第1章　Zabbix 数据库表结构解析

由于 Zabbix 前端效率极低，大批量更新会造成 oracle tx 锁，所以对于这种大批量的更新，一般都用数据库语句直接 update。想过通过 patch php 代码的方法来解决这个问题（Zabbix 本身的类似问题就是这样解决的），但是 php 一个是不会，一个是太长。而且，数据 update 快。这样就需要对 Zabbix 数据库的数据结构有清晰的了解。

另一方面，Zabbix 收集了大量的裸数据，其他人可以通过这些数据来进行分析，同样，也需要了解数据库的结构。

在使用过程中，我们也是摸石头过河，一边摸索一边使用。这里会对 Zabbix 数据库的表结构和常用的操作做一些说明。

注意：

Zabbix 数据库中的表的名称都是复数，比如存放 Host 信息的表的名字是 Hosts 等。
数据库操作有风险，一旦出问题会造成 Zabbix crash。需要谨慎操作。
普通的查询可以在备库上进行。两边数据是实时同步的。

1.1　表结构概述

Zabbix 的数据库设计是很有特点的，针对 Zabbix 中的每一个资源，都有一张表与其对应，比如 hosts 表，items 表等。而这每一张表中，都有一个 id 字段，比如 hosts 表中有 hostid 列，items 表中有 itemid 列。而资源之间的关联关系，是通过外键来完成的。比如 host 和 item 的关联关系，就是在 items 表中使用 hostid 与 hosts 表中的资源进行关联的。

下面就以 hostid 来举例说明，其他 itemid，trigger 等都类似。

hostid 是每一个 host 的唯一标识，我们从数据库查询的时候，一般都是以 hostid 为查询条件。我们点开一个 host，然后看 URL：
http://192.168.201.234/zabbix/hosts.php?form=update&hostid=10108&groupid=0&sid=27716d11b8954723

我们可以看到 URL 里的 GET 参数有这几个：

1. form：表示当前页面的操作，这里的 update 是因为我是从 Configuration-Hosts 中点击 host 进入的，所以是一个更新的操作
2. hostid：点击的 host 的 hostid
3. groupid：这里不需要 groupid 这个字段，所以这个 0 没有意义
4. sid：sessionid，标识用户用的

这里顺便说一句，Zabbix 的前端界面的 URL，会有很多像上面的 URL 那样。有的是 itemid，有的是 triggerid，我们更改一下这个 id，就自然能够跳转到对应的界面上去了。这一点正是 Zabbix 的灵活之处，再我们进行二次开发的时候，非常有用。

有的朋友要说了，我只要用好 Zabbix 就行了，不想去了解 Zabbix 的数据库表结构，觉得没

必要也没用。其实非也，从 Zabbix 数据库的表结构，我们可以知道 Zabbix 资源的数据结构。另外，活用 SQL 查询 Zabbix 数据库，能够提升我们的效率，举个例子，我们想看某个机房的网卡出口流量之和，怎么办呢？我们可以很复杂的定义一个 aggregate 类型的 item，然后在前端点来点去。但是如果我们用 SQL，就一条 SQL 就可以非常简单的解决问题。

另一方面，Zabbix 的水平高低，或者说是否真正了解 Zabbix 的一个非常大的标志，就是是否了解 Zabbix 的数据库。大家可能看这一节很吃力，但没关系，如果是刚接触 Zabbix 不久，这也是正常的，可以先看一遍留个印象，等之后对 Zabbix 有了些了解以后再来仔细看看。

1.2 Hosts 表

批注 [d1]: 改下结构

“Host”就是指一台被监控的机器。我们先看 Hosts 表结构，如下：

```
mysql> desc hosts;
+--------------------+---------------------+------+-----+---------+-------+
| Field              | Type                | Null | Key | Default | Extra |
+--------------------+---------------------+------+-----+---------+-------+
| hostid             | bigint(20) unsigned | NO   | PRI | NULL    |       |
| proxy_hostid       | bigint(20) unsigned | YES  | MUL | NULL    |       |
| host               | varchar(64)         | NO   | MUL |         |       |
| status             | int(11)             | NO   | MUL | 0       |       |
| disable_until      | int(11)             | NO   |     | 0       |       |
| error              | varchar(128)        | NO   |     |         |       |
| available          | int(11)             | NO   |     | 0       |       |
| errors_from        | int(11)             | NO   |     | 0       |       |
| lastaccess         | int(11)             | NO   |     | 0       |       |
| ipmi_authtype      | int(11)             | NO   |     | 0       |       |
| ipmi_privilege     | int(11)             | NO   |     | 2       |       |
| ipmi_username      | varchar(16)         | NO   |     |         |       |
| ipmi_password      | varchar(20)         | NO   |     |         |       |
| ipmi_disable_until | int(11)             | NO   |     | 0       |       |
| ipmi_available     | int(11)             | NO   |     | 0       |       |
| snmp_disable_until | int(11)             | NO   |     | 0       |       |
| snmp_available     | int(11)             | NO   |     | 0       |       |
| maintenanceid      | bigint(20) unsigned | YES  | MUL | NULL    |       |
| maintenance_status | int(11)             | NO   |     | 0       |       |
| maintenance_type   | int(11)             | NO   |     | 0       |       |
| maintenance_from   | int(11)             | NO   |     | 0       |       |
| ipmi_errors_from   | int(11)             | NO   |     | 0       |       |
| snmp_errors_from   | int(11)             | NO   |     | 0       |       |
| ipmi_error         | varchar(128)        | NO   |     |         |       |
| snmp_error         | varchar(128)        | NO   |     |         |       |
| jmx_disable_until  | int(11)             | NO   |     | 0       |       |
| jmx_available      | int(11)             | NO   |     | 0       |       |
| jmx_errors_from    | int(11)             | NO   |     | 0       |       |
| jmx_error          | varchar(128)        | NO   |     |         |       |
| name               | varchar(64)         | NO   | MUL |         |       |
```

| flags | int(11) | NO | | 0 | |
| templateid | bigint(20) unsigned | YES | MUL | NULL | |
+---+---+---+---+---+---+

