3 自动发现□LLD□

概述

自动发现□LLD□提供了一种在计算机上为不同实体自动创建监控项，触发器和图形的方法。例如□Zabbix可以在你的机器上自动开始监控文件系统或网络接口，而无需为每个文件系统或网络接口手动创建监控项。此外，可以配置Zabbix根据定期执行发现后的得到实际结果，来移除不需要的监控项。

在Zabbix中，支持六种类型的发现项目：

- 系统文件的发现；
- 网络接口的发现；
- CPU和CPU内核的发现
- SNMP OID的发现
- 使用ODBC SQL查询的发现
- Windows服务的发现

用户可以自己定义发现类型，只要它们遵循特定的JSON协议。

发现过程的一般架构如下。

首先，用户在“配置”→“模板”→“发现”列中创建一个发现规则。发现规则包括（1）发现必要实体（例如，文件系统或网络接口）的项目和（2）应该根据该项目的值创建的监控项，触发器和图形的原型

发现必要实体的项目就像其他地方所看到的常规项目：服务器向该项目的值询问Zabbix agent□或者该项目的任何类型的设置□□agent以文本值进行响应。区别在于agent响应的值应该包含特定JSON格式的发现实体的列表。这种格式的自定义检查者发现的细节才是最重要的，因为返回值必须包含宏→值对。例如，项目“net.if.discovery”可能会返回两对键值□“{#IFNAME}”→“lo”和 “{#IFNAME}”→“eth0”□

Zabbix agent版本2. 0支持自动发现项目“vfs.fs.discovery”和 “net.if.discovery”□
从Zabbix agent版本2. 4起支持发现项目“system.cpu.discovery”□
从Zabbix server和proxy版本2. 0起支持发现SNMP OID□
从Zabbix server和proxy版本3. 0起支持使用ODBC SQL查询的发现。
在使用IBM DB2数据库运行的Zabbix proxy上，自动发现规则的返回值限制为2048字节。此限制不适用于Zabbix server□因为返回值不会被存储在数据库中。

这些宏用于名称，键值和其他原型字段中，然后用接收到的值为每个发现的实体创建实际的监控项，触发器，图形甚至主机。请参阅使用LLD宏的选项的完整列表。

当服务器接收到发现项目的值时，它会查看宏→值对，每对都根据原型生成实际监控项，触发器和图形。在上面的“net.if.discovery”示例中，服务器将生成环路接口“lo”的一组监控项，触发器和图表，另一组用于界面“eth0”□

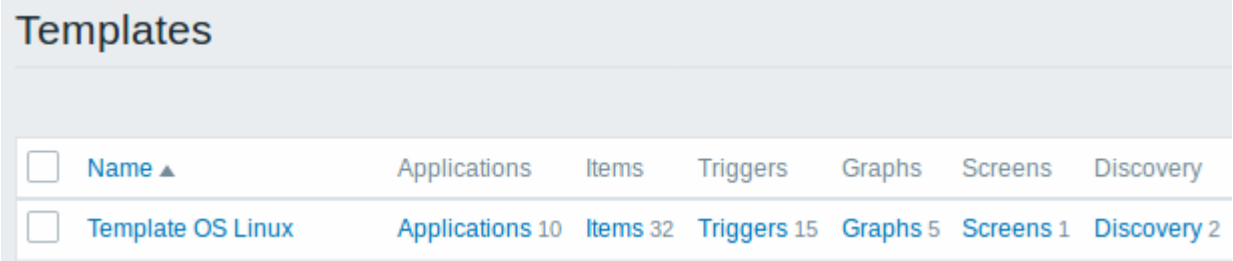
以下部分将详细说明上述过程，并作为一个指导上述类型的所有发现。最后一节描述了发现项目的JSON格式，并给出了文件系统发现实现的Perl脚本的示例。

3. 1 文件系统的发现

要配置文件系统的发现，请执行以下操作：

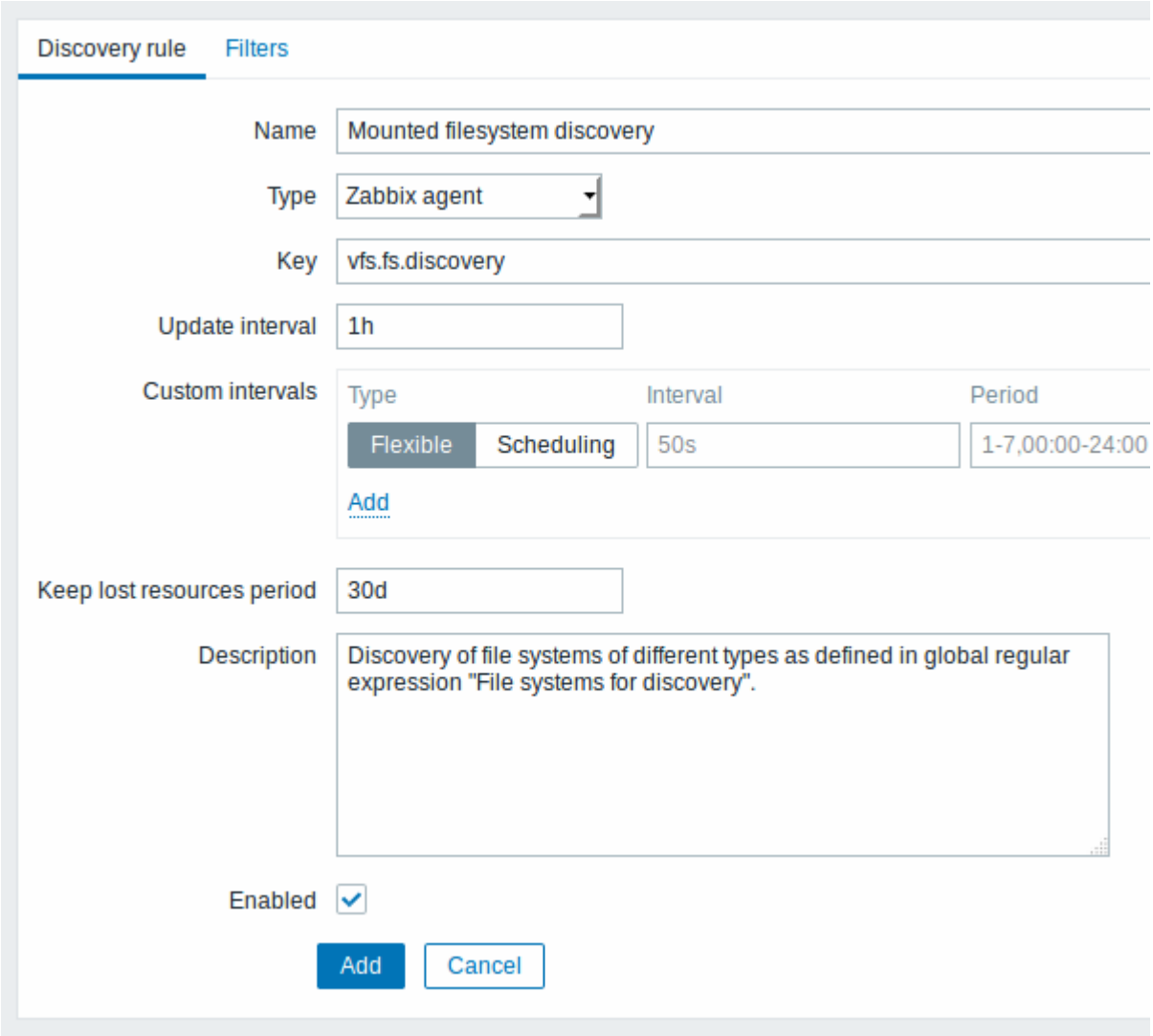
- 转到：*配置* → *模板*

- 在一个合适的模板的行点击*发现*



- 单击屏幕右上角的*创建发现规则*
- 填写以下详细信息。

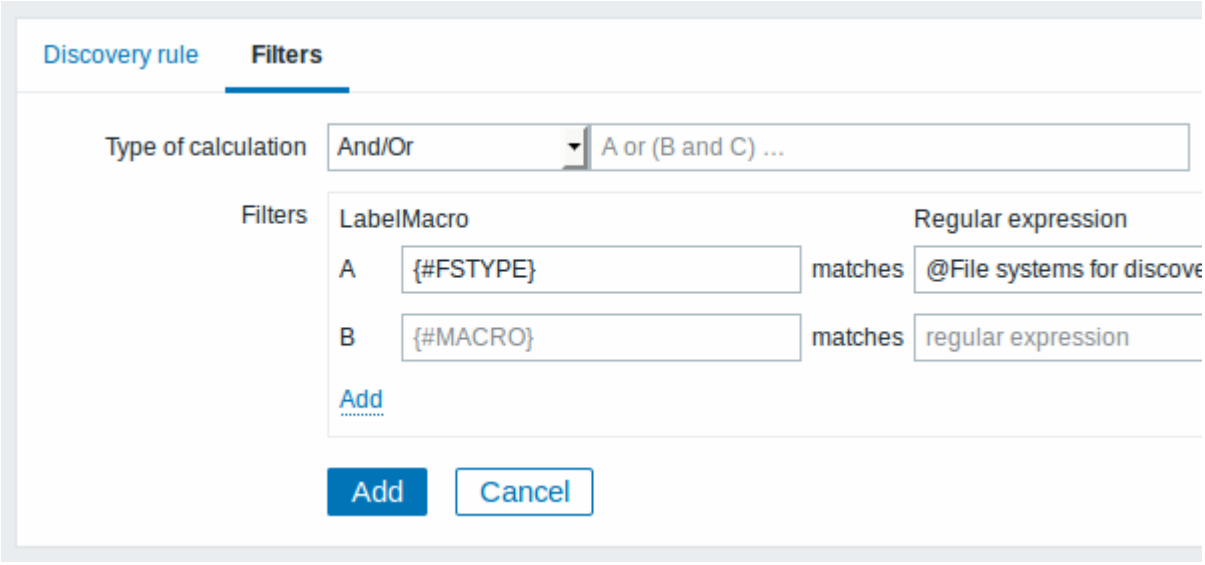
发现规则选项卡包含常规发现规则属性：



参数	**描述**
名称	发现规则名称。
类型	执行发现的检查类型；可以是 *Zabbix agent*或*Zabbix agent*（*主动*）文件系统发现。
键值	许多平台上的Zabbix agent程序内置了“vfs.fs.discovery”键值的项目（有关详细信息，请参阅 支持的项目键列表），并将返回一个JSON其中包含计算机上存在的文件系统列表及其类型。

参数	描述
数据更新间隔(秒)	此字段设置Zabbix执行发现的频率。一开始，当你只是设置文件系统发现时，您可能希望将其设置为段间隔时间，但一旦发现它可以将其设置为30分钟或更长时间，因为文件系统通常不会更改。 *注意*：如果设置为“0”，则不会轮询该项。但是，如果灵活间隔也存在非零值，则在灵活间隔持续时间内将轮询该项。
自定义时间间隔	您可以创建用于检查项目的自定义规则： **灵活** - 创建更新*间隔*（不同频次的间隔）的 **调度** - 创建自定义轮询调度。 有关详细信息，请参阅自定义时间间隔。从Zabix 3.0.0起支持调度
保留失去的资源期间(天)	该字段允许你设置发现的实体将被发现状态变为“不再支持”（最多3650天）后将被保留（不会被删除）的天数。 *注意*：如果设置为“0”，将立即删除实体。不建议使用“0”，因为错误地编辑过滤器可能会在实体中删除所有的历史数据。
描述	输入说明文字。
已启用	如果选中，该规则将被执行。

