都会占用内存，另一方面系统的动态端口数也是有限的.
很明显这些连接几乎都处在 TIME_WAIT 状态,所以在继续往下走之前，需要了解下 TIME_WAIT 这个关键字.

TIME_WAIT

我们知道 一条 tcp 连接从开始到结束会经历多个状态，换句话说，可以把 一条 tcp 连接看成是一个 状态机. 这个状态图如下:

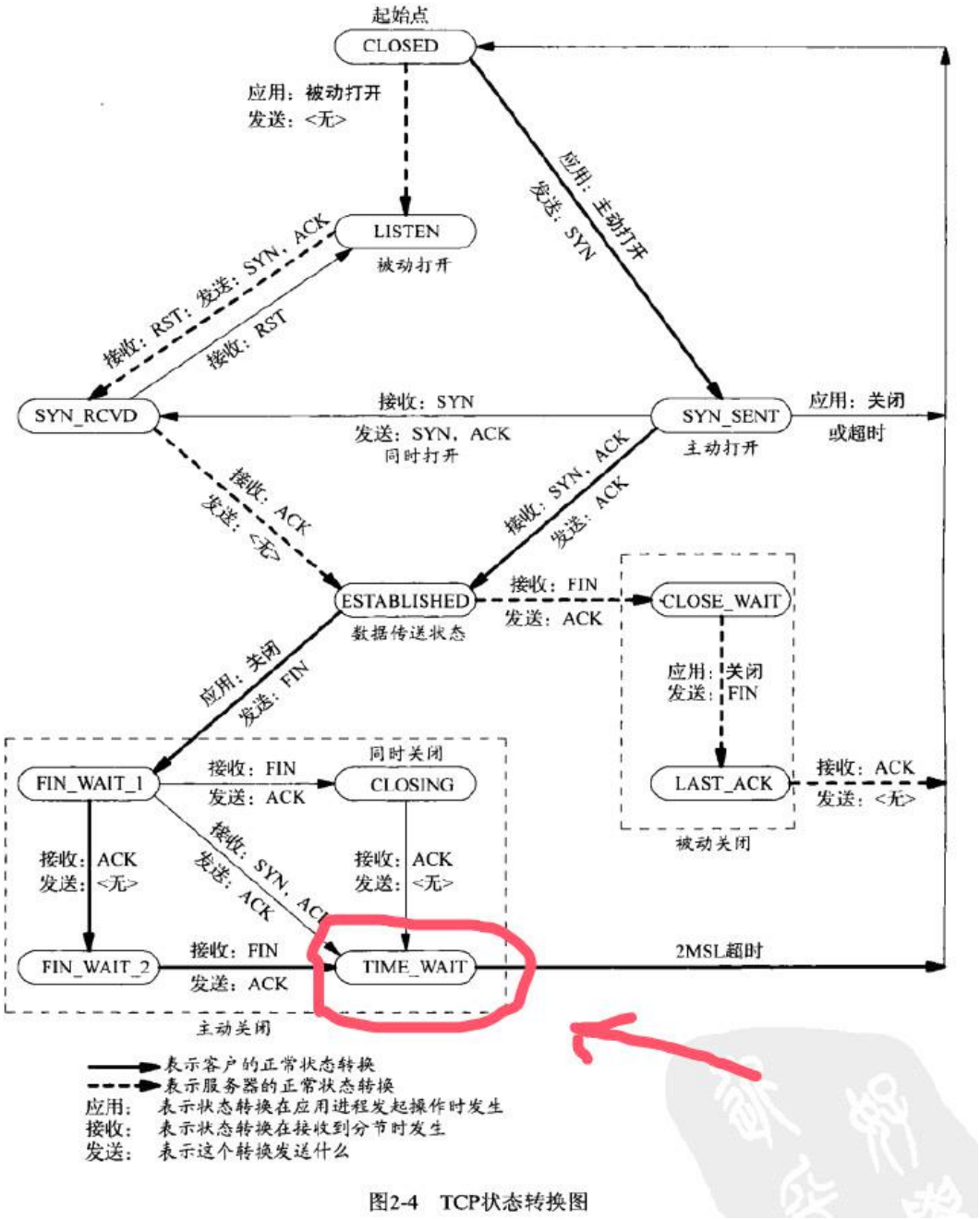

图2-4 TCP状态转换图

可以看到，凡是主动进行关闭 tcp 连接的一方，都会经过 TIME_WAIT 这个状态.接下来再经过 2MSL 的时间后内核再完全释放相应的文件描述符和端口.
(顺便提一下，MSL 是最大分段寿命，是一个 TCP 分段可以存在于互联网系统中的最大时间，在 Linux 下可以用命令

```
$cat /proc/sys/net/ipv4/tcp_fin_timeout
```

查看 MSL 的数值)
到这个地方可以推断出，是 8086 端口(即 InfluxDB) 主动关闭了 tcp 连接，导致挤压了大量的处于 TIME_WAIT 状态下的连接在等待内核释放. 于是自然而然得会想到，能不能限制 InfluxDB 能打开的最大连接数，让它尽可能复用每一条 tcp 连接?

```
# The maximum number of HTTP connections that may be open at once.
connections that
# would exceed this limit are dropped.  Setting this value to 0 disables
the limit.
max-connection-limit = 0
```

将其该为 100，然后重启数据库.

上文中提到的 InfluxDB 版本为 1.2，处理 query 请求的代码在 services/httpd/handler.go 中，函数签名为

```
func (h *Handler) serveQuery(w http.ResponseWriter, r *http.Request,
user meta.User)

#在其中发现一句代码
rw.Header().Add("Connection", "close")
```

确实是 Influxdb 主动关闭的连接，并且通知 client (也就是文中提到的 proxy) 在请求完成后也关闭连接.

为了验证这一点，我将上面那句代码注释掉后重新编译 InfluxDB，在 client 正确复用连接的情况下，连接数确实可以保持在 10 以内.

InfluxDB1.2 之前出现这种情况。