- hostid：唯一标识 Host 在 Zabbix 及数据库的 id。不同表之间的关联也是用的 id。和这个类似，Zabbix 中任意一种资源都有自己的 id，比如 itemid，groupid 等。
- proxy_hostid：如果使用了‘Proxy-Server’架构，这个字段表示的是监控这台机器的 Proxy 的 hostid。有一点需要注意，每个 Proxy 在 Hosts 表里有两条记录（其他 Host 只有一条记录），一条是和普通机器一样的、作为被监控机器的记录；另一条记录是作为 Proxy 的。作为 Proxy 的那条记录，ip 字段的值为“0.0.0.0”。proxy_hostid 中的值就是 Proxy 记录中的 hostid。举个例子，我有一台 Proxy 的 ip 为 1.2.3.4，那么在 Hosts 表里有两条记录，一个是 ip 为“1.2.3.4”的记录，hostid 为“1”；另一个是 ip 为“0.0.0.0”的记录，hostid 为“2”。在这个背景下，有一台机器，他的 proxy 是之前提到的那台机器，那么他在 Hosts 表中的 proxy_hostid 的值为“2”。
- host：机器的 hostname。注意，在 1.8.8（即我们使用的）版本的 Zabbix 中，如果有两台 hostname 一样的机器，那么 Zabbix 会 crash 直接退出。之后在 1.8.10（记不清了）取消了这个功能。其实我也觉得这功能挺脑残的。
- dns：DNS 名称。
- useip：是否用 ip 监控。
- port：监控使用的端口。
- status：机器目前的状态。“0”为正常监控，“1”为 disable。“2”不清楚，从数据库里找不到 status 为“2”的机器。google 了下，这个好像是 Zabbix 自身的一个 host available 检查有关。“3”表示是个 Template。（是不是很奇怪为啥 Template 也在 Hosts 表中？其实 Template 就是个 Host。详细的以后再说）
- disable_util， error， available， errors_from(ipmi_disable_util， ipmi_error … 和 snmp_disable_until…都是此类)：这几个都是 Zabbix Poller 会去修改的值。我看了下 poller.c 的代码，当 poller 在第一次取不到值（根据值的类型不同会更新相应的列，Item 类型为 snmp 就会更新 snmp_XXX，默认为“zabbix”类型）的时候，会等 15 秒（CONFIG_UNREACHABLE_DELAY）来重试，并且日志会显示“first network error”，如果 15 秒后依然取不到值，zabbix 会在数据库更新这个 host 取不到值的信息，即这几列。并且日志里显示“another network error”。
- lastaccess：这一列是专门为 proxy 准备的（如上文 ip 为“0.0.0.0”）。lastaccess 表示的是 proxy 最后一次工作的时间。这里的“工作”指 Zabbix Server 收到 Proxy 数据。
- inbytes，outbytes：不知道有什么用，1.8.8 的代码中也没有找到使用这两个字段的代码。我估计是 Zabbix 以后会使用的。
- useipmi，ipmi_*(除 8.中提到的)：使用 IPMI 时后的参数。不展开说。
- snmp_*(除 8.中提到的)：同上，SNMP 参数。
- maintenanceid，maintenance_*：这是 Zabbix 另一个机制 Maintaince 有关，用于使 Host 置于维护状态而不会报警。

常用操作

前面只讲了 Hosts 一张表，所以这里只能介绍一些针对 Host 的操作。

更新机器的 proxy。找到 proxy 的 hostid，更新对用 host 的 proxy_hostid：

```
select hostid from hosts where host='ProxyA' and ip='0.0.0.0';   -- get hostid: 1234
```

update hosts set proxy_hostid=1234 where host='Host_To_Update_Proxy';

enable/disable host:

update hosts set status='0' where host='Host_To_Enable';
update hosts set status='1' where host='Host_To_Disable';

1.3 Items 表

Items 表也是 Zabbix 的核心表之一，它记录了 Item 的所有设置。我们知道，在 Zabbix 中，我们做多的操作就是对于 items 的了，添加监控项，删除监控项，更新监控项配置等等。这一节中，我们一起看下 Item 在数据库存储的表——items 表。

首先我们看一下表结构：

```
+----------------------+---------------------+------+-----+---------+-------+
| Field                | Type                | Null | Key | Default | Extra |
+----------------------+---------------------+------+-----+---------+-------+
| itemid               | bigint(20) unsigned | NO   | PRI | NULL    |       |
| type                 | int(11)             | NO   |     | 0       |       |
| snmp_community       | varchar(64)         | NO   |     |         |       |
| snmp_oid             | varchar(255)        | NO   |     |         |       |
| hostid               | bigint(20) unsigned | NO   | MUL | NULL    |       |
| name                 | varchar(255)        | NO   |     |         |       |
| key_                 | varchar(255)        | NO   |     |         |       |
| delay                | int(11)             | NO   |     | 0       |       |
| history              | int(11)             | NO   |     | 90      |       |
| trends               | int(11)             | NO   |     | 365     |       |
| status               | int(11)             | NO   | MUL | 0       |       |
| value_type           | int(11)             | NO   |     | 0       |       |
| trapper_hosts        | varchar(255)        | NO   |     |         |       |
| units                | varchar(255)        | NO   |     |         |       |
| multiplier           | int(11)             | NO   |     | 0       |       |
| delta                | int(11)             | NO   |     | 0       |       |
| snmpv3_securityname  | varchar(64)         | NO   |     |         |       |
| snmpv3_securitylevel | int(11)             | NO   |     | 0       |       |
| snmpv3_authpassphrase | varchar(64)        | NO   |     |         |       |
| snmpv3_privpassphrase | varchar(64)        | NO   |     |         |       |
| formula              | varchar(255)        | NO   |     | 1       |       |
| error                | varchar(128)        | NO   |     |         |       |
| lastlogsize          | bigint(20) unsigned | NO   |     | 0       |       |
| logtimefmt           | varchar(64)         | NO   |     |         |       |
| templateid           | bigint(20) unsigned | YES  | MUL | NULL    |       |
| valuemapid           | bigint(20) unsigned | YES  | MUL | NULL    |       |
| delay_flex           | varchar(255)        | NO   |     |         |       |
| params               | text                | NO   |     | NULL    |       |
| ipmi_sensor          | varchar(128)        | NO   |     |         |       |
| data_type            | int(11)             | NO   |     | 0       |       |
```

authtype	int(11)	NO		0	
username	varchar(64)	NO			
password	varchar(64)	NO			
publickey	varchar(64)	NO			
privatekey	varchar(64)	NO			
mtime	int(11)	NO		0	
flags	int(11)	NO		0	
filter	varchar(255)	NO			
interfaceid	bigint(20) unsigned	YES	MUL	NULL	
port	varchar(64)	NO			
description	text	NO		NULL	
inventory_link	int(11)	NO		0	
lifetime	varchar(64)	NO		30	
snmpv3_authprotocol	int(11)	NO		0	
snmpv3_privprotocol	int(11)	NO		0	
state	int(11)	NO		0	
snmpv3_contextname	varchar(255)	NO			