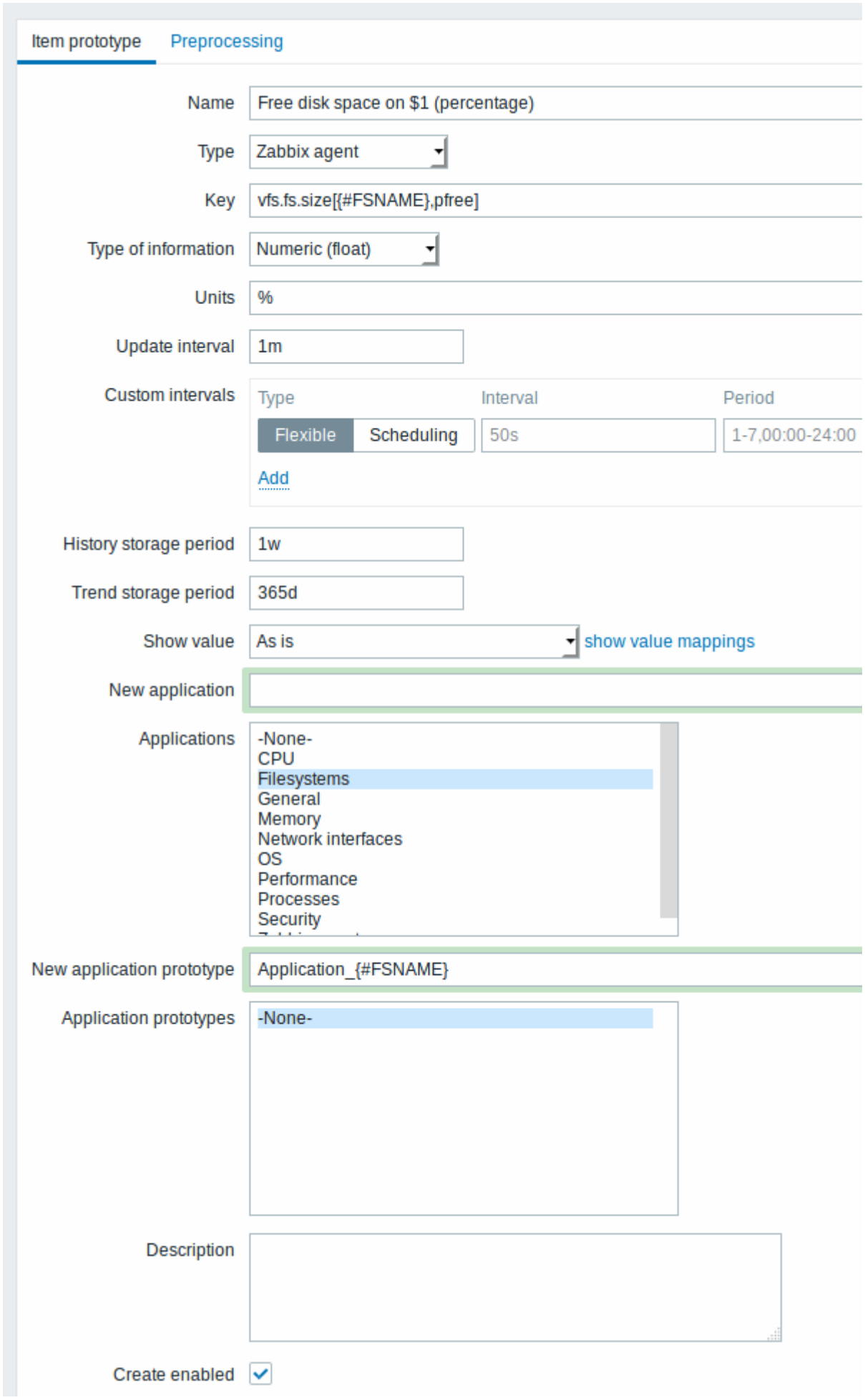
过滤器选项卡包含发现规则过滤器定义：



参数	描述
计算方式	计算过滤器的可用选项如下： **与** -所有过滤器满足； **或** - 只需一个过滤器满足； **与/或** - 不同的宏名称用*与*，相同的宏名称用*或*; **自定义表达式** - 提供定义自定义计算的过滤器的。该公式必须包括列表中的所有过滤器。限于255个符号
过滤器	过滤器可用于仅为特定文件系统生成实际监控项，触发器和图形。它支持POSIX扩展正则表达式。例如，如果你只对C :, D :,和E□文件系统感兴趣，则可以将{#FSNAME}放入“宏”和"^C|^D|^E"正则表达式到“正则表达式”文本字段。也可以使用{#FSTYPE}宏（例如"^ext|^reiserfs"□的文件系统类型以及使用{#FSDRIVETYPE}宏（例如"fixed"□的驱动器类型（仅由Windows agent支持）进行过滤。你可以在“正则表达式”字段中输入正则表达式或引用全局 正则表达式□ 为了测试正则表达式，你可以使用“grep -E”□例如： `for f in ext2 nfs reiserfs smbfs; do echo $f | grep -E '^ext|^reiserfs' || echo "SKIP: $f"; done` 从Zabbix **3.0.0**起支持Windows上的宏{#FSDRIVETYPE}□ Zabbix **2.4.0**起支持定义多个过滤器。 注意，如果响应中缺少过滤器中的某些宏，则找到的实体将被忽略。.

如果要正确发现不同的文件系统名称，则必须将MySQL中的Zabbix数据库创建为区分大小写。
发现规则历史记录不被保留。

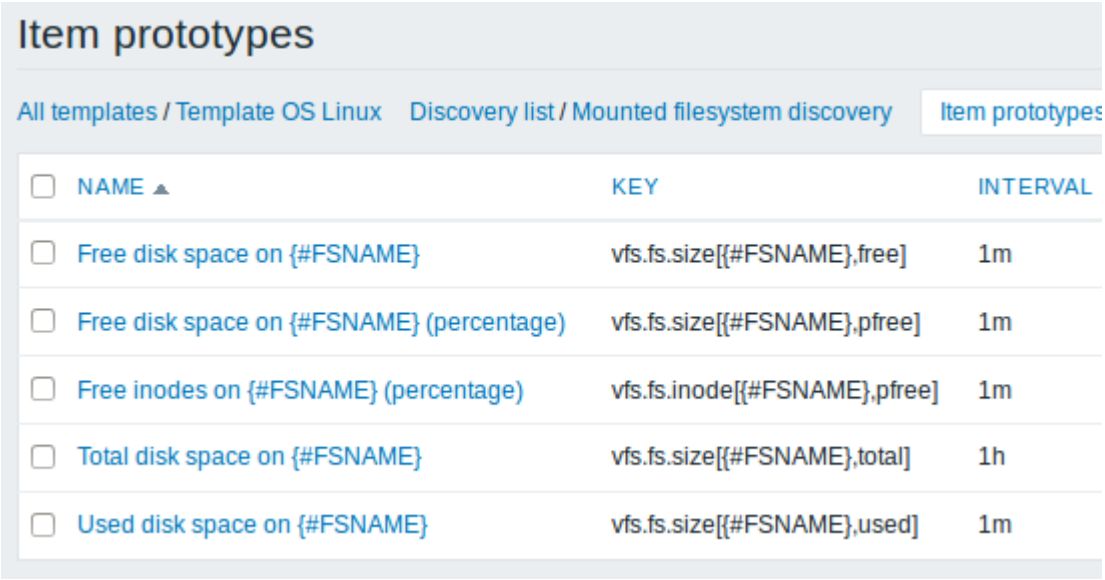
创建规则后，转到该规则的项目，然后点击“创建监控项原型”创建项目原型。请注意在需要文件系统名称时使用宏{#FSNAME}的宏。当发现规则被处理时，该宏将被替换为发现的文件系统。


Item prototype
Preprocessing
Name
Free disk space on $1 (percentage)
Type
Zabbix agent
Key
vfs.fs.size[{#FSNAME},pfree]
Type of information
Numeric (float)
Units
%
Update interval
1m
Custom intervals
Type
Interval
Period
Flexible
Scheduling
50s
1-7,00:00-24:00
Add
History storage period
1w
Trend storage period
365d
Show value
As is
show value mappings
New application
Applications
-None-
CPU
Filesystems
General
Memory
Network interfaces
OS
Performance
Processes
Security
New application prototype
Application_{#FSNAME}
Application prototypes
-None-
Description
Create enabled

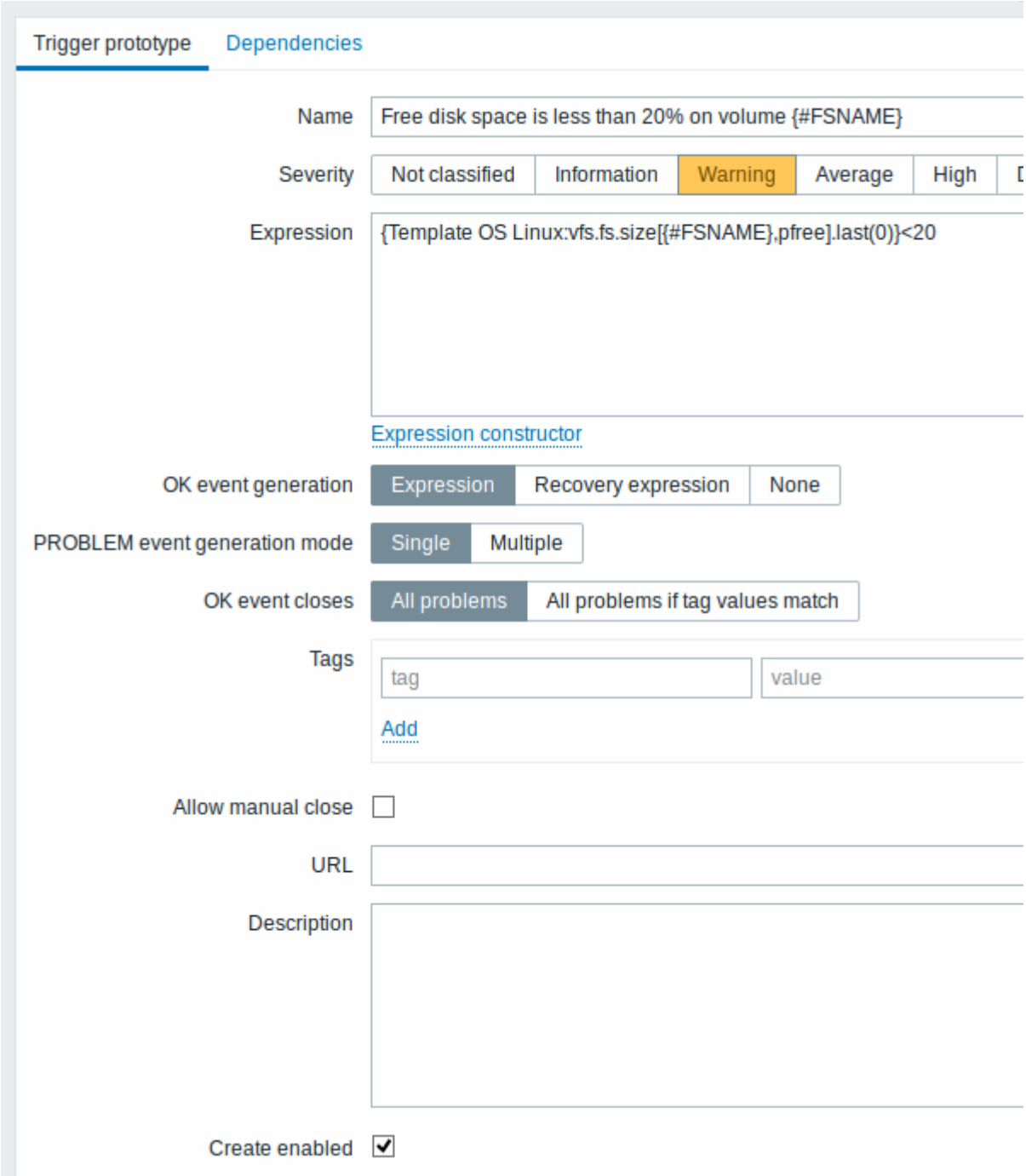
监控项原型特有的属性：

参数	描述
新应用原型	您可以定义一个新的应用原型。 在应用原型中，你可以使用自动发现□LLD□宏，在发现后，将用实际值替换创建特定于发现实体的应用。有关更多具体信息，请参阅应用发现说明□
应用原型	从现有应用原型中选择。
创建已启用	如果选中，项目将被添加到启用状态。 如果未选中，该项目将被添加到已发现的实体，但处于禁用状态。

我们可以为我们感兴趣的每个文件系统度量创建几个项目原型：



然后，我们以类似的方式创建触发器原型：



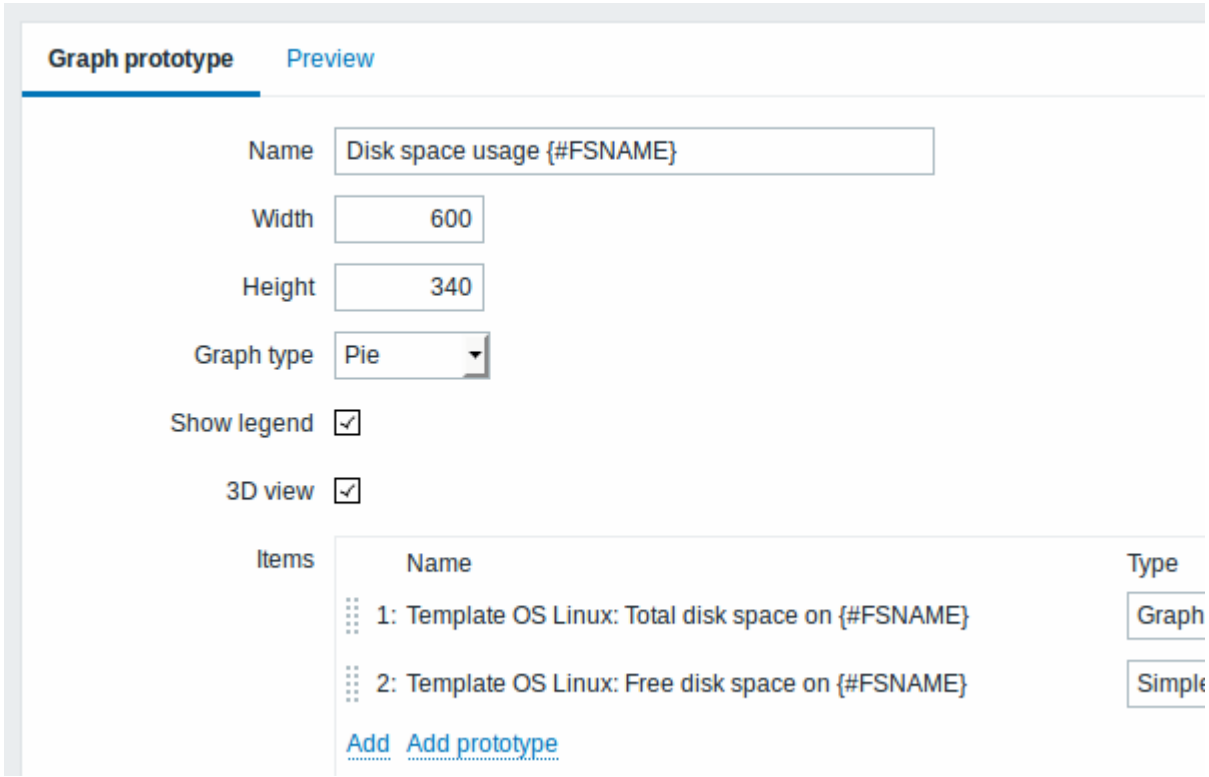
触发原型特有的属性：

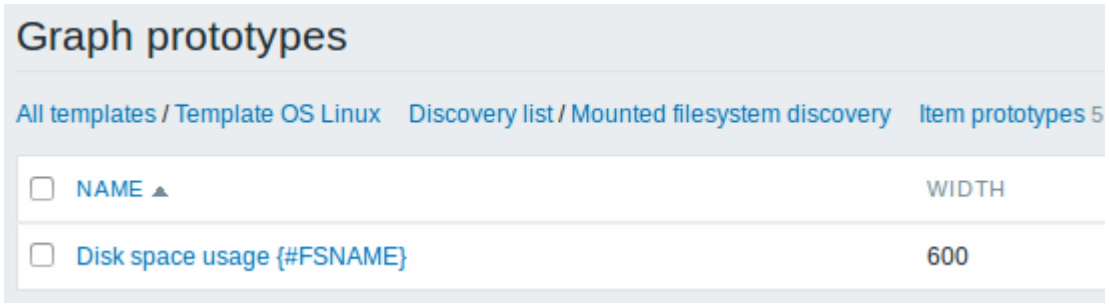
参数	描述
创建启用	如果选中，触发器将被添加到启用状态。 如果未选中，触发器将被添加到已发现的实体，但处于禁用状态。

当从原型创建真实触发器时，对表达式中使用什么常量（在我们的示例中为'20'）是比较灵活的。了解具有上下文的用户宏可以实现这种灵活性。

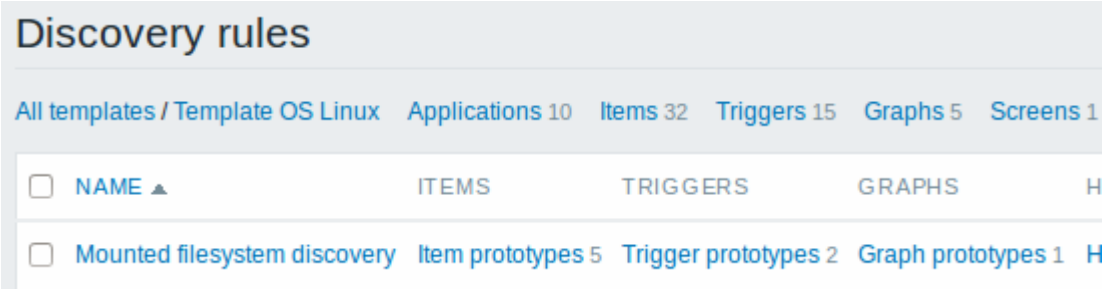
在*依赖关系*选项卡，也可以定义触发器原型之间的依赖关系（自Zabbix 3.0起支持）。触发器原型可以依赖于来自相同自动发现□LLD□规则另一个触发器原型或常规触发器。触发原型也可不依赖于不同的LLD规则触发原型或常规触发器的产生的触发器原型。。主机触发器原型不能依赖于模板的触发器。

我们也可以创建图形原型：



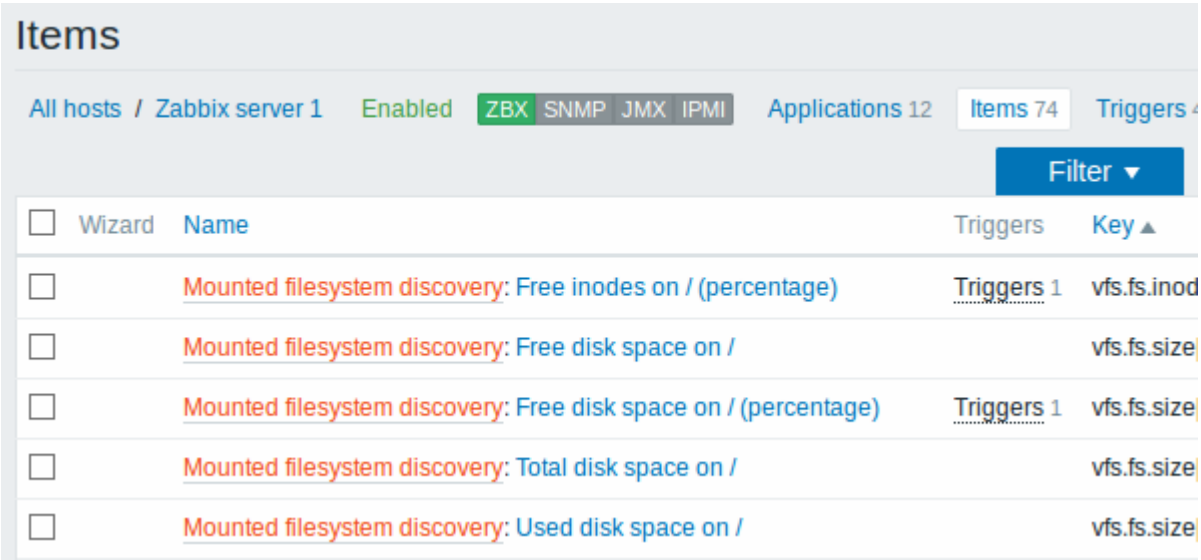


最后，我们创建了一个发现规则，如下图所示。它有五个监控项目原型，两个触发器原型和一个图形原型。



注意：有关配置主机模板，请参阅虚拟机监控中有关主机模板配置的部分。

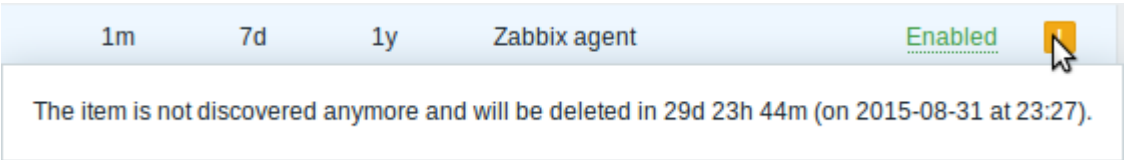
下面的屏幕截图说明了主机配置中发现的监控项，触发器和图形的样子。发现的实体前缀有橙色链接到他们来自的发现规则。