- itemid：item 的 id
- type：item 的 type，和前端见面配置 item 的 type 的对应。数据库中，这一列的值是 0 到 17 的数字，分别代表：
  - 0：Zabbix agent
  - 1：SNMPv1 agent
  - 2：Zabbix trapper
  - 3：simple check
  - 4：SNMPv2 agent
  - 5：Zabbix internal
  - 6：SNMPv3 agent
  - 7：Zabbix agent(active)
  - 8：Zabbix aggregate
  - 9：web item
  - 10：external check
  - 11：database monitor
  - 12：IPMI agent
  - 13：SSH agent
  - 14：TELNET agent
  - 15：calculated
  - 16：JMX agent
  - 17：SNMP trap
- snmp_community：
- snmp_old：
- hostid：item 所在的 host 的 hostid。如果该 item 是属于 template，那么这里显示的是 templateid
- name：item 的名字

- key_：item 的 key
- delay：这里的 delay，实际就是在配置 item 时候配置的“Update Interval”，我也不明白为什么这里数据库字段名要叫做 delay
- history：前端配置中的存储 history 的时间
- trends：前端配置中的存储 trend 的时间
- status：item 的状态
  - 0：item 是 enabled 状态
  - 1：item 是 disabled 状态
- value_type：item 返回值的类型
  - 0：numeric float
  - 1：character
  - 2：log
  - 3：numeric unsigned
  - 4：text
- trapper_hosts：当 item 为 trapper 类型的时候，记录了
- units：和 items 配置一样
- multiplier：item 配置一样
- delta：
- snmpv3_securityname：
- snmpv3_securitylevel：
- snmpv3_authpassphrase：
- snmpv3_privpassprase：
- formula：
- error：item 的错误信息
- lastlogsize：
- logtimefmt：
- templateid：
- valuemapid：
- delay_flex：
- params：
- ipmi_sensor：
- data_type：
- authtype：
- username：
- password：
- publickey：
- privatekey：
- mtime：
- flags：
- filter：
- interfaceid：
- port：
- description：
- inventory_link：

- lifetime：
- snmpv3_authprotocol：
- snmpv3_privprotocol：
- state：当前 item 的状态
  - 0：正常
  - 1：not supported
- snmpv3_contextname：

1.4 Trigger 在数据库中的结构

批注 [d2]: 标题再斟酌

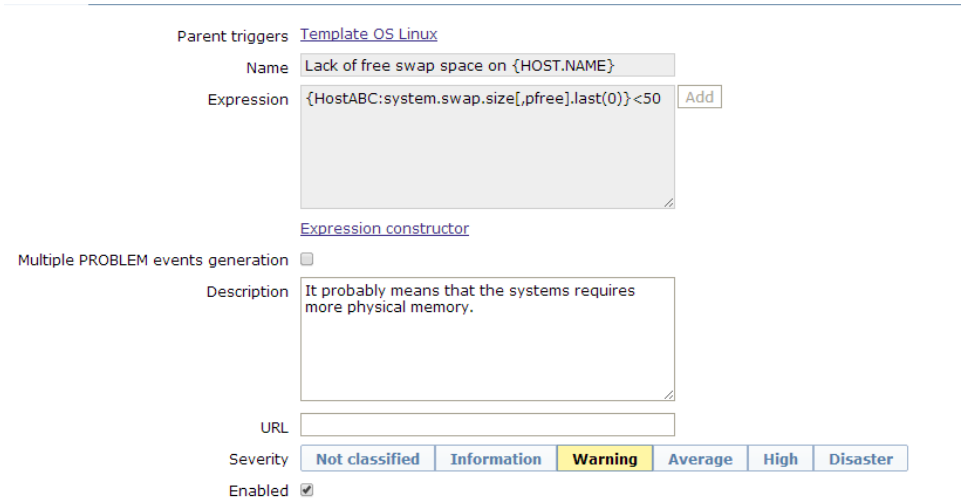
Trigger 是我们 Zabbix 的重要部分，我们平时在工作中，除了 Host 和 Item，就属 Trigger 了。而且 Trigger 相对 Host 和 Item 来说，更加的复杂。Host 和 Item 在数据库中对应的 hosts 表和 items 表都是非常平面的表，结构简单，host 和 item 的属性在表中一目了然。而 Trigger 在数据库中对应的 triggers 表是则相对复杂，它和其他表的关联关系很强，需要仔细分析。

万变不离其宗，首先先看 triggers 表的内容，我们看看 triggers 的表结构：

```
mysql> desc triggers;
+-------------+---------------------+------+-----+---------+-------+
| Field       | Type                | Null | Key | Default | Extra |
+-------------+---------------------+------+-----+---------+-------+
| triggerid   | bigint(20) unsigned | NO   | PRI | NULL    |       |
| expression  | varchar(2048)       | NO   |     |         |       |
| description | varchar(255)        | NO   |     |         |       |
| url         | varchar(255)        | NO   |     |         |       |
| status      | int(11)             | NO   | MUL | 0       |       |
| value       | int(11)             | NO   | MUL | 0       |       |
| priority    | int(11)             | NO   |     | 0       |       |
| lastchange  | int(11)             | NO   |     | 0       |       |
| comments    | text                | NO   |     | NULL    |       |
| error       | varchar(128)        | NO   |     |         |       |
| templateid  | bigint(20) unsigned | YES  | MUL | NULL    |       |
| type        | int(11)             | NO   |     | 0       |       |
| state       | int(11)             | NO   |     | 0       |       |
| flags       | int(11)             | NO   |     | 0       |       |
+-------------+---------------------+------+-----+---------+-------+
```

经过前面几节对于 hosts 表和 items 表的分析，我们能马上“感觉”到这些字段的含义了吧。这“感觉”两字打上引号的意思是大家看到这些字段，就能猜出是什么意思，我相信大多数读者朋友到这里已经有这个能力了。

如果还有朋友一下没看出来，我们就找一个 trigger 看一下：

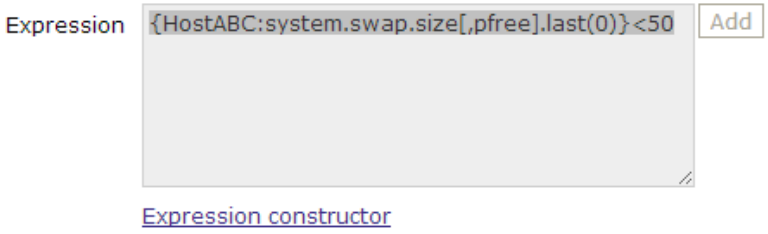


从 URL 中可以得到 triggerid 为 13588，我们从该数据库里把这个 trigger 取出来：

```
mysql> select * from triggers where triggerid=13588\G;
*************************** 1. row ***************************
   triggerid: 13588
  expression: {13191}<50
description: Lack of free swap space on {HOST.NAME}
         url:
      status: 0
       value: 0
    priority: 2
  lastchange: 1392535037
    comments: It probably means that the systems requires more physical memory.
       error:
  templateid: 10012
        type: 0
       state: 0
       flags: 0
```

现在大家能看明白了主要字段的作用了吧。trigger 的核心是 expression，即我们定义的 trigger 的逻辑。我们选取的这个 trigger 的逻辑是：{HostABC:system.swap.size[,pfree].last(0)}<50，但数据库里没有这个，数据库里 expression 字段是{13191}<50，50 这个阈值是吻合的，那么 13191 是什么东西呢？有的朋友可能猜到了——这个是某个 id，类似 triggerid 的 id。当时我也找了很久，都找不到这个是什么 id，后来才发现，是 Zabbix 一个非常坑爹的设计。

首先我们再看看上图中的设置 trigger 逻辑的部分：



我们可以看到这一串逻辑是属于“Expression”这个属性的，然后当时我就想当然的认为数据库里的 expressions 表就是记录这个的逻辑。但是，当我看 expressions 的内容时，才发现，这个 expressions 表记录的内容，并不是 trigger 里的 expression，而是 Administration-General 里的“Regular expressions”，内容如下：

```
mysql> select * from expressions\G;
*************************** 1. row ***************************
     expressionid: 1
           regexpid: 1
         expression: ^(btrfs|ext2|ext3|ext4|jfs|reiser|xfs|ffs|ufs|jfs|jfs2|vxfs|hfs|ntfs|fat32|zfs)$
expression_type: 3
   exp_delimiter: ,
  case_sensitive: 1
*************************** 2. row ***************************
     expressionid: 2
           regexpid: 2
         expression: ^lo$
expression_type: 4
   exp_delimiter: ,
  case_sensitive: 1
*************************** 3. row ***************************
     expressionid: 3
           regexpid: 3
         expression: ^(Physical memory|Virtual memory|Memory buffers|Cached memory|Swap space)$
expression_type: 4
   exp_delimiter: ,
  case_sensitive: 1
*************************** 4. row ***************************
     expressionid: 4
           regexpid: 2
         expression: ^Software Loopback Interface
expression_type: 4
   exp_delimiter: ,
  case_sensitive: 1
*************************** 5. row ***************************
     expressionid: 6
           regexpid: 4
```

```
        expression: ext4
expression_type: 3
   exp_delimiter: ,
  case_sensitive: 1
*************************** 6. row ***************************
     expressionid: 7
           regexpid: 4
        expression: ^ext
expression_type: 3
   exp_delimiter: ,
  case_sensitive: 1
```

再看下配置的 Expressions：

Name	Expressions
File systems for discovery	1»^(btrfs|ext2|ext3|ext4|jfs|reiser|xfs|ffs|ufs|jfs|jfs2|vxfs|hfs|ntfs|fat32|zfs)$[Result is TRUE]
Network interfaces for discovery	1»^lo$ [Result is FALSE] 2»^Software Loopback Interface[Result is FALSE]
Storage devices for SNMP discovery	1»^(Physical memory|Virtual memory|Memory buffers|Cached memory|Swap space)$[Result is FALSE]
testRegularExpression	1»ext4 [Result is TRUE] 2»^ext[Result is TRUE]

确定 expressions 表是对应这个的了。那我们要寻找的 13191 是什么呢？是 function，内容在 functions 表：

```
mysql> select * from functions where functionid=13191;
+------------+--------+-----------+----------+-----------+
| functionid | itemid | triggerid | function | parameter |
+------------+--------+-----------+----------+-----------+
|      13191 |  23777 |     13588 | last     | 0         |
+------------+--------+-----------+----------+-----------+
```

还是看这个例子，trigger 的 expression 是{HostABC:system.swap.size[,pfree].last(0)}<50，对比 functions 表，function 和 parameter 列我们都能明白它们的作用。itemid 就是关联的 item，即这里的 HostABC:system.swap.size[,pfree]：

```
mysql> select itemid,name,key_ from items where itemid=23777;
+--------+---------------------+-------------------------+
| itemid | name                | key_                    |
+--------+---------------------+-------------------------+
|  23777 | Free swap space in % | system.swap.size[,pfree] |
+--------+---------------------+-------------------------+
```

现在大家能把 triggers 是怎么设置的搞明白了吧。我们再看个复杂的，expression 是“{HostABC:agent.ping.last()}=0 | {HostABC:system.cpu.util[,iowait].last()}=0”，数据库里是这样的：

```
mysql> select * from triggers where triggerid=13601\G;
*************************** 1. row ***************************
  triggerid: 13601
 expression: {13204}=0 | {13205}=0
```

```
description: test1
        url:
     status: 0
      value: 0
   priority: 0
lastchange: 1394205728
   comments:
      error:
templateid: NULL
       type: 0
      state: 0
      flags: 0
```

再复杂的 expression 也由多个 function 拼成的。

1.5 Events 表

我们知道，当 Zabbix server 获取到一个数据，它就会检查跟这个 item 相关的 trigger，然后无论是否触发 action，都会生成一个 event。这一节中，我们看看 event 在数据库中的存储。首先我们看看 events 表的表结构：

```
mysql> desc events;
+-------------+---------------------+------+-----+---------+-------+
| Field       | Type                | Null | Key | Default | Extra |
+-------------+---------------------+------+-----+---------+-------+
| eventid     | bigint(20) unsigned | NO   | PRI | NULL    |       |
| source      | int(11)             | NO   | MUL | 0       |       |
| object      | int(11)             | NO   |     | 0       |       |
| objectid    | bigint(20) unsigned | NO   |     | 0       |       |
| clock       | int(11)             | NO   |     | 0       |       |
| value       | int(11)             | NO   |     | 0       |       |
| acknowledged | int(11)            | NO   |     | 0       |       |
| ns          | int(11)             | NO   |     | 0       |       |
+-------------+---------------------+------+-----+---------+-------+
```