请注意，如果已经存在具有相同唯一性条件的现有实体，例如具有相同键值或具有相同名称的图形的项目，则不会创建发现的实体。

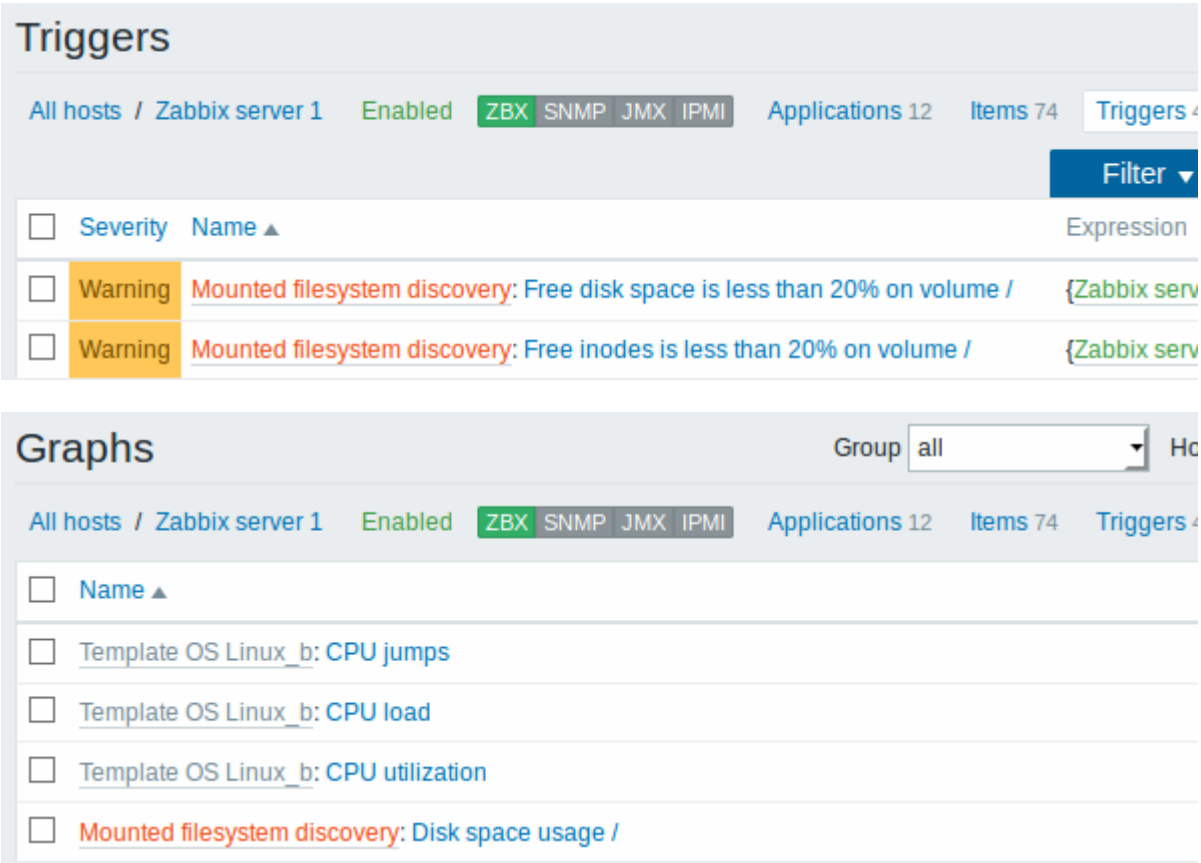
如果发现的实体（文件系统，接口等）停止发现（或不再通过过滤器），则由自动发现规则□LLD□创建的项目（类似地，触发器和图形）将被自动删除。这时，监控项，触发器和图表将在*保留失去的资源期间*字段中定义的日期过去后被删除。

当发现的实体变为“不再支持”时，项目列表中将显示生命周期指示符。将鼠标指针移动到其上，并显示一条消息，指示在删除项目之前剩下多少天。。



如果实体被标记为删除，但未在预期时间被删除（禁用的发现规则或项目主机），则在下次发现规则被处理时，它们将被删除。

标记为删除的其他实体的实体，如果在发现规则级别上更改，则不会更新。例如，如果基于LLD的触发器标记为要删除的项目，则它们将不会更新。



3.2 网络接口的发现

网络接口的发现与文件系统的发现完全相同，只是你使用发现规则的键值是“net.if.discovery”而不是“vfs.fs.discovery”，并使用宏{#IFNAME}而不是{#FSNAME}。

你可能希望基于“net.if.discovery”创建的监控项原型示例：“net.if.in[{#IFNAME},bytes]”，“net.if.out[{#IFNAME},bytes]“。

有关过滤器的更多信息，请参阅上文。

3.3 CPU和CPU内核的发现

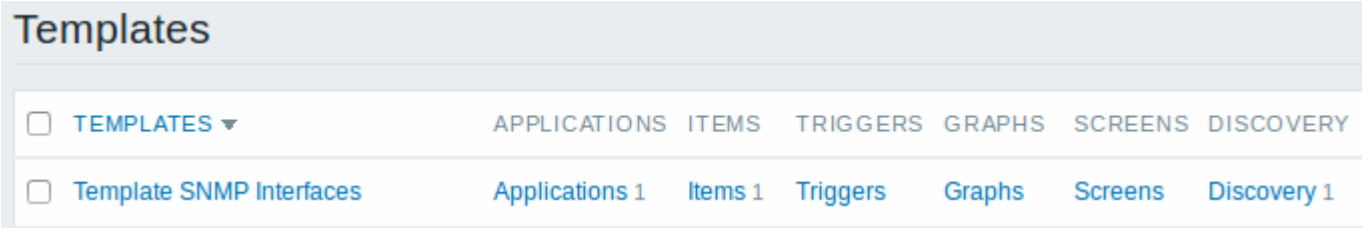
CPU和CPU内核的发现以与网络接口发现类似的方式完成相同，除了发现规则的键值是“system.cpu.discovery”之外。此发现键返回两个宏 - {#CPU.NUMBER}和{#CPU.STATUS}分别标识CPU序号和状态。要注意，实际物理处理器，内核和超线程之间不能做出明确的区分。在Linux、UNIX和BSD系统上的{#CPU.STATUS}返回处理器的状态，可以是“online”还是“offline”。在Windows系统上，同一个宏可能表示第三个值 - “unknown” - 这表示处理器已被检测到，但尚未收集到任何信息

CPU发现依赖于agent的收集器进程去收集和获取数据。如果agent的测试-t命令行不起作用，这将返回一个NOT_SUPPORTED状态以及附带的消息，表明收集器进程尚未启动。

可以基于CPU发现规则创建的监控项原型包括例如： “system.cpu.util[{#CPU.NUMBER}, <type>, <mode>]” 或 “system.hw.cpu[{#CPU.NUMBER}, <info>]“。

3.4 SNMP OID的发现

例如，我们将在交换机上执行SNMP发现。首先，进入“配置”→“模板”。



要编辑模板的发现规则，请单击“自动发现”列中的链接。

然后，按“创建发现规则”，并在下面的屏幕截图中填写表单与详细信息。

与文件系统和网络接口发现不同，项目不一定必须具有“snmp.discovery”键值 - SNMP agent的项目类型就足够了。

要发现的OID在SNMP OID字段中以下列格式定义： `discovery[{#MACRO1}, oid1, {#MACRO2}, oid2, …,]`

其中*{#MACRO1}*, *{#MACRO2}* …是有效的lld宏名称和*oid1*, *oid2*… 是能够为这些宏生成有意义的值的OID□已发现OID索引的内置宏{#SNMPINDEX}将应用于发现的实体。发现的实体按{#SNMPINDEX}宏值分组。

为了理解清楚，在我们的交换机上执行几个snmpwalk□

```
$ snmpwalk -v 2c -c public 192.168.1.1 IF-MIB::ifDescr
IF-MIB::ifDescr.1 = STRING: WAN
IF-MIB::ifDescr.2 = STRING: LAN1
IF-MIB::ifDescr.3 = STRING: LAN2

$ snmpwalk -v 2c -c public 192.168.1.1 IF-MIB::ifPhysAddress
IF-MIB::ifPhysAddress.1 = STRING: 8:0:27:90:7a:75
IF-MIB::ifPhysAddress.2 = STRING: 8:0:27:90:7a:76
IF-MIB::ifPhysAddress.3 = STRING: 8:0:27:2b:af:9e
```

并将SNMP OID设置为： `discovery[{#IFDESCR}, ifDescr, {#IFPHYSADDRESS}, ifPhysAddress]`

现在，这个规则会发现设置为{#IFDESCR}宏实体WAN□LAN1和LAN2□{#IFPHYSADDRESS}宏设置为8□0□27□90□7A□75□8□0□27□90□7A□76和8□0□27□2B□AF□9E□{#SNMPINDEX}宏设定为所发现的OID索引1，2和3：

```
{
    "data": [
        {
            "{#SNMPINDEX}": "1",
            "{#IFDESCR}": "WAN",
            "{#IFPHYSADDRESS}": "8:0:27:90:7a:75"
        },
        {
```

```
            "{#SNMPINDEX}": "2",
            "{#IFDESCR}": "LAN1",
            "{#IFPHYSADDRESS}": "8:0:27:90:7a:76"
        },
        {
            "{#SNMPINDEX}": "3",
            "{#IFDESCR}": "LAN2",
            "{#IFPHYSADDRESS}": "8:0:27:2b:af:9e"
        }
    ]
}
```

如果一个实体没有指定的OID则该实体将忽略相应的宏。例如，如果我们有以下数据：

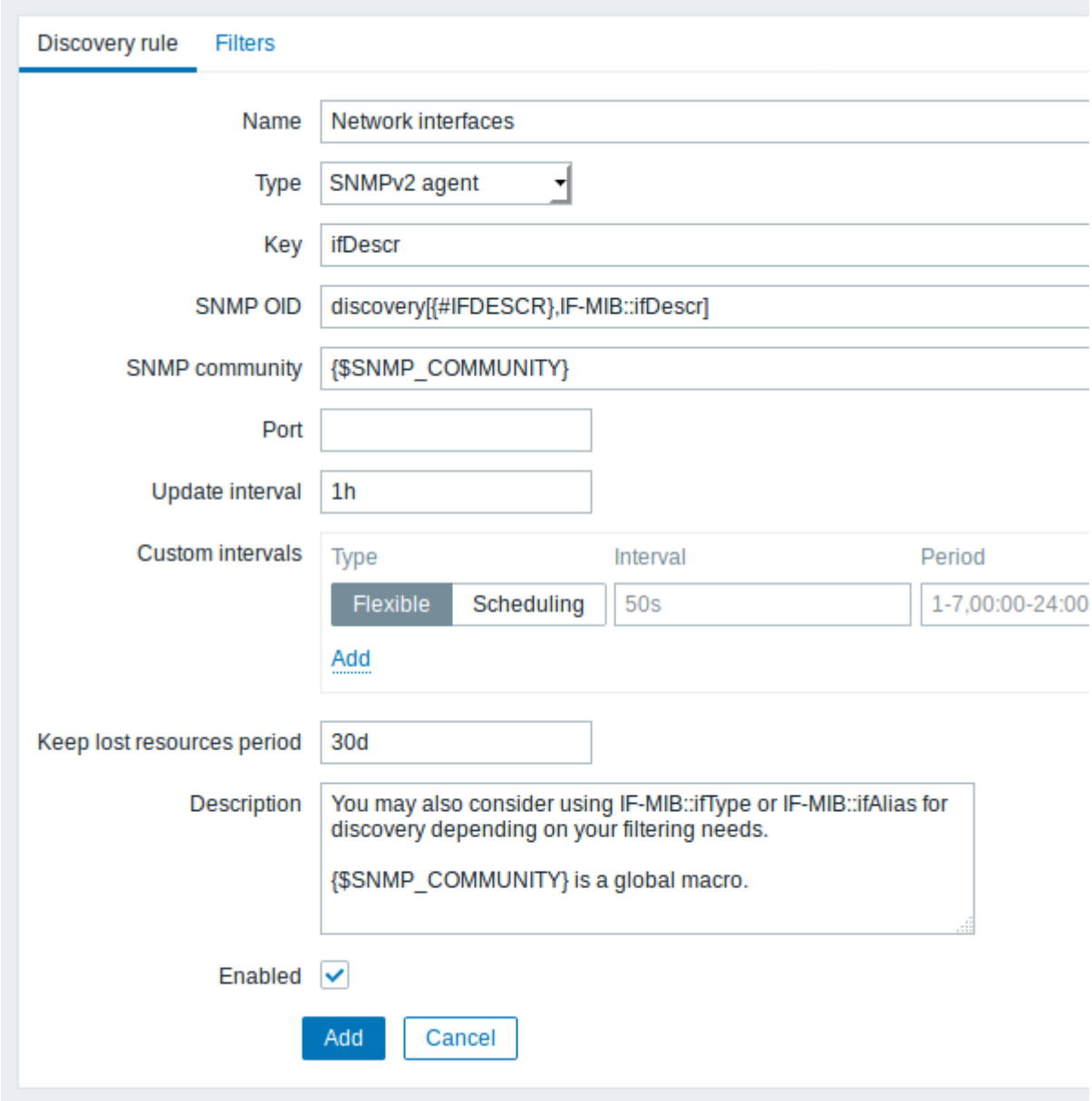
```
ifDescr.1 "Interface #1"
ifDescr.2 "Interface #2"
ifDescr.4 "Interface #4"

ifAlias.1 "eth0"
ifAlias.2 "eth1"
ifAlias.3 "eth2"
ifAlias.5 "eth4"
```