我估计大家对“source”，“object”，“objectid”，“ns”这几个可能有点疑问，剩下来的大家应该都能理解它的含义。

1. source：event 可能由多种源头生成，这里的 source 就是记录了这个 event 是由于什么事件而生成的。
   a) 0：由 trigger 生成的 event
   b) 1：由 discovery rule 生成的 event
   c) 2：由 agent auto-registration 生成的 event
   d) 3：internal 的 event
2. object：这个字段记录了和 event 关联的 Zabbix 对象。
   a) 对于 trigger 相关的 events，这里的值只可能是 0
   b) 对于 discovery 相关的 event，“1”表示是 discovered host，“2”表示是 discovered service
   c) 对于 auto-registration 的 event，这里值一定是“3”

d) 对于 interval 的 event，“0”表示 trigger，“4”表示 item，“5”表示 low-level discovery

3. objectid：根据前面 object 里的定义，这里可能为 triggerid，也可能是 discovered hostid
4. ns：我印象中 1.8.8 的 Zabbix 是没有这个字段的，后来查询了 Zabbix 的官网，发现是在 2.0.0 加入这个纳秒的记录的。原因是这样的，如果只有 timestamp，那么这个{ITEM.VALUE}会发生错乱。所以在 ZBXNEXT-457 中有人提了 bug。地址在 https://support.zabbix.com/browse/ZBXNEXT-457
5. value：和 object 字段类似，根据 source 的不同，这里的值有不同的含义
   a) 对于 trigger 类型的 event：
      i. 0：trigger 的状态为 OK
      ii. 1：trigger 的状态为 PROBLEM
   b) 对于 discovery 类型的 event：
      i. 0：host 或者 service 正在工作
      ii. 1：host 或者 service 停止工作
      iii. 2：host 或者 service 被侦测到
      iv. 3：host 或者 service 丢失了
   c) 对于 internal 类型的 event：
      i. 0：normal 状态
      ii. 1：unknown 或者 not supported 状态

第2章 History 和 Trends

History 和 Trends 都是存储历史数据的地方，但是他们有什么区别，一直是大家使用 Zabbix 中的疑惑。这一章中，我们从数据库入手，将这个问题讲清楚。

首先我们看数据库中和 history 和 trends 相关的表：

```
mysql> show tables like '%history%';
+----------------------------+
| Tables_in_zabbix (%history%) |
+----------------------------+
| history                    |
| history_log                |
| history_str                |
| history_str_sync           |
| history_sync               |
| history_text               |
| history_uint               |
| history_uint_sync          |
| proxy_dhistory             |
| proxy_history              |
| user_history               |
+----------------------------+
mysql> show tables like '%trends%';
+----------------------------+
| Tables_in_zabbix (%trends%) |
```

```
+----------------------------+
| trends                     |
| trends_uint                |
+----------------------------+
```

trends 表是比较简单的，一共两个表，其中 trends_uint 表示的是 unsigned int，即这两张表的功能是一样的，只是存储的数据类型不同。

2.1 sync 字段的含义

而 history 表就比较多了，但从表明来看，我们能够很清楚的看到，大多数 history 表只是分了不同的数据类型，比如 str 表示字符串，log 表示 log 文件类型。其中比较奇怪的 sync 这个属性，我们知道 sync 一般是“synchronize”的缩写。我们看看是什么地方使用了，老办法，grep 源代码：

```
$ grep -r 'history_sync' *
libs/zbxdbcache/dbcache.c: *               for writing float-type items into history/history_sync tables.      *
libs/zbxdbcache/dbcache.c:         const char          *ins_history_sync_sql = "insert into history_sync (nodeid,itemid,clock,ns,value) values ";
libs/zbxdbcache/dbcache.c:                 zbx_strcpy_alloc(&sql, &sql_alloc, sql_offset, ins_history_sync_sql);
libs/zbxdbcache/dbcache.c:                         zbx_strcpy_alloc(&sql, &sql_alloc, sql_offset, ins_history_sync_sql);
libs/zbxdbhigh/dbschema.c:         {"history_sync",        "id",     ZBX_HISTORY_SYNC,
libs/zbxdbhigh/dbschema.c:CREATE TABLE history_sync (\n\
libs/zbxdbhigh/dbschema.c:CREATE INDEX history_sync_1 ON history_sync (nodeid,id);\n\
libs/zbxdbhigh/dbschema.c:CREATE TABLE `history_sync` (\n\
libs/zbxdbhigh/dbschema.c:CREATE INDEX `history_sync_1` ON `history_sync` (`nodeid`,`id`);\n\
libs/zbxdbhigh/dbschema.c:CREATE TABLE history_sync (\n\
libs/zbxdbhigh/dbschema.c:CREATE INDEX history_sync_1 ON history_sync (nodeid,id);\n\
libs/zbxdbhigh/dbschema.c:CREATE SEQUENCE history_sync_seq\n\
libs/zbxdbhigh/dbschema.c:CREATE TRIGGER history_sync_tr\n\
libs/zbxdbhigh/dbschema.c:BEFORE INSERT ON history_sync\n\
libs/zbxdbhigh/dbschema.c:SELECT history_sync_seq.nextval INTO :new.id FROM dual;\n\
libs/zbxdbhigh/dbschema.c:CREATE TABLE history_sync (\n\
libs/zbxdbhigh/dbschema.c:CREATE INDEX history_sync_1 ON history_sync (nodeid,id);\n\
libs/zbxdbhigh/dbschema.c:CREATE TABLE history_sync (\n\
libs/zbxdbhigh/dbschema.c:CREATE INDEX history_sync_1 ON history_sync (nodeid,id);\n\
libs/zbxdbupgrade/dbupgrade.c:     return DBmodify_proxy_table_id_field("history_sync");
libs/zbxdbupgrade/dbupgrade.c:     return DBdrop_index("history_sync", "id");
```

我们仔细看一下，dbschema 中的 CREATE 和 SELECT 应该不是我们需要的地方，应该是在 dbcache.c 中，我们看看代码，我把包含“history_sync”的地方都拿出来：

```
    Line 1342:    *               for writing float-type items into history/history_sync tables.      *
    Line 1349:    const char      *ins_history_sync_sql = "insert into history_sync
```

(nodeid,itemid,clock,ns,value) values ";

Line 1349: const char *ins_history_sync_sql = "insert into history_sync (nodeid,itemid,clock,ns,value) values ";

Line 1378: zbx_strcpy_alloc(&sql, &sql_alloc, sql_offset, ins_history_sync_sql);

Line 1390: zbx_strcpy_alloc(&sql, &sql_alloc, sql_offset, ins_history_sync_sql);

仔细的研究了一下，发现这里并不能看出这 sync 表的用途，于是再返回去看，发现了这一行：

```
libs/zbxdbhigh/dbschema.c:      {"history_str_sync",    "id",
ZBX_HISTORY_SYNC,
```

这一行的“ZBX_HISTORY_SYNC”，这种全部大写的命名方式，似乎有点可以挖掘的地方。

我们 grep 看看

```
$ grep -r 'ZBX_HISTORY_SYNC' *
libs/zbxdbhigh/dbschema.c:      {"history_sync",        "id",     ZBX_HISTORY_SYNC,
libs/zbxdbhigh/dbschema.c:              {"itemid",              NULL,       "items",
"itemid",       0,      ZBX_TYPE_ID,    ZBX_NOTNULL |
libs/zbxdbhigh/dbschema.c:              {"clock",       "0",    NULL,   NULL,   0,
ZBX_TYPE_INT,   ZBX_NOTNULL | ZBX_HISTORY_SYN
libs/zbxdbhigh/dbschema.c:              {"value",       "0.0000",       NULL,
NULL,   0,      ZBX_TYPE_FLOAT, ZBX_NOTNULL | ZBX_HIS
libs/zbxdbhigh/dbschema.c:              {"ns",  "0",    NULL,   NULL,   0,
ZBX_TYPE_INT,   ZBX_NOTNULL | ZBX_HISTORY_SYNC, 0},
libs/zbxdbhigh/dbschema.c:      {"history_uint_sync",   "id",     ZBX_HISTORY_SYNC,
libs/zbxdbhigh/dbschema.c:              {"itemid",              NULL,       "items",
"itemid",       0,      ZBX_TYPE_ID,    ZBX_NOTNULL |
libs/zbxdbhigh/dbschema.c:              {"clock",       "0",    NULL,   NULL,   0,
ZBX_TYPE_INT,   ZBX_NOTNULL | ZBX_HISTORY_SYN
libs/zbxdbhigh/dbschema.c:              {"value",       "0",    NULL,   NULL,   0,
ZBX_TYPE_UINT,  ZBX_NOTNULL | ZBX_HISTORY_SYN
libs/zbxdbhigh/dbschema.c:              {"ns",  "0",    NULL,   NULL,   0,
ZBX_TYPE_INT,   ZBX_NOTNULL | ZBX_HISTORY_SYNC, 0},
libs/zbxdbhigh/dbschema.c:      {"history_str_sync",    "id",     ZBX_HISTORY_SYNC,
libs/zbxdbhigh/dbschema.c:              {"itemid",              NULL,       "items",
"itemid",       0,      ZBX_TYPE_ID,    ZBX_NOTNULL |
libs/zbxdbhigh/dbschema.c:              {"clock",       "0",    NULL,   NULL,   0,
ZBX_TYPE_INT,   ZBX_NOTNULL | ZBX_HISTORY_SYN
libs/zbxdbhigh/dbschema.c:              {"value",       "",     NULL,   NULL,   255,
ZBX_TYPE_CHAR,  ZBX_NOTNULL | ZBX_HISTORY_SYN
libs/zbxdbhigh/dbschema.c:              {"ns",  "0",    NULL,   NULL,   0,
ZBX_TYPE_INT,   ZBX_NOTNULL | ZBX_HISTORY_SYNC, 0},
zabbix_server/nodewatcher/history.c:    if (0 != (table->flags & ZBX_HISTORY_SYNC))
zabbix_server/nodewatcher/history.c:            if (0 != (table->flags &
ZBX_HISTORY_SYNC) && 0 == (table->fields[f].flags & ZBX_HIST
```

zabbix_server/nodewatcher/history.c: if (0 != (table->flags & ZBX_HISTORY_SYNC))
zabbix_server/nodewatcher/history.c: if (0 != (table->flags & ZBX_HISTORY_SYNC))
zabbix_server/nodewatcher/history.c: if (0 != (table->flags & ZBX_HISTORY_SYNC) && 0 == (table->fields[f].flags &
zabbix_server/nodewatcher/history.c: if (0 != (table->flags & ZBX_HISTORY_SYNC))
zabbix_server/nodewatcher/history.c: if (0 == (t->flags & (ZBX_HISTORY | ZBX_HISTORY_SYNC)))
zabbix_server/trapper/nodehistory.c: if (0 != (table->flags & ZBX_HISTORY_SYNC))
zabbix_server/trapper/nodehistory.c: if (0 != (table->flags & ZBX_HISTORY_SYNC) && 0 == (table->fields[f].flags & ZBX_HIST
zabbix_server/trapper/nodehistory.c: if (0 != (table->flags & ZBX_HISTORY_SYNC))
zabbix_server/trapper/nodehistory.c: if (0 != (table->flags & ZBX_HISTORY_SYNC) && 0 == (table->fields[f].flags & ZBX_HIST
zabbix_server/trapper/nodehistory.c: if (0 != (table->flags & ZBX_HISTORY_SYNC) && 0 == (table->fields[f].flags & ZBX_HIST
zabbix_server/trapper/nodehistory.c: if (NULL != table && 0 == (table->flags & (ZBX_HISTORY | ZBX_HISTORY_SYNC)))
zabbix_server/trapper/nodehistory.c: if (NULL != table && 0 != (table->flags & ZBX_HISTORY_SYNC))

我们看下 grep 出来的代码，大部分的代码都是类似 0 != (table->flags & ZBX_HISTORY_SYNC)这样，这个是在判断这个表是不是 ZBX_HISTORY_SYNC。这时，我们发现，包含“ZBX_HISTORY_SYNC”的文件除了 dbschema.c 外，就是 nodewatcher/history.c 和 nodehistory.c，大家应该明白了吧，这个表，是在 Master-Child 架构中使用的。

我们先看 nodewatcher/history.c，一共有哪些 function 使用了 ZBX_HISTORY_SYNC 和这些方法的说明：

1. process_history_table_data：process new history data，意为处理新的历时数据
2. process_history_tables：process new history data from tables with ZBX_HISTORY* flags，意为处理带有 ZBX_HISTORY*标识的表的数据

再看 history.c 的主函数 main_historysender，它的说明是“periodically sends historical data to master node ”，即“周期性的将历史数据从子节点发送到父节点。

看到这里我们应该能明白了，带有 sync 后缀的 history 表，是用作 Master-Child 同步数据用的。那我们在看 history 和 trends 的区别的时候，把 sync 相关的表现略去了。