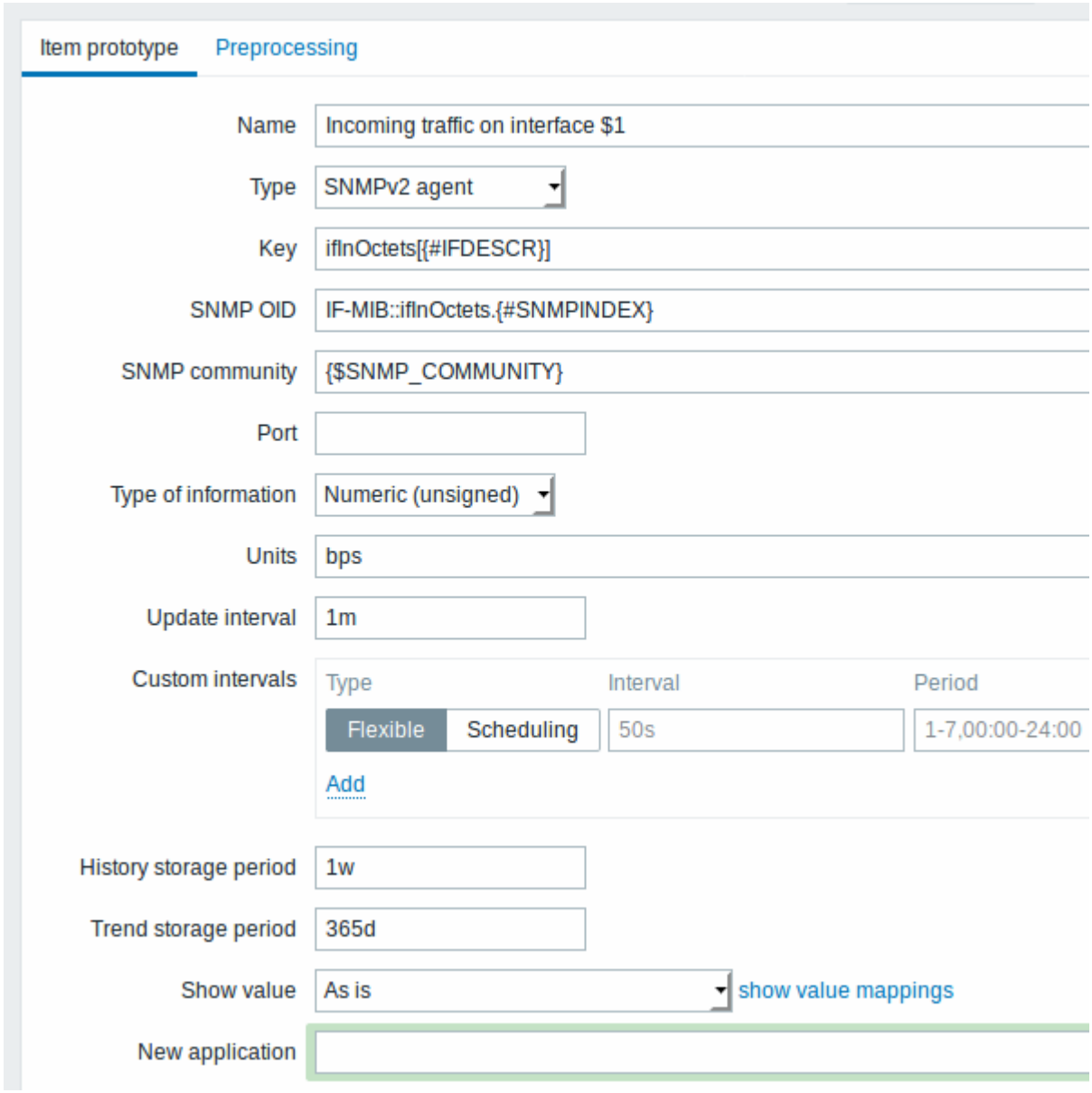
那么在这种情况下SNMP发现`discovery[{#IFDESCR}, ifDescr, {#IFALIAS}, ifAlias]` 将返回以下结构：

```
{
    "data": [
        {
            "{#SNMPINDEX}": 1,
            "{#IFDESCR}": "Interface #1",
            "{#IFALIAS}": "eth0"
        },
        {
            "{#SNMPINDEX}": 2,
            "{#IFDESCR}": "Interface #2",
            "{#IFALIAS}": "eth1"
        },
        {
            "{#SNMPINDEX}": 3,
            "{#IFALIAS}": "eth2"
        },
        {
            "{#SNMPINDEX}": 4,
            "{#IFDESCR}": "Interface #4"
        },
        {
            "{#SNMPINDEX}": 5,
            "{#IFALIAS}": "eth4"
        }
```

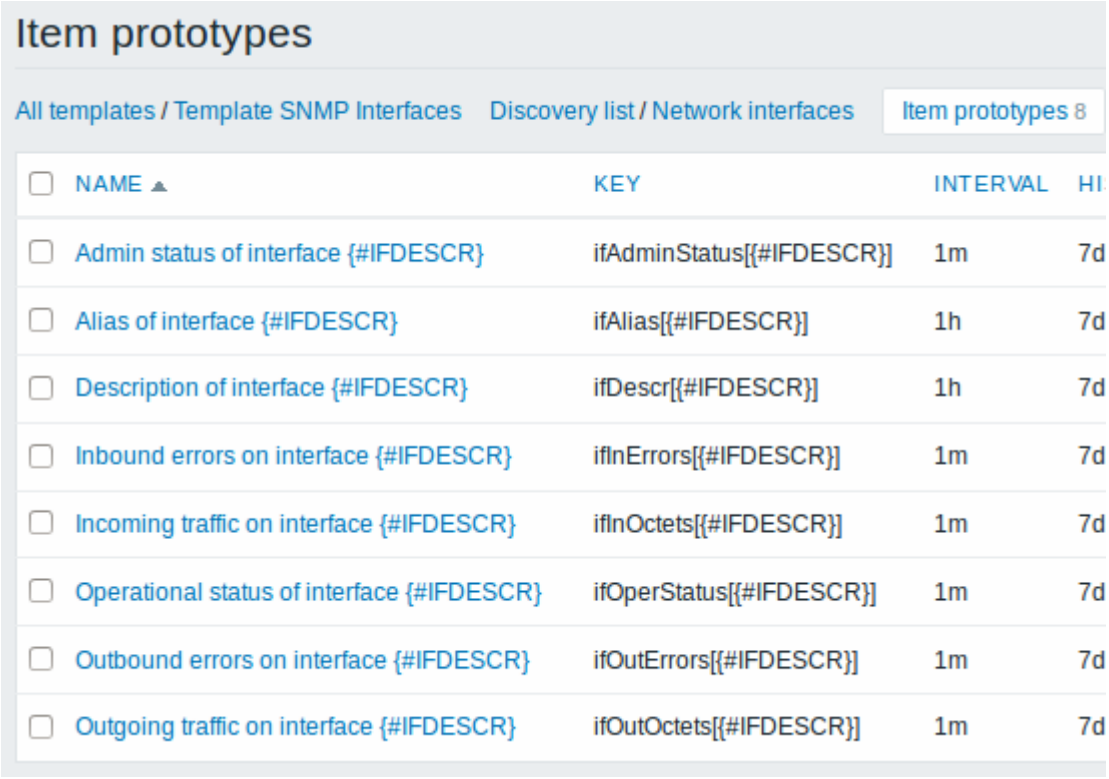
```
    ]
}
```



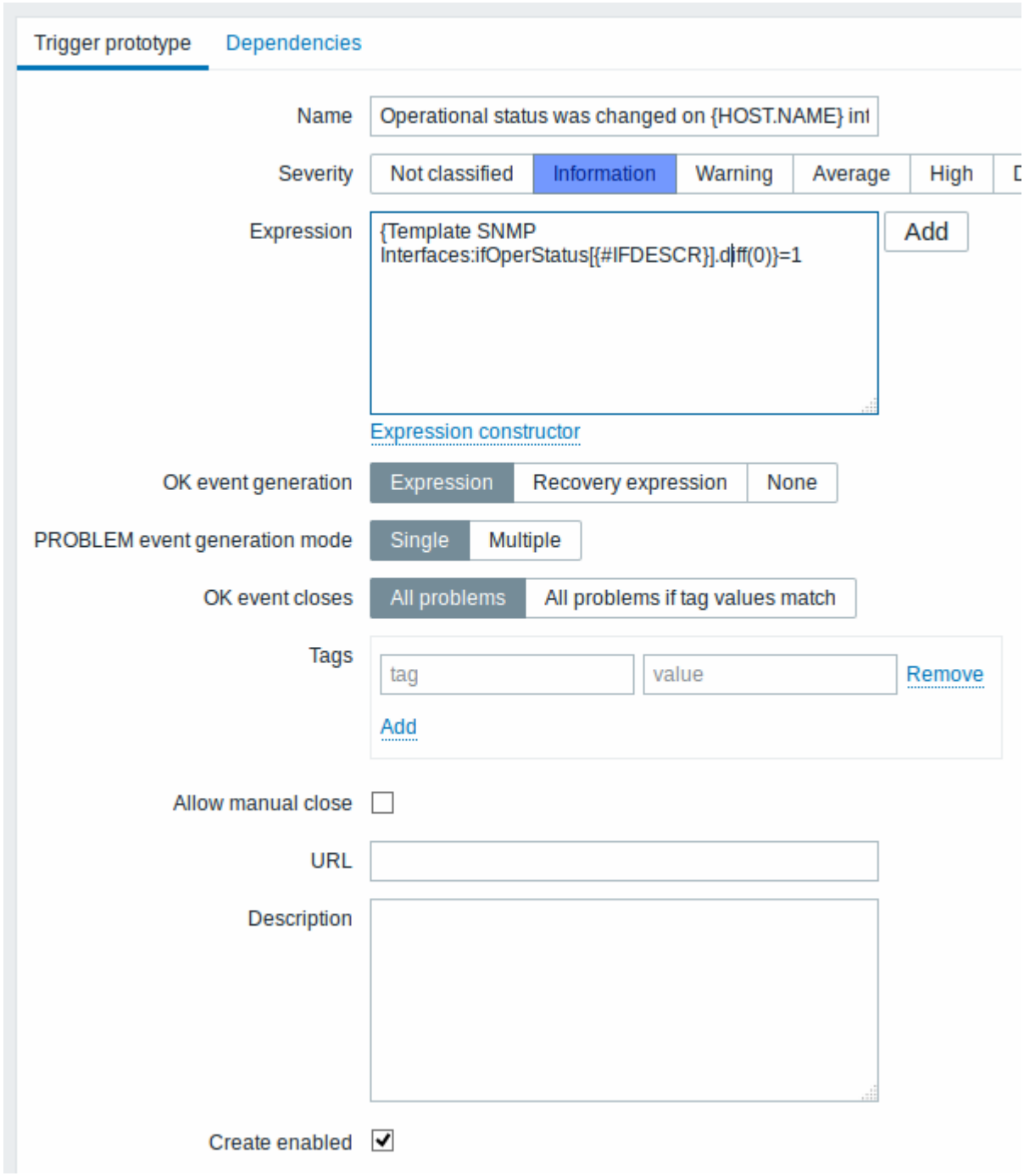
以下屏幕截图显示了我们如何在监控项原型中使用这些宏：

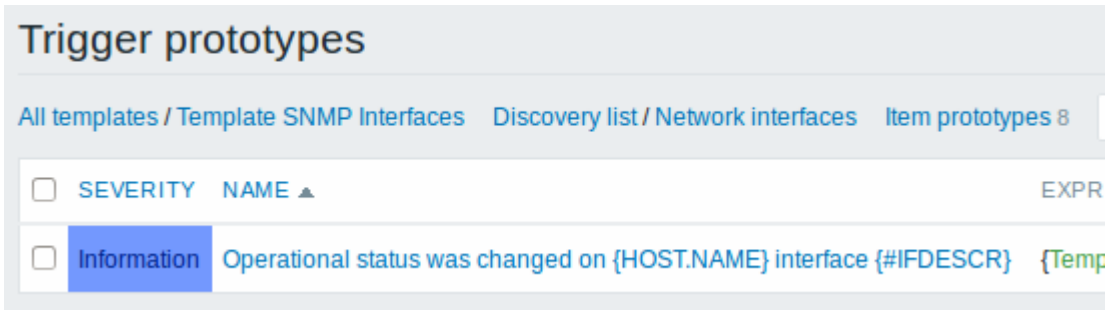


再次，根据需要创建尽可能多的监控项原型：

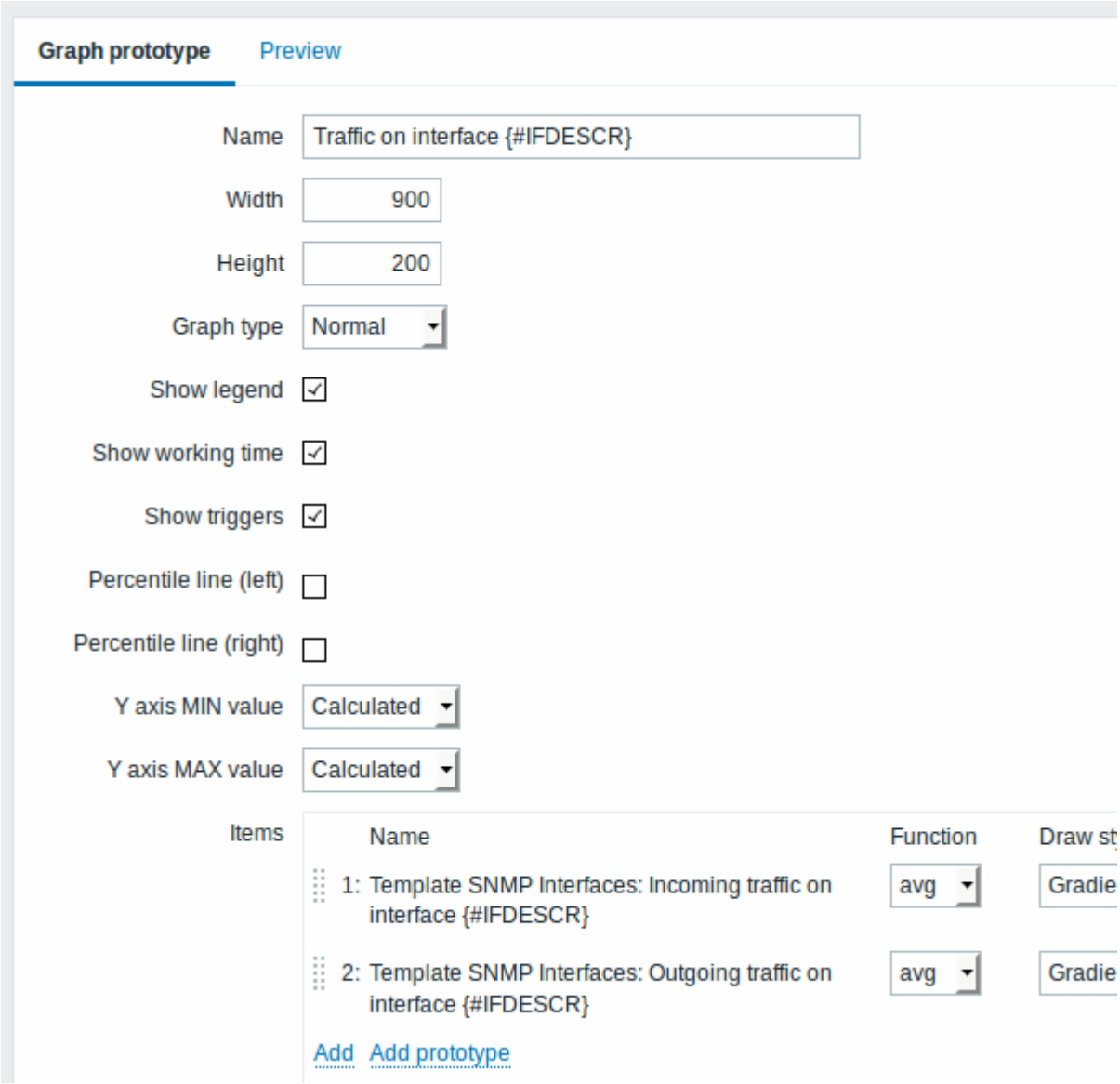

Item prototypes
All templates / Template SNMP Interfaces
Discovery list / Network interfaces
Item prototypes 8
NAME ▲
KEY
INTERVAL
HI
Admin status of interface {#IFDESCR}
ifAdminStatus[{#IFDESCR}]
1m
7d
Alias of interface {#IFDESCR}
ifAlias[{#IFDESCR}]
1h
7d
Description of interface {#IFDESCR}
ifDescr[{#IFDESCR}]
1h
7d
Inbound errors on interface {#IFDESCR}
ifInErrors[{#IFDESCR}]
1m
7d
Incoming traffic on interface {#IFDESCR}
ifInOctets[{#IFDESCR}]
1m
7d
Operational status of interface {#IFDESCR}
ifOperStatus[{#IFDESCR}]
1m
7d
Outbound errors on interface {#IFDESCR}
ifOutErrors[{#IFDESCR}]
1m
7d
Outgoing traffic on interface {#IFDESCR}
ifOutOctets[{#IFDESCR}]
1m
7d

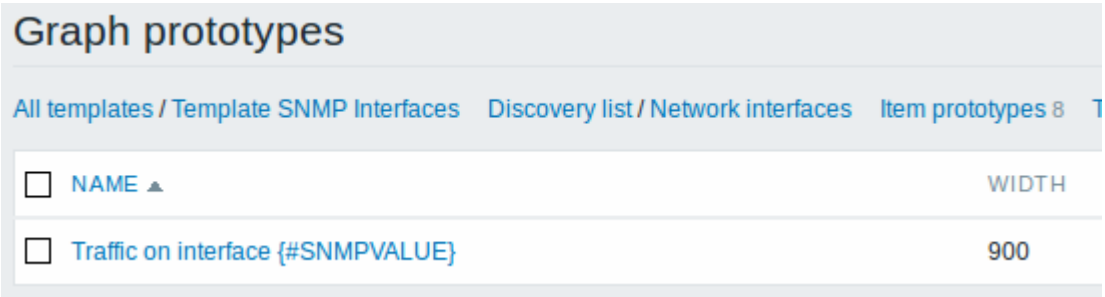
以及触发原型：


Trigger prototype
Dependencies
Name
Operational status was changed on {HOST.NAME} int
Severity
Not classified
Information
Warning
Average
High
Expression
{Template SNMP Interfaces:ifOperStatus[{#IFDESCR}].diff(0)}=1
Add
Expression constructor
OK event generation
Expression
Recovery expression
None
PROBLEM event generation mode
Single
Multiple
OK event closes
All problems
All problems if tag values match
Tags
tag
value
Remove
Add
Allow manual close
URL
Description
Create enabled

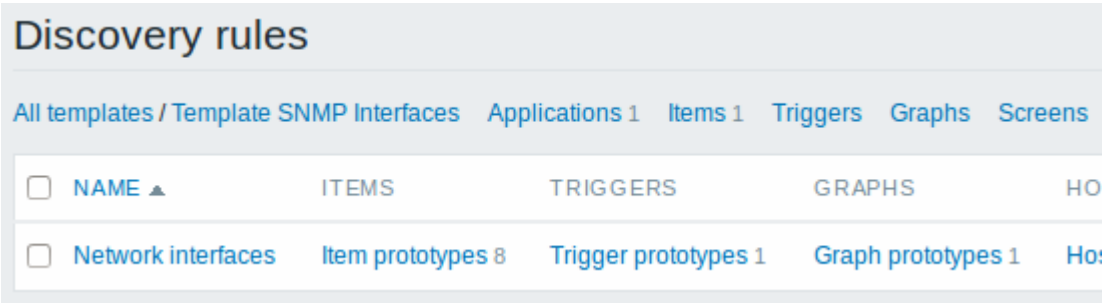

Trigger prototypes
All templates / Template SNMP Interfaces
Discovery list / Network interfaces
Item prototypes 8
SEVERITY
NAME
EXPR
Information
Operational status was changed on {HOST.NAME} interface {#IFDESCR}
{Temp