总结一下，history 和 trends 表在数据库中就可以看成两张表，history 表和 trends 表。

2.2 history 和 trends 的区别

上一小节中，我们主要把 history 在数据库中的几个表刨根问底了一番，把数据库中的相关表分为了两类——history 表和 trends 表。

他们的相同点是他们都是存储历史数据的，不同点是，他们存储数据的粒度不同。每一次 Zabbix 接收到 Item 的数据以后，会将其存入 history 表。下面是 history 表结构：

```
mysql> desc history;
+--------+---------------------+------+-----+---------+-------+
| Field  | Type                | Null | Key | Default | Extra |
+--------+---------------------+------+-----+---------+-------+
| itemid | bigint(20) unsigned | NO   | MUL | NULL    |       |
| clock  | int(11)             | NO   |     | 0       |       |
| value  | double(16,4)        | NO   |     | 0.0000  |       |
| ns     | int(11)             | NO   |     | 0       |       |
+--------+---------------------+------+-----+---------+-------+
```

这个是很简单的结构，clock 和 ns 保存着接收 item 的时间，itemid 唯一标识了 item，value 即为接收到的数据。

而 trends 表的作用是将 history 表的数据根据小时的维度进行归档。它会针对每一个 itemid，计算每个小时的最小值，最大值和平均值，我们看 trends 表结构：

```
mysql> desc trends;
+-----------+---------------------+------+-----+---------+-------+
| Field     | Type                | Null | Key | Default | Extra |
+-----------+---------------------+------+-----+---------+-------+
| itemid    | bigint(20) unsigned | NO   | PRI | NULL    |       |
| clock     | int(11)             | NO   | PRI | 0       |       |
| num       | int(11)             | NO   |     | 0       |       |
| value_min | double(16,4)        | NO   |     | 0.0000  |       |
| value_avg | double(16,4)        | NO   |     | 0.0000  |       |
| value_max | double(16,4)        | NO   |     | 0.0000  |       |
+-----------+---------------------+------+-----+---------+-------+
```

itemid，clock 和 value_min，value_avg，value_max 都容易理解，num 这个字段表示了该小时使用了多少数据来计算最小值、最大值和平均值。

2.3 housekeeper 和 trends 表

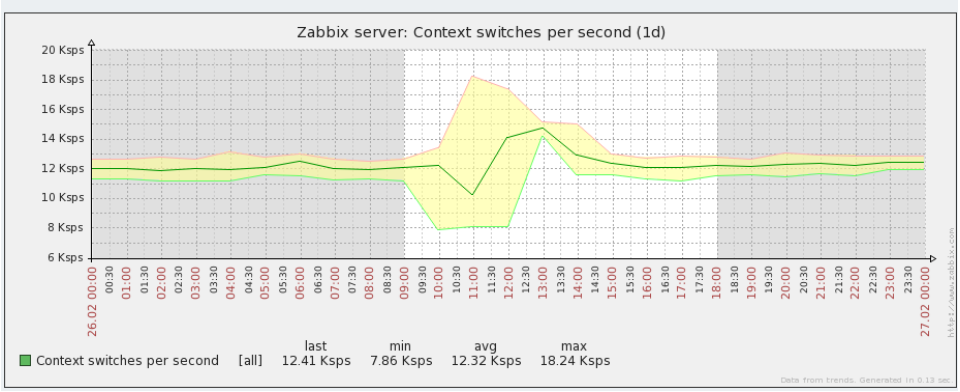
history 表的作用大家已经清楚了，就是存储所有 item 的历史数据。不知道大家记得不记得，衡量 Zabbix 的性能有一个重要的指标，就是 vps——Value Per Second。我们看一下如果 vps 为 1000 的时候，即每秒 Zabbix 要处理 1000 个数据，每一个数据都会在 history 表中有一行，我们算一下，一天有 86400 秒，每秒 1000 行数据，那么每天就是 86400000 行，8 千万行！如果一直这样无限制膨胀下去，history 表的性能会非常的差，从而拖累整个 Zabbix 数据库的性能。

应对这个问题，Zabbix 提供了两方面的解决办法，一个是 housekeeper 机制，另一个则是 trends 机制。

housekeeper 非常简单，就是定期删除 history 表中的数据。对于小规模的 Zabbix 来说，这是个省时省力的方法，但对于大规模的 Zabbix 数据库，使用 housekeeper 效率非常的差，特别

是 InnoDB 引擎的 MySQL，因为大表的删除非常的慢。针对规模很大的 Zabbix，我的建议是每周 truncate 一次 history 相关的表，大家可能担心这么暴力的删除会不会影响数据完整性，别着急，我们看完下面 trends 表的作用，大家就明白了。

trends 表的工作原理，在本章的前面几节大家应该已经理解清楚了。这里我们看看 trends 在 Zabbix 的用处。其实在 Zabbix 中，trends 只在很少几个地方出现，最重要的就是在 Graph 中了：



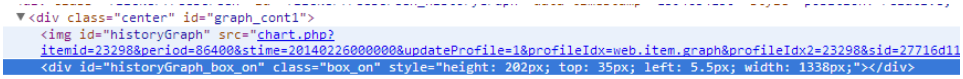
我们看某个时间点，图中其实有三个数据。从小到大分别是这个时间点的最小值，平均值和最大值。这里调用的就是 trends 表的数据。我们仔细想一想，history 的数据真的要保存这么久吗？对于上个星期的数据，我们真的要精确到每一分钟的数据吗？答案是不需要的，对于时间越久的数据，我们需要的粒度就更粗。

2.4 Graph 对于 history 和 trends 的选择

在这一节中，我们会分析 Graph 选择从 history 表选数据，还是从 trends 表选数据。

首先先看看这个图是怎么画出来的，我们先进入一个 item 的 graph（注意，这里不要选择 Monitoring-Graph 中的 graph，选择 Monitoring-Latest Data 中某个 item 的 graph），这里使用了 chrome 的查看元素功能，右击页面上的 graph，选择“查看元素”：