和图形原型：

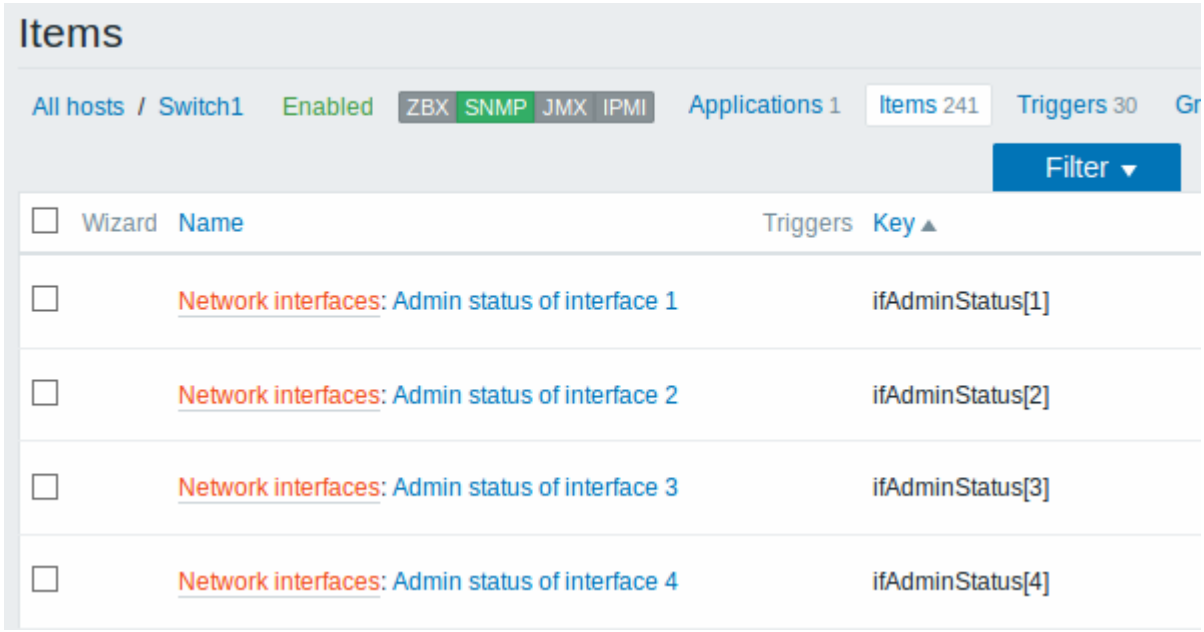


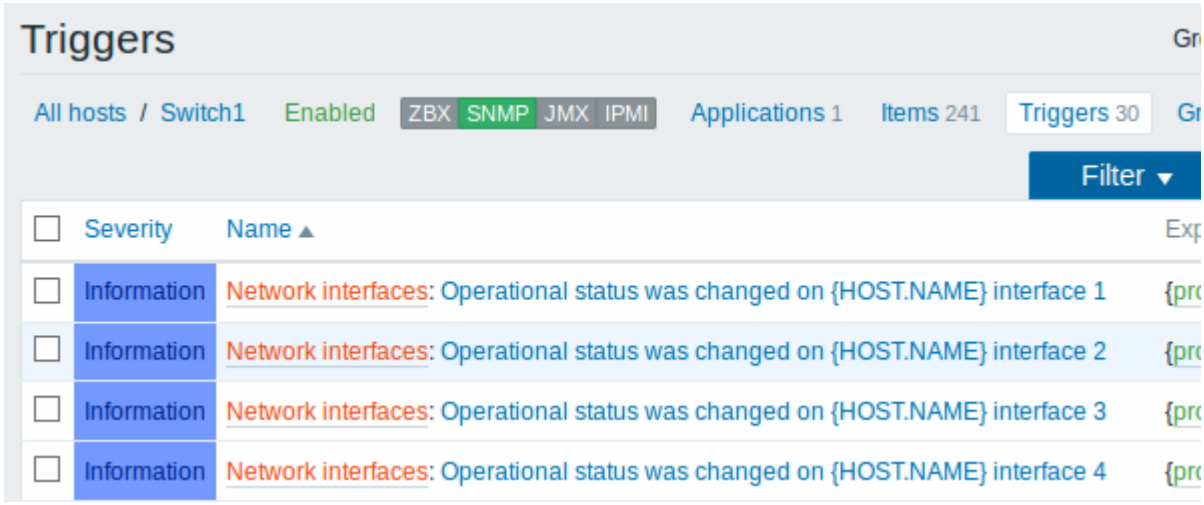


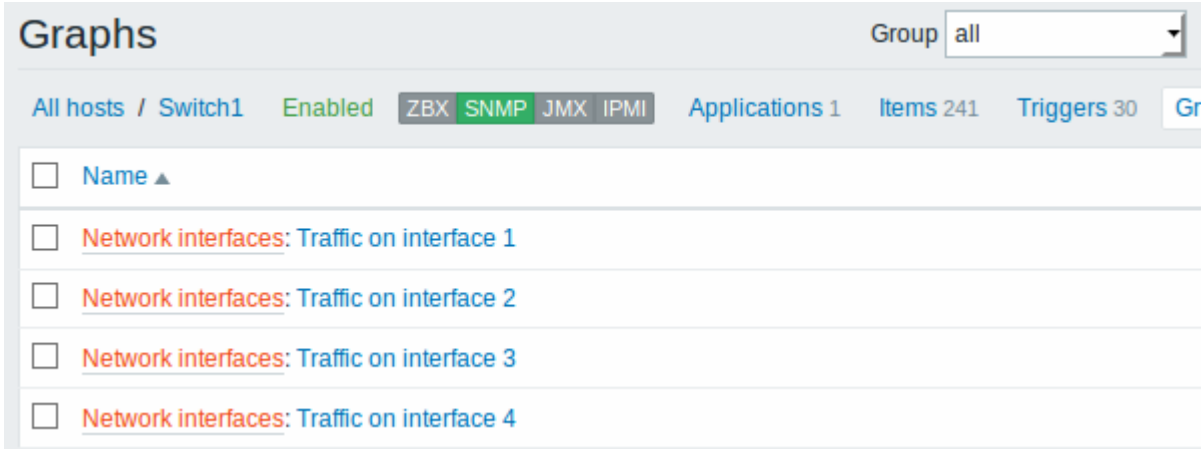
我们的发现规则摘要：



当服务器运行时，它将根据SNMP发现规则返回的值创建实际监控项，触发器和图形。在主机配置中，它们的前缀是橙色链接到它们来自的发现规则。







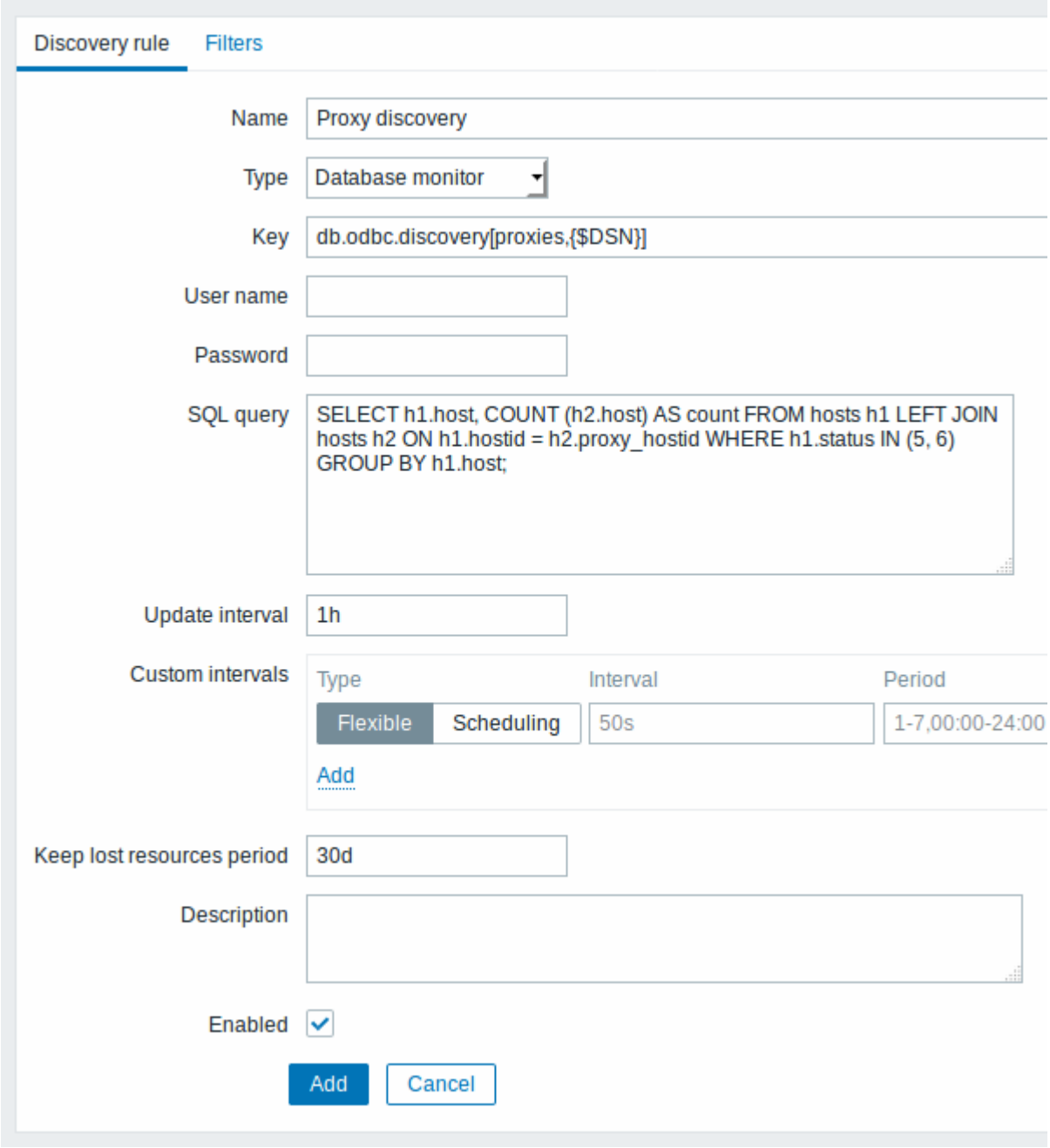
3.5 ODBC SQL查询的发现

这种类型的发现使用SQL查询完成，其结果自动转换为适合于自动发现□LLD□的JSON对象。使用“数据库监控”类型的项目执行SQL查询。因此，ODBC监控页面上的说明都适用于“数据库监控”发现规则，唯一的区别是应该使用”db.odbc.discovery[<description>,<dsn>]“键代

替”db.odbc.select[<description>,<dsn>]”□

举例来说明SQL查询如何转换为JSON□我们可以通过在Zabbix数据库上执行ODBC查询来执行Zabbix proxies 自动发现□LLD□□这对于自动创建”zabbix[proxy,<name>,lastaccess]“ 内部项目来监视哪些proxies是存活的很有用。

让我们从发现规则配置开始：



这里，对Zabbix数据库的执行查询用于选择所有Zabbix proxies以及它们正在监视的主机数量。例如，可以使用主机数量来过滤掉空 proxies□

```
mysql> SELECT h1.host, COUNT(h2.host) AS count FROM hosts h1 LEFT JOIN hosts
h2 ON h1.hostid = h2.proxy_hostid WHERE h1.status IN (5, 6) GROUP BY
h1.host;
+---------+-------+
| host    | count |
```

```
+---------+-------+
| Japan 1 |     5 |
| Japan 2 |    12 |
| Latvia  |     3 |
+---------+-------+
3 rows in set (0.01 sec)
```

通过“db.odbc.discovery []”项目的内部工作，此查询的结果将自动转换为以下JSON：

```
{
    "data": [
        {
            "{#HOST}": "Japan 1",
            "{#COUNT}": "5"
        },
        {
            "{#HOST}": "Japan 2",
            "{#COUNT}": "12"
        },
        {
            "{#HOST}": "Latvia",
            "{#COUNT}": "3"
        }
    ]
}
```

It can be seen that column names become macro names and selected rows become the values of these macros.