会看到这个元素在 html 中对应的内容，如下图是我环境中的截图：



把这个链接在新窗口打开就可以看到单独的一个 graph 了。从 url 分析，这个是由 chart2.php 绘制的，我们看下 chart.php 代码中绘制 graph 的代码片段：

```
/*
 * Display
```

```
 */
$timeline = CScreenBase::calculateTime(array(
	'profileIdx' => get_request('profileIdx', 'web.screens'),
	'profileIdx2' => get_request('profileIdx2'),
	'updateProfile' => get_request('updateProfile', true),
	'period' => get_request('period'),
	'stime' => get_request('stime')
));

$graph = new CChart();
$graph->setPeriod($timeline['period']);
$graph->setSTime($timeline['stime']);

if (isset($_REQUEST['from'])) {
	$graph->setFrom($_REQUEST['from']);
}
if (isset($_REQUEST['width'])) {
	$graph->setWidth($_REQUEST['width']);
}
if (isset($_REQUEST['height'])) {
	$graph->setHeight($_REQUEST['height']);
}
if (isset($_REQUEST['border'])) {
	$graph->setBorder(0);
}
$graph->addItem($_REQUEST['itemid'], GRAPH_YAXIS_SIDE_DEFAULT,
CALC_FNC_ALL);
$graph->draw();
```

代码还是比较容易看懂的，大多数的代码都是在设置这个 graph 的参数，比如高度 height 和宽度 width 等。那数据从什么地方来呢？数据无论是从 history 还是 trends 来，肯定需要一个东西，就是 itemid。按照这个思路，我们发现了一行代码：

```
$graph->addItem($_REQUEST['itemid'], GRAPH_YAXIS_SIDE_DEFAULT,
CALC_FNC_ALL);
```

这里的 addItem 是$graph 的一个方法，而$graph 是 CChart 类的一个实例：

```
$graph = new CChart();
```

找到 CChart 定义的地方：

```
$ grep -r CChart *
chart.php:$graph = new CChart();
chart2.php:$graph = new CChart($dbGraph['graphtype']);
chart3.php:	$graph = new CChart(get_request('graphtype',
GRAPH_TYPE_NORMAL));
include/classes/class.cchart.php:class CChart extends CGraphDraw {
```

我们接着看 class.cchart.php 中的 addItem 方法：

```
public function addItem($itemid, $axis = GRAPH_YAXIS_SIDE_DEFAULT, $calc_fnc = CALC_FNC_AVG, $color = null, $drawtype = null, $type = null) {
        if ($this->type == GRAPH_TYPE_STACKED) {
            $drawtype = GRAPH_ITEM_DRAWTYPE_FILLED_REGION;
        }

        $item = get_item_by_itemid($itemid);
        $this->items[$this->num] = $item;
        $this->items[$this->num]['name'] = itemName($item);
        $this->items[$this->num]['delay'] = getItemDelay($item['delay'], $item['delay_flex']);

        if (strpos($item['units'], ',') !== false) {
            list($this->items[$this->num]['units'], $this->items[$this->num]['unitsLong']) = explode(',', $item['units']);
        }
        else {
            $this->items[$this->num]['unitsLong'] = '';
        }

        $host = get_host_by_hostid($item['hostid']);

        $this->items[$this->num]['hostname'] = $host['name'];
        $this->items[$this->num]['color'] = is_null($color) ? 'Dark Green' : $color;
        $this->items[$this->num]['drawtype'] = is_null($drawtype) ? GRAPH_ITEM_DRAWTYPE_LINE : $drawtype;
        $this->items[$this->num]['axisside'] = is_null($axis) ? GRAPH_YAXIS_SIDE_DEFAULT : $axis;
        $this->items[$this->num]['calc_fnc'] = is_null($calc_fnc) ? CALC_FNC_AVG : $calc_fnc;
        $this->items[$this->num]['calc_type'] = is_null($type) ? GRAPH_ITEM_SIMPLE : $type;

        if ($this->items[$this->num]['axisside'] == GRAPH_YAXIS_SIDE_LEFT) {
            $this->yaxisleft = 1;
        }

        if ($this->items[$this->num]['axisside'] == GRAPH_YAXIS_SIDE_RIGHT) {
            $this->yaxisright = 1;
        }
        $this->num++;
    }
```

我们可以从头看下这个 addItem 方法的代码，它的工作是将 items 表中的数据取出来。那我们还是没有找到是从 history 表还是 trends 表的数据。没办法，只能看看 draw()方法了。draw()方法比较长，这里就不全部贴出来了，大家可以看下代码，大致了解下 draw 方法的大致流程。

在浏览完 draw 方法后，可以得到这样的结论，draw 方法是根据 item 的数据来绘制每一个点，而 item 数据的获取，就在 draw 方法的一开始：
$this->selectData();

我们看看同样在 class.cchart.php 中的 selectData 方法，找到这样一行：
if (($real_item['history'] * SEC_PER_DAY) > (time() - ($this->from_time + $this->period / 2)) && ($this->period / $this->sizeX) <= (ZBX_MAX_TREND_DIFF

这里有两条逻辑，加上代码中的注释，整理如下：

1. ($real_item['history'] * SEC_PER_DAY) > (time() - ($this->from_time + $this->period / 2)，注释为“should pick data from history or trends”，即“是从 history 还是 trends 获取数据”
2. ($this->period / $this->sizeX) <= (ZBX_MAX_TREND_DIFF / ZBX_GRAPH_MAX_SKIP_CELL)，注释为“is reasonable to take data from history?”，即“从 history 获取数据是否可行”

它们的含义分别为：

1. item 保存 history 的天数一共的秒数大于当前时间减去 graph 开始的时间和持续时间一半的和。
2. graph 的持续时间除以 X 轴的长度（像素）小于等于 ZBX_MAX_TREND_DIFF / ZBX_GRAPH_MAX_SKIP_CELL。其中持续时间除以 X 轴的长度表示的意思是 X 轴每一个像素上显示的时间长度。而 ZBX_GRAPH_MAX_SKIP_CELL 表示的含义是当多少个像素上没有数据的时候，会显示一个断点。组合起来，这个逻辑就是：每一个像素显示的时间长度小于等于断点时候每个像素的时间长度

总的来说，除了 history 表还使用 trends 表来绘制 graph，原因就是防止一个像素点的数据过多

When you view graphs of a period greater than a week the long term history table is used as a data source to generate the graphs. I also think you're seeing a scaling issue. There is too much data to pack into a graph and as such the data must be scaled appropriately. Each pixel will cover a large expanse of time and may also cover a wide array of data. As such extreme high and extreme low data points are likely to be missed unless when drawing the graph, hence why the data is also given in the table. Try viewing the graph one week at a time and you will likely see better data resolution. This is not a fault of Zabbix, but a problem for any graph.

批注 [d3]: 用一个例子补充一下

批注 [d4]: 确认下最好

批注 [d5]: 仔细看下，对于条件 2 要搞清楚