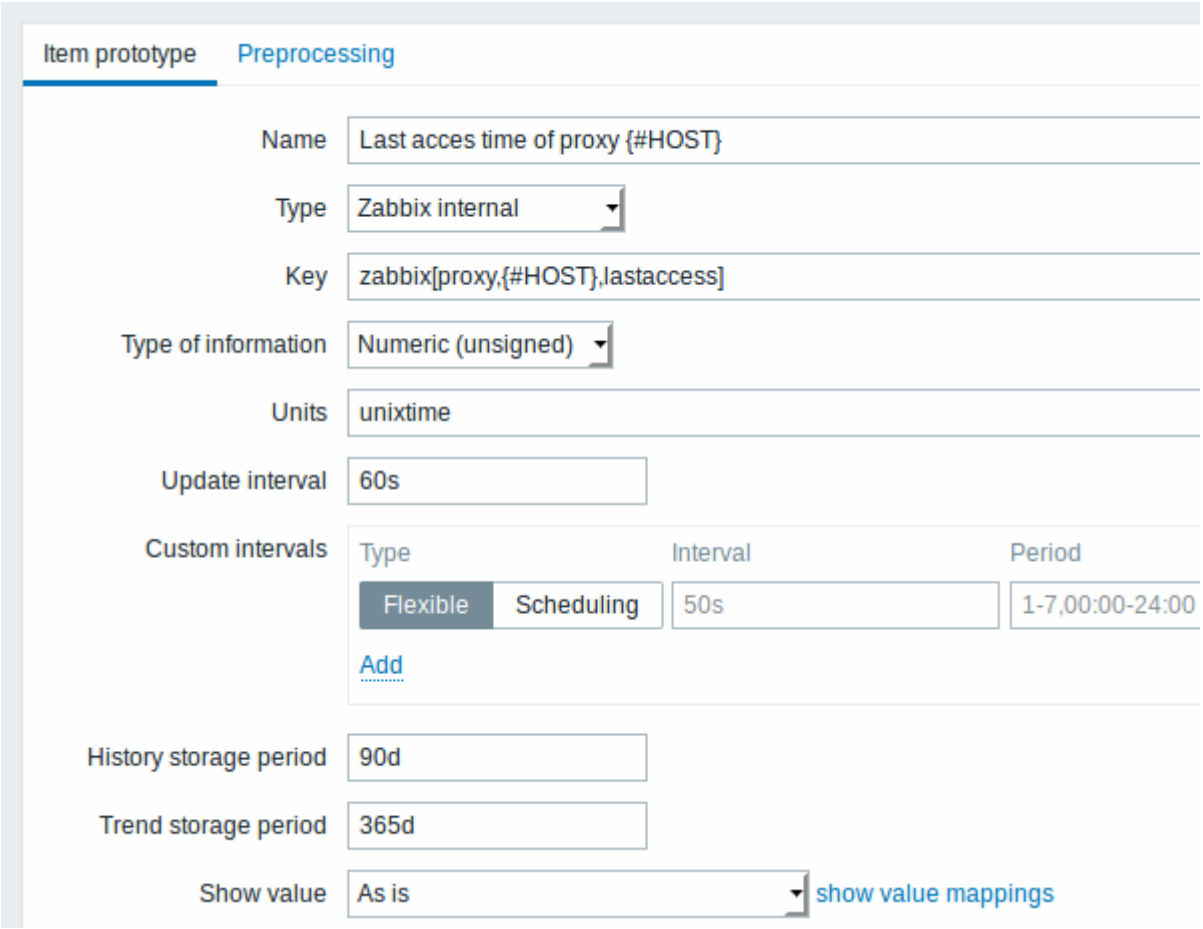
可以看出，列名称成为宏名称，选定的行将成为这些宏的值。

如果将列名称变换为宏名称不明显，建议在上述示例中使用像”COUNT(h2.host) AS count“这样的列别名。

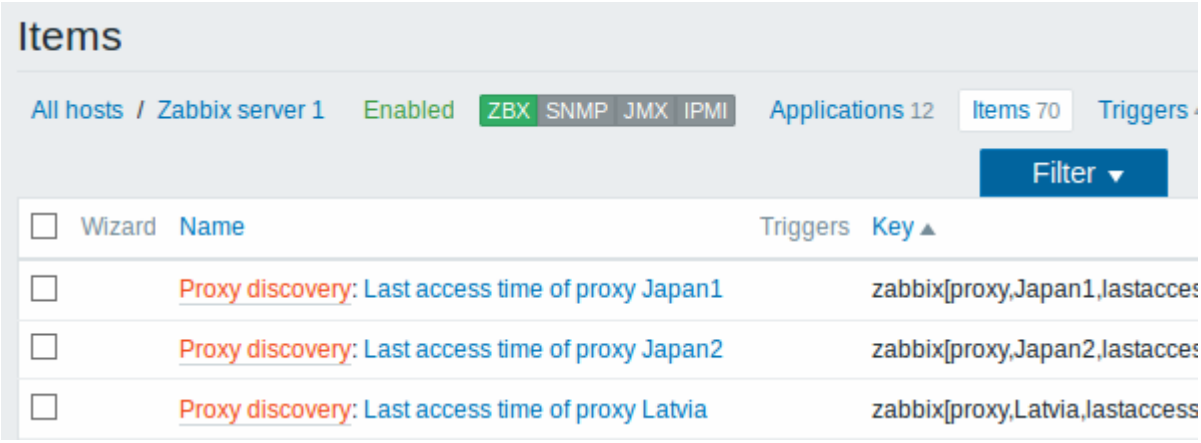
如果列名称无法转换为有效的宏名称，则不支持发现规则，错误消息将详细列出违规列号。如果需要其他帮助，获取的列名称在Zabbix服务器日志文件中的DebugLevel = 4下提供：

```
$ grep db.odbc.discovery /tmp/zabbix_server.log
 ...
 23876:20150114:153410.856 In db_odbc_discovery() query:'SELECT h1.host,
COUNT(h2.host) FROM hosts h1 LEFT JOIN hosts h2 ON h1.hostid =
h2.proxy_hostid WHERE h1.status IN (5, 6) GROUP BY h1.host;'
 23876:20150114:153410.860 db_odbc_discovery() column[1]:'host'
 23876:20150114:153410.860 db_odbc_discovery() column[2]:'COUNT(h2.host)'
 23876:20150114:153410.860 End of db_odbc_discovery():NOTSUPPORTED
 23876:20150114:153410.860 Item [Zabbix
server:db.odbc.discovery[proxies,{$DSN}]] error: Cannot convert column #2
name to macro.
```

现在我们了解SQL查询如何转换为JSON对象，我们可以在项目原型中使用{#HOST}宏：



执行发现后，将为每个proxy创建一个监控项：



3.6 Windows服务的发现

Windows服务发现的方式与文件系统的发现相同。在发现规则中使用的关键是“service.discovery”并且支持以下宏用于过滤器和监控项/触发器/图形原型：

```
{#SERVICE.NAME}
{#SERVICE.DISPLAYNAME}
{#SERVICE.DESCRIPTION}
```

```
{#SERVICE.STATE}
{#SERVICE.STATENAME}
{#SERVICE.PATH}
{#SERVICE.USER}
{#SERVICE.STARTUP}
{#SERVICE.STARTUPNAME}
```

基于Windows服务发现，你可以创建一个监控项原型，如"service.info[{#SERVICE.NAME},<param>]"□□其中*param*接受以下值：*state*, *displayname*, *path*, *user*, *startup* 或 *description*。例如，要获取服务的显示名称，您应该使用"service.info[{#SERVICE.NAME},displayname]"项目。如果没有指定*param*值（"service.info[{#SERVICE.NAME}]"□□则使用默认参*param*态。

{□SERVICE.STATE}和{□SERVICE.STATENAME}宏返回相同的内容，但{□SERVICE.STATE}返回数值（0-7），而{□SERVICE.STATENAME}返回文字(*running*, *paused*, *start pending*, *pause pending*, *continue pending*, *stop pending*, *stopped* or *unknown*)□{□SERVICE.STARTUP}和{□SERVICE.STARTUPNAME}也是如此，其中一个返回数字值（0-4），而另一个文本(*automatic*, *automatic delayed*, *manual*, *disabled*, *unknown*)□

3.7 为同一项目设置多个LLD规则

从Zabbix agent版本3.2，可以使用zabbix_agentd.conf文件中的"Alias"参数来更改自动发现项目键值，以便为同一项目配置多个LLD规则。

3.8 创建自定义LLD规则

也可以创建完全自定义的LLD规则，发现任何类型的实体 - 例如数据库服务器上的数据库。

为此，应该创建一个返回JSON的自定义项目，指定找到的对象以及可选的一些属性。每个实体的宏数量不受限制 - 而内置的发现规则返回一个或两个宏（例如，两个用于文件系统发现）。

下面举例说明JSON格式。假设我们运行一个旧的Zabbix 1.8 agent□不支持"vfs.fs.discovery"□□但是我们仍然需要发现文件系统。这是一个用于Linux的简单Perl脚本，用于发现挂载的文件系统并输出JSON□其中包含文件系统名称和类型。使用它的一种方式是具有键值"vfs.fs.discovery_perl"的参数：

```
#!/usr/bin/perl

$first = 1;

print "{\n";
print "\t\"data\":[\n\n";

for (`cat /proc/mounts`)
{
    ($fsname, $fstype) = m/\S+ (\S+) (\S+)/;

    print "\t,\n" if not $first;
```

```
    $first = 0;

    print "\t{\n";
    print "\t\t\"{#FSNAME}\":\"$fsname\",\n";
    print "\t\t\"{#FSTYPE}\":\"$fstype\"\n";
    print "\t}\n";
}

print "\n\t]\n";
print "}\n";
```

对于LLD宏名允许的符号为 **0-9** , **A-Z** , **_** , **.**

名称中不支持小写字母。

其输出的示例（为了清楚起见重新格式化）如下所示。用于自定义发现检查的JSON必须遵循相同的格式。

```
{
  "data":[
  { "{#FSNAME}":"/",                                  "{#FSTYPE}":"rootfs"   },
  { "{#FSNAME}":"/sys",                               "{#FSTYPE}":"sysfs"    },
  { "{#FSNAME}":"/proc",                              "{#FSTYPE}":"proc"     },
  { "{#FSNAME}":"/dev",                               "{#FSTYPE}":"devtmpfs" },
  { "{#FSNAME}":"/dev/pts",                           "{#FSTYPE}":"devpts"   },
  { "{#FSNAME}":"/lib/init/rw",                       "{#FSTYPE}":"tmpfs"    },
  { "{#FSNAME}":"/dev/shm",                           "{#FSTYPE}":"tmpfs"    },
  { "{#FSNAME}":"/home",                              "{#FSTYPE}":"ext3"     },
  { "{#FSNAME}":"/tmp",                               "{#FSTYPE}":"ext3"     },
  { "{#FSNAME}":"/usr",                               "{#FSTYPE}":"ext3"     },
  { "{#FSNAME}":"/var",                               "{#FSTYPE}":"ext3"     },
  { "{#FSNAME}":"/sys/fs/fuse/connections",           "{#FSTYPE}":"fusectl"  }
  ]
}
```

然后，在发现规则的“过滤器”字段中，我们可以将“{#FSTYPE}”指定为宏，将“rootfs|ext3”指定为正则表达式。

你不一定使用具有自定义LLD规则的宏名称FSNAME/FSTYPE□你可以随意使用任何名称。

3.9 在用户宏上下文中使用LLD宏

具有上下文的用户宏可用于在触发器表达式中实现更灵活的阈值。可以在用户宏级别上定义不同的阈值，然后根据发现的上下文使用触发器常量。当宏中使用的自动发现□LLD□宏被解析为真实值时，会出现在发现的上下文。

为了说明我们可以从例子中使用上述数据和假设下面的文件系统将被发现：/, /home, /tmp, /usr, /var□

我们可以为主机定义一个可用磁盘空间触发器原型，其中阈值由具有上下文的用户宏表示：

```
{host:vfs.fs.size[{#FSNAME},pfree].last()}<{$LOW_SPACE_LIMIT:"{#FSNAME}"}
```

然后添加用户宏：

- {$LOW_SPACE_LIMIT} **10**
- {$LOW_SPACE_LIMIT:/home} **20**
- {$LOW_SPACE_LIMIT:/tmp} **50**

现在，一旦文件系统被发现，事件将被告知是否产生/□/usr以及/var文件系统具有小于**10**% 的可用磁盘空间，该/home文件系统–小于 **20**% 的可用磁盘空间或□/tmp的文件系统–小于**50**%的可用磁盘空间 。

触发功能参数中的用户宏上下文内不支持LLD宏。